

Exercise 10

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, Composer and simple JSON to XML mapping.

Software Requirements

(see separate document for installation of these)

- Java Development Kit 8
- Ballerina Tools 0.95.6
- 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.

1. Before we start working with Ballerina, we need a SOAP/XML service to interact with. This is going to be run in Docker. You can start the service like this:

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

This offers the docker based service (which is a WAR file running in Tomcat) at port 8888 (mapped from the original 8080 that the Docker image offers).

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

```
mitmdump --port 8080 -d -d -d --reverse http://localhost:8888
```

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

3. Check that it is running:
Browse: <http://localhost:8080/pay/services/paymentSOAP?wsdl>
4. Unzip Ballerina 0.95.8 into ~/ballerina-tools-0.95.8

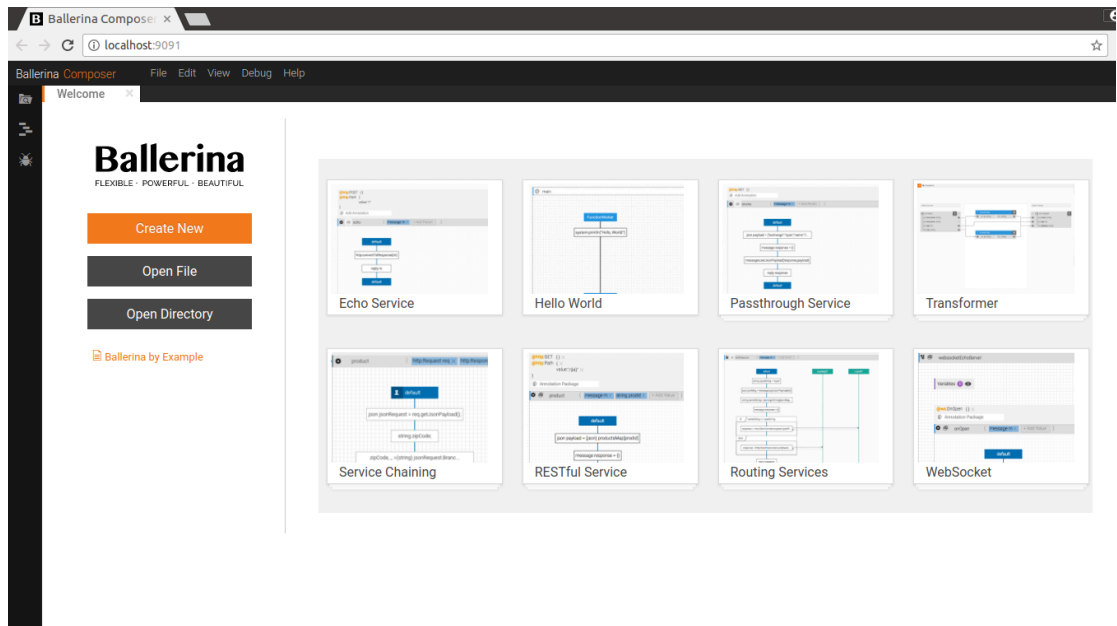
5. Make a directory for your code:

```
mkdir ~/ex10
```

6. Now start the Ballerina composer:

```
cd ~/ballerina-tools-0.95.8  
bin/composer
```

7. You should see something like this pop up in a browser:



8. Ballerina is a language specifically designed for orchestration of services, modeled 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).
9. The SOAP service we are calling has two methods. The first is just an “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. There are HTTP level logs of both interactions shown below in Figures 1 and 2.

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>

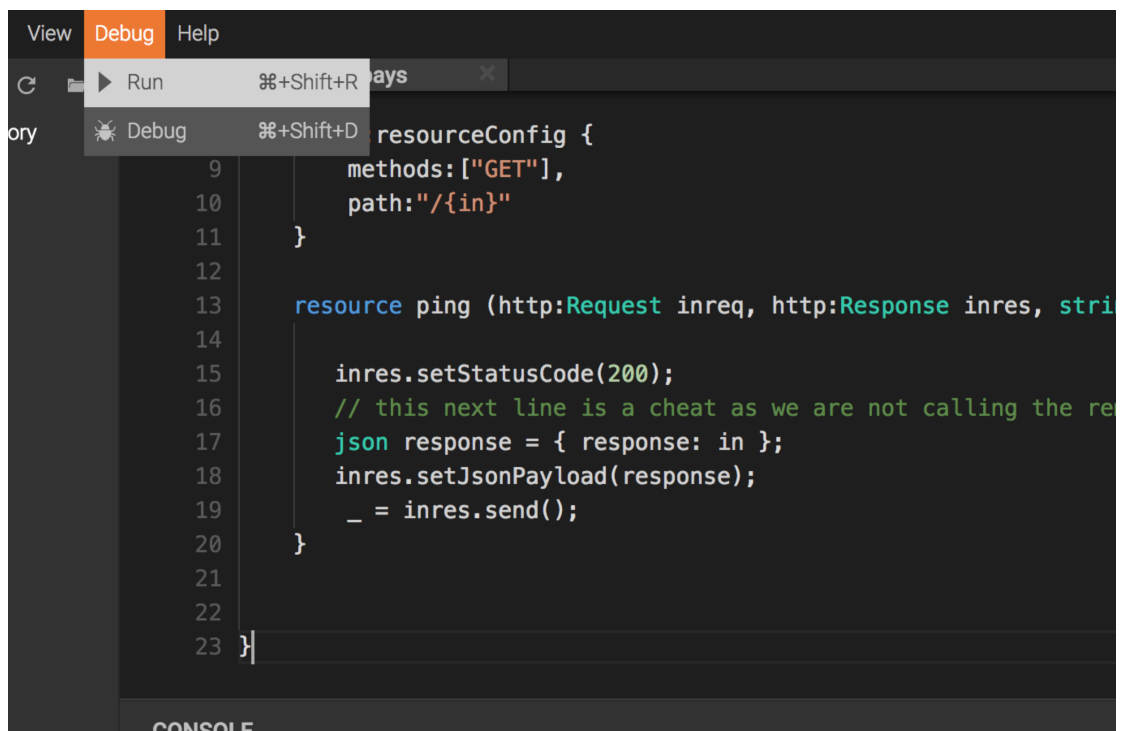
<< 200 186b
Server: Apache-Coyote/1.1
Content-Type: text/xml; charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Tue, 02 Jan 2018 11:56:19 GMT
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <pingResponse xmlns="http://freo.me/payment/">
      <out>hello</out>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

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>
```

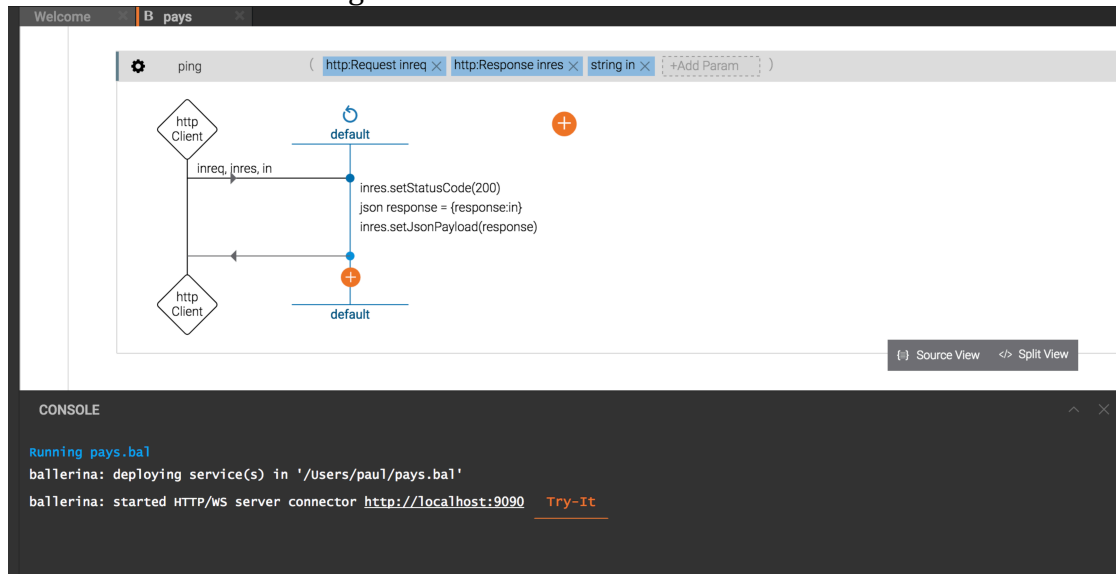
10. You can, of course, use SOAPUI to try these out yourself.
11. First we will guide through creating a JSON/HTTP proxy to the first method, and then adapting the second method will be more freeform.
12. Click on **Create New** in the composer. Now click on Source View.
13. There is a “skeleton” ballerina program at <https://freo.me/skel-bal>
Copy and paste this into the screen and then save this as something (like ~/ex10/pay.bal).
14. You can run this already, by choosing **Debug -> Run** from the menu.



```
View  Debug  Help
Run  ⌘+Shift+R  Run  ⌘+Shift+D
Debug  ⌘+Shift+D

resourceConfig {
  9   methods: ["GET"],
  10  path: "{in}"
  11 }
  12
  13 resource ping (http:Request inreq, http:Response inres, string inpath) {
  14
  15   inres.setStatusCode(200);
  16   // this next line is a cheat as we are not calling the resource
  17   json response = { response: in };
  18   inres.setJsonPayload(response);
  19   _ = inres.send();
  20 }
  21
  22
  23 }
```

15. You should see something like this:

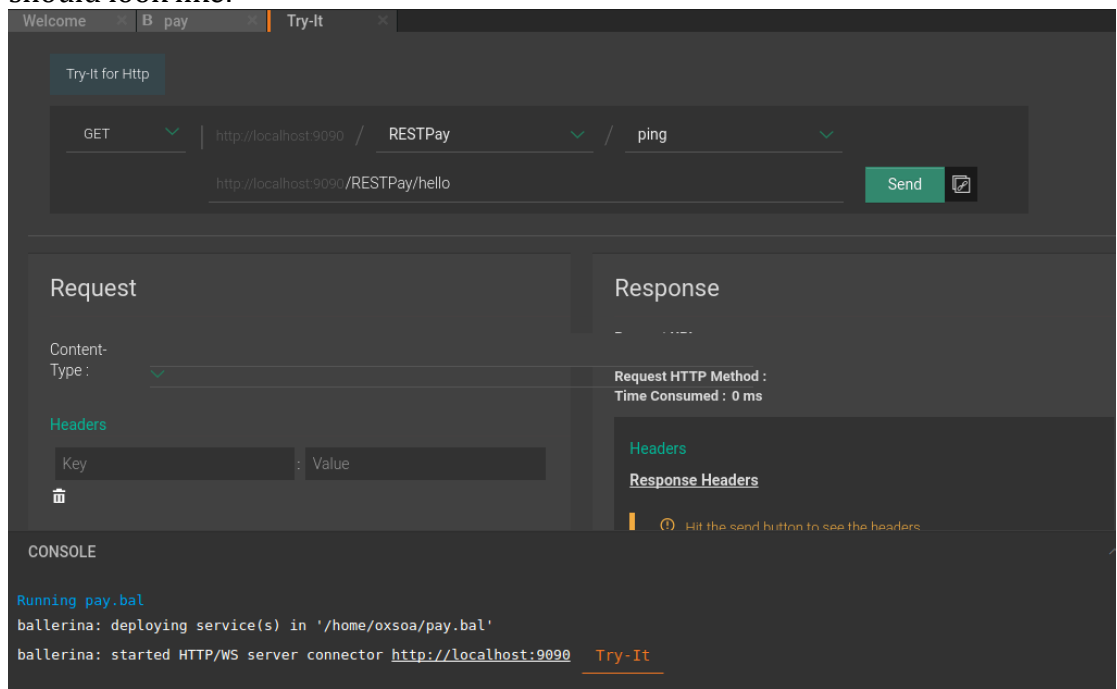


16. We'll look at the code in a moment, but let's try it out first.

17. Click on Try-It

18. Click on **Select Service**, and choose **RESTPay**, and then on **Select Resource** and choose **ping**.

19. It will create a URL below. Replace the **{in}** with "hello". Your screen should look like:



20. Now click **Send**.

21. There seems to be a slight rendering bug in this version, so you can't see all the response. You should see the response below:

The screenshot displays a web interface for viewing HTTP details. At the top, it shows 'Request HTTP Method : GET' and 'Time Consumed : 26 ms'. Below this, there's a section for 'Headers' which is further divided into 'Response Headers' and 'Request Headers'. The 'Response Headers' section lists 'Content-Type : application/json' and 'transfer-encoding : chunked'. The 'Request Headers' section lists 'Content-Type : text/plain', 'Host : localhost:9090', 'Connection : Keep-Alive', 'User-Agent : Ballerina Composer', and 'Accept-Encoding : gzip,deflate'. At the bottom, there's a 'Body' section showing a JSON response: `{"response": "hello"}`.

You might need to run this a couple of times to get a reasonable response time!

22. Close the Try-It tab, and turn back to the source view and we can now try to understand what is happening here.

23. Let's look at the code line by line:

```
import ballerina.net.http;

service<http> RESTPay {
    endpoint<http:HttpClient> soapPayService {
        create http:HttpClient("http://localhost:8080", {});
    }

    @http:resourceConfig {
        methods:["GET"],
        path:"/{in}"
    }
    resource ping (http:Request inreq, http:Response inres, string in) {

        inres.setStatusCode(200);
        // this next line is a cheat as we are not calling
        // the remote soap service to access this

        json response = { response: in };
        inres.setJsonPayload(response);
        _ = inres.send();
    }
}
```

24. The first line imports Ballerina's support for HTTP, since we are both offering and calling HTTP endpoints. The **service<http>** line is like defining a class in Java, except we are explicitly defining a service.
25. The next 3 lines define a remote HTTP endpoint (which is not used yet), which we are calling **soapPayService**. This endpoint will call HTTP on <http://localhost:8080> (where the mitmdump/docker service live).
26. The next three lines annotate the following resource with the method (GET) and path (`/{in}`). This is similar to the JAXWS annotations in Java.
27. The resource is now defined. Any parameters from the definition are automatically defined as input parameters (e.g. **in**).
28. It also takes a `inreq` and `inresp` as parameters (rather than returning a response).
29. We set the response status code to be 200.
30. Ballerina has native support for JSON, so the next line creates a JSON using the input parameter **in**.
31. We set that as the response JSON, which will automatically set the content type correctly.
32. Finally the syntax `_ =` explicitly catches and disregards the error return code from the response **send**.

33. The code has not yet called the remote SOAP service. Ballerina doesn't natively support SOAP, but it has excellent built-in support for XML.
34. Instead of simply returning a new JSON created with **in**, we instead want to:
- Create an XML message (body) to send to the backend SOAP endpoint, which encapsulates the **in** value.
 - Wrap that in a SOAP Envelope/SOAP Body.
 - Send that to the backend.
 - Parse the resulting XML to get the response.
 - Wrap that response in JSON and send that back.
35. Here is the code for each part. Start adding this code before the **setStatusCode** line.

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

36. The first line defines an XML namespace (which is required by our specific SOAP Body). The second line uses *inline XML* (indicated by **xml ``**) to create an XML element and the **{{}}** indicates not to treat the **in** as a literal, but as an expression.

37. The next two lines create the SOAP wrapper:

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

38. We now need to create an empty request and then populate it to send to the SOAP backend:

```
http:Request outreq = {};  
outreq.setXmlPayload(env);
```

39. The SOAP service also expects an HTTP header with the SOAP Action:

```
outreq.addHeader("Action", "http://freo.me/payment/ping");
```

40. We can now send that request to the backend, which we defined above in the skeleton:

```
var outresp, _ =  
    soapPayService.post("/pay/services/paymentSOAP",  
        outreq);
```

41. Once again, we are ignoring any errors from the SOAP service, by using the `_` syntax to grab them and throw them away.

If you wanted to look at any errors, you could do:

```
var outresp, error = ...  
println(error);
```

42. Ballerina has got some useful XML processing to access the XML result, but it isn't yet as powerful as the JSON processing. As a result, it's actually easier to convert the XML SOAP message that came back into JSON and then parse it:

```
var jr = outresp.getXmlPayload().  
    toJSON({preserveNamespaces:false});  
var responseText = jr.Envelope.Body.pingResponse.out;
```

43. This gets the XML payload, and converts it to JSON, throwing away the namespace information (which we don't really need). Then we simply grab the text element that is at `/Envelope/Body/pingResponse`.
44. Now we can create the Response JSON as before (replacing the previous equivalent line):

```
json response = {return: responseText };
```

45. Your code should now look like:

```
import ballerina.net.http;

service<http> RESTPay {
    endpoint<http:HttpClient> soapPayService {
        create http:HttpClient("http://localhost:8080", {});
    }

    @http:resourceConfig {
        methods: ["GET"],
        path:("/{in}"
    }

    resource ping (http:Request inreq, http:Response inres, string in) {

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

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

        http:Request outreq = {};
        outreq.setXmlPayload(env);

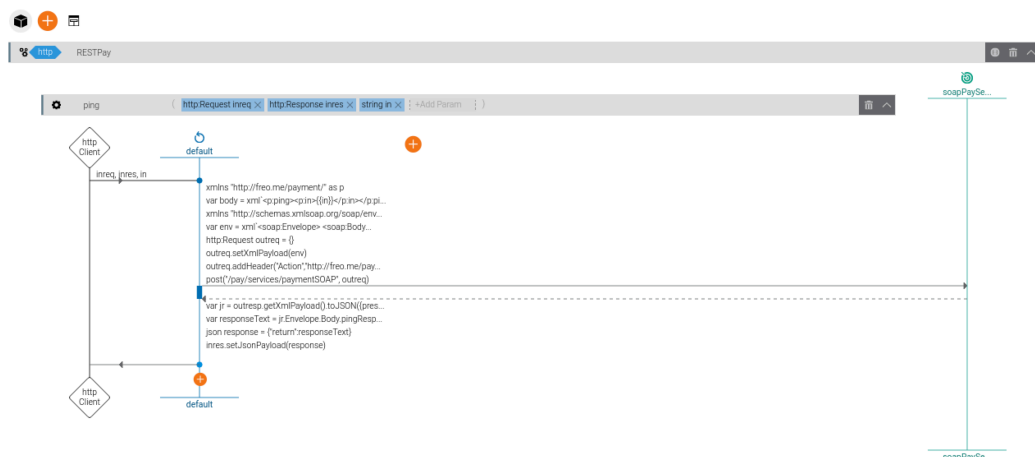
        outreq.addHeader("Action", "http://freo.me/payment/ping");

        var outresp, _ =
            soapPayService.post("/pay/services/paymentSOAP", outreq);
        var jr =
            outresp.getXmlPayload().toJSON({preserveNamespaces:false});

        var responseText = jr.Envelope.Body.pingResponse.out;
        json response = {return: responseText };

        inres.setJsonPayload(response);
        _ = inres.send();
    }
}
```

46. You can also view this as an interaction diagram, by clicking on Design View. Here is the overall view, scaled to fit on one page:



47. Try It as before. This time check and validate that the message is passing correctly from the Ballerina runtime to the SOAP backend by looking at the MITMDUMP screen.

Part 2

48. Now we need to take an incoming JSON in a POST, parse it, and then convert it into a more complex XML. You should know almost enough to do this. There is one further aspect which will help, which is that we can define a structure that matches the JSON, and parse it into there. This will catch any invalid JSON.

Here is the incoming JSON (also in <https://freo.me/json-bal>)

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

In Ballerina, this matches a struct like:

```
struct payment {
    string cardnumber;
    string postcode;
    string name;
    int month;
    int year;
    int cvc;
    string merchant;
    string reference;
    float amount;
}
```

49. Copy and paste the struct definition from <https://freo.me/struct-bal> and put it at the same level as the service in the .bal file (so it is globally defined for this ballerina program).

50. Add the following new resource below the other one:

```
@http:resourceConfig {
    methods:["POST"],
    path:"/"
}
resource pay (http:Request inreq, http:Response inres) {
}
```

51. You can parse the incoming JSON into a **payment** struct by using this code:

```
var pay, err = <payment>inreq.getJsonPayload();
```

52. You can handle the error like this:

```
if (err != null) {  
    println(err);  
    inres.setStatusCode(400); // BAD REQUEST  
    _ = inres.send();  
    return;  
}
```

53. You need to construct an XML that looks like this (also in <https://freo.me/xml-bal>):

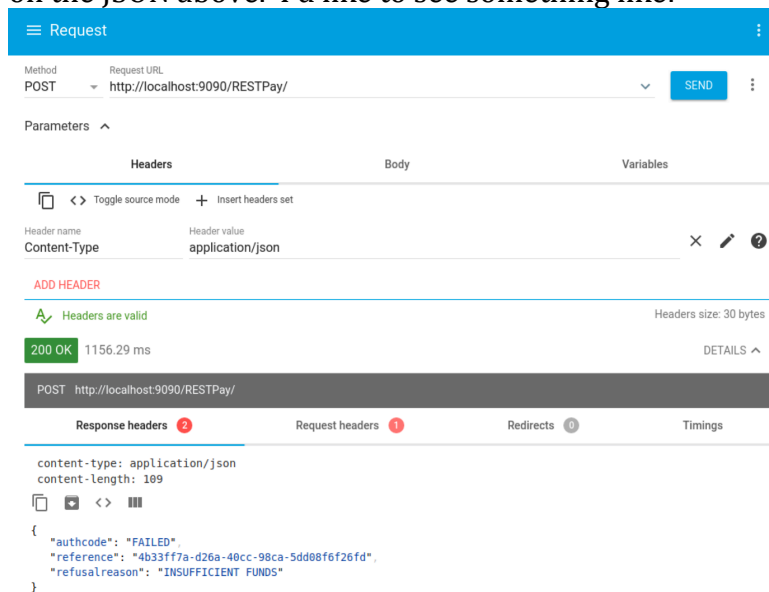
```
<p:authorise>  
  <p:card>  
    <p:cardnumber>4544950403888999</p:cardnumber>  
    <p:postcode>P0107XA</p:postcode>  
    <p:name>P Z FREMANTLE</p:name>  
    <p:expiryMonth>6</p:expiryMonth>  
    <p:expiryYear>2017</p:expiryYear>  
    <p:cvc>999</p:cvc>  
  </p:card>  
  <p:merchant>A0001</p:merchant>  
  <p:reference>test</p:reference>  
  <p:amount>11.11</p:amount>  
</p:authorise>
```

54. 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"  
}
```

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

56. 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:



57. My version is available here: <https://freo.me/final-bal>

58. **Extension:** Use ballerina to build this into a Docker container.

59. We need to do a quick fix so we can run Docker without sudo rights:

```
sudo usermod -aG docker oxsoa
sudo chown "$USER":"$USER" /home/"$USER"/.docker -R
sudo chmod g+rwX "/home/$USER/.docker" -R
```

60. Now we can use the ballerina compiler to compile the program into a container. Close the composer. From the ballerina tools directory:

```
bin/ballerina docker --tag pay:1.0.0 -y ~/ex10/pay.bal
```

61. Now we can run it:

```
docker run --net=host -p 9090:9090 -it pay:1.0.0
```

62. We need --net=host because we are calling localhost in the Ballerina and that doesn't work inside docker.

63. Check its working.

64. **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.