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 1.0.3
- Visual Studio Code
- Docker

Pre-requisites
SOAP service running in Docker from Exercise 14

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. Make a directory for your code:

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

2. Start vscode with a Ballerina file:

```
code mediator.bal
```

3. Save the file (there is a bug with the editor that means there is no code completion until you save it).

4. Let's start by creating a simple HTTP server first.

5. Type the following code.

```
import ballerina/http;

service mediate on new http:Listener(8080) {
    resource function hi(http:Caller caller, http:Request request) {
        var err = caller -> respond("hello world");
    }
}
```

6. Save it.

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

```
cd ~/mediation
ballerina build mediator.bal
```

You should see:

```
oxsoa@oxsoa:~/mediation$ ballerina build mediator.bal
Compiling source
        mediator.bal

Generating executables
        mediator.jar
```

8. Now run it:

```
$ ballerina run mediator.jar
[ballerina/http] started HTTP/WS listener 0.0.0.0:8080
```

9. curl or browse http://localhost:8080/mediate/hi

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

11. Change the path the service is available at by adding the following annotation above the "service" stanza:

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

12. Next modify the resource properties with the following annotation, which

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

needs to go just above the **resource** definition :
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) {
```

13. We can use this "name" in our output.

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

14. 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?
```

15. Ballerina has a "union" type system. This line means that the function can return either "nil" () or an error. This is equivalent to `returns error|()`

16. Your code should look like:

```
import ballerina/http;

@http:ServiceConfig {
    basePath: "/pay"
}
service mediate on new http:Listener(8080) {
    @http:ResourceConfig {
        path: "/hello/{name}",
        methods: ["GET"]
    }
    resource function hi(http:Caller caller, http:Request request, string name) returns error? {
        check caller -> respond("hello "+name);
    }
}
```

You don't need to build before run
You can simply run the .bal file:

```
$ ballerina run mediator.bal
Compiling source
        mediator.bal

Generating executables
Running executables

[ballerina/http] started HTTP/WS listener
0.0.0.0:8080
```

17. Now the path you need to try should be http://localhost:8080/pay/hello/paul

```
~/mediate$ http -v localhost:8080/pay/hello/paul
GET /pay/hello/paul HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/1.0.3


HTTP/1.1 200 OK
content-encoding: gzip
content-length: 36
content-type: text/plain
date: Tue, 19 Nov 2019 11:39:31 GMT
server: ballerina/1.0.3

hello paul
```

PART B - SOAP "Ping"

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

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

19. 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 8000 --set flow_detail=3 --mode
    reverse:http://localhost:8888
```

    This puts the port back to 8000, but lets us see all the traffic to the backend.

20. Check that it is running:
    Browse: http://localhost:8000/pay/services/paymentSOAP?wsdl

21. 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.

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>
```

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

22. First we will guide through creating a JSON/HTTP proxy to the first method, and then adapting the second method will be more freeform.

23. Now let's define a **record** type of message to consume as the JSON input:

```
type Ping record {
    string name;
};
```

24. Your code should look like:

```
import ballerina/http;
import wso2/soap;

type Ping record {
    string name;
};

@http:ServiceConfig {
…
```

25. We are going to change our HTTP service to take a JSON POST message.

    Edit your resource definition to read:

```
@http:ResourceConfig {
        path: "/ping",
        methods: ["POST"],
        body: "ping"
    }
    resource function ping(http:Caller caller, http:Request request, Ping ping)
        returns error? {
```

26. Let's test that we can successfully read the data from the body and respond. We want a JSON response as well.

    Modify the function body to look like the bold part:

```
resource function ping(http:Caller caller, http:Request request, Ping ping)
        returns error? {
        json response = {
            pingResponse: ping.name
        };
        check caller -> respond(response);
}
```

27. Run this and test it:

```
http -v localhost:8080/pay/ping/ name="paul"
POST /pay/ping/ HTTP/1.1
Accept: application/json, */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 16
Content-Type: application/json
Host: localhost:8080
User-Agent: HTTPie/1.0.3

{
    "name": "paul"
}

HTTP/1.1 200 OK
content-encoding: gzip
content-length: 42
content-type: application/json
date: Tue, 19 Nov 2019 12:08:47 GMT
server: ballerina/1.0.3

{
    "pingResponse": "paul"
}
```

28. Now we just need to call the backend with some XML and then parse the result.

    We will use the SOAP module for Ballerina to help us.

    On a command-line type:

```
$ ballerina pull wso2/soap
wso2/soap:0.4.1 [central.ballerina.io -> home repo]
[==========>] 10600/10600
```

29. Add the import to the top of your code:

```
import wso2/soap;
```

30. Let's define a SOAP client object to call the backend. Below the import but above the Service definition add:

```
soap:Soap11Client backend =
new("http://localhost:8000/pay/services/paymentSOAP");
```

31. 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.
You can find out more here:
https://ballerina.io/learn/by-example/xml-literal.html
https://ballerina.io/learn/by-example/xml-namespaces.html

32. We can use the XML from SOAPUI to send the right information.

In your resource function body, add the following:

```
xmlns "http://freo.me/payment/" as pay;
xml body = xml `<pay:ping>
                    <pay:in>${ping.name}</pay:in>
                </pay:ping>`;
```

What I've done here is to copy the XML from SOAP UI. I've also extracted the namespace and defined it properly. Finally I've used a *template* to insert the data from the incoming JSON message into it.

Now we need to know the SOAP Action to send with the message. This is in the WSDL:

```
<wsdl:operation name="ping">
<soap:operation soapAction="http://freo.me/payment/ping"/>
```

We can now send this message to the SOAP backend:

```
var soapResponse = check backend ->
      sendReceive(body, "http://freo.me/payment/ping");
```

33. Your code should look like:

```
import ballerina/http;
import wso2/soap;

soap:Soap11Client backend = new("http://localhost:8000/pay/services/paymentSOAP");

type Ping record {
    string name;
};

@http:ServiceConfig {
    basePath: "/pay"
}
service mediator on new http:Listener(8080) {
    @http:ResourceConfig {
        path: "/ping",
        methods: ["POST"],
        body: "ping"
    }
    resource function ping(http:Caller caller, http:Request request, Ping ping)
        returns error? {
        xmlns "http://freo.me/payment/" as pay;
        xml body = xml `<pay:ping>
                <pay:in>${ping.name}</pay:in>
            </pay:ping>`;

        var soapResponse = check backend ->
                sendReceive("http://freo.me/payment/ping", body);
        json response = {
            pingResponse: ping.name
        };
        check caller -> respond(response);
    }
}
```

We now need to extract the data from the response body. This is fairly simple, but Ballerina does force us to deal with the case where there is no payload returned, because the strong typing includes that as an option.

You can see this in the return type of the soapResponse object (see https://github.com/wso2-ballerina/module-soap/blob/master/src/soap/soap_types.bal).

Don't type this code in! Its just for reference!

```
public type SoapResponse record {|
    SoapVersion soapVersion = SOAP11;
    xml[] headers?;
    xml payload?;
    http:Response httpResponse;
|};
```

As you can see the payload is either xml or missing, since there is a question mark.

To handle this, we can use "optional field access" (see https://ballerina.io/learn/by-example/optional-field-access.html)

Here is the code to send an error back if the payload is empty.

```
xml? payload =  soapResponse?.payload;
if (payload is ()) {
     http:Response err = new;
     err.statusCode = 400;
     check caller ->respond(err);
} else
{
     // extract the data from the XML data
```

At this point we can see the value of a Type Guard in Ballerina (https://ballerina.io/learn/by-example/type-guard.html)

Once we are in the "else" part of the code, the type system knows that the payload variable must be an XML object, and we can now use XML functions and access without worrying about nils.

To access the data from the XML, we can use XML Access code (https://ballerina.io/learn/by-example/xml-access.html):

```
string data = payload[pay:out].getTextValue();
```

34. We need to handle something called tainting at this point. Ballerina assumes 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 marker as "secure" meaning that you cannot pass data without untainting it. If we try to use this data without validating that we have untainted it we will get an error.

    Let's see that. Change the json response to return the "data" element:

```
json response = {
    pingResponse: data
};
```

35. Your resource function code should look like this:

```
resource function ping(http:Caller caller, http:Request request, Ping ping) returns error? {
    xmlns "http://freo.me/payment/" as pay;
    xml body = xml `<pay:ping>
            <pay:in>${ping.name}</pay:in>
        </pay:ping>`;

    var soapResponse = check backend->sendReceive("http://freo.me/payment/ping", body);
    xml? payload = soapResponse?.payload;
    if (payload is ()) {
        http:Response err = new;
        err.statusCode = 400;
        check caller->respond(err);
    } else
    {
        string data = payload[pay:out].getTextValue();
        json response = {
            pingResponse: data
        };
        check caller->respond(response);
    }
}
```

36. Try to compile this. You should see an error:

```
~/mediate$ ballerina build mediator.bal
Compiling source
        mediator.bal
error: .::mediator.bal:37:35: tainted value passed
to untainted parameter 'message'
```

37. Since we actually trust the backend service we can simply say that the data is "untainted":

```
    pingResponse: <@untainted>data
```

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

39. Try calling http://localhost:8080/pay/ping/ with a JSON message containing name="paul". e.g:

```
http -v localhost:8080/pay/ping/ name="paul"
POST /pay/ping/ HTTP/1.1
Accept: application/json, */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 16
Content-Type: application/json
Host: localhost:8080
User-Agent: HTTPie/1.0.3

{
    "name": "paul"
}

HTTP/1.1 200 OK
content-encoding: gzip
content-length: 49
content-type: application/json
date: Tue, 19 Nov 2019 13:54:38 GMT
server: ballerina/1.0.3

{
    "pingResponse": "paul"
}
```

40. Check that the message is really being sent to the backend. You can do this by looking at MITMDUMP. You should see:

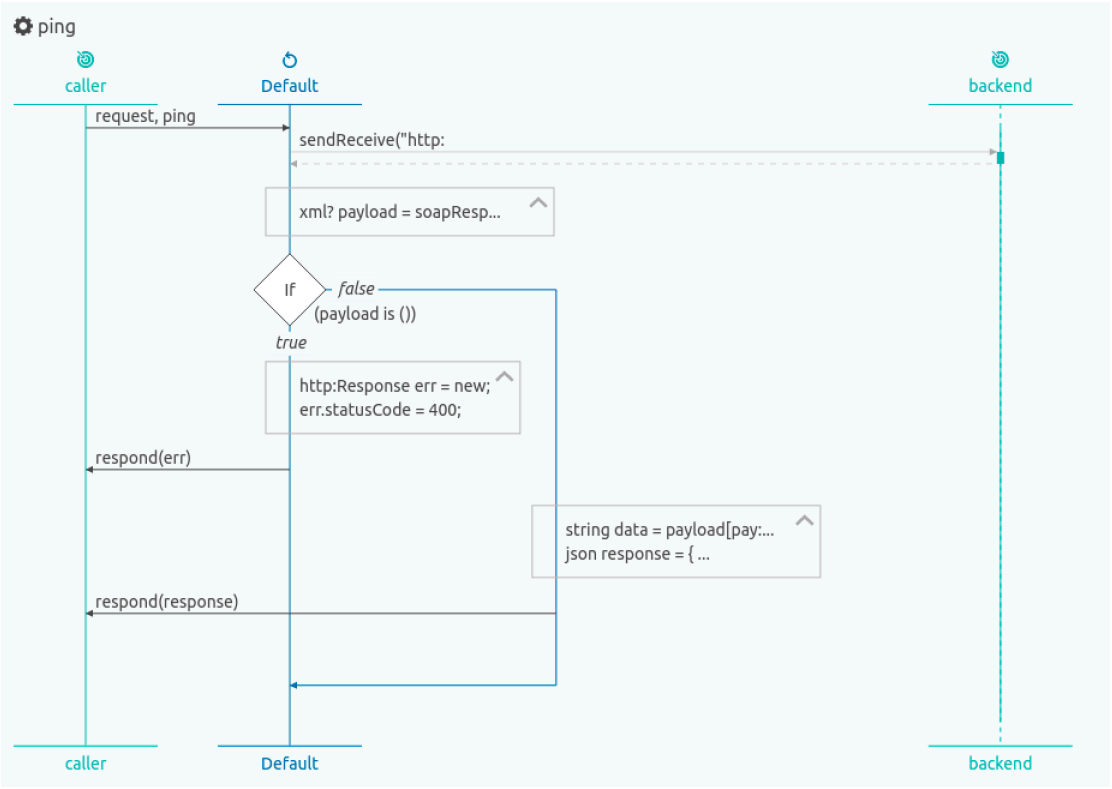
```
127.0.0.1:61492: POST
    content-type: text/xml
    SOAPAction: http://freo.me/payment/ping
    host: localhost
    user-agent: ballerina/1.0.3
    connection: keep-alive
    content-length: 294

    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <soap:Header/>
      <soap:Body>
        <pay:ping xmlns:pay="http://freo.me/payment/">
          <pay:in>paul</pay:in>
        </pay:ping>
      </soap:Body>
    </soap:Envelope>

    200
    Server: Apache-Coyote/1.1
    Content-Type: text/xml;charset=ISO-8859-1
    Transfer-Encoding: chunked
    Date: Tue, 19 Nov 2019 13:54:38 GMT

    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
        <pingResponse xmlns="http://freo.me/payment/">
          <out>paul</out>
        </pingResponse>
      </soap:Body>
    </soap:Envelope>
```

41. Ballerina is a language specifically designed for orchestration of services, modelled on sequence diagrams. Click on the icon:  and you should see the following sequence diagram that is generated from the code.



PART C - ACTUAL CARD PAYMENT

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

43. 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>PO107XA</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>
```

44. I want you to parse an incoming JSON, e.g.:

```
{
  "cardNumber": "4544950403888999",
  "postcode": "PO107XA",
  "name": "P Z FREMANTLE",
  "month": 6,
  "year": 2017,
  "cvc": 999,
  "merchant": "A0001",
  "reference": "test",
  "amount": 11.11
}
```

45. You should know almost enough to complete this but there are a couple of things you will need to know extra.

46. Firstly, obviously you need to copy and paste the resource definition for **ping** and convert the new version to be **authorise**. I'd like the path to be /pay/authorise

47. Here is a record definition for the incoming JSON that you can use:

    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

48. 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).

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

50. 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>
```

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

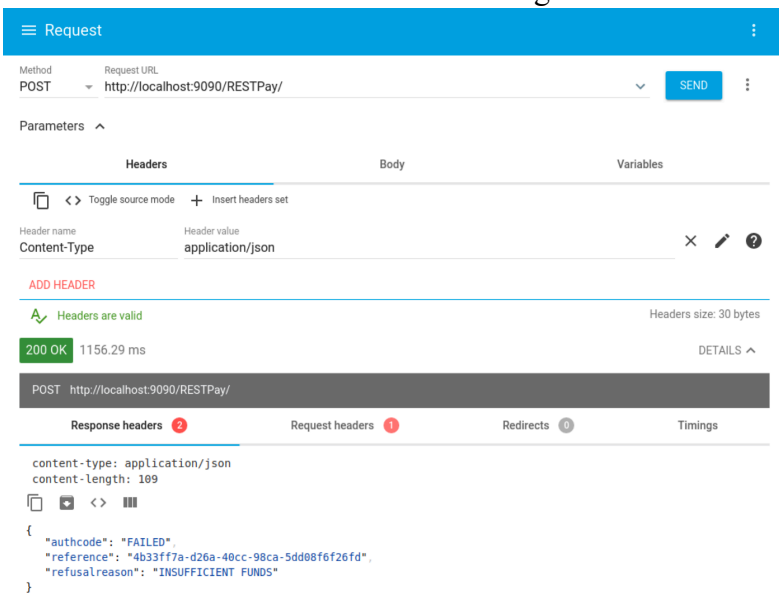
```
{
"authcode": "FAILED",
"reference": "b23f8aad-766d-46c9-98c2-f328ce8ed594",
"refusalreason": "INSUFFICIENT FUNDS"
}
```

or

```
{
"authcode": "AUTH0234",
"reference": "07b82cad-cb55-428d-b02c-c5619bbbed4d",
"refusalreason": "OK"
}
```

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

53. 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:



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

55. **Extension**:

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

    https://freo.me/mediator-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.

56. **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.

57. **Extension 3:** Make a proper Ballerina project and module:
In a new directory:

```
ballerina new mediation
cd mediation
ballerina add soap2rest
```

copy your code into the right place.
Delete main.bal

See if you can add an effective test for your service
Build in the mediation directory using:
`ballerina build -a`