REST Gateway Demo Documentation

Release 1.0

Hiranya Jayathilaka

September 07, 2015

Contents

Contents:

CHAPTER 1

Introduction

Apache Synapse is a lightweight Enterprise Service Bus (ESB) that supports a wide range of communication protocols and messaging standards. It enables integrating heterogeneous systems in a simple and unified manner without requiring the system integrators to modify any of the existing applications or write custom adapters. Instead, the Synapse ESB is configured using a simple XML based metalanguage to define high-level APIs, services, endpoints and mediation flows to construct fully automated business processes and implement powerful Enterprise Integration Patterns (EIPs).

Synapse provides comprehensive support for RESTful services as well as traditional SOAP/WS-* services. This allows developers to use Synapse as a gateway (proxy) for both SOAP and RESTful services. More specifically, Synapse can be used to provide a REST gateway for a collection of SOAP services. This is a particularly useful feature by which an existing SOAP service can be made available to a completely different group of consumers (e.g. RESTful mobile clients) without having to change any code related to the backend web service. In this tutorial we will look at how to use Apache Synapse as a REST gateway for a set of given SOAP services. We will look at how to define RESTful APIs in Synapse using its metalanguage and then map RESTful API calls into SOAP web service requests without compromising any features or semantics of the backend SOAP services.

This demonstration consists of 3 parts (available in 3 separate directories).

- Backend SOAP service
- Synapse ESB configuration
- Sample client scripts

Read on to learn more about each of these components.

1.1 Backend SOAP Service

We are going to use a simple order management demo service as our backend SOAP service. It supports the following basic ope

- addOrder - Place a new order for a drink
- getOrder - Get the details regarding an existing order
- updateOrder - Update an existing order
- deleteOrder - Delete/Cancel an existing order
- getAllOrders - Get a list of all existing orders

The service has been implemented in Java using the Apache Axis2 framework. Therefore it can be deployed in Axis2 or any Axis2 compatible service container. To build the deployable service artifact, go into the `service/OrderManagementService` directory and run the following Maven command.

```
mvn clean install
```

This will create a binary file named `OrderManagementService.aar` in the `target` directory which can be directory deployed.

1.2 Synapse ESB Configuration

As mentioned earlier, Apache Synapse is configured using a simple XML based metalanguage to receive, process and mediate service requests. In order to use Synapse as a gateway between REST clients and SOAP services, we need a set of Synapse configuration files that can process RESTful service requests by converting them into standard SOAP requests. In this demo application we configure Synapse to receive RESTful service requests and convert them into SOAP requests that can be sent to our backend order management service. SOAP responses from the backend service are converted back to the original REST style and sent back to the client. This way a RESTful client can interact with our SOAP based order management service using pure REST calls, without ever knowing that the backend is actually based on SOAP. Therefore the Synapse ESB acts as a transparent proxy between the REST client and the SOAP service in this case.

The set of Synapse configuration files for this demo can be found in the `esb/synapse-config` directory. These files can be directly deployed in Apache Synapse by copying them to the `repository/conf/synapse-config` directory of the Synapse installation. For the tutorial we will be using WSO2 ESB, an open source ESB product that uses Synapse as its mediation engine. In that case these configuration files should be copied to the `repository/deployment/server/synapse-configs/default` directory of the ESB installation.

1.3 Sample Client Scripts

In order to interact with this demo application, some RESTful client tool is necessary. A simple command line tool such as `curl` can be used for this purpose. In order to simplyfy the testing procedure, a set of sample Python scripts (based on httplib) are provided in the `client` directory.

You may use any other REST client application to interact with this demo and try it out. If you are looking for a more UI oriented client tool, please check out the Chrome Advanced REST Client.

CHAPTER 2

Getting Familiar with Apache Synapse

Apache Synapse is a lightweight Enterprise Service Bus (ESB) that caters a wide range of service integration and messaging req

- Message logging
- Message filtering
- Content based routing
- URL rewriting
- Message transformation
- Protocol switching
- Load balancing and fail-over routing
- Persistent messaging support
- Message enriching

With these basic features, Synapse enables the developers and system integrators to connect multiple heterogeneous systems and implement a wide range of enterprise integration patterns (EIPs). This is particularly useful in large enterprise settings, where a number of disparate systems must co-exist and work together in perfect synchronism to support complex business processes and workflows.

What makes Synapse even more attractive is the fact that Synapse facilitates a zero-code approach for connecting systems. This implies that multiple systems can be connected via Synapse without having to write any custom adaptation code. Instead, Synapse only needs to be configured using a very high-level, XML-based metalanguage. The language is simple, intuitive and abstracts out all the wire-level and other implementation details of the message flows and systems being integrated. Therefore the configurations are easy to develop, highly reusable and easy to maintain over time. The backend systems can undergo architecture or implementation level changes without inducing any changes to the Synapse configuration.

Since the ESB acts as a hub that sits between several systems, it's crucial that the ESB does not introduce any unnecessary latencies to the message flows that go through it. Synapse really stands out from other similar ESB products with regard to this aspect. Benchmarking results have shown that Synapse and the ESB products based on Synapse are the among the fastest in the world, can handle thousands of concurrent connections and can handle enormous volumes of traffic at sub millisecond latencies. Some recent benchmark results involving Synapse can be found at http://wso2.org/library/articles/2013/01/esb-performance-65.

2.1 Simple Proxy Services

This section provides a quick tutorial aimed at giving a little taste of what Synapse really is and how to use it in practice.

- Start by downloading the Apache Synapse binary distribution. The latest stable release as of now is v2.1.0
- Simply extract the downloaded archive to install the ESB.

```
unzip synapse-2.1.0-bin.zip
tar xvf synapse-2.1.0-bin.tar.gz
```

- Synapse ships with a set of prepackaged sample configurations. Each sample is identified by a unique numeric identifier. For this tutorial we will be using sample 150. To start ESB with this sample configuration simply run the following command from the `bin` directory of the Synapse installation.

```
Unix/Linux: sh synapse.sh -sample 150
Windows: synapse.bat -sample 150
```

- Sample 150 consists of a simple proxy service configuration. Proxy services are used to intercept web service requests before they reach their actual target destinations and perform some additional processing (can perform both pre-processing and post-processing). The exact configuration used in sample 150 looks like this.

```
<proxy name="StockQuoteProxy">
   <target>
       <endpoint>
           <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
       </endpoint>
       <outSequence>
           <send/>
       </outSequence>
   </target>
   <publishWSDL uri="file:repository/conf/sample/resources/proxy/sample_proxy_1.wsdl"/>
</proxy>
```

- This tells Synapse to expose a proxy service named `StockQuoteProxy`. Any requests received by this service will be forwarded to the backend endpoint `http://localhost:9000/services/SimpleStockQuoteService`. The responses coming back from the service will be sent back to the client as they are. Therefore this proxy service does not really perform any additional processing on the messages. It simply acts as a transparent pass-through pipe between the client and the server.
- To try this out we should first start a sample backend service at `http://localhost:9000/services/SimpleStockQuoteService`.
  Synapse ships with all the artifacts needed for this. Simply head over to the `samples/axis2Server/src/SimpleStockQuoteService` directory and invoke ANT.

```
ant
```

- This would build a mock web service named `SimpleStockQuoteService` and deploy it into a sample Axis2 server that ships with Synapse. To start this sample backend server, head over to `samples/axis2Server` and run the appropriate startup script.

```
Unix/Linux: sh axis2server.sh
Windows: axis2server.bat
```

- Now you have a backend server running. To invoke the proxy service, head over to `samples/axis2Client` directory and run the following ANT command.

```
ant stockquote -Daddurl=http://localhost:8280/services/StockQuoteProxy
```

- This would invoke your proxy service, which in turns invokes the backend Axis2 service and gets you the desired response. You can also use a tool such as SOAP-UI to try out the proxy service.
- Finally, launch your web browser and navigate to the URL `http://localhost:8280/services/StockQuoteProxy?wsdl` to see the WSDL exposed by Synapse for the proxy service we configured.

2.2 Messaging Model

Previous section gave a breif outline of what it takes to install and run simple integration scenarios using Synapse. This section provides more detailed information about how Synapse works and what the underlying messaging model looks like.

The smallest configurable unit in Synapse is known as a mediator. A mediator can be viewed as a black box that takes an input message, performs some processing on it and produces an output message. Synapse ships with a large number of built-in mediators that are designed to handle various tasks such as logging, XSLT transformation, database lookups and URL rewriting. Using the Synapse configuration language we can combine multiple mediators to form complex message flows (sequences). Multiple sequences can be further combined to form high-level services.

Therefore the task of configuring Synapse boils down to defining the required sequences and services. In the previous section we configured a proxy service. A proxy service typically consists of an in-sequence and an out-sequence. The in-sequence mediates all the requests received by the proxy service and the out-sequence mediates all the responses sent by the backend service. In sample 150 however we only have an out-sequence configured. But it also have a target endpoint configured. From this Synapse infers that the requests must be directly forwarded to the target endpoint without performing any processing on it. The single `send` mediator configured in the out-sequence instructs Synapse to simply pass the response along to the client that started the invocation. If we need to perform some additional processing on the requests in this proxy service, all we need to do is to define an in-sequence in the proxy service and specify the required set of mediators. In the out-sequence it is possible to add more mediators and add more processing capabilities to the response flow. It is even possible to forward the response of the backend service to a different service and thereby chaining multiple services together to construct complex workflows.

Synapse mediators provide a high-level abstraction which allows the developers to configure services without considering the actual application layer protocol and the message format used to send/receive messages. All the messages are converted to the SOAP format before they are injected into the Synapse mediation engine. Therefore the message flow designer can simply assume that all messages are SOAP messages and invoke mediators on them. Therefore Synapse provides a uniform model for dealing with all types of messages and protocols.

2.3 REST API Support

Starting from version 2.1.0, Apache Synapse has comprehensive support for exposing REST APIs on the ESB and mediating RESTful service requests. Our demo application relies primarily on this REST mediation support of Synapse. A REST API configured in Synapse is somewhat similar to a webapp deployed in a servlet container. Each API has a unique name and it is anchored at a specific URL context. Within the API we can define one or more resources. Each resource is equivalent to a proxy service, with their own in-sequences and out-sequences. Each resource can

be configured to handle a particular URI template and/or HTTP method combination. Lets consider the following example API.

```
<api name="StockQuoteAPI" context="/stockquote">
  <resource uri-template="/view/{symbol}" methods="GET">
    <inSequence>
      ...
    </inSequence>
    <outSequence>
     ...
    </outSequence>
  </resource>
  <resource url-pattern="/order/*" methods="POST">
    <inSequence>
       ...
    </inSequence>
  </resource>
</api>
```

This API is anchored at the `/stockquote` context. Therefore it will handle any HTTP request whose request URL path starts with `/stockquote`. Then the API defines 2 resources. One resource will handle GET requests to the path `/stockquote/view/{symbol}` and the other will handle POST requests to the path `/stockquote/order/*`. Within each resource we can define in-sequences and out-sequences with any suitable mediator configuration to process the RESTful service requests.

2.4 More Documentation and Samples

To learn more about Apache Synapse, please refer to the official Synapse documentation. This includes a complete catalog of samples that Synapse ships with and detailed instructions on how to try them out. More specifically there are samples on Synapse message flows, mediators, proxy services, protocol switching and a number of other interesting scenarios.

2.5 Configuration Language

The configuration language specification for Synapse can be found at official Synapse documentation. The language is simple, intuitive and XML-based. Therefore most services and message flows can be constructed for Synapse by manually editing XML. If graphical tooling support is required, WSO2 Developer Studio which is based on Eclipse can be used.

Even though this is a high-level metalanguage, it supports all the faimilar constructs of a traditional programming language. If-else constructs, switch-case constructs and try-catch constructs are built into the language in the form of various mediators which can be used to construct powerful message flows with comprehensive control and data flow.

2.6 WSO2 ESB

For this demonstration we will not be using vanilla Apache Synapse. Rather we will be using WSO2 ESB which is an open source ESB based on Synapse. WSO2 ESB uses Synapse as its mediation engine and hence supports all the features (and more) that Synapse does. The same XML-based metalanguage used to configure Synapse is used to configure WSO2 ESB as well. This means any valid Synapse configuration is also a WSO2 ESB configuration and vice versa. In addition to the set of features provided by Synapse, WSO2 ESB provides excellent UI support, powerful management capabilities and flexible tooling support.

The set of samples that ship with Synapse are also being shipped with WSO2 ESB. Please refer the ESB documentation and samples guide to see how to try them out.

CHAPTER 3

REST Gateway Tutorial

3.1 Prerequisites

You will need following software tools to try out this tutorial. Simply download them from the specified URLs for now. The tutorial will guide you through the process of setting them up.

- WSO2 ESB 4.0.3 or higher (You can also use Synapse 2.1.0 instead. But the tutorial only provides instructions for WSO2 ESB.)
- WSO2 Application Server 5.1.0 or higher (You can use any Axis2 compatible application server here. But the tutorial only provides instructions for WSO2 Application Server.)

In addition to the above tools, your computer must be setup with the following infrastructure tools.

- JDK 1.6 or higher (Oracle JDK recommended)
- Python 2.7 or higher
- Apache Maven 3.0 or higher

The tutorial does not specify how to install these infrastructure software. Tutorial assumes that they are already installed and available for use.

3.2 Setting Up the Backend Server

We are going to host our sample order management service in WSO2 Application Server. Therefore lets start by setting up the Application Server.

- Simply extract the downloaded zip archive to install the WSO2 Application Server.

```
unzip wso2as-5.1.0.zip
```

- Go into the `bin` directory of the installation and execute the `wso2server` startup script to start the server.

  Unix/Linux: sh wso2server.sh Windows: wso2server.bat

- Wait for the server to finish initialization. This could take a few seconds depending on how fast your machine is.

Now that the server is up and running, lets deploy the order management service in it.

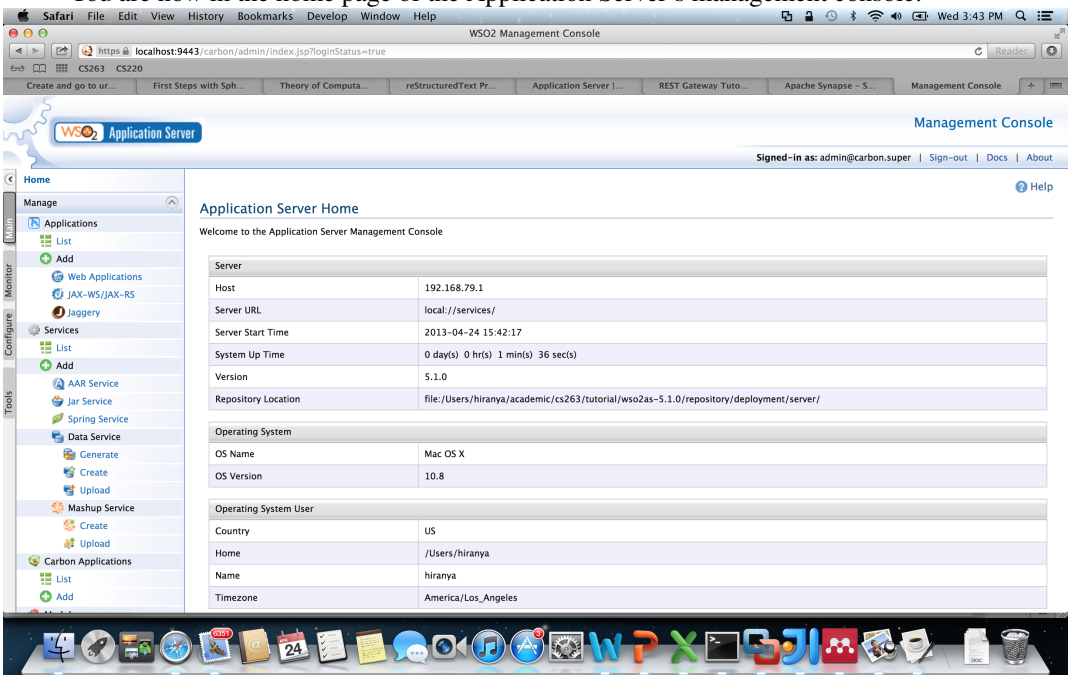
- Use Apache Maven to build the OrderManagementService.aar artifact. Simply go into to the `server/OrderManagementService` directory of this sample package and run the following command. This will create the required artifact in the `server/OrderManagementService/target` directory.

```
mvn clean install
```

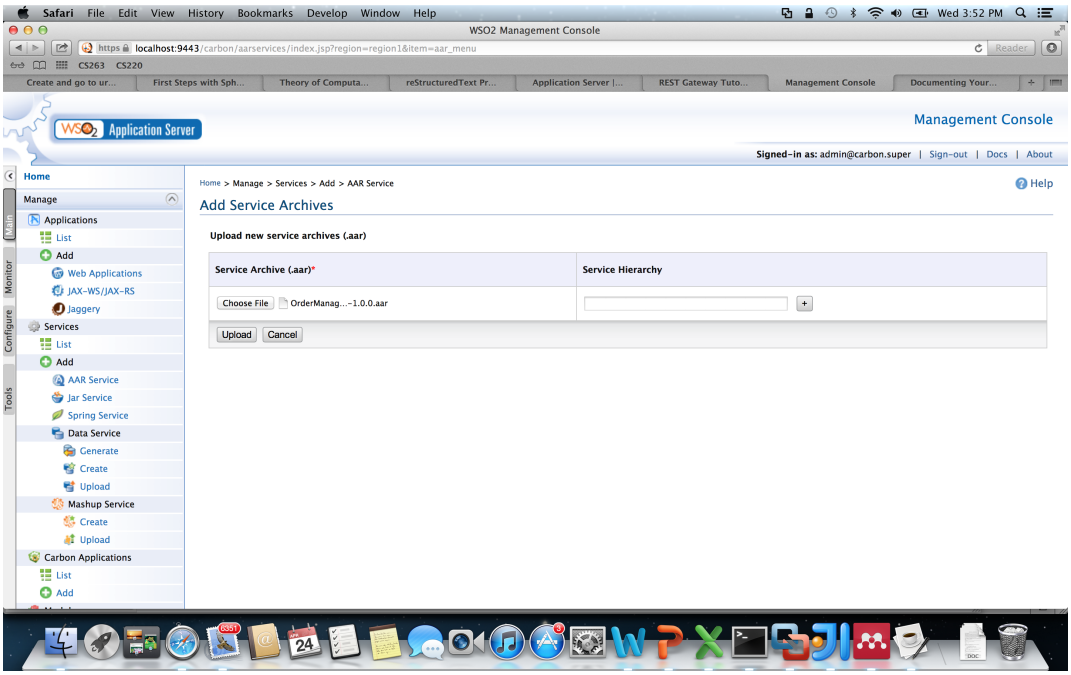
- Launch your web browser and navigate to the WSO2 Application Server management console. This is available at `https://localhost:9443/carbon`.
- Sign in to the console using default administrator credentials.

```
Username: admin
Password: admin
```

- You are now in the home page of the Application Server's management console.



- Click on the `Manage > Services > Add > AAR Service` option in the main menu.



- Browse and select the OrderManagementService.aar file and click on `Upload` to upload the artifact. Service deployment will take a few seconds. Click/refresh the service listing page (under `Manage > Services > List` in the main menu) and you will see the OrderManagementService listed there. Feel free click on it and browse through the various options available. For instance you should be able to see the WSDLs of the service and even invoke it online using the *Try-It* tool built into the management console. The service endpoint will be `http://localhost:9763/services/OrderManagementService`.

3.3 Setting Up the ESB

Now we have our backend SOAP service up and running. So lets get started with the ESB (our REST gateway).

- Extract the downloaded WSO2 ESB archive to install it.

```
unzip wso2esb-4.0.3.zip
```

- Before we can start the server, we need to do some configuration changes. This is because by default WSO2 ESB is configured to use the same set of ports as WSO2 Application Server. Since we are starting both servers in the same machine, we need to change the ESB ports before we can actually start it. To do this go into the `repository/conf` directory of the ESB installation and open the `carbon.xml` file using a text editor. Scroll down to the `Ports` configuration section and change the `Offset` value to `1`. This will increment all the port numbers used by the ESB by 1.

```
<Ports>

    <!-- Ports offset. This entry will set the value of the ports defined below to
     the define value + Offset.
     e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to 9445
     -->
    <Offset>1</Offset>
    ...
```

- Now we need to deploy our REST gateway configuration into the ESB. Go the `repository/deployment/server/synapse-configs/default` directory of the ESB and delete everything already available in there (files as well as folders). Then copy the contents of the `esb/synapse-config` directory of this sample package into the `default` directory of the ESB.

```
cd esb_home/repository/deployment/server/synapse-config/default
rm -rf *
cp -r /path/to/sample/esb/synapse-config/* .
```

- [Optional Step] It is also advisable to bind the ESB HTTP interface to a proper IP address, before starting the ESB. To do this open the `repository/conf/axis2.xml` file in a text editor and look for the HTTP `transportReceiver` configuration. Uncomment the `bind-address` parameter and set its value to the IP address of the machine. Or at least set it to the loop back address.

```
<transportReceiver name="http" class="org.apache.synapse.transport.nhttp.HttpCoreNIOListener">
    <parameter name="port" locked="false">8280</parameter>
    <parameter name="non-blocking" locked="false">true</parameter>
    <parameter name="bind-address" locked="false">127.0.0.1</parameter>
    <!--parameter name="WSDLEPRPrefix" locked="false">https://apachehost:port/somepath</paramete
    <parameter name="httpGetProcessor" locked="false">org.wso2.carbon.transport.nhttp.api.NHttpG
    <!--<parameter name="priorityConfigFile" locked="false">location of priority configuration f
</transportReceiver>
```

- We are all set now. Head over to the `bin` directory of the ESB and start it up.

```
Unix/Linux: sh wso2server.sh
Windows: wso2server.bat
```

3.4 Running the REST Client

In this section we will look at how to run a REST client against our REST gateway and consume the backend SOAP service. For an explanation on how the gateway is configured please refer REST Gateway Implementation.

All the sample client scripts and example data files are available in the `cleint` directory of this sample package.

- We will start by placing a few orders in our order management system. Run the place_order.py script to make a HTTP POST request to the gateway and place an order.

```
python place_order.py -d "Cafe Misto"
```

- The script will display the exact HTTP request and response being exchanged. If all goes well the REST gateway will respond with a HTTP `201 Created` message.

```
send: 'POST /order HTTP/1.1\r\nHost: 127.0.0.1:8281\r\nAccept-Encoding: identity\r\nContent-Leng
reply: 'HTTP/1.1 201 Created\r\n'
header: Content-Type: application/xml; charset=UTF-8
header: Location: http://127.0.0.1:8281/order/a35464d3-1049-4bba-a001-df534c299bdb
header: Server: WSO2 Carbon Server
header: Date: Thu, 25 Apr 2013 01:54:26 GMT
header: Transfer-Encoding: chunked
response: <order xmlns="http://ws.apache.org/ns/synapse"><drink>Cafe Misto</drink><cost>7.98</co

Order submitted successfully...
Your order can be reviewed at:  http://127.0.0.1:8281/order/a35464d3-1049-4bba-a001-df534c299bdb
```

- Go ahead and place several orders by repeatedly executing the `place_orders.py`. Use the `-d` option to change the name of the drink being ordered.

- Once you have placed a few orders, perform a GET request on the gateway to retrieve a complete list of all orders. This can be done by executing the `get_all_orders.py` script. When executed the REST gateway will respond with a live Atom feed of all the pending orders.

```
python get_all_orders.py
```

- Our backend SOAP service assigns a unique identifier to each order submitted. This identifier can be used to retrieve a specific order through the REST gateway. Extract any order ID value from the output of the Atom feed of the `get_all_orders.py`. Order ID values are embedded into almost all the URLs found in the Atom feed. For instance in the Atom feed you will see URLs like `http://127.0.0.1:8281/payment/order/a35464d3-1049-4bba-a001-df534c299bdb`. The section after the last `/` character, that is `a35464d3-1049-4bba-a001-df534c299bdb` is an order ID. Run the `get_order.py` script as follows to retrieve a single order.

```
python get_order.py -o a35464d3-1049-4bba-a001-df534c299bdb
```

- The order ID can also be used to update a specific order. This is done by making a HTTP PUT request on the remote order resource. Use the `update_order.py` as follows to change the drink name or add some flavor to your drink order.

```
python update_order.py -o a35464d3-1049-4bba-a001-df534c299bdb -d Latte -a Vanilla
```

- You can use the order ID to delete/cancel orders too. This is done via a HTTP DELETE request on the REST gateway. Use the `delete_order.py` as follows to try this out.

```
python delete_order.py -o a35464d3-1049-4bba-a001-df534c299bdb
```

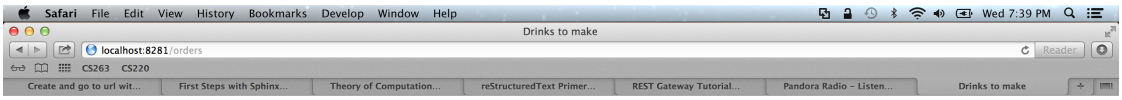
- Try running `get_order.py` on a deleted order ID. You will notice that the gateway responds with a HTTP `404 Not Found` response.

3.5 Testing Content Negotiation

Our REST gateway configuration in Synapse/ESB is capable of performing HTTP content negotiation. That is if the client indicates which content type it prefers for the response messages, the gateway can attempt to generate the response messages in that specified format. Client can indicate its preferred contnt type by sending a media type name in the HTTP `Accept` header of the request. Synapse has been configured to look at this header value and send the response in the client preferred format. To try this out, run the `get_all_orders.py` as follows with the `-f` option.

```
python get_all_orders.py -f "application/xml"
python get_all_orders.py -f "application/json"
python get_all_orders.py -f "text/html"
```

Notice how the Synapse will change the response format depending on a media type specified by the client. You can also try this out by browsing to the URL `http://localhost:8281/orders` using a web browser. Most web browsers send `Accept: text/html` header, so Synapse will send a valid HTML output that can be rendered on a browser.

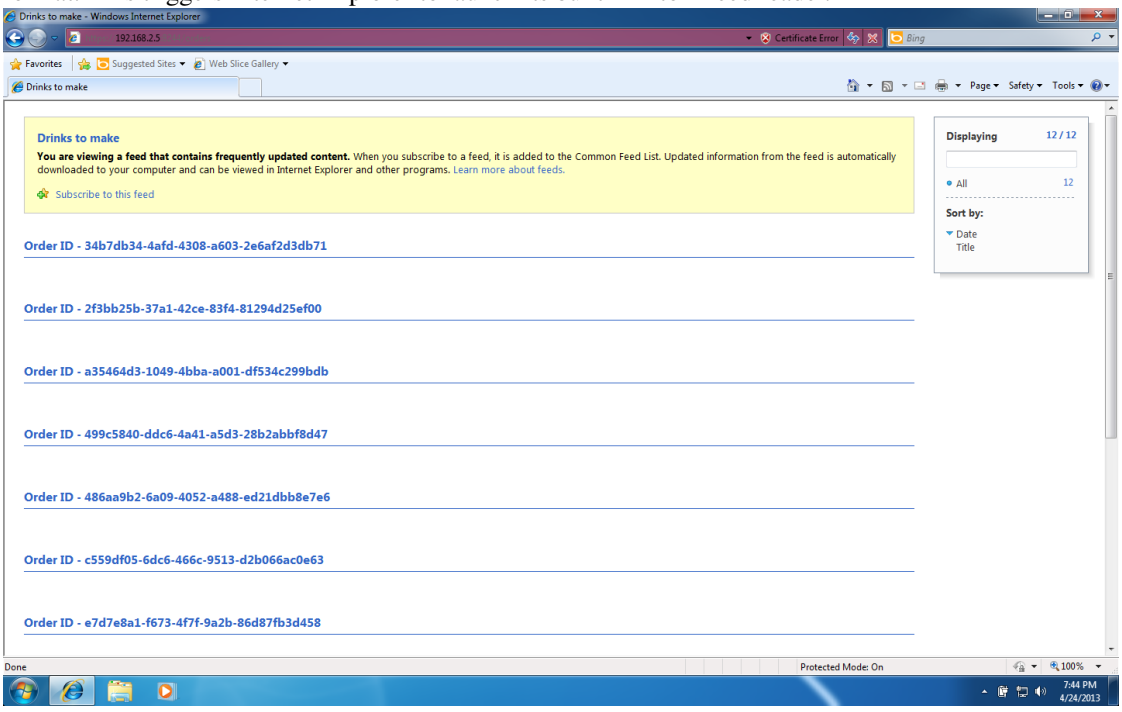


Pending Orders

ID	urn:starbucks:barista:coffees-to-make
Author	Barista System
Updated	2013-04-24T07:39:12Z

Order ID	Drink Name	Additions	Cost	URL
34b7db34-4afd-4308-a603-2e6af2d3db71	Caffe Misto		4.99	34b7db34-4afd-4308-a603-2e6af2d3db71
2f3bb25b-37a1-42ce-83f4-81294d25ef00	Frapacinno	vanilla	7.98	2f3bb25b-37a1-42ce-83f4-81294d25ef00
a35464d3-1049-4bba-a001-df534c299bdb	Frapacinno	vanilla	7.98	a35464d3-1049-4bba-a001-df534c299bdb
499c5840-ddc6-4a41-a5d3-28b2abbf8d47	None		2.99	499c5840-ddc6-4a41-a5d3-28b2abbf8d47
486aa9b2-6a09-4052-a488-ed21dbb8e7e6	None		2.99	486aa9b2-6a09-4052-a488-ed21dbb8e7e6
c559df05-6dc6-466c-9513-d2b066ac0e63	None		2.99	c559df05-6dc6-466c-9513-d2b066ac0e63
e7d7e8a1-f673-4f7f-9a2b-86d87fb3d458	Caffe Misto		4.99	e7d7e8a1-f673-4f7f-9a2b-86d87fb3d458
2be111a4-caf4-4613-8d80-58ed45370a09	None		2.99	2be111a4-caf4-4613-8d80-58ed45370a09
b7b17013-b6fb-4c6c-a89a-6ff92e25de42	None		2.99	b7b17013-b6fb-4c6c-a89a-6ff92e25de42
b40f9b6d-6454-4ba4-8469-8a5496b05026	Cafe Misto		4.99	b40f9b6d-6454-4ba4-8469-8a5496b05026
91d4d380-75d3-4fd5-85c4-80eb6f158e55	Frapacinno	vanilla	7.98	91d4d380-75d3-4fd5-85c4-80eb6f158e55
56acd99e-e8b0-4ad2-bb8d-35420fb3be02	None		2.99	56acd99e-e8b0-4ad2-bb8d-35420fb3be02

Internet Explorer doesn't seem to be sending this header, so in that case Synapse defaults to Atom as the response format. This triggers Internet Explorer to launch its built-in Atom feed reader.



3.6 Tracing Messages

Use a tool like TCPMon to trace the messages between client and ESB and ESB and Application Server. This will give a clear idea of the actual content transformations performed by the ESB on each request-response invocation.

CHAPTER 4

REST Gateway Implementation

In this section we will look at how our REST-to-SOAP gateway is implemented. We will delve into the specifics of the Synapse configuration which makes the transformation of REST to SOAP and back possible.

Our gateway demo consists of 2 REST API configurations.

- StarbucksOrderAPI - Handles placement, retrieval, updating and cancellation of individual orders
- StarbucksOrderListAPI - Handles the retrieval of all pending orders

4.1 Order API (StarbucksOrderAPI)

This API is anchored at the `/order` context. It consists of 2 resources, one to handle POST requests and another to handle GET, PUT and DELETE requests.

```
<api context="/order" name="StarbucksOrderAPI">
  <resource methods="POST" url-mapping="/">
    ...
  </resource>
  <resource methods="GET PUT DELETE" uri-template="/{orderId}">
    ...
  </resource>
</api>
```

According to this configuration, a POST request on the `/order` URL path would be handled by the first resource. Any GET, PUT or DELETE requests on the `/order/{orderId}` path would be handled by the second resource. The `/{orderId}` segment in the URL path is called a template variable, and the client should fill that part in when invoking this API. That is the actual URL path of the second resource should be something like `/order/my-order-id` where the string `my-order-id` fills in the `orderId` variable. Synapse makes it possible to access URI template variables through the Synapse configuration language. We will shortly see how we can use the `orderId` value sent by the client in our request processing logic. (Refer RFC6570 to learn more about URI templates)

The first resoruce accepts a simple XML document as its POST payload and transforms it into a SOAP placeOrder request which we can send to our backend order management service. This transformation is done using a `payloadFactory` mediator.

```
<payloadFactory>
  <format>
      <ucsb:addOrder>
        <ucsb:order>
          <xsd:name>$1</xsd:name>
          <xsd:additions>$2</xsd:additions>
        </ucsb:order>
```

```
      </ucsb:addOrder>
  </format>
  <args>
    <arg expression="//sb:drink" />
    <arg expression="//sb:additions" />
  </args>
</payloadFactory>
```

The `format` section defines the structure of the the placeOrder SOAP request. It defines two variables `$1` and `$2` which needs to be filled by the values extracted from the original RESTful service request. The `args` section defines how to extract these values from the REST request using simple XPath expressions.

We do a similar back transformation from SOAP to REST in the out-sequence of the resource configuration. Also since the SOAP service always responds to the ESB with `200 OK` responses, we use a `proeprty` mediator to change it to `201 Created`.

```
<property name="HTTP_SC" scope="axis2" value="201" />
```

The second resource that handles GET, PUT and DELETE requests is conceptually similar to the first one but there is more mediation logic. We use a `switch` mediator to differentiate between HTTP methods and handle them using separate sequences. GET requests are transformed into SOAP getOrder requests, PUT requests are transformed into SOAP updateOrder requests and DELETE requests are transformed into SOAP deleteOrder requests. These SOAP requests require the order ID to be passed in, therefore we extract the order ID from the URI template defined in our resource. This is done by executing a built-in Synapse XPath extension named `$ctx:uri.var.*`.

```
<payloadFactory>
  <format>
    <ucsb:getOrder>
      <ucsb:orderId>$1</ucsb:orderId>
    </ucsb:getOrder>
  </format>
  <args>
    <arg expression="$ctx:uri.var.orderId" />
  </args>
</payloadFactory>
```

In this case we use the XPath expression `$ctx:uri.var.orderId` to extract the value of the `orderId` template variable and put it in a getOrder SOAP request.

4.2 Order List API (StarbucksOrderListAPI)

This API is anchored at the `/orders` context and consists of a single resource that handles HTTP GET requests.

```
<api name="StarbucksOrderListAPI" context="/orders">
  <resource methods="GET" faultSequence="StarbucksFault">
    ...
  </resource>
</api>
```

One of the first things this resource does is extracting the value of the HTTP `Accept` header and storing it in a Synapse property variable.

```
<property name="STARBUCKS_ACCEPT" expression="$trp:Accept" />
```

We use this value later in the out-sequence to serialize the output into a format preferred by the client (content negotiation). Once the required values have been extracted from the request, Synapse transforms the RESTful GET request into a SOAP getAllOrders request.

```
<payloadFactory>
  <format>
    <ucsb:getAllOrders />
  </format>
</payloadFactory>
```

In the out-sequence we run a `switch` mediator on the value we extracted from the `Accept` header of the request, and formats the message into the client preferred output format.

```
<switch source="$ctx:STARBUCKS_ACCEPT">
  <case regex=".*atom.*">
    ...
  </case>
  <case regex=".*text/html.*">
    ...
  </case>
  <case regex=".*json.*">
    ...
  </case>
  <case regex=".*application/xml.*">
    ...
  </case>
  <default>
    ...
  </default>
</switch>
```

Note that based on the value of the `Accept` header we can serialize the output in one of XML, JSON, HTML or Atom formats. If the `Accept` header is not specified or the client requests for a format that we don't support, we fall back to Atom. The exact transformations from SOAP to HTML and SOAP to Atom are performed using the `xslt` mediator. SOAP to JSON and SOAP to POX (XML) transformation are naturally supported by Synapse without any additional mediators.

4.3 Error Handling

Synapse supports a concept of fault sequences which provides try-catch semantics in the mediation engine. A special fault sequence can be registered with each message flow, service or API which gets triggered when an unexpected error condition occurs. One such error condition that may occur in our demo application is that the client invoking the StarbucksOrderAPI with an invalid order ID value. When this value is sent to the backend SOAP service it sends an error response. A special fault sequence has been defined in Synapse to handle this situation and respond to the user with a `404 Not Found` response. Another fault sequence catches all other unexpected runtime errors and responds to the user with a `500 Internal Server Error` response.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search