PYMMA

OpenESB Enterprise Edition

REST Binding Component

Tutorial

October 2018

Document number 770-016

Please send feedback to: contact@pymma.com

This document has been written by non- native English speakers. Despite our best effort and the reviews made by Word, Ginger© and Grammarly©, the document can contain faults and bad sentences. We make an apology for this. Any review and corrections are welcome. Thank you for your feedback.

About Pymma Consulting

Pymma Consulting, an EU company founded in 1999, is a software systems technical architecture bureau. Pymma provides expertise in the conception and execution of integration projects and programs. Working since its inception with top 500 companies and government entities, Pymma has evolved unique skills in the integration and services marketplace. Proactive in the open source landscape, Pymma supports, develops and sponsors open-source projects such as OpenESB, Drools and Gravitee.io, and is the architect of the proprietary systems integration platform *OpenESB Enterprise Edition ("OEEE")*. Pymma is based in London, UK, with satellite activity in the Netherlands, France and Canada.

To reach Pymma: contact@pymma.com or www.pymma.com

About The author

Jeanne ROMEFORT is an engineering student at Mines Saint-Etienne. She did an internship at PYMMA from March 2018 to July of the same year. She worked as a developer during this time and she is expected to graduate from Mines Saint-Etienne in September 2019.

Contents

Introduction

Today (2018 Q3) REST over HTTP is a very useful protocol to invoke and publish web services. One of the main differences between REST and the other protocols supported by OpenESB, is that REST uses the HTTP operations and fault codes to define the exchanges between partners. It doesn't give them the opportunity to create business-focused operations and faults. This anomaly that mixes protocol and business operations is the strength and the weakness of REST. On one side, using the HTTP operations such as GET, PUT or POST, uniformises the communication between partners and make design and implementation easier, but on the other side impoverishes it.

This significant difference has an impact on the way the REST binding component works and the way to use it. For this reason, REST BC settings appear different from the other Binding Components. This feeling is mainly due to the fact, REST communications are strongly linked to the HTTP protocol.

About REST (from Wikipedia)[1]

Representational State Transfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. Web Services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations.

"Web resources" were first defined on the World Wide Web as documents or files identified by their URLs. However, today they have a much more generic and abstract definition that encompasses everything or entity that can be identified, named, addressed, or handled, in any way whatsoever, on the web. In a RESTful web service, requests made to a resource's URI will elicit a response with a payload formatted in either HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When HTTP is used, as is most common, the operations available are GET, POST, PUT, DELETE, and other predefined CRUD HTTP methods. By using a stateless protocol and standard operations, REST systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system, even while it is running.

The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding's dissertation explained the REST principles that were known as the "HTTP object model" beginning in 1994 and were used in designing the HTTP 1.1 and Uniform

[1] https://en.wikipedia.org/wiki/Representational_state_transfer

Resource Identifiers (URI) standards. The term is intended to evoke an image of how a well-designed Web application behaves: it is a network of Web resources (a virtual state-machine) where the user progresses through the application by selecting links, such as /user/tom, and operations such as GET or DELETE (state transitions), resulting in the next resource (representing the next state of the application) being transferred to the user for their use.

About REST Binding Component

The current REST Binding component developed by Pymma is an evolution of the legacy REST Binding Component embedded in the OpenESB legacy version 2.x. It is based on Jersey and supports the specification JSR 311.

REST BC Features

The REST BC provides OpenESB Enterprise Edition with the following features:

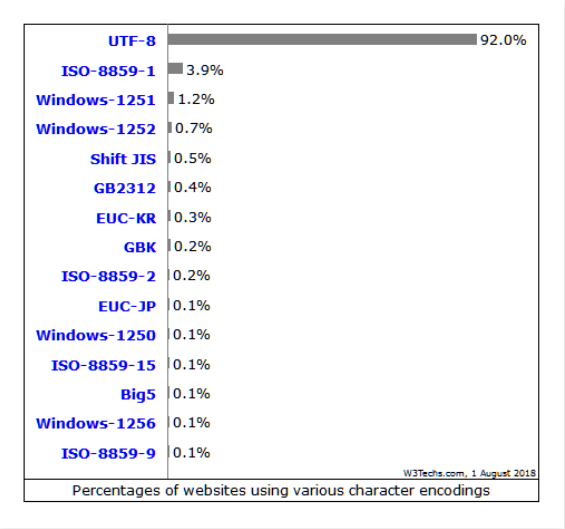
- Publish and Invoke RESTful web services with OpenESB Enterprise Edition
- Step by Step wizard to create a REST WSDL document.
- Dynamic setting of URLs, HTTP headers, Query parameters and path parameters during the process.
- Advanced request validation and HTTP code status generation
- Supports GET, PUT, POST, DELETE, and HEAD methods.
- Supports XML, JSON, and plain text content types.
- Advanced JSON-XML and XML-JSON conversion
- Support Jersey Filter
- Support Component Configuration and Application Variables
- Full multithreading code for great performances
- Supports Secure Sockets Layer (SSL).

REST BC limitations

The REST BC Component does not aim to replace a complex API management system, but to add a solid REST over HTTP connector to OpenESB and provide a simple way to access to and publish advanced business processes. We list some REST BC limitations that could be removed in the next versions.

- Media Type: REST BC supports: Application/JSON, Application/XML, Text/plain. These media types are matching the exchange between applications and processes. The other media types, such as the binary types, are not in the scope of OpenESB exchanges.
- REST BC comes with a wizard for Request-Response operation pattern only. One Way operation must be developed manually but are supported by the REST BC Component.
- Due to the strong link between the message and the protocol, it is not possible to use the WSDL Wizard at the CASA level. A concrete WSDL must be developed at the Service Unit Level.

- Charset: REST BC supports UTF-8 only, since today (2018 Q3) more than 90% of the exchanges are based on UTF-8. We did not get requests for other charsets.



Before starting

Before starting, get a hand on skill in REST API development (with Jersey for example)[2], it will be useful to understand the concepts on which the REST BC relies.

REST BC installation and Configuration

Installation

Install the REST BC like any other component. For more information, please have a look at the documentation 770-001 OpenESB EE installation.

Configuration

The REST BC has many runtime properties.

Property	Comment
NMR Core Thread Pool Size	The number of core threads used concurrently for processing NMR messages. The default value is 16.
NMR Max Thread Pool Size	The maximum number of threads used concurrently for processing NMR messages. The default value is 64
Default HTTP Listener Port	The default HTTP port number of the REST Binding Component. The default value is 9696

[2] https://jersey.github.io/

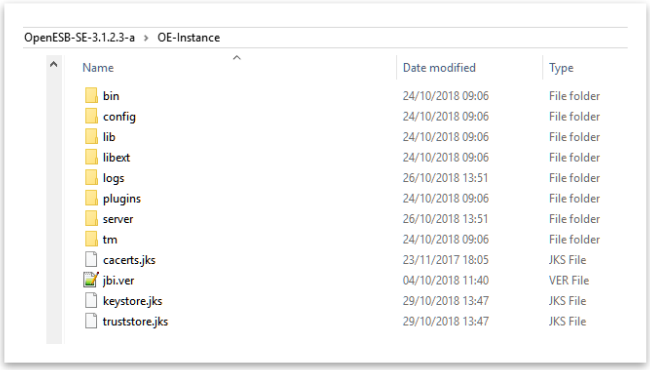
Default HTTP Listener Threads	The maximum number of threads to process RESTful services concurrently using HTTP. The default value is 32.
Default HTTPS Listener Port	The default HTTP secure port number of the REST Binding Component. The default value is 9697.
Default HTTPS Listener Threads	The maximum number of threads to process RESTful services concurrently using HTTPS. The default value is 32.
Truststore Password	The default truststore password, which is used to access the truststore used for key/certificate management when establishing SSL connections. By default: “changit”
Keystore Password	The default keystore password, which is used to access the keystore used for key/certificate management when establishing SSL connections. By default: “changit”
Mutual SSL	Enable mutual SSL
Host name verification	Hostname verification is a little-known part of HTTPS that involves a server identity check to ensure that the client is talking to the correct server and has not been redirected by a man in the middle attack. The check involves looking at the certificate sent by the server and verifying that the dnsName in the subjectAltName field of the certificate matches the host portion of the URL used to make the request. To use it you must have the certificate of your partner in your cacert file. If you don’t have it don’t use this feature.

Install the certificate and allows SSL

In the OpenESB launcher, “OpenESB.bat” or “OpenESB.sh”, two options setup the directories where the keystore and the cacert files are. These options are:

- "-Djavax.net.ssl.keyStore=%OPENESB_HOME%/keystore.jks"
- "-Djavax.net.ssl.trustStore=%OPENESB_HOME%/cacerts.jks"

You can set up any other directory for your configuration. You must define your own keystore, truststore and cacert file regarding your security requirements. Nevertheless, to help the user to access to the services hosted in HTTPS, we added a truststore and a keystore file in the REST component. Keystore and Truststore passwords are “changeit”. If you want to use the Hostname verifier, you must add the certificate of your partner in cacert.



For advanced information on OpenESB security, feel free to contact our experts at contact@Pymma. com

REST BC Big Picture

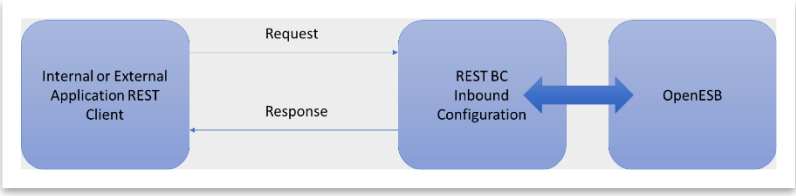
The REST Binding Component has been designed to give OpenESB Enterprise Edition applications the ability to provision and consume RESTful web services. There are two main configurations for the REST BC: The Inbound and the Outbound configurations. A REST BC supports many concurrent configurations per instance.

The configuration is stored in the WSDL of the services at the binding level. Since, REST over HTTP does not define a fix list of parameters at the protocol level, there is no way to store the HTTP parameters in a form or in a schema as it is done for other Binding Components. So, a CDATA section in the WSDL document is used to store the configurations. There is one CDATA section per operation defined in the REST BC. Example:

```
<rest:operation>
&lt;![CDATA[
# optional, name of HTTP listener to bind to, defaults to "default-listener"
http-listener-name=default-listener

# required, path to this resource
path=/testPathParams/{name}/{age}

# optional, HTTP verb to access the resource, defaults to GET
method=PUT

# optional, acceptable MIME types for request payload, defaults to "application/json",
"application/xml"
consume-types=[ "application/json", "application/xml" ]

# optional, all possible MIME types of response payload, defaults to
"application/json", "application/xml"
produce-types=[ "application/json", "application/xml" ]

forward-as-attachment=false

]]&gt;</rest:operation>
```

Inbound configuration

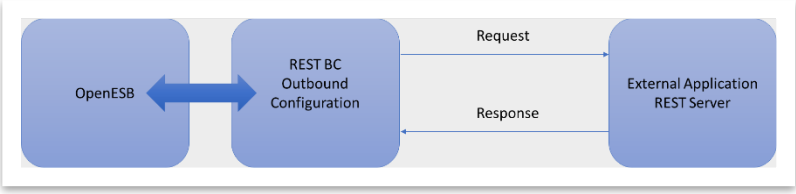
Setting up an Inbound configuration means that the REST BC publishes an OpenESB service invoked by an internal or external partner.



The inbound configuration defines the path of the service, the supported methods, the media type consumed and produced, the supported language... The charset linked to the **Content-Type** must be set to UFT-8 and **Accept-Charset** must at least contain UFT-8.

Outbound configuration

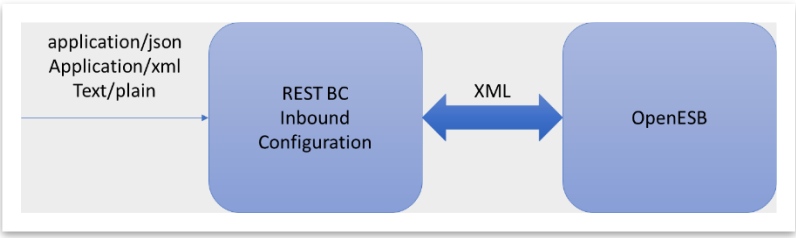
Setting up an outbound configuration means that the REST BC invokes a REST over HTTP web services. This service can be defined by OpenESB but any other internal or external partner.



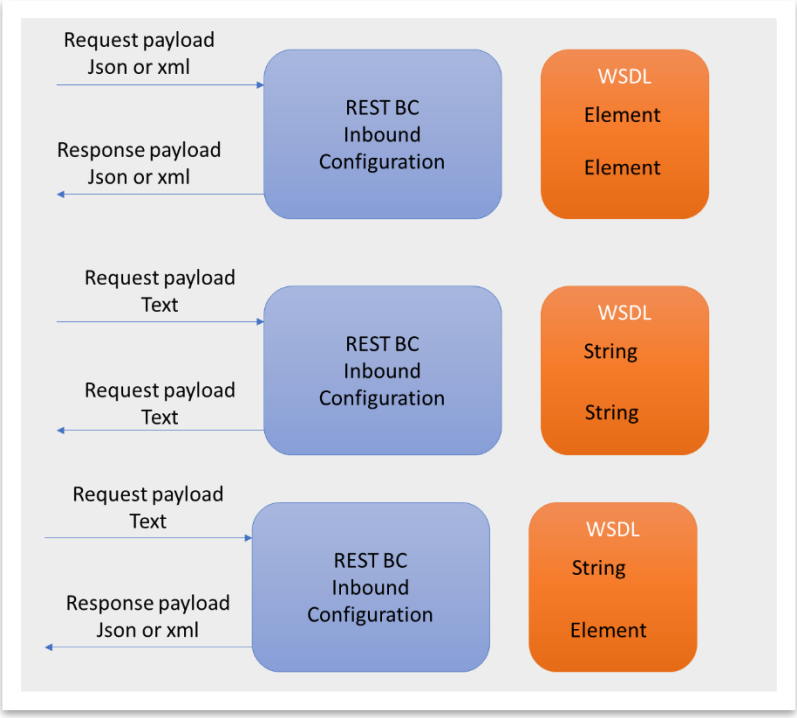
The outbound configuration defines the URL and the method of the invoked service, the media type of the request and the expected type for the response... The charset linked to the **Content-Type** must be UTF-8 and **Accept-Charset** must at least contain UFT-8.

REST BC Media type management

OpenESB bus and components rely on the XML to provide a strong typed and unambiguous definition of the exchanges between components. On the other side, REST over HTTP uses JSON, XML, CSV, String and many other formats for its payload messages. As explained above, OpenESB Enterprise Edition REST BC supports application/xml, application/json and text/plain media type. It means that the HTTP request and response body must be XML or JSON documents or for the other cases, a String.



It is the REST BC role to convert REST payload to XML and vs. Let's see, how to set up the inbound and outbound WSDL with different scenarios.



REST WSDL request and response messages **must** be set up with the correct types. Don't use the default type xsd:anyType except if you don't need it.

1. If the payload is an XML document (content-type=application/xml), the selected type must be a schema element.
2. If the payload is a text (content-type=text/plain) the selected type must be a String (Build-IN).
3. If the payload is a JSON document (content-type=application/json), the selected type must be a schema element. The schema element must match the JSON structure

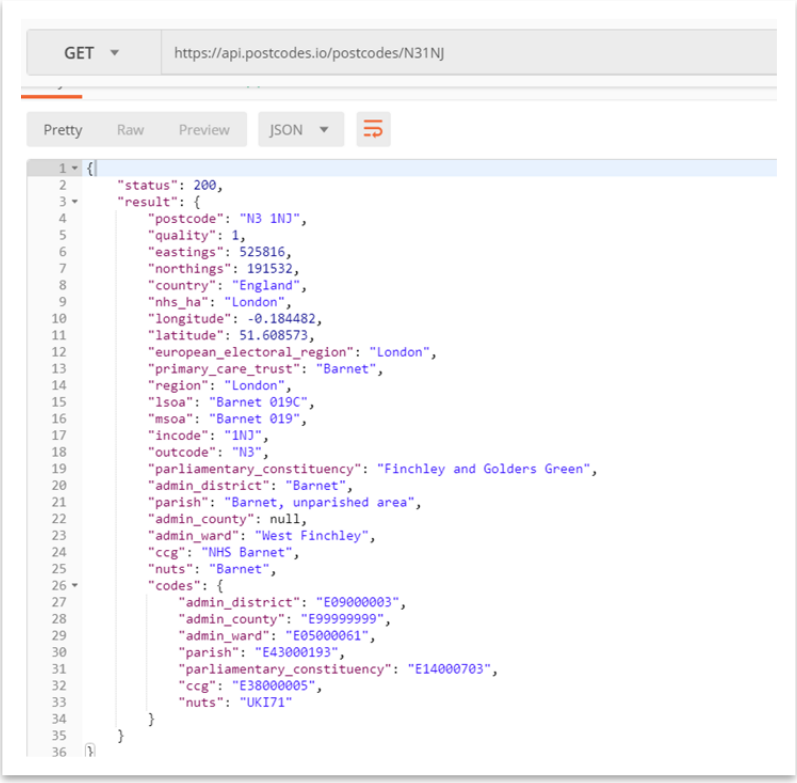
More on JSON support

As explained above, OpenESB internal processes are based on XML to take advantage of a strict definition of the message structure and contract of service. When a partner communicates with a

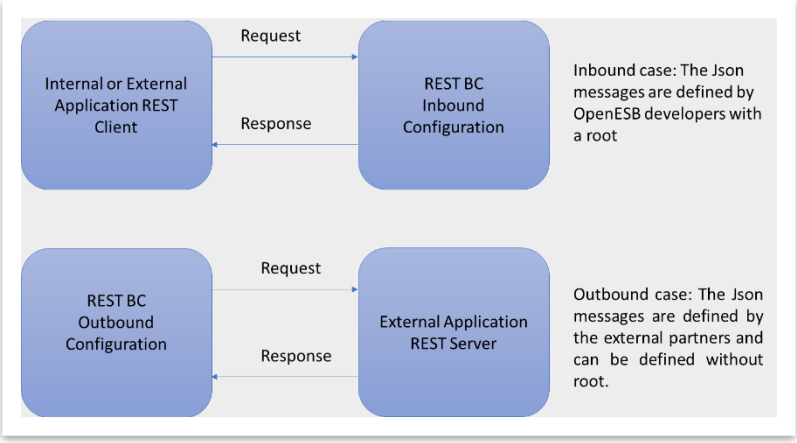
JSON message, a conversion JSON to XML and XML to JSON must be implemented. Even if JSON and XML look like similar, they have two main differences. The first is the namespace and the second is the Root.

The namespace concept is not implemented in JSON. During the conversion JSON to XML, the REST BC gets the current namespace of the WSDL message and add it to the XML document. On the other way, the namespace is removed from the XML and does not appear in the JSON document. This implies that all the elements used by the message (request or response) MUST have one and only one namespace. The conversion does not manage multiple namespaces in the same document.

The root element in XML is mandatory but not in JSON.



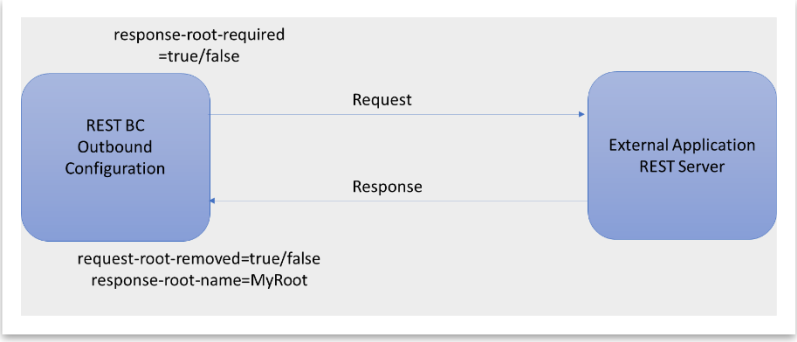
Above, the JSON response provided by the postcode.io API has not Root; status and result elements are at the same level. The conversion must add a root to the XML document. The root issue comes up only with the Outbound services. The Inbound messages are defined by OpenESB developers who must define a root in the JSON messages.



To solve this issue, REST BC has three parameters which are:

- response-root-required
- request-root-removed
- response-root-name

If response-root-required=true, REST BC adds a root element in the XML document after the conversion. The root name is defined by the parameter response-root-name. In the same way, the REST request body can be defined without root. In that case, set the parameter request-root-removed= true.



Root parameters are set in the outbound configuration.

```
&lt;![CDATA[
# required, URL of external resource
# for example: http://somehost.com/users/{id}/address
url=http://dummy.restapiexample.com/api/v1/create

# optional, HTTP verb to access the resource, defaults to GET
method=POST

# optional, acceptable media types of response can be added as array elements
in JSON format
accept-types=[ "application/json" ]
```

```
# optional, preferred natural languages for the response, added as array
elements in JSON format
accept-languages=[ ]

# optional, content type of outgoing payload, defaults to any type
content-type=application/json

# optional, additional custom HTTP headers can be added as pairs in JSON
format in this property
# for example:  { "authorization" : "257984234", "last-modifed" : "2009-04-
23:12:00:00" }
headers={ }

# optional, style for the parameters, valid values are QUERY, MATRIX,
defaults to QUERY
param-style=Query

# optional, additional custom HTTP parameters can be added as pairs in JSON
format in this property
# for example:  { "userid" : "abc" , "userpassword" : "123" }
params={ }

# optional, adding HTTP Basic Authentication header to the HTTP request,
# if the two properties below are specified
basic-auth-username=
basic-auth-password=

# optional, if the JSON document expected for a request has no root, remove
it with the option
#request-root-removed=true

#It the response of a REST request has no root, you can add one with this
option response-root-required. The root name element is defined by response-
root-name.
#response-root-required=true
#response-root-name=myRoot

]]&gt;
```

JSON and XML Schema

As explained above OpenESB uses XML documents clearly defined by Schemas. When the REST BC receives a JSON message, it converts the JSON document in XML and sent it to another component through the BUS. The result of the conversion must be defined and validated by an XML schema. This schema must be added to your projects. The process that creates an XML schema from a JSON document is simple.

1. Get the JSON document
2. Convert it into XML
3. Add a root if required
4. Get an XSD from the XML.

Online tools offer many conversions: JSON to XML, XML to XSD and vs. They are useful and straightforward. You can also use the JSON encoder developed by Pymma (see Appendix).

Whatever the tools you use, keep in mind that a well-formed schema with a root and a namespace is required to process the JSON messages.

Configurations

Inbound configuration

The inbound operation properties are defined in the table below:

Property	Description
Path	The path to the operation resource. This property is required. A URI is defined by the elements: scheme://authority/path [? query] [# fragment]. In an inbound operation, Scheme and authority are defined by the Rest BC configuration. So, you just have to define the Path in the configuration: Example: in the following URI http://api.pymma.com:9696/testmyapi/test01, http://api.pymma.com is defined in the REST BC configuration, so the configuration path is /testmyapi/test01. The REST BC support the path-parameters. Paths such as: /testmyapi/{code}/{name} are supported.
HTTP Listener	The name of the HTTP listener to bind to. The default value is default-listener. If you use SSL for the services, set up the HTTP-listener with the value default-listener-ssl. This property is optional.
Consume Types	The acceptable MIME types for the request payload, specified in JSON format. Enter the types in square brackets with each type contained in double-quotes. Separate multiple values by a comma. For example: ["text/plain", "application/xml"]. REST BC supports three content types: application/xml, application/json and text/plain. The charset can be added to the content-type, but it MUST be utf-8. The format is: Application/json;charset=utf-8
Produce Types	The acceptable MIME types for the response payload, specified in JSON format as above. This property is optional but must be set if you require a type for the response. For example: ["text/plain", "application/xml"]. REST BC supports three content types: application/xml, application/json and text/plain.
Forward as Attachment	An indicator of whether to forward the payload as an attachment. Select the check box to have the payload forwarded as an attachment. This property is optional.
User Defined	A list of user-defined properties in java.util.Properties format (key and value pairs). For example: serverName=test This property is optional.

Outbound configuration

The outbound operation properties are defined in the table below:

Property	Description
URL	The URL to the external resource. An URI is defined by the elements: scheme://authority/path [? query] [# fragment]. REST BC supports the main options of the URL. This property is required.
Accept Type	The acceptable media types for the response. REST BC support application/json, application/xml or text/plain. Enter the types in square brackets with each type contained in double-quotes. Separate multiple values by a comma. For example: ["application/json",application/xml] This property is optional.
Accept language	The preferred natural languages for the response, specified in JSON format. This property is optional.
Content type	The content type of the outbound payload. It must be application/json, application/xml or text/plain. If no value is specified, this defaults to application/xml This property is optional.
Header	Custom HTTP headers for the outbound payload. Enter the headers in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example: { "host" : "MyServer.com", "Content-Subtype" : "application/json/customers"} Custom headers are optional.
Parameter Style	A style for the URI parameters. Select one of the following options: **Query** – Name and value pairs that specify attributes of the full URI (external resource). Query parameters are delimited by an ampersand (&) and are separated from the rest of the URI by a question mark (?). **Matrix** – Name and value pairs that specify attributes of one segment in a URI. Matrix parameters can occur after the segment in the URI that they modify. Matrix parameters are delimited by a semicolon (;) and are also separated from the segment they modify by a semicolon.
Parameters	Custom HTTP parameters for the URI. Enter the parameters in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example: { "status" : "Active", "billing" : "Current"} Custom parameters are optional.
Basic Auth User Name	The login ID of the user for authentication. If the property is populated, a basic authentication header is added to the HTTP request.
Basic Auth Password	The login password corresponding with the above user name.
User Defined	A list of user-defined properties in java.util.Properties format (key and value pairs). For example: serverName=test This property is optional.
response-root-required	Indicates if the JSON to XML converter adds a root to the XML document during the conversion. The values can be true or false. Example response-root-required=true. This property is optional
response-root-name	Name of the root element added to the outbound response message. Example: response-root-name=myRoot. Only used if response-root-required equals true.

	This property is optional
request-root-removed=true	Indicates if the XML to JSON converter removes XML document root during the conversion. The values can be true or false. Example request-root-removed=true. This property is optional

REST BC Inbound example

Before going forward, let's start with a simple project that support many operations and details how to use the HTTP parameters (Header, Path, Query). In this example, you will understand how to set up the Inbound configuration and use the different parameters defined by the REST specification.

This short tutorial targets the intermediate developers and supposes you already develop with OpenESB. Else, you will find more information on Pymma documentation.

BPEL Project

1. Create a BPEL Project

Create a BPEL project and name it "TestRestBCInbound".

XML Schema

Create a new XSD and name it: "person". The XSD code is:

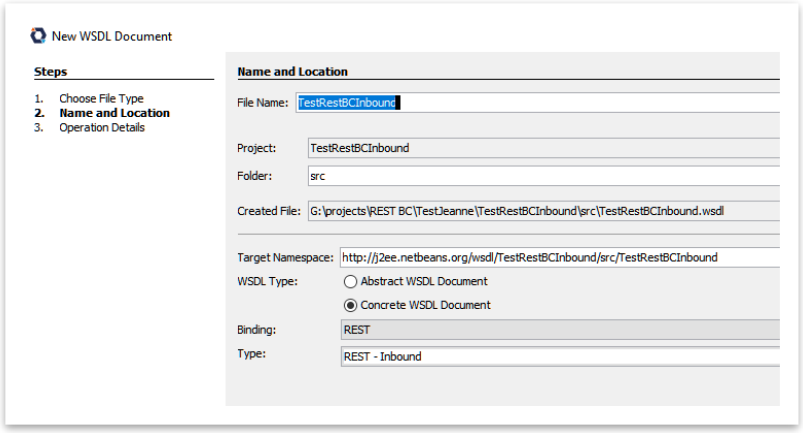
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/person"
    xmlns:tns="http://xml.netbeans.org/schema/person"
    elementFormDefault="qualified">
    <xsd:complexType name="personCT">
        <xsd:sequence>
            <xsd:element name="firstName" type="xsd:string"></xsd:element>
            <xsd:element name="lastName" type="xsd:string"></xsd:element>
            <xsd:element name="age" type="xsd:int"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="person" type="tns:personCT"></xsd:element>
</xsd:schema>
```

In this XSD, a person is defined by the first name, last name and age.

Create a WSDL

Create a concrete WSDL[3] and name it "TestRestBCInbound". Select "Concrete WSDL Document" and "REST – Inbound"

[3] Since REST is using the HTTP protocol features, it is easier for the beginner to start the development by a concrete WSDL. This does not cancel our recommendation to use abstract WSDL in the BPEL project and then define the concrete part of the WSDL in CASA.



Click on Next to create the Operations.

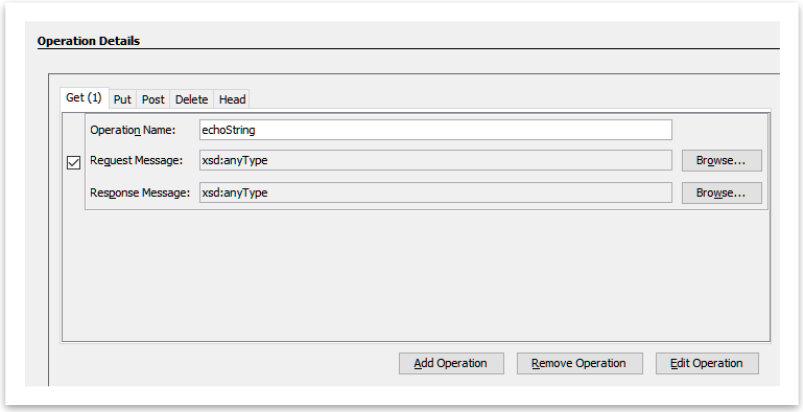
We create 5 operations for this tutorial.

Operation Name	Operation	Input Type	Output Type
EchoString	Get	N/A	String
EchoElement	Put	Person	Person
EchoWithParams	Get	N/A	Person
EchoWithPathParams	Put	Person	Person
EchoWithHeaderParams	Put	Person	Person

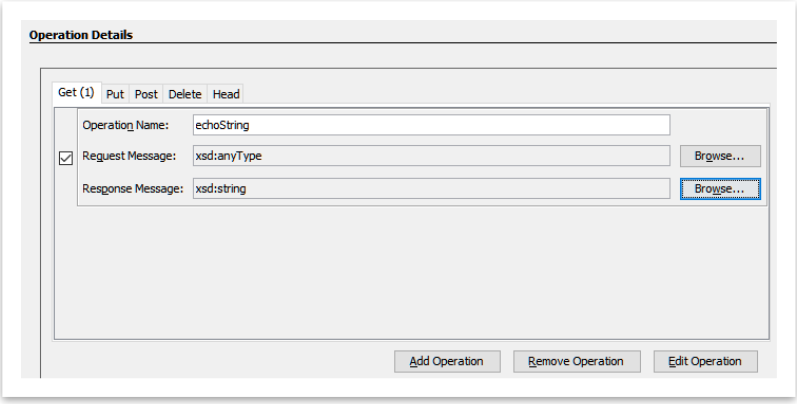
Let's detail the first and the second operations:

Operation EchoString

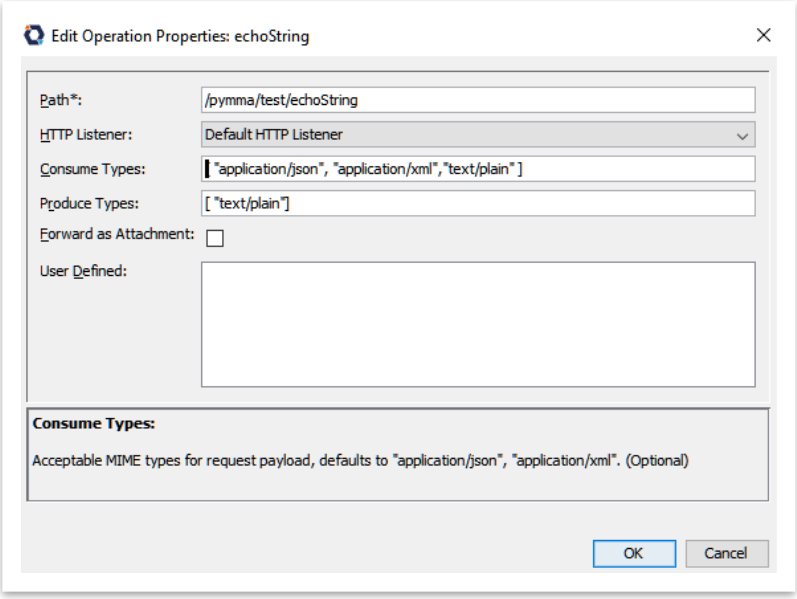
Click on Add Operation and enter the Operation Name. By default, the HTTP Operation is GET.



Since there is no request message payload with an HTTP get operation, we let the type to anyType. If, the response message payload is a String, the type MUST be a String.



Click on Edit operation to define the REST Inbound configuration

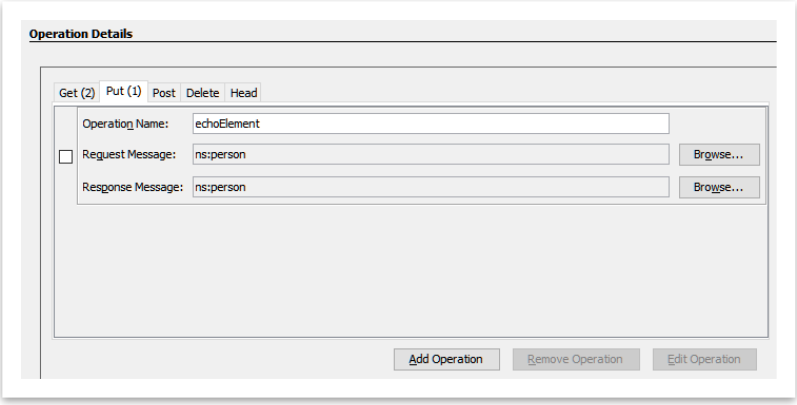


Type the path of the resources (the echoString service). Since there is no payload defined for a Get request, the consumed type can be anything. The response payload is a string, so to provide a safe answer, we force the produce type to “text/plain”.

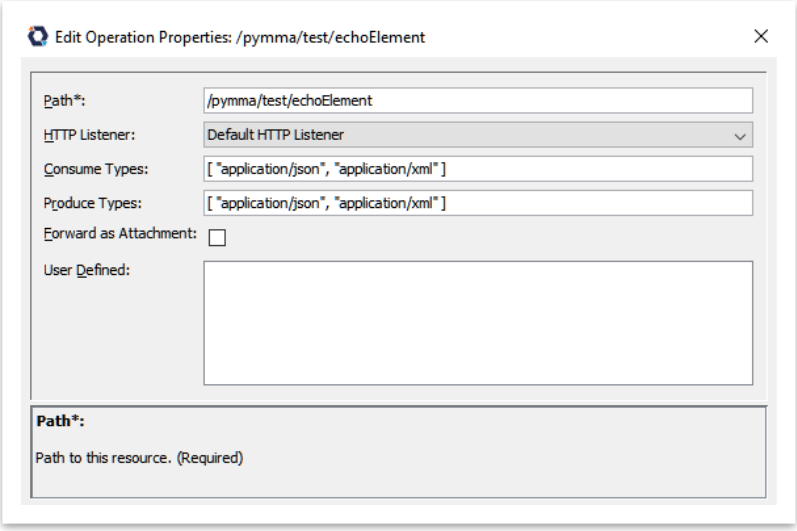
Click OK to save the configuration.

Operation EchoElement

EchoElement is an HTTP PUT operation. Request and Response payload type is a person. The type MUST be set to person and cannot be String or anyType.



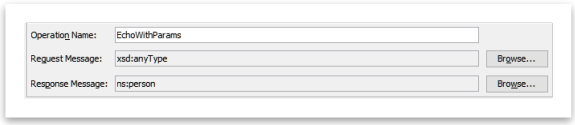
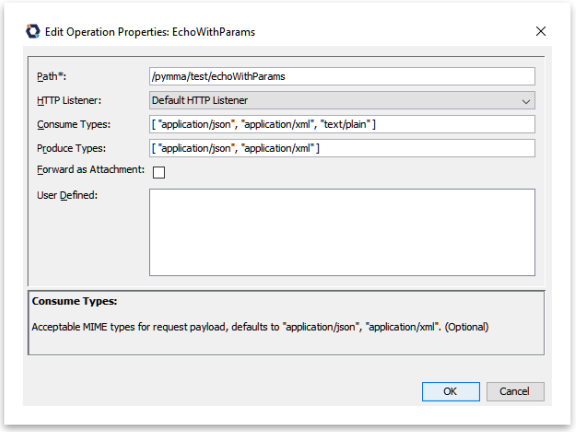
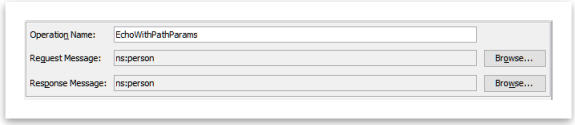
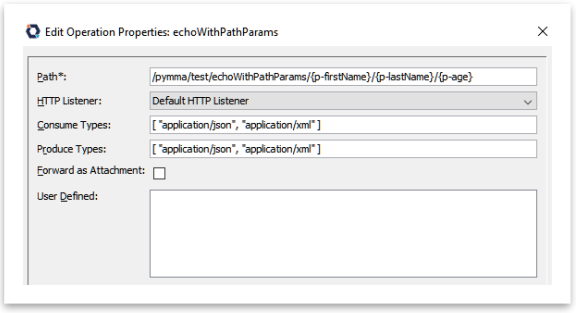
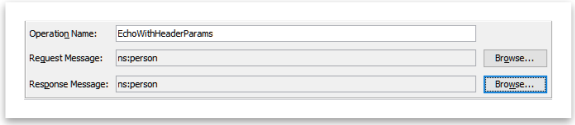
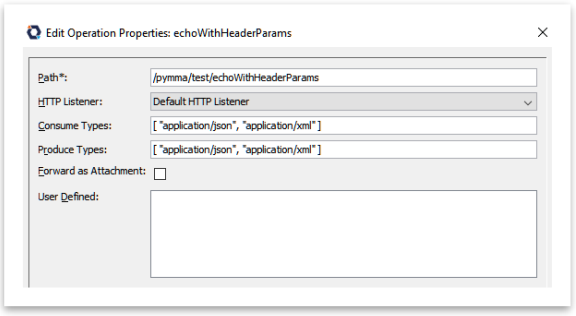
The inbound configuration is:



Notice that since we expect to receive and reply a person as payload, the consume and produce type cannot be “Text/Plain”.

Click OK to save the configuration.

The other operations

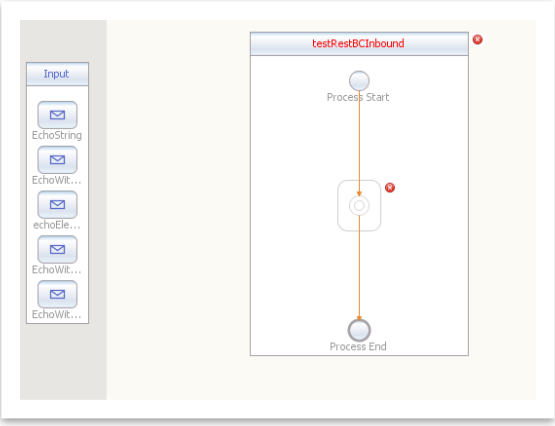
EchoWithParams 	
EchoWithPathParams 	
EchoWithHeaderParams 	

Click Finish to save the WSDL. Take time to analyse the WSDLs where the inbound configurations are located.

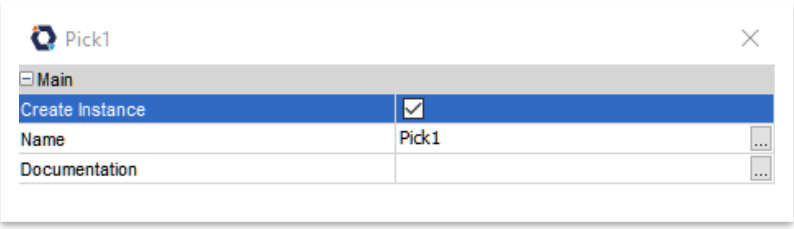
Operation echoString

BPEL design

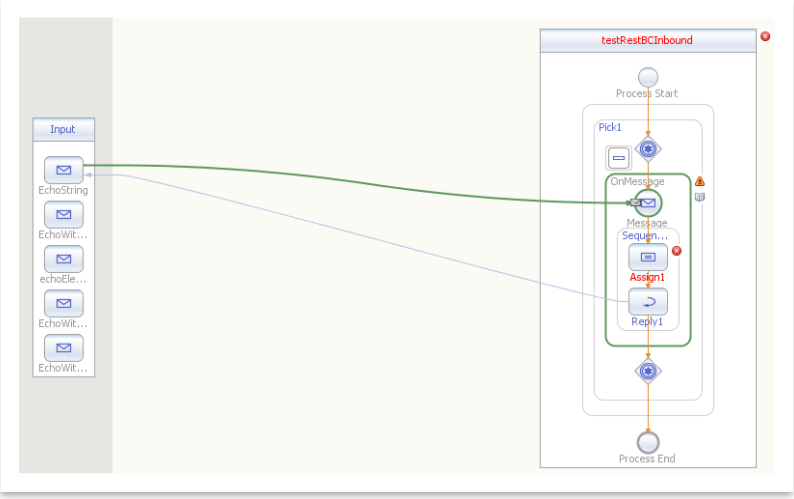
2. **Add the WSDL on the left lane of the BPEL editor. Notice that the operations are listed in the Partnerlink.**



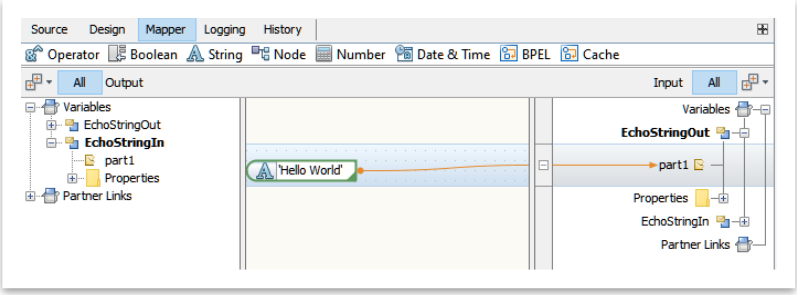
Add a Pick to the BPEL. Click right on the Pick, select properties and select create instance.



Link the onMessage activity with the first REST Operation. Add an Assign and Reply Activity.



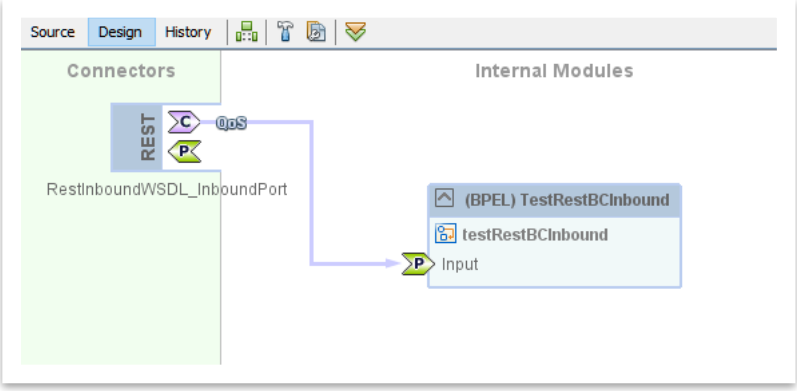
Set the Assign activity is as follows:



Composite application

Create a composite application and name it TestRestBCInboundCA.

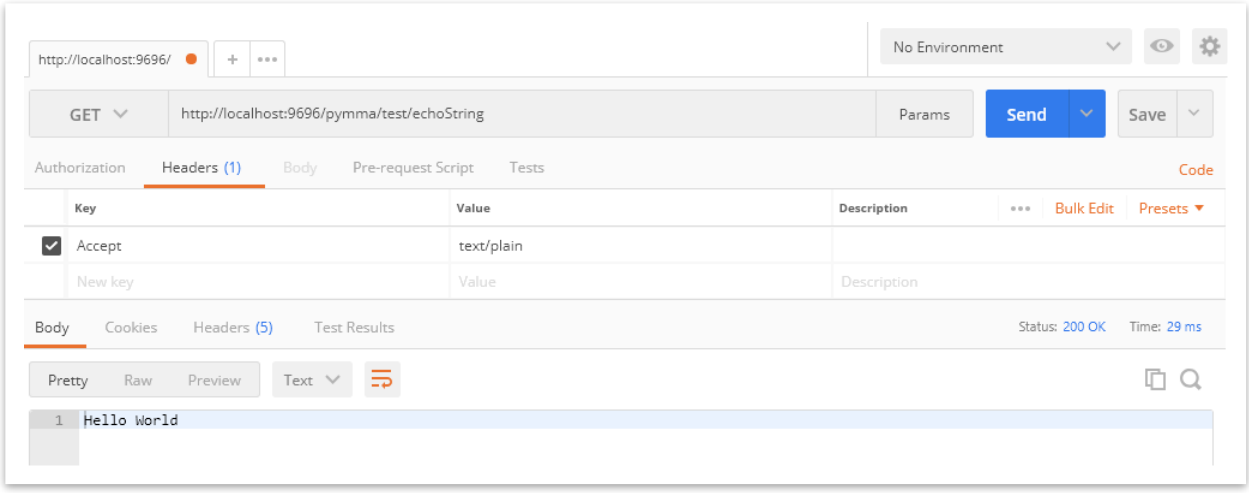
Drag and drop the BPEL project TestRestBCInbound in the lane Internal modules and build the composite application.



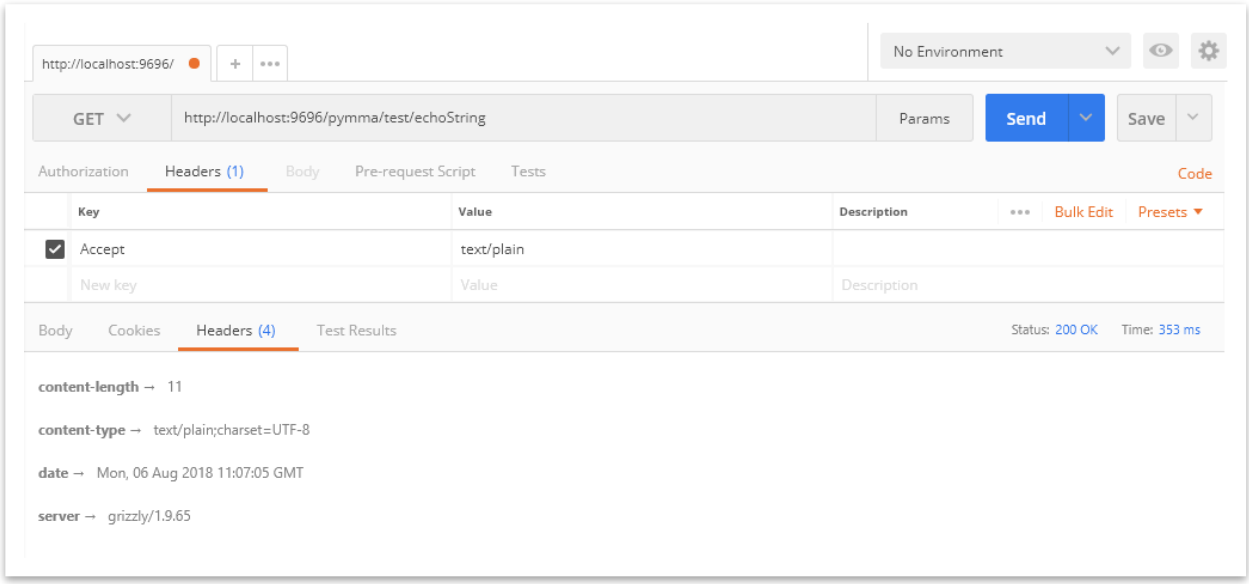
Check if the REST BC (> v3.1.x) is already installed and started on the OpenESB instance. Else install and start it. Then deploy the application.

Test echoString

Use any tool to test the services. In this document we use POSTMAN as REST client.



The Get operation echoString has no request body (payload) the header must specify that the client accepts “text/plain”. The response body is “Hello World” as expected.

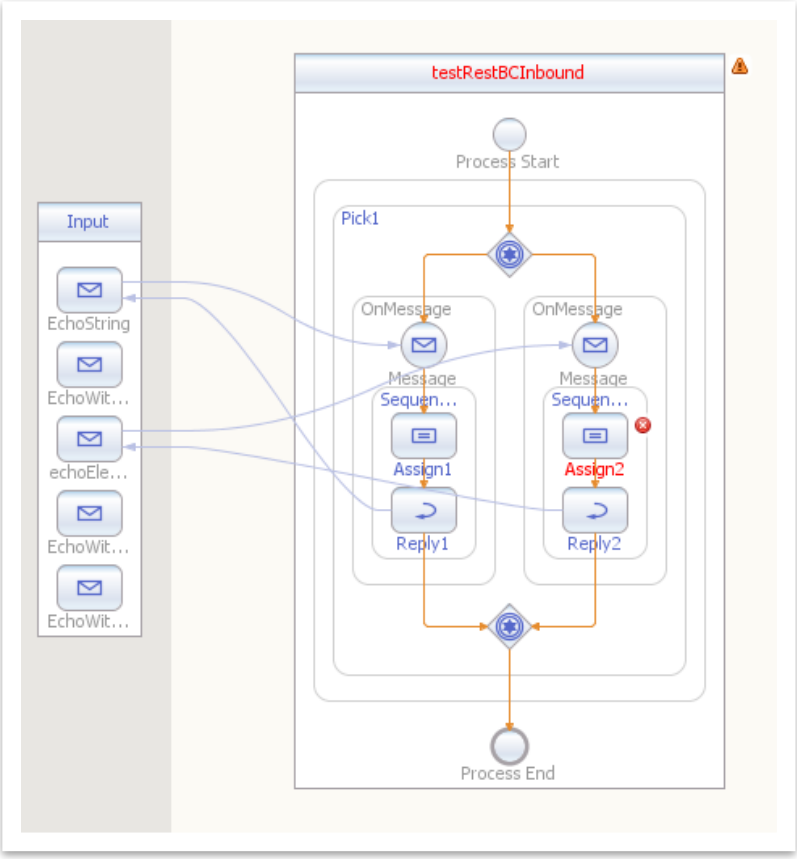


The header returned by OpenESB forces the charset to UTF-8.

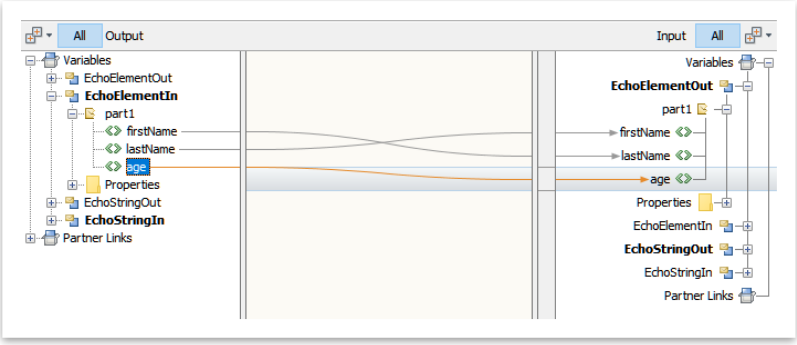
Operation echoElement

BPEL design

Add an onMessage activity to the Pick. Add an assign and a reply activities. Link the onMessage and the reply to the operation “echoElement”.



Set the Assign activity as follows:



Save the BPEL and redeploy the composite application.

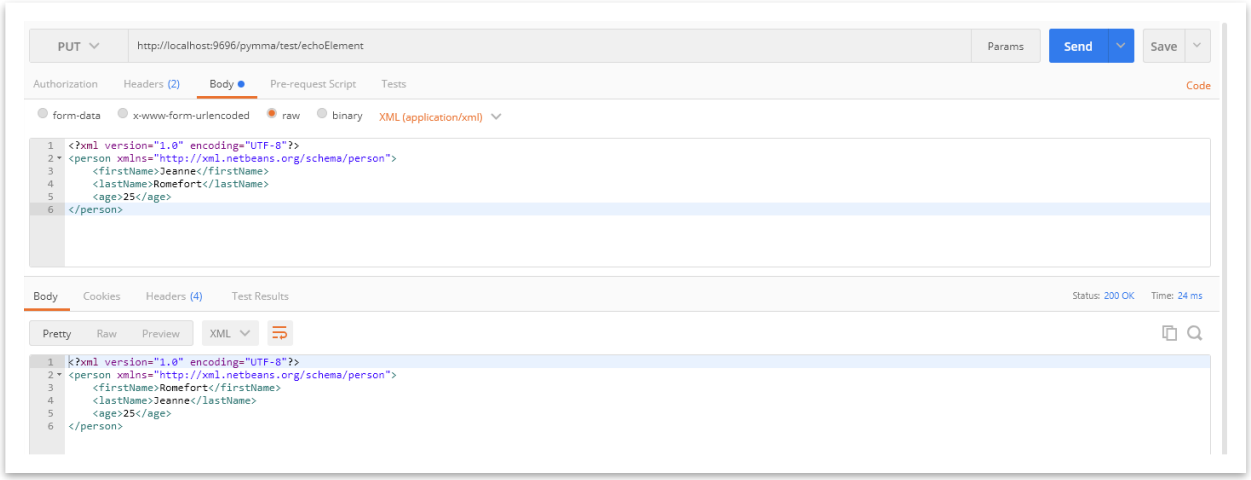
Test echoElement

To test the echoElement service you can use an XML or Json document as request payload.

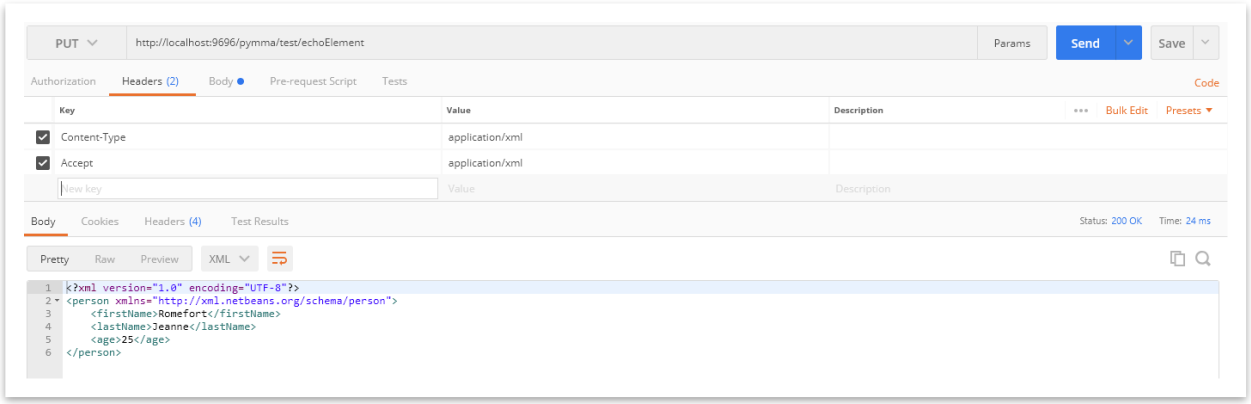
```
<?xml version="1.0" encoding="UTF-8"?>
<person xmlns="http://xml.netbeans.org/schema/person">
    <firstName>Romefort</firstName>
    <lastName>Jeanne</lastName>
    <age>25</age>
```

```
</person>
```

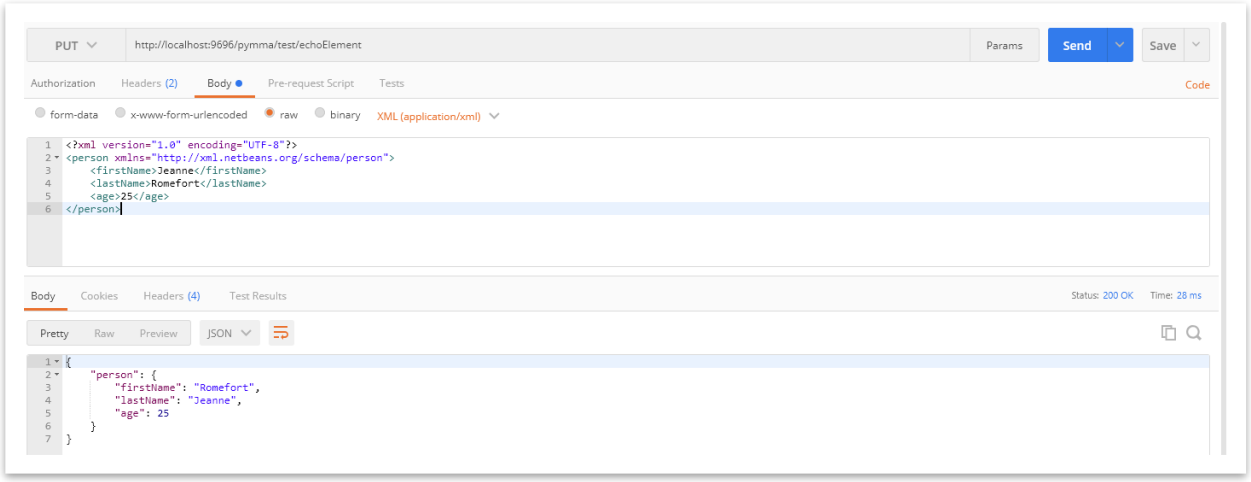
```
{
    "person": {
        "firstName": "Romefort",
        "lastName": "Jeanne",
        "age": 25
    }
}
```



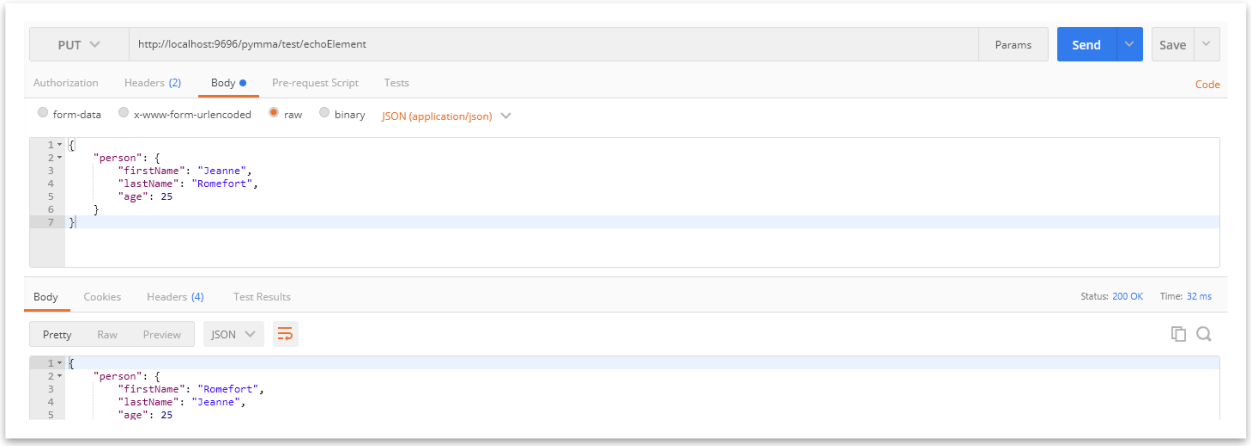
The first name and the last name have been switched. The content-type and accept-type are application/xml.



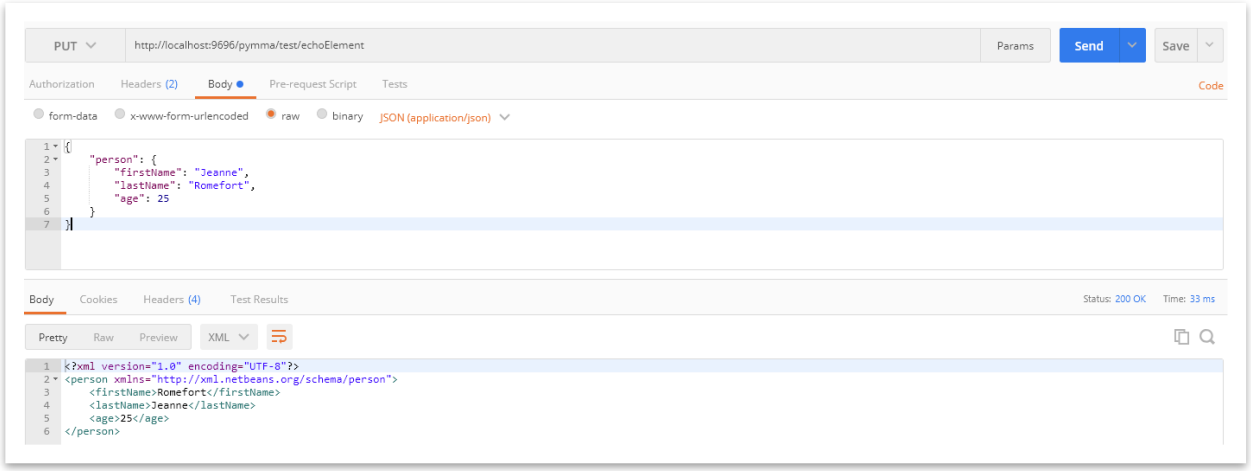
Let's change the accept-type to application/json.



Change the request body to a Json document.



Last, let's change the accept-type to application/xml.



Notice that the response has an XML namespace. The REST BC knows the message namespace defined in the WSDL and assign it to the request or the response payload.

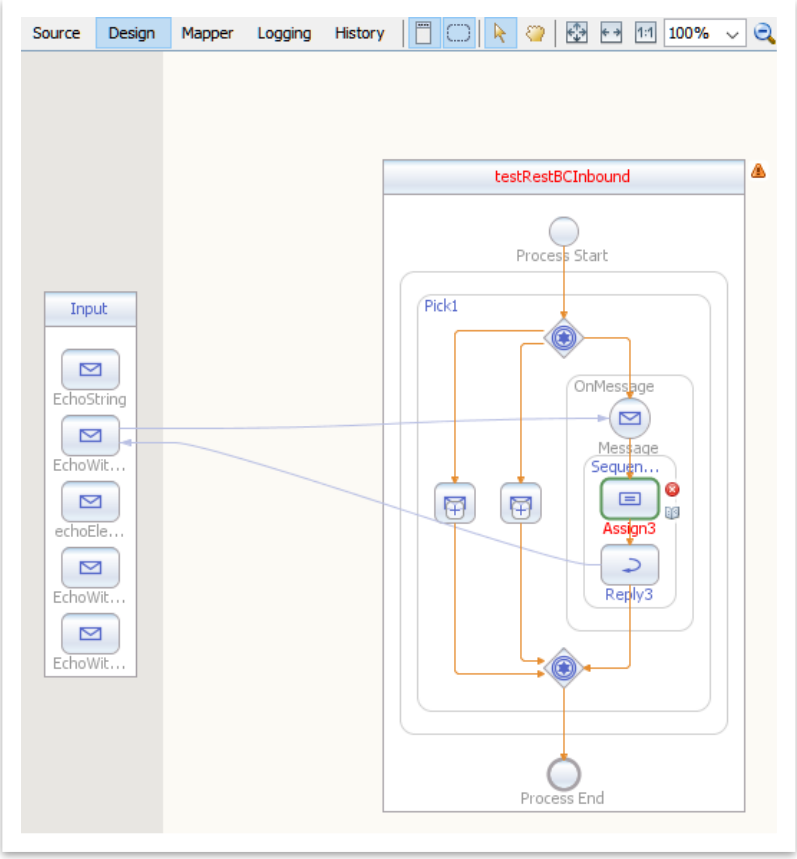
Operation echoWithParams

This operation details how to use query parameters in a business process. Since the query parameters are not defined in the message, but at the protocol level, we don't define them at the WSDL level. They must be listed in the REST API documentation.

In our example, we defined three query parameters: q-firstName, q-lastName, q-age.

BPEL design

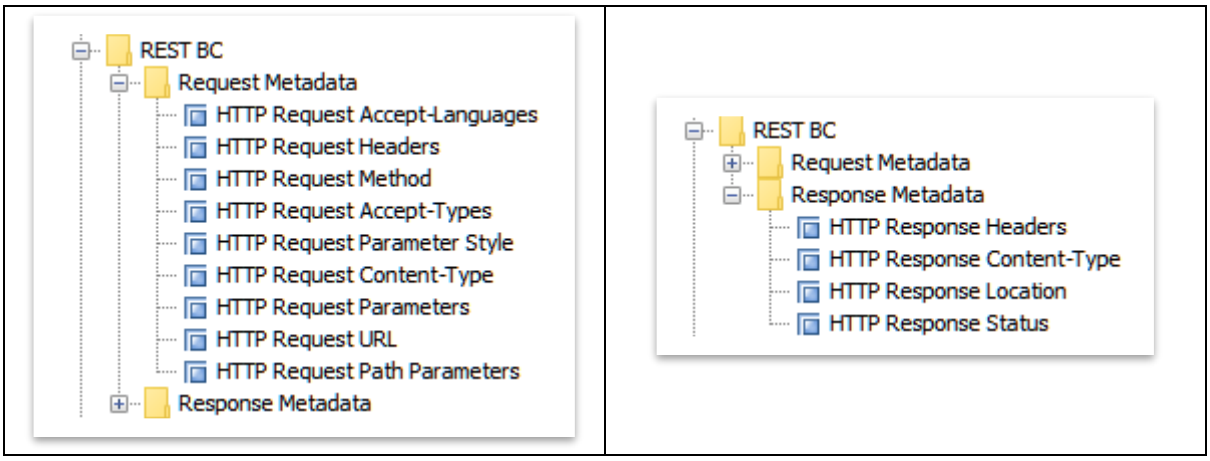
Add an onMessage activity to the Pick. Add an assign and a reply activities. Link the onMessage and the reply to the operation "echoWithParams".



Access to the HTTP Query parameters

NMProperties are properties associated with the Normalised messages exchanged between OpenESB components. When the rest BC receives an HTTP request, it copies the request parameters to the NM Properties. So, the other components such as the BPEL can access these values. The BPEL mapper provides easy access to these properties.

The pictures below list the properties generated by the REST BC and accessible by the BPEL



A complete description of the REST properties is given in a next chapter.

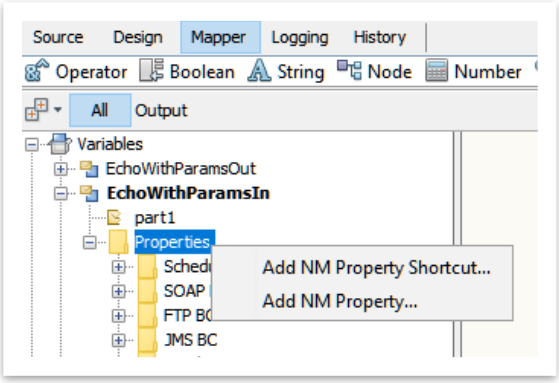
So, thanks to the NM properties, the Assign activity picks up the HTTP parameters and inject them in the process defined by the BPEL.

The NM Property Shortcut gives easy access to the HTTP query parameters. It the HTTP request URL is defined as follows: http:api.pymma.com/pymma/test/echoWithParams?q-firstName=Jeanne&q-lastName=Romefort&q-age=25, the BPEL Mapper provides access to these values through the shortcut **org.glassfish.openesb.rest.params.{name of the parameter}.**

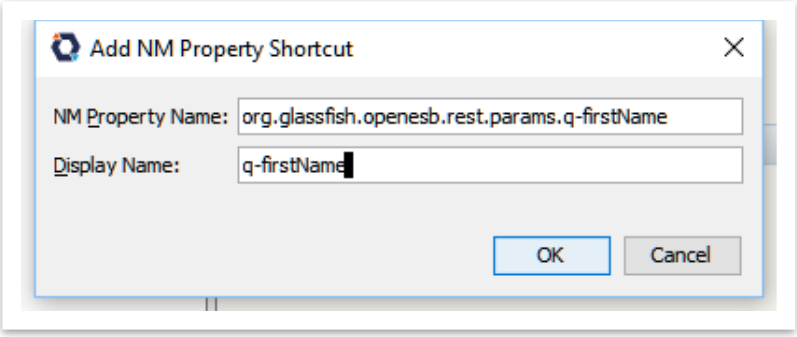
Query Parameter	Shortcut NM Property name
q-firstName	org.glassfish.openesb.rest.params.q-firstName
q-lastName	org.glassfish.openesb.rest.params.q-lastName
q-age	org.glassfish.openesb.rest.params.q-age

Let's create Shortcut access in the BPEL Mapper.

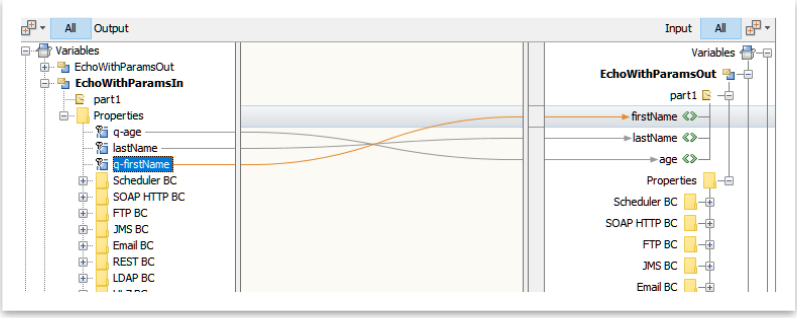
Select the inbound message (EchoWithParamsIn) expand it. Click right on properties.



Select "Add NM property ShortCut" and fill the form as follows:

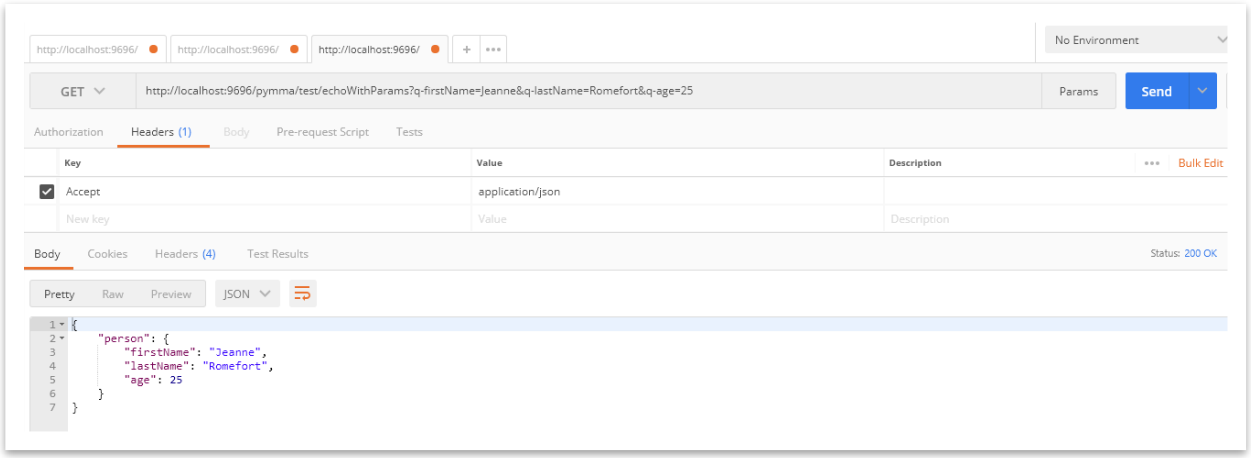


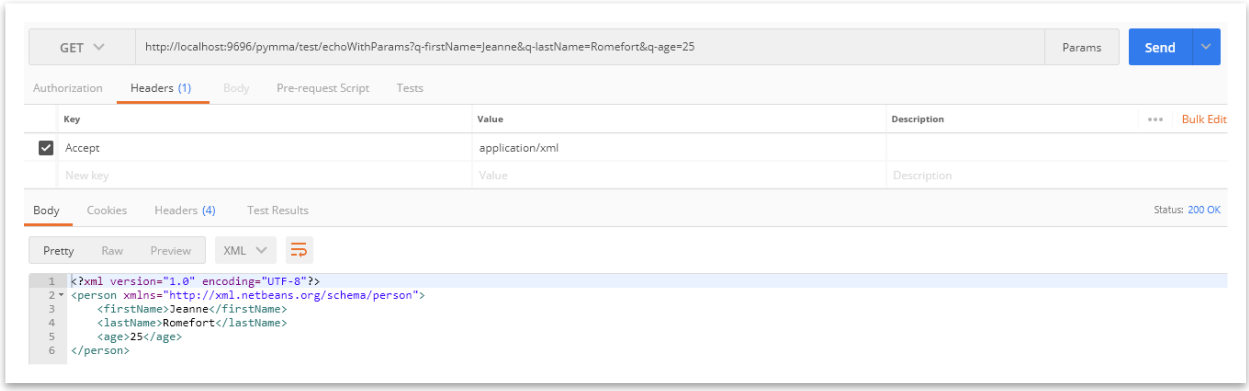
Add the properties q-lastName and q-age then map them to the outbound variable.



Save the BPEL and deploy the composite application.

Test echoWithParams





The query parameters are injected into the BPEL processes through the NM properties.

Operation echoWithPathParams

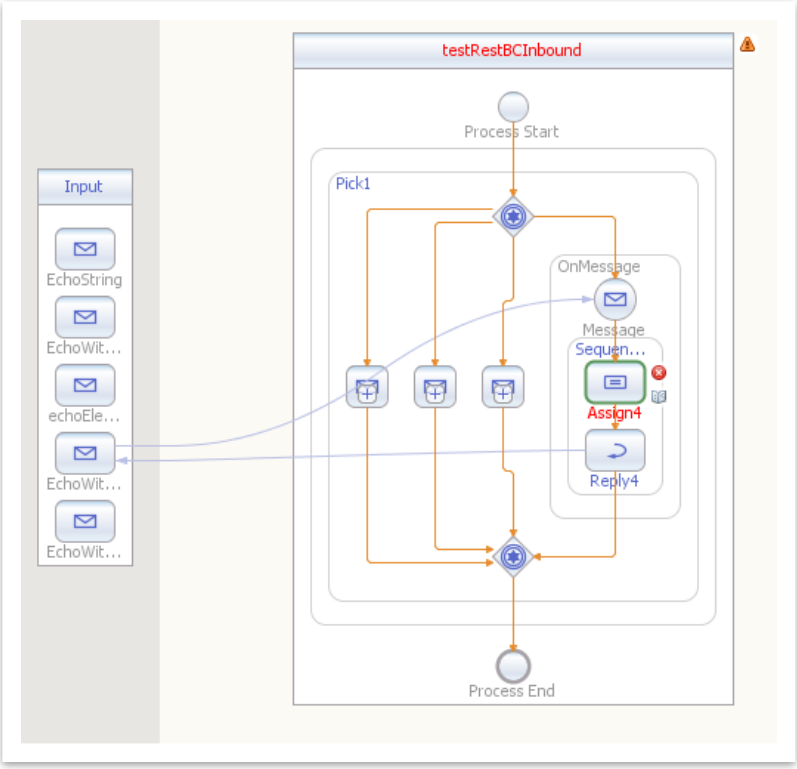
This operation details how to assign path parameters in a business process. Since the path parameters are not defined in the message, but at the protocol level, we don't define them at the WSDL level. They must be listed in the REST API documentation.

In our example, we defined three path parameter p-firstName, p-lastName, p-age. The service URL looks like http://api.pymma.com/pymma/test/echoWithPathParams/{p-firstName}/{p-lastName}/{p-age}.

The parameters in the URL are read by the REST BC and put in the NM Message properties.

BPEL design

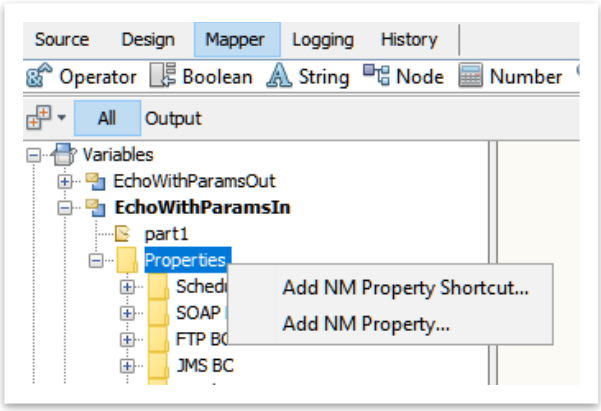
Add an onMessage activity to the Pick. Add an assign and a reply activity. Link the onMessage and the reply to the operation "echoWithPathParams".



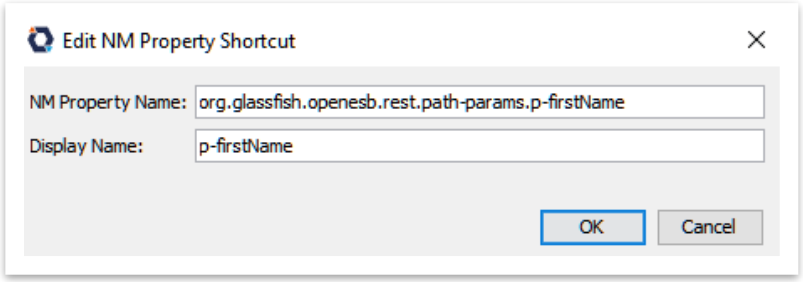
The REST BC captures the path variable of an URL and put them in the NM Properties. The NM Property Shortcut in the BPEL Mapper, gives easy access to the HTTP query parameters. If the HTTP request URL is defined as follows: http:api.pymma.com/pymma/test/echoWithPathParams/{p-firstName/{p-lastName}/{q-age}, the BPEL Mapper provides an access to these values through the shortcut **org.glassfish.openesb.rest.path-params.{name of the parameter}.**

Query Parameter	Shortcut NM Property name
q-firstName	org.glassfish.openesb.rest.path-params.p-firstName
q-lastName	org.glassfish.openesb.rest.path-params.p-lastName
q-age	org.glassfish.openesb.rest.path-params.p-age

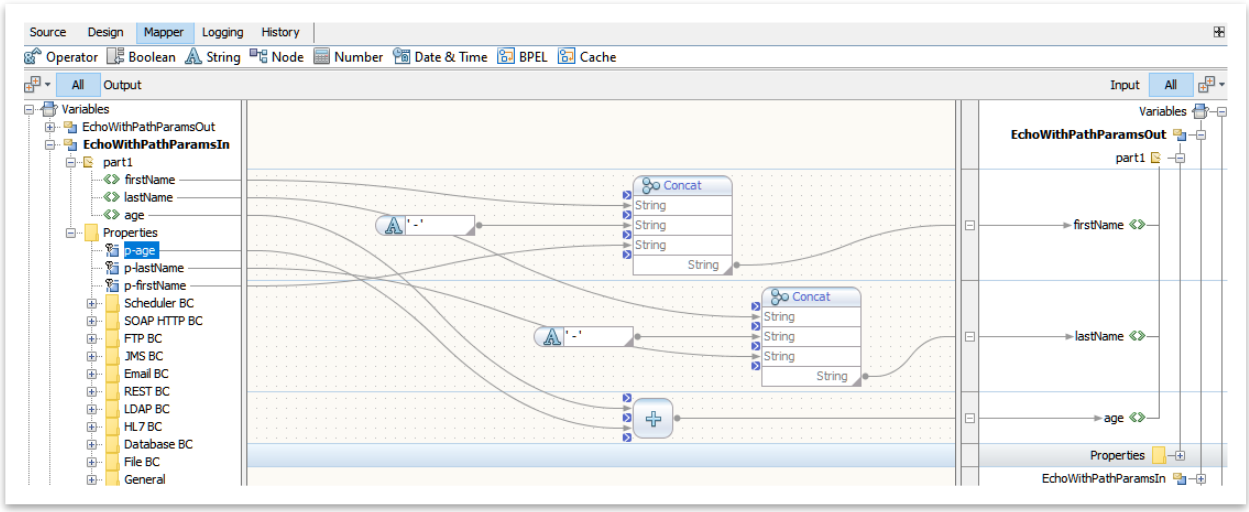
Let's create Shortcut access in the BPEL Mapper.



Select “Add NM property ShortCut” and fill the form as follows:

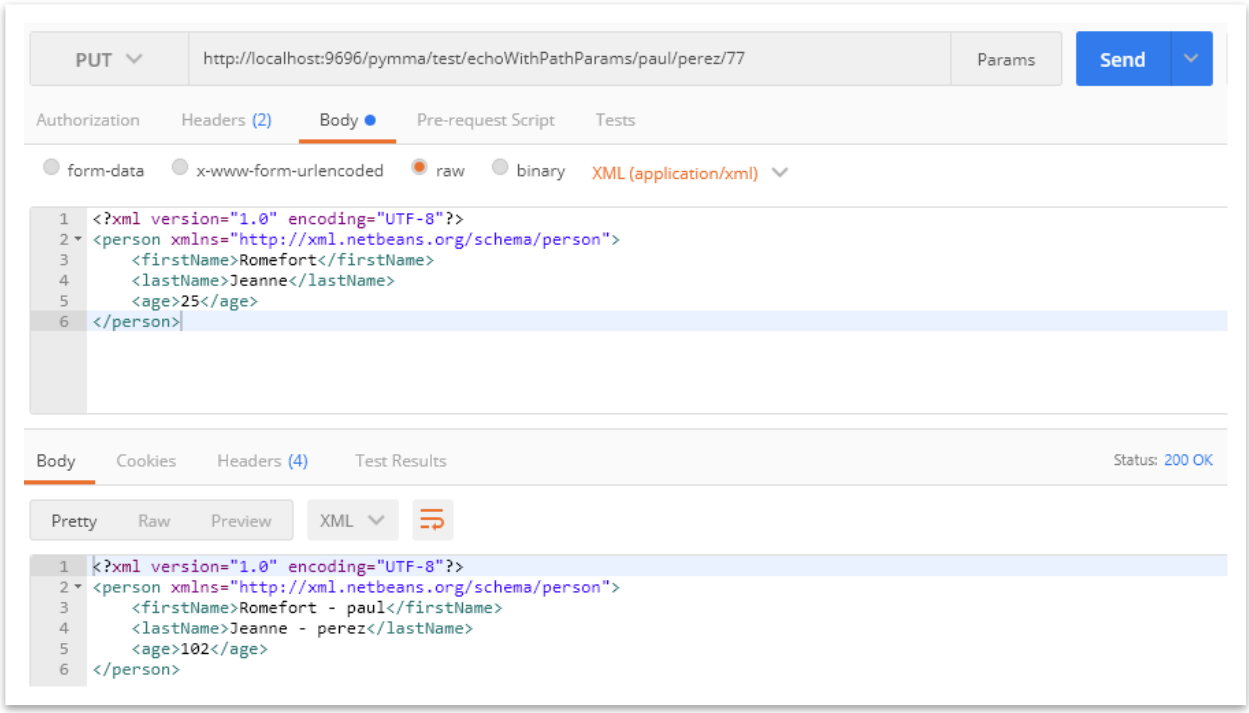


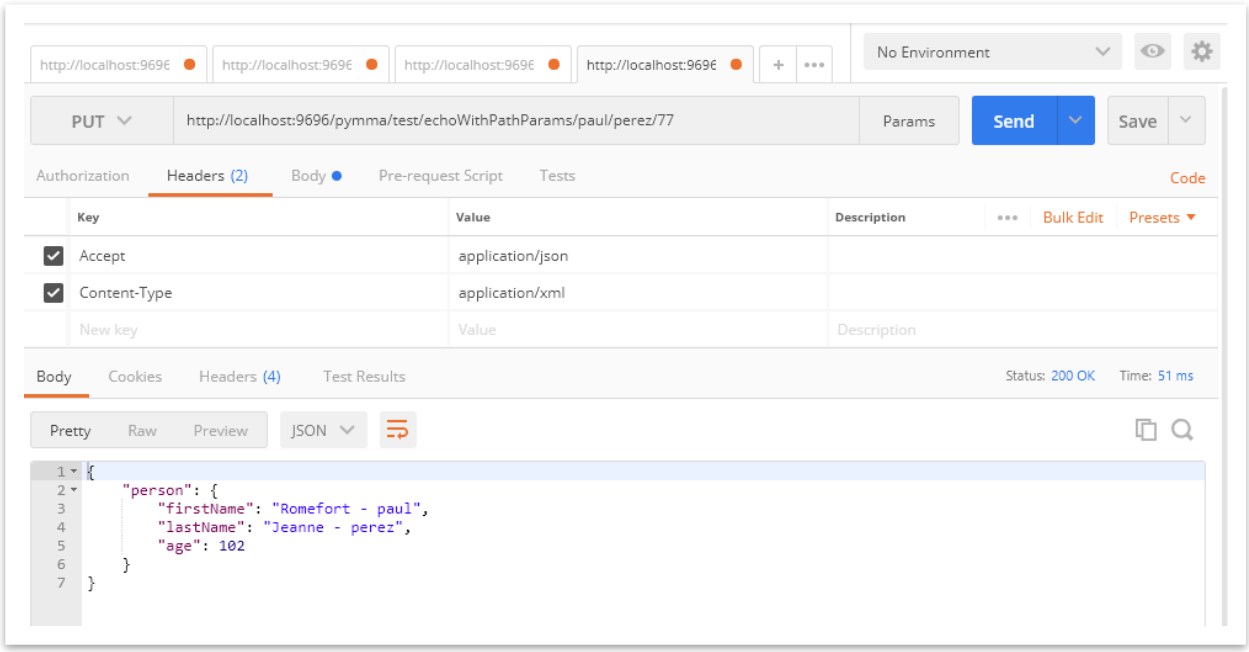
Add the properties p-lastName and p-age then map them to the outbound variable as follows:



Save the BPEL and deploy the composite application.

Test echoWithPathParams





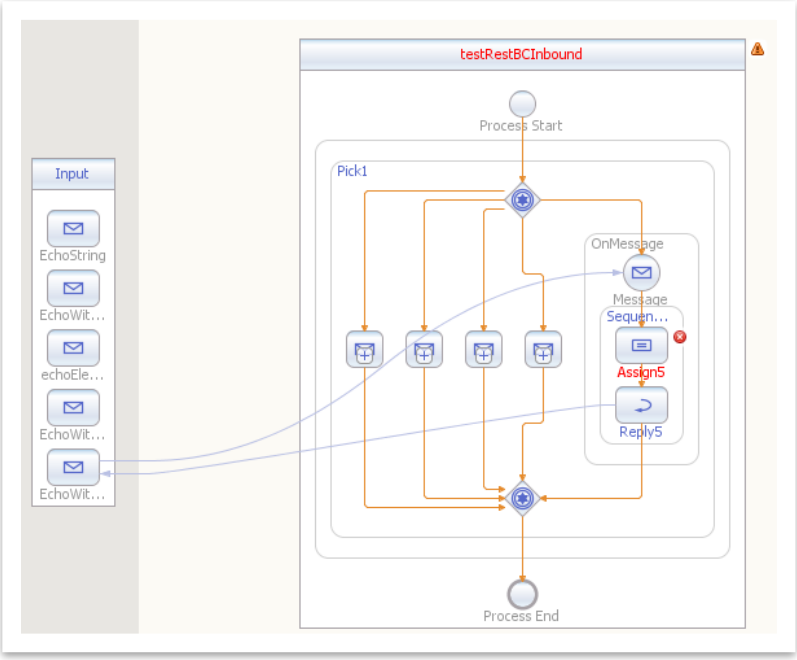
Operation echoWithHeaderParams

This operation details how to uses the parameters defined in the HTTP header and assign them in a business process. Since the header parameters are not Defined in the message, but at the protocol level, we don't define them at the WSDL level. They must be listed in the REST API documentation.

In our example, we defined three path parameter h-firstName, h-lastName, h-age. These parameters do not appear in the URL but in the HTTP header. They are read by the REST BC and put in the NM properties.

BPEL Design

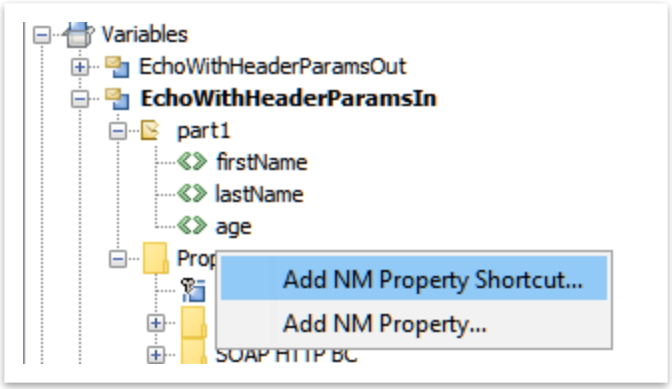
Add an onMessage activity to the Pick. Add an assign and a reply activity. Link the onMessage and the reply to the operation “echoWithHeaderParams”.

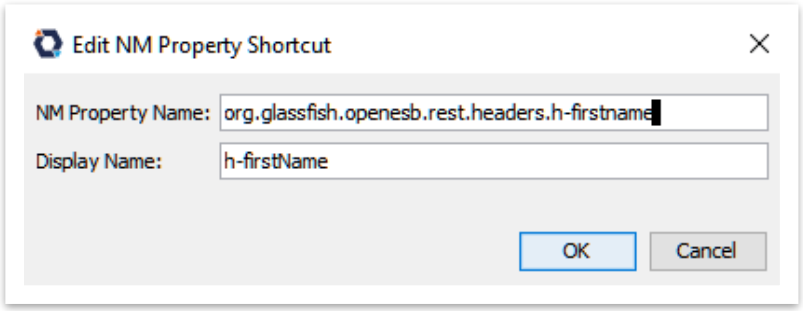


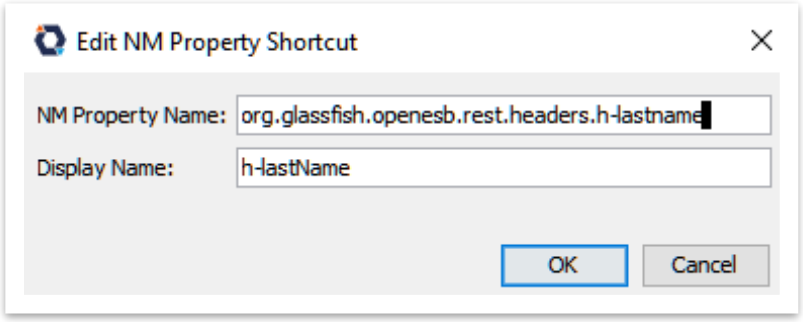
The REST BC captures the header parameters of a request and put them in the NM Properties. The NM Property Shortcut in the BPEL Mapper, gives easy access to the HTTP header parameters. The BPEL Mapper provides access to these values through the shortcut **org.glassfish.openesb.rest.headers.{name of the parameter}.**

Query Parameter	Shortcut NM Property name
h-firstName	org.glassfish.openesb.rest.headers.h-firstName
h-lastName	org.glassfish.openesb.rest.headers.h-lastName
h-age	org.glassfish.openesb.rest.headers.h-age

Let’s create Shortcut access for the three parameters in the BPEL Mapper.

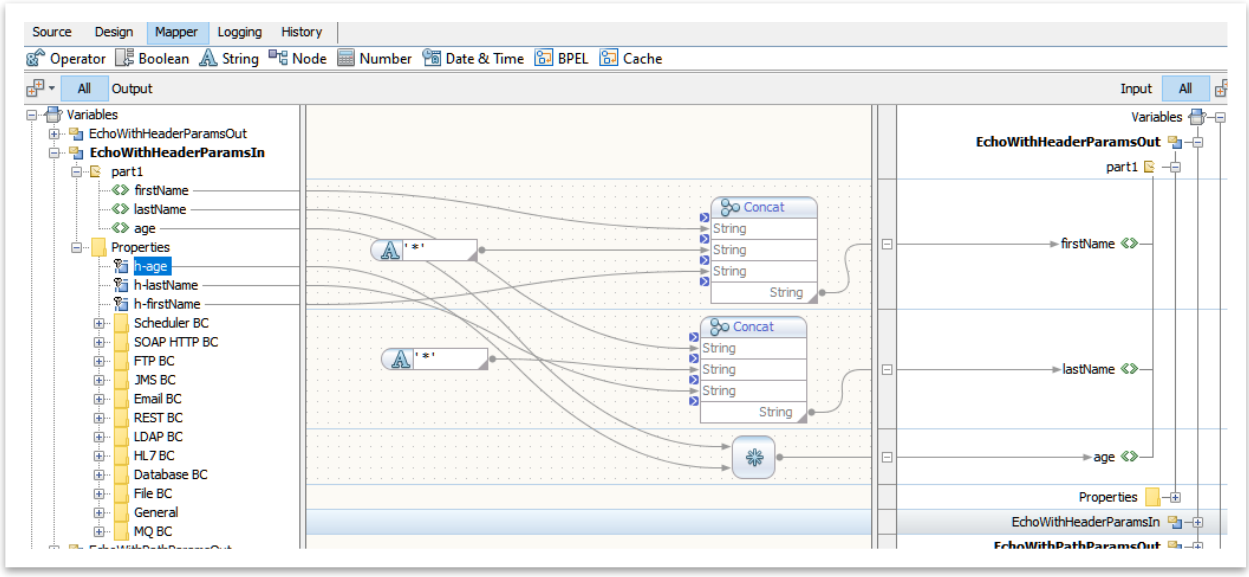






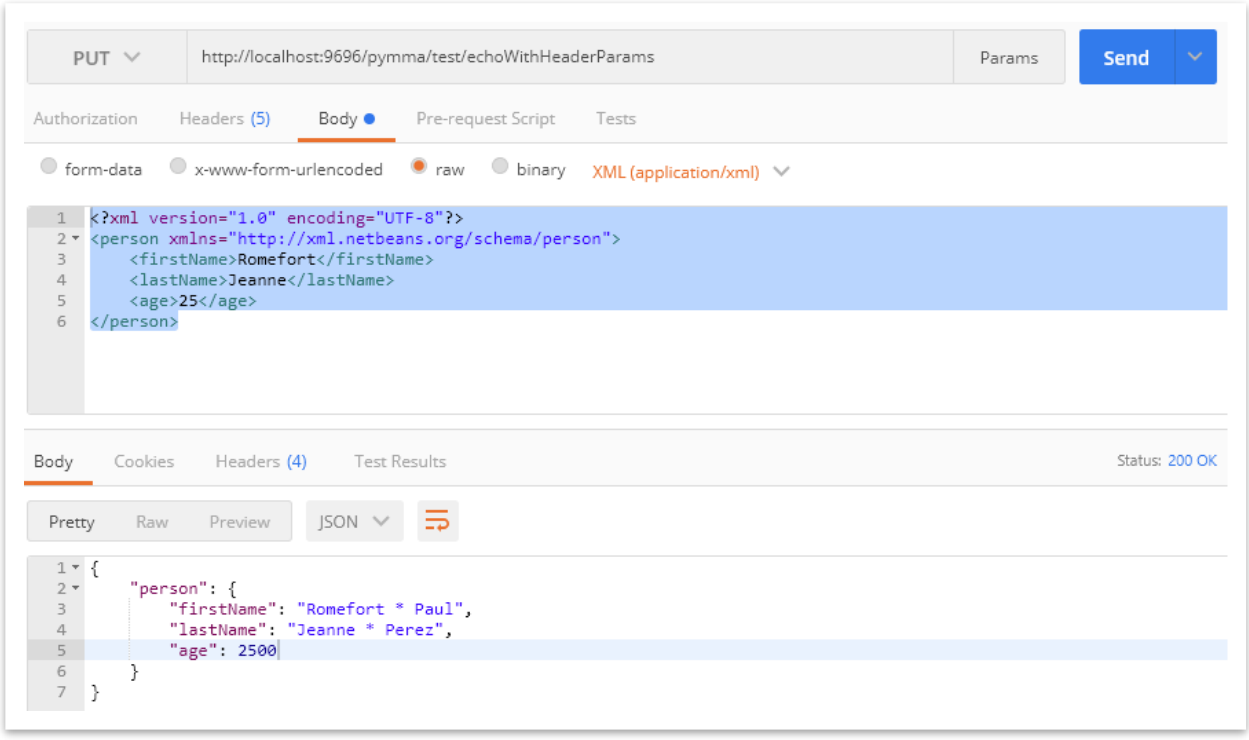
Please notice, that the properties h-fistname and h-lastname have no capital letters since by convention, the header parameters are set in lower case. So event if the client adds the property h-firstName in the header, expect to receive h-firstname.

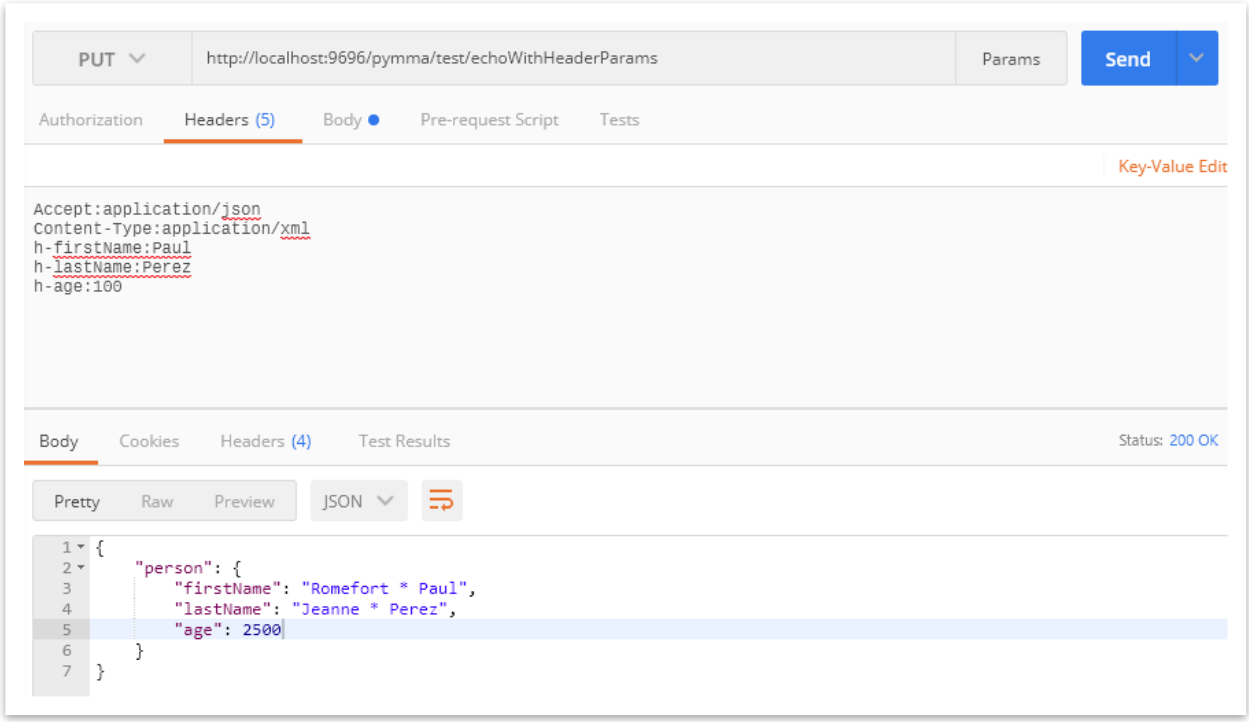
Add the properties h-lastname and h-age then map them to the outbound variable as follows:



Save the BPEL and deploy the composite application.

Test echoWithHeaderParameters





Through this example, you learn how to use the path, query and header parameters defined by HTTP, in the inbound configurations. Outbound configurations use the same process to define the path, query and header parameters.

REST BC Outbound example

The Rest BC Outbound process is very similar to the Inbound process. Query, Path and Header parameters are set in the same way than for the inbound process. So, we will not come back on the way to use parameters in the outbound configuration.

Nevertheless, the Outbound process requires more flexibility and work, because in many cases, the external services invoked by OpenESB, do not observe the inbound and outbound messages definition best practices and complexifies the exchanges between the REST BC and the external service implementation. The most common issues we face when invoking a REST service is the absence of a formal definition of the exchange with the external partner and the lake of message root in the JSON documents.

Often REST API projects have a bottom-up approach and don't use a clear contractual definition of the services and publish a posteriori, a "documentation" automatically generated from the code. There is neither message structure agreement nor a structure validation as it exists internally in OpenESB.

In the OpenESB Enterprise Edition 3.1.2 we added new features to fix the lake of root in a message by using the REST BC WSDL properties. These new features don't solve all the issue we met when converting XML to JSON to XML, but a large part of our tests provided good results during the conversion. In the future REST BC version, we will try to improve the conversion with the new libraries available.

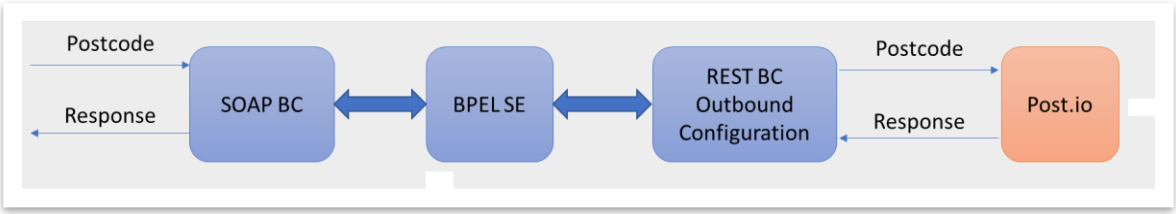
API Post Code

For an example, we use a service provided by https://codeposts.io We want to invoke the service: api.postcodes.io/postcodes/{postcode} with the Get Operation. This service provides administrative information on the UK postcode. With the postcode N3 2JS as path parameter, the service replies information of the UK postcode:


```
{
  "status": 200,
  "result": {
    "postcode": "N3 2JS",
    "quality": 1,
    "eastings": 525315,
    "northings": 190390,
    "country": "England",
    "nhs_ha": "London",
    "longitude": -0.192121,
    "latitude": 51.598422,
    "european_electoral_region": "London",
    "primary_care_trust": "Barnet",
    "region": "London",
    "lsoa": "Barnet 028A",
    "msoa": "Barnet 028",
    "incode": "2JS",
    "outcode": "N3",
    "parliamentary_constituency": "Finchley and Golders Green",
    "admin_district": "Barnet",
    "parish": "Barnet, unparished area",
    "admin_county": null,
    "admin_ward": "Finchley Church End",
    "ccg": "NHS Barnet",
    "nuts": "Barnet",
    "codes": {
      "admin_district": "E09000003",
      "admin_county": "E99999999",
      "admin_ward": "E05000051",
      "parish": "E43000193",
      "parliamentary_constituency": "E14000703",
      "ccg": "E38000005",
      "nuts": "UKI71"
    }
  }
}
```


Notice that in the response, the JSON element result and status are at the same level, so there are two roots defined in the JSON document.

We want to implement a process as defined in the picture below.



To accept and process the response given by Post.io, at the design time, we must define internally an XML schema that matches the JSON structure used for the operation "api.postcodes.io/postcodes/{postcode}".

This XML Schema can be defined manually, but when the JSON structure becomes complex, the schema definition becomes difficult and error-prone. Automatic conversion is recommended. Each developer has her/his own trick to convert an XSD from a JSON document. The next paragraph provides a semi-automatic way to get quickly an XML Schema.

JSON document to XML Schema conversion

Online conversion step 01: JSON to XML

To convert the JSON to XML, we use an online tool that does the conversion: https://www.freeformatter.com/json-to-xml-converter.html. Any other conversion tool can be used for this step.

The converter provides the following XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <result>
      <admin_county null="true" />
      <admin_district>Barnet</admin_district>
      <admin_ward>Finchley Church End</admin_ward>
      <ccg>NHS Barnet</ccg>
      <codes>
         <admin_county>E99999999</admin_county>
         <admin_district>E09000003</admin_district>
         <admin_ward>E05000051</admin_ward>
         <ccg>E38000005</ccg>
         <nuts>UKI71</nuts>
         <parish>E43000193</parish>
         <parliamentary_constituency>E14000703</parliamentary_constituency>
      </codes>
      <country>England</country>
      <eastings>525096</eastings>
      <european_electoral_region>London</european_electoral_region>
      <incode>3JB</incode>
      <latitude>51.59243</latitude>
      <longitude>-0.19552</longitude>
      <lsoa>Barnet 028D</lsoa>
```

```
      <msoa>Barnet 028</msoa>
      <nhs_ha>London</nhs_ha>
      <northings>189718</northings>
      <nuts>Barnet</nuts>
      <outcode>N3</outcode>
      <parish>Barnet, unparished area</parish>
      <parliamentary_constituency>Finchley and Golders
Green</parliamentary_constituency>
      <postcode>N3 3JB</postcode>
      <primary_care_trust>Barnet</primary_care_trust>
      <quality>1</quality>
      <region>London</region>
   </result>
   <status>200</status>
</root>
```

Please notice that the Converter adds the root element in the XML document.

Online conversion step: 02: Create the schema

Then we use the same online site to generate an XSD: https://www.freeformatter.com/xsd-generator.html. Freeformatter generates the XSD below:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="result">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="admin_county">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute type="xs:string" name="null"/>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="admin_district"/>
              <xs:element type="xs:string" name="admin_ward"/>
              <xs:element type="xs:string" name="ccg"/>
              <xs:element name="codes">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="admin_county"/>
                    <xs:element type="xs:string" name="admin_district"/>
                    <xs:element type="xs:string" name="admin_ward"/>
                    <xs:element type="xs:string" name="ccg"/>
                    <xs:element type="xs:string" name="nuts"/>
                    <xs:element type="xs:string" name="parish"/>
                    <xs:element type="xs:string"
name="parliamentary_constituency"/>
                  </xs:sequence>
```

```
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="country"/>
              <xs:element type="xs:int" name="eastings"/>
              <xs:element type="xs:string" name="european_electoral_region"/>
              <xs:element type="xs:string" name="incode"/>
              <xs:element type="xs:float" name="latitude"/>
              <xs:element type="xs:float" name="longitude"/>
              <xs:element type="xs:string" name="lsoa"/>
              <xs:element type="xs:string" name="msoa"/>
              <xs:element type="xs:string" name="nhs_ha"/>
              <xs:element type="xs:int" name="northings"/>
              <xs:element type="xs:string" name="nuts"/>
              <xs:element type="xs:string" name="outcode"/>
              <xs:element type="xs:string" name="parish"/>
              <xs:element type="xs:string"
name="parliamentary_constituency"/>
              <xs:element type="xs:string" name="postcode"/>
              <xs:element type="xs:string" name="primary_care_trust"/>
              <xs:element type="xs:byte" name="quality"/>
              <xs:element type="xs:string" name="region"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element type="xs:short" name="status"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Then we modify the XML schema header to add a namespace.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://xml.netbeans.org/schema/postCode"
           xmlns:tns="http://xml.netbeans.org/schema/postCode"
           elementFormDefault="qualified">
    <xs:element name="root">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="result">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="admin_county">
                                <xs:complexType>
                                    <xs:simpleContent>
                                        <xs:extension base="xs:string">
                                            <xs:attribute type="xs:string"
name="null"/>
                                        </xs:extension>
                                    </xs:simpleContent>
                                </xs:complexType>
                            </xs:element>
                            <xs:element type="xs:string"
name="admin_district"/>
                            <xs:element type="xs:string" name="admin_ward"/>
```

```
                            <xs:element type="xs:string" name="ccg"/>
                            <xs:element name="codes">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element type="xs:string"
name="admin_county"/>
                                        <xs:element type="xs:string"
name="admin_district"/>
                                        <xs:element type="xs:string"
name="admin_ward"/>
                                        <xs:element type="xs:string"
name="ccg"/>
                                        <xs:element type="xs:string"
name="nuts"/>
                                        <xs:element type="xs:string"
name="parish"/>
                                        <xs:element type="xs:string"
name="parliamentary_constituency"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element type="xs:string" name="country"/>
                            <xs:element type="xs:int" name="eastings"/>
                            <xs:element type="xs:string"
name="european_electoral_region"/>
                            <xs:element type="xs:string" name="incode"/>
                            <xs:element type="xs:float" name="latitude"/>
                            <xs:element type="xs:float" name="longitude"/>
                            <xs:element type="xs:string" name="lsoa"/>
                            <xs:element type="xs:string" name="msoa"/>
                            <xs:element type="xs:string" name="nhs_ha"/>
                            <xs:element type="xs:int" name="northings"/>
                            <xs:element type="xs:string" name="nuts"/>
                            <xs:element type="xs:string" name="outcode"/>
                            <xs:element type="xs:string" name="parish"/>
                            <xs:element type="xs:string"
name="parliamentary_constituency"/>
                            <xs:element type="xs:string" name="postcode"/>
                            <xs:element type="xs:string"
name="primary_care_trust"/>
                            <xs:element type="xs:byte" name="quality"/>
                            <xs:element type="xs:string" name="region"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element type="xs:short" name="status"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Create another Schema to enter the postcode parameter. Name it inputTest.xsd and use the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/inputTest"
    xmlns:tns="http://xml.netbeans.org/schema/inputTest"
    elementFormDefault="qualified">
    <xsd:complexType name="InputRequestCT">
        <xsd:sequence>
            <xsd:element name="PostCodeValue"
type="xsd:string"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="InputRequest" type="tns:InputRequestCT"></xsd:element>
</xsd:schema>
```

JSON document to XML schema alternative

OpenESB XSD generator

To help the use of JSON encoder along, OpenESB Enterprise Edition embedded a Java tool generating an XSD file from a JSON file.

How to generate an XSD file?

In the shared library encoderlib-installer-xxx.jar, the Aymeric-json.jar contains a class com.pymma.encoder.json.JSONtoXSD.

This main class is used to generate an XML document and an XML Schema from a JSON message. The main method has 5 parameters

JSON2XSD parameters		
Parameter	What is it?	Comment
Parameter 01	JSON file	This is the name of the JSON file to use to generate the XSD
Parameter 02	Element root	The root element that will be added in the XML file and in the XML Schema. Avoid the element or attribute names used in the XML document or in the JSON message
Parameter 03	Namespace	The namespace of the generated XML document
Parameter 04	XML file name	Name of the XML document generated as an example
Parameter 05	XSD file name	Name of the XSD to be used in your project

To generate an XSD file, use the Java command in a console:

java -cp./* com.pymma.encoder.json.JSONtoXSD person.json MyRoot http://pymma.com/testJson person.xml personTemp.xsd

The jar files required to run JSON2XSD.class are:

- aymeric-json.jar
- encoderfwrt.jar

- staxon.jar
- wiztools-commons-lib.jar
- xbean.jar
- xom.jar
- xsd-gen.jar

These jars can be found in the shared library encoderlib-installer-xxx.jar provided with the Enterprise Edition of OpenESB.

Example of outbound configuration

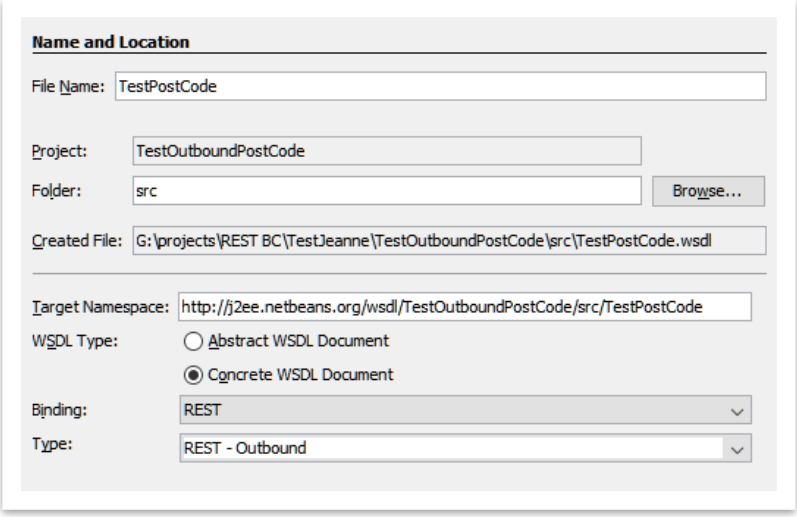
Create a BPEL project

Create a BPEL project and name it “TestOutboundPostCode”.

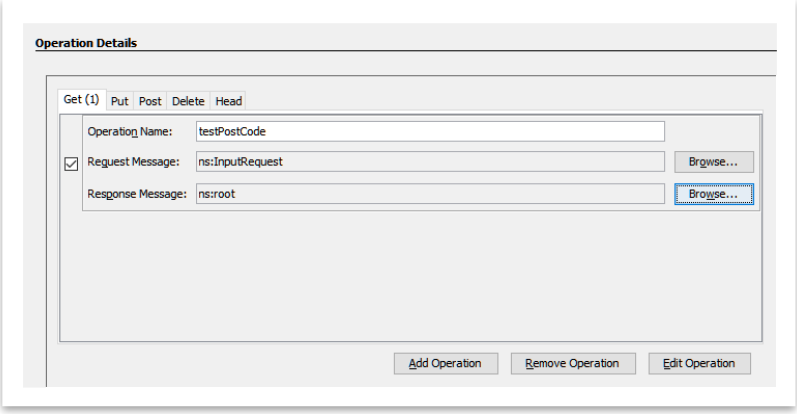
Create the WSDLs

REST WSDL Outbound

Create a REST WSDL Outbound and name it TestPostCode.

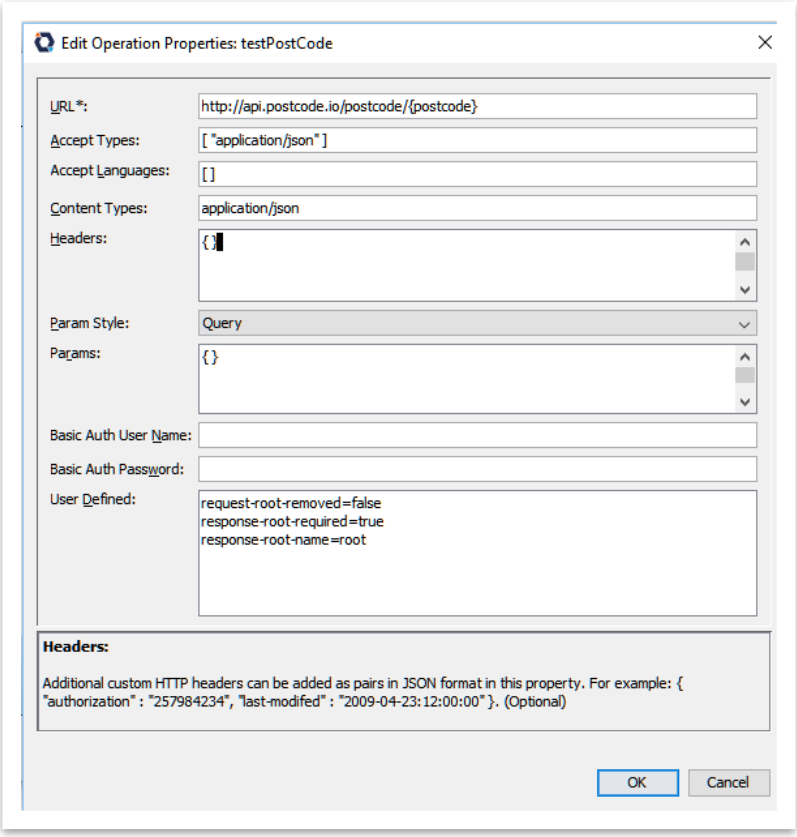


Create a Get operation and name it testPostCode



Use the InputRequest element as a request message and the root element as a response message.

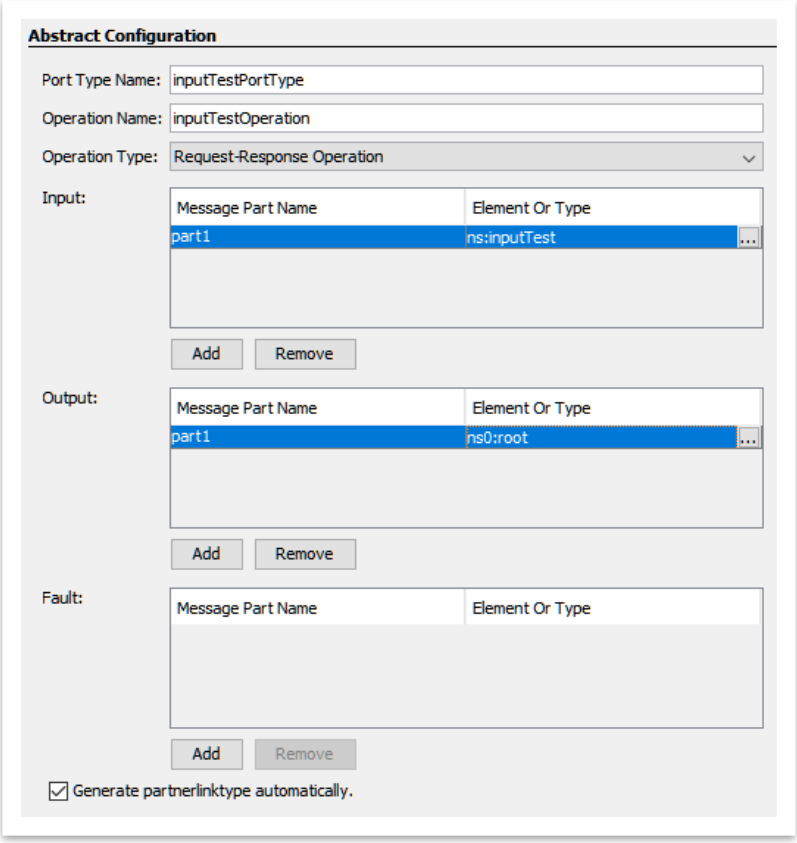
Set up the operation as follows:



Notice the parameters: request-root-removed=false, response-root-required=true and response-root-name=root. For more information, see above.

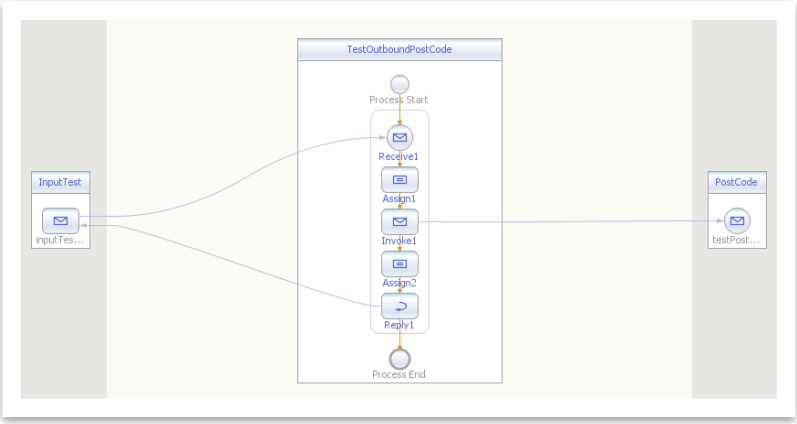
Save the WSDL

Create another abstract WSDL name it inputTest.wsdl with InputTest as input and the element root as output.

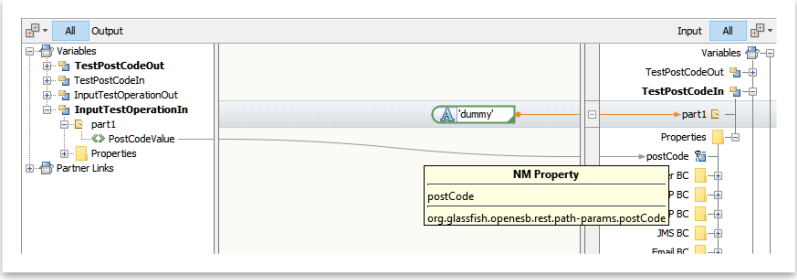


Create a BPEL

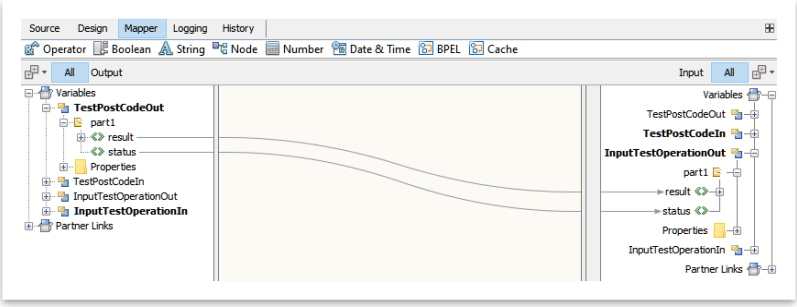
Create a BPEL name it testOutboundPostCode. Use the InputTest.wsdl as input and TestPostCode as output. Design it as follows:



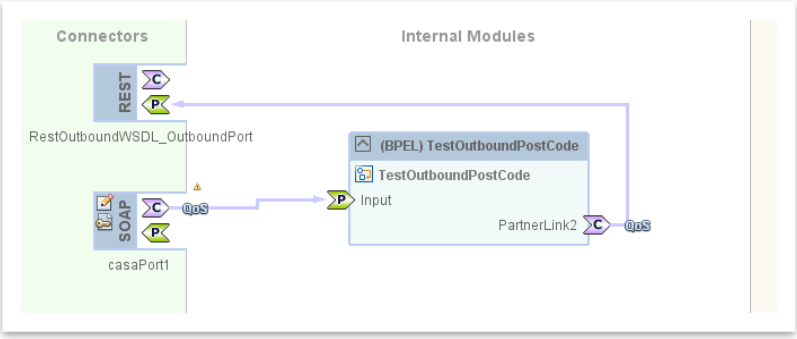
The first assign activity sets up the path-parameters postcode. Add a value “dummy” to part1 to avoid null messages



The second assign returns postcode.io answer to the SOAP client.



Composite application



Add a Soap connector to the input endpoint, then deploy the application.

Test postcode.io service

Create a test for the SOAP Input.

```
<soapenv:Envelope
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:inp="http://xml.netbeans.org/schema/inputTest">
  <soapenv:Body>
    <inp:InputRequest>
      <inp:PostCodeValue>N3 2JS</inp:PostCodeValue>
    </inp:InputRequest>
```

```
  </soapenv:Body>
</soapenv:Envelope>
```

The response from postcodes.io is

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <root xmlns="http://xml.netbeans.org/schema/postCode"
xmlns:msgns="http://j2ee.netbeans.org/wsdl/TestOutboundPostCode/src/inputTest
">
      <result
xmlns:msgns="http://j2ee.netbeans.org/wsdl/TestOutboundPostCode/src/TestPostC
ode"
xmlns:ns0="http://j2ee.netbeans.org/wsdl/TestOutboundPostCode/src/inputTest">
        <admin_county>null</admin_county>
        <admin_district>Barnet</admin_district>
        <admin_ward>Finchley Church End</admin_ward>
        <ccg>NHS Barnet</ccg>
        <codes>
          <admin_county>E99999999</admin_county>
          <admin_district>E09000003</admin_district>
          <admin_ward>E05000051</admin_ward>
          <ccg>E38000005</ccg>
          <nuts>UKI71</nuts>
          <parish>E43000193</parish>
          <parliamentary_constituency>E14000703</parliamentary_constituency>
        </codes>
        <country>England</country>
        <eastings>525315</eastings>
        <european_electoral_region>London</european_electoral_region>
        <incode>2JS</incode>
        <latitude>51.598422</latitude>
        <longitude>-0.192121</longitude>
        <lsoa>Barnet 028A</lsoa>
        <msoa>Barnet 028</msoa>
        <nhs_ha>London</nhs_ha>
        <northings>190390</northings>
        <nuts>Barnet</nuts>
        <outcode>N3</outcode>
        <parish>Barnet, unparished area</parish>
        <parliamentary_constituency>Finchley and Golders
Green</parliamentary_constituency>
        <postcode>N3 2JS</postcode>
        <primary_care_trust>Barnet</primary_care_trust>
        <quality>1</quality>
        <region>London</region>
      </result>
      <status>200</status>
    </root>
  </SOAP-ENV:Body>
```

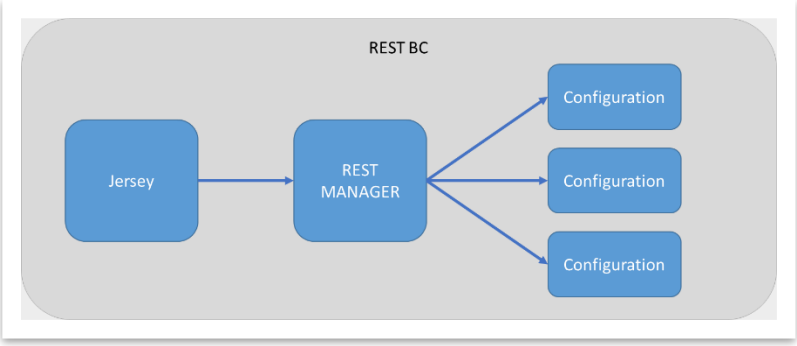
```
</SOAP-ENV:Envelope>
```

Fault management

REST fault management is more complex than for the other protocols, since for a part, it uses the HTTP codes as business faults and for the other part, the faults defined in the business processes. To get a good understanding of the REST BC Error, you must know a bit more about the component architecture.

Fault created by a wrong request

The picture below shows a simple view of the inbound part of the REST BC architecture.



When a message is sent to the REST BC, Jersey library analyses the request and transforms it into many parameters:

- Listener name = String
- Header = HttpHeader
- Method = String
- Path = String

Then, the REST manager search for a configuration that matches these parameters. When you deploy a REST WSDL, the REST BC creates a new endpoint with the configuration defined in the WSDL. An inbound configuration defines a listener, a header, a method and a path. So, an endpoint is defined by these same parameters.

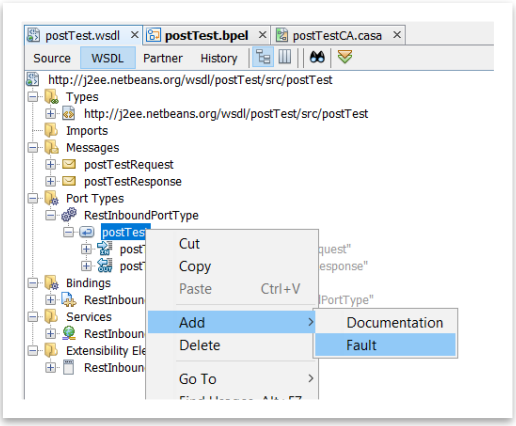
If a request does not match with a REST endpoint deployed on the OpenESB instance, the REST manager returns a message with an HTTP code 4xx (Whatever the processes behind the REST BC).

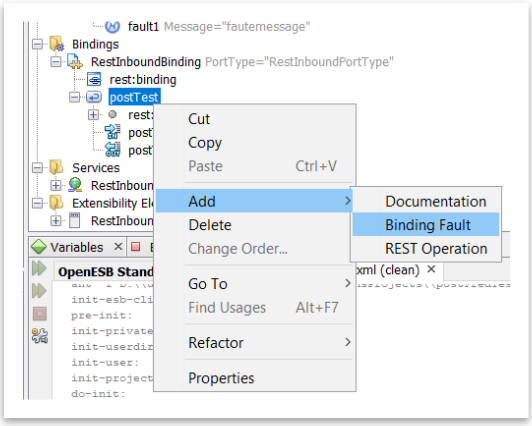
Error description	Code returned
Wrong path	Error code 404
Method not implemented	Error code 405
Content Type empty when content lengh >0	Error code 417
Unsupported media type	Error code 415

Type generated not accepted by the consumer	Error code 406

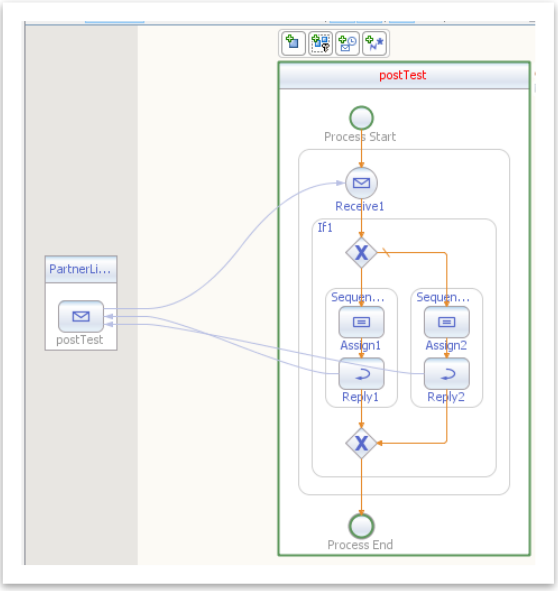
Fault in the process

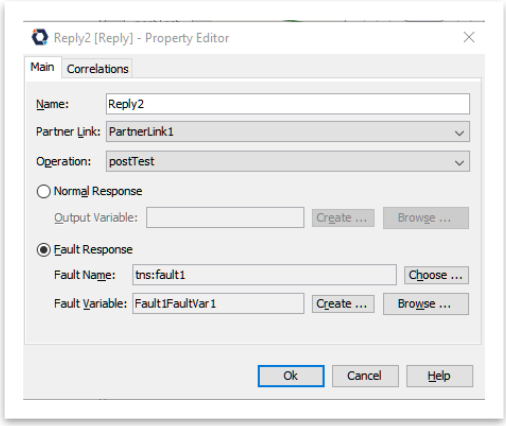
When an error occurs in a process, it must be returned to the REST BC. To do it the REST WSDL must define a fault in the Port Type and then in the Binding.

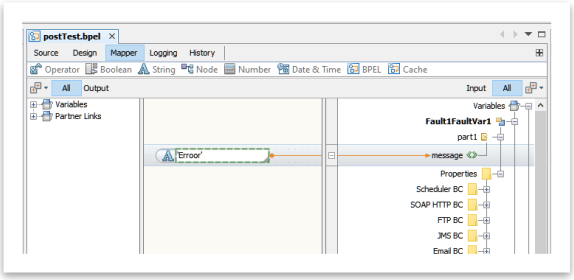




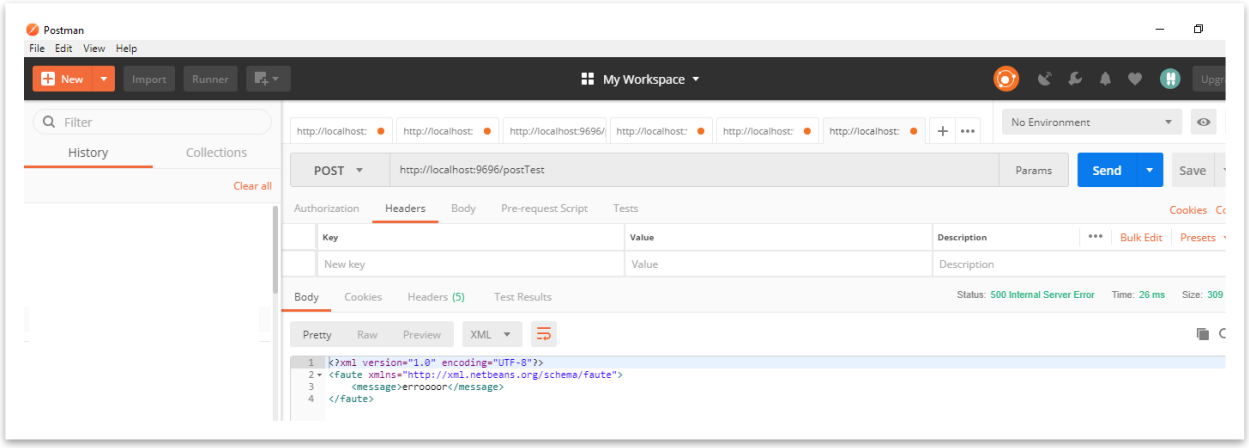
then the process replies to the request with a fault.



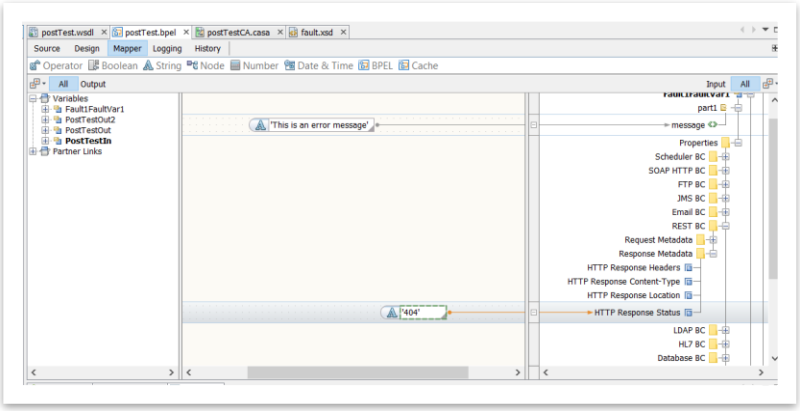




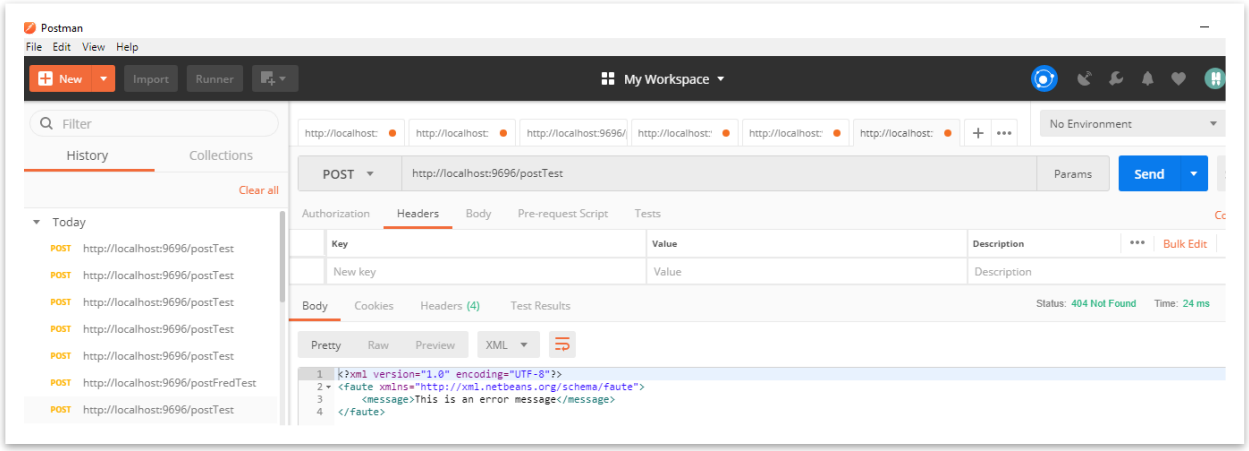
By default, the REST BC returns a response Status 500. In postman, the fault is understood as an Internal Server Error.



In the BPEL mapper the proper HTTP Response Status is set up to match your API policy.



Other elements such as the header parameters can be added to the response message.



The HTTP code 404 Not found, is read and display by Postman.

Fault management recap

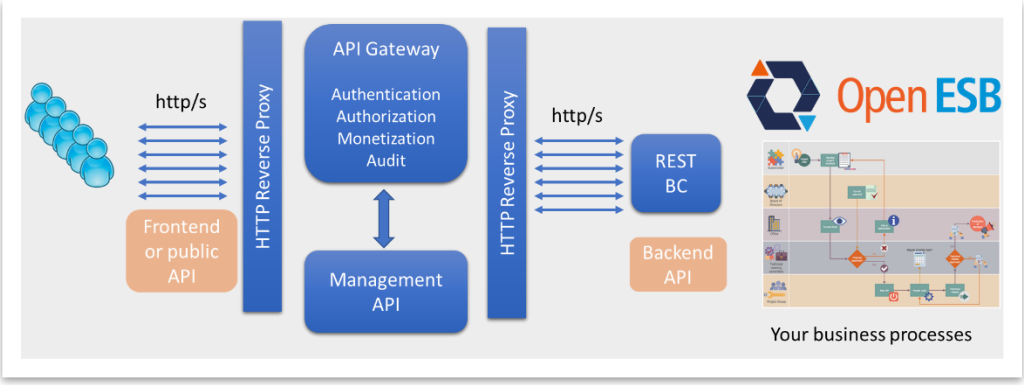
REST faults are generated at two levels in OpenESB. The first level is the REST manager that creates the REST error message when the request does not match an endpoint (or a configuration) deployed on the OpenESB instance. The second is the process level where complete and documented business faults are issued. API designers must be aware of this difference when they define their API.

Conclusion and thought

API Management and API Gateway

The REST Binding Component aims to offer a REST channel to your business services. The component allows you to invoke and publish a complete REST API to exchange with your partners and applications and create new business processes with OpenESB.

Nevertheless, the OpenESB is an integration platform, but not an API management system or an API Gateway. For public exposure, we recommend inserting an API gateway between OpenESB and Internet for management and security reasons. OpenESB REST API must be seen as a "backend API" unexposed to the mainstream.



For more information on the API/Gateway and API/ Management have a look on the gravitee.io web site

Think about the process first

One of the main misconceptions with the REST API design is to specify the REST API first and then the business processes and their services. This approach must be replaced by a top-down design. Always start by understanding your business specifications, then define your services, their granularity and their interfaces, and from their design and implement your REST API.

To conclude

REST is different from the other protocols such as SOAP or File, since it uses HTTP operations and Status Code and don't transport them in the messages exchanged between partners. Consequently, the OpenESB Enterprise Edition REST BC is a bit different from the other binding Components. Nevertheless, once this difference understood, OpenESB REST BC is very efficient and a simple BPEL project can support a complete API with dozens of services and give a direct access to your back office and legacy system.

Backed by a rigorous development process, OpenESB Enterprise Edition is a perfect tool to add a REST channel to your services and integration platform.

Appendix

Rest Binding Component MNR properties

General NMR properties

Property Name in Source	Property Name in Mapper	Description and Use
org.glassfish.openesb.messaging.groupid	Group ID	Uniquely identifies a message with the group to which a message belongs. This property is optional.
org.glassfish.openesb.messaging.messageid	Message ID	Uniquely identifies a message. For batch processing this might be a record number (for example, a record in a file) or a GUID. This property is mandatory.
org.glassfish.openesb. messaging.lastrecord	Last Record	The value is a string representation of boolean ("true" or "false"). This property can be used to signal the last record in a group or the last record in a file. This property is mandatory.
org.glassfish.openesb.exchange.endpointname	Endpoint Name	The value a string representation of the endpoint name set on the exchange. This represents the endpoint name of the "owner" of the message and could be made available by JBI runtime.

REST Binding Component NM Properties (Request)

Property Name in Source	Property Name in Mapper	Description
org.glassfish.openesb.rest.url	HTTP Request URL	The HTTP URL of the external resource to be invoked.
org.glassfish.openesb.rest.method	HTTP Request Method	The HTTP method to use when invoking the resource defined above. Available methods are GET, POST, PUT, HEAD, and DELETE.
org.glassfish.openesb.rest.content-type	HTTP Request Content-Type	The content type header for the requesting entity, if any.
org.glassfish.openesb.rest.accept-types	HTTP Request	The acceptable media types for the request, specified in JSON format. Enter the types in

	Accept-Types	square brackets with each type contained in double-quotes. Separate multiple values by a comma. For example: ["application/json", “application/xml”, "text/plain"]
org.glassfish.openesb.rest.accept-languages	HTTP Request Accept Languages	The preferred natural languages specified in JSON format.
org.glassfish.openesb.rest.headers	HTTP Request Headers	Custom HTTP headers for the request. Enter the headers in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example:{ "host" : "MyServer.com", "Content-Subtype" : "application/json/customers"}
org.glassfish.openesb.rest.headers.*	Not applicable	Arbitrary custom HTTP headers for the request. For example, to add the content type as a custom header parameter, you would enter a property similar to org.glassfish.openesb.rest.headers.content-type. These properties can only be defined using the BPEL Designer's Source view.
org.glassfish.openesb.rest.params	HTTP Request Parameters	Custom parameters for the request. Enter the parameters in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example:{ "status" : "Active", "billing" : "Current"}
org.glassfish.openesb.rest.params*	Not applicable	Arbitrary custom properties for the request. For example, if you are querying for telephone fields, you might have a set of properties like the following: • org.glassfish.openesb.rest.params.areaCode • org.glassfish.openesb.rest.params.number • org.glassfish.openesb.rest.params.extension These properties can only be defined using the BPEL Designer's Source view.
org.glassfish.openesb.rest.param-style	HTTP Request Parameter Style	A style for the URI parameters. The following values are supported: • **Query** – Name and value pairs that specify attributes of the full URI (external resource). Query parameters are

		delimited by an ampersand (&) and are separated from the rest of the URI by a question mark (?). • **Matrix** – Name and value pairs that specify attributes of one segment in a URI. Matrix parameters can occur after the segment in the URI that they modify. Matrix parameters are delimited by a semicolon (;) and are also separated from the segment they modify by a semicolon.
org.glassfish.openesb.rest.path-params	HTTP Request Path Parameters	Custom HTTP parameters for the URI. Enter the parameters in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example: { "status" : "Active", "billing" : "Current"}
org.glassfish.openesb.rest.path-params.*	Not applicable	Arbitrary custom HTTP parameters for the URI. For example, the following properties define custom user login properties: • org.glassfish.openesb.rest.path-params.userName • org.glassfish.openesb.rest.path-params.password These properties can only be defined using the BPEL Designer's Source view.
org.glassfish.openesb.rest.basic-auth-username	Not applicable	The login ID of the user for authentication. If the property is populated, a basic authentication header is added to the HTTP request.
org.glassfish.openesb.rest.basic-auth-password	Not applicable	The login password corresponding with the above user name. This property can only be access from the BPEL Designer's Source view.

REST Binding Component NM Properties (Response)

Property Name in Source	Property Name in Mapper	Description
org.glassfish.openesb.rest.response.status	HTTP Response Status	The status code for the response.
org.glassfish.openesb.rest.response.url	HTTP Response Location	The location header for the response.
org.glassfish.openesb.rest.response.content-type	HTTP Response Content-Type	The content type header for the response.
org.glassfish.openesb.rest.response.headers	HTTP Response Headers	Other headers for the response. Enter the headers in curly brackets as name value pairs with the name and value each in double-quotes and separated by a space, a colon, and another space. Separate multiple name value pairs by a comma. For example: { "host" : "MyServer.com", "Content-Subtype" : "application/json/customers"}
org.glassfish.openesb.rest.response.headers.*	Not Applicable	Arbitrary custom HTTP headers for the response. For example, to add the content type as a custom header parameter, you would enter a property similar to org.glassfish.openesb.rest.headers.content-type. These properties can only be defined using the BPEL Designer's Source view.

HTTP Status Codes

For your convenience, we put the HTTP status code in this document

Value	Description	Reference
100	Continue	[RFC7231, Section 6.2.1]
101	Switching Protocols	[RFC7231, Section 6.2.2]
102	Processing	[RFC2518]
103	Early Hints	[RFC8297]
104-199	Unassigned	
200	OK	[RFC7231, Section 6.3.1]
201	Created	[RFC7231, Section 6.3.2]
202	Accepted	[RFC7231, Section 6.3.3]
203	Non-Authoritative Information	[RFC7231, Section 6.3.4]
204	No Content	[RFC7231, Section 6.3.5]
205	Reset Content	[RFC7231, Section 6.3.6]
206	Partial Content	[RFC7233, Section 4.1]
207	Multi-Status	[RFC4918]
208	Already Reported	[RFC5842]
209-225	Unassigned	
226	IM Used	[RFC3229]
227-299	Unassigned	
300	Multiple Choices	[RFC7231, Section 6.4.1]
301	Moved Permanently	[RFC7231, Section 6.4.2]
302	Found	[RFC7231, Section 6.4.3]
303	See Other	[RFC7231, Section 6.4.4]
304	Not Modified	[RFC7232, Section 4.1]
305	Use Proxy	[RFC7231, Section 6.4.5]
306	(Unused)	[RFC7231, Section 6.4.6]
307	Temporary Redirect	[RFC7231, Section 6.4.7]
308	Permanent Redirect	[RFC7538]
309-399	Unassigned	
400	Bad Request	[RFC7231, Section 6.5.1]
401	Unauthorized	[RFC7235, Section 3.1]
402	Payment Required	[RFC7231, Section 6.5.2]
403	Forbidden	[RFC7231, Section 6.5.3]
404	Not Found	[RFC7231, Section 6.5.4]
405	Method Not Allowed	[RFC7231, Section 6.5.5]
406	Not Acceptable	[RFC7231, Section 6.5.6]
407	Proxy Authentication Required	[RFC7235, Section 3.2]

408	Request Timeout	[RFC7231, Section 6.5.7]
409	Conflict	[RFC7231, Section 6.5.8]
410	Gone	[RFC7231, Section 6.5.9]
411	Length Required	[RFC7231, Section 6.5.10]
412	Precondition Failed	[RFC7232, Section 4.2][RFC8144, Section 3.2]
413	Payload Too Large	[RFC7231, Section 6.5.11]
414	URI Too Long	[RFC7231, Section 6.5.12]
415	Unsupported Media Type	[RFC7231, Section 6.5.13][RFC7694, Section 3]
416	Range Not Satisfiable	[RFC7233, Section 4.4]
417	Expectation Failed	[RFC7231, Section 6.5.14]
418-420	Unassigned	
421	Misdirected Request	[RFC7540, Section 9.1.2]
422	Unprocessable Entity	[RFC4918]
423	Locked	[RFC4918]
424	Failed Dependency	[RFC4918]
425	Too Early	[RFC8470]
426	Upgrade Required	[RFC7231, Section 6.5.15]
427	Unassigned	
428	Precondition Required	[RFC6585]
429	Too Many Requests	[RFC6585]
430	Unassigned	
431	Request Header Fields Too Large	[RFC6585]
432-450	Unassigned	
451	Unavailable For Legal Reasons	[RFC7725]
452-499	Unassigned	
500	Internal Server Error	[RFC7231, Section 6.6.1]
501	Not Implemented	[RFC7231, Section 6.6.2]
502	Bad Gateway	[RFC7231, Section 6.6.3]
503	Service Unavailable	[RFC7231, Section 6.6.4]
504	Gateway Timeout	[RFC7231, Section 6.6.5]
505	HTTP Version Not Supported	[RFC7231, Section 6.6.6]
506	Variant Also Negotiates	[RFC2295]
507	Insufficient Storage	[RFC4918]
508	Loop Detected	[RFC5842]
509	Unassigned	
510	Not Extended	[RFC2774]
511	Network Authentication Required	[RFC6585]

512-599	Unassigned	