

## **REST-service Framework (HTTP/REST Services). v1.5.2.**

REST-service component is a server-side framework that allows easy creation and working with HTTP and REST services within any ASP.NET application. Once a REST-service is defined it can be consumed via regular URL, or using client-side javascript call that resembles the standard C# style function call that is expected to be used within server-side code calling the same method. The REST-service can return any type of data, such as string, byte array, reference to a file that should be sent to a client as a call response, etc. In addition HTML templates functionality can be utilized by building an *ascx* file template (like regular web user controls) and data model that REST-service method is supposed to return – the framework will process ASCX template by applying data model to it and returning its result to the client. The REST-service framework also provides control over security, and has built-in configuration that is 100% controlled by a developer, including ability to use class alias instead of real class to hide the internal namespaces and classes' structure used by the server code backend.

### **Contents**

Part 1: Calling REST-service via URL. ....	2
Part 2: Calling REST-service via Javascript from the client.....	2
Part 3: Using javascript callback functionality to provide handling for the REST-service result.....	3
Part 4: Using javascript dynamic callback on the client side. ....	3
Part 5: Creating REST-service. ....	5
Part 6: Configuring REST-services framework.....	6
Part 7: Using REST-service Filters. ....	8
Part 8: Using HTML Templates. ....	10
Part 9: Using HTML Templates with REST-services. ....	11
Part 10: Returning file as a response from REST-service.....	12
Part 11: Returning JSON data as a response from REST-service.....	12
Part 12: Returning other data types as a response from REST-service. ....	13
Part 13: Mapping HTTP Message Body to a complex type parameter.....	14
Part 14: Client side custom error handling. ....	17
Part 15: Client side automatic data parsing from JSON to an object. ....	17
Part 16: Using RestClientManager control. ....	18
Part 17: Using ASP.NET Output Cache with REST-services. ....	18
Part 18: ASP.NET Routing support and CRUD (Create, Read, Update, and Delete) compatibility.....	19
Part 19: Adding handler mapping for REST extension to IIS.....	22
Part 20: REST-service Metadata Explorer. ....	23

**Disclaimer:** The main purpose of building this framework was to allow the creation of HTTP or REST services in such a way that is easy to use, easy to setup, to not be limited by returned type of data, and making sure that the internal application code can be reused for service purposes. Although a number of measures were taken, it was not the goal to achieve 100% compatibility with the RESTful architecture.

## Part 1: Calling REST-service via URL.

The following call demonstrates how to send a request to REST-service  
`TestWeb25.Components.GetExtraInfo(int infold, int userId)` method:

```
http(s)://<hostname>/TestWeb25/Components/InfoPreviewHandler.rest?GetExtraInfo&infold=356&userId=577  
OR  
http(s)://<hostname>/TestWeb25.Components.InfoPreviewHandler.rest?GetExtraInfo&infold=356&userId=577
```

where:

- `TestWeb25.Components` – namespace where the class is defined;
- `InfoPreviewHandler` – class name that represents REST-service;
- `GetExtraInfo` – method name within the class;
- `infold, userId` – two parameters that method accepts.

Use this code to build this URL for you:

```
TestWeb25.Components.InfoPreviewHandler info = new InfoPreviewHandler();  
string urlToCall = info.GetUrlFunctionCall("GetExtraInfo", 356, 577);
```

## Part 2: Calling REST-service via Javascript from the client.

The following call demonstrates how to send a request to REST-service:

```
TestWeb25.Components.InfoPreviewHandler.GetExtraInfo(356, 577);
```

Before using this call, the calling class javascript proxy has to be pre-registered using the following code:

```
//register javascript client for specific type  
WebRestClient.RegisterJSClient(typeof(InfoPreviewHandler));
```

OR, alternatively:

```
//register javascript client for specific type  
TestWeb25.Components.InfoPreviewHandler info = new InfoPreviewHandler();  
info.RegisterJSClient();
```

Use this code to build javascript call for you:

```
TestWeb25.Components.InfoPreviewHandler info = new InfoPreviewHandler();  
string jsToCall = info.GetJSClientFunctionCall("GetExtraInfo", 356, 577);
```

REST-service framework also provides ability to use Aliases instead of real names for namespaces and classes. In order to use this functionality aliases should be configured in `web.config` (see configuration section later in this document) and the call from above will look like:

```
myInfoProvider.GetExtraInfo(356, 577);
```

where *myInfoProvider* is an alias for *TestWeb25.Components.InfoPreviewHandler* class.

By default REST-service javascript proxy uses HTTP GET method to call the server-side. You may customize the HTTP method by setting the following property prior calling REST-service:

```
webRestExplorer.currentHttpMethod = 'POST';
```

The REST-service proxy will send applicable parameters via form's data or query string depend on the method it uses. Please note that only GET, POST, PUT, and DELETE HTTP methods can be used; using DELETE and PUT HTTP methods may require additional IIS configuration.

### Part 3: Using javascript callback functionality to provide handling for the REST-service result.

Javascript call to the REST-service uses the same technic as AJAX that is asynchronous call to the server with processing returned data within callback function. Therefore there is a javascript callback function will be called upon request completion – the name of the function is the same as original method function call with “Callback” suffix. One parameter is passed in holding the data received from the server.

Use this code to build javascript callback function for you (it provides function name only):

```
TestWeb25.Components.InfoPreviewHandler info = new InfoPreviewHandler();  
string jsCallback = info.GetJSCallbackName("GetExtraInfo");
```

You can also build your own javascript function to use as a callback (this would require pre-registration within web-page that uses such function):

```
//get a callback reference that can be used to deal with returned data.  
WebRestClient.RegisterJSCallback(typeof(InfoPreviewHandler), "GetExtraInfo",  
                                "callMyFunction(data);");
```

where *callMyFunction* is a custom function call that should be used as a callback, and “data” is the default parameter name that will contain data returned by the REST-service method to the client.

There are multiple built-in javascript templates that you can use – they provide basic functionality:

- generic alert – provides client alert using REST-service response as its content, it’s useful for client notifications and debugging purposes:

```
//get a callback reference that can be used to deal with returned data.  
WebRestClient.RegisterJSCallback(typeof(InfoPreviewHandler), "GetExtraInfo",  
                                Web.Enhancements.Rest.Templates.JavaScriptTemplateType.BasicAlert);
```

- populating specific HTML element with inner HTML:

```
//get a callback reference that can be used to deal with returned data.  
WebRestClient.RegisterJSCallback(typeof(InfoPreviewHandler), "GetExtraInfo",  
                                Web.Enhancements.Rest.Templates.JavaScriptTemplateType.PopulateDivWithHtml,  
                                "infoBlock");
```

where “infoBlock” is the name of HTML element where inner HTML content will be set.

### Part 4: Using javascript dynamic callback on the client side.

There are many scenarios where the callback function is not known upfront and often has a dynamic nature. It’s very easy to provide callback dynamically to the call by adding an extra parameter to the javascript REST-function call.

Consider this example that calls REST-service:

```
myInfoProvider.GetExtraInfo(356, 577);
```

Adding an extra parameter would provide a dynamic callback to the service call:

```
myInfoProvider.GetExtraInfo(356, 577, myDynamicCallbackFunction);
```

The “myDynamicCallbackFunction” will be called upon REST-service call completion passing single parameter that holds the data received from the server:

```
function myDynamicCallbackFunction(serverData) {  
    ...serverData parameter holds the data received from the server... }  
}
```

There is no need to register dynamic callbacks on the server-side.

You may pass an unlimited number of parameters into callback function. They will be passed into the callback function following the default “*serverData*” parameter that contains the data returned by the service call. Adding extra parameters would provide a dynamic callback with parameters to the service call:

```
var userId = 123; var username = "Alex";  
myInfoProvider.GetExtraInfo(356, 577, myDynamicCallbackFunction, userId, username);
```

The “*myDynamicCallbackFunction*” with two parameters will be called upon REST-service call completion passing single parameter that holds the data received from the server:

```
function myDynamicCallbackFunction(serverData, userId, username) {  
    ...serverData parameter holds the data received from the server;  
    userId and username holds the past values...  
}
```

You may optionally force REST-service proxy to automatically convert your JSON string data into JavaScript object should JSON data come from the server (based on Content-Type header value). This can simply be done by adding the following code on the client side:

```
$(document).ready(function () {  
    if (typeof (webRestExplorer) != "undefined") {  
        webRestExplorer.jsonSettings.autoParse = true;  
    }  
});
```

The proxy uses *jQuery* to parse the JSON when available; otherwise it uses JavaScript *eval* function. Alternatively you may override the conversion function as following:

```
$(document).ready(function () {  
    if (typeof (webRestExplorer) != "undefined") {  
        webRestExplorer.jsonSettings.parseJson = function (data) {  
            //...convert it here and return  
        };  
    }  
});
```

## Part 5: Creating REST-service.

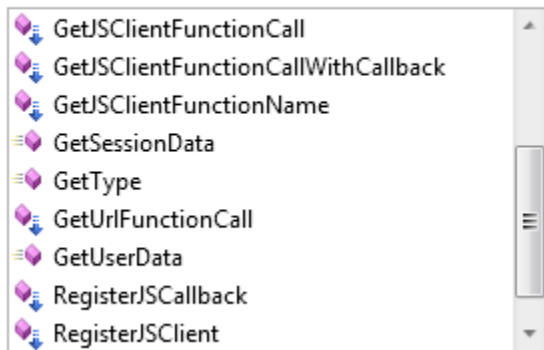
Any server-side class and its methods can be turned into REST-service calls by simply adding *IWebRestService* interface to the class and applying *WebRequestMethodAttribute* to the method itself. The following code provides an example implementing REST-service:

```
using System;
using Web.Enhancements.Rest;
using Web.Enhancements.Rest.Filters;

namespace TestWeb25.Components
{
    public class InfoPreviewHandler : IWebRestService
    {
        [WebRequestMethodAttribute()]
        public string GetExtraInfo(int infoId)
        {
            return string.Format("This is return from InfoPreviewHandler.GetExtraInfo method.\nLooks like you supplied value of {0}.", infoId.ToString());
        }
    }
}
```

The *IWebRestService* requires no methods, properties, etc. to be implemented; it serves as a declaration of intention to use class as a REST-service that framework uses for security purposes, as well as ability to provide additional methods, or helpers, within each class to build javascript calls, callback functions, and register javascript proxy. This, itself, provides greater integration with .NET runtime and development process. The following snapshot demonstrates javascript helpers appear in the REST-service class:

```
InfoPreviewHandler info = new InfoPreviewHandler();
info.|
```



There is some extra configuration can be provided within *WebRequestMethodAttribute* usage:

```
[WebRequestMethodAttribute(SessionMode=System.Web.SessionState.SessionStateBehavior.Required,
    ContentType = "text/html", ContentDisposition = "")]
```

- *SessionMode* – ability to set ASP.NET Session requirement for specific REST-service method;
- *ContentType* – allows to set ContentType header to the response output; by default the output set to “text/html”, for everything else correct ContentType must be set;
- *ContentDisposition* – allows to set ContentDisposition header to the response output; this is useful when file-like data is being sent as an output, so user client can handle it properly.

The REST-service class requires having a default constructor.

No static methods can be used as REST-methods.

No client-side overloading is supported. If .NET class has multiple overloaded methods, only single method among them can be used as REST-service method.

Although the service method parameters are required to be supplied by the client, default values are accepted as shown below and would be used in if service call lacks a parameter:

```
public class InfoPreviewHandler : IWebRestService
{
    [WebRequestMethodAttribute()]
    public string GetExtraInfo(int infoId = 50)
    {
        return string.Format("This is return with value of {0}.", infoId.ToString());
    }
}
```

## Part 6: Configuring REST-services framework.

1. The “rest” file extension must be registered in IIS by adding Handler Mapping between “rest” file extension and ASP.NET runtime (see additional help at [http://technet.microsoft.com/en-us/library/cc771240\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc771240(WS.10).aspx)).

2. The following section in web.config demonstrates REST-services framework configuration:

```
<configuration>
  <configSections>
    <section name="webRestServices" type="Web.Enhancements.Rest.WebRestConfigHandler,
                                              Web.Enhancements" />
  </configSections>
  ...
  ...
  ...
  <webRestServices cacheLocation="ApplicationVariable" restFileExtension="rest">
    <serviceDebug includeExceptionDetailInErrors="true" />
    <assureAssemblies>
      <add assembly="TestWeb25, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    </assureAssemblies>
    <typeMapping>
      <add alias="TestWeb25.Components.InfoPreviewHandler"
           type="TestWeb25.Components.InfoPreviewHandler, TestWeb25,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"/>
      <add alias="backendUserExperience" type="TestWeb25.Components.UserExperienceHandler,
                TestWeb25"/>
      <add alias="Calculator" type="TestWeb25.Components.UserGui.Providers.Calculator,
                TestWeb25"/>
    </typeMapping>
  </webRestServices>
  ...
```

The *webRestServices* configuration section must be added to the *configSections* in web.config as shown above. This will assure REST framework of being able to read its configuration.

The actual REST-services framework configuration section consists of the following nodes:

- *webRestServices* – top-level node for REST-services framework configuration;
- *cacheLocation* – an attribute of *webRestServices* that controls cache location used by REST-services framework; the options are:
  - *ApplicationVariable* – application variable that provides application life-time caching, used as default location;
  - *ApplicationCache* – application cache storage – this location is controlled by application caching mechanism and tends to be less permanent especially in high memory usage web-application scenario; although this may degrade the effectiveness of data caching, it might be required by other application design considerations;
- *restFileExtension* – an optional attribute allows to change the file extension used by REST-service; if this set to other than “rest” (default value), then the same value should be used within IIS Handler mapping and *httpHandler* section in web.config;
- *serviceDebug* – an optional node in the config section that allows configure exception handling within REST-service framework;
- *includeExceptionDetailInErrors* – an optional attribute (required if *serviceDebug* section is configured) that allows to turn off and on the exception details being sent back to a client within REST-service response; by default (“true”) the service will send full exception details back to the client; set this to “false” to disable details and issue 500 server error instead;
- *assureAssemblies* – an optional node that allows to provide list of assemblies where classes used as REST-services are defined; this information is used by REST-service framework to speed up its component discovery process; note that this information is not used if aliases are used, and therefore may be ignored;
  - `<add assembly="[full assembly name]" />` - allows to add an assembly to *assureAssemblies* section;
- *typeMapping* – an optional node that allows to provide a list of aliases to be used instead of real namespace and class names;
  - `<add alias="[class alias]" type="[assembly name]" />` - allows to add an alias to *typeMapping* section;
  - note that it is not permissible to begin alias name from “base.” prefix as this prefix is reserved by REST-service framework for internal purposes;
  - it is also not recommended to begin alias name from well-known prefixes such “sys.”, “Sys.”, etc. as they might be used by Microsoft and other component developers.

3. Add the following *httpHandler* configuration to web.config:

```
<system.web>
  <httpHandlers>
    <add verb="*" path="*.rest" type="Web.Enhancements.Rest.WebRestHandlerFactory,
                                             Web.Enhancements" />
  </httpHandlers>
</system.web>
```

## Part 7: Using REST-service Filters.

The REST-services framework allows building an additional common functionality for REST-services within the application by using Filters. The functional idea of these filters is very similar to Web-services filters and WCF behaviors. Filters can be declared at the method level (applied to this method only), as well as to the REST-service class. The latter would apply filters to all REST-service methods within specified class. There are four built-in filters (see code snapshot below) providing the following functionality:

- *WebMethodFilter* – allows to restrict REST-service method being used by specific HTTP method (the options are: *GetOrPost*, *GetOnly*, *PostOnly*, *DeleteOnly*, *PutOnly*); please note that using DELETE and PUT HTTP methods require additional IIS configuration;
- *RequiresSslFilter* – allows to restrict REST-service method used by secure requests only (via SSL);
- *Authorize* – allows to restrict REST-service method being used by authenticated requests only; in addition to authentication check it allows to set Roles and/or Users who can use the service method;
- *ValidateRequest* - allows to force request validation before processing REST-service method (standard ASP.NET request validation procedure will be applied);

```
using Web.Enhancements.Rest.Filters;
...
[WebMethodFilter(HttpMethod = HttpMethodRequirement.GetOrPost, Priority = 2)]
[RequiresSslFilter(Priority = 1)]
[Authorize(Roles = "Admin", Users = "")]
[ValidateRequest(Priority = 3)]
[WebRequestMethodAttribute()]
public string GetExtraInfo(int infoId)
{
    return string.Format("...", infoId.ToString());
}
```

Providing an optional *Priority* parameter to filters guarantees the order in which filters would be applied to incoming request.

Developers can build their own filters by creating custom *FilterAttribute* (requires either implementation of *Web.Enhancements.Rest.Filters.IWebRestFilterAttribute* interface or inheritance from *Web.Enhancements.Rest.Filters.WebRestFilterAttribute*) and actual filter (requires implementation of *Web.Enhancements.Rest.Filters.IWebRestFilter* with three required methods – *BeforeHandlerCreated*, *BeforeMethodCall* and *AfterMethodCall* as shown below). Custom filters can provide any additional functionalities that web-application may require and perfectly suitable for generic functionality applicable to REST-services (such as security, compression, etc.).

```
namespace Web.Enhancements.Rest.Filters
{
    /// <summary>
    /// Interface that any Web REST filter must implement in order of being processed
    /// through REST service pipeline.
    /// </summary>
    public interface IWebRestFilter
    {
        /// <summary>
        /// Method that is called prior the REST service handler created.
        /// This allows to cancel the call at earlier stages.
        /// The session and list parameters are not available at this time.
    }
}
```



```

    /// This method should be used wisely to consume as low as possible resources,
    /// and especially fast to allow IIS to have overall better performance.
    /// <param name="httpContext">Current HTTP content.</param>
    /// <param name="serviceContext">Current REST service context.</param>
    void BeforeHandlerCreated(System.Web.HttpContext httpContext,
                             IWebRestServiceContext serviceContext);

    /// <summary>
    /// Method that is called just prior the REST service method call.
    /// </summary>
    /// <param name="httpContext">Current HTTP content.</param>
    /// <param name="serviceContext">Current REST service context.</param>
    void BeforeMethodCall(System.Web.HttpContext httpContext,
                          IWebRestServiceContext serviceContext);

    /// <summary>
    /// Method that is called just after the REST service method is completed.
    /// </summary>
    /// <param name="context">Current HTTP context.</param>
    /// <param name="returnValue">Value is going to be returned by service.</param>
    void AfterMethodCall(System.Web.HttpContext context, ref object returnValue);
}

```

## Part 8: Using HTML Templates.

The same REST-service framework provides “out of the box” HTML templates capability that is built on top of existing custom Web User Controls (ASCX). The functionality performs loading control dynamically within HTTP request context and optionally applying dynamic object as its data source.

To create an HTML template an *ascx* file should be created and control must to be inherited from *Web.Enhancements.Rest.Templates.HtmlTemplate<TDataSource>*. The following code snapshot represents code-behind file:

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using Web.Enhancements.Rest.Templates;

namespace TestWeb25
{
    public partial class TestControl : HtmlTemplate<string>
    {
        public TestControl()
            : base()
        {
        }
    }
}
```

The class above uses type of string objects as its data source type; of cause any object type can be used as data source and it's entirely up to developer to decide. Using *Generics* for data source leads to development flexibility and comfort within template creation process (IntelliSense). It allows to create a custom data source as template model providing great flexibility to the template feature. The data source is accessible via *DataSource* property and, as expected, has a type of *Generic* object declared as *TDataSource* at control inheritance declaration. It is also possible to pass additional data into template processing using built-in dictionary available via *TempData* property.

Based on the class above the following ASCX template is created:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="TestControl.ascx.cs"
Inherits="TestWeb25.TestControl" %>

<span style="color:Red">Hello there, <%= this.DataSource %>.</span>
```

To render the control, the following code should be used:

```
using Web.Enhancements.Rest.Templates;
...
return HtmlTemplateFactory.RenderTemplate<string>("TestControl.ascx", userName);
```

OR, if additional data (other than *TDataSource* model) is required to be passed into template:

```
Dictionary<string, object> extra = new Dictionary<string, object>();
extra.Add("colorToUse", "Gray");
return HtmlTemplateFactory.RenderTemplate<string>("TestControl.ascx", userName, extra);
```

The *HtmlTemplateFactory* has multiple methods suitable for different scenarios within the application.

Of course, the HTML Templates can be used to build Javascript response and therefore can be served as Javascript Templates.

## Part 9: Using HTML Templates with REST-services.

Although it is not recommended, the straight-forward scenario can be used within REST-service method to provide response using HTML Templates: build an HTML string from template and return it as a return value from REST-service method to a client. Unfortunately it creates inefficient code where every method would have HTML parsing call and such methods are very hard to test as they return HTML strings values. Luckily, there is another, more flexible alternative. The REST-service framework has the *Web.Enhancements.Rest.Templates.TemplatedResult<TDataSource>* object that is designed to be used as a return parameter from REST-service method if result is intended to be processed using HTML templates:

```
using Web.Enhancements.Rest.Templates;
...
[WebRequestMethodAttribute()]
public TemplatedResult<string> GetGreeting2(string userName)
{
    TemplatedResult<string> result = new TemplatedResult<string>("TestControl.ascx",
                                                                userName);
    return result;
}
```

OR, if additional data (other than *TDataSource* model) is required to be passed into template:

```
using Web.Enhancements.Rest.Templates;
...
[WebRequestMethodAttribute()]
public TemplatedResult<string> GetGreeting2(string userName)
{
    TemplatedResult<string> result = new TemplatedResult<string>("TestControl.ascx",
                                                                userName);
    result.TempData.Add("colorToUse", "Gray");
    return result;
}
```

The REST-service handler will process response accordingly by using the virtual path to the template's ASCX file location and data model provided as a method return. In addition, using such solution for building REST-service provides perfect conditions for unit testing the REST-services components.

It is not required to declare the return type as *TemplatedResult<TDataSource>*. It is acceptable to declare return type as an *object* and decide what to return at the runtime. The REST-service handler will interpret what is actually returned dynamically upon receiving return value and process result accordingly.

## Part 10: Returning file as a response from REST-service.

Although the REST-service method can return an array of bytes that will be sent to a client, there is an alternative, and much more efficient way to provide a static file type response. This is to use specially built Template called *StaticFileResult*. Use the sample code below:

```
using Web.Enhancements.Rest.Templates;
...
[WebRequestMethodAttribute(ContentType="application/pdf")]
public StaticFileResult GetHelpFile(int roleId)
{
    StaticFileResult result = new StaticFileResult()
    {
        FilePath = if ((roleId == 1) ? "files/Admin.pdf" : "files/User.pdf"),
        ContentDisposition = "filename=HelpFile.pdf"
    };
    return result;
}
```

The service will automatically write file's content into the output stream that saves resources and speeds up the response.

It is not required to declare the return type as *StaticFileResult*. It is acceptable to declare return type as an *object* and decide what to return at the runtime. The REST-service handler will interpret what is actually returned dynamically upon receiving return value and process result accordingly. For this purpose the *StaticFileResult* class allows to pass *ContentType* and *ContentDisposition* along with *FilePath* to avoid conflicts with other data that might be returned. The *ContentType* and *ContentDisposition* settings in the *StaticFileResult* will override the same settings in *WebRequestMethodAttribute*.

## Part 11: Returning JSON data as a response from REST-service.

Returning JSON formatted data from REST-service is as easy as it possibly can be. The REST-service framework will automatically convert an object of any type returned by REST-service method and return JSON-formatted data within its response as long as the *ContentType* contains "json" as part of its value. Consider the following code as it demonstrates the REST-service method returning JSON as a response:

```
[WebRequestMethodAttribute(ContentType = "application/json")]
public Employee RetrieveEmployeeInfo(int userId)
{
    Employee emp = new Employee()
    {
        FirstName = "Mark",
        LastName = "Jason",
        UserName = "Jasonman",
        UserId = userId
    };
    return emp;
}
```

You may use any *ContentType* within your application for returning JSON-formatted data, but the word "json" must be present. Be aware that failing to provide *ContentType* will cause *object.ToString()*

method call result appear in the response, therefore the following sample result's response would produce "TestWeb25.Components.Employee". There is also an assumption that if *System.String* data returned by the REST-service method then, even *ContentType* set to a "json"-like string, the result already contains JSON data created within the method itself, therefore the JSON serialization would not be performed by REST-service framework.

It is possible to control the serializer used by framework to convert object to JSON data. There are 2 types can be used: *System.Web.Script.Serialization.JavaScriptSerializer* (used by default) and *System.Runtime.Serialization.Json.DataContractJsonSerializer*. The *DataContractJsonSerializer* can be set as shown below:

```
[WebRequestMethodAttribute(ContentType = "application/json,  
                             JsonSerializer = JsonSerializerType.DataContractJsonSerializer")]
```

NOTE: for non-JSON REST-method returns the *JsonSeriallizer* parameter in *WebRequestMethodAttribute* constructor is ignored.

## Part 12: Returning other data types as a response from REST-service.

This the full set of rules that REST-service framework uses to process the output data returned from method:

- If data type of *System.String* is returned by the method, it's written directly into the output response;
- If data type *System.Byte[]* (array only) is returned by the method, it's written directly into the output stream regardless of the *ContentType* setting;
- If data type returned by the method implements *ITemplatedResult* interface, the framework processes data through template and writes result into the response stream;
- If data type returned by the method implements *IStaticFileResult* interface, the framework writes static file content directly into the response stream;
- If any other data type is returned by the method, the following rules are applied:
  - If *ContentType* contains "json" word, the framework serializes data into JSON and writes result into the output stream;
  - For every other scenario method *Object.ToString()* is used to gather result and it's written into the output stream.

### Part 13: Mapping HTTP Message Body to a complex type parameter.

The RESTful services require ability to read and map data of the entire HTTP Message Body to a single service method parameter. This is especially useful when REST-style service URLs with *POST* or *PUT* HTTP method is used. The *HttpBodyParameter* attribute provides necessary information to the framework on reading, mapping, and converting data to a parameter specified type. The *ParameterName* is required and links the HTTP Message Body to the parameter. The *ConversionHint* is an option, but desired parameter that helps the framework to properly identify and convert received data to a parameter declared type. If complex type is used, such type must be serializable in order being properly converted. Below are two examples of using this feature:

```
[WebRequestMethodAttribute()]
[HttpBodyParameter(ParameterName= "user", ConversionHint = ParameterConversionHint.Json)]
public string RetrieveData(int userId, Employee user, string extraData)
{
    string outData = "";
    //Get data
    return outData;
}

[WebRequestMethodAttribute()]
[HttpBodyParameter(ParameterName = "incomingData")]
public string ResendData(string incomingData)
{
    return incomingData;
}
```

The *ConversionHint* can take the following values:

- *Text* - the data is treated as plain text; in this case the parameter type should be *string*.
- *Json* - the JSON deserialization is enforced; assure the parameter type is JSON compatible.
- *JsonContract* - the JSON deserialization using data contract is enforced; assure the parameter type implements *DataContract*.
- *Xml* - the XML deserialization is enforced; assure the parameter type is XML compatible.
- *Binary* - the data is treated as binary. In this case the parameter type should be a *byte* array.
- *Default* – an internal logic is used for conversion; framework uses this value when *ConversionHint* is omitted from the declaration:
  - When parameter type is *string* or *byte[]*, the framework simply reads the data from the *InputStream* and passes it as parameter value;
  - For other basic types (*int*, *byte*, etc.) framework uses the same conversion mechanism as for *ContentType* parameters mapped to *Form* or *QueryString* parameter;
  - If for *InputStream* indicates the JSON data (like “*application/json*”), then *Json* deserialization is used;
  - If *ContentType* for *InputStream* indicates the XML data (like “*text/xml*”), then *Xml* deserialization is used;
  - Failure to find the data parser will resolve into an *InvalidOperationException*.
- *XmlOrJson* - the JSON or XML deserialization is enforced based on *ContentType* value; the XML takes precedence if content type has no explicit information.
- *JsonOrXml* - the JSON or XML deserialization is enforced based on *ContentType* value; the JSON takes precedence if content type has no explicit information.

- *XmlOrJsonContract* - the XML or JSON using data contract deserialization is enforced based on content type setting; the XML takes precedence if content type has no explicit information.
- *JsonContractOrXml* - the XML or JSON using data contract deserialization is enforced based on content type setting; the JSON takes precedence if content type has no explicit information.
- *FormOrQueryVars* - the Form or Query String variables will be used to set properties within complex data types. The *PropertyWebBindingAttribute* can be used to set the mapping between form variable names and properties.

Please compare two code snippets below demonstrating the advantages that can be achieved using the *FormOrQueryVars* hint:

```
[WebRequestMethodAttribute()]
[HttpBodyParameter(ParameterName = "incomingData")]
public string ResendData(int userId, string incomingData, string extraData)
{
    if (userId <= 0)
        throw new Exception("UserId is incorrect");
    if (extraData.Length == 0)
        throw new Exception("extraData is incorrect");

    return incomingData;
}

[WebRequestMethodAttribute()]
[HttpBodyParameter(ParameterName = "data",
    ConversionHint = ParameterConversionHint.FormOrQueryVars)]
public string ResendData2(DataExchangeInfo data)
{
    if (data.UserId <= 0)
        throw new Exception("UserId is incorrect");
    if (data.ExtraData.Length == 0)
        throw new Exception("extraData is incorrect");

    return data.DataToSendOut;
}

Where:
public class DataExchangeInfo
{
    public int UserId { get; set; }
    [PropertyWebBinding(MappingName = "incomingData")]
    public string DataToSendOut { get; set; }
    [PropertyWebBinding(MappingName = "extraData")]
    public string ExtraData { get; set; }

    public DataExchangeInfo()
    {
    }
}
```

Below is an example of HTTP Message and code that is designed to handle it. The most important parts of message are emphasized in red. This code also includes the REST call mapped to specific method via ASP.NET Routing as:

"Users" = "www.test.com/Users" with default method *CreateNewUser* for PUT HTTP method.

#### The request:

```
PUT http://www.somesite.com/Users HTTP/1.1
Accept: */*
Content-Type: application/json
Referer: http://www.somesite.com/Default.aspx
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; windows NT 6.1; WOW64; Trident/5.0)
Host: www.somesite.com
Content-Length: 77
Connection: Keep-Alive
Pragma: no-cache

{
  "UserId":0,
  "LastName":"Jason",
  "FirstName":"Mark",
  "UserName":"Jasonman"
}
```

#### The code:

```
[WebRequestMethodAttribute()]
[WebMethodFilter(HttpMethodRequirement.PutOnly)]
[DefaultHttpPut()]
[HttpBodyParameter(ParameterName="empData",ConversionHint= ParameterConversionHint.Json)]
public int CreateNewUser(Employee empData)
{
    //...saves data
    return empData.UserId;
}
```



## Part 14: Client side custom error handling.

The client-side REST-services proxy supports custom error handling. By default this functionality is turned off. Since the REST-services are intended to operate with any type of data returned, the error handling is based on parsing the data coming from the server assuming the string data type. Providing the keyword that error message starts with automatically turns on the error handling. The keyword should match what your custom error message generated by exception handling procedure on the server side; this assures independent error messaging between web-site and user when REST services are used. The default error handling uses javascript “alert” to notify user about the error. If custom client side handling is desired, then custom javascript method delegate must be assigned to the javascript REST-service error handler. Both options are shown below.

REST-service proxy accepts two settings via javascript:

```
<script type="text/javascript">
  if (typeof (webRestExplorer) != "undefined") {
    webRestExplorer.errorMessageStartsWith = "ErrorInfo:";
    webRestExplorer.errorHandler = myHandler;
  }
</script>
```

- The “errorMessageStartsWith” – a keyword that error message would start with; REST-service proxy will handle an error if incoming message from the server begins with given keyword (string);
- The “errorHandler” – you can assign a custom function that should handle the error; this overrides the default error message handler.

It is important to note that this javascript must follow the REST-service proxy declaration; therefore it should follow any request for proxy registration (see Part 2 for details).

## Part 15: Client side automatic data parsing from JSON to an object.

In some scenarios it is useful to enforce automatic parsing for JSON-formatted data into a javascript object that would be passed into callback function. The conversion requires response *ContentType* indicates json-type data coming to the client. The following code would turn on conversion:

```
webRestExplorer.jsonSettings.autoParse = true;
```

The default value is “false”. By default browser native or jQuery, if available, JSON parser is used to convert the data. You may also provide custom JSON parser as shown below:

```
webRestExplorer.jsonSettings.parseJson = customParserFunctionCall;
```

## Part 16: Using RestClientManager control.

The RestClientManager control is included as part of REST-services solution. It allows registering javascript client proxy classes generated for REST-service enabled .NET classes, custom script files references, and configuring custom error handling. It provides an alternative to a code-driven registration process.

Consider this sample for functionality evaluation:

```
<rest:RestClientManager ID="ScriptManager1" runat="server">
  <Classes>
    <rest:ClassReference FullClassName="TestWeb25.Components.InfoPreviewHandler"
      Assembly="TestWeb25" />
  </Classes>
  <Scripts>
    <rest:ScriptReference Path="../Includes/CustomHandlers.js" />
    <rest:ScriptReference Path="../Includes/MainJavaScriptFunction.js" />
  </Scripts>
  <ErrorHandling ErrorMessageStartsWith="ErrorInfo:" ClientDelegateName="myHandler" />
</rest:RestClientManager>
```

This is very convenient way to make sure availability of all REST-service components and javascript functions that used by the page. The manager itself assures that all javascript functions that might be used by REST-service callbacks are declared at the right moment and have correct scope.

## Part 17: Using ASP.NET Output Cache with REST-services.

REST-service supports ASP.NET Output Cache. An appropriate attribute *WebRestCache* should be added to the REST-service method:

```
[WebRequestMethodAttribute(ContentDisposition = "filename=HelpFile.pdf")]
[WebRestCache(Duration = 20, VaryByParam = "All", Location = OutputCacheLocation.Server)]
public string GetInfo(string documentName)
```

- The “Duration” – a cache duration (in seconds);
- The “VaryByParam” – comma or semi-colon delimited list of parameter names that cache should be differentiated by; this parameter also supports “None” and “All”;
- The “Location” – location of the cache (Server is default choice).

**IMPORTANT NOTE:** The REST-service mapper module should be integrated with application pipeline in order for REST-services being compatible with ASP.NET Output Cache key-generation component. This can be easily achieved by setting the following declaration in application web.config:

```
<httpModules>
  <add name="WebRestMapperModule" type="Web.Enhancements.Rest.WebRestMapperModule,
    Web.Enhancements" />
</httpModules>
```

## Part 18: ASP.NET Routing support and CRUD (Create, Read, Update, and Delete) compatibility.

Beginning from v1.1 REST-services Framework is compatible with ASP.NET Routing model as well with CRUD (Create, Read, Update, and Delete) principle. These two are often important when building REST API functionality within your application.

Below is the example of configuring ASP.NET Routing table for REST-services, where:

- The “*ns0*”, “*ns1*” – the namespace placeholders where the REST-service class is located, required unless an **ALIAS** for specific service class is used (see Part 6: Configuring REST-services framework):
  - (up to 50 namespace parts supported);
  - example: the namespace *Application.Standards.Services* represented by 3 parts would be described in the route as */{{ns0}}/{{ns1}}/{{ns2}}*;
- The “*class*” – the name of the REST-service class, required as **Class Name** or **ALIAS** if used;
- The “*method*” – the name of the service method, not required if default mapping to HTTP Method is intended;
- The “*infoId*” – developer-defined parameter for the method (no limit on number of parameters).

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.Add("Test1", new Route("api/{ns0}/{ns1}/{class}/{method}/{infoId}",
        new WebRestRouteHandler()));

    RouteTable.Routes.Add("Test2", new Route("api/{ns0}/{ns1}/{class}/{infoId}",
        new WebRestRouteHandler()));
}
```

The first route “*Test1*” would handle requests explicitly declaring the REST method name. The second route “*Test2*” would be able to handle requests without specifying the REST method; in this case the method would be chosen based on HTTP Method used to send a request. Both routes can work side-by-side within the application.

**IMPORTANT:** it is beneficial to build more code-specific routes to avoid too wide match across different parts of the application, and its components. The above route “*Test1*”, although the actual URL request being the same (*<rooturl>/api/TestWeb25/Components/InfoHandler/2*), can be build specifying the actual namespace parts and setting them as default values for corresponded placeholders, would work better because another class with the same name structure and parameter names can be used within the same application allowing more flexibility. Consider this sample:

```
RouteValueDictionary routeDict = new RouteValueDictionary();
routeDict.Add("ns0", "TestWeb25");
routeDict.Add("ns1", "Components");
RouteTable.Routes.Add("Test2", new Route("api/TestWeb25/Components/{class}/{infoId}",
    routeDict, new WebRestRouteHandler()));
```

or, even stricter:

```
RouteValueDictionary routeDict = new RouteValueDictionary();
routeDict.Add("ns0", "TestWeb25");
routeDict.Add("ns1", "Components");
routeDict.Add("class", "InfoHandler");
RouteTable.Routes.Add("Test2", new Route("api/TestWeb25/Components/InfoHandler/{infoId}",
    routeDict, new WebRestRouteHandler()));
```

It is not required to set each namespace part as separate default parameter. You may ignore part of namespace in URL, or entire namespace, or use the names that appropriate to your application. In this case you should use a reserved keyword “*namespace*” to describe the full “*namespace*” parameter for the REST-service runtime being translated to the calling target. These are 2 examples of such:

```
RouteValueDictionary routeDict = new RouteValueDictionary();
routeDict.Add("namespace", "TestWeb25.Components");
RouteTable.Routes.Add("Test2", new Route("api/TestWeb25/Components/{class}/{infoId}",
    routeDict, new WebRestRouteHandler()));

RouteValueDictionary routeDict = new RouteValueDictionary();
routeDict.Add("namespace", "TestWeb25.Components");
RouteTable.Routes.Add("Test2", new Route("api/Test/Favorites/{class}/{infoId}",
    routeDict, new WebRestRouteHandler()));
```

Since the REST-service Framework does not place any restrictions on method names, a special attribute is used to map a REST-service method name to a specific HTTP Method (VERB), and let REST-service runtime ability to resolve those correctly. The following is the sample code showing how the *GetExtraInfo* method is mapped to the GET HTTP Method using *DefaultHttpGet* attribute:

```
[WebRequestMethodAttribute()]
[DefaultHttpGet()]
public string GetExtraInfo(int infoId)
{
    return string.Format("This is return from GetExtraInfo method.");
}
```

REST-Service Framework has predefined attributes for the following HTTP Methods: GET, POST, DELETE, PUT, SEARCH, COPY, MOVE, LOCK, UNLOCK, OPTIONS. You may use the *DefaultHttpMethodAttribute* to set another HTTP Method that you need to use.

Although any number of methods in the REST-service class can be mapped to HTTP Methods, single HTTP Method (VERB) should be mapped to a single method only. This would avoid ambiguous resolution of request by REST Runtime. The runtime would not produce an exception, but rather would use the first method with desired mapping attribute.

The route registration via attribute declaration is also supported (since v1.5.2). Route can be declared via special attribute at class or method level. There is no limitation on the number of routes declared at any level.

```
[WebRequestMethodAttribute()]
[RegisterRoute(Route = "api/messages/{id}", BindToMethod = "Delete", LimitByHttpMethods = "DELETE")]
[RegisterRoute(Route = "api/messages/{id}", BindToMethod = "Update", LimitByHttpMethods = "POST,PUT")]
public class MyDataService : IWebRestService
{
    ...
}
```

Where:

- The “*Route*” – the route string declaration;
- The “*BindToMethod*” – this optional parameters allows to bind the route to a class method when attribute declared at class level;

- The “*LimitByHttpMethods*” – an optional comma-delimited list of HTTP methods that route should be limited to;

It is important to note when class is inherited from another class, the route attributes at class level are not inheritable; the route attributes at a method level are inheritable unless method is overridden. This rule avoids routing conflicts that can be introduced by inheritance.

In order to prevent undesired route registration that might have an adverse effect on the application, especially in scenarios whether code is reused between the applications, the REST-service Framework uses explicit registration technique. Use the *RegisterTypeRoutes* method at the *WebRestClient* class to register all routes declared within particular class. Method should be called once for every class within an application lifetime. Below is an example that registers all routes declared within scope of *MyDataService* class:

```
Web.Enhancements.Rest.WebRestClient.RegisterTypeRoutes(typeof(MyDataService));
```

It is recommended to use this method within *Application's Start* event.

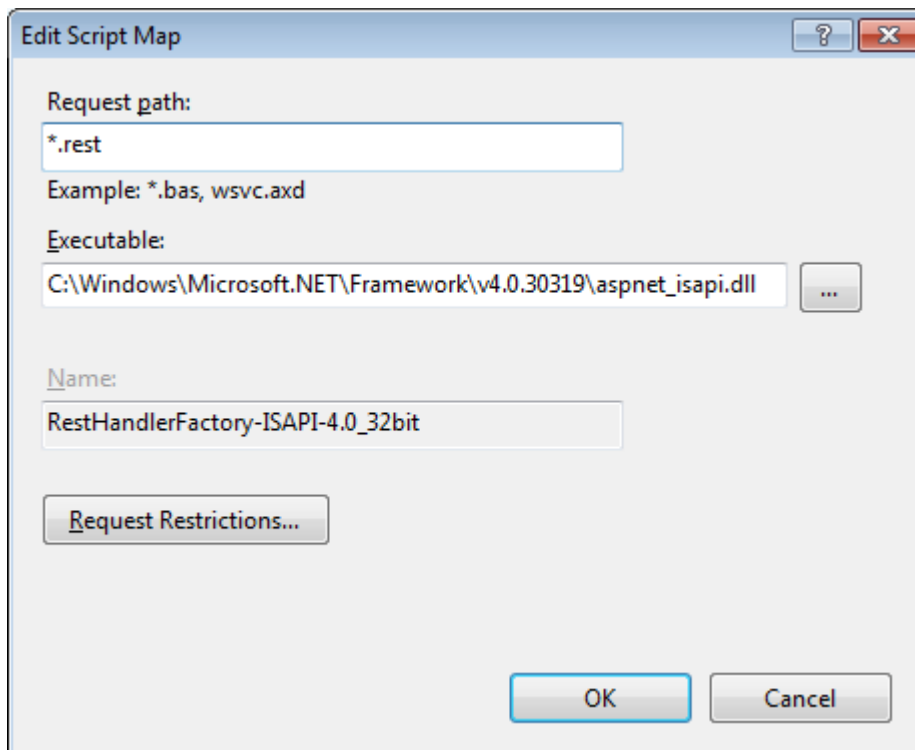
## Part 19: Adding handler mapping for REST extension to IIS.

Using service requires additional configuration to the IIS - the Handler Mapping should be added for “\*.rest” extension and ASP.NET 4 runtime. Depending on server, either 32 bit or 64 bit, or both should be added.

To setup the handler mapping, the steps are:

- Open IIS Manager
- Select IIS ROOT level
- Select “Handler Mappings”
- On the right menu panel click “Add Script Map”
- Add script map as:

### 32 bit runtime:



### 64 bit runtime:

Executable: c:\Windows\Microsoft.NET\Framework64\v4.0.30319\aspnet\_isapi.dll

Name: RestHandlerFactory-ISAPI-4.0\_64bit

## Part 20: REST-service Metadata Explorer.

As the number of services growing over time, it becomes essential ability exploring the existing services. REST-services Framework allows performing this task via Metadata Explorer. By default the feature is turned off for security considerations. You can quickly enable it by setting the configuration parameter in web.config as shown below:

```
<webRestServices cacheLocation="ApplicationVariable" restFileExtension="rest"
  allowExploreMetadata="true">
```

- *allowExploreMetadata* – turns on and off Metadata Explorer; optional parameter, by default is “false”;

The following URL will invoke the Metadata Explorer for specified assembly:

```
<root>/base.metadata.rest?assemblyName=TestWeb25
```

Once invoked, the Metadata Explorer would analyze the assembly and find all REST services within the assembly creating the linked list:

Assembly: TestWeb25, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

- [TestWeb25.Components.InfoPreviewHandler](#)
- [TestWeb25.Components.UserExperienceHandler](#)
- [TestWeb25.Components.UserGui.Providers.Calculator](#)

Clicking on specific service (class) would invoke another call to the Metadata Explorer that provides the details for each service methods:

Class: TestWeb25.Components.InfoPreviewHandler

Version: '1.0.0.0'

Filter: Web.Enhancements.Rest.Filters.RequiresSslFilterAttribute

- TestWeb25.Components.InfoPreviewHandler.GetExtralInfo(Int32 infold)  
URL: <root>/TestWeb25/Components/InfoPreviewHandler.rest?GetExtralInfo&infold=...  
Non-explicit method URL (GET only): <root>/TestWeb25/Components/InfoPreviewHandler.rest?infold=...  
Mapped to HTTP verb: GET  
Content Type: text/html  
Return Object Type: System.String  
Filter: Web.Enhancements.Rest.Filters.WebMethodFilterAttribute: GetOrPost  
Filter: Web.Enhancements.Rest.Filters.RequiresSslFilterAttribute
- TestWeb25.Components.InfoPreviewHandler.GetStaticFile(String documentName)  
URL: <root>/TestWeb25/Components/InfoPreviewHandler.rest?GetStaticFile&documentName=...  
Content Type: text/html  
Return Object Type: Web.Enhancements.Rest.Templates.StaticFileResult  
Cache: Duration: 20 sec; Location: Server; VaryByParam: All.

Metadata Explorer supports *System.ComponentModel.Description* attribute to provide description to the service or its methods.