Course Exercises Guide

Supporting REST and JOSE in IBM DataPower Gateway V7.5

Course code WE752 / ZE752 ERC 1.0

IBM Training

September 2016 edition

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Contents

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

DataPower®	DB™	developerWorks®
Rational®	Redbooks®	Tivoli®
WebSphere®	400®	

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

VMware and the VMware "boxes" logo and design, Virtual SMP and VMotion are registered trademarks or trademarks (the "Marks") of VMware, Inc. in the United States and/or other jurisdictions.

Other product and service names might be trademarks of IBM or other companies.

Exercises description

FLY airline case study

The exercises in this course build upon a common case study: the FLY airline services. The services are composed of a Booking Service web service and a Baggage Service web service. The services are implemented as a BookingServiceBackend MPGW and a BaggageStatusMockService MPGW, both running within the FLYService domain.

The Booking Service has one operation: BookTravel. The SOAP request that is named BookingRequest contains billing details, payment card details, booking type, and the reservation code. The SOAP response is a BookingResponse, which contains the confirmation code and much of the original message. The endpoint is:
`http://<dp_internal_ip>:9080/BookingService/`

The Baggage Service has two operations:

- BaggageStatus. The SOAP request that is named BaggageStatusRequest contains the passenger's last name and their reference number. The SOAP response is BaggageStatusResponse, which contains the status of each bag that is attached to the passenger's reference number.
- BagInfo. The SOAP request that is named BagInfoRequest contains the ID number of the bag in question. The SOAP response is BagInfoResponse, which contains the status of the bag and which passenger it belongs to. The Baggage Service does not have a WSDL, and cannot be proxied by using a web service proxy. The endpoint is:
  `http://<dp_internal_ip>:2068/BaggageService/`

Technically, the FLY airline services are self-contained MPGWs that mimic a web services back end that might be on WebSphere Application Server.

This application minimizes its dependencies on data sources by relying on data from a flat file, and allowance of read-only operations.

Exercises

This course includes the following exercises:

- **Exercise 1**: Using DataPower to implement REST services

  Create an MPGW that supports a JSON payload. Use SoapUI, the probe, and the CLI debugger to test and observe the behavior. Add several rules to support a REST request from a client to the back-end web services.

- **Exercise 2**: Create and verify a JWS

  Upload the key materials and import the crypto objects that are used for signing and encrypting. Create an MPGW that creates a JWS. Use cURL to send a JSON payload to the MPGW that returns a JWS. Create an MPGW that verifies and decrypts, and sends the request to the back-end service. Use cURL to send the JWS to this verification MPGW. Configure for compact serialization and JSON serialization.

- **Exercise 3**: Create and decrypt a JWE

Add support to the MPGWs to generate a JWE, and decrypt the JWE. Generate a JWS and encrypt it into a JWE. Decrypt the JWE into a JWS, and verify the signature. Test with cURL.

- **Exercise 4**: Use GatewayScript to work with a JWS and a JWE

  Use the classes and methods in the jose module to write GatewayScripts that manipulate a JWS and a JWE.

Note

The lab exercises were written on DataPower V7.5.1.1 firmware. A consistent problem at this level is a failure of the “save configuration” operation in the WebGUI. If this failure happens to you, a workaround exists. Instead of clicking “save configuration”, click “review changes” instead. Scroll to the bottom of the Review Configuration Changes page and click **Save Config**.

In the exercise instructions, you see that each step has a blank preceding it. You might want to check off each step as you complete it to track your progress.

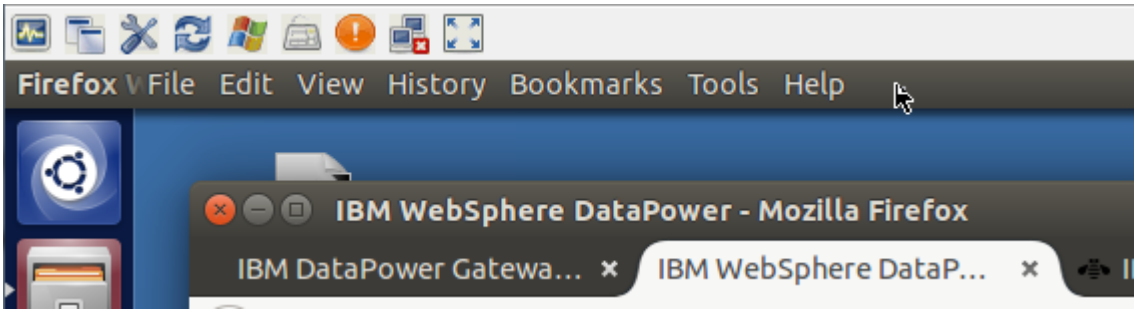
Most exercises include required sections, which must always be completed. These exercises might be required before doing later exercises. If you have sufficient time and want an extra challenge, some exercises also include optional sections that you might want to do.

If you are using the IBM remote lab environment:

As of September 2016, the environment that is used to support the IBM-supplied images and DataPower gateways is Skytap. Each student is supplied an Ubuntu student image and a DataPower gateway.

- Ignore all offers to upgrade any of the software on the image. The image and exercise steps are designed to operate at the supplied levels of the contained software.
- The supplied image is Ubuntu 14.04 LTS. The desktop uses Unity, which is different from the more common Gnome desktop. Some hints on using Unity are at: http://www.howtogeek.com/113330/how-to-master-ubuntus-unity-desktop-8-things-you-need-to-know/
- A noticeable difference is that the menus on the windows within the desktop are not typically visible. When a window is the “active” window, that window does not have any menu items, but the application type is displayed in the black bar that spans the top of the desktop. In the following screen capture, observe that the browser window does not have a menu bar, and that its type of “Firefox Web Browser” is listed in the black bar.

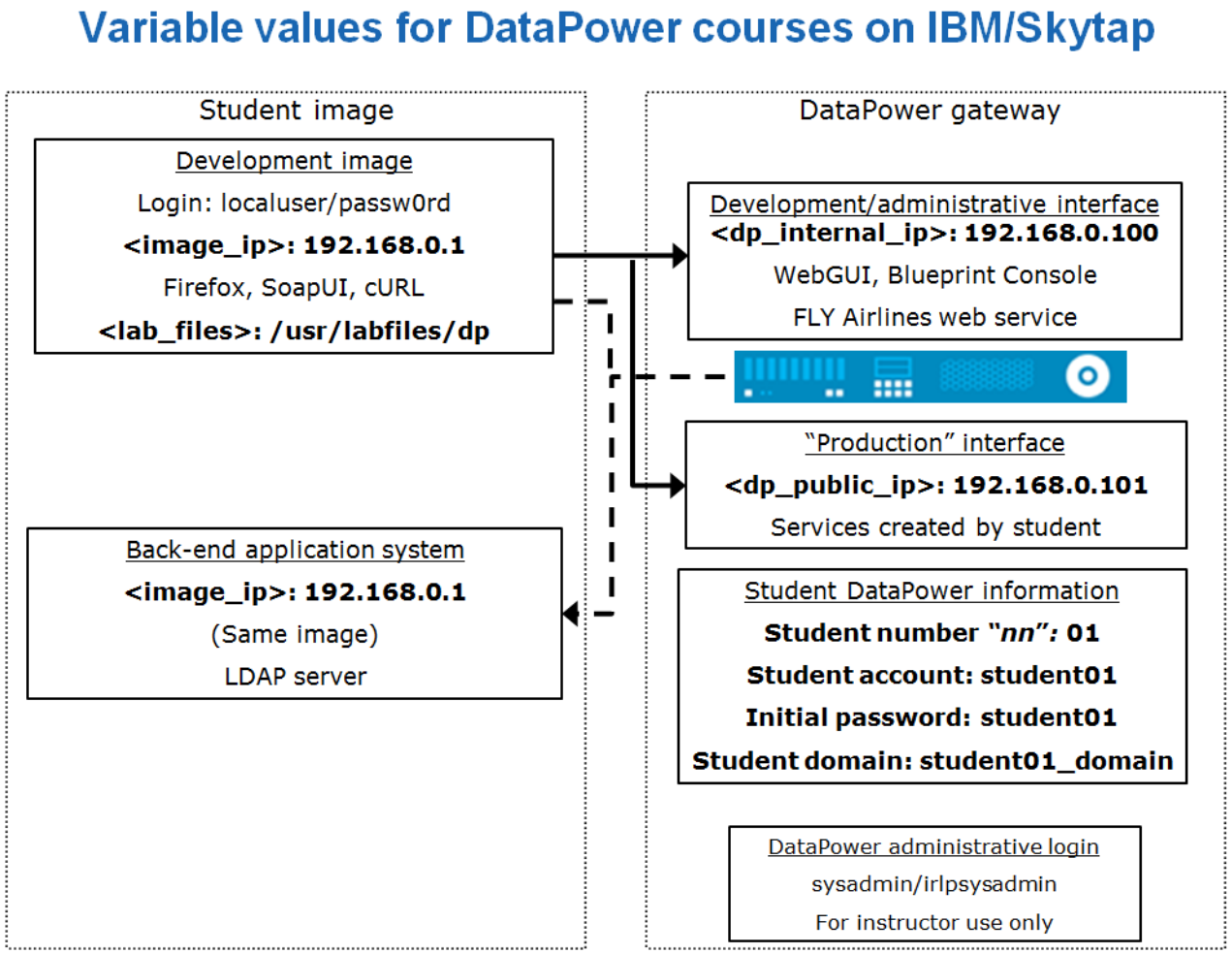
If you hover the mouse over the black bar, the menu items for the active window are displayed.



Another noticeable difference is that when a window is maximized, the Close/Minimize/Restore buttons are not visible until you hover the mouse in the black bar.

This Unity behavior is discussed in the “Hidden Global Menus” section of the previously mentioned howtogeek.com page.

- The IBM supplied environment has values that are preassigned for some of the variables, such as the IP addresses of the student image and the DataPower gateway, the student number, and the initial password. The following graphic shows those assignments.



Important

Online course material updates might exist for this course. To check for updates, see the Instructor wiki at: http://ibm.biz/CloudEduCourses

Exercise 1. Using DataPower to implement REST services

Estimated time

01:30

Overview

This exercise shows you how to use the DataPower gateway to expose web services with JSON data and a REST interface. You learn how to validate JSON input against a JSON schema. The request and response data is transformed between JSON and SOAP by using GatewayScript code, stylesheets, and XQuery/JSONiq code. A GatewayScript parses the typical REST URI input parameters and converts them to a SOAP request format. The system log, probe, and CLI debugger are used to observe and debug the configuration.

Objectives

After completing this exercise, you should be able to:

- Create a service policy to handle JSON and REST requests and responses
- Use a GatewayScript to build a SOAP request from HTTP query parameters or JSON
- Enable and use the CLI debugger
- Define and use stylesheet parameters
- Convert a SOAP response to a JSON-formatted data structure by using XQuery/JSONiq

Introduction

JSON is a lightweight and easily consumed approach to structuring data. Clients are moving to JSON for its simplicity, and for the variety of tools and languages that can work with it. The first part of the exercise demonstrates how to support handling JSON data in the HTTP body. JSON by itself does not make it RESTful. In fact, the beginning JSON data part of the exercise does not use a REST-based URL.

The next step beyond JSON is REST. A key benefit of using a REST interface is also its simplicity. Resources are described in the URL, and the action is specified from the HTTP method. For example, the following HTTP methods can perform the actions that are indicated:

- GET: Fetches a resource from the server.
- DELETE: Removes a resource from the server.
- PUT: Modifies, or overwrites, an existing resource. It also can be used to create a resource at a specific URL.
- POST: Creates a resource in the collection.

IBM DataPower Gateways provide functions for supporting JSON data and for enabling a REST interface in front of an existing web service. You can create a multi-protocol gateway service that accepts an HTTP request without any XML, and builds a SOAP request with the original request parameters. The service can also take the SOAP response and convert it to a REST-based response, such as JSON. The second part of the exercise has the students create rules that receive a REST GET request and convert it into a SOAP request that goes to the FLYServices back end. The back end returns a SOAP response, so the response rules must be configured to convert the SOAP to a JSON structured response. The exercise provides several stylesheets, GatewayScripts, and XQuery/JSONiq scripts that demonstrate parsing and building SOAP requests and JSON responses.

Requirements

- Access to the DataPower gateway
- Completion of the previous exercises (see the Preface section in the exercise instructions for details)
- SoapUI, for sending requests to the DataPower gateway
- The BaggageStatusMockService web service that runs on the DataPower gateway in the `FLYServices` domain
- Access to the `<lab_files>` directory

Important

The exercises in this course use a set of lab files that might include scripts, applications, files, solution files, PI files, and others. The course lab files can be found in the following directory:

`C:\labfiles` for the Windows platform

`/usr/labfiles` directory the Linux platform

The exercises point you to the lab files as you need them.

Exercise instructions

Preface (optional)

All exercises in this chapter depend on the availability of specific equipment in your classroom.

- <*lab_files*>: Location of the student lab files. Default location is: `/usr/labfiles/dp/`
- <image_ip>: IP address of the student image (use `/sbin/ifconfig` from a terminal window to obtain value).
- <*dp_internal_ip*>: IP address of the DataPower gateway development and administrative functions that are used by internal resources such as developers.
- <*dp_public_ip*>: IP address of the public services on the gateway that is used by customers and clients.
- <*dp_WebGUI_port*>: Port of the WebGUI. The default port is `9090`.
- <*nn*>: Assigned student number. If the course has no instructor, use "`01`".
- <*studentnn*>: Assigned user name and user account. If the course has no instructor, use "`student01`".
- <*studentnn_password*>: Account password. In most cases, the initial value is the same as the user name. You are prompted to create a password on first use. Write it down.
- <*studentnn_domain*>: Application domain that the user account is assigned to. If the course has no instructor, use "`student01_domain`".
- <*FLY_baggage_port*>: Port number that the back-end BaggageServices web services listen on. The default port is 2068.
- <*mpgw_baggage_port*>: `12`*nn*`9` where "*nn*" is the two-digit student number. This port number is the listener port of the BaggageServiceProxy that mediates between the REST or JSON client and the Baggage Services back-end application.

1.1. Initialize the lab environment

Some setup activities are required to properly configure the lab environment and determine IP addresses and ports.

__ 1. If you did not yet complete the setup activities, you must go to **Appendix B: Lab environment setup**. Complete those activities before proceeding.

These activities need to be done only once for the course.

1.2. Compare the REST interface to the SOAP interface of the back-end web service

In this section, you compare what the back-end web service supports as a request and response with what is used for a JSON or a RESTful interface.

Currently, GET-type requests are supported only on the back-end web service.

BaggageStatus request and response

__ 1. The SOAP request to find the bags that belong to a passenger looks like the following XML:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:fly="http://www.ibm.com/datapower/FLY/BookingService/" >
  <soapenv:Header/>
  <soapenv:Body>
    <fly:BaggageStatusRequest>
      <fly:refNumber>
        refNo
      </fly:refNumber>
      <fly:lastName>
        lastName
      </fly:lastName>
    </fly:BaggageStatusRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

__ 2. The REST request that equates to this SOAP request is an HTTP GET:

`http://servername:port/BaggageService/Passenger/Bags`**`?refNumber=11111&lastName=Johnson`**

In this URL, you are requesting information on the baggage that belongs to a specific passenger. The search parameters identify the reference number for the particular passenger, and that passenger's last name.

__ 3. The SOAP response for such a query is:

```
<soapenv:Envelope
    xmlns:fly="http://www.ibm.com/datapower/FLY/BaggageService/"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" >
  <soapenv:Header />
  <soapenv:Body>
    <fly:BaggageStatusResponse>
      <fly:refNumber>11111</fly:refNumber>
      <fly:firstName>James</fly:firstName>
      <fly:lastName>Johnson</fly:lastName>
      <bag xmlns="http://www.ibm.com/datapower/FLY/BaggageService/" >
        <id>1589</id>
        <destination>LHR</destination>
        <status>On Belt</status>
        <lastKnownLocation>DEL</lastKnownLocation>
        <timeAtLastKnownLocation>Thu Jun 19 07:27 UTC
2014</timeAtLastKnownLocation>
      </bag>
      <bag xmlns="http://www.ibm.com/datapower/FLY/BaggageService/" >
        <id>1289</id>
        <destination>LHR</destination>
        <status>On Belt</status>
        <lastKnownLocation>DEL</lastKnownLocation>
        <timeAtLastKnownLocation>Thu Jun 19 07:27 UTC
2014</timeAtLastKnownLocation>
      </bag>
    </fly:BaggageStatusResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

__ 4. The REST response, in the JSON format, that matches the SOAP response is:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1325"
    },
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1589"
    }
  ],
  "refNumber":"11111"
}
```

BagInfo request and response

__ 1. You can request the information on a specific bag by its ID number:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:fly="http://www.ibm.com/datapower/FLY/BaggageService/" >
  <soapenv:Header/>
  <soapenv:Body>
    <fly:BagInfoRequest>
      <fly:id>1986</fly:id>
    </fly:BagInfoRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

__ 2. The HTTP GET request from a RESTful perspective is:

`http://`*`servername`*`:`*`port`*`/BaggageService/Bag?id=1589`

This request is for information on a specific bag, regardless of which passenger is associated with it.

__ 3. The SOAP response would be:

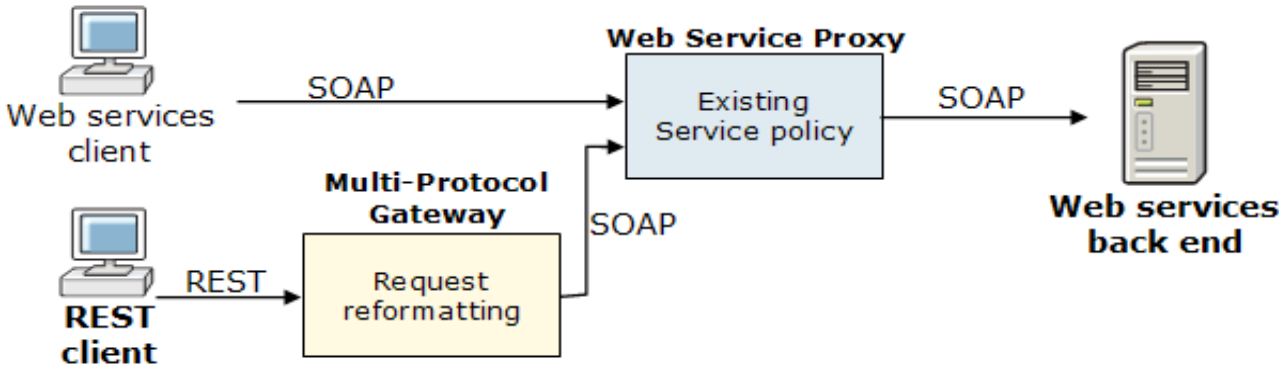
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:fly="http://www.ibm.com/datapower/FLY/BaggageService/" >
   <soapenv:Header/>
   <soapenv:Body>
      <fly:BagInfoResponse>
         <fly:id>12345</fly:id>
         <fly:destination>XYZ</fly:destination>
         <fly:status>blah blah</fly:status>
         <fly:lastKnownLocation>ZYX</fly:lastKnownLocation>
         <fly:timeAtLastKnownLocation>12:00</fly:timeAtLastKnownLocation>
         <fly:refNumber>11111</fly:refNumber>
         <fly:lastName>My Name</fly:lastName>
      </fly:BagInfoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

__ 4. The related REST response that also uses JSON is:

```
{
  "status":"On Belt",
  "lastName":"Johnson",
  "destination":"LHR",
  "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
  "lastKnownLocation":"DEL",
  "refNumber":"11111",
  "id":"1589"
}
```

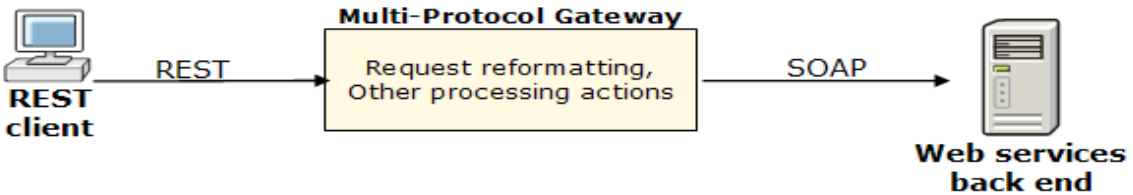
1.3. The high-level design of the REST service

If another DataPower service is proxying the back-end web service, such as a web service proxy or a multi-protocol gateway, the service design might look like the facade pattern that was mentioned in the presentation.



In this pattern, an existing web service proxy is already in front of the back-end web services. This web service proxy might provide services that are based on DataPower, such as authentication, routing, and transformation. To allow the REST clients to use the same services as the SOAP clients, the multi-protocol gateway service that provides the REST interface sits in front of the web service proxy. Any enhancement to the web service proxy for the SOAP client also becomes available to the REST client without any change to the REST multi-protocol gateway.

You can use a bridge pattern in a situation in which you have no existing SOAP clients for the web service and are not planning to provide a SOAP interface through the gateway.



The conversion of the REST request to a SOAP request and the response conversion occurs within the multi-protocol gateway. Any other necessary enhancements, such as authentication, happen within this service.

This **bridge pattern** is the design that is used in this exercise.

1.4. Create a multi-protocol gateway service that handles a JSON request

The client wants to support an HTTP request that passes JSON data, but one that is not yet in a RESTful format. In this section, you create a BaggageServiceProxy MPGW. This service converts the HTTP request that carries JSON data into the SOAP request that the back end expects, and convert the SOAP response to a response that contains the JSON data that the client expects.

__ 1. Connect to the DataPower WebGUI:

`https://<dp_internal_ip>:<dp_WebGUI_port>`

__ 2. Log in with your student account `<studentnn>` and password `<studentnn_password>`, and select your student domain `<studentnn_domain>`.

Note

Since the Blueprint Console is new and still a “technology preview”, it still has a few bugs. You are going to use the WebGUI to configure the resources in this exercise.

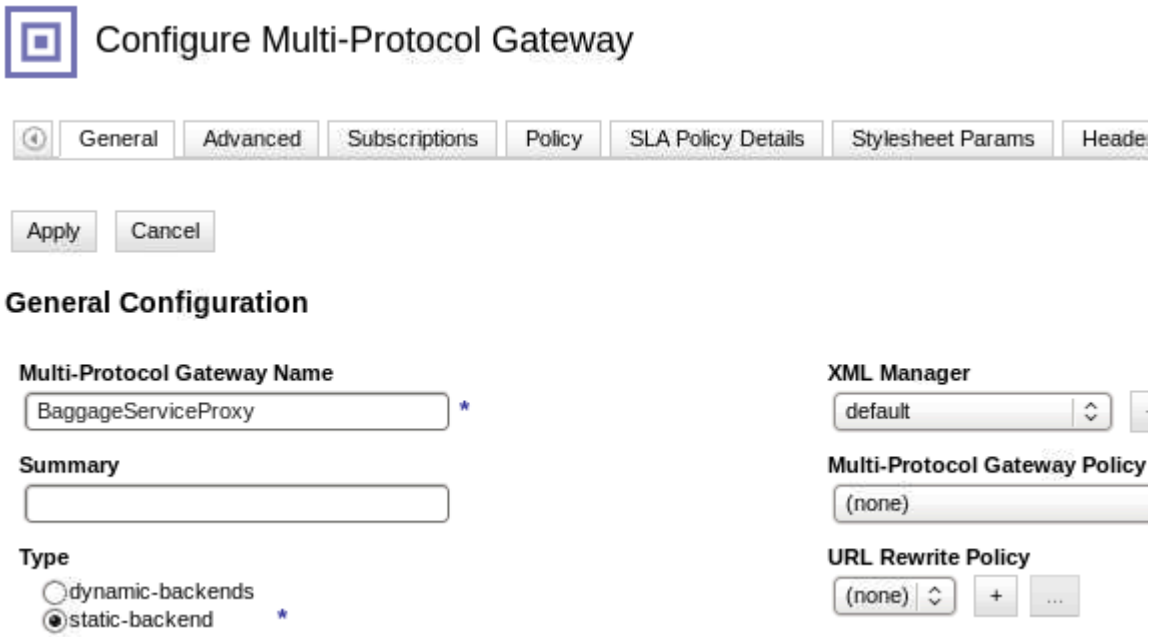
__ 3. Click the **Multi-Protocol Gateway** icon.

__ 4. The Multi-Protocol Gateway catalog list appears. Click **Add** to configure a new MPGW.

__ 5. Configure the multi-protocol gateway service.

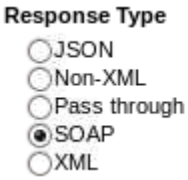
__ a. Modify the service with the following configuration:

- Enter the following name: `BaggageServiceProxy`
- Use the existing **default** XML manager.
- Set the service type to **static-backend**.



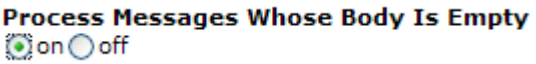
__ b. Set the message types:

- Scroll down to the **Request Type** and select **JSON**. The client sends a JSON-formatted request.
- Scroll down to the **Response Type** and select **SOAP**. The response from the back-end web service is a SOAP message.



__ c. Specify the **Default Backend URL** as:
`http://<dp_internal_ip>:<FLY_baggage_port>/BaggageService`

__ d. On the **Advanced** tab, set the **Process Messages Whose Body Is Empty** setting to **on**, which is necessary for a RESTful interface (to be coded in later sections of the exercise).



__ 6. Back on the **General** tab, create an HTTP front side handler.

__ a. Click the “+” (new) icon to create a handler.

__ b. Select **HTTP Front Side Handler** from the list.

__ c. Configure the HTTP front side handler object with a name of `http_fsh_Baggage_12nn9` and with the port number *<mpgw_baggage_port>*.

__ d. Have the handler listen on the *<dp_public_ip>* interface.

__ e. Make sure that you select the **POST**, **GET**, **PUT**, **HEAD**, **DELETE**, and **URL with ?** check boxes. Click **Apply**. The handler window closes.

__ f. Click **Add** to add the new handler to the Front Side Protocol list.

Note

Although the first client request is not a REST request, you configure the handler to support the standard HTTP methods that are needed for a REST interface. They are used in later sections.

__ 7. Create the multi-protocol gateway service policy.

__ a. Click the plus sign (**+**) button in the Multi-Protocol Gateway Policy field to create a service policy.

__ b. Enter the following policy name: `BaggageServicePolicy`

__ c. Click **Apply Policy**.

__ d. Click **New Rule**.

__ e. Enter the following name: `BaggageServicePolicy_JSON_req`

__ f. Specify the rule direction as **Client to Server**.

__ 8. Configure a match action that matches on any URI.

__ a. Double-click the **Match** action icon.

__ b. Create a matching rule `MatchAnyURI` that matches on a URL of "*".

__ c. Click **Done** to save the Match action.

__ 9. Add a Validate action. This action validates the incoming JSON data structure against a JSON schema.

__ a. Drag the **Validate** action to the rule configuration path after the **Match** action.

__ b. Double-click the icon to configure it.

__ c. Select **Validate Document via JSON Schema URL;** the page refreshes

__ d. Upload `<lab_files>/REST/BaggageStatus-schema.json`.

__ e. To the right of the **Upload** button, click **View**.

__ f. This file is the JSON schema that the input document should adhere to. It is composed of an integer passenger reference number, and the passenger's last name as a string. Both fields are required.

__ g. Close the View window.

__ h. Click **Done** to complete the Validate action configuration.

__ 10. Add a GatewayScript action to change the incoming JSON data structure to a SOAP request.

__ a. In the policy editor, drag a **GatewayScript** action to the right of the **Validate** action.

__ b. For the GatewayScipt file, upload `<lab_files>/REST/BaggageJson2Soap.js`.

__ c. Use **View** to see the file contents.

__ d. About 16 lines down into the file, you see a reference to a debugLevel variable. This variable is configurable in the action. It is used to control the amount of debugging information that is written to the system log. The "`console.debug`" statement details what is written to the log at the debug level. You configure this variable in a few steps.

__ e. Near the top, "`session.input.readAsJSON`" reads the INPUT context JSON structure and places it into a "`json`" variable. A few lines further down, the "`json`" variable is used to retrieve the passenger reference number and last name, and place them into a JavaScript variable.

__ f. Near the end of the script, the SOAP message is constructed and placed into the OUTPUT context ("`session.output.write()`").

__ g. Close the View window.

__ h. It is time to set the debugLevel variable from the script. Click the **Advanced** tab.

__ i. Click **Add Parameter**.

__ j. In the new window, enter a **Stylesheet Parameter Name** of `debugLevel` and a **Stylesheet Parameter Value** of `1`.

__ k. Click **Submit**.

__ l. Click **Done** to complete the GatewayScript action.

__ 11. The output of a GatewayScript action is binary (non-XML) data. The back-end service wants a SOAP (XML) request. Use a Transform action and a utility stylesheet to do the conversion.

__ a. Drag a **Transform** action to the rule configuration path after the **GatewayScript** action.

__ b. Double-click the icon to configure it.

__ c. For the Transform File, change the DataPower directory to **store:///**.

__ d. Select the **identity.xsl** file from that directory.

__ e. Click **Done**.

__ 12. The request rule is complete. Click **Apply Policy** to save your work up to this point.

__ 13. Create a response rule.

__ a. Click **New Rule**.

__ b. Specify a rule name of: `BaggageServicePolicy_JSON_reply`

__ c. Specify a rule direction of: `Server to Client`

__ 14. Configure the Match action to use the **MatchAnyURI**.

__ 15. Add a Transform action that uses XQuery and JSONiq to convert the SOAP response to a JSON data structure.

__ a. Drag a **Transform** action to the rule configuration path.

__ b. Double-click it to configure it.

__ c. Set the Document Processing Instructions to **Transform with a processing control file, if specified**.

__ d. Set the Input Language to **XML**.

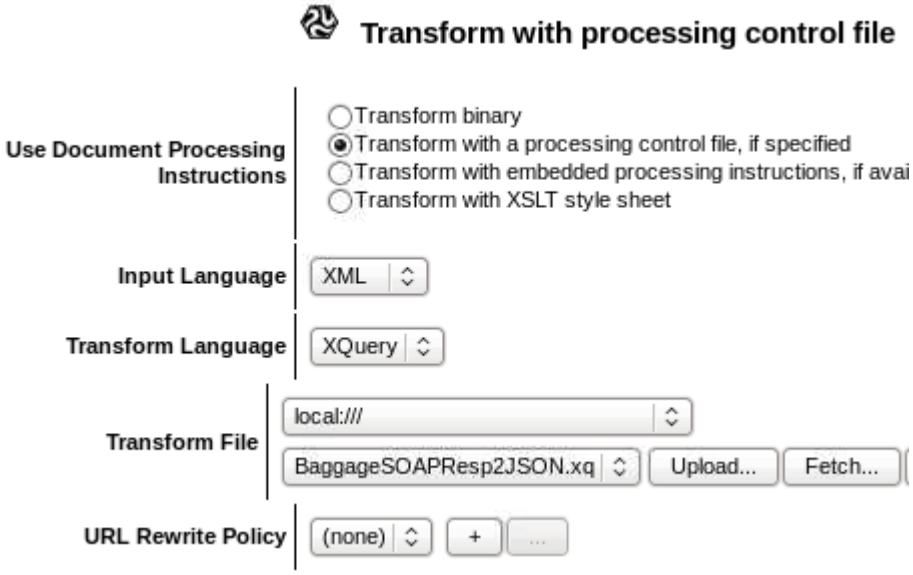
__ e. Set the Transform Language as **XQuery**.

__ f. Upload `<lab_files>/REST/BaggageSOAPResp2JSON.xq`.

__ g. Click **View** to review the file.

__ h. The XQueries pull the elements from the `<BaggageStatusResponse>` and use them to build the JSON structure. They also set the HTTP Content-Type header to “application/json”.

__ i. Close the View window.



__ j. Click **Done** to complete the configuration.

__ 16. Click **Apply Policy** to save the response rule.

__ 17. Click **View Status** in the policy editor to check the state of the service policy and its contained objects.

__ 18. If the Transform action ("xformng") is **down**, follow the steps in the following Troubleshooting block. Otherwise, close the policy editor, click **Apply** to save the MPGW, and continue after the block.

Troubleshooting

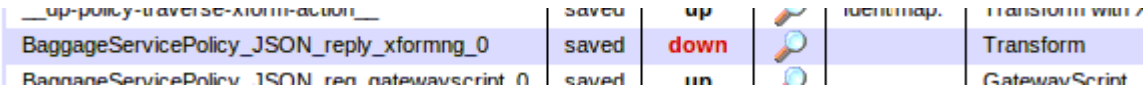
If the **Transform with processing control file** action in the response rule is down, a known bug was not yet fixed. The following steps are a workaround until the fix is delivered in a firmware fix pack.

__ a. Close the policy editor and **Apply** the MPGW.

__ b. Start to enter `processing` into the search field in the navigation bar.

__ c. Select **Processing Action** in the list.

__ d. The catalog of processing actions is displayed. Select the Transform action that is in the **down** state.



__ e. The Objects version of the Action configuration page is displayed. Notice that the **Transform File** field is blank. When you created this Transform action in the policy editor, you did supply this value. The bug is that the value is not saved. You fix that in the next steps.

__ f. Enter the value for the Transform File: `local:///BaggageSOAPResp2JSON.xq`

__ g. Click **Apply**.

__ h. Click **Control Panel > Multi-Protocol Gateway > BaggageServiceProxy**.

__ i. Click **View Status**

__ j. Now the processing policy and Transform action should show as **up**.

__ k. Close the Object status window.

__ 19. Click **Save Configuration**.

1.5. Test the BaggageServiceProxy by sending a JSON request

Use SoapUI to send a JSON request to the BaggageServiceProxy, which converts the JSON request to a SOAP request. After the back-end service completes the request, it returns the SOAP response. The BaggageServiceProxy converts the SOAP response to a JSON structure. Although the BaggageServiceProxy supports a JSON request, it is not yet being handled as a REST request. You update the service in later sections to make it support REST.

__ 1. Before testing, verify that the log level for the system log is at “debug”. You can do that from **Control Panel > Troubleshooting**.

__ 2. Show and enable the probe for the BaggageServiceProxy.

__ 3. Open SoapUI.

__ 4. In the project tree, expand **BaggageServices** until **queryJohnson** is visible.

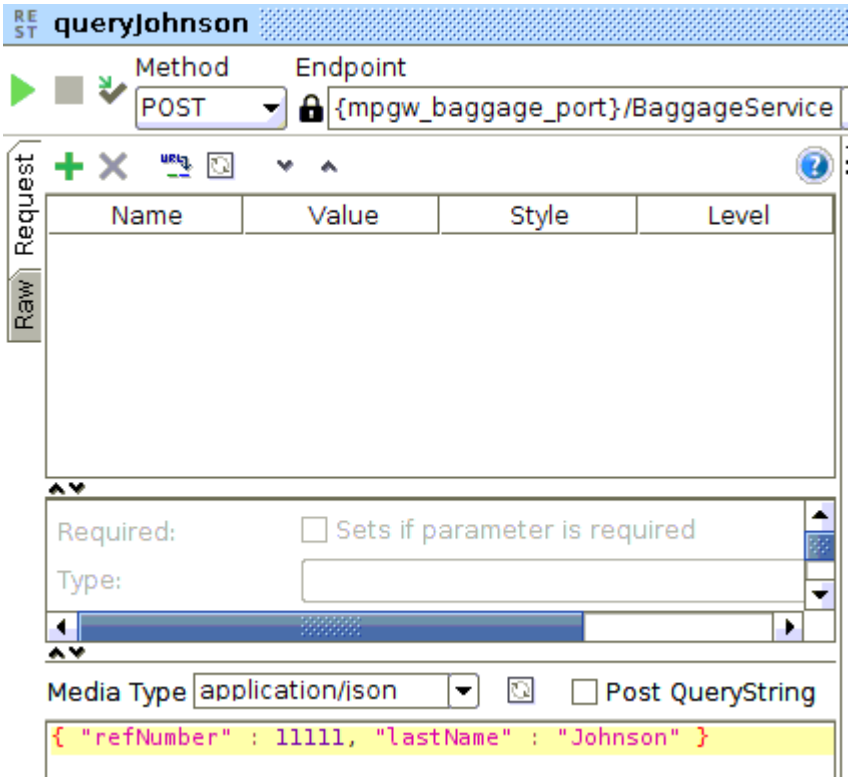
__ 5. In the “queryJohnson” request window, verify that the HTTP method is set to **POST**.

Note

Recall that this request is a request that passes JSON data; it is not yet using a RESTful interface. If it was using a RESTful interface, it would use a GET HTTP request. In a later section, you convert the request to a RESTful one.

__ 6. In the URL address field, update the port number to use the port for your service: `http://${dp_public_ip}:${mpgw_booking_port}/BaggageService`

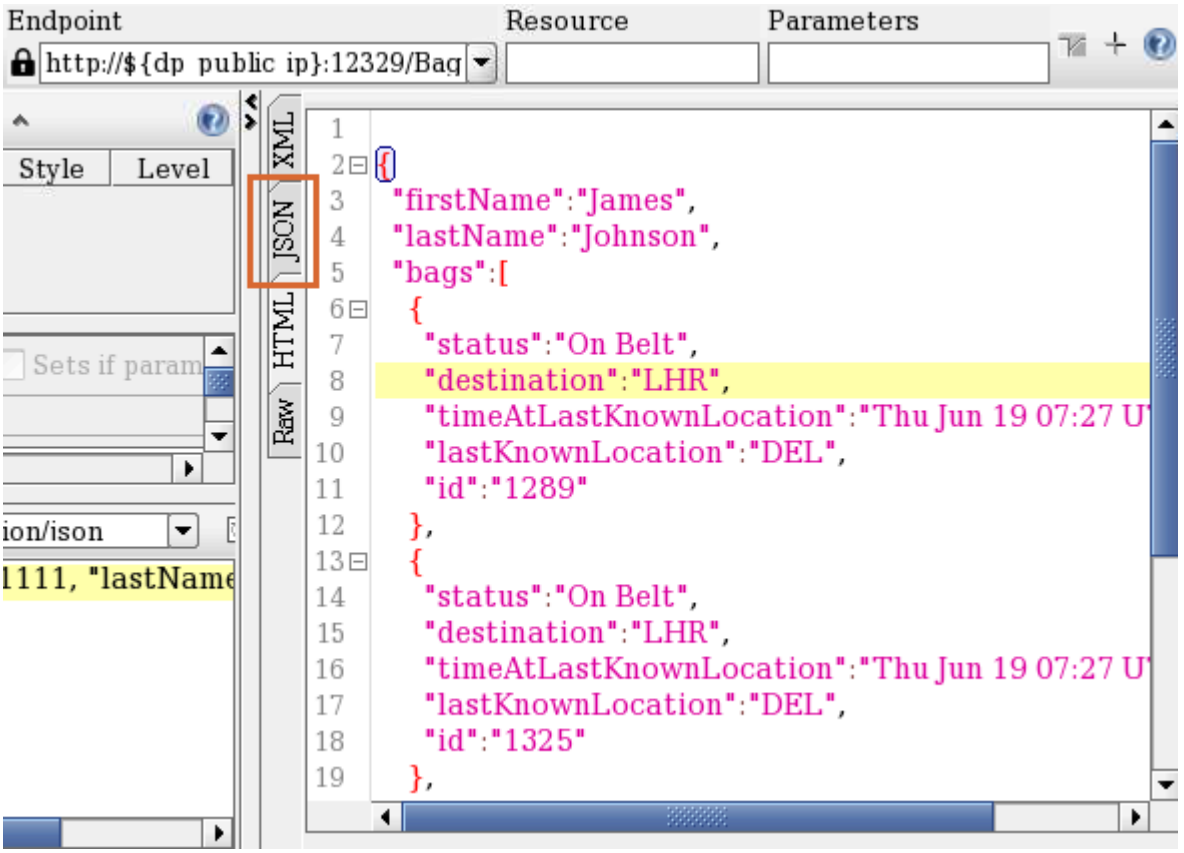
__ 7. In the request tab, the Media Type should be **application/json**, and you should see the JSON array that passes the passenger reference number and last name.



__ 8. Click the green **Submit** arrow to send the request.

__ 9. In the response tab, be sure to click the **JSON** view tab.

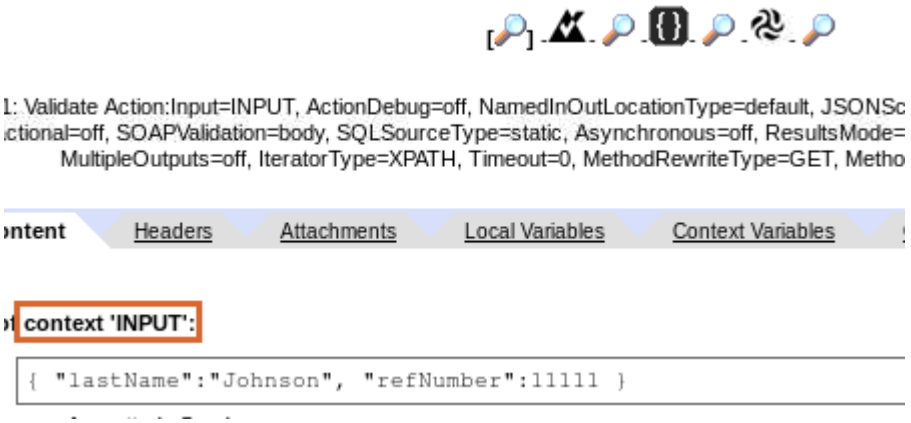
__ 10. The response tab should contain a JSON array of the bag status for all of the bags that are associated with passenger Johnson.



__ 11. In the Transaction List probe window, click **Refresh**.

__ 12. Click the magnifying glass for the request.

__ 13. A probe window for the request opens. You can see the JSON data in the INPUT context.



__ 14. Click the magnifying glass after the Validate action. Because the Validate does not change the message, the JSON data in the INPUT context is still listed.

__ 15. Click the magnifying glass after the GatewayScript action. Because this action returns binary (non-XML) data, the PIPE context displays the related SOAP request data, but as binary data.

__ 16. Click the magnifying glass after the Transform action. The OUTPUT context now shows the SOAP request as formatted XML data.

__ 17. Investigate any other areas of interest for the request that you might have.

__ 18. Close the request probe.

__ 19. In the Transaction List probe window, click the magnifying glass for the response.

__ 20. The INPUT context shows the binary (text) version of the SOAP response message.

__ 21. Click the magnifying glass after the Transform action. Recall that this action is the XQuery/JSONiq processing.

__ 22. The OUTPUT context shows the JSON array that is returned to the client.

__ 23. Close the probe window.

__ 24. If you want to see the log entries for this transaction, click **View Log** on the Transaction List probe window. Close the log window after you finish.

1.6. Test the Validate action in BaggageServiceProxy

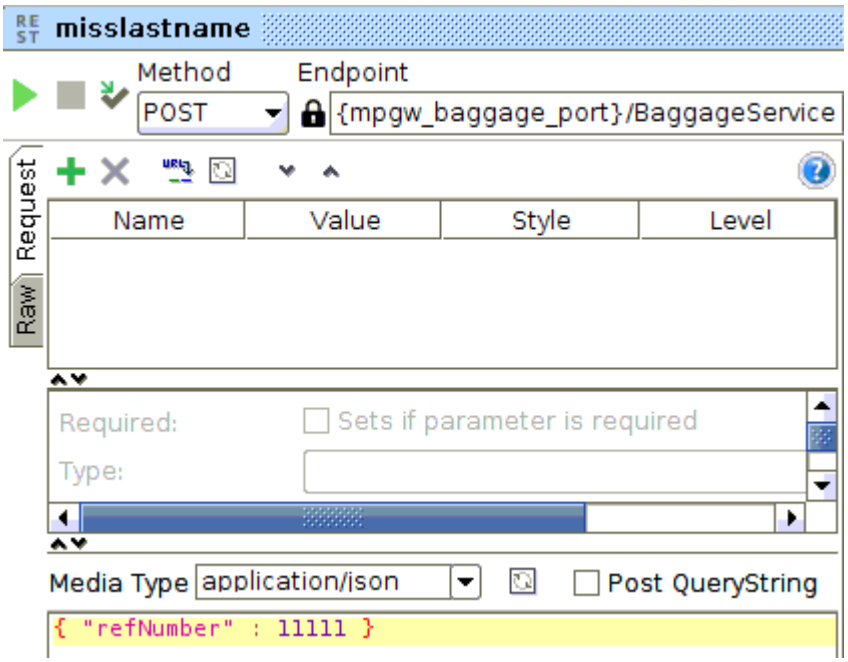
Two more requests are in SoapUI to exercise the JSON schema validation.

__ 1. In SoapUI, double-click the **misslastname** request.

__ 2. In the “misslastname” request window, verify that the HTTP method is set to **POST**.

__ 3. In the URL address field, update the port number to use the port for your service: `http://${dp_public_ip}:${mpgw_booking_port}/BaggageService`

__ 4. In the request tab, the Media Type should be **application/json**, and you should see the JSON array. But in this test, the lastName entry is missing.



__ 5. Click the green **Submit** arrow to send the request.

__ 6. In the XML view tab of the response tab, you get a generic SOAP fault message. Because the service policy has no error handling, the default error handling takes place.

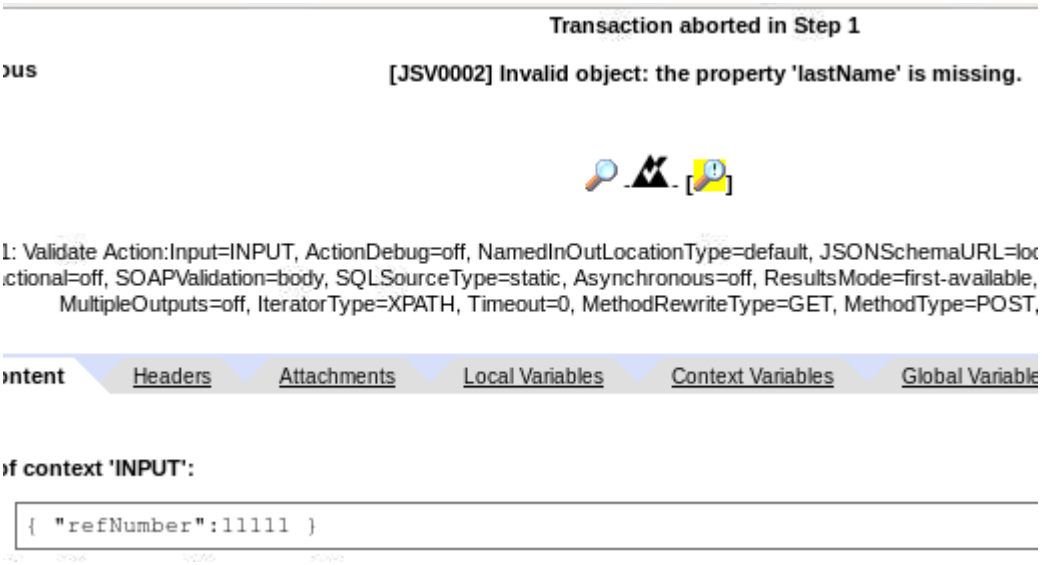
__ 7. In the Transaction List probe window, click **Refresh**.

__ 8. Scroll to the bottom of the transaction list to find the newest entry.

__ 9. Only a single probe entry is available for this transaction because it never sent a request to the back end. Click the magnifying glass for this entry.

__ 10. The probe has only one action, rather than the several actions you saw in the successful execution. Click the magnifying glass after the Validate action.

__ 11. The probe indicates that the transaction was canceled because of an invalid object: “lastName” is missing. As soon as the Validate action failed, error processing was invoked, and the transaction was terminated.



__ 12. Close the probe.

__ 13. In the Transaction List probe window, click **View Log**.

__ 14. Examine the log for another indication of the validation failure.

__ 15. Close the log window.

Optional

Another **refNoString** request is in SoapUI. For this test, the type of refNumber is a string, rather than an integer as specified in the schema. If you want, you can use SoapUI and the probe to see how this situation is handled.

1.7. Use the stylesheet parameter in the GatewayScript action

When you defined the GatewayScript action, you added a stylesheet parameter to it. In this section, you see that parameter in action.

__ 1. In the BaggageServiceProxy MPGW, open the policy editor.

__ 2. Select the request rule in the Configured Rules section. This action puts the selected rule in the editor pane.

__ 3. Double-click the **GatewayScript** action.

__ 4. Click the **Advanced** tab.

__ 5. Change the **debugLevel** field to **5**. If debug is set to 5 or higher, it sends more information to the system log.

__ 6. Click **Done**.

__ 7. Click **Apply Policy** in the policy editor.

__ 8. If necessary, click **Apply** for the MPGW.

__ 9. The log entries do not appear unless the log level is set to **debug**. You can go to **Control Panel > Troubleshooting** to verify or set the log level. Return to the Configure Multi-Protocol Gateway page after you work with the log level.

__ 10. In SoapUI, open the **queryJohnson** request again.

__ 11. Verify the HTTP method as POST, and the endpoint URL as before.

__ 12. Click the green **Submit** arrow.

__ 13. In the Transaction List probe window, click **View Log**.

__ 14. In the Log window, scroll down to the request entries.

__ 15. Look for a group of entries for the category **gatewayscript-user**.

__ 16. These entries appear because the debugLevel of 5 causes the “console.debug” statements to write to the log.

request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): JSON Request is {"lastName":"Johnson","refNumber":11111}
request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): Inbound URL:http://172.16.78.24:12329/BaggageService
request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): Input size is:47
request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): Current headers {"Accept-Encoding":"gzip,deflate","Content-Type":"application/json","Content-Length":"47","Hos (java 1.5)","Via":"1.1 AgAAACYBAAA-","X-Client-IP":"172.16.80.5","X-Global-Transaction-ID":"8
request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): Received request from Johnson and referenece number is 11111
request	172.16.80.5	0x8580005c	mpgw (BaggageServiceProxy): Starting BaggageJson2Soap

__ 17. Another way to find these entries is to set the **Category** field in the Log window to **gatewayscript-user**. This setting filters the messages to show only those messages in this category.

__ 18. Close the Log window when you are finished reviewing the entries.

__ 19. If you want, you can leave the **debuglevel** at 5, or choose to reset it to 1.

1.8. Use the CLI GatewayScript debugger

The GatewayScript support in DataPower includes a CLI-based GatewayScript debugger. In this section, you work with the debugger.

__ 1. You must first enable the debugger support in the GatewayScript action. Open the policy editor.

__ 2. Select the request rule.

__ 3. Double-click the **GatewayScript** action.

__ 4. Click **Enable Gateway Debug**. This selection enables the debugger for this action.

__ 5. Click **Done** to save this setting.

__ 6. Click **Apply Policy** to save the service policy. If it is enabled, click **Apply** at the service level.

__ 7. In SoapUI, open the **queryJohnson** request again.

__ 8. Click the green **Submit** arrow to send the request.

__ 9. Because the GatewayScript has a debugger statement in it, and you enabled debugging, the script “blocks” on the first debugging statement.

__ 10. You run the debugger from the DataPower CLI. In Linux, the CLI is invoked from within a terminal window. Open a new terminal window.

__ 11. In the terminal window, at the command prompt enter ssh *<dp_internal_ip>*. You are using SSH to communicate with the gateway.

__ 12. Log in with your student user name, password, and domain. An example for student 99 is provided here:

```
DP98
Unauthorized access prohibited.
login: student99
Password: ********
Domain (? for all): student99_domain

Welcome to DataPower XI52 console configuration.
Copyright IBM Corporation 1999-2016

Version: XI52.7.5.1.1 build 277888 on Jun 15, 2016 3:28:
Serial number: 0000000
```

__ 13. To work with the “debug-action” command, you need to be in global configuration mode. To enter that mode, enter `co` at the command prompt.

__ 14. Enter `show debug-actions`

__ 15. Any currently running debug sessions are listed. You should have only one. You want the Session ID, which is the first number in the response. In the following example, the session ID is 99 (This number is not related to the student number.):

```
xi52[student99_domain]# co
Global configuration mode
xi52[student99_domain](config)# show debug-actions

 Session ID    Transaction ID       Service Name       File Location
      Remote Address     In Use                Remote User         User Locatio
n                Elapsed Time
 ----------    --------------       ------------       -------------
      --------------     ------                -----------         ------------
-                ------------
 99            42609                BaggageServiceProxy local:///BaggageJson2Soa
p.js   172.16.80.107      No
                  00:19:32
```

Note

If you have any debug sessions that you want to cancel, you can use either the WebGUI or the CLI. For the WebGUI, use the list of active debug sessions under **Debug Action Status** to cancel any of the sessions. In the CLI, enter: no debug-action *<session ID>*

__ 16. Enter: debug-action *<session_id>*

__ 17. The terminal window refreshes with the GatewayScript code. The debugger should be halted at the debugger; statement. The indicator "**=>**" shows the "current line".

```
  7:            session.output.write("oops error " + JSON.stringify(e
ng()));
  8:    } else {
  9:
=>10:debugger;
  11:           var refNo = json.refNumber;
  12:           var lastName = json.lastName;
  13:
  14:           console.info("Received request from %s and referenec
```

__ 18. Type n and the => indicator moves to the next line. This line is assigning the retrieved reference number to a variable refNo.

__ 19. Type `n` again, and then type: `print refNo`
The value of `refNo` is displayed:

```
  10:debugger;
  11:           var refNo = json.refNumber;
=>12:           var lastName = json.lastName;
  13:
  14:           console.info("Received request from %s a
%i", lastName, refNo);
  15:
  16:           // Debug level is a stylesheet parameter
n JS action
  17:           // Check debug level , if debug level is
uest headers and service variables
  18:           // Default value is 0
  19:           if (!session.parameters.debugLevel)
  20:                   session.parameters.debugLevel =
  21:           if (session.parameters.debugLevel >= 5)
  22:           {
(debug) print refNo
11111
(debug)
```

__ 20. Type `c` to continue debugging.

__ 21. You can find a list of debugger commands by searching in the DataPower IBM Knowledge Center for “GatewayScript debugger commands”.

__ 22. When you want to exit the debugger, enter: `quit`

Attention

Your request is blocked until you exit the debugging session. The SoapUI request typically times out due to the long response time from the service.

__ 23. Enter `exit` at the command prompts until you get the “Goodbye” message.

__ 24. Close the terminal window.

__ 25. In the policy editor, open the GatewayScript action.

__ 26. Disable the debugger.

__ 27. Click **Done**.

__ 28. Click **Apply Policy** in the policy editor.

__ 29. In the queryJohnson request tab of SoapUI, your request should time out due to the debugging session. The session timeout that is configured in SoapUI is the cause.

1.9. Add a REST interface to the baggage status request

The initial request for baggage status was through a JSON data request, but it was not in a REST interface structure. The client is now going to use a proper REST request to get the status. The request comes in as an HTTP GET request, with the URI that contains the “search” criteria. No data is passed in the HTTP body. In this section, you add support to BaggageServiceProxy to respond to the REST-formatted request.

__ 1. Open the policy editor in the BaggageServiceProxy MPGW.

__ 2. Click **New Rule**.

__ 3. Enter a rule name of: `BaggageServicePolicy_BagsByPassenger_Req`

__ 4. Set the rule direction to **Client to Server**.

__ 5. Double-click the **Match** action to configure it.

__ 6. This new Match action checks for two conditions: an HTTP method of GET, and the correct URI for this request. Click new (**+**) to create a Matching Rule.

__ 7. Name the rule: `BagsByPassenger_Req`

__ 8. Click **Add** beneath the Rules section.

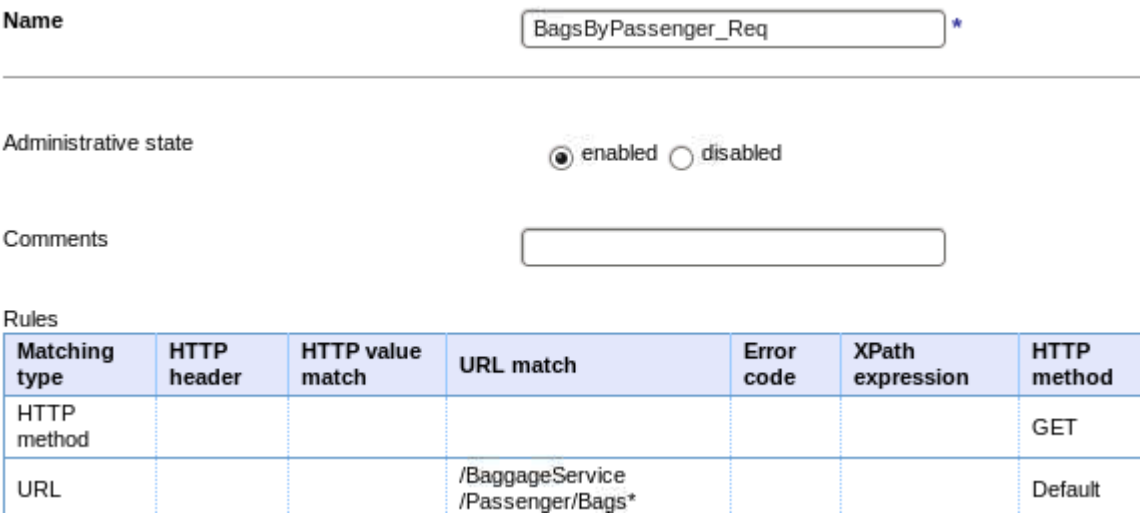
__ 9. Specify a Matching type of **HTTP method**, and an HTTP method of **GET**.

__ 10. Click **Apply**.

__ 11. Click **Add**.

__ 12. Specify a Matching type of **URL**, and a URL match of: `/BaggageService/Passenger/Bags*`

__ 13. Click **Apply**.



__ 14. Verify that **Combine with Boolean OR** is set to **off**. This setting indicates that both matches must occur for the match to be satisfied.

__ 15. Click **Apply** and **Done** to save the matching rule and Match action.

__ 16. You need to convert the REST GET request to a SOAP request. Drag a **GatewayScript** action to the rule configuration path.

__ 17. Upload `<lab_files>/REST/BagsJson2SOAP.js`.

__ 18. Click **View** to examine the code.

__ 19. Notice that the code is similar to the previous GatewayScript code. One important difference is that the reference number and last name must be parsed from the URI, rather than retrieved from the POSTed data.

__ 20. Close the View window.

__ 21. Click the **Advanced** tab.

__ 22. Add the stylesheet parameter `debugLevel` with a value of **1** to be able to get more log entries as needed.

__ 23. Click **Done**.

__ 24. To type the output of the GatewayScript to XML, you use the identity transform as before. Drag a **Transform** action after the **GatewayScript**.

__ 25. For the Transform File, change the DataPower directory to: `store:///`

__ 26. Select the **identity.xsl** file from that directory.

__ 27. Click **Done**.

__ 28. The request comes into this rule as an HTTP GET method. Web services expect a POST method. Rewrite the HTTP method next. Drag the **Advanced** action after the **Transform**.

__ 29. Configure the **Advanced** action to be a **Method Rewrite** action.

__ 30. In that action, specify a Method of **POST.**

__ 31. Click **Done**.

__ 32. The URI for a REST request is usually different from one used for a SOAP request. You need to change the original URI to one that is acceptable to the back-end web service. Drag the **Advanced** action after the **Method Rewrite**.

__ 33. Configure the **Advanced** action as a **Set Variable** action.

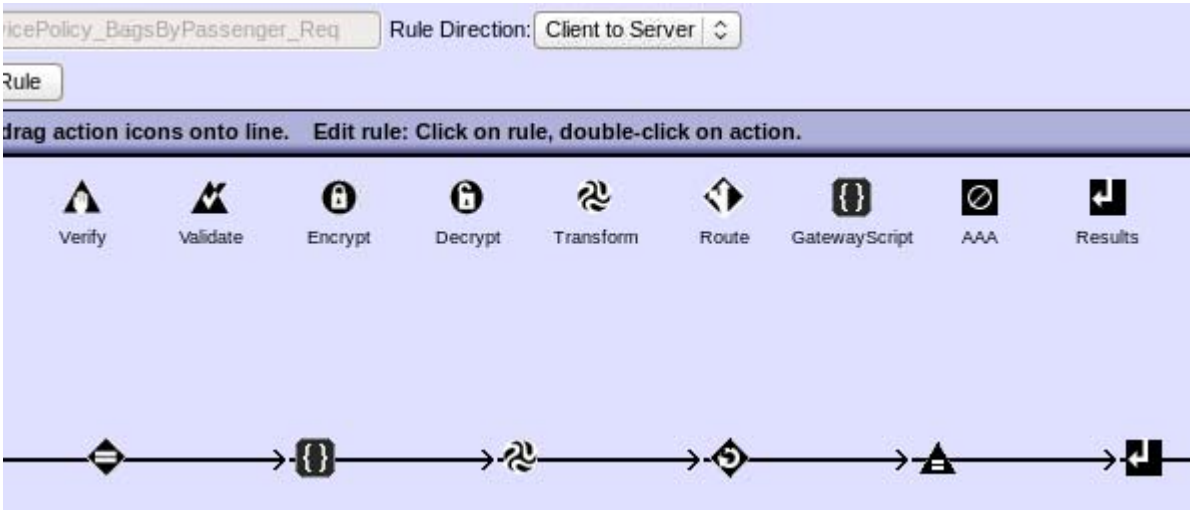
__ 34. For the Variable name, use **Var Builder** to select the service variable **var://service/URI**.

__ 35. Specify **/BaggageService** as the assignment.

__ 36. Click **Done**.

__ 37. Click **Apply Policy** to save the rule. A Results action is automatically added to the rule to move the intermediate results to the OUTPUT context.

__ 38. The completed request rule should look like the following figure:



__ 39. If necessary, click **Apply** at the MPGW level.

1.10. Test the REST interface for Baggage Status

__ 1. Open SoapUI.

__ 2. Open the **RESTqueryJohnson** request window.

__ 3. In the RESTqueryJohnson request window, verify that the HTTP method is set to **GET**.

__ 4. In the URL address field, make sure that the endpoint address is:
`http://${dp_public_ip}:${mpgw_booking_port}/BaggageService/Passenger/Bags?refNumber=11111&lastName=Johnson`

__ 5. No data is in the HTTP body.

__ 6. Click the green **Submit** arrow to send the request.

__ 7. In the response tab, be sure to click the **JSON** view tab. You get a “not JSON content” message.

__ 8. Click the XML view tab. The standard generic error SOAP fault from DataPower is displayed. What is failing?

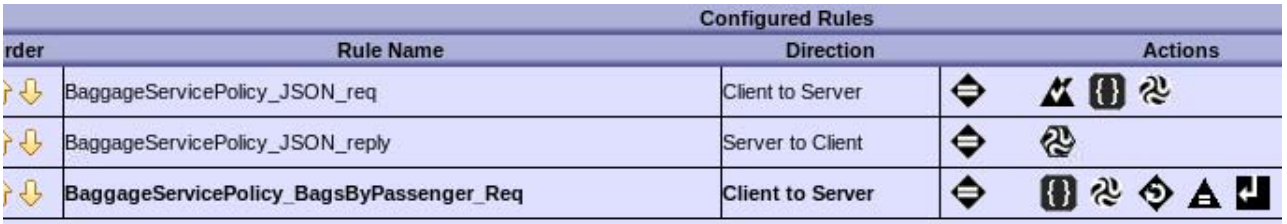
__ 9. Open the Transaction List probe window.

__ 10. Click **Refresh**.

__ 11. The request entry is in red. Click the magnifying glass.

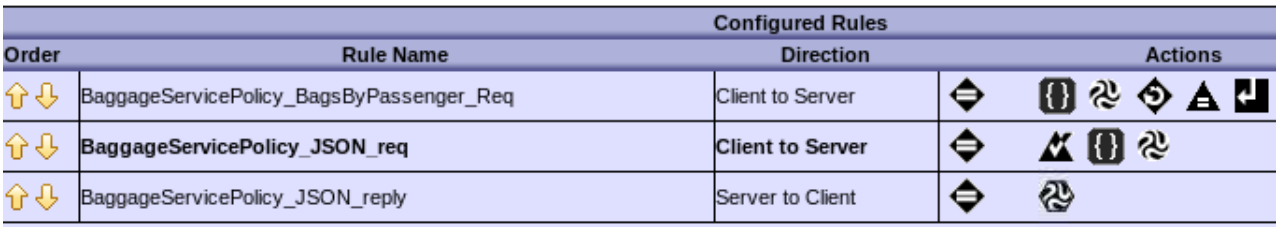
__ 12. What is the first and only action? It is the Validate action. Was there a Validate action in this new request rule? No! So what is happening?

__ 13. In the policy editor, look at the Configured Rules section of the policy editor. Note the processing order of the rules.



__ 14. Recall that the Match action for the original BaggageServicePolicy_JSON_req matches on all URLs, and it is the first request rule to be processed. Hence, the new request is processed by the old rule. How can you fix it?

__ 15. Because the new request rule has a more specific Match action, you move it to the front of the processing order. Use the yellow Order arrows to do that.



__ 16. Click **Apply Policy**.

__ 17. In SoapUI, send the **RESTqueryJohnson** again.

__ 18. Click the JSON view tab in the response tab.

__ 19. You should see the JSON array of baggage results for the passenger.

__ 20. Feel free to use the probe, system log, or CLI debugger to further investigate the processing activity. You can verify that no JSON data is passed in the HTTP body.

1.11. Add new REST interface to find a specific bag

The client wants to support another RESTful interface. This new interface allows a client to submit the bag ID number so that the bag can be found, regardless of which passenger it belongs to.

__ 1. Switch to the policy editor to create a rule.

__ 2. Click **New Rule**.

__ 3. Enter a rule name of: `BaggageServicePolicy_FindBag_Req`

__ 4. Set the rule direction to **Client to Server**.

__ 5. Double-click the **Match** action to configure it.

__ 6. This new Match action checks for two conditions: an HTTP method of GET, and the correct URI for this request. Click new (**+**) to create a Matching Rule.

__ 7. Name the rule: `FindBag_Req`

__ 8. Click **Add** beneath the Rules section.

__ 9. Specify a Matching type of **HTTP method**, and an HTTP method of **GET**.

__ 10. Click **Apply**.

__ 11. Click **Add**.

__ 12. Specify a Matching type of **URL**, and a URL match of **/BaggageService/Bag***

__ 13. Click **Apply**.

__ 14. Verify that **Combine with Boolean OR** is set to **off**. This setting indicates that both matches must occur for the match to be satisfied.

__ 15. Click **Apply** and **Done** to save the matching rule and Match action.

__ 16. You need to convert the REST GET request to a SOAP request. Drag a **GatewayScript** action to the rule configuration path.

__ 17. Upload `<lab_files>/REST/FindBagJson2SOAP.js`.

__ 18. Click **View** to examine the code.

__ 19. Notice that this code is similar to the previous GatewayScript code. In this case, the bag ID is parsed from the URI.

__ 20. Close the View window.

__ 21. Click the **Advanced** tab.

__ 22. Add the stylesheet parameter `debugLevel` with a value of **1** to be able to get more log entries as needed.

__ 23. Click **Done**.

__ 24. To type the output of the GatewayScript to XML, you use the identity transform as before. Drag a **Transform** action after the **GatewayScript**.

__ 25. For the Transform File, change the DataPower directory to **store:///**.

__ 26. Select the **identity.xsl** file from that directory.

__ 27. Click **Done**.

__ 28. The request comes into this rule as an HTTP GET method. Web services expect a POST method. Rewrite the HTTP method next. Drag the **Advanced** action after the **Transform**.

__ 29. Configure the Advanced action to be a **Method Rewrite** action.

__ 30. In that action, specify a Method of **POST.**

__ 31. Click **Done**.

__ 32. The URI for a REST request is usually different from one used for a SOAP request. You need to change the original URI to one that is acceptable to the back-end web service. Drag the **Advanced** action after the **Method Rewrite**.

__ 33. Configure the Advanced action as a **Set Variable** action.

__ 34. For the Variable name, use **Var Builder** to select the service variable **var://service/URI**.

__ 35. Specify **/BaggageService** as the assignment.

__ 36. Click **Done**.

__ 37. Click **Apply Policy** to save the rule. A Results action is automatically added to the rule to move the intermediate results to the OUTPUT context.

__ 38. The SOAP response for finding a bag is different from the response for getting baggage status. A response rule needs to be created to receive the different format and transform it into a different JSON structure to the client.

__ 39. Click **New Rule**.

__ 40. Enter a rule name of: `BaggageServicePolicy_FindBag_Resp`

__ 41. Set the rule direction to **Server to Client**.

__ 42. Double-click the **Match** action to configure it.

__ 43. In this rule, the SOAP response is analyzed to see whether it is a `<BagInfoResponse>`. Click new (**+**) to create a matching rule.

__ 44. Name it: `FindBag_Resp`

__ 45. Click **Add** to create a rule.

__ 46. Set the Matching type to **XPath**.

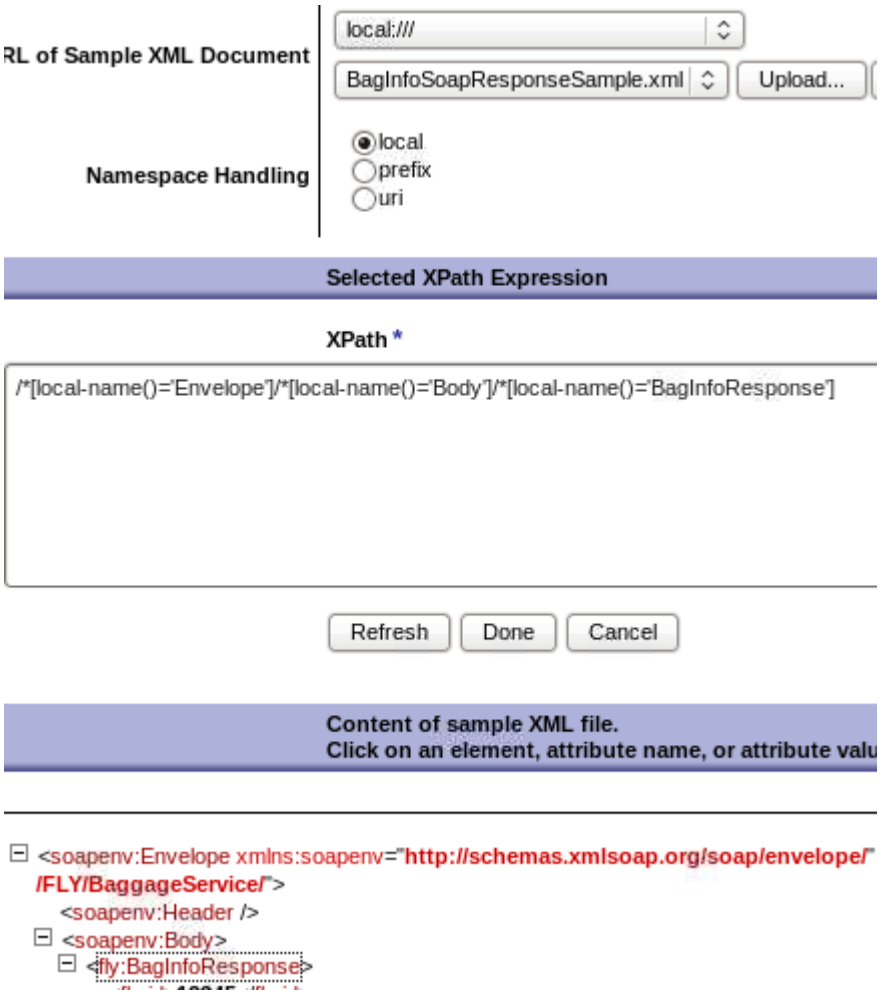
__ 47. The page refreshes. Click **XPath Tool.**

__ 48. You use a sample response file to help build the XPath expression. Upload `<lab_files>/REST/BagInfoSoapResponseSample.xml`.

__ 49. For Namespace Handling, select **local**.

__ 50. In the lower part of the window, the sample XML response is displayed. Select the **<BagInfoResponse>** element.

__ 51. The resulting XPath is placed in the XPath* section of the page.



__ 52. Click **Done**.

__ 53. The resulting XPath expression is placed in the Edit Rules window. Click **Apply**.

__ 54. Click **Apply** to complete the matching rules.

__ 55. Click **Done** to save the Match action.

__ 56. Add a Transform action that uses XQuery and JSONiq to convert the SOAP response to a JSON data structure.

__ a. Drag a **Transform** action to the rule configuration path.

__ b. Double-click it to configure it.

__ c. Set the Document Processing Instructions to **Transform with a processing control file, if specified**.

__ d. Set the Input Language to **XML**.

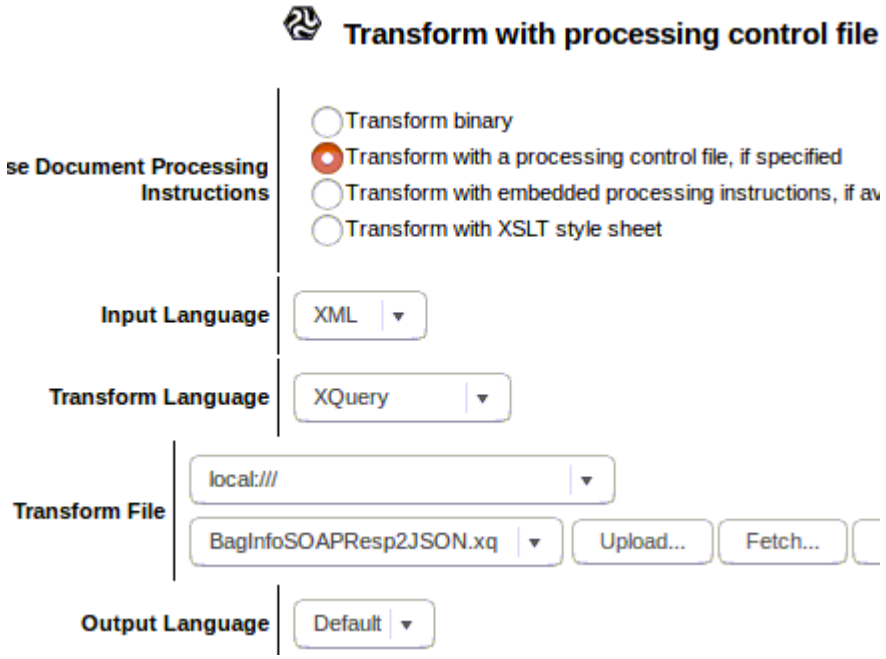
__ e. Leave the Transform Language as **XQuery**.

__ f. Upload `<lab_files>/REST/BagInfoSOAPResp2JSON.xq`.

__ g. Click **View** to review the file.

__ h. The XQuery pull the elements from the `<BagInfoResponse>` and use them to build the JSON structure. They also set the HTTP Content-Type header to “application/json”.

__ i. Close the View window.



__ j. Click **Done** to complete the configuration.

__ 57. Click **Apply Policy**.

Recall that the order of the rules is critical. You want the most general rules – the MatchAnyURI ones – to be at the end of the processing order. Move the new request and response rules ahead of the general rules. The result should resemble the following figure:

Configured Rules		
Rule Name	**Direction**	**Actions**
BaggageServicePolicy_BagsByPassenger_Req	Client to Server	
BaggageServicePolicy_FindBag_req	Client to Server	
BaggageServicePolicy_FindBag_Resp	**Server to Client**	
BaggageServicePolicy_JSON_req	Client to Server	
BaggageServicePolicy_JSON_reply	Server to Client	

When processing the rules, the firmware easily distinguishes between request and response rules.

__ 58. Click **Apply Policy** to save the new order.

__ 59. Click **View Status** in the policy editor to check the state of the service policy and its contained objects. You might need to deal with the same firmware bug as you did earlier.

__ 60. If the Transform action (“BaggageServicePolicy_FindBag_Resp_xformng_0”) is **down**, follow the steps in the following Troubleshooting block. Otherwise, close the policy editor, click **Apply** to save the MPGW, and continue after the block.

Troubleshooting

If the **Transform with processing control file** action in the response rule is down, a known bug was not yet fixed. The following steps are a workaround until the fix is delivered in a firmware fix pack.

__ a. Close the policy editor and **Apply** the MPGW.

__ b. Start to enter `processing` into the search field in the navigation bar.

__ c. Select **Processing Action** in the list.

__ d. The catalog of processing actions is displayed. Select the Transform action that is in the **down** state.

__ e. The Objects version of the Action configuration page is displayed. Notice that the **Transform File** field is blank. When you created this Transform action in the policy editor, you did supply this value. The bug is that the value is not saved. You fix that in the next steps.

__ f. Enter the value for the Transform File: `local:///BagInfoSOAPResp2JSON.xq`

__ g. Click **Apply**.

__ h. Click **Control Panel > Multi-Protocol Gateway > BaggageServiceProxy**.

__ i. Click **View Status**

__ j. Now the processing policy and Transform action should show as **up**.

__ k. Close the Object status window.

__ 61. Click **Save Configuration**.

1.12. Test the retrieval by bag ID

Verify that the REST request to find a bag by its ID returns the current details about the bag.

__ 1. In SoapUI, open the **RESTqueryBagId** request window.

__ 2. In the RESTqueryBagId request window, verify that the HTTP method is set to **GET**.

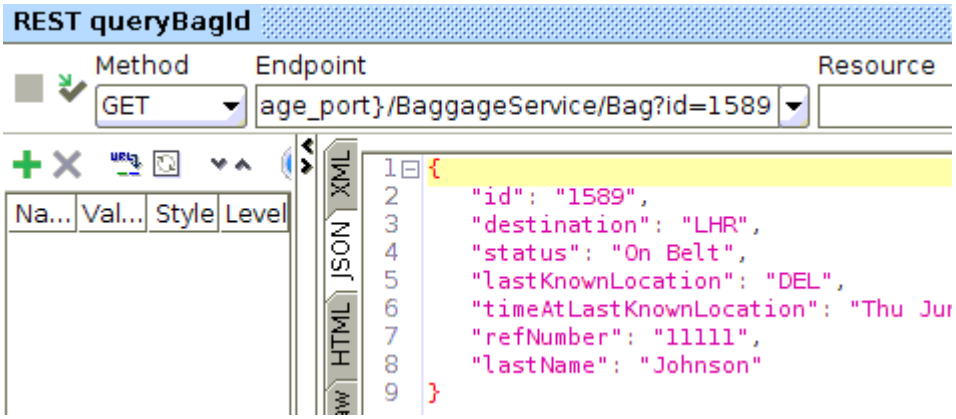
__ 3. In the URL address field, make sure that the endpoint address is:
`http://${dp_public_ip}:${mpgw_booking_port}/BaggageService/Bag?id=1589`

__ 4. No data is in the HTTP body.

__ 5. Click the green **Submit** arrow to send the request.

__ 6. In the response tab, be sure to click the **JSON** view tab.

__ 7. You should see the current information on the specific bag:



__ 8. Use the debugger, log, or probe to further investigate the service behavior.

Information

A “production-level” service checks more of the request URI for validity. All responses would be JSON data or an HTTP status code, per a REST design.

Both the back-end application and the BaggageServiceProxy would have a more robust error-handling design.

__ 9. Disable the probe and any running debugging sessions. Review the earlier **Note** paragraph for approaches to cancel debug sessions.

__ 10. Save your configuration.

End of exercise

Exercise review and wrap-up

In this exercise, you configured a service that received JSON data in the HTTP body, transformed it into a SOAP request, and converted the SOAP response to a JSON response. Next, you changed the service to support REST GET requests. You used XSL stylesheets, GatewayScript, and XQuery/JSONiq to transform the message. The system log, probe, and CLI debugger were used to debug and examine the configuration behavior.

Exercise 2. Creating and verifying a JWS

Estimated time

01:00

Overview

A JWS is used to sign a payload that is either a JSON object or the URI request parameters in a REST request. This exercise covers the configuration of the JSON Web Sign action to generate a compact serialized JWS and a JSON serialized JWS. As part of this exercise, you also configure the JSON Web Verify action to verify the signature within the JWS.

Objectives

After completing this exercise, you should be able to:

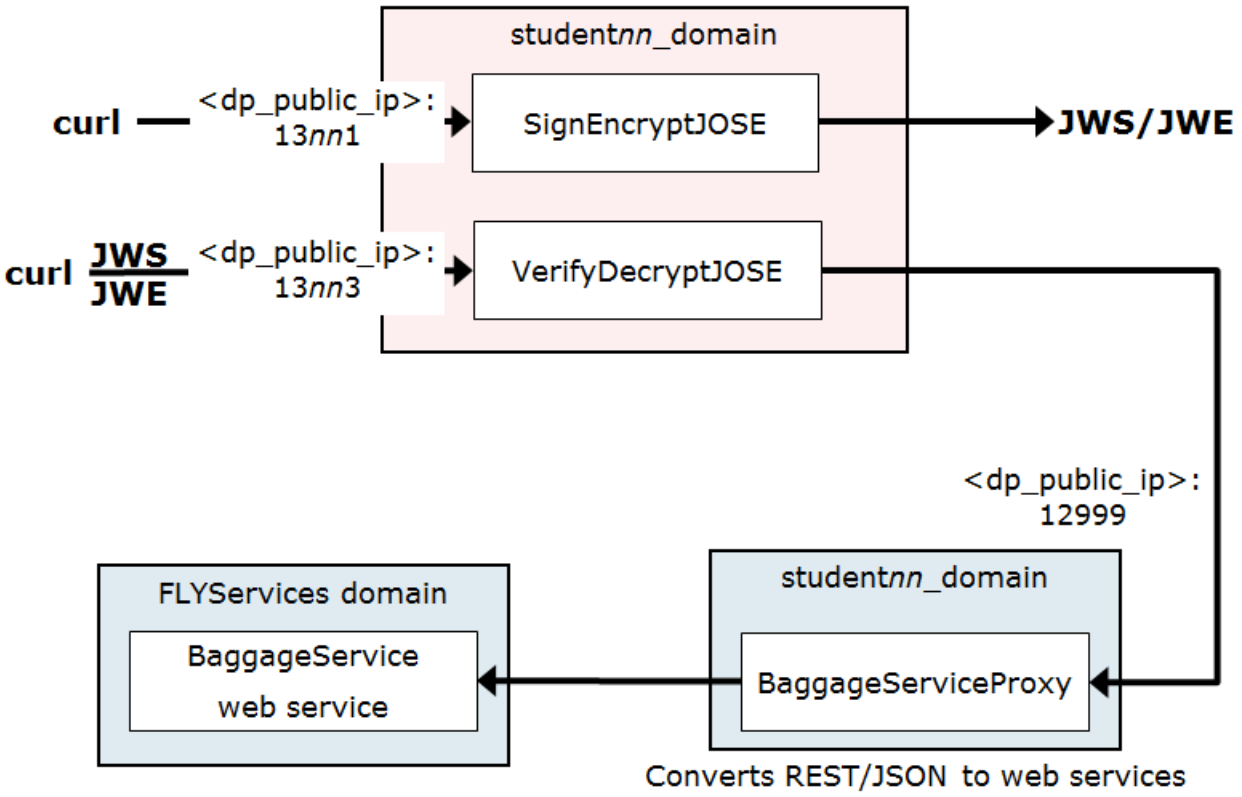
- Configure a JSON Web Sign action to generate a compact serialized and a JSON serialized JWS
- Configure a JSON Web Verify action to verify a compact serialized and a JSON serialized JWS

Introduction

In this exercise, the student configures the processing actions in the policy editor to create and verify a JWS. Both compact and JSON serializations are used.

The student creates several multi-protocol gateways (MPGWs), and uses the BaggageServiceProxy MPGW to access a back-end web service to retrieve baggage information for a specific passenger. The services, domains, and functions are:

- `SignEncryptJOSE MPGW (student`*`nn`*`_domain)`: This MPGW is used to create a JWS or a JWE. “cURL” is used to send data to this service, and the output is a JWS or JWE.
- `VerifyDecryptJOSE MPGW (student`*`nn`*`_domain)`: This MPGW takes as input a JWS or JWE, and processes it to properly form the request to the back-end service. The back end of the service calls the BaggageServiceProxy MPGW. The response is returned to the cURL command in the terminal window.
- `BaggagServiceProxy (student`*`nn`*`_domain)`: This service is the MPGW that you configured in the previous exercise. It receives JSON or REST GET requests, and transforms them into a SOAP request. It converts the SOAP response into a JSON response.
- BaggageService web service (FLYServices): This supplied service is the actual back-end web service that manages the baggage services.



The exercise starts with some initial setup sections: testing the back-end service and importing the key materials. Most of the exercise concerns the JWS.

The flow of the JWS portions of this exercise is:

1) Configure the SignEncryptJOSE MPGW to create a compact serialized JWS for an HTTP GET request.
2) Use cURL to test the JWS generation.
3) Configure the VerifyDecryptJOSE MPGW to verify the JWS input, and pass the request on to the BaggageServiceProxy.
4) Use cURL to send the JWS to the VerifyDecryptJOSE MPGW. The response should be the baggage status.
5) Configure the SignEncryptJOSE MPGW to create a JSON serialized, single-signature JWS for an HTTP POST request.
6) Use cURL to test the JWS generation.
7) Configure the VerifyDecryptJOSE MPGW to verify the JWS input, and pass the request on to the BaggageServiceProxy.
8) Use cURL to send the JWS to the VerifyDecryptJOSE MPGW.
9) Configure the SignEncryptJOSE MPGW to create a JSON serialized, multi-signature JWS for an HTTP POST request.
10) Use cURL to test the JWS generation.

11) Configure the VerifyDecryptJOSE MPGW to verify the JWS input, and pass the request on to the BaggageServiceProxy.

12) Use cURL to send the JWS to the VerifyDecryptJOSE MPGW.

Requirements

To complete this exercise, you need:

- Access to the **DataPower** gateway
- **cURL**, for sending requests to the DataPower gateway
- The **BaggageStatusMockService** web service that runs on the DataPower gateway in the `FLYServices` domain
- The **BaggageServiceProxy** service that runs on the DataPower gateway in the `BaggageServiceProxy_domain` domain
- Access to the `<lab_files>` directory

Section 1. Preface

Remember to use the domain and port address that you were assigned in the exercise setup. **Do *not*** use the default domain.

The references in exercise instructions refer to the following values:

- <*lab_files*>: Location of the student lab files. Default location is: `/usr/labfiles/dp/`
- <*dp_internal_ip*>: IP address of the DataPower gateway development and administrative interface.
- <*dp_public_ip*>: IP address of the public services on the gateway that customers and clients use.
- <*nn*>: Assigned student number. If the course has no instructor, use "`01`".
- <*studentnn*>: Assigned DataPower user name and user account. If the course has no instructor, use "`student01`".
- <*studentnn_password*>: DataPower account password. In most cases, the initial value is the same as the user name. You are prompted to create a password on first use. Write it down.
- <*studentnn_domain*>: Application domain that the user account is assigned to. If the course has no instructor, use "`student01_domain`".
- <*mpgw_baggage_port*>: `12`*nn*`9`, where "*nn*" is the two-digit student number. This port number is the listener port of the BaggageServiceProxy that mediates between the REST or JSON client and the Baggage Services back-end application.
- <*SignEncryptJOSE_port*>: `13`*nn*`1`, where "*nn*" is the two-digit student number. This number is the listener port for the SignEncryptJOSE MPGW.
- <*VerifyDecryptJOSE_port*>: `13`*nn*`3`, where "*nn*" is the two-digit student number. This number is the listener port for the VerifyDecryptJOSE MPGW.

Section 2. Test the back-end services

Some of your MPGWs call is a BaggageServiceProxy multi-protocol gateway (MPGW) for the FLY Airline baggage services. In this section, you test the availability.

2.1. Log in to Linux and prepare a terminal window

The steps are as follows:

__ 1. Log in to the **localuser** account in the Linux system. This account is where you complete the steps for this exercise. The default is `localuser` as the user name and `passw0rd` as the password.

__ 2. Open a terminal window. Look for an icon in the launcher bar on the left side of the desktop.

__ 3. Change the directory to the JWS directory:

```
cd <lab_files>/JWS
```

2.2. Use cURL to test the REST-based GET request

This request passes a passenger reference number and the passenger last name as request parameters in the URL. It should return a list of bags that are associated with that passenger, and each bag's status.

__ 1. In the terminal window, enter a cURL command to retrieve the bags that belong to a passenger:

```
curl -G
"http://<dp_public_ip>:12999/BaggageService/Passenger/Bags?refNumber=11111&l
astName=Johnson"
```

Reminder

cURL commands are case-sensitive. You must use quotation marks around the URL because of the request parameters.

The response should be a JSON object that contains the passenger name and an array of the associated bags:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

2.3. Use cURL to test the POST request that passes a JSON object

This request is a non-RESTful request to get the same information.

__ 1. Enter the `curl` command into the terminal window:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:12999/BaggageService
```

You should get the same response, a JSON object that contains the passenger name and the associated bags.

__ 2. Keep the terminal window open so that you can reuse the commands in later steps.

Section 3. Import the key material objects and files

A set of key materials is already generated for you. In this section, you upload the PEM files and import the crypto objects.

The **Emi** and **Erin** key materials are used for encrypting. The **Sam**, **Seth**, and **Simon** key materials are used for signing.

The naming pattern is:

- **Emi** is the key reference
- **Emi-privkey** is the crypto key
- **Emi-cert** is the crypto certificate
- **EmiIDCred** is the crypto identification credential that relates the private key and certificate

3.1. Log in to your domain in the WebGUI

__ 1. From the web browser, enter the URL to the WebGUI:

`https://<dp_internal_ip>:9090`

__ 2. Enter the login information:

- User Name: <*studentnn*>
- Password: <*studentnn_password*>
- Domain: <*studentnn_domain*>

3.2. Upload the PEM files

__ 1. In the navigation bar, enter `file m` in the **Search** field.

__ 2. Select **File Management**.

__ 3. On the File Management page, click the **Actions** link for the `cert:` directory.

__ 4. Click **Upload Files**.

__ 5. On the next page, click **Browse**.

__ 6. In the file browser, navigate to: <*lab_files*>/JWSJWEsetup

__ 7. Select **Emi-privkey.pem**.

__ 8. Click **Upload**.

__ 9. Repeat the same steps for the other PEM files:

- `Emi-sscert.pem`
- `Erin-privkey.pem`
- `Erin-sscert.pem`
- `Sam-privkey.pem`
- `Sam-sscert.pem`

- `Seth-privkey.pem`
- `Seth-sscert.pem`
- `Simon-privkey.pem`
- `Simon-sscert.pem`

3.3. Import the key material objects

__ 1. In the navigation bar, enter `import` in the **Search** field.

__ 2. Click **Import Configuration**.

__ 3. Browse to the same location as the PEM files: `<lab_files>/JWSJWEsetup`

__ 4. Select **WE752CryptoMaterials.zip**.

__ 5. Import all of the objects in the `.zip` file.

__ 6. If you want to review the crypto objects, you can do so. Because the PEM files are already in the `cert:` directory, the imported objects are “up”.

Information

The “E” key materials (Emi, Erin) are used for encryption activities. The “S” key materials (Sam, Seth, Simon) are used for signature activities.

__ 7. Click **Save Configuration**.

Section 4. Create a compact serialized JWS

In this section, you create the SignEncryptJOSE MPGW. This MPGW is also used in the following exercises. You configure the service policy to create a compact serialized JWS.

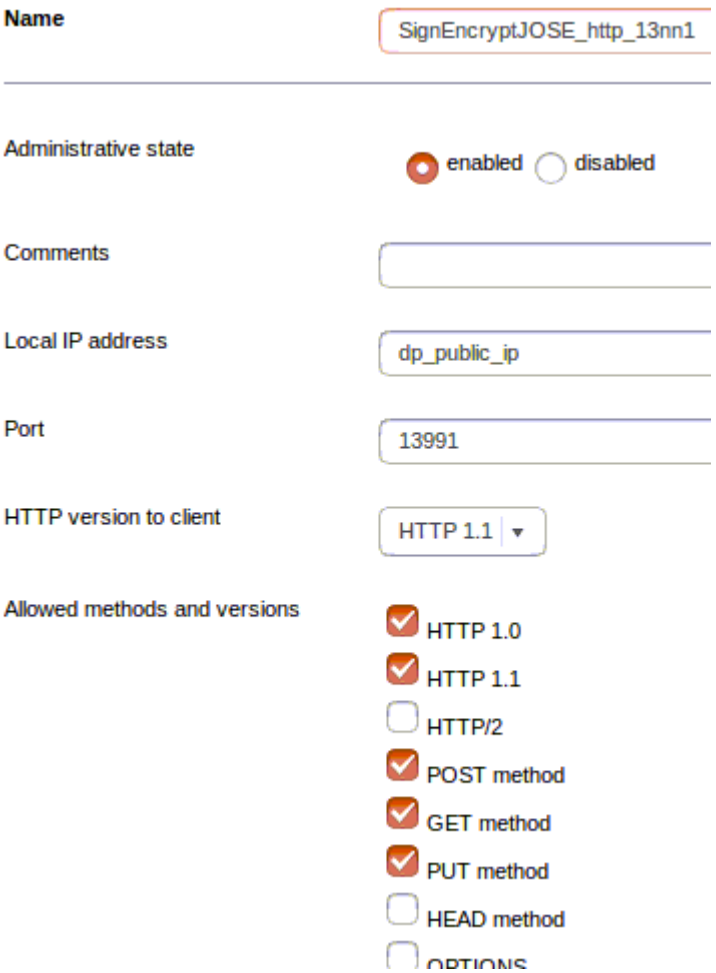
4.1. Create the front side handler (FSH)

__ 1. Enter `http f` in the **Search** field of the navigation bar. Select **HTTP Front Side Handler**.

__ 2. Click **Add** to create an HTTP FSH.

__ 3. Configure the handler as follows:

- Name: `SignEncryptJOSE_http_13nn1`
- Local IP address: *<dp_public_ip>*
- Port: *<SignEncryptJOSE_port>*
- Be sure to select the **GET method** check box, which is not selected by default.



__ 4. Click **Apply**. The FSH shows as “down” since it is not yet associated with a service.

4.2. Start the configuration of the SignEncryptJOSE MPGW

__ 1. Click **Control Panel**.

__ 2. Click **Multi-Protocol Gateway**.

__ 3. Click **Add** to start a new MPGW.

__ 4. Specify the following values:

- Multi-Protocol Gateway Name: `SignEncryptJOSE`
- Type: `dynamic-backends`
- Request Type: `Non-XML`
- Response Type: `Pass through`
- Front Side Protocol: `SignEncryptJOSE_http_13nn1`
- Multi-Protocol Gateway Policy: `default`

__ 5. Click **Apply**. This MPGW does not have any useful behavior yet because it uses the **default** service policy. You configure a new one in the next step.

4.3. Configure the service policy

The input to the service is the request parameter string that is used to build the JWS: `refNumber=11111&lastName=Johnson`. The rule in this service policy converts the input string to a compact serialized JWS.

__ 1. Click the **+** (New) button for the Multi-Protocol Gateway Policy.

__ 2. Enter a Policy Name of `SignEncryptJOSE` and click **Apply Policy**.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `SignURIcompact` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/SignURIcompact*`

__ 6. Add a **Sign** action to the rule.

__ 7. Configure the Sign action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Sign** action with new options.

__ 8. Leave the Serialization at **Compact**.

__ 9. Click the **+** button to create a JWS Signature object.

__ 10. Enter a name: `SamSignature`

__ 11. Select the **RS256** Algorithm. This choice is an RSA-based algorithm.

__ 12. Select the **Sam-privkey** Private Key that you imported earlier.

__ 13. Because you are building a compact serialized JWS, any header parameters that you want to include *must* be in a protected header. To help the header matching operation of the JSON Web Verify, add a protected header parameter with a Name of `kid` with a Value of `Sam`.

__ 14. Click **Apply** for the JWS Signature object.

__ 15. On the JSON Web Sign page, click **Done**.

__ 16. On the configuration path, hover over the JSON Web Sign action. The input context is set to INPUT.

__ 17. Drag the **Advanced** action to the rule. Configure the action to be a **Set Variable** action.

__ 18. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 19. Click **Apply Policy**.

__ 20. The policy editor adds a Results action to the rule. It moves the output of the JSON Web Sign action (the JWS) to the OUTPUT context.

__ 21. Close the policy editor.

__ 22. Click **Apply** for the MPGW.

Section 5. Test the compact serialized JWS generation

In this section, you send the request parameter string to the MPGW to create a JWS string. The JWS string is used later in the exercise as part of the URL. This section uses a cURL command to send the request parameter string to the service. The JWS string output is piped into a text file for use in a later section.

__ 1. Switch back to the terminal window, and remain in the JWS subdirectory.

__ 2. Enter a cURL command to generate the JWS:

```
curl --data-binary @URIstring.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/SignURIcompact >
SignedURIcompactJWS.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file has the three parts of a compact serialized JWS: the base64url-encoded JWS protected header, a period, the base64url-encoded payload, another period, and the base64url-encoded signature.

__ 5. Use any internet-available base64url decoder site to decode the header and payload. The protected header decodes as “`{"alg":"RS256","kid":"Sam"}`”. The payload decodes as “`refNumber=11111&lastName=Johnson`”.

__ 6. Leave the file open, as you are going to use this string for the next test. **Do *not*** add any extra line breaks.

__ 7. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 6. Verify a compact serialized JWS

In this section, you create the VerifyDecryptJOSE MPGW. This MPGW is also used in the following exercises. You configure the service policy to verify a compact serialized JWS.

6.1. Create the front side handler (FSH)

__ 1. Enter `http f` in the **Search** field of the navigation bar. Select **HTTP Front Side Handler**.

__ 2. Click **Add** to create an HTTP FSH.

__ 3. Configure the handler as follows:

- Name: `VerifyDecryptJOSE_http_13nn3`
- Local IP address: *<dp_public_ip>*
- Port: *<VerifyDecryptJOSE_port>*
- Be sure to select the **GET method** check box, which is not selected by default

__ 4. Click **Apply**. The FSH shows as “down” since it is not yet associated with a service.

6.2. Start the configuration of the SignEncryptJOSE MPGW

__ 1. Click **Control Panel**.

__ 2. Click **Multi-Protocol Gateway**.

__ 3. Click **Add** to start a new MPGW.

__ 4. Specify the following values:

- Multi-Protocol Gateway Name: `VerifyDecryptJOSE`
- Type: `static-backend`
- Default Backend URL:
  `http://`*<dp_public_ip>*`:`*<mpgw_baggage_port>*`/BaggageService`
- Request Type: `Non-XML`
- Response Type: `Non-XML`
- Front Side Protocol: `VerifyDecryptJOSE_http_13nn3`
- Multi-Protocol Gateway Policy: `default`

__ 5. Click **Apply**. This MPGW does not have any useful behavior yet because it uses the **default** service policy. You configure a new one in the next step.

6.3. Configure the service policy

The input to the service is the compact serialized JWS that you generated previously. This service policy verifies the JWS, places the decoded payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Click the **+** (New) button for the Multi-Protocol Gateway Policy.

__ 2. Enter a Policy Name of `VerifyDecryptJOSE` and click **Apply Policy**.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `VerifyURIcompact` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/VerifyURIcompact*`

Note

Do not forget the asterisk (“*”) at the end of the URL match field. Because the URL contains more text passed the URI path, you must specify that the following text (“?xyz987.yyy...”) is acceptable for this matching rule.

__ 6. Add a **GatewayScript** action to the rule.

__ 7. Upload `<lab_files>/JWS/SignedURI2JWS.js` to the action. This GatewayScript extracts the JWS from the input URI and places it into the output context for the JSON Web Verify action that follows.

__ 8. Click **Done**.

__ 9. Add a **Verify** action to the rule.

__ 10. Configure the Verify action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Verify** action with new options.

__ 11. Set the Identifier Type to **Single Identifier - Certificate**.

__ 12. Notice that the only option is to specify the verifying certificate. No other option identifies any of the possible protected header parameters. Since you want to specify some header information, you change the type in the next step.

Important

The recommendation is to use the “Signature Identifiers” choice so that more protected header parameters can be specified, even when you have just one signature to verify.

__ 13. Set the Identifier Type to **Signature Identifiers**.

__ 14. Click the **+** button to create a Signature Identifier object.

__ 15. Enter a Name: `SamSigID`

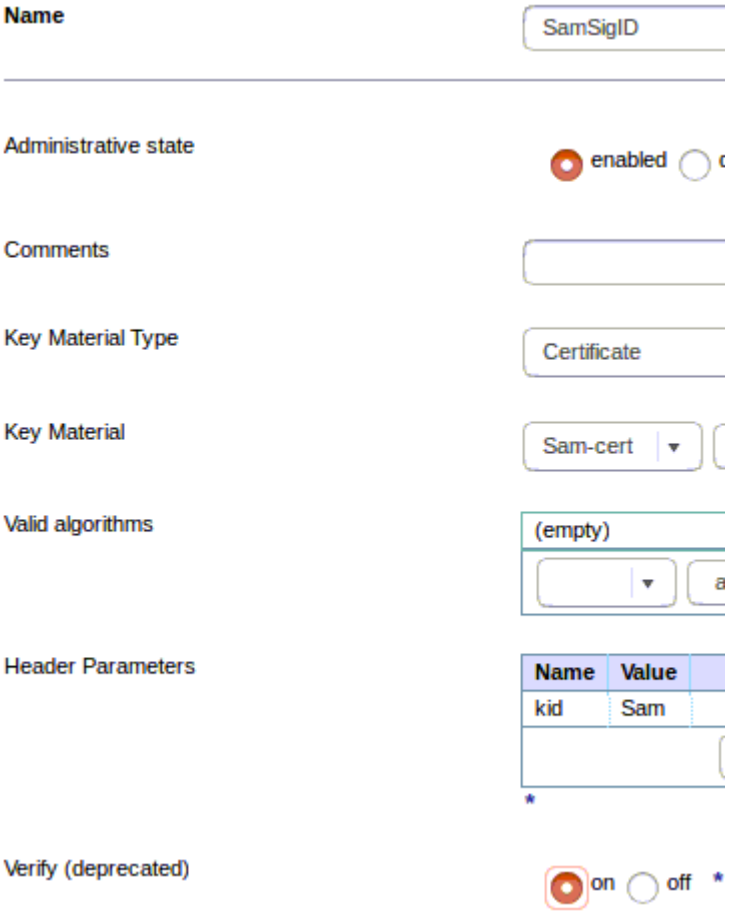
__ 16. Select the Key Material Type of **Certificate**.

__ 17. Select the **Sam-cert** Key Material that you imported earlier.

__ 18. Leave the **Valid algorithms** list empty. By default, this setting allows any valid algorithm value.

__ 19. Because you want to test the match of the “kid” parameter, you must specify the header details. Add a Header Parameter with a Name of `kid` with a Value of `Sam`.

__ 20. Leave the **Verify** option as **on** because you want the signature verification operation to occur. This option is deprecated.



__ 21. Click **Apply** for the Signature Identifier object.

__ 22. On the JSON Web Verify page, set **Strip Signature** to **on** because you want the output context to contain the decoded request parameters.

__ 23. Click **Done**.

__ 24. Drag another **GatewayScript** action to the rule.

__ 25. Upload `<lab_files>/JWS/UnsignedURI2Variable.js` into the action. This GatewayScript creates the URI that the BaggageServiceProxy expects.

__ 26. Click **Done**.

__ 27. Click **Apply Policy**.

__ 28. The response from the BaggageServiceProxy must be handled; click **New Rule**.

__ 29. Specify a Rule Name of `Generic Response` and set the direction to **Server to Client**.

__ 30. Configure the Match action to match on all URIs.

__ 31. Click **Apply Policy**. The policy editor adds a Results action for you.

__ 32. Close the policy editor.

__ 33. Click **Apply** for the MPGW.

Section 7. Test the compact serialized JWS verification and call the back-end baggage service

In this section, you use a cURL command to simulate a client that sends a REST request with an attached JWS to the service. The JWS is verified, and the decoded payload is sent to the back-end baggage service.

__ 1. Use gedit to construct a cURL command that contains the JWS as a URI string. Type in the cURL command that follows in a few steps. Open the `SignedURIcompactJWS.txt` file that contains the JWS. Copy the contents (no added line breaks) into the cURL command string. Copy the complete cURL command string.

__ 2. Switch back to terminal window, and remain in the JWS subdirectory.

__ 3. Paste the cURL command string into the terminal window and press **Enter**:

```
curl -G
"http://<dp_public_ip>:<VerifyDecryptJOSE_port>/VerifyURIcompact?<contents
of SignedURIcompactJWS.txt>"
```

__ 4. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 5. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 6. Save the configuration.

Section 8. Generate a JSON serialized JWS with a single signature

A JSON serialized JWS is not designed to be URL-safe, nor a size that is restricted. It is expected to be passed in an HTTP body, rather than in the URI. In this section, you modify the service policy of the SignEncryptJOSE MPGW to create a JSON serialized JWS.

8.1. Extend the service policy

The input to the service is a JSON object that is passed in the HTTP body:

```
{
  "refNumber" : 11111,
  "lastName" : "Johnson"
}
```

This object is used to build the JSON serialized JWS.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `SignBodyJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/SignBodyJSON*`

__ 6. Add a **Sign** action to the rule.

__ 7. Configure the Sign action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Sign** action with new options.

__ 8. Set the Serialization to **Flattened JSON**. You can use this value because the JWS contains only one signature.

__ 9. Select the previously defined **SamSignature** as the Signature object. The SamSignature object specifies the RS256 algorithm, the Sam-privkey private key, and the kid=Sam header parameter.

__ 10. Click **Done**.

__ 11. On the configuration path, hover over the JSON Web Sign action. The input context is set to INPUT.

__ 12. Drag the **Advanced** action to the rule. Configure the action to be a **Set Variable** action.

__ 13. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 14. Click **Apply Policy**.

__ 15. The policy editor adds a Results action to the rule. It moves the output of the JSON Web Sign action (the JWS) to the OUTPUT context.

__ 16. Close the policy editor.

__ 17. Click **Apply** for the MPGW.

Section 9. Test the JSON serialized JWS generation

In this section, you use a cURL command to send the JSON object to the service. The output is piped into a text file for use in a later section.

__ 1. Switch back to terminal window, and remain in the JWS subdirectory.

__ 2. Enter a cURL command to generate the JWS:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/SignBodyJSON >
SignedBodyJSONJWS.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file is formatted as a JSON object. It has a payload element, and a signatures array element. Only one signature is in the array.

__ 5. Use any internet-available base64url decoder site to decode the header and payload. The protected header decodes as “`{"alg":"RS256","kid":"Sam"}`”. The payload decodes as “`{ "refNumber" : 11111, "lastName" : "Johnson" }`”.

__ 6. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 10. Verify a JSON serialized, single signature JWS

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to verify this form of the JWS, and pass it on to the baggage service.

10.1. Modify the service policy

The input to the service is the JSON serialized JWS that you generated previously. This service policy verifies the JWS, places the decoded payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `VerifyBodyJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/VerifyBodyJSON*`

__ 6. Add a **Verify** action to the rule.

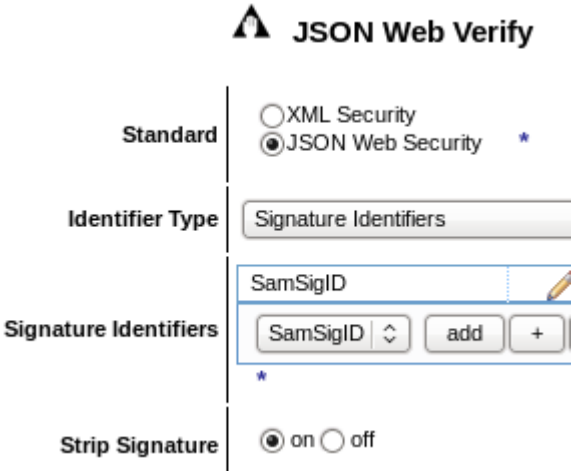
__ 7. Configure the Verify action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Verify** action with new options.

__ 8. Set the Identifier Type to **Signature Identifiers**.

__ 9. Select the Signature Identifier that you created earlier: **SamSigID**. This object specifies the verification certificate as Sam-cert, and the kid=Sam header parameter to match.

__ 10. Click **add** to put the SamSigID into the list.

__ 11. Set **Strip Signature** to **on** because you want the output context to contain the decoded JSON object.



__ 12. Click **Done**.

__ 13. If this request was a REST request, you would add a **Set Variable** action to build the correct URI to send to the back end. However, it is just a request that sends JSON in the HTTP body. Because the URI does not match any of the specific matching rules in the BaggageServiceProxy, that service's last matching rule processes the request. Fortunately, that last rule matches on any URI, so this request gets correctly processed.

__ 14. Click **Apply Policy**.

__ 15. Close the policy editor.

__ 16. Click **Apply** for the MPGW.

Section 11. Test the JSON serialized JWS verification and call the back-end baggage service

In this section, you use a cURL command to send the JSON serialized JWS to the service. The JWS is verified, and the decoded payload is sent to the back-end baggage service.

__ 1. Switch back to the terminal window, and remain in the JWS subdirectory.

__ 2. Enter this cURL command string into the terminal window:

```
curl --data-binary @SignedBodyJSONJWS.txt
http://<dp_public_ip>:<VerifyDecryptJOSE_port>/VerifyBodyJSON
```

__ 3. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 4. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 5. Save the configuration.

Section 12. Generate a JSON serialized JWS with multiple signatures

In this section, you modify the service policy of the SignEncryptJOSE MPGW to create a JSON serialized JWS that contains two signatures. Because the JSON Web Sign action does not support multiple signatures, a GatewayScript provides the function.

12.1. Extend the service policy

The input to the service is a JSON object that is passed in the HTTP body:

```
{
  "refNumber" : 11111,
  "lastName" : "Johnson"
}
```

This object is used to build the JSON serialized JWS.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `SignBodyMultiSigJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/SignBodyMultiSigJSON*`

__ 6. Add a **GatewayScript** action to the rule.

__ 7. Upload `<lab_files>/JWS/BuildMultiSignatureJWS.js` into the action. This GatewayScript creates a JWS that contains two signatures (“Sam” and “Seth”). In a later unit, the GatewayScript to manipulate JWSs and JWEs is explored.

__ 8. Click **Done**.

__ 9. Drag the **Advanced** action to the rule. Configure the action to be a **Set Variable** action.

__ 10. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 11. Click **Apply Policy**.

__ 12. The policy editor adds a Results action to the rule. It moves the output of the GatewayScript (the JWS) to the OUTPUT context.

__ 13. Close the policy editor.

__ 14. Click **Apply** for the MPGW.

Section 13. Test the JSON serialized JWS generation for multiple signatures

In this section, you use a cURL command to send the JSON object to the service. The output is piped into a text file for use in a later section.

__ 1. Switch back to the terminal window, and remain in the JWS subdirectory.

__ 2. Enter a cURL command to generate the JWS:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/SignBodyMultiSigJSON >
SignedBodyMultiSigJSONJWS.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file is formatted as a JSON object. It has a single payload element, and a signatures array element. Now two signatures are in the array.

__ 5. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 14. Verify a JSON serialized, multi-signature JWS

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to verify this form of the JWS, and pass it on to the baggage service.

14.1. Modify the service policy

The input to the service is the JSON serialized, multi-signature JWS that you generated previously. This service policy verifies the JWS, places the decoded payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `VerifyBodyMultiSigJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/VerifyBodyMultiSigJSON*`

__ 6. Add a **Verify** action to the rule.

__ 7. Configure the Verify action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Verify** action with new options.

__ 8. Set the Identifier Type to **Signature Identifiers**.

__ 9. Select the Signature Identifier that you created earlier: **SamSigID**. This object specifies the verification certificate as Sam-cert, and the kid=Sam header parameter to match.

__ 10. Click **add** to put the SamSigID into the list.

__ 11. Click **+** (New) to create another Signature Identifier object.

__ 12. Enter a Name of: `SethSigID`

__ 13. Select a Key Material Type of **Certificate** and Key Material of **Seth-cert**.

__ 14. Leave the **Valid algorithms** list empty. All valid algorithm values are acceptable.

__ 15. Add a Header Parameter with a Name of `kid` and a Value of `Seth`.

__ 16. Leave the **Verify** setting as **on**.

__ 17. Click **Apply**.

__ 18. Back on the JSON Web Verify page, verify that both signature identifiers are in the list.

__ 19. Set **Strip Signature** to **on** because you want the output context to contain the decoded JSON object.

__ 20. Click **Done**.

__ 21. If this request was a REST request, you would add a **Set Variable** action to build the correct URI to send to the back end. However, it is just a request that sends JSON in the HTTP body. Because the URI does not match any of the specific matching rules in the BaggageServiceProxy, that service's last matching rule processes the request. Fortunately, that last rule matches on any URI, so this request gets correctly processed.

__ 22. Click **Apply Policy**.

__ 23. Close the policy editor.

__ 24. Click **Apply** for the MPGW.

Section 15. Test the JSON serialized, multi-signature JWS verification and call the back-end baggage service

In this section, you use a cURL command to send the JSON serialized JWS to the service. The JWS is verified, and the decoded payload is sent to the back-end baggage service.

__ 1. Switch back to the terminal window, and remain in the JWS subdirectory.

__ 2. Enter this cURL command string into the terminal window:

```
curl --data-binary @SignedBodyMultiSigJSONJWS.txt
http://<dp_public_ip>:<VerifyDecryptJOSE_port>/VerifyBodyMultiSigJSON
```

__ 3. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 4. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 5. Save the configuration.

End of exercise

Exercise review and wrap-up

In this exercise, you worked with two different forms of input data: a string that represents a URI request parameter, and a JSON object that might be carried in an HTTP body. You used a JSON Web Sign action to create two different forms of the JWS: compact serialized and JSON serialized. You also uploaded a GatewayScript that builds a multi-signature JWS. You configured another MPGW that used a JSON Web Verify action to verify the JWSs that are received. This MPGW retrieved the payload from the JWS and passed the request to the baggage service back end.

Exercise 3. Creating and decrypting a JWE

Estimated time

01:00

Overview

A JWE is used to encrypt a payload that is either a JSON object or the URI request parameters in a REST request. This exercise covers the configuration of the JSON Web Encrypt action to generate a compact serialized JWE and a JSON serialized JWE. As part of the exercise, a JSON Web Decrypt is used to decrypt the encrypted payload that is passed in the JWE. The last section of the exercise encrypts a JWS into a JWE, decrypts the JWS that is in the JWE, and verifies the JWS.

Objectives

After completing this exercise, you should be able to:

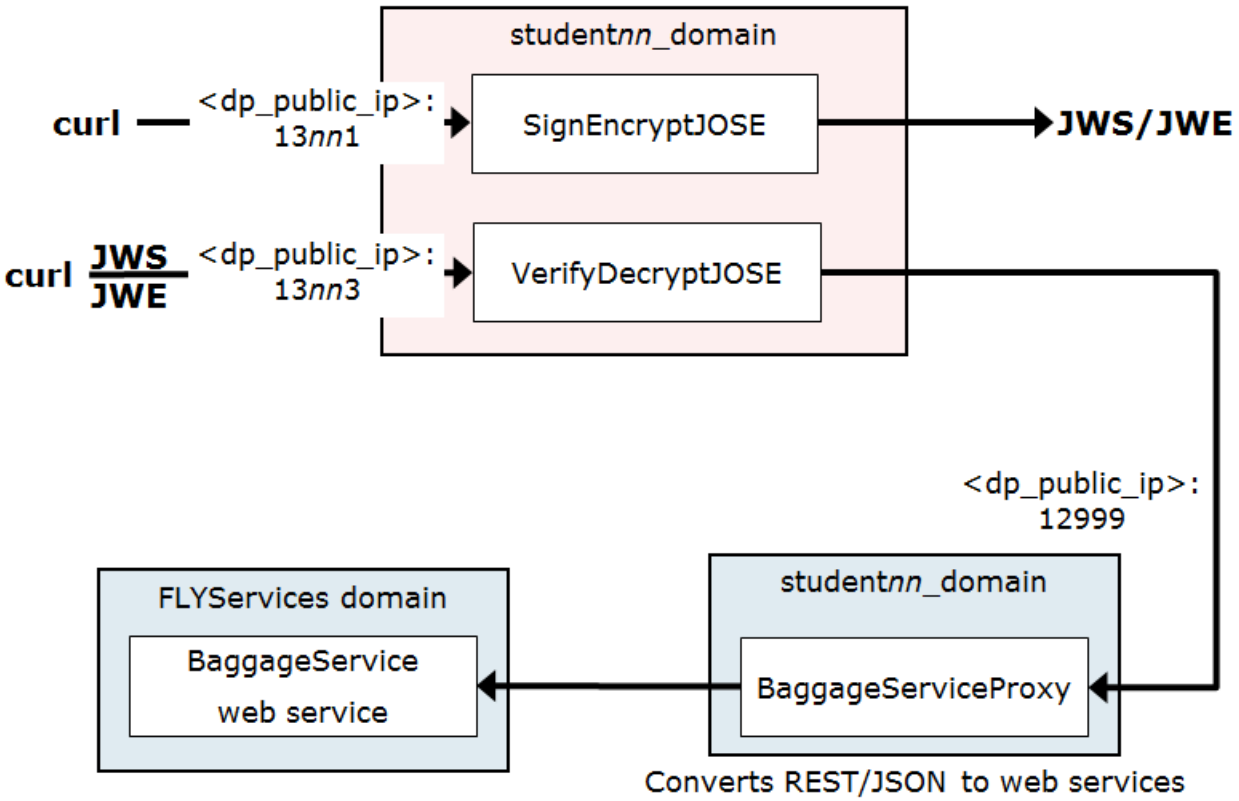
- Configure a JSON Web Encrypt action to generate a compact serialized and a JSON serialized JWE
- Configure a JSON Web Verify action to verify a compact serialized and a JSON serialized JWS
- Encrypt a JWS into the JWE, decrypt the JWE to get the JWS, and verify the JWS

Introduction

In this exercise, the student configures the processing actions in the policy editor to encrypt a payload in a JWE, and to decrypt that payload. Both compact and JSON serializations are used.

The student creates several multi-protocol gateways (MPGWs), and accesses a back-end web service to retrieve baggage information for a specific passenger. The services, domains, and functions are:

- `SignEncryptJOSE MPGW (student`*nn*`_domain)`: This MPGW is used to create a JWS or a JWE. “cURL” is used to send data to this service, and the output is a JWS or JWE.
- `VerifyDecryptJOSE MPGW (student`*nn*`_domain)`: This MPGW takes as input a JWS or JWE, and processes it to properly form the request to the back-end service. The back end of the service calls the BaggageServiceProxy MPGW. The response is returned to the cURL command in the terminal window.
- `BaggagServiceProxy (student`*nn*`_domain)`: This service is the MPGW that you configured in the previous exercise. It receives JSON or REST GET requests, and transforms them into a SOAP request. It converts the SOAP response into a JSON response.
- `BaggageService web service (FLYServices)`: This supplied service is the actual back-end web service that manages the baggage services.



The exercise assumes that the initial setup sections, testing the back-end service and importing the key materials, were completed during the previous JWS exercise. This exercise focuses on the JWE.

The flow of the JWE sections of this exercise is:

1) Configure the SignEncryptJOSE MPGW to create a compact serialized JWE for an HTTP GET request.
2) Use cURL to test the JWE generation.
3) Configure the VerifyDecryptJOSE MPGW to decrypt the JWE input, and pass the request on to the BaggageServiceProxy.
4) Use cURL to send the JWE to the VerifyDecryptJOSE MPGW. The response should be the baggage status.
5) Configure the SignEncryptJOSE MPGW to create a JSON serialized, single-recipient JWE for an HTTP POST request.
6) Use cURL to test the JWE generation.
7) Configure the VerifyDecryptJOSE MPGW to decrypt the JWE input, and pass the request on to the BaggageServiceProxy.
8) Use cURL to send the JWE to the VerifyDecryptJOSE MPGW.
9) Configure the SignEncryptJOSE MPGW to create a JSON serialized, multi-recipient JWE for an HTTP POST request.
10) Use cURL to test the JWE generation.

11) Configure the VerifyDecryptJOSE MPGW to decrypt the JWE input, and pass the request on to the BaggageServiceProxy.
12) Use cURL to send the JWE to the VerifyDecryptJOSE MPGW.
13) Configure the SignEncryptJOSE MPGW to create a compact serialized JWS, and then encrypt it into a JWE.
14) Use cURL to test the JWS/JWE generation.
15) Configure the VerifyDecryptJOSE MPGW to decrypt the JWE input, verify the JWS contents, and pass the request on to the BaggageServiceProxy.
16) Use cURL to send the JWE to the VerifyDecryptJOSE MPGW.

Requirements

To complete this exercise, you need:

- Access to the **DataPower** gateway
- **SoapUI**, for sending requests to the DataPower gateway
- The **BaggageStatusMockService** web service that runs on the DataPower gateway in the `FLYServices` domain
- The **BaggageServiceProxy** service that runs on the DataPower gateway in the `BaggageServiceProxy_domain` domain
- Access to the `<lab_files>` directory
- Completion of the previous JOSE signing exercise

Section 1. Preface

Remember to use the domain and port address that you were assigned in the exercise setup. **Do *not*** use the default domain.

The references in exercise instructions refer to the following values:

- <*lab_files*>: Location of the student lab files. Default location is: `/usr/labfiles/dp/`
- <*dp_internal_ip*>: IP address of the DataPower gateway development and administrative interface.
- <*dp_public_ip*>: IP address of the public services on the gateway that are used by customers and clients.
- <*nn*>: Assigned student number. If the course has no instructor, use "`01`".
- <*studentnn*>: Assigned DataPower user name and user account. If the course has no instructor, use "`student01`".
- <*studentnn_password*>: DataPower account password. In most cases, the initial value is the same as the user name. You are prompted to create a password on first use. Write it down.
- <*studentnn_domain*>: Application domain that the user account is assigned to. If the course has no instructor, use "`student01_domain`".
- <*SignEncryptJOSE_port*>: `13`*nn*`1`, where "*nn*" is the two-digit student number. This number is the listener port for the SignEncryptJOSE MPGW.
- <*VerifyDecryptJOSE_port*>: `13`*nn*`3`, where "*nn*" is the two-digit student number. This number is the listener port for the VerifyDecryptJOSE MPGW.

Section 2. Create a compact serialized JWE

In this section, you modify the service policy in the SignEncryptJOSE MPGW that was created in an earlier exercise. You modify the service policy to create a compact serialized JWE.

2.1. Modify the service policy

The input to the service is the request parameter string that is used to build the JWE: `refNumber=11111&lastName=Johnson`. The rule in this service policy converts the input string to a compact serialized JWE.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `EncryptURIcompact` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/EncryptURIcompact*`

__ 6. Add an **Encrypt** action to the rule.

__ 7. Configure the Encrypt action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Encrypt** action with new options.

__ 8. Leave the Serialization at **Compact**.

__ 9. Set the Algorithm to: `A128CBC-HS256`

__ 10. Click the **+** button to create a JWE Header object.

__ 11. Enter a Name: `EmiJWEHeader`

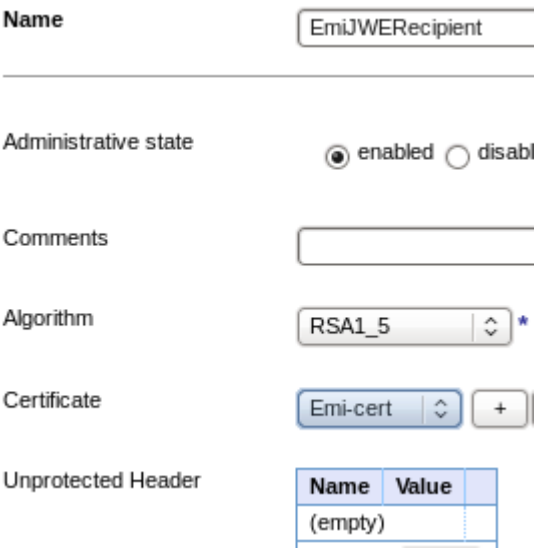
__ 12. Because you are building a compact serialized JWE, any header parameters that you want to include *must* be in a protected header. To help the header matching operation of the JSON Web Decrypt, add a protected header parameter with a Name of `kid` and a Value of `Emi`.

__ 13. Click the **+** button to create a JWE Recipient object.

__ 14. Name the object: `EmiJWERecipient`

__ 15. Select the **RSA1_5** (RSA) algorithm.

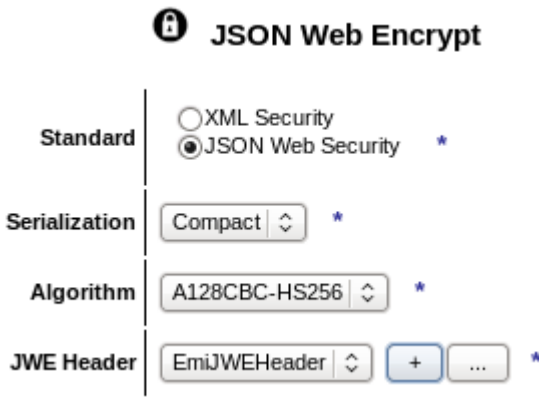
__ 16. Select the **Emi-cert** certificate that you imported in the previous exercise.



__ 17. Click **Apply**.

__ 18. Click **Apply** on the JWE Header page because it is now complete.

__ 19. On the JSON Web Encrypt page, click **Done**.



__ 20. On the configuration path, hover over the JSON Web Encrypt action. The input context is set to INPUT.

__ 21. Drag the **Advanced** action to the rule. Configure the action to a **Set Variable** action.

__ 22. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 23. Click **Done**.

__ 24. Click **Apply Policy**.

__ 25. The policy editor adds a Results action to the rule. It moves the output of the JSON Web Encrypt action (the JWE) to the OUTPUT context.

__ 26. Close the policy editor.

__ 27. Click **Apply** for the MPGW.

Section 3. Test the compact serialized JWE generation

In this section, you use a cURL command to send the request parameter string to the service. The output is piped into a text file for use in a later section.

__ 1. Open a terminal window. Look for an icon in the launcher bar at the left side of the desktop.

__ 2. Change the directory to the JWE directory:

```
cd <lab_files>/JWE
```

__ 3. Enter a cURL command to generate the JWE:

```
curl --data-binary @URIstring.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/EncryptURIcompact >
EncryptedURIcompactJWE.txt
```

__ 4. Use gedit to examine the output file.

__ 5. Notice that the file has the five parts of a compact serialized JWE: the base64url-encoded JWE protected header, a period, the base64url-encoded encrypted content encryption key (CEK), another period, the base64url-encoded initialization vector, another period, the base64url-encoded ciphertext, another period, and the base64url-encoded authentication tag.

__ 6. Use any internet-available base64url decoder site to decode the header and payload. The protected header decodes as “`{"enc":"A128CBC-HS256","kid":"Emi","alg":"RSA1_5"}`”.

__ 7. Leave the file open, as you are going to use this string for the next test. Do *not* add any extra line breaks.

__ 8. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 4. Decrypt a compact serialized JWE

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to decrypt a compact serialized JWE.

4.1. Configure the service policy

The input to the service is the compact serialized JWS that you generated previously. This service policy verifies the JWS, places the decoded payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `DecryptURIcompact` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/DecryptURIcompact*`

__ 6. Add a **GatewayScript** action to the rule.

__ 7. Upload `<lab_files>/JWE/EncryptedURI2JWE.js` into the action. This GatewayScript extracts the JWE from the input URI and places it into the output context for the JSON Web Decrypt action that follows.

__ 8. Click **Done**.

__ 9. Add a **Decrypt** action to the rule.

__ 10. Configure the Decrypt action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Decrypt** action with new options.

__ 11. Set the Identifier Type to **Single Identifier - Private Key**.

__ 12. Notice that the only option is to specify the decrypting private key. No other option identifies any of the possible protected header parameters.

Important

The suggestion is to use the “Recipient Identifiers” choice so that more protected header parameters can be specified, even when you have just one recipient. This situation is similar to the one that was identified for the JSON Web Verify action in an earlier exercise.

__ 13. Set the Identifier Type to **Recipient Identifiers**.

__ 14. Click the **+** button to create a Recipient Identifier object.

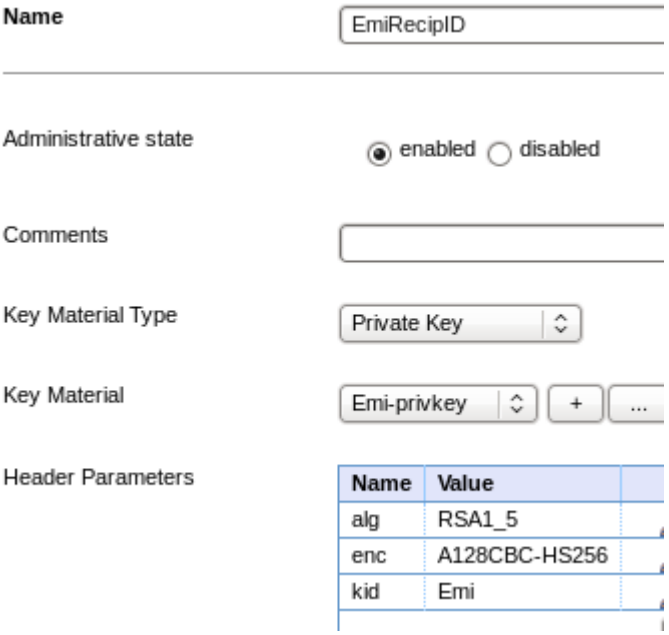
__ 15. Enter a Name: `EmiRecipID`

__ 16. Select the Key Material Type of **Private Key**.

__ 17. Select the **Emi-privkey** Key Material that you imported in an earlier exercise.

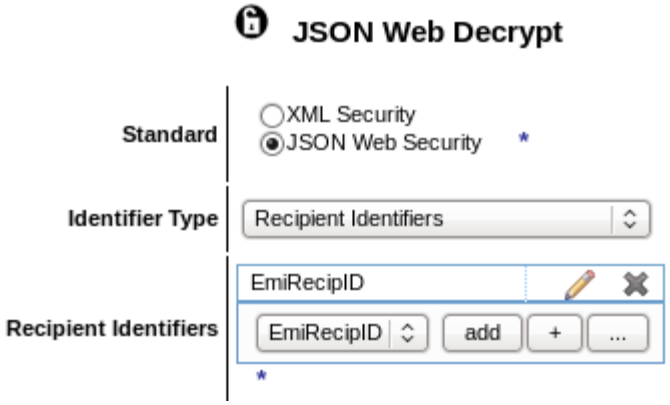
__ 18. Add several Header Parameters to aid in the header matching operation of the Decrypt:

- Name: `alg` Value: `RSA1_5`
- Name: `enc` Value: `A128CBC-HS256`
- Name: `kid` Value: `Emi`



__ 19. Click **Apply** for the Recipient Identifier object.

__ 20. On the JSON Web Decrypt page, click **Done**.



__ 21. Drag another **GatewayScript** action to the rule.

__ 22. Upload `<lab_files>/JWE/UnencryptedURI2Variable.js` into the action. This GatewayScript creates the URI that the BaggageServiceProxy expects. It contains the same code as the `JWS/UnsignedURI2Variable.js` module.

__ 23. Click **Done**.

__ 24. Click **Apply Policy**.

__ 25. Close the policy editor.

__ 26. Click **Apply** for the MPGW.

Section 5. Test the compact serialized JWE decryption and call the back-end baggage service

In this section, you use a cURL command to send the JWE to the service. The JWES is decrypted, and the plaintext payload is sent to the back-end baggage service.

__ 1. Use gedit to construct a cURL command that contains the JWE as a URI string. Type in the cURL command that follows in a few steps. Open the `EncryptedURIcompactJWE.txt` file that contains the JWE. Copy the contents (no added line breaks) into the cURL command string. Copy the complete cURL command string.

__ 2. Switch back to terminal window, and remain in the JWE subdirectory.

__ 3. Paste the cURL command string into the terminal window and press **Enter**:

```
curl -G
"http://<dp_public_ip>:<VerifyDecryptJOSE_port>/DecryptURIcompact?<contents
of EncryptedURIcompactJWE.txt>"
```

__ 4. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 5. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 6. Save the configuration.

Section 6. Generate a JSON serialized JWE with a single recipient

A JSON serialized JWE is not designed to be URL-safe, nor is it designed to be a size that is restricted. It is expected to be passed in an HTTP body, rather than in the URI. In this section, you modify the service policy of the SignEncryptJOSE MPGW to create a JSON serialized JWE.

6.1. Extend the service policy

The input to the service is a JSON object that is passed in the HTTP body:

```
{
  "refNumber" : 11111,
  "lastName" : "Johnson"
}
```

This object is used to build the JSON serialized JWE.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `EncryptBodyJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/EncryptBodyJSON`

__ 6. Add an **Encrypt** action to the rule.

__ 7. Configure the Encrypt action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Encrypt** action with new options.

__ 8. Set the Serialization to **General JSON**.

__ 9. Set the Algorithm to: `A128CBC-HS256`

__ 10. Click the **+** button to create a JWE Header object.

__ 11. Enter a Name: `EmiJSONJWEHeader`

__ 12. Because you are building a JSON serialized JWE, any header parameters that you want to be specific to a recipient *must* be in an unprotected header. This configuration differs from the configuration for compact serialization. Do *not* add a protected header parameter for “kid” here.

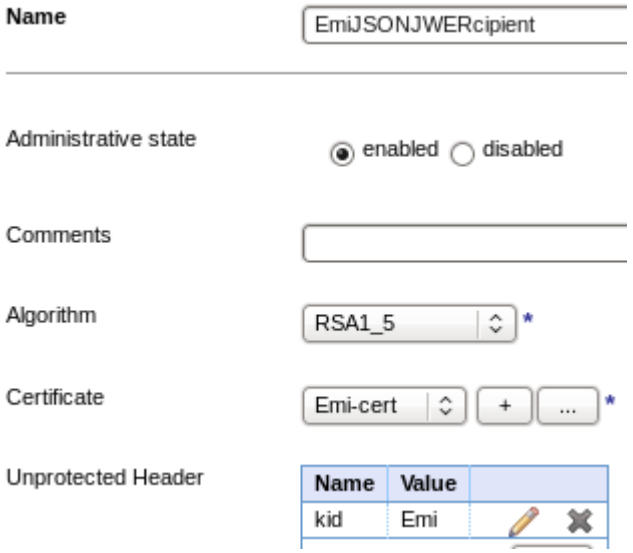
__ 13. Click the **+** button to create a JWE Recipient object.

__ 14. Specify a Name of: `EmiJSONJWERecipient`

__ 15. Select the **RSA1_5** (RSA) algorithm.

__ 16. Select the **Emi-cert** certificate that you imported in the previous exercise.

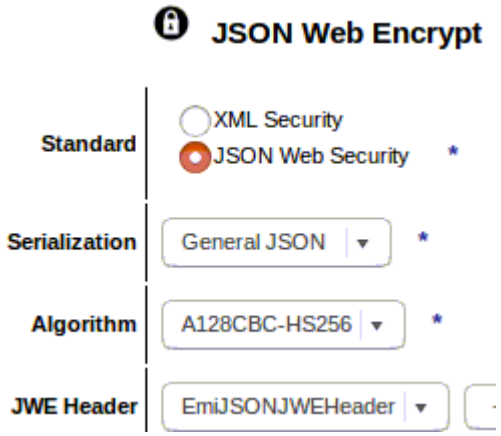
__ 17. To help the header matching operation of the JSON Web Decrypt, add an unprotected header parameter with a Name of `kid` and a Value of `Emi`.



__ 18. Click **Apply** for the JWE Recipient.

__ 19. Click **Apply** on the JWE Header page because it is now complete.

__ 20. On the JSON Web Encrypt page, click **Done**.



__ 21. On the configuration path, hover over the JSON Web Encrypt action. The input context is set to INPUT.

__ 22. Drag the **Advanced** action to the rule. Configure the action to be a **Set Variable** action.

__ 23. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 24. Click **Apply Policy**.

__ 25. The policy editor adds a Results action to the rule. It moves the output of the JSON Web Encrypt action (the JWE) to the OUTPUT context.

__ 26. Close the policy editor.

__ 27. Click **Apply** for the MPGW.

Section 7. Test the JSON serialized JWE generation

In this section, you use a cURL command to send the JSON object to the service. The output is piped into a text file for use in a later section.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter a cURL command to generate the JWE:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/EncryptBodyJSON >
EncryptedBodyJSONJWE.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file is formatted as a JSON object. It has a recipients array with a single recipient element, a protected element, a ciphertext element, an initialization vector element, and an authentication tag element.

__ 5. Use any internet-available base64url decoder site to decode the header. The protected header decodes as “{"enc":"A128CBC-HS256","alg":"RSA1_5"}”.

__ 6. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 8. Decrypt a JSON serialized, single-recipient JWE

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to verify this form of the JWE, and pass it on to the baggage service.

8.1. Modify the service policy

The input to the service is the JSON serialized JWE that you generated previously. This service policy decrypts the JWE, places the plaintext payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `DecryptBodyJSON` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/DecryptBodyJSON`

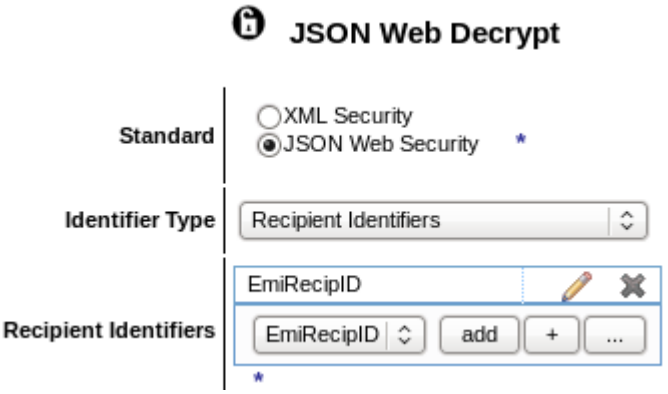
__ 6. Add a **Decrypt** action to the rule.

__ 7. Configure the Decrypt action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Decrypt** action with new options.

__ 8. Set the Identifier Type to **Recipient Identifiers**.

__ 9. Select the Recipient Identifier that you created earlier: **EmiRecipID**. This object specifies the decryption key as Emi-privkey, `alg=RSA1_5`, `enc=A128CBC-HS256`, and the `kid=Emi` header parameters to match.

__ 10. Click **add** to put the EmiRecipID into the list.



__ 11. Notice that the Decrypt action does not need the serialization type. It determines that from the format of the input message.

__ 12. Click **Done**.

__ 13. Click **Apply Policy**.

__ 14. Close the policy editor.

__ 15. Click **Apply** for the MPGW.

Section 9. Test the JSON serialized, single-recipient JWE decryption and call the back-end baggage service

In this section, you use a cURL command to send the JSON serialized JWE to the service. The JWE is decrypted, and the plaintext payload is sent to the back-end baggage service.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter this cURL command string into the terminal window:

```
curl --data-binary @EncryptedBodyJSONJWE.txt
http://<dp_public_ip>:<VerifyDecryptJOSE_port>/DecryptBodyJSON
```

__ 3. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 4. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 5. Save the configuration.

Section 10. Generate a JSON serialized JWE with multiple recipients

In this section, you modify the service policy of the SignEncryptJOSE MPGW to create a JSON serialized JWE that contains two recipients. Because the JSON Web Encrypt action does not support multiple signatures, a GatewayScript provides the function.

10.1. Extend the service policy

The input to the service is a JSON object that is passed in the HTTP body:

```
{
  "refNumber" : 11111,
  "lastName" : "Johnson"
}
```

This object is used to build the JSON serialized JWS.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `BuildMultiRecipientJWE` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/BuildMultiRecipientJWE`

__ 6. Add a **GatewayScript** action to the rule.

__ 7. Upload `<lab_files>/JWE/BuildMultiRecipientJWE.js` into the action. This GatewayScript creates a JSON serialized JWE that contains two recipients, each with a different algorithm. You can use **View** to review the code. In a later unit, the GatewayScript to manipulate JWSs and JWEs is explored.

__ 8. Click **Done**.

__ 9. Drag the **Advanced** action to the rule. Configure the action to be a **Set Variable** action.

__ 10. Set the Variable Name to `service/mpgw/skip-backside` and set the Variable Assignment to **1**.

__ 11. Click **Apply Policy**.

__ 12. The policy editor adds a Results action to the rule. It moves the output of the GatewayScript (the JWE) to the OUTPUT context.

__ 13. Close the policy editor.

__ 14. Click **Apply** for the MPGW.

Section 11. Test the JSON serialized JWE generation for multiple recipients

In this section, you use a cURL command to send the JSON object to the service. The output is piped into a text file for use in a later section.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter a cURL command to generate the JWE:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/BuildMultiRecipientJWE >
MultiRecipientJWE.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file is formatted as a JSON object. It has a recipients array with two recipients, a protected header element, the ciphertext, the initialization vector, and the authentication tag element.

__ 5. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 12. Decrypt a JSON serialized, multi-recipient JWE

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to verify this form of the JWE, and pass it on to the baggage service.

12.1. Modify the service policy

The input to the service is the JSON serialized, multi-recipient JWE that you generated previously. This service policy decrypts the JWE, places the plaintext payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `DecryptMultiRecipientJWE` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/DecryptMultiRecipientJWE`

__ 6. Add a **Decrypt** action to the rule.

__ 7. Configure the Decrypt action to use a Standard of **JSON Web Security**. This selection causes the page to refresh as a **JSON Web Decrypt** action with new options.

__ 8. Set the Identifier Type to **Recipient Identifiers**.

__ 9. Select the Recipient Identifier that you created earlier: **EmiRecipID**. This object specifies the decrypt key as Emi-privkey, and the Header Parameters of `alg=RSA1_5`, `enc=A128CBC-HS256`, and `kid=Emi` to match.

__ 10. Click **add** to put the EmiRecipID into the list.

__ 11. Click **+** (New) to create another Recipient Identifier object.

__ 12. Enter a Name of: `ErinRecipID`

__ 13. Select the Key Material Type of **Private Key**.

__ 14. Select the **Erin-privkey** Key Material that you imported in an earlier exercise.

__ 15. Add several Header Parameters to aid in the header matching operation of the Decrypt:

- Name: `alg` Value: `RSA-OAEP`
- Name: `enc` Value: `A128CBC-HS256`
- Name: `kid` Value: `Erin`

__ 16. Click **Apply**.

__ 17. On the JSON Web Decrypt page, the two recipient identifiers should be in the list. Click **Done**.

__ 18. Click **Apply Policy**.

__ 19. Close the policy editor.

__ 20. Click **Apply** for the MPGW.

Section 13. Test the JSON serialized, multi-recipient JWE decryption and call the back-end baggage service

In this section, you use a cURL command to send the JSON serialized JWS to the service. The JWS is verified, and the decoded payload is sent to the back-end baggage service.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter this cURL command string into the terminal window:

```
curl --data-binary @MultiRecipientJWE.txt
http://<dp_public_ip>:<VerifyDecryptJOSE_port>/DecryptMultiRecipientJWE
```

__ 3. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 4. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 5. Save the configuration.

Section 14. Generate a JSON serialized JWE with a signed payload

If you are concerned about just message integrity and message confidentiality, you can use just a JWE. If you also want non-repudiation, you must also use JWS. In this section, you modify the service policy of the SignEncryptJOSE MPGW to create a JSON serialized JWS, and then encrypt that into a JSON serialized JWE. Because you should now have some experience with signing and encrypting, the instructions are less detailed. You can reuse many of the objects that you created in earlier steps.

14.1. Extend the service policy

The input to the service is a JSON object that is passed in the HTTP body:

```
{
  "refNumber" : 11111,
  "lastName" : "Johnson"
}
```

This object is used to build the JSON serialized JWS.

__ 1. Edit the **SignEncryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `SignEncryptBody` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/SignEncryptBody`

__ 6. Add a **JSON Web Sign** action to the rule.

__ 7. Specify **General JSON** serialization.

__ 8. Use **SamSignature** as the referenced Signature object to sign it with Sam's criteria.

__ 9. Click **Done**. The output from this action is the JSON serialized JWS.

__ 10. Add a **JSON Web Encrypt** action to the rule.

__ 11. Specify **General JSON** serialization, and an algorithm of `A128CBC-HS256`.

__ 12. Because you want to use Emi's criteria to encrypt the JWS, and you are using JSON serialization, you can reuse **EmiJSONJWEHeader** as the referenced JWE header.

__ 13. Click **Done**. The output from this action is a JSON serialized JWE that contains the JWS as an encrypted payload.

__ 14. Add a **Set Variable** action to the rule. Have it set **service/mpgw/skip-backside** to **1**.

__ 15. Click **Apply Policy**.

__ 16. The policy editor adds a Results action to the rule. It moves the output of the JSON Web Encrypt (the JWE) to the OUTPUT context.

__ 17. Close the policy editor.

__ 18. Click **Apply** for the MPGW.

Section 15. Test the JSON serialized JWE generation for a JWS

In this section, you use a cURL command to send the JSON object to the service. The output is piped into a text file for use in a later section.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter a cURL command to generate the JWE:

```
curl --data-binary @RefnumLastnameRequest.txt
http://<dp_public_ip>:<SignEncryptJOSE_port>/SignEncryptBody >
SignedEncryptedJWE.txt
```

__ 3. Use gedit to examine the output file.

__ 4. Notice that the file is formatted as a JSON object. It has a recipients array with one recipient, a protected header element, the ciphertext, the initialization vector, and the authentication tag element. Recall that the ciphertext contains the encrypted JWS.

__ 5. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

Section 16. Verify a JSON serialized JWS that is inside a JWE

In this section, you modify the service policy of the VerifyDecryptJOSE MPGW to decrypt the JWE, then to verify the JWS, and pass it on to the baggage service.

16.1. Modify the service policy

The input to the service is the JSON serialized JWE that you generated previously. This service policy decrypts the JWE, places the plaintext payload into the URI service variable, and calls the BaggageServiceProxy. The response is returned to the terminal window.

__ 1. Edit the **VerifyDecryptJOSE** MPGW.

__ 2. Click the **...** (Edit) button for the Multi-Protocol Gateway Policy.

__ 3. Click **New Rule**.

__ 4. Specify a Rule Name of `DecryptVerifyBody` and set the direction to **Client to Server**.

__ 5. Configure the Match action to match on a URL of: `/DecryptVerifyBody`

__ 6. Add a **JSON Web Decrypt** action to the rule.

__ 7. Set the Identifier Type to **Recipient Identifiers**.

__ 8. Because you used Emi's criteria to encrypt, you can reuse **EmiRecipID** as the referenced recipient identifier.

__ 9. Click **Done**. The output from this action is the JSON serialized JWS.

__ 10. Add a **JSON Web Verify** action to the rule.

__ 11. Set the Identifier Type to **Signature Identifiers**.

__ 12. Because you used Sam's criteria to encrypt, you can reuse **SamSigID** as the referenced signature identifier.

__ 13. Strip the signature so that the output is the original payload.

__ 14. Click **Done**. The output from this action is the JSON object that is used by the back-end service.

__ 15. Click **Apply Policy**.

__ 16. Close the policy editor.

__ 17. Click **Apply** for the MPGW.

Section 17. Test the JSON serialized JWE decryption and JWS verification and call the back-end baggage service

In this section, you use a cURL command to send the JSON serialized JWE to the service. This JWE contains the encrypted JWS as its payload. The JWE is decrypted into the JWS. The JWS is verified, and the decoded payload is sent to the back-end baggage service.

__ 1. Switch back to the terminal window, and remain in the JWE subdirectory.

__ 2. Enter this cURL command string into the terminal window:

```
curl --data-binary @SignedEncryptedJWE.txt
http://<dp_public_ip>:<VerifyDecryptJOSE_port>/DecryptVerifyBody
```

__ 3. You should get back the baggage status that is returned from the BaggageServiceProxy:

```
{
  "firstName":"James",
  "lastName":"Johnson",
  "bags":[
    {
      "status":"On Belt",
      "destination":"LHR",
      "timeAtLastKnownLocation":"Thu Jun 19 07:27 UTC 2014",
      "lastKnownLocation":"DEL",
      "id":"1289"
    },
       ...
}
```

__ 4. If you want, you can enable the probe and examine the context contents at different steps in the rule processing.

__ 5. Save the configuration.

End of exercise

Exercise review and wrap-up

In this exercise, you worked with two different forms of input data: a string that represents a URI request parameter, and a JSON object that might be carried in an HTTP body. You used a JSON Web Encrypt action to create two different forms of the JWS: compact serialized and JSON serialized. You also uploaded a GatewayScript that builds a multi-recipient JWE. You configured another MPGW that used a JSON Web Decrypt action to verify the JWEs that are received. This MPGW retrieved the payload from the JWE and passed the request to the baggage service back end.

The last few sections of the exercise combined signatures and encryption, which is a common situation in real-world applications. You created a JWS, and then encrypted it into a JWE. You then decrypted the JWE to retrieve the JWS, and then verified the JWS.

Exercise 4. Using GatewayScript to work with a JWS and a JWE

Estimated time

01:00

Overview

Instead of using the processing actions, you can work with a JWS and a JWE by using the JOSE objects and methods that are available in GatewayScript. In this exercise, you write GatewayScript to create and verify signatures in a JWS, and encrypt and decrypt a JWE.

Objectives

After completing this exercise, you should be able to:

- Use the JWSHeader, JWSSigner, and JWSVerifier classes to manipulate a JWS
- Use the JWEHeader, JWEEncrypter, and JWEDecrypter classes to manipulate a JWE

Introduction

In the previous exercises, you used JSON Web Sign, JSON Web Verify, JSON Web Encrypt, and JSON Web Decrypt processing actions to manipulate a JWS and a JWE. In this exercise, you use the GatewayScript action to do similar processing. The 7.5 firmware provides a **jose** module that contains many classes and methods to work with JWSs and JWEs.

In the first section of this exercise, you create an XML firewall service. You create a processing rule that has a GatewayScript action to generate a JWS. cURL is used to send a payload to the service to generate the JWS. The returned JWS is stored in a file.

In the next section, you add a rule that receives a JWS and verifies the signature. You code a GatewayScript to do the verification, and return the original payload. A cURL command is used to send the JWS that was stored in a file in the previous section.

The third section is similar to the first section, except that you are now working with encryption. The GatewayScript action encrypts a payload into a JWE. The returned JWE is stored in a file.

In the fourth section, another GatewayScript is used to decrypt the JWE. A cURL command is used to send the JWE that was stored in a file in the previous section.

The last section suggests several optional activities.

The instructions in this exercise are at a high level, and require the students to create the GatewayScript on their own. Code hints are available in the lecture materials, and in the sample scripts on the appliance in the `store:///gatewayscript` directory. You can use the CLI debugger facility to debug your GatewayScript.

Requirements

To complete this exercise, you need:

- Access to the **DataPower** appliance
- **cURL**, for sending requests to the DataPower appliance
- Access to the `<lab_files>` directory

Section 1. Preface

Remember to use the domain and port address that you were assigned in the exercise setup. **Do *not*** use the default domain.

The references in exercise instructions refer to the following value:

- <*lab_files*>: Location of the student lab files. Default location is: `/usr/labfiles/dp/`
- <*dp_internal_ip*>: IP address of the DataPower appliance development and administrative interface.
- <*dp_public_ip*>: IP address of the public services on the appliance that are used by customers and clients.
- <*nn*>: Assigned student number. If the course has no instructor, use "`01`".
- <*studentnn*>: Assigned DataPower user name and user account. If the course has no instructor, use "`student01`".
- <*studentnn_password*>: DataPower account password. In most cases, the initial value is the same as the user name. You are prompted to create a password on first use. Write it down.
- <*studentnn_domain*>: Application domain that the user account is assigned to. If the course has no instructor, use "`student01_domain`".
- <*SignEncryptJOSE_port*>: `13`*`nn`*`1` where "*nn*" is the two-digit student number. This number is the listener port for the SignEncryptJOSE MPGW.
- <*VerifyDecryptJOSE_port*>: `13`*`nn`*`3` where "*nn*" is the two-digit student number. This number is the listener port for the VerifyDecryptJOSE MPGW.
- <*GWS_XMLFW_port*>: `13`*`nn5`* where "*nn*" is the two-digit student number. This number is the listener port for the GWSFirewall XML firewall service.

Section 2. Create a JWS by using a GatewayScript

In this section, you create an XML firewall service to contain the rules for working with JWSs and JWEs from GatewayScripts. You then update the service policy to generate a compact serialized JWS and store it in a file for later use.

__ 1. Use gedit to review the payload at `<lab_files>/JWSJWEGWS/payload.json`. This JSON is the payload that you sign and encrypt later.

__ 2. Use gedit to create a GatewayScript file named: `<lab_files>/JWSJWEGWS/CreateCompactJWS.js`

__ 3. Follow this structure to create the necessary script:

- __ a. Specify the “required” **jose** module.
- __ b. Use the `readAsBuffer` method to obtain the inbound payload from the input context.
- __ c. Add a **debugger** statement for possible CLI debugging.
- __ d. Create a JWSHeader object to define the header parameters for the JWS. Use the key from the earlier exercise: `Sam-privkey`. Specify a signing algorithm of: `RS256`
- __ e. Set a header parameter that is named `kid` with the value `Sam-privkey` in the protected header. This parameter is used for signature matching in the verify behavior.
- __ f. Create a JWSSigner instance by using the parameters that are defined in the JWSHeader object.
- __ g. Update the JWSSigner instance with the payload.
- __ h. Create the JWS in the **compact** serialization format.
- __ i. Place the JWS in the output context.

__ 4. Create a loopback XML firewall that is named `GWSFirewall` with the request type of Non-XML.

__ 5. Set the Front End Local address as *<dp_public_ip>* and a Port Number of *<GWS_XMLFW_port>*.

__ 6. Create a Processing Policy named: `JOSEModules`

__ 7. Create a request rule.

__ 8. Configure the Match action to match the URL: `*/JWSCompactSign`

__ 9. Add a GatewayScript action and upload the script that you created a few steps ago.

__ 10. Apply the service.

__ 11. Save your configuration.

__ 12. Open a terminal window, and change the directory to: `<lab_files>/JWSJWEGWS`

__ 13. Send the payload file to the XML firewall, and capture the resulting JWS in a file named: `jwscompact.txt`

```
curl --data-binary @payload.json http://<dp_public_ip>:
<GWS_XMLFW_port>/JWSCompactSign > jwscompact.txt
```

__ 14. Use gedit to review the compact serialized JWS in the output file. It is used as input to the next step.

__ 15. Keep the terminal window open for the next sections.

Section 3. Use a GatewayScript to verify a signature in a JWS

In this section, you create a GatewayScript that verifies a signature in a JWS. Then, you add another processing rule to support the behavior. Lastly, you use cURL to test the behavior.

__ 1. Use gedit to create a GatewayScript file that is named: `<lab_files>/JWSJWEGWS/VerifyCompactJWS.js`.

Follow this structure to create the necessary script:

__ a. Specify the “required” **jose** module.

__ b. Use the `readAsBuffer` method to obtain the inbound JWS from the input context.

__ c. Add a **debugger** statement for possible CLI debugging.

__ d. Parse the received JWS to get the JWSObject instance.

__ e. Get the signature from the JWSObject instance. Although compact serialization supports only one signature, the retrieval method returns an array.

__ f. For the individual header, check for the **kid** parameter value of **Sam-privkey**. When it is found, set the key for the header to **Sam-cert**. This value is the certificate name to do the verification.

__ g. Create a JWSVerifier instance that references the JWSObject instance.

__ h. Validate (verify) the signature.

__ i. If the verification succeeds, get the original payload from the JWSObject and place it in the output context.

__ 2. In the XML firewall service, create another request rule.

__ 3. Configure the Match action to match on the URL: `*/JWSCompactVerify`

__ 4. Add a GatewayScript action and upload the script that you created a few steps ago.

__ 5. Apply the service.

__ 6. Save your configuration.

__ 7. Switch back to your terminal window.

__ 8. Send the JWS file to the XML firewall service. The result should be the original JSON payload.

```
curl --data-binary @jwscompact.txt http://<dp_public_ip>:
<GWS_XMLFW_port>/JWSCompactVerify
```

Section 4. Use a GatewayScript to encrypt a payload

In this section, you modify the service policy to create a JWE that contains the encrypted payload.

__ 1. Use gedit to create a GatewayScript file that is named: `<lab_files>/JWSJWEGWS/CreateCompactJWE.js`

Follow this structure to create the necessary script:

__ a. Specify the “required” **jose** module.

__ b. Use the `readAsBuffer` method to obtain the inbound payload from the input context.

__ c. Add a **debugger** statement for possible CLI debugging.

__ d. Create a JWEHeader object to define the header parameters for the JWE. Specify an encryption algorithm “enc” of **A128CBC-HS256**.

__ e. In the JWEHeader instance, set a protected header parameter that is named `alg` with the value `RSA1_5`. This parameter specifies the algorithm to use to encrypt the CEK that is passed in the JWE.

__ f. In the JWEHeader instance, set the key for encrypting the CEK as: `Emi-cert`

__ g. Create a JWEEncrypter instance by using the parameters that are defined in the JWEHeader object.

__ h. Update the JWEEncrypter instance with the payload.

__ i. Create the JWE in the **compact** serialization format.

__ j. Place the JWE in the output context.

__ 2. Edit the XML firewall service policy.

__ 3. Create another request rule.

__ 4. Configure the Match action to match the URL: `*/JWECompactEncrypt`

__ 5. Add a GatewayScript action and upload the script that you created a few steps ago.

__ 6. Apply the service.

__ 7. Save your configuration.

__ 8. Switch to the terminal window.

__ 9. Send the payload file to the XML firewall, and capture the resulting JWE in a file named: `jwecompact.txt`

```
curl --data-binary @payload.json http://<dp_public_ip>:
<GWS_XMLFW_port>/JWSCompactEncrypt > jwecompact.txt
```

__ 10. Use gedit to review the JWE in the output file. It is used as input to the next step.

Section 5. Use a GatewayScript to decrypt a payload in a JWE

In this section, you create a GatewayScript that decrypts the payload in the JWE that you previously created. Then, you use another processing rule to support the behavior. Lastly, you use cURL to test the behavior.

__ 1. Use gedit to create a GatewayScript file that is named: `<lab_files>/JWSJWEGWS/DecryptCompactJWE.js`

__ a. Follow this structure to create the necessary script:

__ b. Specify the “required” **jose** module.

__ c. Use the `readAsBuffer` method to obtain the inbound JWE from the input context.

__ d. Add a **debugger** statement for possible CLI debugging.

__ e. Parse the received JWE to get the JWEObject instance.

__ f. In the JWEObject instance, set the decryption key as: `Emi-privkey`

__ g. Create a JWEDecrypter instance that references the JWEObject instance.

__ h. Decrypt the JWE.

__ i. If the decryption succeeds, write the resulting plaintext (the original payload) to the output context.

__ 2. In the XML firewall service, create another request rule.

__ 3. Configure the Match action to match on the URL: `*/JWECompactDecrypt`

__ 4. Add a GatewayScript action and upload the script that you created a few steps ago.

__ 5. Apply the service.

__ 6. Save your configuration.

__ 7. Switch back to your terminal window.

__ 8. Send the JWE file to the XML firewall service. The result should be the original JSON payload.

```
curl --data-binary @jwecompact.txt http://<dp_public_ip>:
<GWS_XMLFW_port>/JWECompactDecrypt
```

Section 6. Optional opportunities

If you have time and interest, you can try some of these other JWS and JWE-related activities:

- Create a JSON serialized JWS and verify it. Select either “general JSON” or “flattened JSON” serialization at your discretion.
- Create a JSON serialized JWE and decrypt it.
- Create a multi-signature JWS and verify it.
- Create a multi-recipient JWE and decrypt it.
- Create a JWS, and encrypt it into a JWE. Decrypt the JWE, and verify the enclosed JWS.

End of exercise

Exercise review and wrap-up

In this exercise, you coded GatewayScript files to create a JWS, verify a JWS, create a JWE, and decrypt a JWE.

Appendix A. Exercise solutions

This appendix describes:

- The dependencies between the exercises and other tools.
- How to load the sample solution configurations for the various exercises. The solutions were exported from the appliance into a `.zip` file. You can import a sample solution into your domain.

Part 1: Dependencies

Certain exercises depend on previous exercises and on other resources, such as the need for the back-end application server to support service calls. The back-end application server that is on each student image uses the variable `<image_ip>`. The `image_ip` variable is the literal IP address of the student image. Students can discover their IP address by opening a terminal window, and entering: `/sbin/ifconfig`

Table 1. Dependencies

Exercise	Depends on exercise	Uses cURL	Uses SoapUI	Uses Baggage web service	Uses Booking web service
1: Using DataPower to implement REST services			Yes	Yes	
2: Create and verify a JWS	1	Yes		Yes	
3: Create and decrypt a JWE	2	Yes		Yes	
4: Use GatewayScript to work with a JWS and a JWE	3	Yes			

If the class is using the standard images and setup, the LDAP server is running on the student image. The Baggage and Booking services are running as services on the DataPower gateway. Therefore, each student is using a different IP address for the student image. Assuming that each student has their own DataPower virtual gateway, each student also has different IP addresses for the gateway.

If the exercises are run in the IBM remote lab environment, like Skytap, the IP addresses might be the same for each student because each student has a unique entry point into the virtualized environment.

Part 2: Importing solutions

Note

The solution files use port numbers that might already be in use. You must change the port numbers of the imported service. You might also find it necessary to update the location of the back-end application server that provides the web services.

__ 1. Determine the `.zip` file to import from the following table:

Table 2. Exercise solution files

Exercise	Compressed solution file name
1: Using DataPower to implement REST services	`RESTJOSE_REST.zip`
2: Create and verify a JWS	
3: Create and decrypt a JWE	`RESTJOSE_SignEncrypt.zip`
4: Use GatewayScript to work with a JWS and a JWE	`RESTJOSE_GatewayScript.zip`

__ a. The `.zip` file names begin with the naming convention `dev_Ex`*`NN`*, where *`NN`* represents the two-digit exercise number.

__ b. To import a solution to begin a new exercise, import the solution for the previous exercise. For example, if you are ready to start Exercise 10, you would import `<`*`lab_files`*`>/Solutions/dev_Ex10_caching.zip`.

__ 2. Import the `.zip` solution file into your application domain.

__ a. From the **Control Panel**, in the vertical navigation bar, click **Administration > Configuration > Import Configuration**.

__ b. Make sure that the selection for **From** is **ZIP Bundle** and the selection for **Where** is **File**.

__ c. Click **Browse** and navigate to your respective `.zip` solution file.

__ d. Click **Next**.

__ e. In the next page, leave the files selected. Scroll down and click **Import**.

__ f. Make sure that the import is successful. Click **Done**.

__ 3. *Be sure to update the port numbers and application server location* to your local values. Because private keys (key files) are not exported, *you also must create keys and certificates*. In some exercise solutions, the key files are exported in the `local:` directory. After import, you move those files into the `cert:` directory.

__ 4. The lab exercises call two web services, **Booking Service** and **Baggage Service**. Both of the web services are in the FLY service domain. To do the labs on another DataPower appliance, be sure to import the `dev_FLYservices_domain.zip` file in the **FLYServices** domain.

Appendix B. Lab environment setup

The appendix instructs how to set up the lab environment, including:

- Defining the literal variable values in SoapUI
- Testing the Booking and Baggage web service back ends
- Identifying the IP address of your student image
- Populating a convenient table with all the required variables that are used in this course

Part 1: Configure the SoapUI variables for use

The SoapUI tool supports specification of properties to reduce the redundant entry of the same value for testing. For these exercises, the client testing usually accesses the public interface of the student-created DataPower services. Rather than requiring the students to constantly enter the same value, the public IP address of the appliance is configured as a SoapUI property.

__ 1. Obtain the required variables for this course. The variable information can be found in at least on one of the following locations, based on the type of course you are taking:

- On the image desktop as a background
- On the image background from the SPVC you logged in to
- In an email you received with instructions for this course
- From your instructor if you are in a classroom (virtual or literal) environment
- In the exercise guide itself

The variables are:

- The DataPower appliance's public IP address `<dp_public_ip>`
- The DataPower appliance's internal IP address `<dp_internal_ip>`
- The (your) student image IP address `<image_ip>`
- Your student number (it is a two-digit number) `<nn>`

__ 2. Open SoapUI by using the icon on the desktop.

Attention

If you get a "new version available" message, close the message window and do not download or install any upgrades.

__ 3. Click **File - Preferences**.

__ 4. Click the **Global Properties** choice.

__ 5. Update the values for the following variables.

__ a. Double-click the **Value** cell for the **dp_internal_ip** property.

__ b. Replace the value (`x.x.x.x` or `1.2.3.4`) with the literal value of the *<dp_internal_ip>* address in the cell. That is, replace 1.2.3.4 with the IP address of the DataPower appliance that is being used for your class.

__ c. Click **Enter** while the cursor is in the cell. This action registers the new value.

__ d. Double-click the **Value** cell for the **dp_public_ip** property.

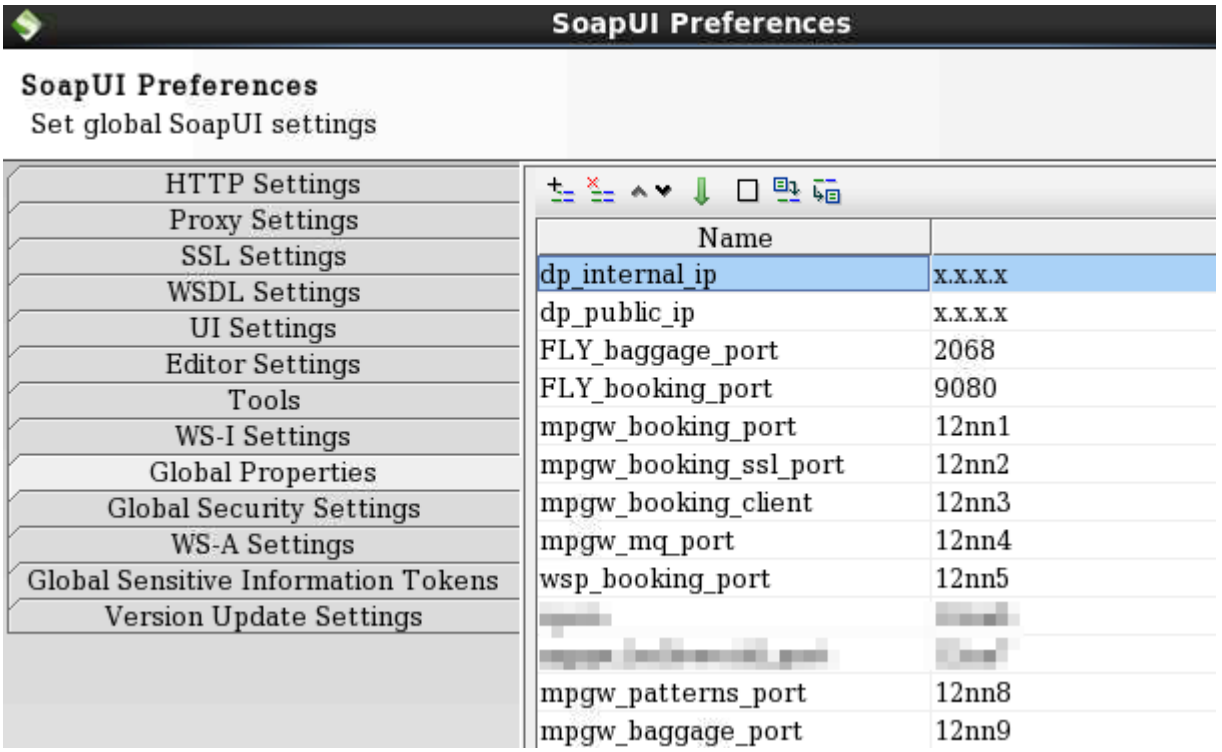
__ e. Replace the value (`x.x.x.x` or `1.2.3.4`) with the literal value of the *<dp_public_ip>* address in the cell. That is, replace 1.2.3.4 with the IP address of the DataPower appliance that is being used for your class.

__ f. Click **Enter** while the cursor is in the cell. This action registers the new value.

__ g. Double-click the **Value** cell for the **mpgw_booking_port** property.

__ h. Replace "*nn*" with your appropriate student number. For example, if you are `student 01`, the value for `mpgw_booking_port` of `12`*nn*`1` is updated to `12011`.

__ i. Click **Enter** while the cursor is in the Value cell. This action registers the new value.



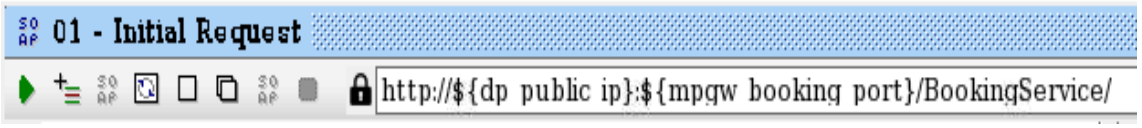
__ j. Repeat the previous steps g – i for the remaining values.

__ k. Click **OK**.

__ l. Click **File > Save Preferences**.

SoapUI is now configured for all exercises in this course. The messages that are sent to DataPower when using SoapUI reference these variables. No further SoapUI configuration is required, unless stated in the specific exercise.

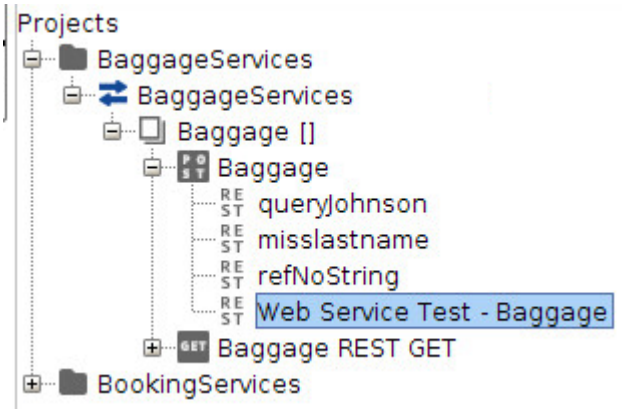
When SoapUI recognizes the `dp_public_ip` reference in a request (“`${dp_public_ip}` ”), it substitutes the correct IP address into the URL.



Part 2: Confirm that the Booking and Baggage web services are up

Test the Booking web service and the Baggage web service. The following steps ensure that the back-end web services are operational. In addition to testing the availability of the web service, it is also a useful troubleshooting technique to verify network connectivity to the back-end web service.

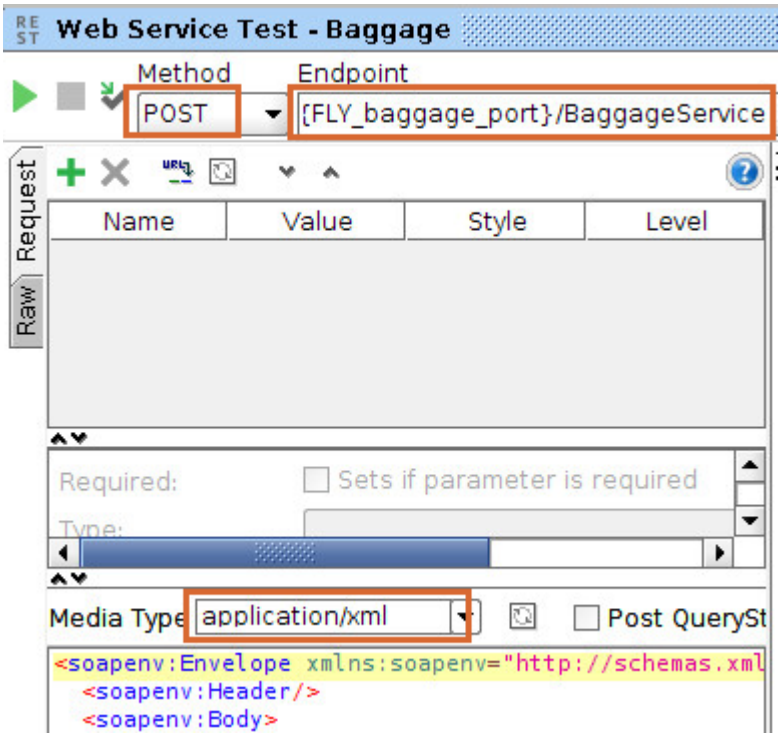
__ 1. In the project tree, expand the **BaggageServices** project until **Web Service Test - Baggage** is visible.



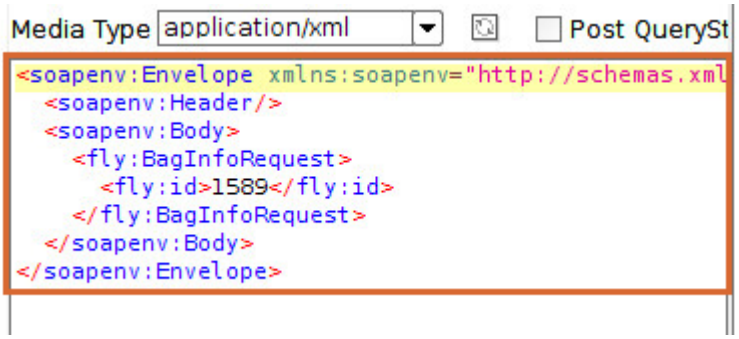
__ 2. Double-click **Web Service Test - Baggage** to open the request window. If a double-click does not work, right-click the request and click **Show Request Editor**.

__ 3. Ensure that the following information is preconfigured in the request message:

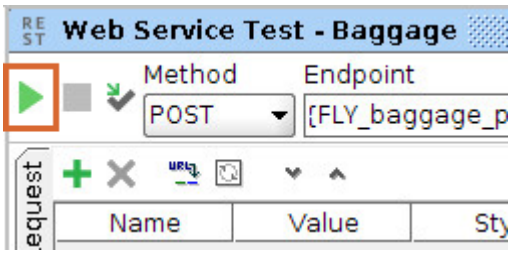
- Method: **POST**
- Endpoint: `http://${dp_internal_ip}:${FLY_baggage_port}/BaggageService`
- Media Type: **application/xml**



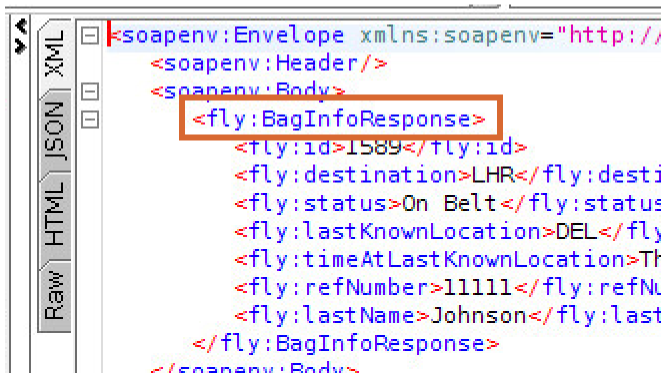
- Soap message:



__ 4. Click the green **Submit** arrow to send the request message directly to the **BaggageService** web service for FLY airlines.



__ 5. Confirm that a successful **BagInfoResponse** response is returned in the response tab.



Important

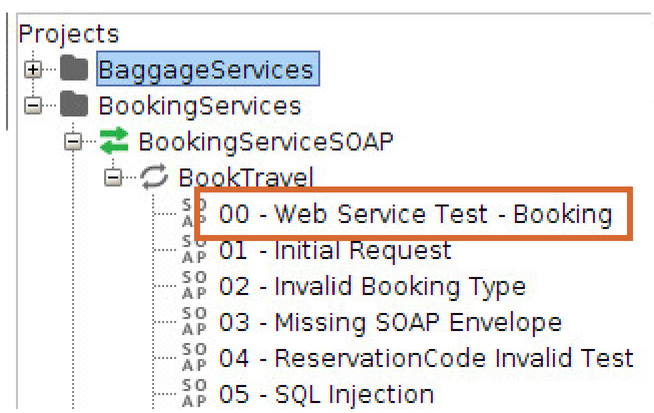
If you do not get the correct response, the failure can be due to several reasons:

- The variables that are entered in SoapUI General Preferences are not installed on the DataPower appliance.
- The DataPower appliance is unreachable from your student image due to some network connectivity issue.

Verify that you entered the correct values for the SoapUI variables. If the values are correct, escalate for assistance.

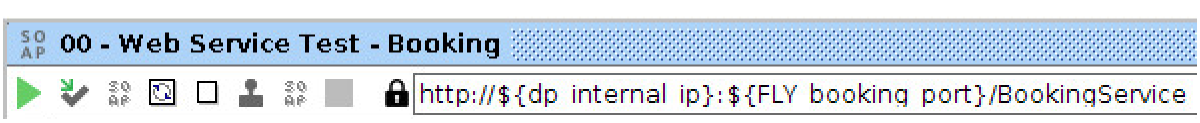
__ 6. Close the **Web Service Test - Baggage** window.

__ 7. In the project tree, expand the **BookingServices** project until **00 – Web Service Test - Booking** is visible.



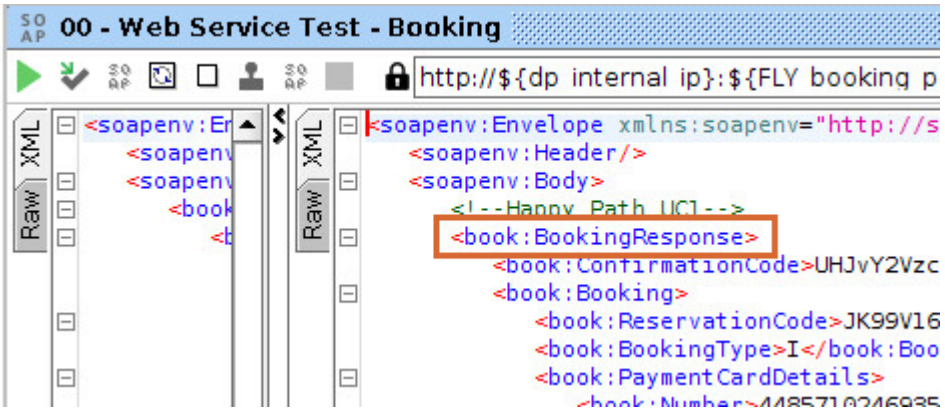
__ 8. Double-click **00 – Web Service Test - Booking** to open the request window. If a double-click does not work, right-click the request and click **Show Request Editor**.

__ 9. Confirm that the URL address field contains:
`http://${dp_internal_ip}:${FLY_booking_port}>/BookingService`



__ 10. Click the green **Submit** arrow that is to the left of the URL address field to send the SOAP request test message directly to the FLY Airlines Booking web service.

__ 11. If everything worked properly, you should see the `<book:BookingResponse>` XML tree in the Response tab.



Important

If you do not get the correct response, the failure can be due to several causes:

- The variables that are entered in SoapUI General Preferences are wrong.
- The FLYService domain that contains the web services is not installed on the DataPower appliance.
- The DataPower appliance is unreachable from your student image due to a network connectivity issue.

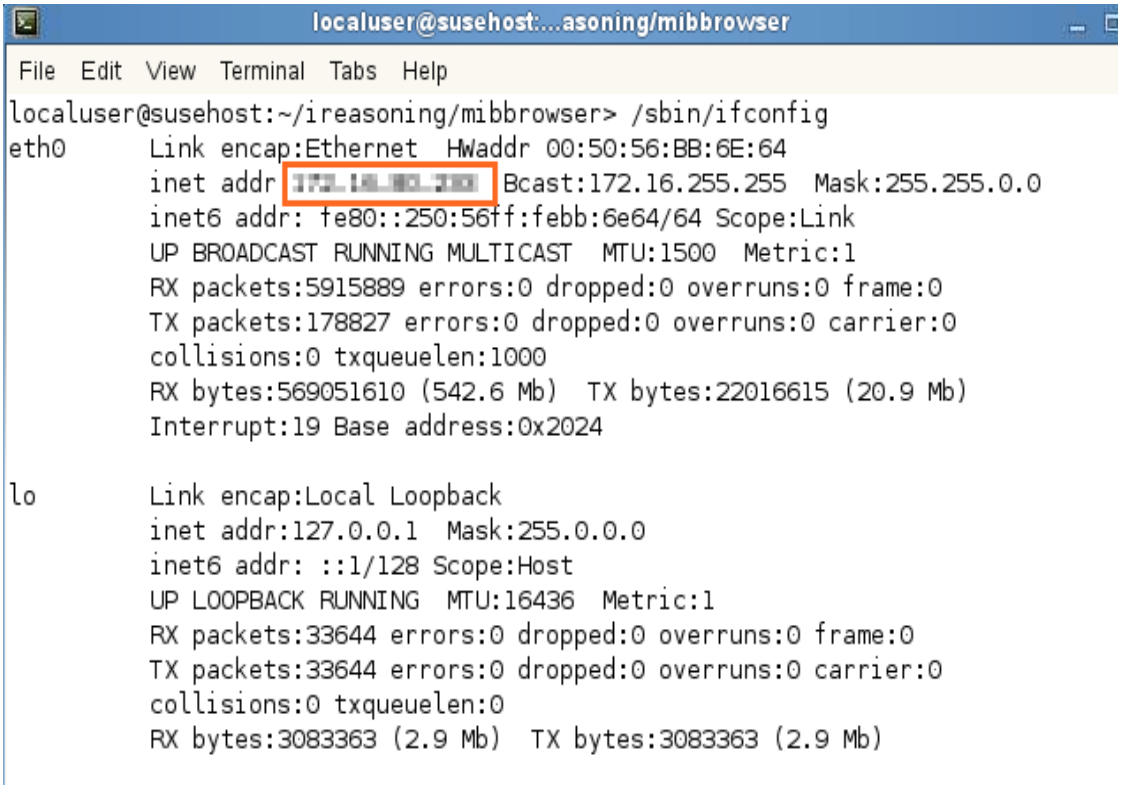
Verify that the values that you entered for the SoapUI variables are correct. If you still have problems, you must contact whatever support you have for the class.

__ 12. Close the **00 - Web Service Test - Booking** window.

Part 3: Identify the student image IP address

On Linux, you can discover the IP address by running the `ifconfig` command. Open a terminal window. The terminal window is available from the icon on the desktop. From within a terminal window, run the `ifconfig` command that includes the full path, as follows:

```
> /sbin/ifconfig
```



When the IP address of the local student image is obtained, update the information in table B1 for the variable *<image_ip>*.

Close the terminal window.

Part 4: Port and variable table values

If you want to have a single reference for all variables that are used in this course, the following table is supplied. You might want to tear these two pages out of your book, or if you have a PDF file, you can print both pages as a quick reference point.

__ 1. Complete the following table with the values that are supplied by the instructor.

Table B-1. Developers course variable and port assignments table

Object	Value (default)
Lab files location	
<*lab_files*> Location of student lab files for this course	/usr/labfiles/dp
Student information	
<*nn*>	
<student*nn*>	
<student*nn*_domain>	
<student*nn*_password>	student<*nn*>
<student*nn*_updated_password>	
<*image_ip*> IP address of the student image	
Logins that are not DataPower	
<*linux_user*>	localuser
<*linux_user_password*>	passw0rd
<*linux_root_user*>	root
<*linux_root_password*>	passw0rd
DataPower information	
<*dp_public_ip*> IP address of the public services on the appliance	
<*dp_internal_ip*> IP address of the DataPower appliance development and administrative functions	
<*dp_WebGUI_port*> Port number for the WebGUI	9090
<*dp_its_http_port*> Port for the HTTP interface to the Interoperability Test Service	9990
<*dp_FLY_baggage_port*>	2068
<*dp_FLY_booking_port*>	9080

Server information	
<SoapUI_keystores>	/usr/labfiles/dp/WSSecurity/Client.jks
<SoapUI_keystores_password>	myjkspw
<ldap_password>	passw0rd
<ldap_server_port> Port number for the LDAP administration console	9080
<ldap_server_root_dir>	/opt/ibm/ldap/V6.3/bin
<ldap_user_name>	cn=root
<http_server_port>	80
<http_server_root_dir>	/opt/IBM/HTTPServer/
<logger_app_port>	1112
Student-built DataPower services	
<mpgw_booking_port>	12nn1
<mpgw_booking_ssl_port>	12nn2
<mpgw_ssl_booking_port>	12nn3
<mpgw_mq_port>	12nn4
<wsp_booking_port>	12nn5
<mpgw_helloworld_port>	12nn7
<mpgw_patterns_port>	12nn8
<mpgw_baggage_port>	12nn9

End of appendix

IBM Training