UNITECH 2019 GABROVO

INTERNATIONAL SCIENTIFIC CONFERENCE
15 – 16 November 2019, GABROVO

DATA EXCHANGE USING WSO2 ENTERPRISE SERVICE BUS

Stefan Pitulić[1], Slaviša Ilić[2], Siniša Ilić[1], Dragana Radosavljević[1]
[1]*Faculty of Technical Sciences, K. Mitrovica University of Priština, Serbia*
[2]*Faculty of Informatics and Computing University Singidunum, Serbia*

Abstract

In this paper the advantages of using Enterprise Service Bus in data exchange using SOAP and RESTful web services on particular examples are presented. The WSO2 open source ESB is used to connect different Web services deployed on different endpoints. Also, the possibility of transformation of messages used in data exchange between different web services is demonstrated in WSO2 API.

Keywords: ESB, connecting web application to ESB, connecting SOAP and RESTful to ESB.

INTRODUCTION

Many applications from time to time require the data from external resources and from various services. Usually this communication is realised on the Intranet or Internet.

Nowadays, data exchange is performed using the Web Services, whose main objective is to provide a service abstraction that allows interoperability between applications built using disparate platforms and environments [1]. Such architecture is known as Service Oriented Architecture (SOA). The data is mainly exchanged using XML message formats, and the services are interconnected by a P2P (point to point) connection. Complexity arises when there is a need to interconnect a large number of services within some organisation (company, clusters of companies, e-government of a country, etc.).

A typical scenario is that an enterprise runs hundreds or thousands of applications, which could be custom built, acquired from a third party or parts of legacy systems [2]. Then each service requires one communication interface with another service. Sometimes it is very hard and time consuming to establish such (multiple point to point) connections and reconfigure active network equipment. Also, there is a problem how to configure connection and access security for each new connection node with the data source.

The solution to this problem is to create a single interface - a single point of access for all communication services and this interface is actually the Enterprise Service Bus (ESB). With the advent of the ESB there is now a way to incorporate web services into a meaningful architecture for integrating applications and services into a backbone that spans the extended enterprise in a large-scale fashion [1]. In this way, all web services and applications can request and receive the data in a supported message format, and each node (where applications are deployed) will be connected via one interface to that unique access point (ESB).

All messages exchanged between the services are transmitted via an ESB that acts as a courier. The ESB can receive messages from one service, pack and forward them to another service, and/or return a message (response) to the sender.

In a general case a set of applications in a production environment may be built in different platforms and share data in different formats. Some applications may be parts of larger applications that use pre-defined format of messages and protocols, some other applications may use modern messages and protocols. That is the reason why ESB is designed to handle many different message formats and protocols and is able to: check messages, change their format, filter them, redirect them to different endpoints, process and change the content of initial messages, etc.

In addition to being the main point of message mediation, the ESB also provides security at the highest level, and it also has the

ability to log events. The ESB enables easy integration of various services without a need for a user to write a source code.

Some of the well-known commercial ESB solutions are: Microsoft BizTalk Server, Oracle Enterprise Service Bus, IBM Integration Bus. In this paper, the implementation of the open-source solution WSO2 is presented.

The main goal of this paper is to present usage of WSO2 ESB [3] and connecting of the RESTful (Representational state transfer) and SOAP (Simple Object Access Protocol) Web Services on particular examples.

SOAP AND RESTFULL WEB SERVICES

SOAP is a standard protocol proposed by the W3C [4,5] to interface Web Services, and that extends the remote procedure call (XML-RPC). Thus, SOAP can be considered as an evolution of XML-RPC protocol, much more complete and mature, that allows to perform remote procedure calls to distributed routines (services) based on an XML interface as interfacing language. Thus, SOAP clients can access to objects and methods that are residing in remote servers, using a standard mechanism that makes transparent the details of implementation, such us the programming language of the routines, the operating system or the platform used by the provider of the service. At the moment, there exist complete implementations of SOAP for Perl, Java, Python, C++ and other languages [6].

SOAP sends and receives messages using XML [7-9], wrapped HTTP-in headings. The interfaces of the methods that can be accessed using SOAP services are specified by a Web Services Description Language (WSDL) [10, 11]. Using an WSDL file, that it is based on a neutral language such as XML, the service can be specified for different languages, so that a Java client can access a Perl server.

REST is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term Representational State Transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation [12, 13]. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1 [14, 15].

REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed.

Although REST was initially described in the context of HTTP, is not limited to that protocol. RESTful architectures can be based on other Application Layer protocols if they already provide a rich and uniform vocabulary for applications based on the transfer of meaningful representational state. RESTful applications maximize the use of the pre-existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it.

CREATION OF TESTING ENVIRONMENT

In order to test the usage of WSO2 ESB several endpoints with web services are created. Two SOAP Web Services (WS): “BibService” and “ServiceB”, each with several functions are created, and one RESTful WS (“TutorialService”) with four functions. Also, two client applications were built: one for exchanging data with SOAP WS and one for retrieving, inserting, updating and deleting data using the RESTful web services.

Initially, client applications were connecting to the WS directly in a peer to peer fashion and were used to issue the request to an endpoint demanding the data from a WS deployed at that endpoint. Our goal was to modify client application to issue request to the ESB (not to a particular endpoint) for a specific WS and receive data from ESB regardless from which endpoint data actually were coming from. For each WS appropriate API had to be developed and implemented in ESB in order to process request from a client application, create own request to the particular endpoint, receive data from a web service and transfer those data to a client application in desired format.

For issuing the WS request (beside client applications we developed) we used also SoapUI (https://www.soapui.org/) application

for testing requests and responses to and from Web Services.

Creation of SOAP Web Services for experiment 1

All Web Services in the experiment were created using the Microsoft .Net platform. The first SOAP WS "BibService" was designed to exchange data related to small library. The service was connected to the MS Access Database and the functions of this WS were:

- ***AddBook****(pWriterID, pBookTitle),*
- ***GetWriterID****(pWriterName),*
- ***ShowAllWriters****(), and*
- ***ShowWritersAndBooks****().*

The second SOAP WS "ServiceB" was designed to exchange data related to Book Writers, it is also connected to (another) MS Access database and there was only one function:

- ***GetNumberOfBooks****(pWriterID)*

The initial architecture for experiment 1 is presented in Fig. 1.

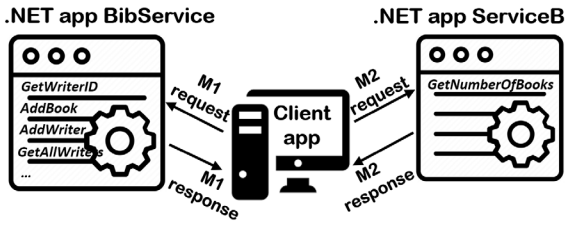

Fig. 1. *Initial architecture of test environment for experiment 1*

When implementing .Net web services, WSDL description for the developed WS can be downloaded from the web address http://hostname/WebServiceName.asmx?WSDL. The WSDL file can be later uploaded to the SoapUI for proper configuring application to issue request to that Web Services.

The request and response messages for function *GetWriterID* (using parameter *pWriterName = "Desanka Maksimovic"*) of WS "BibService" in SoapUI application are presented in Fig. 2.

Creation of RESTful Web Services for experiment 2

One RESTful WS [16] with four functions: POST, GET, PUT, DELETE was developed and deployed. This Web Service was not connected to the database.

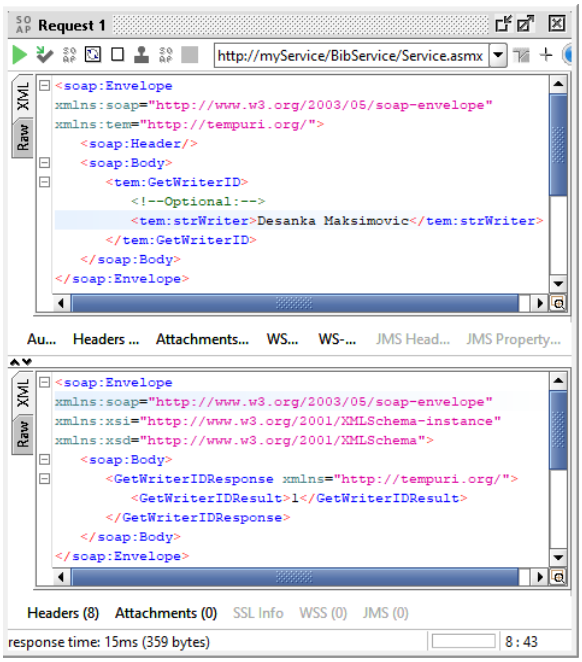

Fig. 2. *Request from SoapUI to the function GetWriterId of WS BibService*

Its purpose was to test aforementioned functions on an initial array of strings. The initial architecture of test environment for experiment 2 is presented in Fig. 3

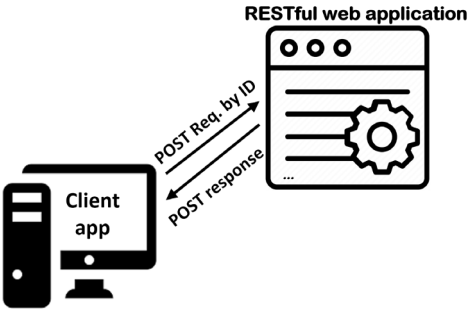

Fig. 3. *Initial architecture of test environment for experiment 2*

CONNECTING WEB SERVICES TO ESB AND API COMPONENTS

As stated above, the WSO2 ESB (https://wso2.com/) solution was used to connect the Web Services and create needed APIs. WSO2 uses modified Eclipse version for building the APIs and configuring endpoints. Because the ESB has to communicate with available Web Services, each WS has to be defined for access within the ESB. For this purpose, the names of WS, appropriate endpoints (Web URLs) and WDSL files had to be defined in ESB API as well as the context of API. In order to create an WSO2 API, certain objects called mediators (the building blocks of each API) were used. Mediators were defined in the development environment, the number of

them was large. The mediators used in our experiments were:

Log mediator - the object that shows events for all messages that pass through ESB WSO2 API.

Send mediator – the object that sends a message to some Endpoint. For each message a Message ID will be assigned.

Property mediator - the container of variables. It can set or unset the value of some variable. It can also extract some content from a message (i.e. from XML, JSON, Envelope, etc.).

PayloadFactory mediator - the object that transforms the content of a message.

Call mediator - the object that is used to send a message outside of ESB-a to some endpoint. The response is then received back to the working area (called Synapse) of the WSO2 ESB.

Respond mediator - the object that stops the message processing within the Synapse and sends back the message to the client as a response.

Implementation of the Experiment 1 (connecting SOAP Web Services to ESB)

The goal of experiment 1 was to connect client application to ESB and create request for new function *GetNumberOfBooks (pWriterName)* (function with the same name of WS BibService has parameter *pWriterID*). The new function (created in WSO2 API) would firstly call function *GetWriterID* from the WS "BibService" and after receiving response (the value WriterID) would call function *GetNumberOfBooks* from the WS "ServiceB" by passing the parameter WriterID obtained from "BibService" previously. The response of the later WS would be then returned back to the client application.

The architecture of the experiment 1 is presented in Fig 4.

In this scenario the client application issues request to the ESB by sending message m1wn (Message 1 contains WriterName). The ESB processes the request and sends a message m1wn (with a Writer Name received from a client request) to the WS "BibService"; "BibService" responds and sends back the message m1id response to the ESB that contains value of WriterID.

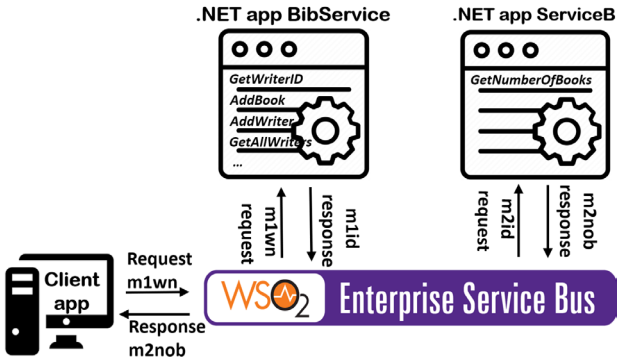

Fig. 4. *The final architecture of SOAP services in experiment 1*

The ESB transforms received envelope in message m1id (extracts the WriterID value) and call another WS "ServiceB" by sending the message m2id request (with WriterID received in m1id), receives the response m2nob (Message 2 Number of Books) and passes the response to the client application.

In Fig 5, the architecture of created API for experiment 1 (by combining and configuring the appropriate mediators) is presented.

From Fig. 5 two sequences of objects can be seen: first group in upper part with arrows going from "Resources" object and second group in lower part with arrows going to "Resources" object.

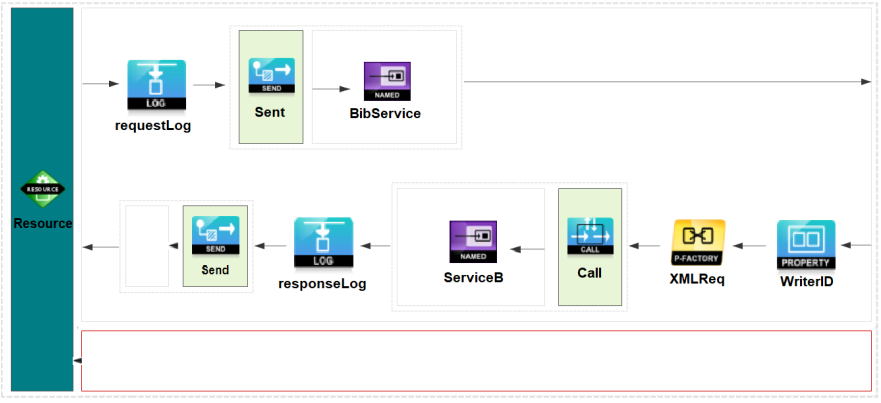

Fig. 5. *Mediation components of API for experiment 1*

The objects in the first group receive request from a client application and send the request to "BibService" WS, and the second group receive response from "BibService" WS, create request to "ServiceB", receive response from "ServiceB" and send response to a client application.

The first mediator in the first group is Log mediator *requestLog* that logs the request event and passes the received SOAP envelope to the Send mediator that forwards the message content to the endpoint "BibService".

In the second group of mediators, the Property mediator *WriterID* extracts the value of *WriterID* from "BibService" response and passes it to the Factory mediator *XMLreq* that creates format of envelope for the Web Service "ServiceB" and function *GetNumberOfBooks*. The created message is then sent to the "ServiceB" endpoint via Call mediator. The response message from "*ServiceB"* is received by Log mediator *responseLog* and sent back to a client application through the Send mediator. If error occurs, a developer may put mediators in the red rectangle at the bottom of Fig. 5 to handle that error.

When API is created, it should be deployed in ESB server through the Management Console as Carbon application in a format of Composite Application Project. The API called *GetNumberOfBooks* is thus deployed in the Management Console.

Implementation of the Experiment 2 (connecting RESTful Web Services to ESB)

The goal of experiment 2 was to connect client application with existing RESTful Web Services via ESB and receive required data in JSON format. A client application needed to exchange data with ESB instead to exchange data with each WS deployed on specific endpoint. In this case, the ESB was configured to return back response to a client application in JSON format by reformatting data received from RESTful WS in XML format.

The desired configuration is presented in Fig. 6.

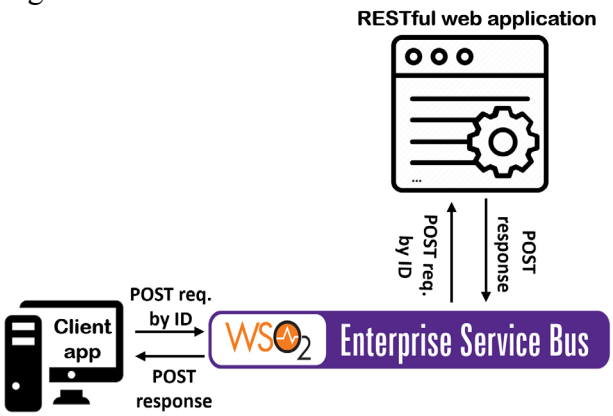

***Fig. 6** – Desired architecture for RESTful service*

In this scenario a client application had to send request to the ESB (POST request by ID), the ESB had to forward request to the RESTful WS, receive response from the RESTful WS (POST response), transform the format of data and send back to a client application (POST response).

Unlike in SOAP Web Services, in RESTful services the context is more dynamic. There are GET, POST, and DELETE methods, so more Resource objects should be defined. In this case only GET method is described. The task of GET function in "TutorialService" WS was to return the string from the string array at required position in XML format.

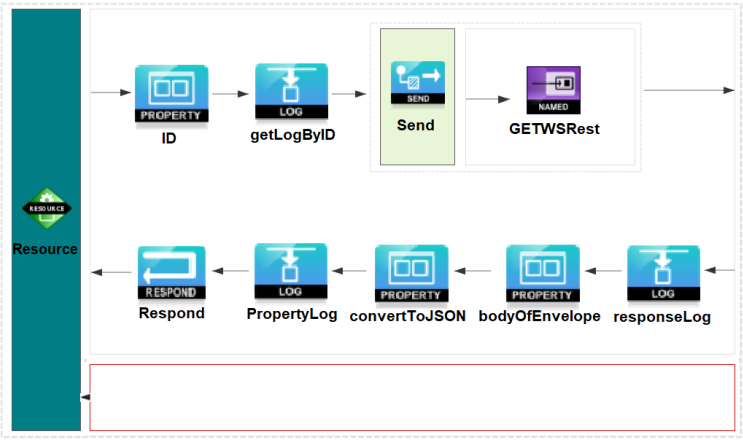

***Fig. 7.** Mediation components of API for experiment 2*

In Fig 7, the architecture of created API for experiment 2 (by combining and configuring the appropriate mediators) is presented.

Like in SOAP API, the first group of mediators receive request from a client application and send the request to RESTful WS, and the second group receive response from RESTful WS and send response to a client application.

The first Property mediator (*ID)* extracts from the context of POST request the parameter -sequence number of the string in an array, and passes request to the Log mediator *GetLogByID*. Log mediator logs the request event and passes the received POST request to the Send mediator that forwards the message content to the endpoint *GetWSRest* (it is actually "TutorialService" WS). The Log mediator *ResponseLog* logs the response event and passes it to the Property mediator *BodyOfEnvelope* that extracts the return value from the received envelope. The envelope is then passed to the PayLoadFactory mediator *ConvertToJSON*, that converts envelope content from XML to JSON. The Log mediator *PropertyLog* logs the content of the parameters in the envelope, and the created response is sent back to a client application via *Respond* mediator. The request and response messages to and from this API is tested via SoapUI application.

CONCLUSION

In modern age many applications can exchange data with another applications regardless of platforms they were developed and implemented using the Web Services (WS). The advantage of using WS is that data can be exchanged in a standard format (usually XML or JSON).

When the number of applications that exchange data, becomes large, it is easier to create a single point of access for all communication services rather than to establish multiple point to point connections. This backbone interface is actually the Enterprise Service Bus (ESB).

In this paper we presented a usage of WSO2, the open source ESB solution for handling SOAP and RESTful Web Services and demonstrated on particular examples how WSO2 can handle multiple WSs in one request and response to a client and how WSO2 can process and transform response messages successfully.

ACKNOWLEDGMENT

This work has been supported by the Ministry of Science and Technological Development of the R. of Serbia under Project No.TR-35026.

REFERENCE

[1] Chappell D. Enterprise Service Bus O'Reilly, 2004.

[2] Menge F. Enterprise Service Bus, Free and open source software conference, 2007.

[3] Indrasiri K. Beginning WSO2 ESB, First Edition, 2016.

[4] Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H, Thatte S, and Winer D. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000, available at: http://www.w3.org/TR/SOAP, 2019.

[5] Avila P. SOAP: revolucion en la red, *Linux actual*, no. 19, pp. 55–59, 2001.

[6] SOAP software, available at: *http://www.soaprpc.com/software* , 2011.

[7] Ray E. T, *Learning XML: creating self describing data*. O´Reilly, January 2001.

[8] Harold E. R. *XML Bible*. IDG Books worldwide, 1991.

[9] Box D. Inside SOAP, available at: *http://www.xml.com/pub/a/2000/02/09/feature/index.html ,* 2019.

[10] Ryman A. Understanding web services, available at: *http://www7.software.ibm.com/vad.nsf/Data/Document4362?OpenDocument&p=1&BCT=1&Footer=1* , 2011.

[11] Vasudevan V. A web services primer, available at: *http://www.xml.com/pub/a/2001/04/04/webservices/index.html* , 2011.

[12] Fielding R. Architectural Styles and the Design of Network-based Software Architectures, *Doctoral dissertation, University of California, Irvine*, 2000.

[13] Fielding and Taylor R. Principled Design of the Modern Web Architecture, *ACM Transactions on Internet Technology (TOIT) (New York: Association for Computing Machinery) 2 (2): 115-150*, 2002.

[14] IETF, RFC 1945, available at: *http://tools.ietf.org/html/rfc1945* , 2019.

[15] HTTP/1.1, RFC 2616, available at: *http://tools.ietf.org/html/rfc2616* , 2019.

[16]RESTful service, available at: https://www.guru99.com/restful-web-services.html, 2019.