RESTful way of data transfer

Article
Istvan Hahn · Oct 5, 2016 13m read

RESTful way of data transfer

This article gives a brief introduction how a RESTful service consumer and a RESTful service provider exchange data. It is a beginner's guide. Data is transferred from a consumer to a provider as parameters of the service. Parameters are part of a service request. The result of the service action a response is returned from a provider to a consumer. Both the service request and response are standard HTTP messages. Since HTTP is a flexible standard regarding to the message contents, RESTful services also enjoy the versatility of data transfer methods.

During this session you can learn what options you have, and how to implement those options both at consumer and provider side using Ensemble.

Before you read

You need to have

- Two Ensemble productions running Ensemble 2016.2 or later version.
- One production configured as a server
- One production configured as a client
- Unrestricted access from the client to the server
- The example productions built during the previous sessions deployed to the server and the client

You need to know

- How to provide RESTful Web Service
- How to consume RESTful Web Service
- Ensemble COS programming

Service parameters

A good application is designed to support (not only but at least) the following two major goals. Entirely match the end user requirements (functionality, user experience, quality, TCO etc.) and deliver it at the lowest possible cost. To reach the second, the development is trying to (re-)use everything already available. When Service Oriented Architecture was introduced, reusability was one of the strongest argument. Reusability means that a software piece can be used in various scenarios even in other projects than it was developed for. It is nothing different in REST. A (RESTful) service therefor have to be as generic as possible in order to be as reusable as it can be. On the other hand to match the requirement of all particular use cases needs flexibility. Thus a service have to understand the context in which it is operating. In case of RESTful services parameters are providing the context for a service run.

The parameters are somehow packaged into a single communication unit (file, mail, HL7 message, SOAP message, etc.). REST is tied to HTTP, therefor RESTful services broadly use HTTP request as the unit of communication. There are several ways to put information fragments in to a single HTTP request.

- The HTTP action/ method/ verb. The well known GET, POST, PUT, DELETE (etc.).

- The URL path. It is the most “RESTish” way. The URL holds the path to a resource. The first part of URL is the service root followed by some parameter.
- The URL query. When we add key, value pairs to the URL after a question mark (?). It is the orthodox HTTP way. But works for REST too.
- Form Data. When a service takes large amount of input parameters, for sure form data is a good option. It has no limitation to the size of the parameters.
- HTTP Message Body. When a service requires complex data structure as a parameter message body is the only place to store. The contents is some serialized form of an object. Mostly the “lowest common multiple” from the serialization methods supported by various platforms is JSON or ECMA 404. The other is XML but XML tied more to SOAP.

There are also other options which are popular in the HTTP world but rarely used (or not at all) in REST. Those basically hurt principals of REST.

- Cookies. Kind of a context variable shared by clients and servers. But REST by nature is stateless. So it is better to keep your cookies in a jar.
- Session. Server side structure to maintain transaction state at server side. Shoo!

In the “Implementation” section you can find “how-to” examples for each parameter type.

Service response

A service supposed to serve requests. A request supposed to be answered. The answer is packed to a single communication unit – like the request. For RESTful services the HTTP response message is the most common unit as those are on top of HTTP.

HTTP response does not have the versatility of placing data as HTTP request, but still have enough to talk about.

- HTTP Status. It is the most important and first to check piece of the response. This tells a service consumer if a request has been successfully processed by the called service provider or not. If not, then what the reason was. Those are the famous 200 .. 500 error codes. Since the status code is generic HTTP, it does not necessarily comes from a RESTful server. It could report error condition of a HTTP server relaying between network zones, malfunction of a message router as well as badly formed request.
- HTTP Message Body. The contents returned by the server. Expectedly it is an object in some serialized format. Mostly JSON serialized. But be careful. REST uses HTTP. Any component in the whole communication stack can return an error code. Like 404 – Resource does not exists. Because servers are not able to distinguish between requests if they are generated by human or robot, therefor the response is intended to be read by human or robot, for the sake of simplicity those are usually responding in HTML format. Any attempt to de-serialize to an object fails. Even if the server is responding in JSON format, sometimes it returns an exception object. The exception has to be recognized by the client, and perform the deserialization of the message body accordingly. HTTP Status supplemented by “ContentType” header field therefore must be evaluated prior interpreting the contents in order to do it right.

You can find “how-to” examples for processing response later in the guide both for server and client side.

Implementation

These is the “Most Wanted” section. Examples for beginners. As I mostly do, no error handling, no validation. Just the bare program logic.

Every example is for Ensemble. Both the client and the server. If you are about to use other client or server platform, please refer to it’s documentation.

Service parameters

The examples are based on the Consuming RESTful Web Services example. We are going to modify the request message as well as the operation. If you would like to keep the original example in working condition, make a backup.

Each example is working with the same modified request message. Please add the following properties to the message class.

- returnRecepient – to whom the echo message is sent
- message – the message text echoed

After the modification the request class looks like this.

```
Class h2.consumerestfulservice.Request Extends Ens.Request
{

Property accept As %Dictionary.CacheClassname;

Property returnRecepient As %String;

Property message As %String;

}
```

There is no need to change anything in the response class. It had no structure anyhow.

URL path

In the example the URL path is holding the parameters. The task is relatively straightforward. The request URL generation is needed to be changed in the client Business Operation as well as in the server Business Service.

There is one “strange” thing in the code. You might recognize the $translate function call. When the URL syntax has been defined and standardized it was not the primary goal to support non-printable, white space or Unicode characters. Thus when for some odd reason someone wants to put such a character into an URL she or he must use URL escaping. The URL escaping replaces for example the space character by a %20. It is fine for the query part of an URL. But in the path it is just inappropriate. The path of an URL ends by a space. Therefor the path must not contain space (while the query might). In certain cases, when a service parameter can have space character inside (for example a name field of a search criteria), the space must be replaced with a “tolerable” character. The browsers are using the hyphen (-). The example does the same. So the $translate function replaces the space in a path parameter to hyphen and back.

The client side Business Operation.

```
Class h2.consumerestfulservice.Operation Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Property Adapter As EnsLib.HTTP.OutboundAdapter;
```

```
Parameter INVOCATION = "Queue";


Method ProcessRequest(pRequest As h2.consumerestfulservice.Request,
                      Output pResponse As h2.consumerestfulservice.Response) As %Stat
us
{
  set st = $$$OK
  set responseName = pRequest.accept
  set returnRecepient = $translate(pRequest.returnRecepient," ","-")
  set message = $translate(pRequest.message," ","-")
  set % = "/"
  set url =..Adapter.URL_%_"echo"_%_returnRecepient_%_message
  set st = ..Adapter.GetURL(url, .callResponse)
  #dim callResponse as %Net.HttpResponse
  set dynamicObject = {}.%FromJSON(callResponse.Data)
  set pResponse = $classmethod(responseName,"%New",dynamicObject)
  Quit st
}

XData MessageMap
{
<MapItems>
  <MapItem MessageType="h2.consumerestfulservice.Request">
    <Method>ProcessRequest</Method>
  </MapItem>
</MapItems>
}

}
```

The server side Business Service

```
Class h2.createrestfulservice.Service Extends Ens.BusinessService
{

Parameter ADAPTER;

Method OnProcessInput(pInput As %Stream.Object, Output pOutput As
 %Stream.Object) As %Status
{
       set pOutput = {}
       set pOutput.returnTo = $translate(pInput(1),"-"," ")
       set pOutput.message = $translate(pInput(2),"-"," ")
       Quit $$$OK
}

}
```

Please note, that pInput must be subclass of %Stream.Object. pInput is a vector of positional parameters. The positional parameter passing applies only to URL path parameters.

URL query

Unlike the URL path parameter, a query parameter is part of the query. A query is constructed of key, value pairs. This type of parameters are handled very easily. It has only one trade off. The EnsLib.HTTP.OutboundAdapter has only “internal” method to call with parameter array. In other words, it is not 100% future proof.

```
Class h2.consumerestfulservice.Operation Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Property Adapter As EnsLib.HTTP.OutboundAdapter;

Parameter INVOCATION = "Queue";

Method ProcessRequest(pRequest As
 h2.consumerestfulservice.Request, Output pResponse As
 h2.consumerestfulservice.Response) As %Status
{
	set st = $$$OK
	set httpRequest = ##class(%Net.HttpRequest).%New()
	do httpRequest.InsertParam("returnRecepient",pRequest.returnRecepient)
	do httpRequest.InsertParam("message",pRequest.message)
	set url = ..Adapter.URL_"/echo"
	set st = ..Adapter.SendFormDataArray(.callResponse,"GET",httpRequest,,,url)
	#dim callResponse as %Net.HttpResponse
	set dynamicObject = {}.%FromJSON(callResponse.Data)
	set responseName = pRequest.accept
	set pResponse = $classmethod(responseName,"%New",dynamicObject)
	Quit st
}

XData MessageMap
{
<MapItems>
	<MapItem MessageType="h2.consumerestfulservice.Request">
		<Method>ProcessRequest</Method>
	</MapItem>
</MapItems>
}

}
```

The service takes the HTTP request in variable %request. Afterwards it reads the parameters straight from the request.

```
Class h2.createrestfulservice.Service Extends Ens.BusinessService
{

Parameter ADAPTER;

Method OnProcessInput(pInput As %Stream.Object, Output pOutput As
 %Stream.Object) As %Status
{
	#dim %request as %CSP.Request
```

```
        set pOutput = {}
        set pOutput.returnTo = %request.Get("returnRecepient")
        set pOutput.message = %request.Get("message")
        Quit $$$OK
}

}
```

It was easy.

Form data

It is as easy as the URL query type. There is only one difference. The client feeds the form data array of the request instead of the parameter array. In fact the server side Business service is exactly the same.

```
Class h2.consumerestfulservice.Operation Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Property Adapter As EnsLib.HTTP.OutboundAdapter;

Parameter INVOCATION = "Queue";


Method ProcessRequest(pRequest As
 h2.consumerestfulservice.Request, Output pResponse As
 h2.consumerestfulservice.Response) As %Status
{
        set st = $$$OK
        set httpRequest = ##class(%Net.HttpRequest).%New()
        do httpRequest.InsertFormData("returnRecepient",pRequest.returnRecepient)
        do httpRequest.InsertFormData("message",pRequest.message)
        set url = ..Adapter.URL_"/echo"
        set st = ..Adapter.SendFormDataArray(.callResponse,"GET",httpRequest,,,url)
        #dim callResponse as %Net.HttpResponse
        set dynamicObject = {}.%FromJSON(callResponse.Data)
        set responseName = pRequest.accept
        set pResponse = $classmethod(responseName,"%New",dynamicObject)
        Quit st
}


XData MessageMap
{
<MapItems>
        <MapItem MessageType="h2.consumerestfulservice.Request">
              <Method>ProcessRequest</Method>
        </MapItem>
</MapItems>
}

}
```

Message body

When a service does a complex job, the parameter structure is complex too. As the parameter structure gets more complex the URL path, URL query or form data parameter types get less useful. In such situation (almost) the only solution is to put the parameters into the message body. The following example shows how the client serializes the parameter object into the request “EntityBody”, and later on how the server reanimates the parameter object.

The next program code shows the client side. The object instance “callRequest” is representing the parameter object. In our case it is a %DynamicObject. It is not necessary to use dynamic objects. It was just an easy solution.

It is also important to remember, that the content type must be “application/json”. It is true for communication to Ensemble server. Servers based on other technology may require other content type.

```
Class h2.consumerestfulservice.Operation Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Property Adapter As EnsLib.HTTP.OutboundAdapter;

Parameter INVOCATION = "Queue";

Method ProcessRequest(pRequest As
 h2.consumerestfulservice.Request, Output pResponse As
 h2.consumerestfulservice.Response) As %Status
{
	set st = $$$OK
	set callRequest = {}
	set callRequest.returnRecepient = pRequest.returnRecepient
	set callRequest.message = pRequest.message
	set httpRequest = ##class(%Net.HttpRequest).%New()
	set httpRequest.ContentType = "application/json"
	do httpRequest.EntityBody.Write(callRequest.%ToJSON())
	set url = ..Adapter.URL_"/echo"
	set st = ..Adapter.SendFormDataArray(.callResponse,"PUT",httpRequest,,,url)
	#dim callResponse as %Net.HttpResponse
	set dynamicObject = {}.%FromJSON(callResponse.Data)
	set responseName = pRequest.accept
	set pResponse = $classmethod(responseName,"%New",dynamicObject)
	Quit st
}


XData MessageMap
{
<MapItems>
	<MapItem MessageType="h2.consumerestfulservice.Request">
		<Method>ProcessRequest</Method>
	</MapItem>
</MapItems>
}

}
```

The server side looks very different from the previous implementations. Processing the parameter structure requires deserialization. Due to the JSON serialization this seems to be fairly easy. Once the dynamic object instance is created, accessing individual parameter values are straightforward. No tricks, no conversion.

```
Class h2.createrestfulservice.Service Extends Ens.BusinessService
{

Parameter ADAPTER;

Method OnProcessInput(pInput As %Stream.Object, Output pOutput As
 %Stream.Object) As %Status
{
	#dim %request as %CSP.Request
	set requestObject = {}.%FromJSON(%request.Content)
	set pOutput = {}
	set pOutput.returnTo = requestObject.returnRecepient
	set pOutput.message = requestObject.message
	Quit $$$OK
}

}
```

There is one extra task with the “body” parameter type. We need to change the Resource Map. It is because the HTTP rarely use the message body with no form data for verbs GET or POST. On the other hand PUT is expected to put the parameter structure into the body. The following example of the Resource Map shows that a single entry for PUT is add to the UrlMap.

```
Class h2.createrestfulservice.ResourceMap Extends %CSP.REST
{

///
/// The UrlMap determines how a Url should map to a HTTP Method and a Target ClassMet
hod
/// indicated by the 'Call' attribute. The call attribute is either the name of a met
hod
/// or the name of a class and method seperated by a ':'. Parameters within the URL p
receded
/// by a ':' will be extracted from the supplied URL and passed as arguments to the n
amed method.
///
/// In this Route Entry GET requests to /class/namespace/classname will call the GetC
lass method
///
/// <Route Url="/class/:namespace/:classname" Method="GET" Call="GetClass"/>
///
XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
<Route Url="/:service" Method="GET" Call="InvokeEnsembleService"/>
<Route Url="/:service/:p1" Method="GET" Call="InvokeEnsembleService"/>
<Route Url="/:service/:p1/:p2" Method="GET" Call="InvokeEnsembleService"/>
```

```
<Route Url="/:service" Method="PUT" Call="InvokeEnsembleService"/>
</Routes>
}

ClassMethod InvokeEnsembleService(service, argv...) As %Status
{
	set status = ##class(Ens.Director).CreateBusinessService(service, .instace)
	if $$$ISOK(status) {
		#dim response as %DynamicObject
		set status = instace.ProcessInput(.argv, .response)
		if $isObject(response) {
			write response.%ToJSON()
		}
	}
	quit status
}

}
```

Service response

In all of our previous examples a response is created (Business Service), returned (Resource Map) and processed (Business Operation). There is no more complication I would like to make. However I need to emphasize, that all above examples are returning HTTP 200 – OK status code. It is misleading and makes the response processing difficult. There will be another article about exception handling. Please watch the upcoming articles.

Testing

All demos could be tried out, tested by standard Ensemble tools such as Testing Services, Message Trace, Event Log, IO Archive or CSP Gateway HTTP Trace. If you are truly after some fancy tool, go for it. But after all what you are testing is the good old Ensemble.

Stay tuned, I'll be back soon with further reading on Ensemble RESTful web services. The next is "RESTful Exception Handling".

#Beginner #REST API #Ensemble
30 3 0 0 3,463

Log in or sign up to continue
Add reply

Source URL: https://community.intersystems.com/post/restful-way-data-transfer