

# Hands-on Lab

## Session 7502

# API Connect Toolkit and DataPower working seamlessly together

Alex Irazabal, IBM

Tony French, IBM

© Copyright IBM Corporation 2017

IBM, the IBM logo and ibm.com are 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](http://www.ibm.com/legal/copytrade.shtml).

This document is current as of the initial date of publication and may be changed by IBM at any time.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

## Introduction

During lab 7502 you will experience the API Connect Toolkit, API Designer and DataPower for Docker all working seamlessly together.

DataPower provides a rich set of API policies for transforming and securing API workloads and you will now get the opportunity to experiment with rate-limiting, JWT security and SOAP transformation during this lab.

The API Toolkit manages DataPower via Docker for you, so you can focus on the API definitions and implementation.

Some key benefits of this feature include:

- Develop and test APIs that use the full complement of IBM API Connect Assembly policies.
- Saved changes are instantly synchronized with DataPower running as a Docker container for rapid testing and feedback.
- Test product and plan level concepts such as rate-limiting.
- DataPower error logging is integrated into the API Designer logging console.
- Request/Response logging is also available from the logging console with latency information.

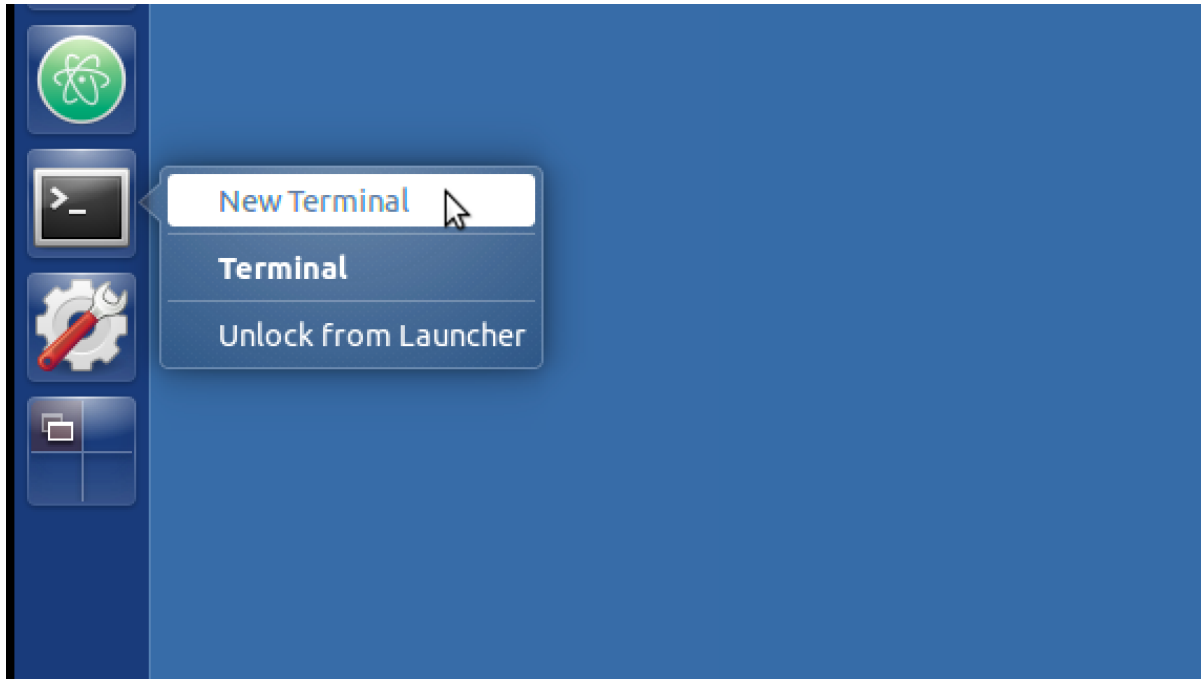
All source content for this lab is available from GitHub at <https://github.com/ibm-datapower/interconnect-labs/tree/master/IC17-7502-DataPower-APIConnect-Toolkit> so don't worry if you don't get it completed today.

Note: this capability requires API Connect version 5070.

## Rate-limit a simple Loopback API

To get started we are going to use the API Connect Toolkit to scaffold a sample Loopback API application.

First open a new Terminal window



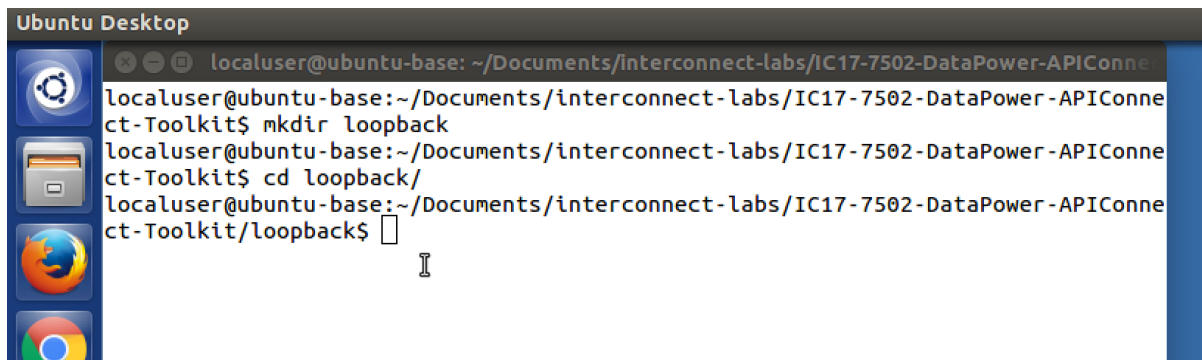
Create a new empty folder using the following commands:

```
cd /home/localuser/Documents/interconnect-labs/IC17-7502-DataPower-
APICConnect-Toolkit/

mkdir loopback

cd loopback
```

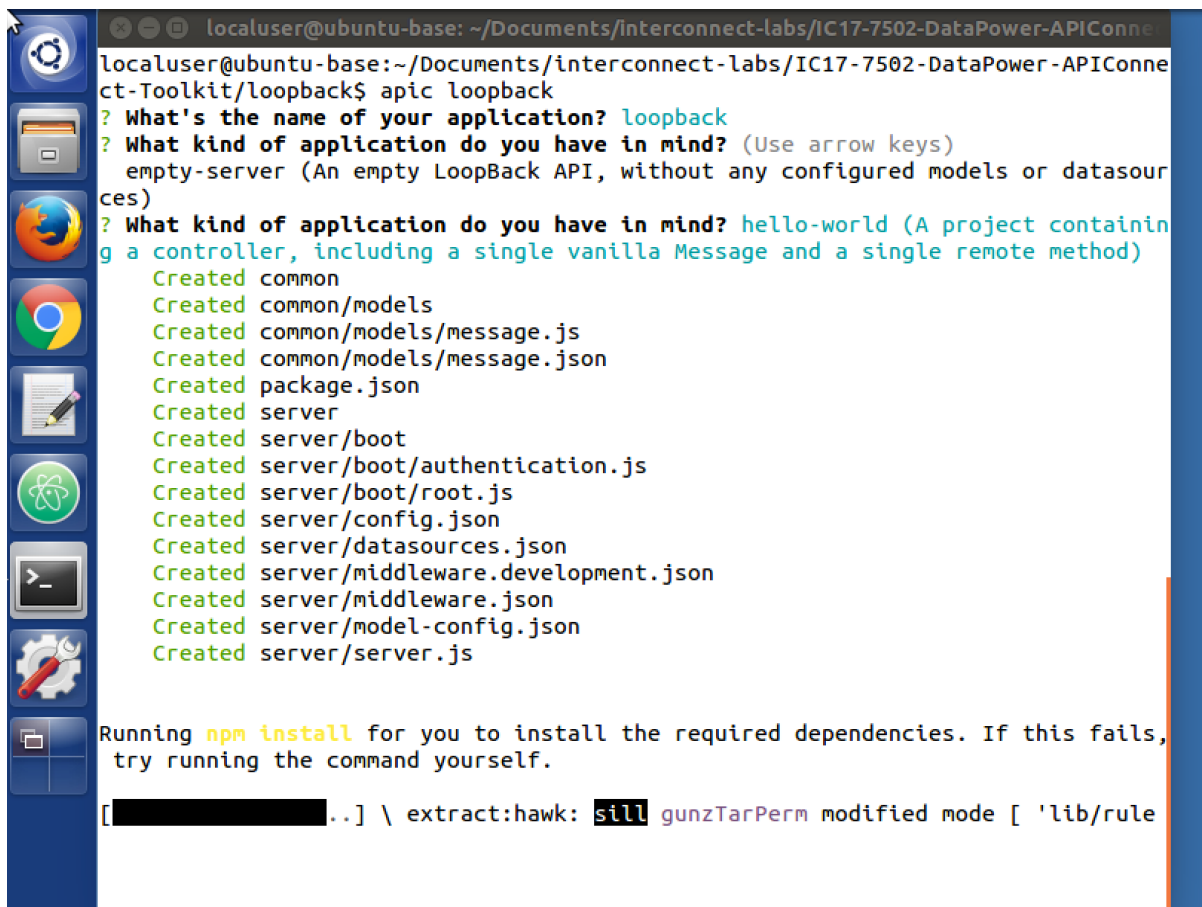
Terminal window should look something like this:



```
localuser@ubuntu-base: ~/Documents/interconnect-labs/IC17-7502-DataPower-APIConnect-Toolkit$ mkdir loopback
localuser@ubuntu-base:~/Documents/interconnect-labs/IC17-7502-DataPower-APIConnect-Toolkit$ cd loopback/
localuser@ubuntu-base:~/Documents/interconnect-labs/IC17-7502-DataPower-APIConnect-Toolkit/loopback$
```

Next, we will scaffold a sample loopback API application. From the loopback directory, execute the following command and select all the defaults

```
apic loopback
```



```
localuser@ubuntu-base:~/Documents/interconnect-labs/IC17-7502-DataPower-APIConnect-Toolkit/loopback$ apic loopback
? What's the name of your application? loopback
? What kind of application do you have in mind? (Use arrow keys)
  empty-server (An empty LoopBack API, without any configured models or datasources)
? What kind of application do you have in mind? hello-world (A project containing a controller, including a single vanilla Message and a single remote method)
Created common
Created common/models
Created common/models/message.js
Created common/models/message.json
Created package.json
Created server
Created server/boot
Created server/boot/authentication.js
Created server/boot/root.js
Created server/config.json
Created server/datasources.json
Created server/middleware.development.json
Created server/middleware.json
Created server/model-config.json
Created server/server.js

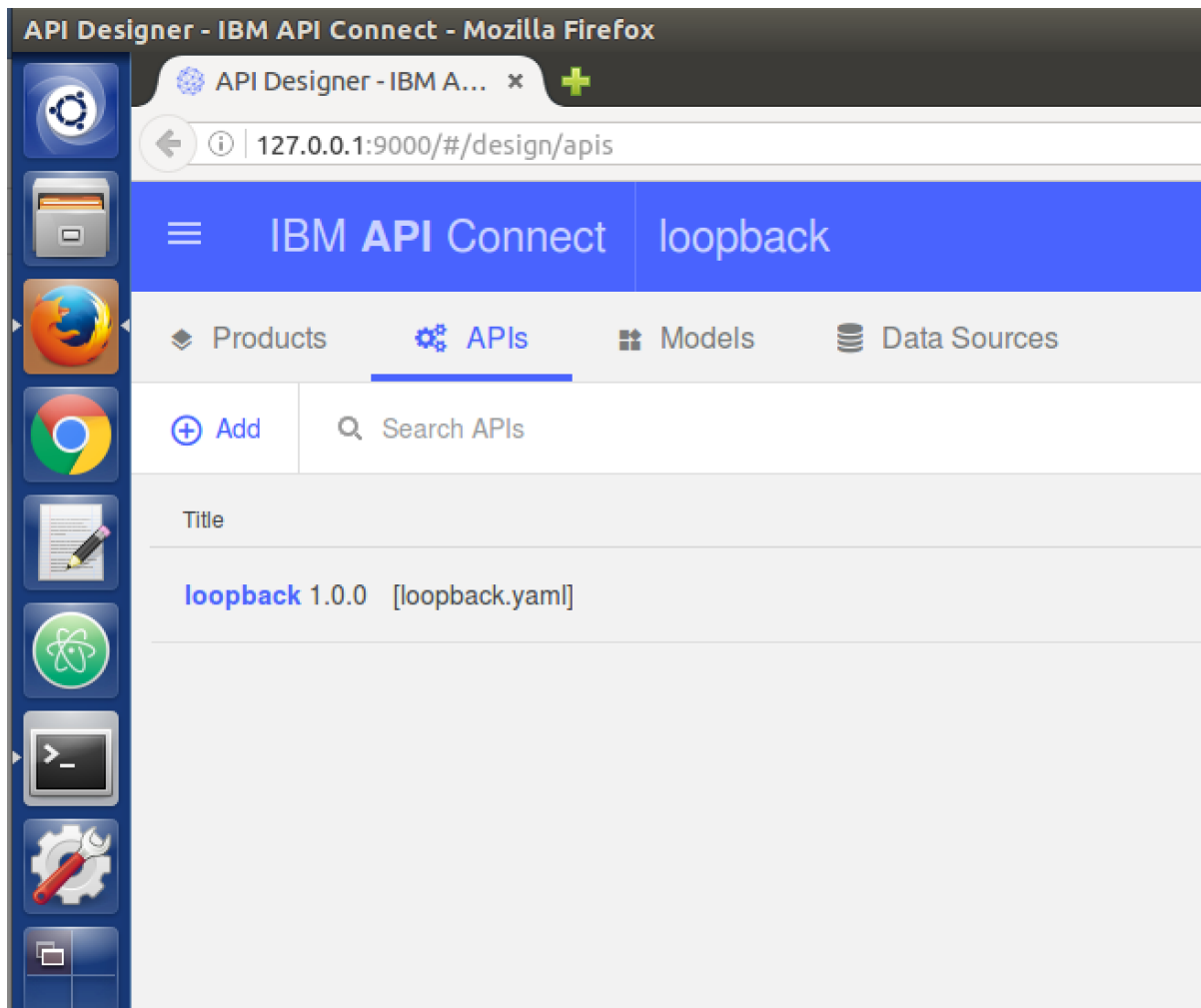
Running npm install for you to install the required dependencies. If this fails, try running the command yourself.

[REDACTED] \ extract:hawk: sill gunzTarPerm modified mode [ 'lib/rule
```

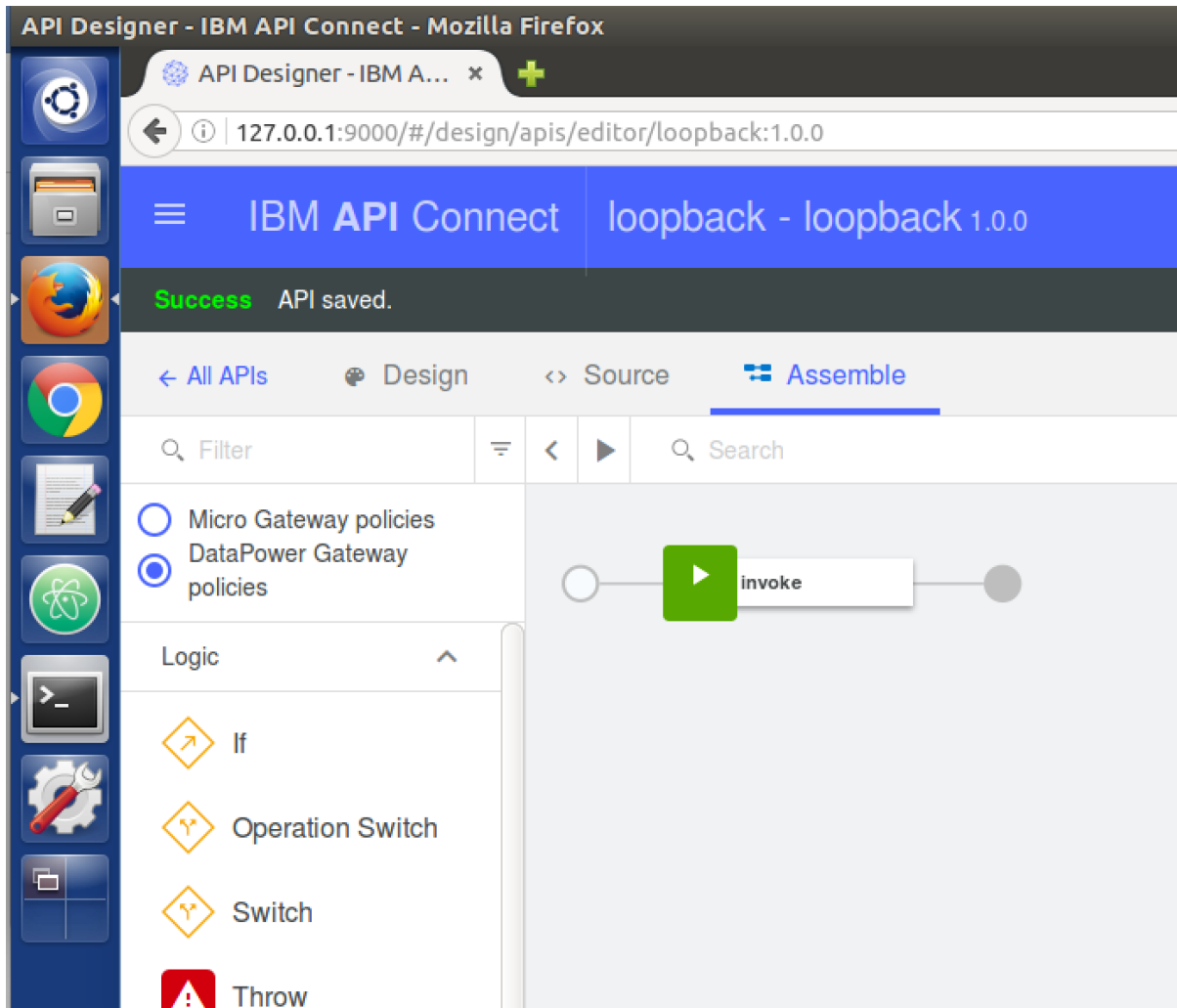
Next type

```
apic edit
```

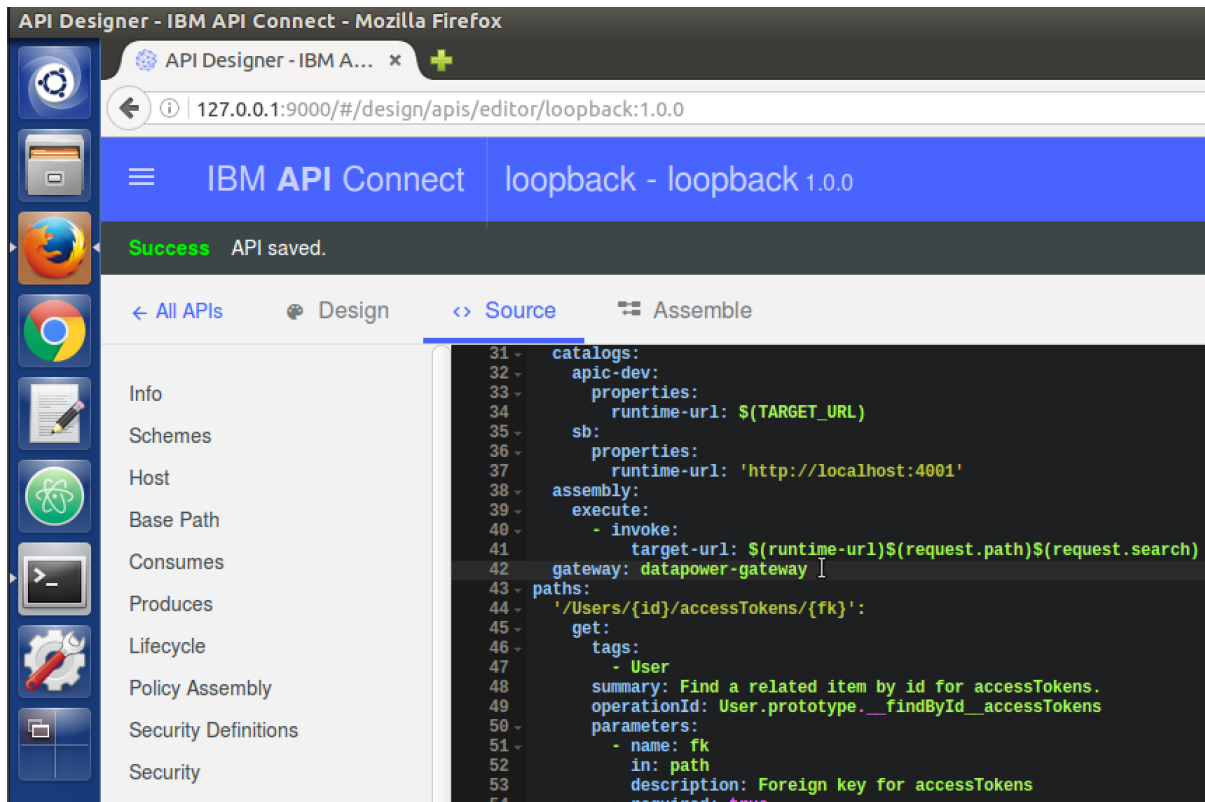
to launch the API Designer



select the loopback API, click the “Assembly” tab and select “DataPower Gateway policies” and click the save icon on the right. This will update the API definition file in the current project to specify DataPower as the default gateway type.



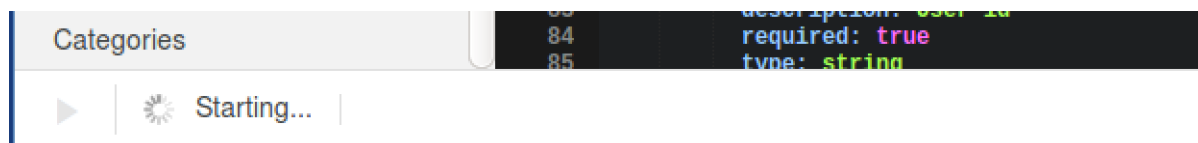
you can also see in the “Source” view that the following “gateway: datapower-gateway” was added on line 42:



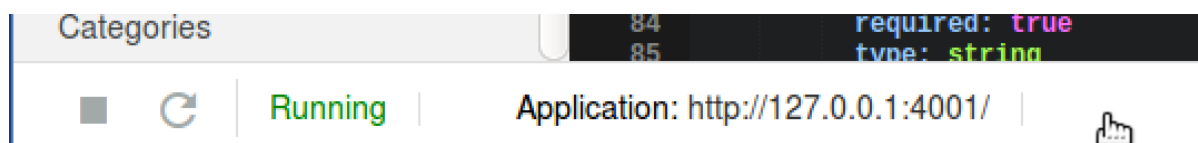
Now that we have specified DataPower as the gateway type, it's time to start both the Loopback API application and the gateway.



Press the play button and wait approximately 2 minutes to allow DataPower to start and retrieve the configuration.



the application will complete the start procedure before the gateway:

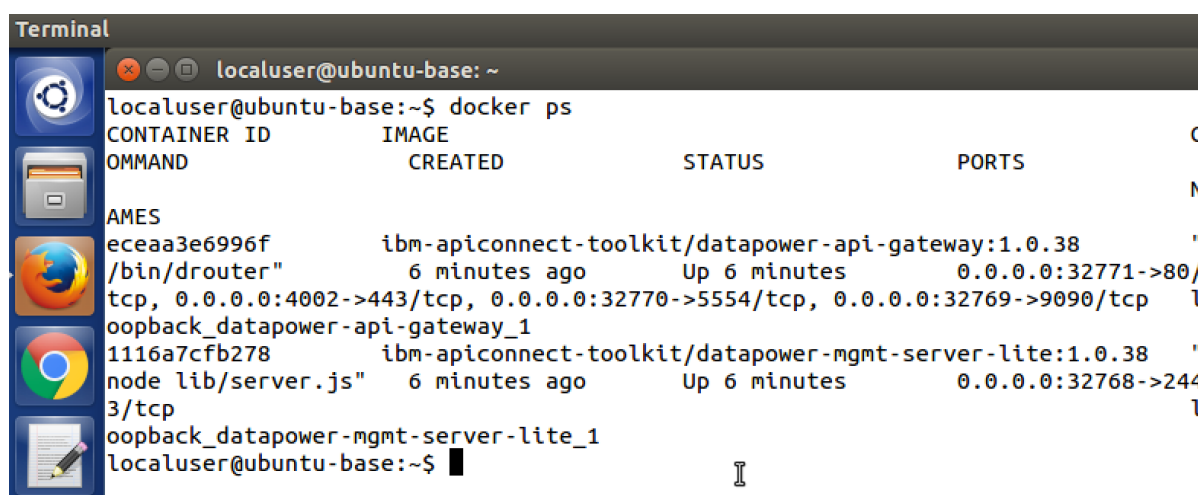


be patient and wait for the following (clicking stop, start or restart will only prolong the operation)

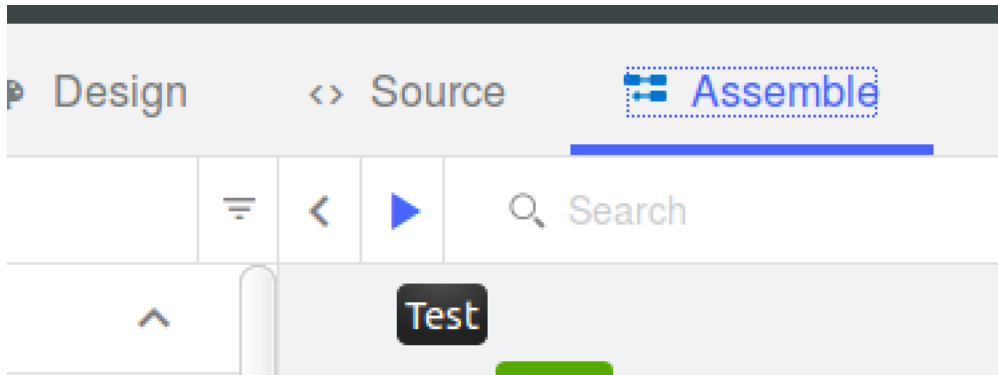


This indicates that the Gateway and Application are both running and listing on ports 4002 and 4001 respectively.

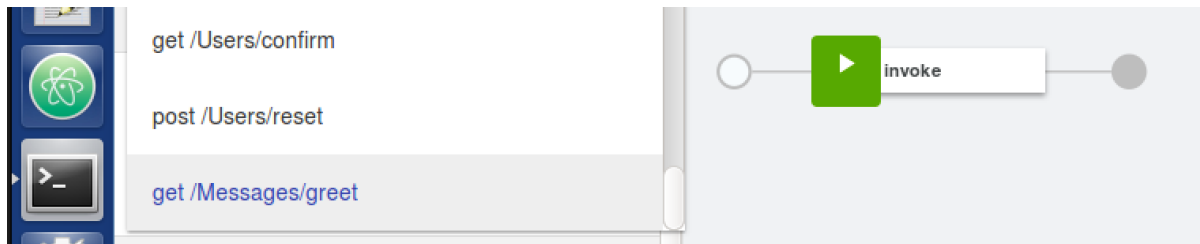
You can switch to a new Terminal screen and see that Docker is running the DataPower image and another image (datapower-mgmt-server-lite) to map the current project to DataPower configuration:



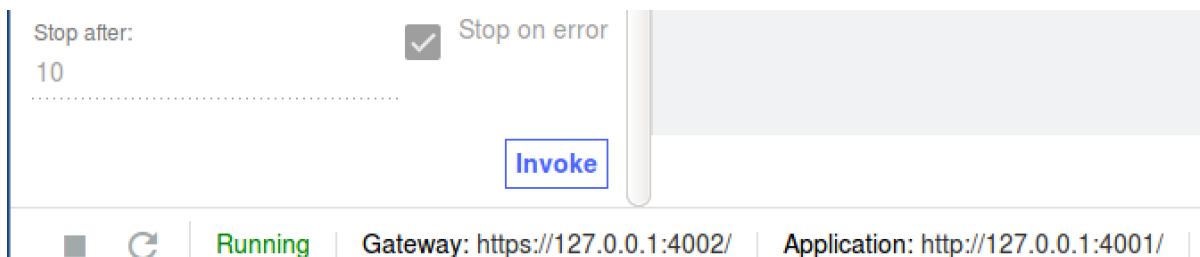
The next step is to test the running API using the Assembly Test Tool. Click on the “Assembly” tab and click top “play” button:



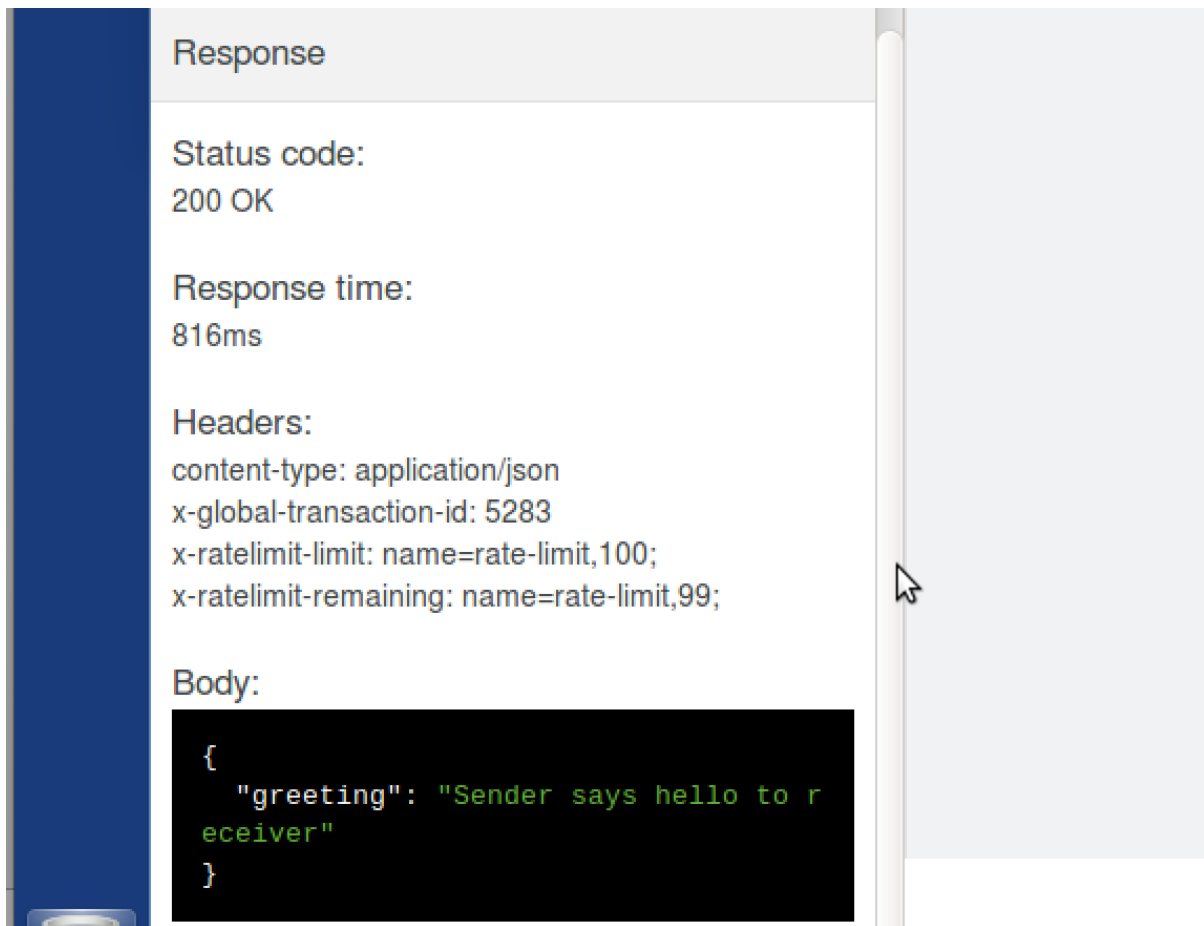
select the get /Messages/greet operation (located at the end of the operation list)



and scroll down to the bottom of the page to find the “Invoke” button (note: you might need to use both scrollbars, left and right to get this button to display, depending on the screen resolution)



click "Invoke" to see the results:



The screenshot shows the 'Response' tab of the API Connect Toolkit. It displays the following information:

- Status code:** 200 OK
- Response time:** 816ms
- Headers:**
  - content-type: application/json
  - x-global-transaction-id: 5283
  - x-ratelimit-limit: name=rate-limit,100;
  - x-ratelimit-remaining: name=rate-limit,99;
- Body:**

```
{
  "greeting": "Sender says hello to receiver"
}
```

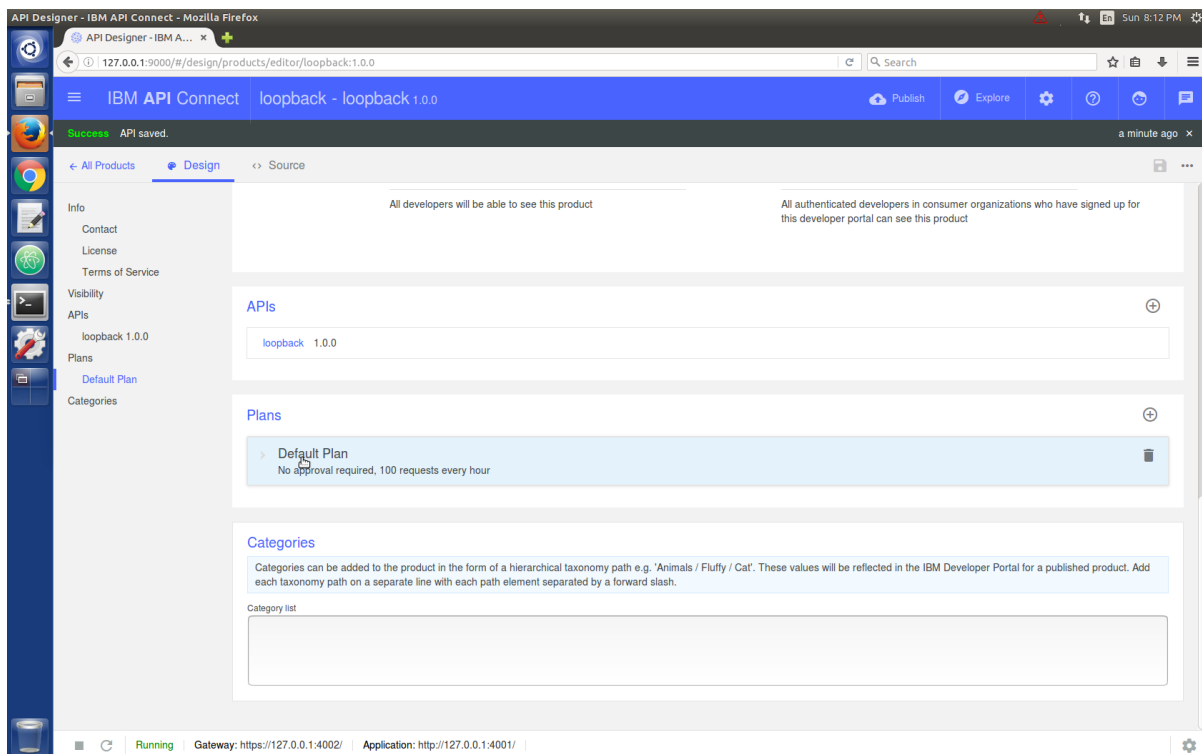
To test the rate-limit feature, its quicker to issue this request using curl. Open a new terminal window and issue the following command:

```
curl -k https://localhost:4002/api/Messages/greet -H "X-IBM-Client-Id: default" -H "X-IBM-Client-Secret: SECRET" | json_pp
```

```
localuser@ubuntu-base: ~  
localuser@ubuntu-base:~$ curl -k https://localhost:4002/api/Messages/greet -H "X-IBM-Client-Id: default" -H "X-IBM-Client-Secret: SECRET" | json_pp  
% Total    % Received % Xferd  Average Speed   Time    Time     Time    Current  
             Dload  Upload   Total   Spent    Left     Speed  
100    44      0    44      0      0    1084      0 --:--:-- --:--:-- --:--:-- 1073  
{  
  "greeting" : "Sender says hello to receiver"  
}  
localuser@ubuntu-base:~$
```

Ok now to the fun part...

Click the “All APIs” tab, click the “Products” tab and select the loopback product. Click the default plan.



Expand the “Default Plan”, select 1 request per minute, click “Enforce hard limit” and click the “Save” button on the top right of the screen.


**Plans**



▼ Title  
Default Plan

---

Name: default Description: Default Plan

---

Rate limits (calls / time interval) 

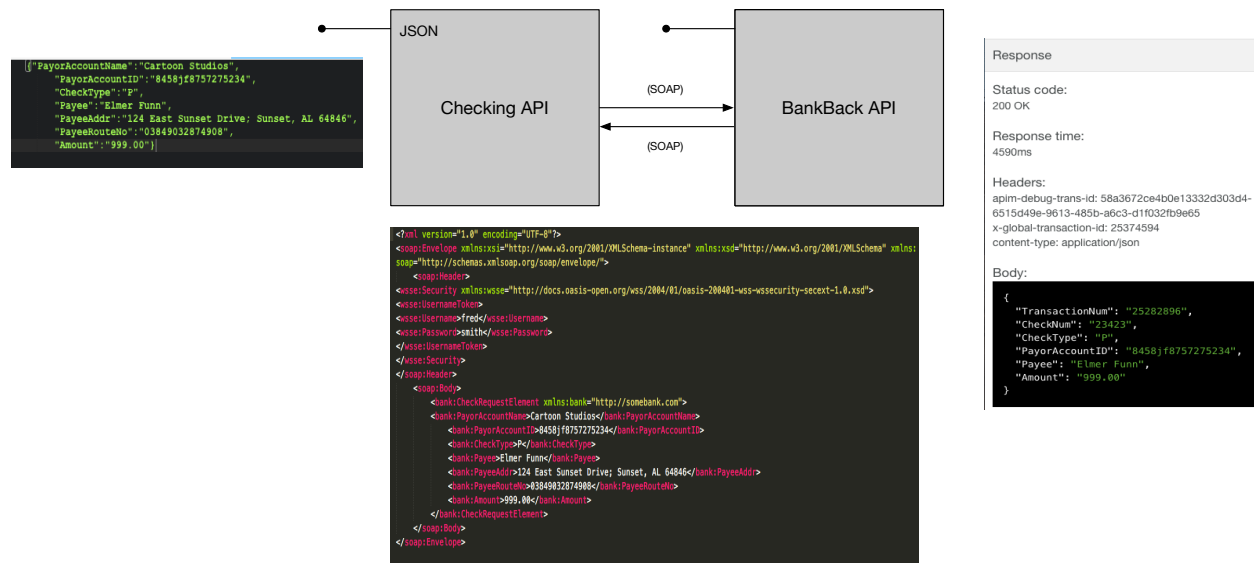
1 / 1 Minute  ☒ Enforce hard limit 

Switch back to the previous Terminal window or create a new one and issue the previous curl command twice. You should observe the second request being refused with a 429 response:

```
localuser@ubuntu-base:~$ curl -k https://localhost:4002/api/Messages/greet -H "X-IBM-Client-Id: default" -H "X-IBM-Client-Secret: SECRET" | json_pp
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100    44    0    44    0    0    688      0  --:--:-- --:--:-- --:--:--    698
{
  "greeting" : "Sender says hello to receiver"
}
localuser@ubuntu-base:~$ curl -k https://localhost:4002/api/Messages/greet -H "X-IBM-Client-Id: default" -H "X-IBM-Client-Secret: SECRET" | json_pp
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100    96    0    96    0    0    354      0  --:--:-- --:--:-- --:--:--    355
{
  "statusCode" : "429",
  "moreInformation" : "Rate Limit exceeded",
  "errorMessage" : "Too Many Requests"
}
localuser@ubuntu-base:~$
```

This part of the lab will teach you to expose a REST API (JSON data) which calls a standard web service backend (SOAP). The exercise exemplifies a common use case: JSON -> SOAP and SOAP -> JSON conversion.

Lab 1 - REST to SOAP (and back again)



In this section, you'll set up an API acting like a back-end service. This service will take a SOAP request as input and extract certain fields from it returning a different response payload.

Click on "terminal" to open a new window. Make a new directory in your home directory: **mkdir lab1**.

This directory will be used to build the solution. The actual lab (and solution) is in the /home/localuser/interconnect-labs/IC17-7502-DataPower-Toolkit/labx directory.

You'll be using some of the files there.

Now let's change directory to lab1: **cd lab1**. Follow the directions below.

### Step 1: Open API Designer

On the command line, type: **apic edit**, to start the API Designer. A browser window will open up on port 9000.

## Step 2: Create new API BankBack

Create a new api in Drafts called BankBack. Click “+” to add a “New API”. Enter the title, and additional properties as depicted below. Setting the “Gateway” property will enable the local DataPower option we will be testing.

Before creating the api, go ahead and “Add a product”. Take the defaults. Unselect “Publishing”, since we will not be doing that yet. Then hit “Create API” to finish and create our API.

The screenshot shows the 'New API' configuration form. It has a title bar 'New API' and a sidebar with categories: Info, Additional properties, API template, Target, Security, and Gateway. The 'Info' section contains four fields: 'Title \*' with value 'BankBack', 'Name \*' with value 'bankback', 'Base Path' with value '/bankback', and 'Version \*' with value '1.0.0'. The 'Additional properties' section is expanded, showing 'API template' set to 'Default', 'Target' with a label 'Target endpoint (if known)', 'Security' with 'Identify using' set to 'Client ID' and a checked 'Enable CORS' checkbox, and 'Gateway' set to 'DataPower Gateway'. At the bottom right are three buttons: 'Cancel', 'Add a product...', and 'Create API'.

Category	Property	Value
Info	Title *	BankBack
	Name *	bankback
	Base Path	/bankback
	Version *	1.0.0
Additional properties	API template	Default
	Target	Target endpoint (if known)
	Security	Identify using: Client ID, Enable CORS: <input checked="" type="checkbox"/>
	Gateway	DataPower Gateway

Set the API “Consumes/Produces” to application/xml as shown below:

**Consumes** ☐ application/json ☒ application/xml

Additional media types

Add media type

**Produces** ☐ application/json ☒ application/xml

Additional media types

Add Media Type

Change the default “Located In” field of ClientIDHeader (APIKey) from “Header” to “Query” in “Security Definitions”.

### Security Definitions



clientIdHeader (API Key)

Name \*

clientIdHeader

Parameter name \*

client\_id

Located In

Query

Description

### Step 3: Add POST operation

Add a new Path to the API. Click on “Paths” on the left menu. Click “+” to add a new path. Enter “/checkRequest”. The screen should look like this now.



## Paths

The screenshot shows the 'Paths' configuration page. At the top, there's a header bar with a plus icon. Below it, a path entry for '/checkRequest' is shown with a trash icon on the right. Underneath, the 'Path' field contains '/checkRequest' and there are links for 'Add Operation' and 'Add Parameter'. The 'Parameters' section indicates 'No parameters defined'. At the bottom, a summary bar shows a 'GET' method for the path '/checkRequest' with a trash icon.

However, we need a POST operation, since this is our back end and will be receiving a SOAP payload. Delete the “GET” by clicking on the trash can icon to the right of the “GET”.

Add a new Operation (top left link under Paths). Select “POST”. Now we need to add the body of the request so we can test it locally (DataPower via test tool).

Click “Add Parameter”, add new parameter. For name enter “body” and “Located in”, select “Body”. The screen should now look like this:

## Paths

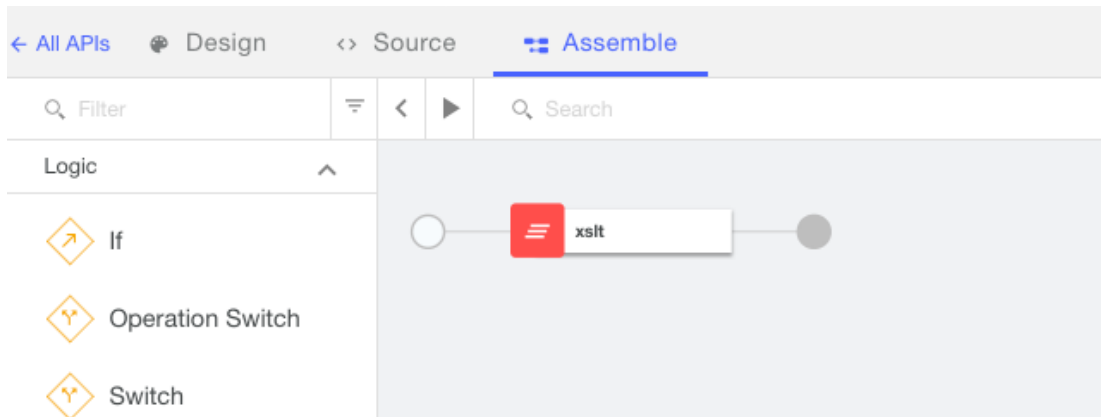
The screenshot shows the 'Paths' configuration page after adding a POST operation and a parameter. The path entry for '/checkRequest' now has a trash icon. The 'Parameters' section contains a table with one parameter:

Name	Located In	Description	Required	Type
body	Body		<input type="checkbox"/>	string

Below the table, a summary bar shows a 'POST' method for the path '/checkRequest' with a trash icon.

#### Step 4: Add XSLT Policy

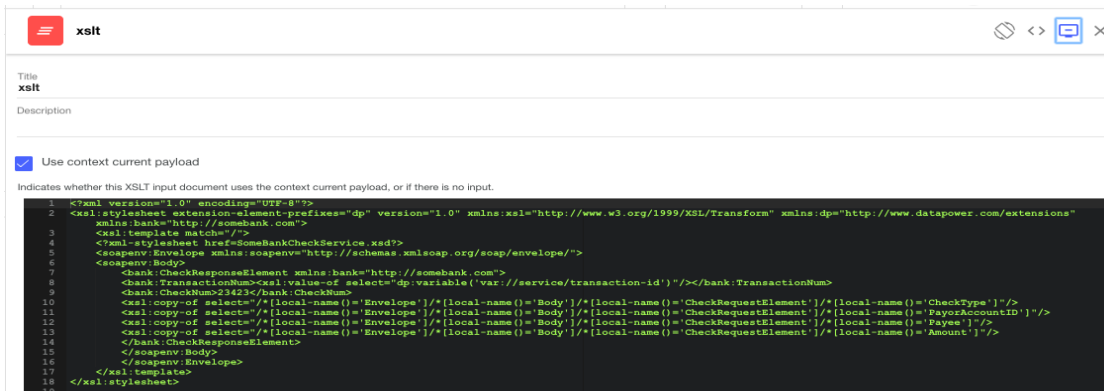
Head over to “Assemble” (top of the screen next to Design/Source) and delete the default invoke policy (click the trash can as you hover over the policy. Now we will add an XSLT policy by dragging the policy from the left palette onto the canvas. When dragging the policy don't let go until you place it on the yellowish square between the two dots. This policy will parse incoming SOAP and extract the data we want returned. The screen should now look like this:



if you click on the xslt policy, you'll get a empty screen. This is where normally you would enter (or copy) your logic. We are going to copy it from the lab files.

### Step 5: Copy XSLT

Select the file under lab1 resources folder named: BankBack.xsl. Double click to edit, select all (ctrl-a) and copy (ctrl-c). Go back to the empty editor open in API Designer and paste the xslt code (ctrl-v). The screen should look like this now:



Make sure the “Use context current payload” is checked off. Otherwise we won't see any payload.

We will be extracting CheckType, PayorAccountID, Payee, and Amount for our SOAP response.

## Step 6: Test

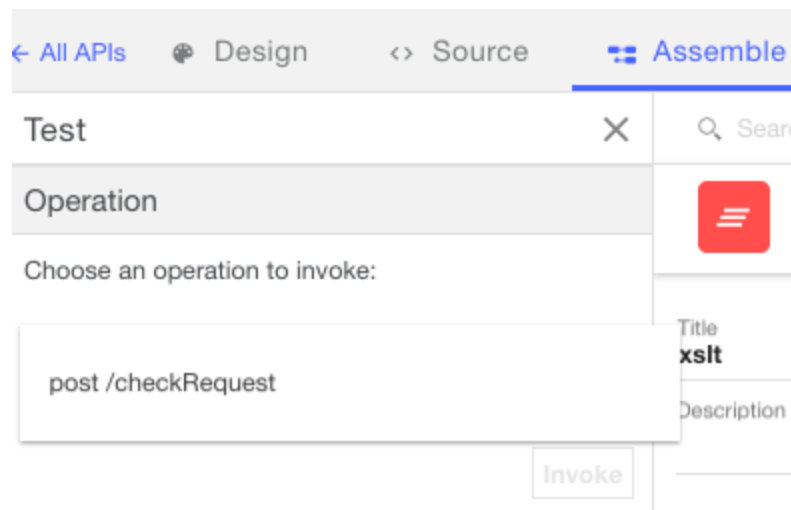
At the bottom of the screen you'll notice a "play" (>) button and the text: "Stopped"



Click on the arrow to start your DataPower gateway.

Click the Floppy disk icon on the top right of the screen to save the API. On the left hand side of the screen there is a "play" icon (>). This is the test button. Click it to open the test pane on the left.

Select operation "post /checkRequest".



We need to provide a "body" parameter to emulate a payload request coming into the API. One of the files provided is called SBCRequest.xml. Open that file with an editor and copy paste into here.

Test ×

```
body
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/env
elope/">
  <soap:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken>
        <wsse:Username>fred</wsse:Username>
        <wsse:Password>smith</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <bank:CheckRequestElement
      xmlns:bank="http://somebank.com">
      <bank:PayorAccountName>Cartoon
Studios</bank:PayorAccountName>

      <bank:PayorAccountID>8458jf8757275234</bank:P
ayorAccountID>

      <bank:CheckType>P</bank:CheckType>
      <bank:Payee>Elmer
Funn</bank:Payee>
      <bank:PayeeAddr>124 East Sunset
Drive; Sunset, AL 64846</bank:PayeeAddr>

      <bank:PayeeRouteNo>03849032874908</bank:Pay
eeRouteNo>

      <bank:Amount>999.00</bank:Amount>
    </bank:CheckRequestElement>
  </soap:Body>
</soap:Envelope>
```

[Show schema](#) | [Generate](#)

☐ Repeat

Repeat the API invocation a set number of times. or

Then scroll down and press “Invoke”. If you get something like this:

**Response**

Status code:  
-1

No response received. Causes include a lack of CORS support on the target server, the server being unavailable, or an untrusted certificate being encountered.

Clicking the link below will open the server in a new tab. If the browser displays a certificate issue, you may choose to accept it and return here to test again.  
<https://localhost:4001/bankback/checkRequest>

This is because is the first time going to this address and the certificates are not trusted. Click on the link and accept the exception. Then click invoke again.

This time, you should get this instead:

Body:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:bank="http://somebank.com">
  <soapenv:Body>
    <bank:CheckResponseElement>
      <bank:TransactionNum>24858114</bank:TransactionNum>
      <bank:CheckNum>23423</bank:CheckNum>
      <bank:CheckType
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">P
      </bank:CheckType>
      <bank:PayorAccountID
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">8458jf8757275234
      </bank:PayorAccountID>
      <bank:Payee
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">Elmer Funn
      </bank:Payee>
      <bank:Amount
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">999.00
      </bank:Amount>
    </bank:CheckResponseElement>
  </soapenv:Body>
</soapenv:Envelope>
```

[Debug](#)

In this section, we will create another API which will be calling the back end we just created. This new rest API takes a JSON request converts it to SOAP, calls the BankBack service and converts the response back to JSON.

You'll need to use the terminal app to get a command line. In the left hand of the screen there is a tool bar. Click on "terminal" to open up a new window. In this lab we will use the actual directory that contains the yaml for the api. Change directory to /home/localuser/interconnect-labs/IC17-7502-DataPower-Toolkit/lab2.

### Step 1: Create new API - Checking

Click "all APIs" on the left of the API Designer page to go back to the list of APIs. Click "Add New API". The new API will be called "Checking". Do not add a product, since we are going to piggy back on the one we already created. Do change the "Gateway" to DataPower.

Click "Create API".

New API

Info

Title \*  
Checking

Name \*  
checking

Base Path  
/checking

Version \*  
1.0.0

Additional properties ^

API template  
Create API using template  
Default

Target  
Target endpoint (if known)


Security  
Identify using  
Client ID ☒ Enable CORS


Gateway  
DataPower Gateway

Cancel Add a product... Create API

Change the default “Located In” field of clientIDHeader in “Security Definitions” from “Header” to “Query” as shown below. We will be passing it in the invoke to the BankBack service.

### Security Definitions



**clientIdHeader (API Key)** 

Name \*

clientIdHeader

Parameter name \*

client\_id

Located In

Query


▼


Description

### Step 2: Add POST /checkRequest

Add a new path by clicking “+” in Paths. Name the new path: “/checkingRequest”. Add a POST operation to the path just created. Add a “body” parameter. The screen should look like this:

### Paths



**/checkRequest** 


Path \*

/checkRequest

[Add Operation](#)


Parameters

[Add Parameter](#)

Name	Located In	Description	Required	Type	
body	Body ▼		<input checked="" type="checkbox"/>	object ▼	

POST

/checkRequest





Step 3: Add Definitions for request/response.

Click Definitions on the left hand pane. Click “+” to add a new Definition. Under “Name” enter “JSONCheckRequest”. Select “Type” as object.

The screenshot shows the 'Definitions' pane in API Designer. At the top, there's a header 'JSONCheckRequest' with edit and delete icons. Below it, a form contains the following fields:

- Name \***: JSONCheckRequest
- Type**: object (dropdown menu)
- Description**: (empty text area)
- Properties**: A table with columns: Property Name, Description, Type, Example, and Actions.

Under the Properties table, there is one property: 'new-property-1' with type 'string'. At the bottom, there is a toggle switch labeled 'Allow additional properties' which is currently turned off.

We have a choice here: we can enter the attributes as properties one by one, or we can have API Designer do it for us. We will take the easy way. Open the “checkingRequest.json” file in the resources folder and copy it.

Click the “edit” icon (pencil) next to the trash can icon on the upper right corner of Definitions.

Provide a schema

The screenshot shows the 'Provide a schema' dialog box. It has four tabs: 'Schema as YAML' (selected), 'Schema as JSON', 'Generate from sample JSON', and 'Generate from sample XML'. The 'Schema as YAML' tab is active, showing a YAML schema in a text editor:

```
1 - properties:
2 -   new-property-1:
3 -     type: string
4 -   additionalProperties: false
5
```

At the bottom right of the dialog, there are two buttons: 'Done' and 'Cancel'.

Select “Generate from sample JSON” tab and paste the JSON from the checkingRequest.json file into here.

Provide a schema

Schema as YAML   Schema as JSON   **Generate from sample JSON**   Generate from sample XML

```
1 [{"PayorAccountName": "Cartoon Studios",
2   "PayorAccountID": "8458jf8757275234",
3   "CheckType": "P",
4   "Payee": "Elmer Funn",
5   "PayeeAddr": "124 East Sunset Drive; Sunset, AL 64846",
6   "PayeeRouteNo": "03849032874908",
7   "Amount": "999.00"}]
```

Generate

Done

Cancel

Click “Generate”. The screen changes to the first tab: “Schema as YAML”. Click “Done”.

Provide a schema

**Schema as YAML**   Schema as JSON   Generate from sample JSON   Generate from sample XML

```
1 description: ''
2 type: object
3 properties:
4   PayorAccountName:
5     type: string
6   PayorAccountID:
7     type: string
8   CheckType:
9     type: string
10  Payee:
11    type: string
12  PayeeAddr:
13    type: string
14  PayeeRouteNo:
15    type: string
16  Amount:
17    type: string
18 example: "  {\"PayorAccountName\": \"Cartoon Studios\",
19           \"PayorAccountID\": \"8458jf8757275234\",
20           \"CheckType\": \"P\",
21           \"Payee\": \"Elmer Funn\",
22           \"PayeeAddr\": \"124 East Sunset Drive; Sunset, AL
23           64846\",
24           \"PayeeRouteNo\": \"03849032874908\",
25           \"Amount\": \"999.00\"}"
```

Done

Cancel

Coming back to the “Definitions” screen, we see the properties defined.

Definitions +

**JSONCheckRequest** ✎ 🗑

Name \*  Type

Description  Edit Preview 🔗

Properties Add Property

*	Property Name	Description	Type	Example	Actions
<input type="checkbox"/>	<input type="text" value="PayorAccountName"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="PayorAccountID"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="CheckType"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="Payee"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="PayeeAddr"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="PayeeRouteNo"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>
<input type="checkbox"/>	<input type="text" value="Amount"/>	<input type="text"/>	<input type="text" value="string"/>	<input type="text"/>	<span>🗑</span>

☐ Allow additional properties

We need to do the same for the response.

Add a new Definition: JSONCheckResponse. Copy the contents of the checkResponse.json file into the schema editor and press “Generate”.

Provide a schema

Schema as YAML Schema as JSON Generate from sample JSON Generate from sample XML

```
1  {
2    "TransactionNum": "24943424",
3    "CheckNum": "23423",
4    "CheckType": "",
5    "PayorAccountID": "",
6    "Payee": "",
7    "Amount": ""}
```

Generate Done Cancel

Your screen should look like this:

JSONCheckResponse

Name \*

JSONCheckResponse

Type

object

Description

Edit Preview ⓘ

Properties

Add Property




*	Property Name	Description	Type	Example	Actions
<input type="checkbox"/>	TransactionNum		string		
<input type="checkbox"/>	CheckNum		string		
<input type="checkbox"/>	CheckType		string		
<input type="checkbox"/>	PayorAccountID		string		
<input type="checkbox"/>	Payee		string		
<input type="checkbox"/>	Amount		string	<input type="text"/>	

☐ Allow additional properties

Step 4: Add Web Service Definitions for back end.

The easiest way to add a service definition (interface) is to import its WSDL. We are going to do just that. Head down to “Services” on the bottom of the left pane menu. Click “+”. We want to upload the file (from the resources folder): SomeBankChecked.wsdl.

## Import web service from WSDL

<p>Upload file</p> 
<p>Load from URL</p> 
<p>Find in registry</p> 

Cancel

Click the checkbox next to “SomeBankService”. Click Done.

### Import web service from WSDL

SomeBankChecked.wsdl [Change file](#)

---

1 SOAP service found

<input checked="" type="checkbox"/> SomeBankService	<a href="#">Show operations</a>
---	---------------------------------

[Back](#)
Cancel
[Done](#)

We now see two operations under “Services”.

#### Services



Services can be loaded here for use within the implementation of this API. Any services loaded here will be available in the assembly view.

SomeBankService

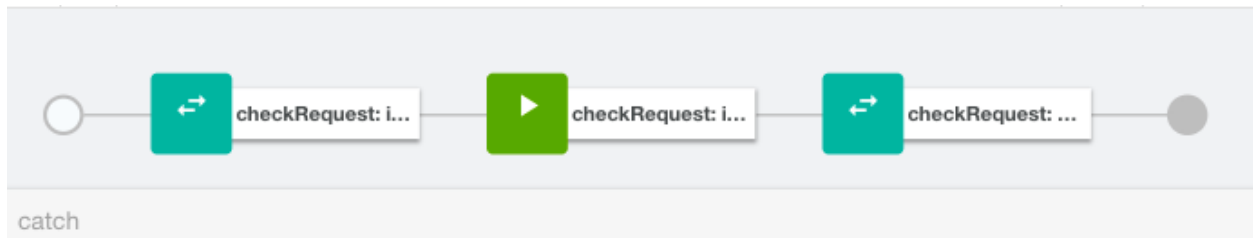


accountEnquiry

checkRequest

Step 5: Add the services to the “Assemble” section

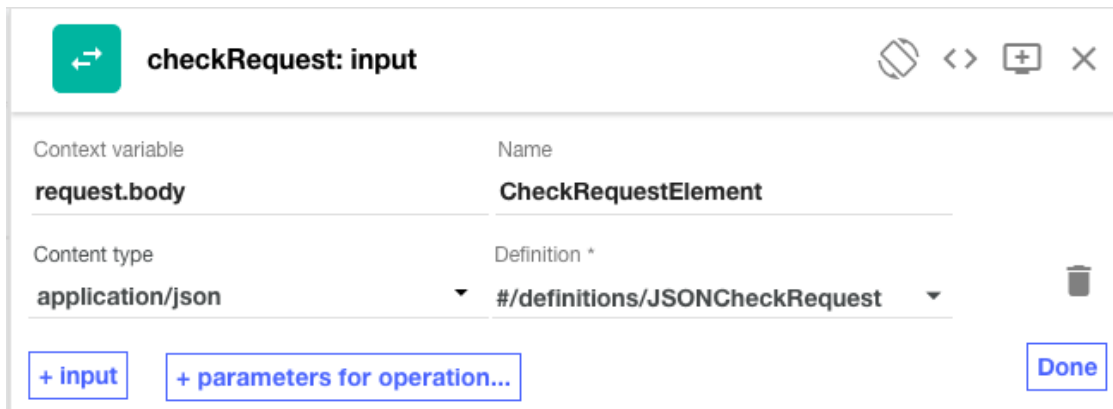
Delete the default invoke policy and drag the “checkRequest” Web Service Operation onto the canvas (bottom of left palette). It should look like this:



We now must edit the first map to take the input JSON request and convert it to SOAP and then modify the second map to take the response from BankBack service and convert it back to JSON.

## Step 6: Edit first map

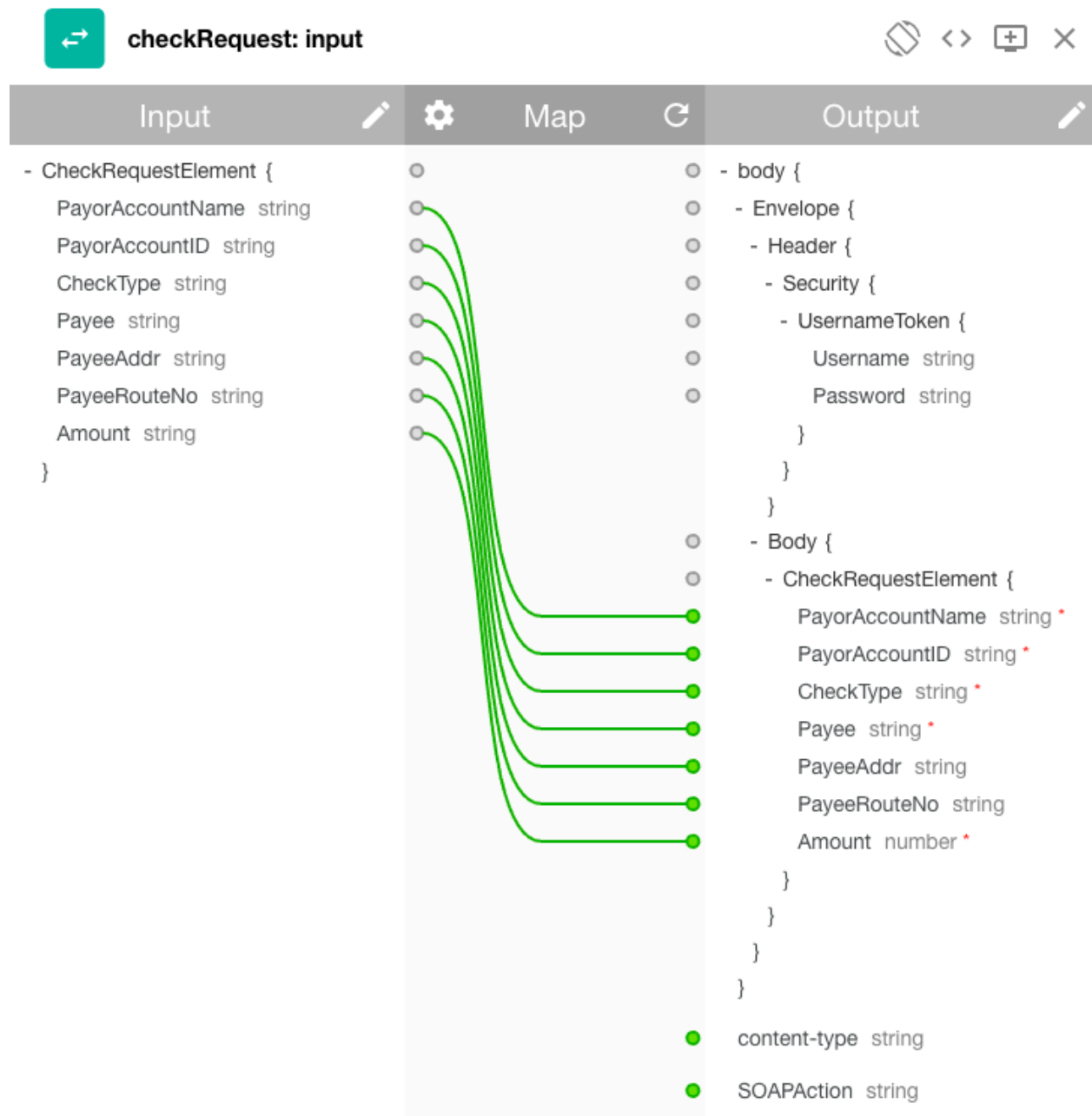
Click on the first map – the input should be empty. Click the edit icon (pencil) to add a new input definition. Click “+input” to add anew input definition. The input should look like this when you are done:



Context variable	Name
<b>request.body</b>	<b>CheckRequestElement</b>
Content type	Definition *
<b>application/json</b>	<b>#/definitions/JSONCheckRequest</b>

[+ input](#) [+ parameters for operation...](#) [Done](#)





Then map the inputs to the outputs by clicking on the Input node (PayorAccountName) and dragging it to the corresponding Output node. The screen should look like this when done.





Step 7: Edit the 2<sup>nd</sup> map






The Output map should be empty. This is the response we want to return. Click the edit (pencil) icon to create a new Output definition. We will be using JSONCheckResponse.

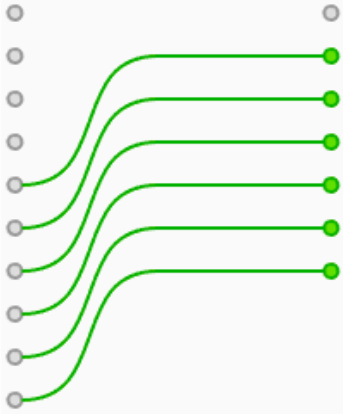
 **checkRequest: output**    

Context variable	Name
<b>message.body</b>	<b>output</b>
Content type	Definition *
<b>application/json</b>	<b>#!/definitions/JSONCheckResponse</b>

[+ output](#) [+ outputs for operation...](#) [Done](#)

When Done we must map the Input nodes to the Output nodes.

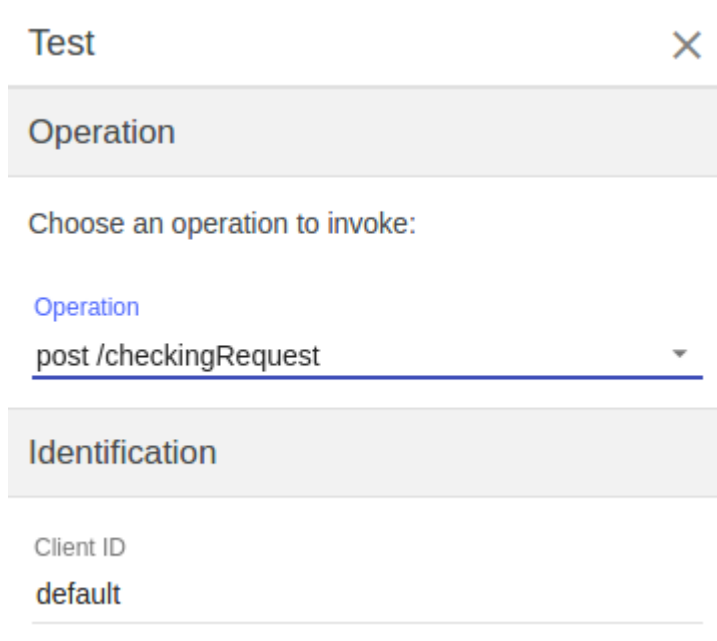
 **checkRequest: output**    

Input	Map	Output
<pre>- input {   - Envelope {     - Body {       - CheckResponseElement {         TransactionNum integer *         CheckNum integer *         CheckType string *         PayorAccountID string *         Payee string *         Amount number *       }     }   } }</pre>		<pre>- output {   TransactionNum string   CheckNum string   CheckType string   PayorAccountID string   Payee string   Amount string }</pre>

Step 8: Modify Invoke to point to the correct URL.

We need to pass the `client_id` to the BankBack service as a query parameter. The easiest way to get that is to go into the Test tool and see what the `client_id` for the test tool is. For the local gateway it is always: default.

Click the “play” (>) button to open the test tool pane. Select the POST `/checkRequest` operation and the `client_id` shows up.



The screenshot shows a window titled "Test" with a close button (X) in the top right corner. The window is divided into two main sections: "Operation" and "Identification".

**Operation**

Choose an operation to invoke:

Operation

post /checkingRequest

**Identification**

Client ID

default

We need to copy this so we can add it to the end of the routing url in the invoke policy.

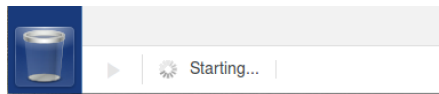
Click on the checkRequest invoke to open the pane. Change the URL field to:

**Error! Hyperlink reference not valid.**default

Where the client\_id is the client\_id copied from the test tool. Click the save icon.

Step 8: Test.

If the gateway is not running, click on the start icon at the bottom left of the screen to start it now. It should change to:



\*\*\*\*\* Please READ \*\*\*\*\*

It will take a few minutes after you click on the start icon and actually seeing the status change.

DO NOT hit the start again since it will act as the stop button and will cancel the start. So you'll have to wait until it stops and try it again.

At the end the status should look like:



Open the test tool again and select the checkRequest operation. Paste the JSON sample request from the resources folder (checkingRequest.json) into the “body” parameter on the test pane.

## Parameters

Content-Type

application/json ▼

Accept

application/json ▼

body \*

```
{ "PayorAccountName": "Cartoon Studios",  
  "PayorAccountID": "8458jf8757275234",  
  "CheckType": "P",  
  "Payee": "Elmer Funn",  
  "PayeeAddr": "124 East Sunset Drive; Sunset,  
AL 64846",  
  "PayeeRouteNo": "03849032874908",  
  "Amount": "999.00" }
```

Click "Invoke". You should see the following screen:

**Response****Status code:**

200 OK

**Response time:**

4590ms

**Headers:**

apim-debug-trans-id: 58a3672ce4b0e13332d303d4-6515d49e-9613-485b-a6c3-d1f032fb9e65

x-global-transaction-id: 25374594

content-type: application/json

**Body:**

```
{
  "TransactionNum": "25282896",
  "CheckNum": "23423",
  "CheckType": "P",
  "PayorAccountID": "8458jf8757275234",
  "Payee": "Elmer Funn",
  "Amount": "999.00"
}
```

Click on the “stop” (square) at the bottom of the screen to stop the gateway. Exit the browser and

This concludes lab 1.

## Lab 2 – creating a secure API with JWT

### Description

This lab will generate and validate a JWT token using Bluemix.

Change directory to `/home/localuser/interconnect-labs/IC17-7502-DataPower-Toolkit/lab2` in a terminal window.

---

### Step 1: import JWTAPI.yaml from the lab2 folder

On the desktop open a command prompt and navigate to the lab2 folder. In this folder you'll see the JWTAPI.yaml file.

Start the API Designer by typing: **apic edit**.

Once API Designer starts you'll see the api already loaded in the API window. Click on the api to display the API editor page. You'll notice that it has 2 paths: `/validate-jwt` and `/generate-jwt`.

Click on `generate-jwt`. It should look like this:

The screenshot shows the API Designer interface for the `/generate-jwt` endpoint. The top bar displays the path `/generate-jwt` with a trash icon. Below this, the 'Path \*' is set to `/generate-jwt`, with links for 'Add Operation' and 'Add Parameter'. The 'Parameters' section contains a table with the following data:

Name	Located In	Description	Required	Type
iss-claim	Header	Issuer claim	<input checked="" type="checkbox"/>	string
aud-claim	Header	Audience claim	<input checked="" type="checkbox"/>	string

At the bottom, there is a 'GET' button and the path `/generate-jwt` with a trash icon.

It has 2 Parameters: iss-claim (issuer) and aud-claim (audience). These are passed in in the header to generate a claim.

## Step 2: Assemble

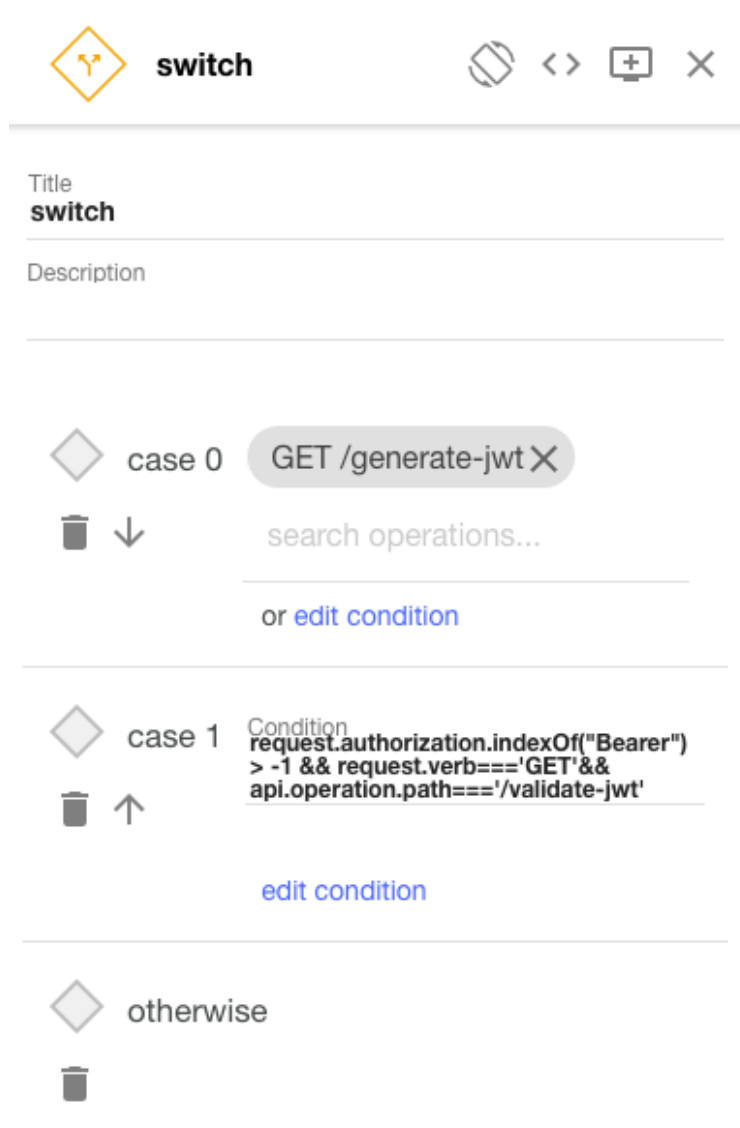
The logic here is pretty straight forward. A property variable was created to set the values of the key used to generate the JWT token. The property is encoded so that it will not be visible. The variable is hs256-key and its value is:

```
{ "alg": "HS256", "kty": "oct", "use": "sig", "k": "o5yErLaE-  
dbgVpSw65Rq57OA9dHyaF66Q_Et5azPa-  
XUjbyP0w9iRWhR4kru09aFfQLXeIODIN4uhjEIYKXt8n76jt0Pjkd2pqk4t9abRF6tnL1  
9GV4pflfL6uvVKkP4weOh39tqHt4TmkBgF2P-  
gFhgssZpjwq6l82fz3dUhQ2nkzoLA_CnyDGLZLd7SZ1yv73uzfE2Ot813zmig8KTME  
MWVcWSDvy61F06vs_6LURcq_IEEvUiubBxG5S2akNnWigfbhWYjMI5M22FOCp  
dcDBt4L7K1-  
yHt95Siz0QUb0MNIT_X8F76wH7_A37GpKKJGqesaiNWmHkgWdE8QWDQ", "kid":  
"hs256-key" }
```

This key was created using an online JWT key generator. There are other ways to create these keys.

The screenshot shows the 'Properties' dialog box in the API Connect Toolkit. The title bar says 'Properties'. Below the title bar is a light blue instruction bar: 'To replace values with one of these API properties, type \$( ) with the name of the property inside the parentheses.' The main area has a header 'hs256-key' with a trash icon on the right. Below this is a 'Property Name' field containing 'hs256-key' and a checked 'Encode' checkbox. There is a 'Description' field which is empty. Below that is a section 'Define catalog specific values for this property below' with an 'Add value' link. This section contains a table with two columns: 'Catalog' and 'Value'. The 'Default' row is highlighted, and the 'Value' cell contains a long string of asterisks representing a masked value.






On the Assemble tab, the switch statement determines the flow of the api based on either a HTTP protocol/path combination or a condition.



If the incoming request is for /generate-jwt, then the flow would do a jwt-generate policy to create the JWT token. If the incoming request is for /validate-jwt, it checks to make sure that the authorization request header has a “Bearer” token. Then it calls the jwt-validate policy. Otherwise, it sets the message.status.code to “302” and the message.status.reason to the value of the error which will be sent back to the consumer. Additionally the message.headers.Location is also entered to supply the location for the redirect. The other GWS policies are there to provide visual feedback of what has occurred during the flow.

Generate-JWT uses the hs-256-key so encrypt the token.



 **jwt-generate**    

Title

**jwt-generate**

---

Description

---

JSON Web Token (JWT)

**generated-jwt**

---

Runtime variable in which to place the generated JWT. If not set, the JWT is placed in the Authorization Header as a Bearer token.

☐ JWT ID Claim

Indicates whether a JWT ID (jti) claim should be added to the JWT. If selected, the jti claim value will be a UUID.

Issuer Claim

**request.headers.iss-claim**

---

Runtime variable from which the Issuer (iss) claim string can be retrieved. This claim represents the Principal that issued the JWT.

Subject Claim

---

Runtime variable from which the Subject (sub) claim string can be retrieved.

Audience Claim

**request.headers.aud-claim**

---

Runtime variable from which the Audience (aud) claim string can be retrieved. Multiple variables are set via a comma-separated string.

Validity Period

**36000**

---

The length of time (in seconds), that is added to the current date and time, in which the JWT is considered valid.

Private Claims

---

Runtime variable from which a valid set of JSON claims can be retrieved.

Sign JWK variable name

**hs256-key**

---

Runtime variable containing the JWK to use to sign the JWT.






Cryptographic Algorithm \*

**HS256**

The iss-claim and aud-claim is passed in as well.

#### Validate-JWT

The validate policy validates against a specific issuer and audience claim (apic/id1). This is done via the following configuration:

 **jwt-validate**    

---

Title

**jwt-validate**

Description

---

JSON Web Token (JWT)

**input-jwt**

Context or runtime variable that contains the JWT to be validated. If not set, the policy looks for the JWT in request.headers.authorization.

Output Claims

**decoded-claims**

Runtime variable to which the full set of claims that are in the JWT is assigned.

Issuer Claim

**apic**

PCRE to use to validate the Issuer (iss) claim.

Audience Claim

**id1**

PCRE to use to validate the Audience (aud) claim.

Decrypt Crypto Object

The crypto object to use to decode the claim.

Decrypt Crypto JWK variable name

Runtime variable containing the JWK to use to decrypt the JWT.

Verify Crypto Object

The crypto object to use to verify the signature.

Verify Crypto JWK variable name

**hs256-key**

Runtime variable containing the JWK to use to verify the signature.

Notice that both generate and validate share the same hs256-key.

Step 3: Test

If the gateway is not running, press the run button at the bottom of the screen.



In order to test this locally just click on the test tool icon (>) on the screen. Select the /generate-jwt operation and enter: APIC for iss-claim and id1 for aud-claim.

The result is the JWT token as shown below:

Test

✕

application/json

▼

Accept

application/xml

▼

Issuer claim

iss-claim \*

apic

Generate

Audience claim

aud-claim \*

id1

Generate

☐ Repeat

Repeat the API invocation a set number of times, or until the stop button is clicked

Stop after:

☒ Stop on error

10

.....

Invoke

Response

Status code:

200 OK

Response time:

9351ms

Headers:

apim-debug-trans-id: 58a3672ce4b0e13332d303d4-097af6f1-3cc3-4472-bec7-6142c3154bc1

x-global-transaction-id: 26051632

content-type: unknown

Body:

eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJhcGljIiwic3ViIjoiiwiYXVkJjoiaWQxIiwiaXhwIjoxNDg5MTI0MDg4LCJpYXQiOiE0ODkwODgwODh9.NfQ84arv0Y8saTC0gvMrG6bv8-Yfou9G\_-gYTf\_27CY

Debug

To test the validate, just copy the token and select the /validate-jwt operation at the top of the test tool and paste the token in the form of: Bearer XXX; where XXX is the token generated in the previous step.

Bearer <jwt-token>

Authorization \*

Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJhcGljIiwic

[Generate](#)

☐ Repeat

Repeat the API invocation a set number of times, or until the stop button is clicked

Stop after:



Stop on error

10

[Invoke](#)

## Response

Status code:

200 OK

Response time:

10557ms

Headers:

apim-debug-trans-id: 58a3672ce4b0e13332d303d4-625ccebd-b3c8-4616-bd00-226be761128f

x-global-transaction-id: 26068128

content-type: unknown

Body:

```
{
  "iss": "apic",
  "sub": "",
  "aud": "id1",
  "exp": 1489124088,
  "iat": 1489088088
}
```

[Debug](#)

As you can see we can extract the iss and aud claims from the token.