An Introduction To RESTful Services With WCF

Contents

[Learning about REST](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070003)  
[An Abstract Example](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070011)  
[Why Should You Care about REST?](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070023)  
[WCF and REST](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070034)  
[WebGetAttribute and WebInvokeAttribute](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070038)  
[UriTemplate and UriTemplateTable](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070046)  
[WebHttpBinding and WebHttpBehavior](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070050)  
[WebServiceHost and WebServiceHostFactory](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070063)  
[Using the Example Code](http://msdn.microsoft.com/en-us/magazine/dd315413.aspx#id0070078)

This is the first in a series of columns about building Windows Communication Foundation (WCF) services using the architectural style known as Representational State Transfer (REST). You could say that 2009 will be the year of REST here in the Service Station column, and it's long overdue in my opinion. REST has been a popular style for many years since its introduction in 2000.

In this first column, I want to discuss some of the basic tenets of REST, as well as present an implementation of a RESTful service using WCF. In future columns, I'll dig into more details of the base ideas of REST, as well as technologies that are built on that base.

**Learning about REST**

An architectural style is a set of constraints that can be applied when building something. And an architectural style of software is something that describes the features that can be used to guide the implementation of a software system. REST is an architectural style that can be used to build software in which clients (user agents) can make requests of services (endpoints). REST is one way to implement a client-server architectural style—in fact, REST explicitly builds on the client-server architectural style.

A man named Roy Thomas Fielding first coined the term REST as a concept in his PhD dissertation (["Architectural Styles and the Design of Network-based Software Architectures](http://ics.uci.edu/~fielding/pubs/dissertation/top.htm)"). He was one of the people who worked on the specification that drives most of the Internet today: Hypertext Transfer Protocol (HTTP). Normally the background of the people describing an architectural style isn't relevant to a discussion of the style, but here I think it is important because I believe one of the best ways to get a basic understanding of REST is to think about the Web and how it works.

I have to assume you, as a developer, are familiar with (and probably, like me, are a heavy daily addict/user of) the Web. Arguably, the Web can be regarded as the largest, most scalable, and most popular distributed application of all time. The constraints of REST are based on the same underlying principles that govern the Web. Those principles are:

User agents interact with resources, and resources are anything that can be named and represented. Each resource can be addressed via a unique Uniform Resource Identifier (URI).

Interaction with resources (located through their unique URIs) is accomplished using a uniform interface of the HTTP standard verbs (GET, POST, PUT, and DELETE). Also important in the interaction is the declaration of the resource's media type, which is designated using the HTTP Content-Type header. (XHTML, XML, JPG, PNG, and JSON are some well-known media types.)

Resources are self-descriptive. All the information necessary to process a request on a resource is contained inside the request itself (which allows services to be stateless).

Resources contain links to other resources (hyper-media).

**An Abstract Example**

A service that uses the architectural style of REST is generally referred to as a RESTful service or endpoint. Just to get a feeling for the ideas behind this architectural style, I'm going to present a small example. Imagine I needed to build a service to work with the data behind *MSDN Magazine*—a service that could tell me all of the years *MSDN Magazine* has been published and each of the articles in each issue. Let's say the requirement is that the editors of the magazine could use this service to add new articles and manage the data for upcoming issues.

When building RESTful services, you can go through a very simple set of basic steps to design your service:

1. What are your resources going to be?
2. What are the URIs that are going to be used to represent those resources?
3. What parts of the uniform interface (HTTP verbs) is each URI going to support?

These are the basic building blocks of a RESTful endpoint: resources and their representations, URIs for those resources, and the parts of the uniform interface to which each URI will respond. There are more advanced features you can take advantage of, such as more explicit use of status codes and using hyperlinks to manage resource state, but for this example I'm going to stick to the basics.

Now I can use these steps to design my hypothetical service. The resources are all of the years that *MSDN Magazine* has been published, all of the issues published each year, and all of the articles published in each magazine. For the purposes of this particular example, I'm going to use a media-type application/xml (XML) to represent these resources, but it's important to remember that RESTful services are not in any way limited to XML as the media type.

Next, I need to determine the URIs for each resource. Right now I only need to determine the relative URIs since the absolute URI will be determined by where I host the endpoint. The list of years will be the root URI of the service (/). Using this syntax, /{year} will return all of the issues for each year; /{year}/{issue} will be the URI for each issue (I'll identify each issue by its month of publication); and /{year}/{issue}/{article} will represent each article (I'll assume each article is numbered from 1 to n in each issue).

Next comes the mapping of URIs to the uniform interface. Since the magazine's history should really be read-only, the root resource will only expose GET. A new year can be added by doing a PUT to the URI /{year}. PUT is used to create new resources when the URI of the new resource is known by the client, as it would be in this case. PUT can also be used to update existing resources when the URI is known. POST is used to create a resource when the URI of the new resource isn't known by the client, so POST will be the verb I'll use when adding a new article resource, which would be sent to the /{year}/{issue} URI.

I can go on with each resource and each verb, but hopefully you've gotten a feel for the steps you go through to determine the design of a RESTful endpoint. You can see the full list of resources, URIs, and verbs in **Figure 1**.

|  |
| --- |
| Figure 1 Operating system support |

|  |  |  |
| --- | --- | --- |
| **Resource** | **URI** | **Verbs** |
| All years | "/ " | GET |
| A particular year's issues | "/{year}" | GET, PUT |
| A particular issue | "/{year}/{issue}" | GET, PUT |
| An article | "/{year}/{issue}/{article}" | GET, POST (the article number will be assigned by the system), PUT, DELETE (delete would be turned off once an issue has been published) |

If I wanted to consume the January 2006 articles as a client, I'd do an HTTP GET to /2006/January. If I were the editor and wanted to add an article to December 2008, the client would POST an article resource to /2008/December and a new article would be added to that issue. This is the pattern I would continue to use over and over to be able to consume this service as a client.

**Why Should You Care about REST?**

So now that I've explained a little about REST, you are probably wondering why you should care. As a developer, you need motivation to learn and adopt any style, technology, or pattern. If you are reading this magazine, you likely are a developer in the Microsoft technology stack. And for implementing client-server applications, you likely have used another architectural style: remote procedure call (RPC). Whether you have used proprietary RPC systems such as DCOM or .NET Remoting, or used interoperable RPC technologies such as SOAP using ASMX or WCF, these are the implementations of the client-server style we've had on the Microsoft platform. So why learn or use REST?

In my mind, there are two main reasons. First, REST offers some significant features and benefits over RPC technologies in many cases. Second, Microsoft is moving many of its own implementations away from RPC technologies (such as SOAP) and toward REST. This means that even if you aren't convinced or motivated to use REST to build your own systems, as more frameworks and technologies from Microsoft (and others) move to REST, you'll need to know how to interact with them.

The following is a list of other advantages (but don't consider this list exhaustive):

**Caching** When RESTful endpoints are asked for data using HTTP, the HTTP verb used is GET. Resources returned in response to a GET request can be cached in many different ways. Conditional GET, which is a way that a client can check with the service if his version of the data is still the current version, is an optional implementation detail a RESTful endpoint can implement that can further improve speed and scalability.

**Scale-Out** REST encourages each resource to contain all of the states necessary to process a particular request. RESTful services are much easier to scale out when they fulfill this constraint and can be stateless.

**Side Effects** RESTful services should have no side effects when you ask for a resource using GET (unfortunately, this constraint is easier to break than some of the other REST constraints).

**Idempotent** The other two main HTTP verbs typically used as part of the uniform interface are PUT and DELETE. PUT is most often used when a user agent wants to modify a resource, and DELETE is self-descriptive. The important bit (and what the word idempotent describes) is that you can use these two verbs on a particular resource more than once, and the effect will be the same as the first time you used them—or at least there won't be any further effect. This is reassuring when building reliable distributed systems in which errors, network failures, or latency might cause code to execute multiple times.

**Interoperability** Many people tout SOAP as being the most interoperable way to implement client-server programs. But some languages and environments still don't have SOAP toolkits. And some that do have toolkits are based on older standards that can't always communicate with toolkits that implement newer standards. REST only requires an HTTP library to be available for most operations (an XML library, of course, is often useful as well), and it is certainly more interoperable than any RCP technology (including SOAP).

**Simplicity** This advantage is more subjective than the others and can mean different things to different people. To me, the simplicity of using REST relates to URIs representing resources and the uniform interface. As an accomplished Web surfer, I understand typing different URIs into my browser to get different resources (this is sometimes known as URI or URL hacking, but it is not nefarious in any way). Because of this experience in using URIs for many years, designing URIs for resources seems very natural to me. Using the uniform interface simplifies development by freeing me from having to build an interface, contract, or API for each service I need to build. The interface (how clients will interact with my service) is set by the constraints of the architecture.

As I said, this is not an exhaustive list nor should you take it as conclusive evidence that REST is the one and only true technology to use always. You should be aware of the advantages so you can leverage them when appropriate.

**WCF and REST**

WCF is the Microsoft framework for building applications that communicate over a network, regardless of the style or protocol. The concept behind WCF was to create a framework that was extensible and pluggable so that developers could learn one programming and configuration model and be able to apply those skills to many different kinds of distributed systems.

While it is true that much of WCF is geared toward RPC (using SOAP), it actually has had the ability to expose and consume REST services since it was first released as part of the .NET Framework 3.0. What it lacked was a programming model needed to make using REST with WCF easy. There also were some pieces of infrastructure that you had to build to make REST work with the .NET Framework 3.0. Both a programming model and those pieces of infrastructure were added to WCF in the .NET Framework 3.5 in the System.ServiceModel.Web assembly. And the .NET Framework 3.5 SP1 added a few small improvements as well.

The programming model centers around two new attributes, WebGetAttribute and WebInvokeAttribute, and a URI template mechanism that enables you to declare the URI and verb to which each method is going to respond. The infrastructure comes in the form of a binding (WebHttpBinding) and a behavior (WebHttp­Behavior) that provide the correct networking stack for using REST. Also, there is some hosting infrastructure help from a custom Service­Host (WebServiceHost) and a ServiceHostFactory (WebServiceHostFactory).

**WebGetAttribute and WebInvokeAttribute**

One of the ways that WCF simplifies building connected systems is by routing network messages to methods on instances of the classes you define as implementations of your service. This allows you to concentrate on the logic of the code in your services rather than on the infrastructure necessary to process network traffic.

By default, WCF does this routing (also known as dispatching) based on the concept of action. For this dispatching to work, an action needs to be present in every message that WCF receives on your behalf. Each unique action is mapped to a particular Action method.

The value of Action is based either on the name of your method (plus the namespace of your service) or a custom value (set via the OperationContractAttribute.Action property). This system of routing is tightly coupled to SOAP since the SOAP specification's Action header is used by default. Luckily, like almost everything else in WCF, this default dispatching infrastructure is replaceable.

When you use the REST infrastructure with WCF, the default dispatcher is replaced by one that routes based not on Action, but instead based on the URI of the incoming request and the HTTP verb being used. This routing (done by a class called WebHttpDispatchOperationSelector) enables you to easily implement a RESTful endpoint. This dispatcher is configured on each endpoint by a behavior named WebHttpBehavior, which must be added to each endpoint from which you want to use this programming model (though you don't often have to do this manually, as you'll see later).

The key to making this work is for the WebHttpDispatch­OperationSelector to know how to map different URIs and verbs to your methods. For this, the WebGetAttribute and WebInvokeAttribute must be added to the methods on your WCF ServiceContract type.

WebGetAttribute tells the dispatcher that the method should respond to HTTP GET requests. WebInvokeAttribute is mapped to HTTP POST by default, but the WebInvokeAttribute.Method property can be set to support any of the other HTTP verbs (PUT and DELETE being the two most common). By default, the URI is determined by the name of the method (added onto the base URI of the endpoint).

This is not really very RESTful, since method names are not what you want to deal with in REST because they represent verbs. What you want to expose as URIs are nouns. For this reason, the WCF REST programming model allows customization of URIs for each method by using templates that can be set via the UriTemplate property on WebGetAttribute or WebInvokeAttribute.

**UriTemplate and UriTemplateTable**

To enable customization of the URI for each method and verb combination, WCF added the ability to define the URI for each resource by using a special template syntax, such as the one I used earlier in this column for describing the *MSDN Magazine* service endpoint. This syntax allows you to define, with replaceable tokens, the URI structure you'd like each method to represent in conjunction with the HTTP verb (via the WebGetAttribute or WebInvokeAttribute). I'll get into the syntax in more detail in a future column.

**Figure 2** shows the WCF ServiceContract definition for the *MSDN Magazine* service (with the appropriate attributes and UriTemplate customizations) and applies the features I mentioned earlier. It extends the existing WCF contract definition system with the WebGetAttribute for those operations that should respond to GET. It also adds WebInvokeAttribute to the operation in order to respond to any other verbs.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 2 WCF ServiceContract Definition

[ServiceContract]

public interface IMSDNMagazineService

{

[OperationContract]

[WebGet(UriTemplate="/")]

IssuesCollection GetAllIssues();

[OperationContract]

[WebGet(UriTemplate = "/{year}")]

IssuesData GetIssuesByYear(string year);

[OperationContract]

[WebGet(UriTemplate = "/{year}/{issue}")]

Articles GetIssue(string year, string issue);

[OperationContract]

[WebGet(UriTemplate = "/{year}/{issue}/{article}")]

Article GetArticle(string year, string issue, string article);

[OperationContract]

[WebInvoke(UriTemplate = "/{year}/{issue}",Method="POST")]

Article AddArticle(string year, string issue, Article article);

}

In this case, in the AddArticle method I added Method="POST" for readability, as the default verb for WebInvokeAttribute is POST. Both the GET and POST methods have URI customization using the UriTemplate attribute. Notice that the UriTemplate syntax allows for multiple variable path segments, and that each of those path segments are passed to the methods as arguments.

**WebHttpBinding and WebHttpBehavior**

In WCF, a binding determines how WCF is going to communicate. A binding is really the configuration that tells WCF how to build what is known as the channel stack, which is the set of objects that will work together to provide the type of communication you want for a particular endpoint.

For a RESTful endpoint, the binding you use is WebHttpBinding. Unlike many other bindings, WebHttpBinding is fairly simple, containing only two components: the HTTP transport and the text message encoder (set to not expect or use SOAP, just plain XML).

As I mentioned earlier, WebHttpBehavior is the object that causes the URI-plus-verb dispatcher to be used, so the WebHttpBinding and the WebHttpBehavior are almost always used together. Here is the code for creating such an endpoint if you are self-hosting a WCF RESTful endpoint:

ServiceHost sh =

new ServiceHost(typeof(MSDNMagazineServiceType));

string baseUri = "http://localhost/MagazineService";

ServiceEndpoint se =

sh.AddServiceEndpoint(typeof(IMSDNMagazineService),

new WebHttpBinding(), baseUri);

se.Behaviors.Add(new WebHttpBehavior());

sh.Open();

Notice that not only do I have to specify the WebHttpBinding when adding the endpoint to the ServiceHost, I also have to explicitly add the WebHttpBehavior to the endpoint to make the URI-plus-verb dispatching system work. Of course, I can do this with configuration as well (see **Figure 3**).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 3 URI-Plus-Verb Dispatching in Configuration

<configuration>

<system.serviceModel>

<services>

<service name="MSDNMagazine.MSDNMagazineServiceType">

<endpoint

address="http://localhost/MagazineService"

binding="webHttpBinding"

contract="MSDNMagazine.IMSDNMagazineService"

behaviorConfiguration="webby"/>

</service>

</services>

<behaviors>

<endpointBehaviors>

<behavior name="webby">

<webHttp/>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

**WebServiceHost and WebServiceHostFactory**

One of the complaints about WCF is that it is sometimes too complex, especially in terms of configuration. To alleviate this concern with RESTful endpoints (again, simplicity is one of the oft-stated advantages of REST), Microsoft added two new types, WebServiceHost and WebServiceHostFactory, to the .NET Framework 3.5.

WebServiceHost is a ServiceHost-derived type, which simplifies self-hosting scenarios of RESTful endpoints. The code to self-host with WebServiceHost looks like this:

string baseUri = "http://localhost/MagazineService";

WebServiceHost sh =

new WebServiceHost(typeof(MSDNMagazineServiceType),

new Uri(baseUri));

sh.Open();

This is a nice optimization, as it avoids the repetitive code of adding the WebHttpBinding and WebHttpBehavior manually. The WebServiceHost class automatically creates the endpoint and configures it with the WebHttpBinding and WebHttpBehavior.

In the managed hosting scenario with WCF, inside Internet Information Services (IIS), WCF normally requires an .svc file that points to the service type, plus entries in the web.config file to inform WCF about the endpoint (the binding and behaviors among other configuration).

To simplify the managed hosting scenario, Microsoft added WebServiceHostFactory, which uses an open WCF extensibility point (using a custom ServiceHostFactory type) in the managed hosting scenario to create a configuration-free experience for many RESTful services. The .svc file looks like this:

<%@ ServiceHost Factory=

"System.ServiceModel.Activation.WebServiceHostFactory"

Service="MSDNMagazine.MSDNMagazineServiceType" %>

WebServiceHostFactory creates an instance of the WebServiceHost, and since the WebServiceHost will auto-configure the endpoint using WebHttpBinding and WebHttpBehavior, there doesn't need to be any configuration for this endpoint in the web.config at all. (Of course, if you need to customize the binding, you have to use the configuration system or build a class that derives from WebServiceHost/WebServiceFactory). If I did need to customize the binding, I could still add the appropriate entries in the configuration file. **Figure 4** shows a configuration file that would turn on HTTP basic authentication on my service endpoint.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 4 Turn on HTTP Basic Authentication

<configuration>

<system.serviceModel>

<services>

<service name="MSDNMagazine.MSDNMagazineServiceType">

<endpoint

address="http://localhost/MagazineService"

binding="webHttpBinding"

contract="MSDNMagazine.IMSDNMagazineService"

behaviorConfiguration="webby"/>

</service>

</services>

<bindings>

<webHttpBinding>

<binding name="secure">

<security mode="Transport">

<transport clientCredentialType="Basic"/>

</security>

</binding>

</webHttpBinding>

</bindings>

<behaviors>

<endpointBehaviors>

<behavior name="webby">

<webHttp/>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

**Using the Example Code**

Through the explanation of the features of WCF in relation to REST, I've laid out most of the implementation of the service I proposed at the beginning of the column. Let me walk through interacting with this service.

Once I have the service up and running, I can hit its root URI with any client, including a Web browser such Internet Explorer. Being able to run quick tests against RESTful endpoints with a browser is, most would agree, a pretty useful consequence of the REST architectural style being based on the way the Web works. You can see the result in **Figure 5**.

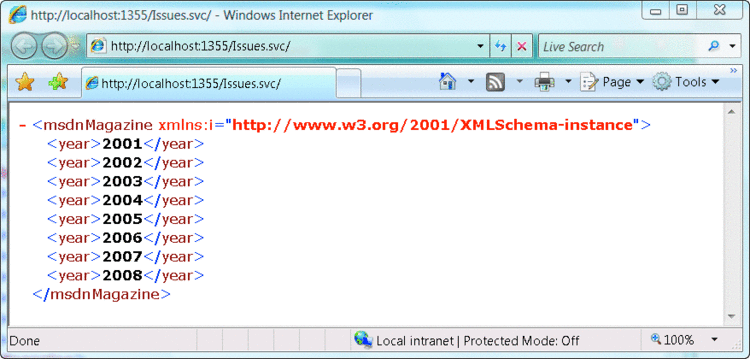


Figure 5 **Root Resource URI**

In this case, I am hosting in the Visual Studio 2008 Web Development Server, which dictates the base URI. The Issues.svc file is the necessary file for WCF in the managed hosting scenario. If I want to see the result for a particular year, I can just add that year to the address (the URI) in the browser (see **Figure 6**).

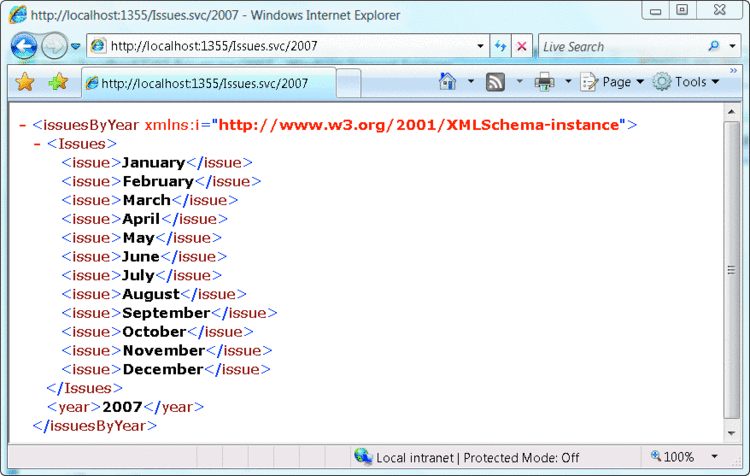


Figure 6 **The Resource Representing the Year 2007**

If I asked for October of 2008, the URI would be localhost:1355/Issues.svc/2008/October, which, at this point, would be an empty set. And if I wanted to add an article, I'd do an HTTP POST to that URI with the XML representation of an article as the HTTP request body.

Another nice feature of HTTP is all of the tools that are available to interact with it. One of my favorites is [Fiddler](http://fiddlertool.com), which allows me to "spy" on HTTP requests and responses. But it also has a feature that allows me to do arbitrary HTTP operations. So I can use the Fiddler Request Builder tab to do my HTTP POST (see **Figure 7**). You can see a request for the October 2008 resource after the POST in **Figure 8**.

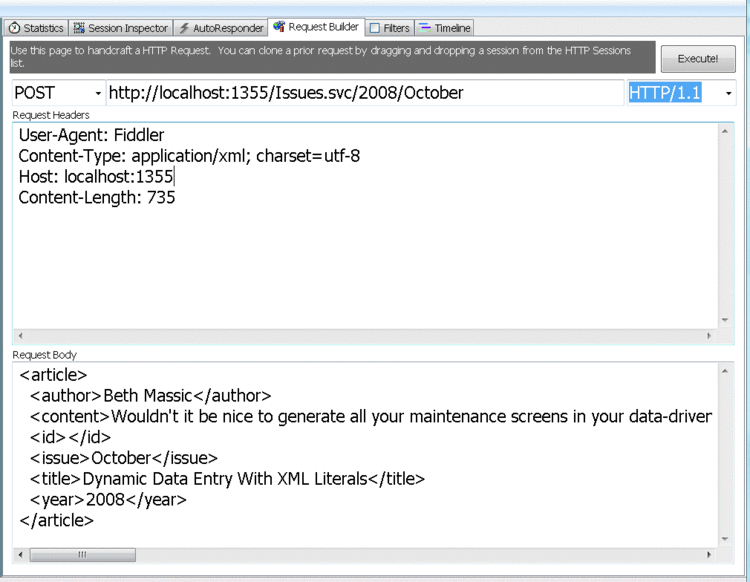


Figure 7 **Making an HTTP POST Request to Create a New Article Resource with Fiddler**

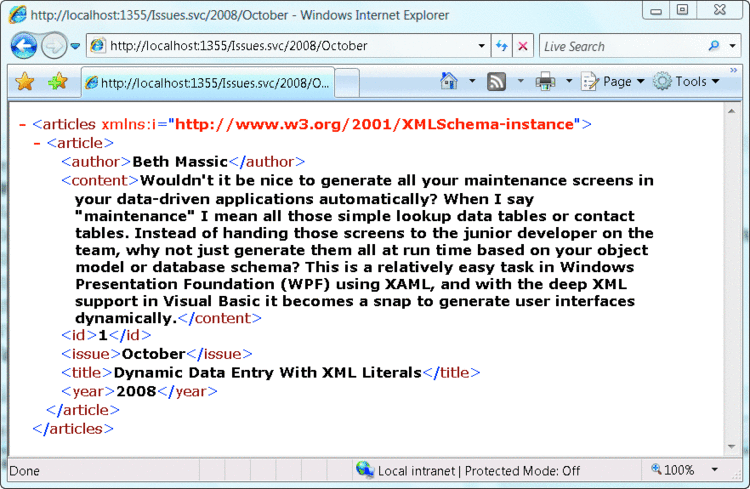


Figure 8 **Article Resource Added**

Although it can be argued that Internet Explorer and Fiddler are real clients, building an actual client-server implementation is an extension of these simple steps I just went through (a more complex example of building a RESTful client will come in a later column). A client needs to know the URI of each resource and what parts of the uniform interface to use against each URI. The client can then interact with the service by using these pieces of information to build out its functionality.

What I've shown you in this first column is the base set of WCF features that enable the REST architectural style to be used in your .NET applications with ease. This base enables other interesting technologies, including things such as Web Feeds (RSS and Atom) and support for JSON-encoded resources for interaction with AJAX.

In the next few columns, I'm going to build on this base knowledge of REST and WCF and delve into the details of several other features in the Microsoft platform that build on this style and technology. I'll also be dealing with some of the more commonly asked REST questions, for example, about security and about the idea of implementing processing endpoints.