

Exercise 15b

Learn about mediating between services and APIs with Ballerina

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 11
- Ballerina Swan Lake alpha 4
- Visual Studio Code
- Ballerina Swan Lake alpha 4 VSIX
- Docker

Overview

Ballerina is a programming language that is designed around services, microservices and mediation. It also has a graphical view, but unfortunately that is temporarily not working (at the time of writing)!

In this lab we are going to work with a “legacy” payment service based on XML and SOAP.

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

Check Ballerina is installed:

```
bal version
```

```
Ballerina Swan Lake Alpha 4  
Language specification v2020-12-17  
Update Tool 1.3.1
```

1. Make a new Ballerina project

```
cd ~  
bal new mediation
```

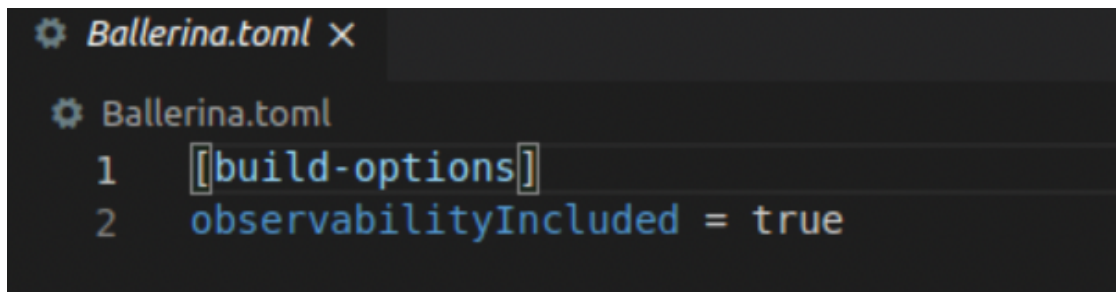
Created new Ballerina package 'mediation' at mediation.

2. This has created a simple package including a config for the project, gitignore and main.bal:

```
cd ~/mediation
```

```
code .
```

3. Take a look at Ballerina.toml



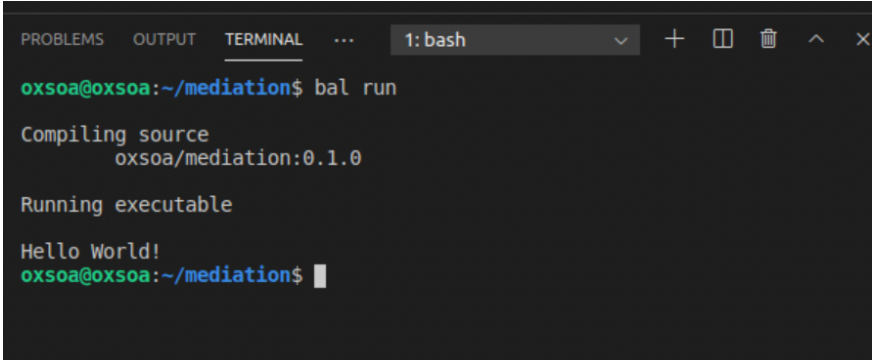
This is a config file for the project.

4. Take a look at main.bal

```
≡ main.bal  ×  
  
≡ main.bal  
1  import ballerina/io;  
2  
3  public function main() {  
4      io:println("Hello World!");  
5  }  
6
```

5. We can quickly run this:

```
bal run
```

A screenshot of a terminal window with a dark background. The terminal shows the command 'bal run' being executed. The output indicates that the source is compiled and an executable is run, resulting in 'Hello World!'. The prompt 'oxsoa@oxsoa:~/mediation\$' is visible at the top and bottom of the terminal output.

```
PROBLEMS OUTPUT TERMINAL ... 1: bash
oxsoa@oxsoa:~/mediation$ bal run
Compiling source
  oxsoa/mediation:0.1.0
Running executable
Hello World!
oxsoa@oxsoa:~/mediation$
```

6. Ballerina is based on the JVM, and this is actually compiling the ballerina code into Java bytecode, making a .JAR file and then executing that.

We can also build manually:

```
bal build
```

```
Compiling source
  oxsoa/mediation:0.1.0
```

```
Generating executable
  target/bin/mediation.jar
```

This JAR file can also be run directly by the JVM:

```
java -jar target/bin/mediation.jar
```

```
Hello World!
```

7. However, we do not really want a “main” method. We are going to create a service instead. Let’s delete main.bal and instead create a new file “mediation.bal” which will expose an HTTP service.

```
rm main.bal
bal clean
code mediation.bal
```

8. Hit Ctrl-s to save, as vscode intelligent completion only kicks in when the file is saved.

9. Let’s start by creating a simple HTTP server first.

10. Type the following code.

```
import ballerina/http;

service / on new http:Listener(8080) {
    resource function get hello() returns json {
        return {
            hello: "world"
        };
    }
}
```

11. Notice that json is a first class type in this language.

12. Save it.

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

A big hint. Under the covers of “yarn dev” was a tool called nodemon that was watching for changes to the files. We can get this to work for Ballerina as well:

```
cd ~/mediation
nodemon -e bal --signal SIGINT --exec bal run
```

```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: bal
[nodemon] starting `bal run`

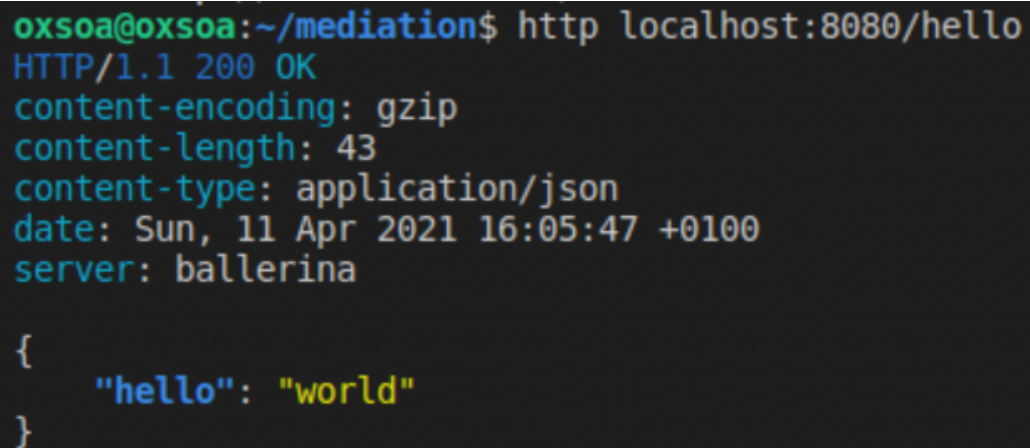
Compiling source
    paul/mediation:0.1.0

Running executable

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

14. Start another terminal and

```
http http://localhost:8080/hello
```



```
oxsoa@oxsoa:~/mediation$ http localhost:8080/hello
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 43
content-type: application/json
date: Sun, 11 Apr 2021 16:05:47 +0100
server: ballerina

{
  "hello": "world"
}
```

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

16. Let's start by defining a new type. Ballerina does have classes, but it also has a simpler concept called a **record** that is better for mapping to JSON, etc. You can read more here:

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

Add this code above your service definition:

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

17. Your code should look like:

```
1  import ballerina/http;
2
3  type Ping record {
4      string name;
5  };
6
7  service / on new http:Listener(8080) {
```

18. Let's add a ping function that expects a POST message and reads a JSON body into a record of type Ping:

```
resource function post ping(@http:Payload Ping p) returns json|error {
  return {
    pingResponse: p.name
  };
}
```

This is reasonably obvious. The `@http:Payload` indicates that the parameter is coming from mapping the payload into a record of type `Ping`.

Note the “union type” `json|error`

This means that this method can return either a json or an error.

19. You can test you code like this:

```
~$ http -v http://localhost:8080/ping name=paul
POST /ping HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 16
Content-Type: application/json
Host: localhost:8080
User-Agent: HTTPie/2.4.0
```

```
{
  "name": "paul"
}
```

```
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 49
content-type: application/json
date: Sun, 11 Apr 2021 17:13:04 +0100
server: ballerina
```

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

20. Start the “backend” SOAP service:

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

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

```
mitmdump --listen-port 8000 --set flow_detail=3 --mode reverse:http://localhost:8888
```

22. Check that it is running:

Browse: <http://localhost:8000/pay/services/paymentSOAP?wsdl>

23. 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.

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

```
http://localhost:8888/pay/services/paymentSOAP
content-type: application/xml

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

<< 200 185b
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Mon, 12 Apr 2021 19:12:23 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>
```

24. The first part will guide creating a mediation for ping. I will then give a more more freeform approach for the payment method..

25. There is a SOAP library for Ballerina, but we don't actually need it. 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>

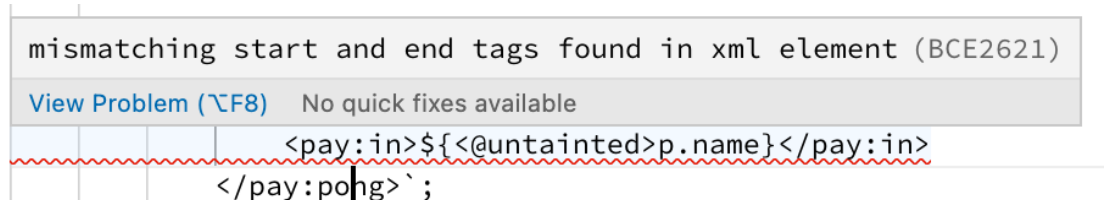
We can really simply create an XML body to send to the payment service. Add this code to the body of your “ping” method. I basically copied this from the SOAPUI method. In addition, I split out the SOAP envelope XML from the body to be clearer and more reusable.

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

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

Notice the simple templating `${p.name}`

Another thing worth noticing is that the vscode plugin knows when your xml is valid or not. For example if I change the end tag to `</pay:pong>` instead you will see:



It is hard to explain how clever this is unless you've struggled with coding XML in other languages!

26. Now we need to send the XML to the HTTP backend. At the top of the file, after the imports, add:

```
configurable string host = "http://localhost:8000";
```

27. Now, back under the xml code, add:

```

http:Client c = check new (host);

var response = check c->post("/pay/services/paymentSOAP", soap);
xml x = check response.getXmlPayload();
string pingresponse = (x/**/<pay:out>/*).text().toString();
return {
    pingResponse: pingresponse
};

```

You can delete the old “return” lines.

What is going on here?

Firstly, let’s understand “check”. This looks for an error and throws it if it finds it (<https://ballerina.io/learn/by-example/check.html>)

Secondly this is an interesting line:

```
string pingresponse = (x/**/<pay:out>/*).text().toString();
```

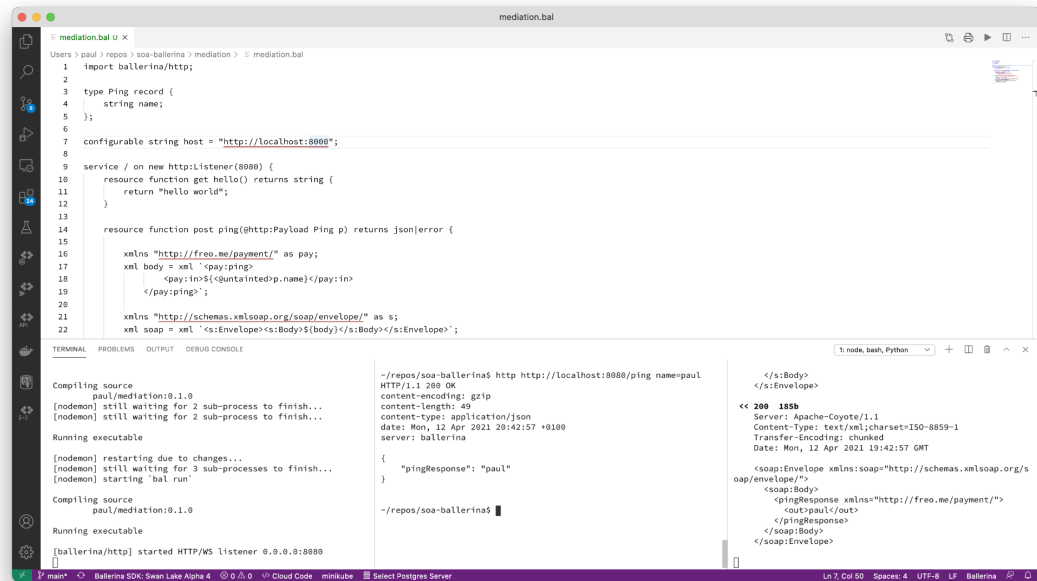
Basically this looks inside the xml of variable “x” at any depth for an element <pay:out>, then for its child (.*) and then gets the text and converts to a string.

28. Your code should now look like:

```
1  import ballerina/http;
2
3  type Ping record {
4      string name;
5  };
6
7  configurable string host = "http://localhost:8000";
8
9  service / on new http:Listener(8080) {
10     resource function get hello() returns string {
11         return "hello world";
12     }
13
14     resource function post ping(@http:Payload Ping p) returns json|error {
15
16         xmlns "http://freo.me/payment/" as pay;
17         xml body = xml `<pay:ping>
18             <pay:in>${<@untainted>p.name}</pay:in>
19             </pay:ping>`;
20
21         xmlns "http://schemas.xmlsoap.org/soap/envelope/" as s;
22         xml soap = xml `<s:Envelope><s:Body>${body}</s:Body></s:Envelope>`;
23
24         http:Client c = check new (host);
25
26         var response = check c->post("/pay/services/paymentSOAP", soap);
27         xml x = check response.getXmlPayload();
28         string pingresponse = (x/**<pay:out>*/).text().toString();
29         return {
30             pingResponse: pingresponse
31         };
32     }
33 }
```

Try it out with the same request as before.

29. This time you should see a successful call in the MITMDUMP terminal.
Here is a screenshot:



The screenshot shows a VS Code editor window titled 'mediation.bal'. The editor contains a Ballerina script for a service named 'mediation'. The script defines a 'Ping record' type and a 'post ping' resource function that sends an HTTP POST request to 'http://freo.me/payment/'. The terminal output shows the compilation and execution of the script. It includes messages about compiling source, running the executable, and starting the HTTP/WS listener. The terminal also displays the output of a curl command, showing a successful HTTP 200 OK response with a 'pingResponse' of 'paul'. On the right side of the terminal, there is a preview of the XML response body, which is a SOAP envelope containing a 'pingResponse' element with the value 'paul'.

```
1 import ballerina/http;
2
3 type Ping record {
4     string name;
5 };
6
7 configurable string host = "http://localhost:8080";
8
9 service / on new http:Listener(8080) {
10     resource function get hello() returns string {
11         return "hello world";
12     }
13
14     resource function post ping(@http:Payload Ping p) returns json:error {
15         xmlns "http://freo.me/payment/" as pay;
16         xml body = xml '<pay:ping>';
17         <pay:in>${@p.name}</pay:in>
18         </pay:ping>';
19
20         xmlns "http://schemas.xmlsoap.org/soap/envelope/" as s;
21         xml soap = xml '<s:Envelope><s:Body>${body}</s:Body></s:Envelope>';
22     }
23 }
```

```
~/repos/soa-ballerina$ http http://localhost:8080/ping name=paul
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 49
content-type: application/json
date: Mon, 12 Apr 2021 20:42:57 +0100
server: ballerina
{
  "pingResponse": "paul"
}

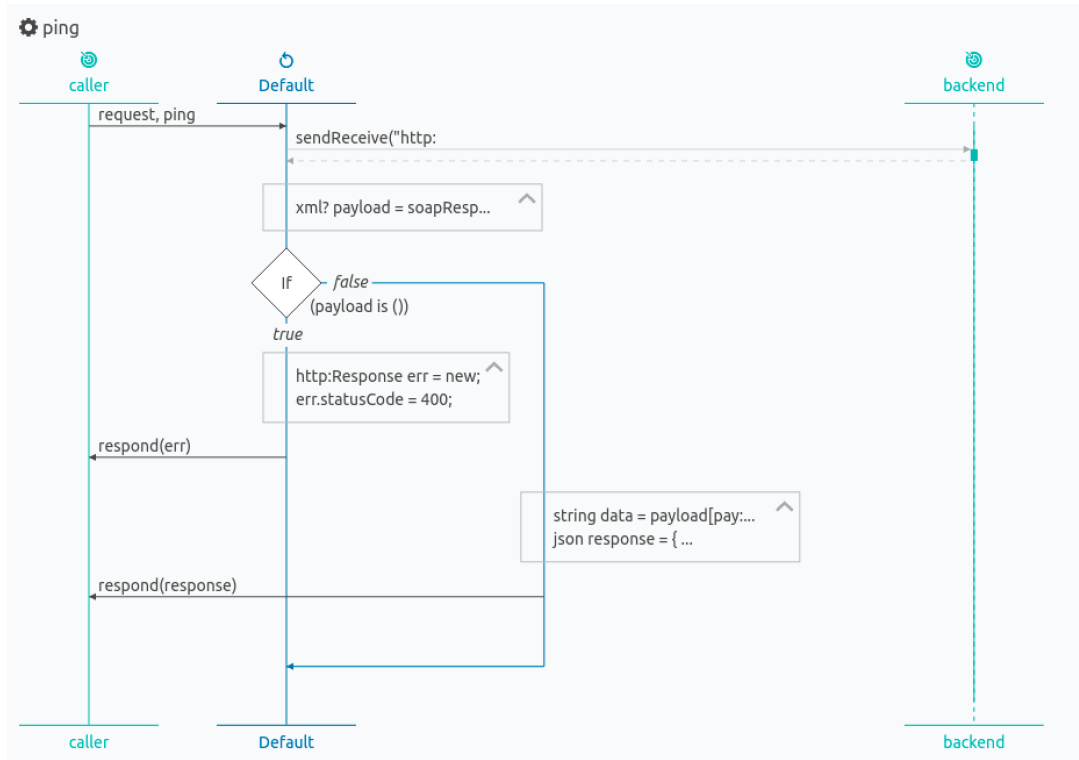
~/repos/soa-ballerina$
```

```
<< 200 185b
Server: Apache-Coyote/1.1
Content-Type: text/xml; charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Mon, 12 Apr 2021 19:42:57 GMT

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



Ballerina is a language specifically designed for orchestration of services, modelled on sequence diagrams. Unfortunately the vscode support for the graphical view is temporarily disabled in this alpha preview. Here is a screenshot from a previous release of some similar but slightly different code.



The cool thing is that these sequence diagrams are automatically generated and updated from the code. According to this link, they should be enabled again soon:

<https://stackoverflow.com/questions/66401970/ballerina-swan-lake-an-ides-how-to-get-sequence-diagrams>

PART C - ACTUAL CARD PAYMENT

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

31. Here is the XML interaction we need to talk to the SOAP service

```
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>
```

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

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

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

34. Firstly, obviously you need to copy and paste the resource function for **ping** and rename the new version to be **authorise**. This will make the request path be `/authorise`

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

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

Also available here <https://freo.me/pay-struct>

You can also read more about Records here:

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

36. 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).

37. 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>
```

You will need to use the same templating as above to insert the data.

38. 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"
}
```

I have created a Postman collection json to test this:

39. If you get stuck, my version is available here:

<https://github.com/pzfreo/soa-ballerina/blob/main/mediation/mediation.bal>

Extensions

1. Try the OpenAPI ballerina to swagger interface:

```
bal openapi -i mediator.bal
```

2. Try generating Ballerina server-side skeletons from the Purchase swagger:

<https://ballerina.io/learn/tooling-guide/cli-tools/openapi/>

3. Take a look at Ballerina's support for gRPC:

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

4. And AWS Lambda:

<https://ballerina.io/learn/by-example/aws-lambda-deployment.html>