REST and .NET 3.5

Part 1 – why REST based services?

Despite what the toolkits would have us believe, SOAP is not the only way to build services based software. SOAP has some good things going for it: standardization, service metadata, vendor acceptance and rich tooling. However, it's not without its issues, particularly when building very large systems that require a broad reach to consumers. It is no accident that Yahoo, Google and Amazon have chosen not to use SOAP to expose their programmable interfaces. In the first part of this two-part article I'll detail some of the issues that SOAP has in building systems. I'll then move on to explain how REpresentational State Transfer, or REST, is an architecture that doesn't suffer from the problems that SOAP does (although has issues of its own). Then, next month, in part 2, I'll show how you can create and consume REST based services using WCF, ASP.NET and LINQ to XML.

What's wrong with SOAP?

Before looking at REST and what it offers. We need to examine challenges in the SOAP world that are hard or impossible to solve. So let's get a clear picture of how SOAP works. SOAP relies on XML messages being sent to a defined endpoint. The service at the endpoint then works out what to do with the message based on the action header that is part of the WS-Addressing standard. When we are talking about services that are part of heterogeneous environments these messages will be sent using HTTP. However, because of the nature of the payload (textual XML) they are always sent using the HTTP verb POST. They could, in theory, use other verbs but why bother, the intent of the message is in the action header not in the way it uses HTTP. HTTP is purely there as a transport, the message is independent of it. Web Service Description Language (WSDL) is an XML dialect used to describe the endpoint including all the operations available there and what messages they require and what messages they may send back. It all seems pretty reasonable, so what are the problems.

HTTP is only a transport

Having the message independent of the transport would appear to be a good thing, so why is it a problem? The web has been proven, empirically, to be massively scalable. The question is, "how"? How do web requests generally work? When you issue a web request there are two important things that occur: the link is to a specific URL which may not be the same machine that delivered the page you are looking at; you are very often using HTTP GET. The first factor means we can easily spread the workload; one machine (or farm) is not acting as a performance pinch point. The second factor means the caching infrastructure delivered as part of HTTP 1.1 can be leveraged to make sure that a proxy server, or even the client machine itself, can service the request without the target web server ever being touched. Compare this to SOAP: all requests hit the same endpoint before they can be filtered by action header and POST cannot be cached. SOAP cannot use the very infrastructure that makes the web so scalable.

The application protocol is not in WSDL

The application protocol is the sequence of valid operations that a service can perform. Take a shopping cart; we might have a number of operations that could be performed:

a) Log in

b) Search for product
c) Put item in cart
d) Remove item from cart
e) Go to checkout
f) Log out

As a human you can see the various workflows that could occur through this series of operations. You can also spot sequences that are wholly inappropriate: go to checkout before putting any items in the cart; Log out before log in. The problem is machines have no way to determine this sequencing unless it gets documented somehow in a machine readable form and this information is not encoded in WSDL. All the service can do if it receives a message for an inappropriate operation is the service orientated equivalent of `throw new InvalidOperationException();`.

Now where was I …?

If a series of operations takes place over an extended period, the client may want to stop and continue the exchanges at a later time or date. To do this they have to, essentially, remember everything they have done so far. Nothing intrinsic about the endpoint will tell them the state of the exchange. Picking up the exchange means understanding the next valid options without the endpoint itself being able to help in any way. Compare this to how the web works – where you got to while using a website is normally encoded in the current URL. In many situations you can pick up the processing simply by re-using the URL. Now obviously business context may preclude this, for example, an insurance quote may only be valid for 48 hours. If you try to purchase the insurance after that by reusing the URL the insurance website would force you to start the process over again. This, however, is a business restriction, not a technical one.

Not everything is XML

In SOAP, if I want to retrieve an image, I have to pack that binary payload into an XML message by either using base64 encoding or by using the MTOM standard. Then when I receive the message I have to decode it to be able to display it. This seems highly over-complicated if I know I am getting image that I could simply display.

What is REST?

REST is a model for creating services that embraces the model that the web uses. It is a model that is based on *resources* that are identified by a unique URI. Then, what you are intending to do with the resource is specified by a transport protocol verb. The model his formally not bound to HTTP but for all intents and purposes it currently is. The verbs generally used are: GET, PUT, DELETE and POST.

- GET – Retrieve the resource
- PUT – Update the resource
- DELETE – Delete the resource
- POST – Create the resource

There is some debate about whether you should also use PUT for creation and POST also gives a general catch-all for operations that don't really conform to CRUD (Create, Read, Update, Delete) operations. Let's look at an example: we want to retrieve the list of products from the acme corp. Order system. SOAP sends a message, using HTTP POST, to the order system endpoint (say, http://www.acme.com/orderservice) with an action header of `GetProducts`. With REST we issue

an HTTP GET to the URI http://www.acme.com/orderservice/products. It is the verb that determines what we are trying to do and the URI that determines what we are trying to do it to. But more than this, REST based systems also return the URIs for the next valid set of operations depending on the context of where we are in the application protocol. So let's look at why this model doesn't suffer from the issues with SOAP detailed above.

HTTP scales

The fact that each resource has a different URL does not imply that they should all be on the same host. There is no need for a single "endpoint". This allows us, for example, to horizontally partition our data on to separate server farms where customers A-M hit one set of machines and customers N-Z hit another. In addition all read operations (namely those that do not alter state on the server) use HTTP GET. GET operations can be cached using the HTTP 1.1 caching directives which allows the service to offload traffic on to the general web infrastructure.

The application protocol is built into the message exchange

A fundamental part of REST is that the next set of valid operations is encoded in the response from the last. This means that as long as you only use actions from the last operation you will always perform a valid operation according to the application protocol.

The URI gives you context

The URI does not have to be a static thing but is inherently a data driven entity. Let's use an expense claim service: we can create a new expense claim passing a number of line items using

POST http://acme.com/expenseclaim

This returns the next valid actions one of which could be submitting the expense claim to accounts. This action could be

POST http://acme.com/expenseclaim/submission/456

The 456 in the URI is the identifier for the created expense claim. The crucial thing here is that the user can save the URI and then go home for the night. In the morning they simply submit to the URI and the context of where they were in the expense claim submission is maintained.

The payload is whatever makes sense

REST does services very often use XML as a payload but it does not bind you to it if some other format makes more sense. For example, if an AJAX application wants to use your service it makes more sense to return the data as JSON than XML; if the resource is an image then use an image.

Is REST Perfect?

So if REST is so great should I always build my systems as REST based ones? Well some people would argue that but there are issues in REST.

No metadata

As may be apparent from the REST benefits above, REST is a pragmatic way of building services and this pragmatism currently means there is no metadata. In other words the consumer has to implicitly know what a service requires and what it may return. Hopefully testing the application should drive

out whether assumptions about message formats are correct and the pragmatism does not absolve the service provider from taking a responsible attitude to service evolution.

No standard for actions

The next valid actions are a crucial concept in REST and yet there is no standard way of encoding this information and no agreed place of putting this information. There has been some discussion about using ATOMPub as a mechanism but this is far from an agreed standard.

Tooling is rudimentary

On the .NET platform there is little tool support for consuming REST services. With the lack of metadata you are left with having to consume the saw format of the message whether this XML, JSON, an image or something else. There is growing support on the service side with .NET 3.5 WCF extensions and the upcoming ASP.NET routing infrastructure shipping in .NET 3.5 SP1.

Conclusion

REST is a very powerful way to build services. The model lends itself to high scalability as it uses the same model as the web. It is particularly applicable to services that are exposed on to the public internet as generally all a consumer needs to be able to do is manipulate XML and talk HTTP - which every platform supports.

Part 2 – Implementing REST Services and Clients using .NET 3.5

In part one of this article, published last month, I discussed what REST was and the advantages that it had over SOAP for some kinds of services. In this month I look at functionality released in .NET 3.5 and some to be released in the upcoming .NET 3.5 SP1. As such I'll be using LINQ to XML, the WCF REST bindings and the ASP.NET Routing infrastructure to create REST based services and clients.

The Scenario

It's probably simplest to show a REST implementation using something concrete. So to that end I will be using a service modelling a book store (not too original I know). This service allows you to retrieve the books based on various criteria and to add books to the store. A book is modelled as the following XML entity:

```
<book xmlns="urn:dm-books">
  <isbn>111-222-333</isbn>
  <title>The DaVinci Code</title>
  <author>Dan Brown</author>
  <price>10.50</price>
</book>
```

A collection of books will have multiple book records in a `<books/>` element in the same namespace.

The supported URI/operation combinations are

Retrieve all books

`GET http://bookstore.develop.com/books`

Retrieve a book via its ISBN

`GET http://bookstore.develop.com/books/isbn/<isbn>`

Find all books that fall within a price range

`GET http://bookstore.develop.com/search/price?min=<min>&max=<max>`

Add a book to the store

`POST http://bookstore.develop.com/books`

Where the POST data is a book entity

Writing a REST Client

Consuming a RESTful service is nothing more than being able to manipulate XML and speak HTTP. These capabilities have been there in .NET since 1.0. However, with the advent of LINQ to XML which have a very concise, natural model for manipulating XML.

Reading Data

Let's look first at the read side - we'll use the API to get all the books. Now, although the data will be returned as XML, our client really will want to work generally in terms of .NET objects. This is a simple task as one of the most powerful things that LINQ does in general is allow us to bridge type systems. Let's look at the code used to consume the service to get all the books.

```
XNamespace ns = "urn:dm-books";

XElement data = XElement.Load("http//bookstore.develop.com/books");
var results = from book in data.Elements(ns + "book")
              select new Tome
              {
                  Name = (string)book.Element(ns + "title"),
                  Writer = (string)book.Element(ns + "author")
              };

foreach (var item in results)
{
    Console.WriteLine(item);
}
```

Were a `Tome` looks like this

```
class Tome
{
    public string Name { get; set; }
    public string Writer { get; set; }
    public override string ToString()
    {
        return string.Format("{0} : {1}", Writer, Name);
    }
}
```

So `XElement.Load` allows us to point at an HTTP location and it performs a GET to retrieve the XML directly from the REST service. Then we can use LINQ to XML to transform the XML into `Tome` objects that we can use in the application.

Writing Data

Ad you can see, reading data is pretty straightforward; but what about writing data? Here we have to do a little more work. We need to take more direct control over the HTTP request and so we use an `HttpWebRequest` object. We need to set the HTTP verb to be POST and specify the content as textual XML.

```
HttpWebRequest req = (HttpWebRequest)WebRequest.Create("http//bookstore.develop.com/books");
req.Method = "POST";
req.ContentType = "text/xml";
StreamWriter sw = new StreamWriter(req.GetRequestStream());

XNamespace ns = "urn:dm-books";

XElement newbook = new XElement(ns + "book",
    new XElement(ns + "isbn", "555-666-777"),
    new XElement(ns + "title", "Saturday"),
    new XElement(ns + "author", "Ian McKeown"),
    new XElement(ns + "price", "17")
    );

newbook.Save(sw);
sw.Dispose();

WebResponse resp = req.GetResponse();
```

We then create the XML using LINQ to XML and save the XML into the request stream. Finally we send the request to the service using `WebRequest.GetResponse`. So the save side is also relatively straightforward.

Client Wrap Up

Over the years we have become used to always having our generated proxy to hand that allows us to pretend we're making a call on a remote object. However, as you can see, working at the XML level is also pretty simple – especially now we have LINQ to XML in our toolkit.

Implementing the Service

We've now seen how to consume a REST service in .NET 3.5, but how about implementing the service? We're going to look at two techniques: WCF and ASP.NET. Both implementations, however, will use broadly the same data store, implemented as a static class:

```
public static class BookStore
{
    static List<Book> books = new List<Book>()
    {
        Book Details
    };

    public static Book[] GetBooks()
    {
        return books.ToArray();
    }

    internal static void AddBook(Book b)
    {
        books.Add(b);
    }
}

public class Book
{
    public string ISBN { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
}
```

Implementing the Service using WCF 3.5

In some ways it seems strange. Microsoft spent a lot of man-hours building WCF, a toolkit that attempts to abstract the service implementation from the network protocol being used to expose that service. In .NET 3.5 they added a layer on top of WCF that was HTTP specific to allow the creation of RESTful services. However, to understand how it works lets remind ourselves how WCF normally works in terms of defining services.

ServiceContracts

In WCF, you define the functionality available at a service endpoint using a `ServiceContract`.

```
[ServiceContract]
interface IBookStore
{
    [OperationContract]
    Books GetBooks();

    [OperationContract]
    Books GetBooksByISBN(string isbn);

    [OperationContract]
    Books SearchBooksByPrice(int min, int max);

    [OperationContract]
    void AddBook(Book book);
}
```

Remember that in SOAP the SOAP Action header defines the purpose of a message. WCF uses the Action to map the message to an operation annotated with the `OperationContract` attribute. This mapping can be influenced by properties of the `OperationContract`. However, with REST we have no SOAP Action header - routing must be based on URI and Verb. .NET 3.5 introduces two new attributes in the `System.ServiceModel.Web` namespace and assembly: `WebGet` and `WebInvoke`. We use these new attributes to further annotate the `ServiceContract`.

```
[ServiceContract]
interface IBookStore
{
    [OperationContract]
    [WebGet(UriTemplate="books")]
    Books GetBooks();

    [OperationContract]
    [WebGet(UriTemplate = "books/isbn/{isbn}")]
    Books GetBooksByISBN(string isbn);
    [OperationContract]

    [WebGet(UriTemplate = "search/price?min={min}&max={max}")]
    Books SearchBooksByPrice(int min, int max);

    [OperationContract]
    [WebInvoke(Method="POST", UriTemplate="books")]
    void AddBook(Book book);
}
```

`WebGet` allows you to specify the URI that maps on to this method. Notice it allows the inclusion of placeholders that map to operation parameters. This URI templating is part of the `UriTemplate` class which is decoupled from the WCF usage and can be found in the `System` namespace in the `System.Core` assembly.

`WebInvoke` also allows you to specify the URI in a template form, but also, this time, the Verb. The data for the operation, if not mapped from the URI will be taken from the body of the HTTP request.

Serialization

This brings us to an interesting question: how do I get from the XML of the message to my objects? WCF uses serialization to bridge the object and XML worlds. By default it uses a serializer called the `DataContractSerializer`. To get the XML we require it forces us to change the `BookStore` a little to provide serialization annotations.

```
[DataContract(Name="book", Namespace="urn:dm-books")]
public class Book
{
    [DataMember(Name="isbn", Order=1)]
    public string ISBN { get; set; }
    [DataMember(Name="title", Order=2)]
    public string Title { get; set; }
    [DataMember(Name="author", Order=3)]
    public string Author { get; set; }
    [DataMember(Name="price", Order=4)]
    public decimal Price { get; set; }
}

[CollectionDataContract(Name = "books", Namespace = "urn:dm-books")]
public class Books : List<Book>
{
}
```

The `BookStore GetBooks` operation also needs to change to return a `Books` object (this allows us to control the name of the element for the collection of books which would default to `ArrayOfBook`).

The `DataContractSerializer` has some limitations in the XML it supports: it only allows element normal form (no attributes); every element must be in the same namespace. If the XML you need to create falls outside of this you can fallback to the `XmlSerializer` by adding an `XmlSerializerFormat` attribute to the interface defining the service contract.

Rewiring the Plumbing

So far all we've done is make some annotations on some .NET types. Someone in the executing code has to care about their existence - the component that does this in WCF 3.5 is the endpoint behavior `WebHttpBehavior`. You must make sure to add this to all REST based endpoints. However, you can automate this by using the new `WebServiceHost` instead of the standard `ServiceHost` when hosting the service.

```
static void Main(string[] args)
{
    WebServiceHost host = new WebServiceHost(typeof(BookStoreService));

    host.Open();

    Console.WriteLine("Service Ready ...");
    Console.ReadLine();
}
```

Finally we need to make sure that no SOAP envelope bits go across the wire. To get the correct POX (Plan Old XML) encoder in place we use the new `WebHttpBinding` for the endpoint.

```
<endpoint address="http://bookstore.develop.com"
          binding="webHttpBinding"
          contract="WCFService.IBookStore" />
```

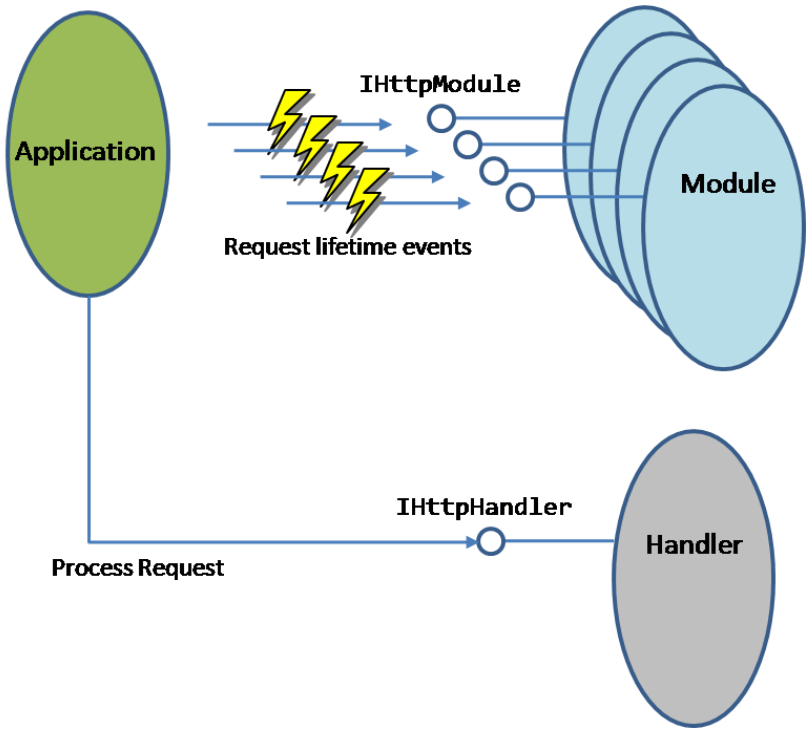
Implementing the Service using ASP.NET

We've seen how you can build a REST implementation on top of WCF but seeing as we're creating an infrastructure that is fundamentally HTTP based what about using the main HTTP processing engine in .NET – ASP.NET? What we will do now is examine how ASP.NET works and what pieces are missing from a raw ASP.NET implementation to make a REST solution simple to implement. Then we will see how the new routing infrastructure in .NET 3.5 SP1 really provides the missing pieces.

How the ASP.NET Plumbing Works

The component that actually processes an ASP.NET based request is called a handler. This is simply a component that implements the `IHttpHandler` interface. However, the actually job of the handler may be very different depending on the request to in general different handlers process different types of requests. So the question is how is the appropriate handler selected for the request? Normally this decision is based on the file extension of the requested resource: .ASPX gets a handler based on the page being requested whereas .ASMX gets another generic one that understands the SOAP plumbing. This is achieved by looking up the handler for an extension in `web.config`.

However, this is not the only processing that takes place during the request: how does security, output caching and session state get wired in? This is the role of a component called a module. A module is simply an object that implements the interface `IHttpModule`. Modules provide interception based processing at various points in the requests lifecycle. This whole infrastructure is put headed up by an application object that represents web application wide functionality. The model looks like this:



How, then, does the standard model for ASP.NET processing fit with creating REST based services? There is a fundamental problem that handler selection is based on extension rather than URI generally so there is no way in normal ASP.NET to associate a handler with a URI – especially if parts of that URI are data driven rather than static.

The ASP.NET Routing Infrastructure

.NET 3.5 SP1 sees the inclusion of a routing infrastructure for ASP.NET based requests that allows you to, in effect, associate a handler with an arbitrary URI. This was originally part of the ASP.NET MVC framework but was seen valuable enough to decouple and make it a standalone part of the ASP.NET infrastructure. The routing functionality is contained in the `System.Web.Routing` assembly.

The pivotal class in `System.Web.Routing` is the `Route` class. This allows you to associate a URI with an implementation of `IRouteHandler`. Here is what `IRouteHandler` looks like:

```
public interface IRouteHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

As you can see, this implementation simply takes the request in the form of a `RequestContext` and returns the appropriate handler. The `RequestContext` contains the `HttpContext` modelling the request and the route that led to this route handler being invoked.

```
public class RequestContext
{
    public RequestContext(System.Web.HttpContextBase httpContext, RouteData routeData);

    public System.Web.HttpContextBase HttpContext { get; }
    public RouteData RouteData { get; }
}
```

Here is an example route handler

```
public class BooksRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (requestContext.RouteData.Values.ContainsKey("isbn"))
        {
            string isbn = (string)requestContext.RouteData.Values["isbn"];
            return new ISBNHandler(isbn);
        }
        else
        {
            return new BooksHandler();
        }
    }
}
```

Here `ISBNHandler` and `BooksHandler` are two implementations of `IHttpHandler`. This sample also shows another feature of the route data – URIs can be tokenized. Here is where the route is created:

```
Route r = new Route("books", new BooksRouteHandler());
RouteTable.Routes.Add(r);
r = new Route("books/isbn/{isbn}", new BooksRouteHandler());
RouteTable.Routes.Add(r);
```

In the creation of the second route we use `{isbn}` as a token that can be accessed via the `RouteData` on the `RequestContext`.

So we have two pieces missing of the plumbing. Firstly where do you put the code that creates the entries in the `RouteTable`? Well this has to happen before the requests take place and so this is put inside the application object's `Application_Start` event handler created using a `global.asax`. The second issue is who looks at this data and routes the request by calling the appropriate route handler to get the correct handler? This is done in a module that you have to enable in `web.config`. Here is the config file entry inside the `system.web` config section.

```
<httpModules>
  <add name="Routing" type="System.Web.Routing.UrlRoutingModule, System.Web.Routing,
                                                                 Version=3.5.0.0,
                                                                 Culture=neutral,
                                                                 PublicKeyToken=31BF3856AD364E35"/>
</httpModules>
```

Implementing the Handler

The final missing piece here is how we implement the handler as we're processing the raw request. Again it is LINQ to XML to the rescue. We can access the request and consume the request stream as XML and then generate the response stream as XML. Let's look at the BooksHandler – this returns all the books on a GET and adds a new book on a POST. For `IHttpHandler` the interesting stuff takes place in `ProcessRequest`.

```
public void ProcessRequest(HttpContext context)
{
    if (context.Request.HttpMethod == "GET")
    {
        context.Response.ContentType = "text/xml";
        XNamespace ns = "urn:dm-books";
        XElement resp = new XElement(ns + "books",
                from b in BookStore.GetBooks()
                select new XElement(ns + "book",
                        new XElement(ns + "isbn", b.ISBN),
                        new XElement(ns + "title", b.Title),
                        new XElement(ns + "author", b.Author),
                        new XElement(ns + "price", b.Price)
                    )
            );

        resp.Save(context.Response.Output);
    }
    else
    {
        XElement data = XElement.Load(new StreamReader(context.Request.InputStream));
        XNamespace ns = "urn:dm-books";
        Book b = new Book
        {
            ISBN = (string)data.Element(ns + "isbn"),
            Title = (string)data.Element(ns + "title"),
            Author = (string)data.Element(ns + "author"),
            Price = (decimal)data.Element(ns + "price")
        };
        BookStore.AddBook(b);
    }
}
```

ASP.NET on its own requires a lot of extra plumbing to base the handler on the URI. However, with the extra functionality supplied by the new routing infrastructure we have the building blocks for creating REST based service on top of the tried and tested ASP.NET infrastructure.

Conclusion

REST is a powerful, pragmatic approach to creating services. LINQ to XML provides a simple and flexible API for processing the XML in both the service and the client. With .NET 3.5, WCF now supports an infrastructure for building REST based services on its familiar communication model. Also, with .NET 3.5 SP1 ASP.NET now has the plumbing necessary for building RESTful services on the proven ASP.NET HTTP processing platform. Which should you use? If you service is already exposed via WCF then WCF is the natural choice. However, if you are building a service from scratch you should give ASP.NET some serious thought – you have total flexibility over the processing model and are closer to the web based nature of REST.