REST

- Introduction

Introduction

Web services can also use other technologies, apart from SOAP, such as RESTful implementations on top of HTTP. Representational State Transfer (REST) is an approach based on the architectural style of the Web itself. The SOA Gateway also provides this URL based approach to access resources.

REST Overview

SOA Gateway allows users to access any web service via a REST-style URL request. In general, this is a more simplistic way of accessing services, useful in demo scenarios, and with clients that do not have support for SOAP, but do have support for retrieving URLs information (such as Microsoft Excel).

A REST request is similar to the WSDL request, but with extra arguments. Generally, it is recommended that the WSDL is retrieved first, as it gives the client the ability to see what fields have been set as keys. All operations that are possible using the WSDL are possible with REST, with some caveats.

Operation	*Notes*
get	MTOM is not supported. In the case where binary objects are returned on request, the XML will be escaped into HTML, and a link to the binary object will also be returned.
add/update	HTTP POST must be used.
delete	HTTP DELETE must be used.

Example

The following is an example of retrieving data with a REST request

```
http://host:port/myService?LIST&ID=4*&Name=J*
```

This will attempt to call the "list" operation, passing in a value of 4* to the ID field (which has been defined as a primary/secondary key) AND the Name field set to J*

Enhanced REST Operations

SOA Gateway provides several operations for each web service so it has enhanced its REST implementation to support them e.g. SELECT and INVOKE. Typically these may require complex parameters in order to be called.

```
http://localhost:56005/adabas_Employees_9?
SELECT
&condition[1].personnel_id>50012100
&condition[1].personnel_id<=50012700
&condition[2].personnel_id=50012900
```

The example above specifies 2 condition blocks. This will return data where the (personnel_id > 50012100 and personnel_id <= 50012700) or personnel_id = 50012900

Database WSDLs

A SOA Gateway database WSDL defines requests which reflect database access.

Supported Requests

1. LIST
2. GET
3. DELETE
4. ADD
5. UPDATE
6. SELECT
7. SELECTCOUNT

URI

As usual in the definitions element there will be a value for the targetNamespace uri:

```
<definitions targetNamespace="uri://46.46.46.46:56421/Customers"
name="CustomersRootCollection">
```

The uri gives us the starting portion of a REST request:

```
http//46.46.46.46:56421/Customers
```

Note that in Portus WSDLs a unique identifier (UNIQID) is prepended to various elements and aslo contained in the uri e.g. in this case Customers in the name CustomersRootCollection.

Messages

For each the above requests there will be a message entry in the WSDL with the following names:

getRequest, listRequest, deleteRequest, addRequest, updateRequest, selectRequest and selectCountRequest

e.g.

<message name="listRequest">

<part name="CustomersGroupListKey" element="asg:CustomersGroupListElement"/>

</message>

.

.

<message name="getRequest">

<part name="CustomersGroupGetKey" element="asg:CustomersGroupGetElement"/>

</message>

.

.

<message name="deleteRequest">

<part name="CustomersGroupDeleteKey" element="asg:CustomersGroupDeleteElement"/>

</message>

.

.

.

Note that there will be some others which are in the WSDL which are not supported in a REST request i.e. selectNext and selectEnd.

e.g.

<message <message name="selectNextRequest">

<part name="CustomersGroupSelectNextRequest" element="asg:CustomersGroupSelectNextElement"/>

</message>

Each message element has a part element which gives further details about the request structure via REST:

LIST

<part name="UNIQIDGroupListKey" element="asg:UNIQIDGroupListElement"/>

<xs:element name="UNIQIDGroupListElement" type="asg:UNIQIDGroupKeyType"/>

<xs:complexType name="UNIQIDGroupKeyType">

<xs:sequence>

<xs:element name="ID" nillable="true" type="xs:int"/>

<xs:element name="Account_ID" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s). A feature of the LIST request is that these parameters can be wild carded as shown below and/or omitted:

http://46.46.46.46:56421/Customers?LIST&ID=*

http://46.46.46.46:56421/Customers?LIST&ID=2

http://46.46.46.46:56421/Customers?LIST&ID=2*

http://46.46.46.46:56421/Customers?LIST&ID=*&Account_ID=2*

http://46.46.46.46:56421/Customers?LIST&Account_ID=2*

http://46.46.46.46:56421/Customers?LIST&Account_ID=*5

GET

<part name="UNIQIDGroupGetKey" element="asg:UNIQIDGroupGetElement"/>

<xs:complexType name="UNIQIDGroupPrimaryKeyType">

<xs:sequence>

<xs:element name="ID" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s). Note that a GET targets a specific row in the database and returns one record or none if not found:

http://46.46.46.46:56421/Customers?GET&ID=25

DELETE

<part name="UNIQIDGroupDeleteKey" element="asg:UNIQIDGroupDeleteElement"/>

<xs:element name="UNIQIDGroupDeleteElement" type="asg:UNIQIDGroupPrimaryKeyType"/>

<xs:complexType name="UNIQIDGroupPrimaryKeyType">

<xs:sequence>

<xs:element name="ID" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s). Note that a DELETE targets a specific row in the database. If successful it returns a ‘delete successful’ message or an error stating that it does not exist.

http://46.46.46.46:56421/Customers?DELETE&ID=25

ADD

<part name="UNIQIDRoot" element="asg:UNIQIDRootAddElement"/>

<xs:element name="UNIQIDRootAddElement" type="asg:UNIQIDRootType"/>

<xs:complexType name="UNIQIDRootType">

<xs:sequence>

<xs:element maxOccurs="unbounded" minOccurs="0" name="UNIQIDGroup"
type="asg:UNIQIDGroupType"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="UNIQIDGroupType">

<xs:sequence>

<xs:element name="ID" nillable="true" type="xs:int"/>

<xs:element name="FirstName" type="xs:string"/>

<xs:element name="Surname" type="xs:string"/>

<xs:element name="Street" type="xs:string"/>

<xs:element name="City" type="xs:string"/>

<xs:element name="State" type="xs:string"/>

<xs:element name="Zip" type="xs:string"/>

<xs:element name="Phone" type="xs:string"/>

<xs:element name="SSN" nillable="true" type="xs:int"/>

<xs:element name="Account_ID" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s). All elements are at the same level i.e. not in a structure so can be sequentially added to the REST request.

```
http://46.46.46.46:56421/Customers?ADD&ID=Value&FirstName=Value&Surname=Value&Street=Value&City=Value&State=Value&Zip&Phone=Value&SSN&Account_ID=Value
```

UPDATE

<part name="UNIQIDRootUpdate" element="asg:UNIQIDRootUpdateElement"/>

<xs:element name="UNIQIDRootUpdateElement" type="asg:UNIQIDRootType"/>

<xs:complexType name="UNIQIDRootType">

<xs:sequence>

<xs:element maxOccurs="unbounded" minOccurs="0" name="UNIQIDGroup" type="asg:UNIQIDGroupType"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="UNIQIDGroupType">

<xs:sequence>

<xs:element name="ID" nillable="true" type="xs:int"/>

<xs:element name="FirstName" type="xs:string"/>

<xs:element name="Surname" type="xs:string"/>

<xs:element name="Street" type="xs:string"/>

<xs:element name="City" type="xs:string"/>

<xs:element name="State" type="xs:string"/>

<xs:element name="Zip" type="xs:string"/>

<xs:element name="Phone" type="xs:string"/>

<xs:element name="SSN" nillable="true" type="xs:int"/>

<xs:element name="Account_ID" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s). All elements are at the same level i.e. not in a structure so can be sequentially added to the REST request. Note that as ID (see UNIQIDGroupPrimaryKeyType)is the primary key, the value passed in the request should exist in the database table.

```
http://46.46.46.46:56421/Customers?UPDATE&ID=existingKeyValue&FirstName=Value&Surname=Value&Street=Value&City=Value&State=Value&Zip&Phone=Value&SSN&Account_ID=Value
```

SELECT

```
<part name="UNIQIDGroupSelectKey" element="asg:UNIQIDGroupSelectElement "/>
<xs:element name="UNIQIDGroupSelectElement" type="asg:UNIQIDGroupSelectType"/>
<xs:complexType name="UNIQIDGroupSelectType">
<xs:sequence>
<xs:element maxOccurs="unbounded" minOccurs="1" name="condition">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" minOccurs="0" name="ID">
<xs:complexType>
<xs:simpleContent>
<xs:extension base="xs:string">
<xs:attribute name="Condition" type="asg:conditionType"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element maxOccurs="unbounded" minOccurs="0" name="Account_ID">
<xs:complexType>
<xs:simpleContent>
<xs:extension base="xs:string">
<xs:attribute name="Condition" type="asg:conditionType"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
```

</xs:element>

</xs:sequence>

</xs:complexType>

At this point we know our input parameter(s).

N.B.

Both select and selectCount have input the elements of which are contained in a structure. The parent element is condition which will contain elements which are primary and secondary keys (in the underlying database). The SOAP equivalent message portion would be:

<!--1 or more repetitions:-->

<condition>

<!--Zero or more repetitions:-->

<ID Condition="?"></ID>

<!--Zero or more repetitions:-->

<Account_ID Condition="?"></Account_ID>

</condition>

(As per the WSDL the Condition type can be EQ, NE, LT, LE, GT, GE, STARTS, CONTAINS and ENDS).

<condition>

<ID Condition="GT">4</ID>

<ID Condition="LE">10</ID>

</condition>

<condition>

<Account_ID Condition="EQ">23</Account_ID>

</condition>

The above element will select records where the ID is greater than 4 AND less than or equal to 10 OR where Account_ID is equal to 23.

There are 2 condition elements so use array notation for those (one being the base).

Use numeric notation for the condition type i.e. for GT use >

e.g.

http://46.46.46.46:56421/Customers?SELECTCOUNT&condition[1].ID>4&condition[1].ID<=10&condition[2].Account_ID=23

Program WSDLs

A SOA Gateway WSDL which is program based supports an INVOKE request.

Simple Example

Excerpt from typical program WSDL:

<xs:element name="invokeInputElement">

<xs:complexType>

<xs:sequence>

<xs:element name="SOABSP_CALCULATRoot">

<xs:complexType>

<xs:sequence>

<xs:element name="SOABSP_CALCULATGroup">

<xs:complexType>

<xs:sequence>

<xs:element name="OPERATION" type="xs:string"/>

<xs:element name="OPERAND_1" nillable="true" type="xs:int"/>

<xs:element name="OPERAND_2" nillable="true" type="xs:int"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

- In the definitions element there will be a value for the targetNamespace uri:

<definitions targetNamespace="uri://www.versatec.info:56421/SOABSP_CALCULAT" …

The uri gives us the starting portion of a REST request:

http://www.versatec.info:56421/SOABSP_CALCULAT

Note also that in Portus WSDLs a unique identifier (UNIQID) is prepended to various elements and also contained in the uri e.g. in this case SOABSP_CALCULAT:

<xs:element name="SOABSP_CALCULATRoot">

- There will be an element with a name of invokeInputElement. This is reflected in the F=INVOKE portion of the REST request.

  http://www.versatec.info:56421/SOABSP_CALCULAT?F=INVOKE

- invokeInputElement will contain an element with a name UNIQIDRoot
- UNIQIDRoot will contain an element with a name UNIQIDGroup
- UNIQIDGroup will contain the elements that are passed in the INVOKE for a REST request e.g.

  <xs:element name="OPERATION" type="xs:string"/>

  <xs:element name="OPERAND_1" nillable="true" type="xs:int"/>

  <xs:element name="OPERAND_2" nillable="true" type="xs:int"/>

  http://www.versatec.info:56421/SOABSP_CALCULAT?F=INVOKE&OPERATION=mul&OPERAND_1=2345&OPERAND_2=6789

Complex Example

Excerpt from more complex program WSDL:

<xs:element name="invokeInputElement">

<xs:complexType>

<xs:sequence>

<xs:element name="QEESPN01Root">

<xs:complexType>

<xs:sequence>

<xs:element name="QEESPN01Group">

<xs:complexType>

<xs:sequence>

<xs:element name="QEESPS01">

```
<xs:complexType>
<xs:sequence>
<xs:element name="REDEFINE_001_IPF">
<xs:complexType>
<xs:sequence>
<xs:element name="TIPO_IPF" type="xs:string"/>
<xs:element name="REDEFINE_002_NUM_IPF">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="10" name="NUMN_IPF" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="FECHA_DESDE" nillable="true" type="xs:decimal"/>
<xs:element name="REDEFINE_003_COD_IPF">
<xs:complexType>
<xs:sequence>
<xs:element name="ALFA2" type="xs:string"/>
<xs:element name="ALFA8" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="FECHA_NAC" type="xs:string"/>
<xs:element maxOccurs="10" name="DUP_F10" nillable="true" type="xs:decimal"/>
```

<xs:element maxOccurs="100" name="D_SALIDA">

<xs:complexType>

<xs:sequence>

<xs:element name="MOMMAP" type="xs:string"/>

<xs:element name="SALIDA_DATA" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

- In the definitions element there will be a value for the targetNamespace uri:

  <definitions targetNamespace="uri://meath-nua:56008/QEESPN01" name="QEESPN01RootCollection">…

  The uri gives us the starting portion of a REST request:

  http//meath-nua:56008/QEESPN01

  In Portus WSDLs a unique identifier (UNIQID) is prepended to various elements and also contained in the uri e.g. in this case QEESPN01

- There will be an element with a name of invokeInputElement. This is reflected in the F=INVOKE portion of the REST request.

http//meath-nua:56008/QEESPN01?F=INVOKE

- invokeInputElement will contain an element with a name UNIQIDRoot
- UNIQIDRoot will contain an element with a name UNIQIDGroup
- UNIQIDGroup will contain the elements that are passed in the INVOKE for a REST request e.g.

<xs:element name="QEESPS01">

<xs:complexType>

<xs:sequence>

<xs:element name="REDEFINE_001_IPF">

<xs:complexType>

<xs:sequence>

<xs:element name="TIPO_IPF" type="xs:string"/>

<xs:element name="REDEFINE_002_NUM_IPF">

<xs:complexType>

<xs:sequence>

<xs:element maxOccurs="10" name="NUMN_IPF" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="FECHA_DESDE" nillable="true" type="xs:decimal"/>

<xs:element name="REDEFINE_003_COD_IPF">

<xs:complexType>

<xs:sequence>

<xs:element name="ALFA2" type="xs:string"/>

<xs:element name="ALFA8" type="xs:string"/>

```
</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="FECHA_NAC" type="xs:string"/>

<xs:element maxOccurs="10" name="DUP_F10" nillable="true" type="xs:decimal"/>

<xs:element maxOccurs="100" name="D_SALIDA">

<xs:complexType>

<xs:sequence>

<xs:element name="MOMMAP" type="xs:string"/>

<xs:element name="SALIDA_DATA" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>
```

- If the element names within UNIQIDGroup are in a structure then the REST request must reflect that:

1. QEESPS01 is the top level element name. It has a child REDEFINE_001_IPF which in turn has a child element TIPO_IPF so the parameter should be specified as:

&QEESPS01.REDEFINE_001_IPF.TIPO_IPF=0

2. If an element can occur more than once (maxOccurs > 1) then use array notation:

&QEESPS01.D_SALIDA[0].MOMMAP=0&QEESPS01.D_SALIDA[0].SALIDA_DATA=0&QEESPS01.D_SALIDA[1].MOMMAP=22&QEESPS01.D_SALIDA[1].SALIDA_DATA=55

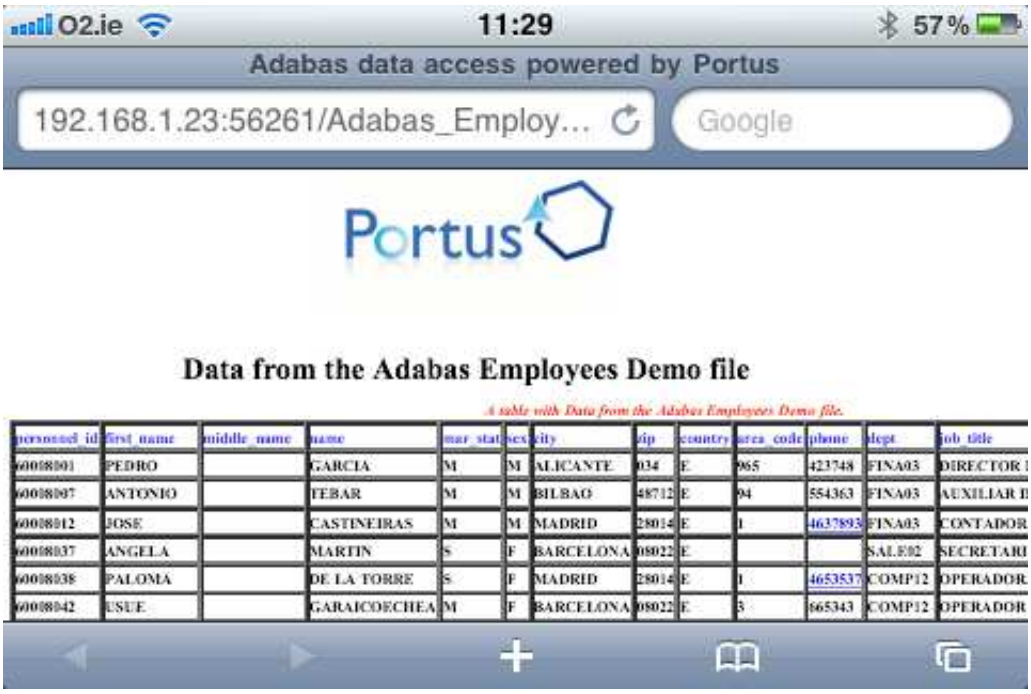
XSL Transformation

REST requests also support XSL transformation. The XSL file should be defined in the Control Centre, and have the same name as the XRD and XSD file. When a REST request is made, the URL for the XSL is added into the returned XML response, thus allowing the client to retrieve the XSL, and apply the transformation.

For clients that do not support client-side XSL transformation, such as some Android and Blackberry devices, it is possible to apply the transformation on the server side (e.g. SOA Gateway will apply the transformation, and return the transformed data). This is determined by the HTTP User-Agent header, and should normally be done transparently. It is possible to force client or server transformation with an option on the REST URL.

```
http://host:port/myService?LIST&__xslTransform=server&ID=1*
```

```
http://host:port/myService?LIST&__xslTransform=client&ID=1*
```



Using different encodings

See here for more information about the __encoding option on REST request.