Creating a Rest API for consuming data on HTML5

# Introduction

Today HTML development is getting serious for project across different types of devices and experiences, and JavaScript is the language for the future web development. So it’s necessary to access to information store on the server quickly, but in a way that is comfortable to the web developer and of course to JavaScript.

# Creating a Web API

So the goal of this paper is explain how to create an extensible Web API using Http for communication channel, REST (Representational State Transfer) as the protocol and Json (JavaScript Object Notation) as the format output.

This API will support authorization process, custom client encoding, lightweight output support and much more.

We’re going to focus on how to create this API using Microsoft technologies, so instead of creating a custom http pipeline for consuming http request, we’re going to use WCF (Windows Communication Foundation) as the primary platform for creating this API.

Of course you can use other Web Technologies, like ASP.NET WebForms, ASP.NET MCV o either PHP. The decision of choosing WCF is for the flexibility it’s brings to the developer in terms of extensibility and customization of the pipeline.

Before we starting talking about the API, let’s start covering some of the concepts of the paper, REST, Json and WCF.

## REST

REST-style architectures consist of clients and servers. Clients initiate request to servers. Servers process requests and return appropriate response. Requests and responses are built around the transfer of representation of resources, and those resources are a combination of different properties of the HTTP channel.

In REST the http verb is used to match the action you want to perform on the server:

* **GET**: List
* **PUT**: Replace
* **POST**: Create
* **DELETE**: Delete

The resource in a REST-style environment is represented by the URI (Universal Resource Idenfiticator) itself, so that means that we are able to setup a complete request in the url.

http://server/northwind/northwind.rse/Customers[City eq ‘London’]?$orderby=ContactName

## JSON

JSON or JavaScript Object Notation, is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. [[1]](#footnote-1)

The goal of JSON is to be a simple, lightweight format to exchange data from endpoints, there are a basic data type support, like Number, String, Boolean, Array, Object and null.

Here is an example of JSON:

{

"firstName": "John",

"lastName": "Smith",

"age": 25,

"address":

{

"streetAddress": "21 2nd Street",

"city": "New York",

"state": "NY",

"postalCode": "10021"

},

"phoneNumber":

[

{

"type": "home",

"number": "212 555-1234"

},

{

"type": "fax",

"number": "646 555-4567"

}

]

}

## WCF

Windows Communication Foundation is a communication API that is available on the .NET framework for building connected, service-oriented application. WCF include the support for invoking services across different and heterogeneous channels, formats and protocols.

# The API

The main goal of the API is creating a platform for creating and consuming services through a JavaScript client API.

The only knowledge from the client is the url, so in our API we will be able to make request like this:

http://server/services.svc/MyContract/MyOperation?param1=1&param2=theString

Based on this URI the server is able to locate the Contract (Class), the Operation (Method) and the parameters for this Operation (encoded in the QueryString). Then the server will execute this request and return back to the client JSON with the response as the body of the response.

There are several ways to define the contract and the operation on the url, but in this API we’re going to use the next two segments after the service file. Of course this can be changed at any time.

Let’s dig into the code.

## WCF SerciceContract and OperationContract

Since we’re going to use WCF as the primary platform for hosting our API, we need to create the main entry point for all requests. WCF have a WebAPI for invoking WCF services through http.

This is the definition of the service

[ServiceContract]

public interface IService

{

[OperationContract]

Stream ProcessRequest();

}

There is only Operation named ProcessRequest, where all the request will be processed, here is the implementation of the service.

public class ServiceEngine : IService

{

[WebGet(UriTemplate = "\*")]

public Stream ProcessRequest()

{

return new OperationExecutor().ProcessRequest();

}

}

All the magic of processing all the request is done by adding the WebGet attribute to the Operation and setting the UriTemplate with the wildcard \*.

This method is also the entry point for our API implementation, and the class OperationExecutor is the class responsible for executing and process the request.

The return type of the Operation is a Stream. This is the way we can serialize the content by ourselves and return back a MemoryStream with the whole response.

## Authorization

As you may imagine, there is no authorization process involve in this API. This is because we need to hook-up this with the Authorization engine on WCF and creating a custom class for holding the authorization process.

WCF have a well-known authorization pipeline for any type of service, there are two defaults implementations one using Windows users and another one using ASP.NET Membership provider. Also we can create our custom class for authorization process, here is the code:

public class RestServiceAuthorizationManager : ServiceAuthorizationManager

{

protected override bool CheckAccessCore(OperationContext operationContext)

{

if (RestConfiguration.IsAuthorizationEnabled)

{

bool result = true;

if (new OperationExecutor().Operation == "LoginUser")

{

result = true;

}

if (!result)

{

//result = new UserRepository().IsValidAuthorization();

}

if (result)

{

operationContext.ServiceSecurityContext.AuthorizationContext.Properties["Principal"] = Thread.CurrentPrincipal;

}

return result;

}

else

{

operationContext.ServiceSecurityContext.AuthorizationContext.Properties["Principal"] = Thread.CurrentPrincipal;

return true;

}

}

}

You can enable or disable Authorization process by setting the RestConfiguration.IsAuthorizationEnabled to true.

Once the authorization is enabled, the process for authorize a user in WCF is return true on this method and set the property Principal in the AuthorizationContext bag to the current principal token of the thread.

Here is where the user has a chance to validate the user and allow continuing executing code in the WCF pipeline.

# Executing the environmental Operation

Before starting with the implementation of the Operation Executor class, which holds the whole pipeline for the request, we have to talk about IoC (Inversion of Control).

We want to make our API testable, that’s means that we need to run the API outside of the WCF runtime pipeline, in that case, inside the Visual Studio Testing Runtime.

Inversion of Control is a new way to define behavior and use a container to resolve the class that will implement the functionality at runtime. Developing the software in that way helps us to compose the functionality at runtime and decide if we’re using the real object or a fake one created especially for testing purposes.

Here is the code for ProcessRequest:

public Stream ProcessRequest()

{

Stream result = null;

Type service = FindTypeByName(ServiceContract);

if (service != null)

{

object value = Activator.CreateInstance(service);

Dictionary<string, object> parameters = ProcessParameters(value);

object operationInvokeResult = InvokeService(parameters, value);

result = ConvertToStream(ConvertResult(operationInvokeResult));

if (RestConfiguration.IsAuthorizationEnabled)

{

IResponseAuthorization auth = UnityContainerProvider.GetUnityContainer().Resolve<IResponseAuthorization>();

if (auth != null && Operation != "LoginUser")

{

auth.AddAuthorizationHeader();

}

}

}

else

{

throw new InvalidOperationException(

string.Format("Can't process the request because the Operation Contract {0} can't be found", ServiceContract));

}

return result;

}

Processing a request is a 4 steps process:

* Find the Service type and activate it.
* Process the parameters of the Operation
* Invoke the operation
* Serialize the result and convert to an Stream
* [Optional] Add the authorization header to the response

Let’s discuss more about this pipeline.

## Find the Service type and activate it

Finding the Service type involve getting the Service string from the Uri and then find this type on the current assemblies. Finding the type itself it just find a class with the same name of the Service.

Here is the code:

private static Type InternalFindType(string value)

{

Type result = null;

Type[] all = typeof(TypeFinder).Assembly.GetTypes();

result = (from p in all

where p.Name == value

select p).FirstOrDefault();

return result;

}

## Process the parameters of the Operation

As we said before in a Rest-style request all the parameters are present in the query string of the Uri. But .NET is a strongly typed language and I need to provide the parameters on the target type, so I need to convert the string representation of the parameter to the target type.

Microsoft provide a class for holding the type conversion for the query string, QueryStringconverter, which has two methods one for query if the class is able to convert this type and another one for convert the string itself.

So if I want to invoke an operation for a Service, first I need a reference to the MethodInfo of the method I want to invoke to access to his parameter information, and bases on this parameter information try to convert the query string.

Here is the complete code:

private Dictionary<string, object> ProcessParameters(object value)

{

QueryStringConverter converter = new QueryStringConverter();

NameValueCollection query = HttpUtility.ParseQueryString(uriProvider.GetUri().Query);

Dictionary<string, object> parameterValues = new Dictionary<string, object>();

MethodInfo methodInfo = value.GetType().GetMethod(Operation, BindingFlags.Public | BindingFlags.Instance);

if (methodInfo != null)

{

ParameterInfo[] parameters = methodInfo.GetParameters();

foreach (var parameter in parameters)

{

if (converter.CanConvert(parameter.ParameterType))

{

object parameterValue = converter.ConvertStringToValue(query[parameter.Name], parameter.ParameterType);

parameterValues.Add(parameter.Name, parameterValue);

}

else

{

throw new InvalidOperationException(string.Format("Can't convert parameter type {0}", parameter.ParameterType.FullName));

}

}

}

else

{

throw new InvalidOperationException(string.Format("{0} is not found", Operation));

}

return parameterValues;

}

## Invoke the operation

Invoking the Operation is invoking the method of the class passing the parameters supplied by the QueryString, we now have the parameters and the instance of the class.

Here is the code:

private object InvokeService(Dictionary<string, object> arguments, object serviceInstance)

{

object result = null;

MethodInfo methodInfo = serviceInstance.GetType().GetMethod(Operation, BindingFlags.Public | BindingFlags.Instance);

if (methodInfo != null)

{

result = methodInfo.Invoke(serviceInstance, arguments.Values.ToArray());

}

return result;

}

## Serialize the result and convert to an Stream

Our API use Json as the content format for the output so we need to serialize the result of the invocation of the Operation.

Here is the code:

private string ConvertResult(object value)

{

string result = null;

if (value != null)

{

DataContractJsonSerializer serializer = new DataContractJsonSerializer(

value.GetType());

MemoryStream ms = new MemoryStream();

serializer.WriteObject(ms, value);

ms.Position = 0;

Encoding encoding = GetResposeEncoding();

result = encoding.GetString(ms.ToArray());

}

return result;

}

In the WCF infrastructure there is a DataContractJsonSerializer which helps us to serialize an instance of a class to Json. That means that the whole object graph will be serialized, including complex object and collections.

Then, using the Response encoding supplied by the client, we convert the in-memory byte representation of the Json to a string and return this string back.

## Add the authorization header to the response

The final step of the processing the request is adds the authorization header to the response. As we said before we can enable or disable the Authorization process.

The authorization header is a base64 string encrypted and hashed, returned back to the client.

This is how we create the AuthorizationString in code:

private string GenerateAuthorizationString(RestToken token)

{

string result;

token.EnsureIsNotNull();

string key = "{0}!{1}!{2}!{3}!{4}";

key = string.Format(key, token.OAuth, token.Secret, token.TwitterId, token.UserId, DateTime.Now.Ticks);

ICryptoTransform transform = new TripleDESCryptoServiceProvider().CreateEncryptor(TripleDESKeys.Key, TripleDESKeys.IV);

byte[] input = Encoding.Default.GetBytes(key);

byte[] buff = new byte[input.Length];

buff = transform.TransformFinalBlock(input, 0, input.Length);

result = Convert.ToBase64String(buff);

return result;

}

# Creating the first Service

We’re going to create a demo service to return back the current DateTime.

public class DemoService

{

public DateTime GetDateTime()

{

return DateTime.Now;

}

}

The url of this operation will be:

http://localhost:12999/Services/Service.svc/DemoService/GetDateTime

# Consuming the services from JavaScript

This API was designed to be consumed from the client, so we need to create a JavaScript library to manage the entire lifecycle, authorization and json requests.

Here is the code:

var api = new RestAPI();

api.getDateTime(processDateTime);

function processDateTime(data) {

alert(data);

}

In JavaScript is normal to pass callback (references to functions) in order to return data back to the client, is the natural way to execute the workflow of your application.

Let’s take a look at the getDateTime function:

RestAPI.prototype.getDateTime = function (callback) {

var url = this.getUrl('DemoService', 'GetDateTime');

this.processRequest(url, callback);

};

The first step is getting the full qualified url

RestAPI.prototype.getUrl = function (service, operation) {

return this.baseAddress + '/' + service + '/' + operation;

}

And then make the process the request to the server,

RestAPI.prototype.processRequest = function (url, callback) {

var xdr = this.getCrossDomainRequest();

var local = this;

if (!xdr) {

jQuery.get(

url,

null,

function (data, textStatus, jqXHR) {

if (jqXHR) {

local.processLoginResponse(data, jqXHR);

callback(data);

}

else {

jQuery.error('getPhotos error: ' + textStatus);

}

}

, 'json'

);

}

else {

xdr = new XDomainRequest();

if (xdr) {

xdr.onload = function () {

var data = eval(xdr.responseText);

local.processLoginResponse(data);

callback(data);

};

xdr.

xdr.open("get", url);

xdr.send();

}

}

};

# Conclusion

This API is the basic setup in WCF to process request and match urls with classes and methods. Of course this can be modified, for example, to add XML support on the response based on a parameter or a content type. For complex types for the input, can be serialized on the body content of the request, and the API could read the content and match this complex object with the parameters of the method. Remember that this case, break the REST style for creating request, since REST is only an uri with empty body.

1. http://en.wikipedia.org/wiki/JSON [↑](#footnote-ref-1)