

# Exercise 15

*Mediate a service interaction using a Ballerina mediation to convert from an inbound REST/JSON call into an existing SOAP/XML call.*

## Prior Knowledge

Basic understanding HTTP verbs, REST architecture, SOAP and XML

## Objectives

Understand Ballerina, and simple JSON to XML mapping.

## Software Requirements

(see separate document for installation of these)

- Java Development Kit 8
- Ballerina Tools 0.990.0
- Visual Studio Code
- Docker

## Overview

In this lab, we are going to take a WSDL/SOAP payment service, which is loosely modeled on a real SOAP API (Barclaycard SmartPay <https://www.barclaycard.co.uk/business/accepting-payments/website-payments/web-developer-resources/smartpay#tabbox1>).

Our aim is to convert this into a simpler HTTP/JSON interface. We probably won't get as far as any truly RESTful concepts as we won't have the opportunity to add resources, HATEOAS, etc. But will look at how those could be added with more time.

## PART A - Simple Ballerina Service

### 1. Unzip Ballerina:

```
cd Downloads
wget https://product-dist.ballerina.io/dev/0.990.0/ballerina-0.990.0.zip
cd ~
unzip ~/Downloads/ballerina-0.990.0.zip
```

### 2. Update the PATH.

```
code ~/.bashrc
```

Find the line that reads:

```
export PATH=$PATH:~/ballerina-0.983.0/bin
```

Change it to read:

```
export PATH=$PATH:~/ballerina-0.990.0/bin
```

### 3. Check ballerina is working:

```
$ ballerina version
Ballerina 0.990.0
```

### 4. Make a directory for your code:

```
mkdir ~/mediation
cd ~/mediation
```

5. Unfortunately we have an out-of-date vscode plugin. Hopefully by the time you do this lab, the marketplace will be updated!

### **TODO instructions to install 0.990 vsix**

6. Start vscode with a Ballerina file:

```
code mediate.bal
```

7. Save the file (there is a bug with the editor that means there is no code completion until you save it).
8. Let's start by creating a simple HTTP server first.
9. Type the following code.

```
import ballerina/http;

service mediate on new http:Listener(9090) {
    resource function hi(http:Caller caller, http:Request request) {
        _ = caller -> respond("hello world");
        return;
    }
}
```

10. Save it.

11. Start a command line in vscode (Ctrl-`)

```
cd mediation
ballerina build mediate.bal
```

You should see:

```
$ ballerina build mediate.bal
Compiling source
    mediate.bal
Generating executable
    mediate.balx
```

12. Now run it:

```
$ ballerina run mediate.balx
Initiating service(s) in 'mediate.balx'
[ballerina/http] started HTTP/WS endpoint 0.0.0.0:9090
```

13. curl or browse <http://localhost:9090/mediate/hi>

14. Let's evolve this service a little bit first before we tackle calling the SOAP backend.

15. Change the path the service is available at by adding the following annotation above the “service” stanza:

```
@http:ServiceConfig {  
    basePath: "/pay"  
}
```

16. Next modify the resource properties with the following annotation:

```
@http:ResourceConfig {  
    methods: [ "GET" ],  
    path: "/hello/{name}"  
}
```

We have defined a path parameter in the annotation, so we can also add this to the function definition (see italics)

```
resource function  
    hi(http:Caller caller, http:Request request, string name) {
```

17. We can use this “name” in our output.

```
_ = caller -> respond("hello " + untaint name);
```

The *untaint* is unusual. Ballerina assumes all data coming over the wire is “tainted” (i.e. it may have SQL, XML or other injection attacks embedded in it). Any function parameter can be marked as “secure” meaning that you cannot pass data without untainting it. In this case we are simply returning the data, so we won’t worry yet.

18. It will be helpful later on if we can deal with errors. So let’s also add this code to the resource signature:

```
resource function  
    hi(http:Caller caller, http:Request request, string name)  
        returns () | error
```

19. Ballerina has a “union” type system. This line means that the function can either return “nil” () or an error.

20. Your code should look like:

```
import ballerina/http;

@http:ServiceConfig {
    basePath: "/pay"
}
service mediate on new http:Listener(9090) {
    @http:ResourceConfig {
        methods: ["GET"],
        path: "/hello/{name}"
    }
    resource function hi(http:Caller caller, http:Request request, string name)
        returns () | error {
        _ = caller -> respond("hello " + untaint name);
        return;
    }
}
```

Rebuild and rerun.

21. Now the path you need to try should be

<http://localhost:9090/pay/hello/john>

```
$ curl -v http://localhost:9090/pay/hello/john
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 9090 (#0)
> GET /pay/hello/paul HTTP/1.1
> Host: localhost:9090
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/plain
< content-length: 10
< server: ballerina/0.990.0
< date: Sat, 8 Dec 2018 12:33:52 GMT
<
* Connection #0 to host localhost left intact
hello john
```

## PART B - SOAP "Ping"

22. Make sure you have the SOAP Payment service running from Exercise 14.

```
docker run -d -p 8888:8080 pizak/pay
```

23. We are also going to intercept all the messages between the ESB and the backend using mitmdump. In a new terminal window start:

```
mitmdump --listen-port 8080 --set flow_detail=3 --mode
reverse:http://localhost:8888
```

This puts the port back to 8080, but lets us see all the traffic to the backend.

24. Check that it is running:

Browse: <http://localhost:8080/pay/services/paymentSOAP?wsdl>

25. The SOAP service we are calling has two methods. The first is just a “ping/echo” that will return whatever string we send it. This is a useful test that the service is working. The second is the actual payment service, which takes various credit card details and then returns a response. We’ll deal with that in Part C.

There are HTTP level log of the ping service shown in Figure 1.

**Figure 1 - Sample “ping” SOAP exchange**

```
127.0.0.1:59537: POST http://localhost:8888/pay/services/paymentSOAP
Content-Type: application/xml
Action: http://freo.me/payment/ping
Host: localhost
User-Agent: ballerina/0.95.6
Transfer-Encoding: chunked
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:p="http://freo.me/payment/">
  <soap:Body>
    <p:ping>
      <p:in>hello</p:in>
    </p:ping>
  </soap:Body>
</soap:Envelope>

<< 200 186b
Server: Apache-Coyote/1.1
Content-Type: text/xml; charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Tue, 02 Jan 2018 11:56:19 GMT
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Body>
    <pingResponse xmlns="http://freo.me/payment/"
      <out>hello</out>
    </pingResponse>
  </soap:Body>
</soap:Envelope>
```

26. First we will guide through creating a JSON/HTTP proxy to the first method, and then adapting the second method will be more freeform.
27. There is a SOAP module for Ballerina, but it hasn’t yet been updated for 0.990.0 which was released this week. However, we don’t really need it because the XML support in Ballerina is so easy to use.
28. I’m giving you a skeleton code that is an extension of the previous code. We’ll call this **mediate-partB.bal**

In your mediation directory, type:

wget <https://freo.me/mediate-skel> -O mediate-partB.bal

29. Edit this file in vscode.

30. It should look like this (the new parts are in italics):

```
import ballerina/http;
import ballerina/io;

http:Client payclient = new("http://localhost:8080/");

@http:ServiceConfig {
    basePath: "/pay"
}
service mediate on new http:Listener(9090) {
    @http:ResourceConfig {
        methods: ["GET"],
        path: "/ping/{echo}"
    }
    resource function ping(http:Caller caller, http:Request request, string echo)
        returns () | error {

        xmlns "http://freo.me/payment/" as pay;

        // use the XML syntax to create the XML body tags
        //xml body =

        xmlns "http://schemas.xmlsoap.org/soap/envelope/" as s;
        xml soap = xml `
```

31. Let's look at the new stuff (and improve it where I have left code out).

32. First, we are going to be calling out to an HTTP service, so we need a client:

```
http:Client payclient = new("http://localhost:8080/");
```

33. Ballerina has built in support for XML types. We are going to use this to craft the XML message to send to the SOAP backend.

```
// use the XML syntax to create the XML body tags
//xml body =
```

First look below at the syntax I've used for the SOAP envelope and body.

```
xmlns "http://schemas.xmlsoap.org/soap/envelope/" as s;
xml soap = xml `<s:Envelope><s:Body>{{body}}</s:Body></s:Envelope>`;
```

34. You can find out more here:

<https://ballerina.io/learn/by-example/xml-literal.html>  
<https://ballerina.io/learn/by-example/xml-namespaces.html>

35. Use SOAPUI to find the XML Body that we need to send. Using the similar syntax to the SOAP part, add it into the code.

36. You need to add the "echo" string parameter into the XML message within the body. The following code hints at how you can do this:

```
// "Templating" - anything in {{ }} will be evaluated as a
// Ballerina expression at runtime.
string data;
xml x = xml `<tag>{{data}}</tag>`;
```

37. We need to put together an http:Request object. This is because we need to set the SOAPAction header. Otherwise we could just post the XML directly.

```
http:Request soapReq = new;
soapReq.addHeader("SOAPAction", "http://freo.me/payment/ping");
soapReq.setPayload(soap);
```



38. Next we actually send the message. That is up to you. Its pretty simple (you can fit it into a single line of code).

Look here for more information:

<https://ballerina.io/learn/by-example/check.html>

<https://ballerina.io/learn/by-example/http-client-endpoint.html>

39. We then untaint the XML payload of the HTTP response, and grab the “body” of the message. We are going to make this really simple by turning it into JSON (JSON is another built-in type in Ballerina).

```
json jsonResp = untaint check
    payResp.getXmlPayload().!toJSON({preserveNamespaces: false});
json jsBody = jsonResp.Envelope.Body;
```

40. You may be wondering what the ! is. It is called “Error Lifting”

<https://ballerina.io/learn/by-example/error-lifting.html>

41. You now need to extract the actual text from the JSON object.

42. Finally we put together a new JSON object with the data we want to send back to the client.


```
json js = {
    pingResponse: responseText
};
```

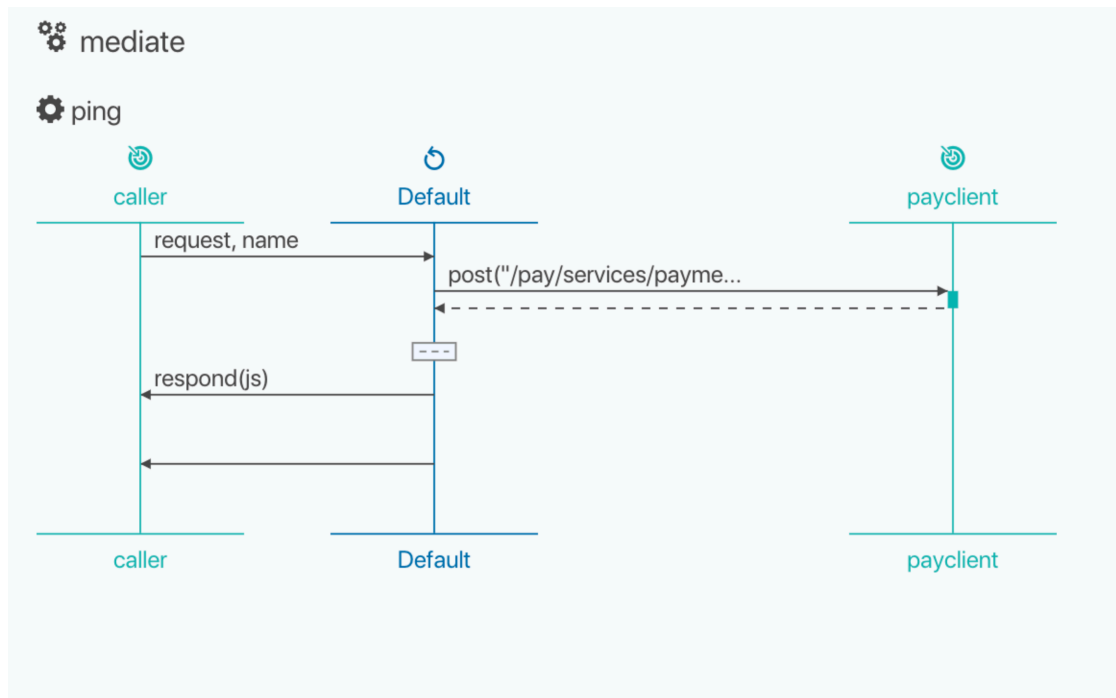
43. That should be everything. Make sure it compiles and then run it.

44. Try calling <http://localhost:9090/pay/ping/test>

45. Check that the message is really being sent to the backend. You can do this by looking at MITMDUMP.

46. Ballerina is a language specifically designed for orchestration of services, modelled on sequence diagrams. You can design flows as sequence diagrams, but I prefer to use the programming language. We will, however, try some examples of the graphical tooling as well. The text editing and graphical editing are completely bidirectional (i.e. you can edit the text, then the graphics, then the text as much as you like and both

stay in sync). Click on the icon:  and you should see the following sequence diagram that is generated from the code.



## PART C - ACTUAL CARD PAYMENT

47. Now we need to take an incoming JSON in a POST, parse it, and then convert it into a more complex XML.

48. Here is the XML interaction we need to talk to the SOAP service (Figure 2)

**Figure 2 - Sample Authorize SOAP exchange**

```
127.0.0.1:60975: POST http://localhost:8888/pay/services/paymentSOAP
Content-Type: application/xml
Action: http://freo.me/payment/authorise
Host: localhost
User-Agent: ballerina/0.95.6
Transfer-Encoding: chunked
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:p="http://freo.me/payment/">
  <soap:Body>
    <p:authorise>
      <p:card>
        <p:cardnumber>4544950403888999</p:cardnumber>
        <p:postcode>P0107XA</p:postcode>
        <p:name>P Z FREMANTLE</p:name>
        <p:expiryMonth>6</p:expiryMonth>
        <p:expiryYear>2017</p:expiryYear>
        <p:cvc>999</p:cvc>
      </p:card>
      <p:merchant>A0001</p:merchant>
      <p:reference>test</p:reference>
      <p:amount>11.11</p:amount>
    </p:authorise>
  </soap:Body>
</soap:Envelope>

<< 200 343b
Server: Apache-Coyote/1.1
Content-Type: text/xml; charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Tue, 02 Jan 2018 16:20:14 GMT
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <authoriseResponse xmlns="http://freo.me/payment/">
      <authcode>FAILED</authcode>
      <reference>8f8371de-af96-4032-b332-3641d84f050c</reference>
      <resultcode>100</resultcode>
      <refusalreason>INSUFFICIENT FUNDS</refusalreason>
    </authoriseResponse>
  </soap:Body>
</soap:Envelope>
```

49. I want you to parse an incoming JSON:

```
{
  "cardNumber": "4544950403888999",
  "postcode": "P0107XA",
  "name": "P Z FREMANTLE",
  "month": 6,
  "year": 2017,
  "cvc": 999,
  "merchant": "A0001",
  "reference": "test",
  "amount": 11.11
}
```

50. You should know almost enough to complete this but there are a couple of things you will need to know extra.
51. Firstly, obviously you need to copy the resource definition for ping and convert the new version to be authorise. I'd like the path to be /pay/authorise
52. You will need to support POST not GET
53. We won't be passing any path parameters, so adjust the function signature appropriately.
54. The JSON body of the POST message can be accessed using:
- ```
request.getJsonPayload()
```
55. One further aspect which will help, which is that we can define a "record" in Ballerina that matches the JSON, and parse the JSON into there. This will validate the JSON as it does this.

The structure definition matching the JSON is below and here:

<https://freo.me/pay-struct>

```
type payment record {  
    string cardNumber;  
    string postcode;  
    string name;  
    int month;  
    int year;  
    int cvc;  
    string merchant;  
    string reference;  
    float amount;  
};
```

You can also read about Records here: <https://ballerina.io/learn/by-example/records.html>

56. Copy and paste this and put it at the same level as the service in the .bal file (so it is globally defined for this ballerina program).

57. You can parse the incoming JSON into the **payment** record by using this code:

```
payment p = payment.convert(json);
```

58. You can read about JSON <-> Map <-> Record conversion here:

<https://ballerina.io/learn/by-example/json-record-map-conversion.html>

59. Don't forget untainting and checking for errors.

60. You need to construct an XML that looks like this. You can cut and paste this from SOAPUI.

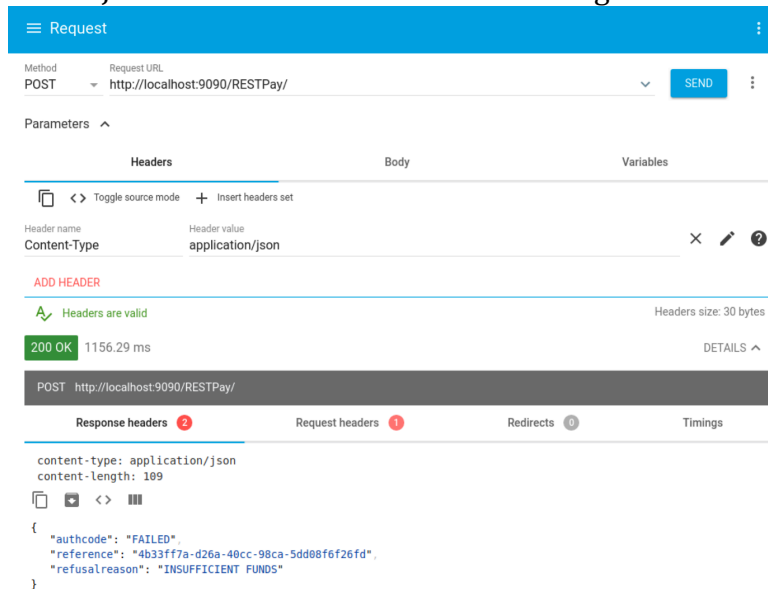
```
<pay:authorise>
  <pay:card>
    <pay:cardnumber>?</pay:cardnumber>
    <pay:postcode>?</pay:postcode>
    <pay:name>?</pay:name>
    <pay:expiryMonth>?</pay:expiryMonth>
    <pay:expiryYear>?</pay:expiryYear>
    <pay:cvc>?</pay:cvc>
  </pay:card>
  <pay:merchant>?</pay:merchant>
  <pay:reference>?</pay:reference>
  <pay:amount>?</pay:amount>
</pay:authorise>
```

61. Finally, I want you to return a JSON that looks like:

```
{
  "authcode": "FAILED",
  "reference": "b23f8aad-766d-46c9-98c2-f328ce8ed594",
  "refusalreason": "INSUFFICIENT FUNDS"
}
or
{
  "authcode": "AUTH0234",
  "reference": "07b82cad-cb55-428d-b02c-c5619bbbed4d",
  "refusalreason": "OK"
}
```

62. Use the previous code as a template and get the payment JSON to XML conversion working.

63. Use Advanced REST client to create a POST request to test this out, based on the JSON above. I'd like to see something like:



64. If you get stuck, my version is available here: <https://freo.me/mediate-final>

#### 65. Extension:

Let's use ballerina to build this into a Docker container. Take a look at:

<https://freo.me/mediate-docker>

Here is a version of the mediate.bal that has Docker extensions built in. Note that I've separated out the listener so that it can be annotated.

I've also updated the code to look in an environment variable (SOAP\_ENDPOINT) for the url of the backend SOAP service.

Copy this file locally, modify the docker image name so that it matches your user and build it.

If you run this, it won't be able to find the dockerised SOAP service. See if you can run them both so they can find each other. Hints: you'll need to set the environment variables and configure the network hostnames so the Ballerina docker can find the SOAP docker.

66. **Extension 2:** Create a docker-compose file to run the Ballerina microservice and payment SOAP microservice (without mitmdump), and correct the Ballerina program so it refers to the payment SOAP endpoint and demonstrate the whole thing successfully running with docker compose.