WCF Web APIs, HTTP your way

.NET Connected Framework Team
1 Nov 2010 3:25 AM

- Comments 0

At PDC in my session "Building Web APIs for the Highly Connected Web" we announced WCF Web APIs, new work we are doing to make HTTP first class in WCF. In this post I am going to describe what we are doing and why. If you are saying, "just show me the bits", then just head on over to wcf.codeplex.com our new site that we just launched!

Why HTTP?



HTTP is ubiquitous and it's lightweight. Every consumer that connects to the web understands it, every browser supports it, and the infrastructure of the world wide web is built around it. That means when you travel HTTP, you get carte blanche status throughout the world wide web. It's like a credit card that is always accepted everywhere. In the past, HTTP's primary usage was for serving up HTML pages. Over time however our web applications have evolved. These newer breed are much more dynamic, aggregating data not only from the company server but from a multitude of services that are hosted in the cloud. Many of those services themselves are now being exposed directly over HTTP in order to have maximum reach.

Whereas in the past the primary consumer was a desktop / laptop PC, we've now moved into the age of devices including phones like IPhone, Android and Windows Phone as well as other portable tablets like the iPad and the upcoming Slate. Each of these devices (including the PC) have different capabilities, however one thing is consistent, they all talk HTTP.

WCF and HTTP, we're going much further.

As the industry evolves, our platform needs to evolve. Since .NET 3.5, we have been continually evolving WCF to provide better support for surfacing services and data over HTTP. We've made good progress but there is more we can do. Developers using WCF have said they want more control over HTTP. We've also heard developers asking for better support for consuming WCF services with web toolkits like jQuery. Additionally, we've heard requests about simplifying configuration, removing ceremony, more testability, and just an overall simplified model. **We hear you and we're taking action.** We're making significant enhancements to our platform to address the concerns. Below is a list of some of the improvements we are focusing on specific to HTTP which we just made available on Codeplex (available in WCF HTTP Preview 1.zip). For the jQuery work check out these excellent posts by Tomek and Yavor.

Our HTTP focus areas

Media Types and Formats



HTTP is extremely flexible allowing the body to be presented in many different media types (content type) with html, pure xml and json, atom and OData being just a few. With WCF Web APIs we're going to make it very easy for services to support multiple formats on a single service. Out of the box, we are planning to support xml, json and OData however we're also making it very easy to add on support for additional media types including those that contain hypermedia (see my talk for an exampe). This gives WCF the flexibility to service a variety clients based on their needs and capabilities.

Below is a snippet which demonstrates taking a contact returned from an operation and representing it as a png file stored in an images folder. PngProcessor derives from MediaTypeProcessor. Processors are a new extensibility point in WCF Web APIs. MediaTypeProcessor is a special processor that you derive from to support a new format.

```
public class PngProcessor : MediaTypeProcessor
{
```

```
    public PngProcessor(HttpOperationDescription operation,

        MediaTypeProcessorMode mode)

        : base(operation, mode)
    {
    }
    public override IEnumerable<string> SupportedMediaTypes
    {
        get
        {
            yield return "image/png";
        }
    }

    public override void WriteToStream(object instance,

        Stream stream, HttpRequestMessage request)

    {
        var contact = instance as Contact;
        if (contact != null)
        {

            var path = string.Format(CultureInfo.InvariantCulture,

                @"{0}bin\Images\Image{1}.png",

                AppDomain.CurrentDomain.BaseDirectory,

                contact.ContactId);

            using (var fileStream = new FileStream(path, FileMode.Open))
            {
                byte[] bytes = new byte[fileStream.Length];
                fileStream.Read(bytes, 0, (int)fileStream.Length);
                stream.Write(bytes, 0, (int)fileStream.Length);
            }
        }
    }

    public override object ReadFromStream(Stream stream,

        HttpRequestMessage request)

    {
        throw new NotImplementedException();
    }
}
```

This *same* Png formatter can sit side by side with formatters for other media types like json, xml, and atom. WCF will automatically select the right processor based on matching the request accept headers passed from the client against the SupportedMediaTypes.

To see different media types in action, check the ContactManager sample that ships with WCF Web APIs.

Registering formatters / processors

In order to register processors, we're exploring a new programmatic configuration model which allows you to configure all processors (including formatters) at a single place within your application. To configure processors, you derive from a HostConfiguration class and override a few methods. You then pass your custom configuration class to the WebHttpServiceHost or WebHttpServiceHostFactory.

In the ContactManager sample we're shipping on Codeplex you wll see the following in the Global.asax.

```
protected void Application_Start(object sender, EventArgs e)
{
    var configuration = new ContactManagerConfiguration();

    RouteTable.Routes.AddServiceRoute<ContactResource>(

        "contact", configuration);

    RouteTable.Routes.AddServiceRoute<ContactsResource>(

        "contacts", configuration);

}
```

Both the ContactResource and ContactsResource are configured with a ContactManagerConfiguration instance. That class registers the processors for each operation.

```
public class ContactManagerConfiguration : HostConfiguration
{

    public override void RegisterRequestProcessorsForOperation(

        HttpOperationDescription operation,

        IList<Processor> processors, MediaTypeProcessorMode mode)
    {
        processors.Add(new JsonProcessor(operation, mode));
        processors.Add(new FormUrlEncodedProcessor(operation, mode));
    }

    public override void RegisterResponseProcessorsForOperation(

        HttpOperationDescription operation,

        IList<Processor> processors, MediaTypeProcessorMode mode)
    {
        processors.Add(new JsonProcessor(operation, mode));
        processors.Add(new PngProcessor(operation, mode));
    }
}
```

Notice above the request is configured to support Json and FormUrlEncoding while the response supports Json and Png. These Register methods are called per operation thus configuration of processors can even be more fine grained. Plus you can reuse your configuration classes even across applications.

JsonProcessor / FormUrlEncodedProcessor.

In addition to supporting representing types in multiple formats, we also can support untyped operations with the new Json primitives that come out of our jQuery work which I mentioned above. The JsonValueSample we've included illustrates how this works.

```
[ServiceContract]
public class ContactsResource
{
    private static int nextId = 1;
    [WebInvoke(UriTemplate = "", Method = "POST")]
    public JsonValue Post(JsonValue contact)
    {
```

```
        var postedContact = (dynamic)contact;
        var contactResponse = (dynamic)new JsonObject();
        contactResponse.Name = postedContact.Name;
        contactResponse.ContactId = nextId++;
        return contactResponse;
    }
}
```

In the snippet above you can see that the Post method accepts a JsonValue and returns a JsonValue. Within it casts the incoming parameter to dynamic (there actually is an extension method AsDynamic which you can use) pulls out the name and then creates a new JsonObject which it sets some properties on and returns.

If you look in the JsonValueSampleConfiguration you will see that it accepts Form Url Encoding (something not previously possible in WCF without a lot of work) for the request and returns Json.

```
public class JsonValueSampleConfiguration : HostConfiguration
{

    public override void RegisterRequestProcessorsForOperation(

        HttpOperationDescription operation,

        IList<Processor> processors,

        MediaTypeProcessorMode mode)

    {
        processors.Add(new FormUrlEncodedProcessor(operation, mode));
    }

    public override void RegisterResponseProcessorsForOperation(

        HttpOperationDescription operation,

        IList<Processor> processors,

        MediaTypeProcessorMode mode)

    {
        processors.ClearMediaTypeProcessors();
        processors.Add(new JsonProcessor(operation, mode));
    }
}
```

This is extremely powerful for folks solely working with Uri Form Encoding and Json and who are comfortable/prefer working without a concrete type.

Queryability



One challenge when exposing data over HTTP is how to allow clients to filter that data. In WCF Web APIs we're introducing IQueryable support on the client and server for addressing these challenges.

Making the service queryable

On the server side, your service operation returns an IQueryable<T> and you annotate it with a [QueryComposition] attribute. Once you do that, your service lights up and is now queryable using the OData uri format.

We've included a QueryableSample which illustrates how this works. Below is a snippet from the CoontactsResource in that sample.

```
[WebGet(UriTemplate = "")]
[QueryComposition]
public IEnumerable<Contact> Get()
{
    return contacts.AsQueryable();
}
```

The Get method above returns an IQueryable of contacts. (Today the method must return IEnumerable<Contact> but this will be fixed in the near future).

With the query composition enabled, the host will now accept requests like "http://localhost:8081/contacts?$filter=Id%20eq%201" which says "find me the contact with an ID equal to 1".

Note: Currently this feature is not compatible with our new WebHttpServiceHost / Processors, it only works with our existing WebServiceHost. This is temporary as we are planning to migrate over to the new host / processor model.

Querying the service, LINQ to WCF

On the client side we're introducing the ability to do LINQ queries directly to resources which are exposed through query composition. We've added a CreateQuery<T> extension method which you can use with the new HttpClient (next section) to create a WebQuery<T>. Once you have that query, you can then apply a Where, or an Order by. Once you start to iterate through

the result, we will automatically do a Get request to the server using the correct URI based on the filter. The results will come back properly ordered and filtered based on your query.

Below is a snippet that shows querying an Orders resource

```
public IEnumerable<Order> GetApprovedOrders()
{
    string address = "http://contoso.com/orders";
    HttpClient client = new HttpClient(address);
    WebQuery<Order> orders = client.CreateQuery<Contact>();


    return orders.Where<Order>(o=>o.State == OrderState.Approved).


        OrderBy(o=o.OrderID);


}
```

Getting first class support for HTTP



HTTP is more than a transport, it is a rich application layer protocol. There's a lot more interesting information than just the body which lives in the headers. It is the headers that most of the web infrastructure actually cares about. For example if you want to allow requests to be cached throughout the web, you need to use entity tags which live where? In the headers. Point blank, if you want to access the full richness of HTTP you need to access those headers.

HTTP Messages

We're introducing support for HttpRequestMessage and HttpResponseMessage. These classes which originally shipped in the REST starter kit allow unfettered and strongly typed access to the underlying HTTP request and response. With these new apis you can access HTTP wherever you are, whether you are authoring a service, or extending the stack and whether you are on the server **or the client.** Another nice thing about these messages is they are easy to use in unit testing. They don't have any implicit dependencies to WCF as WebOperationContext does nor are they statically called. They are lightweight data containers that are very easy to create.

For example, you can author a service which receives the HttpRequestMessage and HttpResponseMessage, and which directly accesses the headers and the body. The HelloWorldResource below supports caching on the client side, as it returns an entity tag “HW” which the client can send in an IfNoneMatch header in subsequent requests. The resource can then then return a status 304 to tell the client to use it’s cached copy. The client in this case might not be the browser but a proxy server sitting in the middle.

```
[ServiceContract]
public class HelloWorldResource {
  [WebGet(UriTemplate="")]
  public void Get(HttpRequestMessage req, HttpResponseMessage resp) {
    if (req.IfNoneMatch.Contains("HW")) {
      resp.StatusCode = HttpStatusCode.NotModified;
      return;
    }

    resp.HttpContent.Create("Hello World Resource", "text/html");
    resp.StatusCode = HttpStatusCode.OK;
    resp.Headers.Tag = "HW"; //set the tag
  }
}
```

The code above would likely be factored into a common set of utility functions rather than being redundantly coded for each operation. The important thing is we’re providing the messages which enables that refactoring.

You can also mix and match using messages with strongly typed objects representing the body. For example you might want to do a redirect on a Get request for a document that has moved.

```
[ServiceContract]
public class DocumentResource
{
    [WebGet(UriTemplate="{name}")]
    public Document Get(string name, HttpResponseMessage resp)
    {
        Document doc;
        //foo has moved
        if (name == "Foo")
        {
            resp.StatusCode = HttpStatusCode.MovedPermanently;
            resp.Headers.Location = new Uri("http://someplace/Foo");
            return null;
        }
        //find the document
        return doc;
    }
}
```

Within the ContactManager sample you will see other examples of mixing messages with concrete types.

HTTP Client

Providing a client for consuming HTTP is equally as important as being able to expose it. For that reason we’re also bringing the HttpClient we shipped in the REST starter kit forward. You can use the new client within desktop applications or within services themselves in order to consume other HTTP services. We’re also providing extensions to the client for supporting queryability, which I will cover in the next section.

Below is a simple example of using HttpClient.

```
var client = new HttpClient();
client.DefaultHeaders.Accept.Add("text/xml");
```

```
var resp = client.Get("http://contoso.com/contacts/1");
resp.Content.ReadAsXmlSerializable<Contact>();
```

Request and response processing



When you work with HTTP, there are various parts of the request and response which need to be processed or transformed. With HttpRequestMessage and HttpResponseMessage we're allowing you to do this processing within the actual operation as there are places where this is appropriate. However, there are other cases that are concerns that are cross-cutting which don't belong in the operation. Take formatting for example. It's very convenient to have the ContactResource simply return and accept a contact, rather than it have to drop down to a message and manually do the formatting. In the same way the ContactResource operation may depend on certain values extracted from segments of the request URI like the ID. In the past we dealt with each of these concerns in a one-off basis. With WCF Web APIs we're exploring a more general purpose way to handle these concerns. We're introducing a request and response pipeline of what we're currently calling Processors. A Processor has a simple execute method with takes inputs and provides outputs. The inputs could be things like the request or response, or outputs from other processors. In this way processors are composable.

Out of the box we use processors today mainly for extracting values from the uri, for content negotation (selecting the format) and for media type formatters. However processors are extensible, and you can introduce your own for adding custom processing within the request or the response.

We've already seen above how to create processors specific for formatting. Here is an example of a different kind of processor that takes a latitude and longitude in a URI for example "http://contoso/map/12.3456,-98.7654" and converts it into Location object. Once the location processor is registered, the MapResource.Get method will automatically get a location object passed in.

```
public class Location
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
public class LocationProcessor : Processor<string, string, Location>
{
    public LocationProcessor()
    {
        this.OutArguments[0].Name = "Location";
```

```
    }

    public override ProcessorResult<Location> OnExecute(

        string latitude,

        string longitude)

    {
        var lat = double.Parse(latitude);
        var lon = double.Parse(longitude);

        return new ProcessorResult<Location> { Output =

            new Location { Latitude = lat, Longitude = lon } };

    }
}
[ServiceContract]
public class MapResource {
  [UriTemplate="{latitude},{longitude}"]
  public Stream Get(Location location) {
    //return the map
  }
}
```

The processor above inherits from Processor<T1, T2, TOutput> meaning that it takes two inputs (strings in this case) and it outputs a location. In the execute method the parameter names conventionally match against outputs coming from other processors in this case the method expects “latitude” and “longitude” params. You might be wondering where these parameters come from. If you look on the MapResource.Get method you see that it has 2 parameters named latitude and longitude respectively. A special processor UriTemplateHttpProcessor automatically extracts values from the uri and returns those values as outputs. In this case it returns latitude and longitude thus making those values available to the LocationProcessor (or any other processor).

The logic above is very simple in that it parses numbers. However, you could imagine expanding the processor to do more. For example it could be rewritten to also accept a more expressive uri like “http://contoso/map/12 deg 34’ 56” N, 98 deg 76’ 54” W ”.

This is just a small illustration of the kinds of things you can do with processors. You could imagine handling concerns related to entity tags like IfMatch / IfNonMatch in processors for example.

There’s a lot more to say about processors and their configuration. Look for more on both topics in future posts. Darrel Miller also has a nice post where he talks about processors here.

Conventions, Resources and Testability

We’ve heard plenty of feedback from folks in the community that they would like to see us offer configuration alternatives to attributes and provide more out of the box conventions. We’ve also heard developers asking for us to ensure that we provide better support for test driven development, and using tools like IoC containers. As we move forward we are definitely thinking about all of the above.

Our current focus for the platform has been to enhance the existing Web HTTP programming model to provide richer support for HTTP. These enhancements will likely roll into the framework soon and will provide a very smooth migration path for existing WCF HTTP customers.

Longer term, we are also exploring a new convention based programming model for configuring HTTP resources (services). With this new model we are also looking at how we can enhance it to be more resource oriented, for example allowing specification of child resources so that URIs can be constructed dynamically rather than being hardcoded. This new model will make it's way to Codeplex soon where we'd like to incubate it with the community.

With the new bits we are delivering we are also being intentional about designing things in a more testable manner for example HttpRequestMessage and HttpResponseMessage allow developers to move away from static calls which are difficult to test. Processors are also easy to test as they each do a single thing, and do not have static dependencies. In addition to the new bits, we are looking at investments we can make into our existing bits to better support testability. For example we are exploring allowing you to plug in an IoC container for service instantiation.

It's still early, we want your help

We're still early in the development of these new features! Not all of these features will make it in the box, but many definitely will. You can help us prioritize by checking out our new bits on Codeplex, participating in the forums and adding work items so others can vote.

OK, what are you waiting for? Head on over to wcf.codeplex.com. The future awaits!!!

- 0 Comments

WCF, WCF REST, WCF Web APIs