

Course Exercises Guide

Creating, Publishing, and Securing APIs with IBM API Connect

Course code WD508 / ZD508 ERC 1.0



December 2017 edition

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered 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.

© Copyright International Business Machines Corporation 2017.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Trademarks	v
Exercises description	vi
Exercise 1. Creating and publishing an API in API Designer	1-1
1.1. Review the Saving Plan REST API sample application	1-5
1.2. Create an API definition in the API designer	1-8
1.3. Invoke a GET operation in the existing API implementation	1-16
1.4. Test the API operation in the API Editor	1-19
1.5. Define a POST API operation with a JSON request message	1-24
1.6. Test the savings API in the Explore view	1-30
Exercise 2. Defining an API that calls an existing SOAP service	2-1
2.1. Review the existing SOAP web service	2-5
2.2. Create a SOAP API definition from a WSDL document	2-9
2.3. Test the API on the embedded DataPower gateway	2-17
2.4. Publish the SOAP API definition	2-23
2.5. Test the API in the API Manager (optional)	2-25
Exercise 3. Creating a LoopBack application	3-1
3.1. Verify your IBM API Connect toolkit installation	3-4
3.2. Create a LoopBack application with the apic command-line utility	3-6
3.3. Examine the structure of a LoopBack application	3-7
3.4. Review the API definition in the API Designer web application	3-14
3.5. Examine the LoopBack model and data source with API Designer	3-22
Exercise 4. Defining LoopBack data sources	4-1
4.1. Create the inventory application and MySQL data source	4-4
4.2. Create the item LoopBack model	4-8
4.3. Discover models from a MySQL data source	4-12
4.4. Test the inventory items model	4-19
4.5. Create a MongoDB data source and the review model	4-25
4.6. Define a relationship between the item and review models	4-28
Exercise 5. Implementing event-driven functions with remote and operation hooks	5-1
5.1. Modify the API base path	5-4
5.2. Create an operation hook	5-8
5.3. Test the operation hook	5-11
5.4. Create a remote hook	5-16
5.5. Test the remote hook	5-22
Exercise 6. Assembling message processing policies	6-1
6.1. Create the financing API definition	6-5
6.2. Create an API operation in the financing API	6-10
6.3. Invoke a SOAP web service with a message processing policy	6-13
6.4. Create the logistics API definition	6-24
6.5. Define message processing policies for the logistics API operations	6-29
6.6. Define the message policies for the GET /shipping API operation	6-32
6.7. Define the message policies for the GET /stores API operation	6-39
6.8. Test the financing and stores API policy assemblies in the DataPower gateway	6-42

Exercise 7. Declaring an OAuth 2.0 Provider and security requirement	7-1
7.1. Create an OAuth 2.0 Provider API	7-4
7.2. Configure OAuth 2.0 authorization in the inventory API	7-13
Exercise 8. Deploying an API implementation to a container runtime	8-1
8.1. Test a local copy of the LoopBack API application	8-5
8.2. Set up the Docker image configuration	8-8
8.3. Load the inventory image to a local Docker registry	8-12
8.4. Deploy the LoopBack application on the Docker image to a swarm	8-13
8.5. Deploy the LoopBack application on the Docker image to a swarm	8-14
8.6. Test the LoopBack application on the Docker swarm	8-16
8.7. Delete the service that is running on the Docker swarm	8-18
Exercise 9. Defining and publishing an API product	9-1
9.1. Change the invoke URL so that it routes to the Docker image and port number	9-5
9.2. Modify the product properties and add the APIs to a product plan	9-10
9.3. Enable API key security in the financing and logistics API	9-17
9.4. Publish an API product and API definitions	9-20
9.5. Review the product in the API Manager server	9-22
Exercise 10. Subscribing and testing APIs	10-1
10.1. Register a client application in the Developer Portal	10-5
10.2. Subscribe to a plan for the inventory API product	10-10
10.3. Test a subscribed API in the test client	10-13
10.4. Update the client application with the client ID and client secret	10-18
10.5. Test a subscribed API with a sample application	10-20
Exercise 11. Troubleshooting the case study	11-1
11.1. Review the inventory items API operation	11-3
11.2. Test the OAuth authorize and token API operations	11-4
11.3. Verify the OAuth 2.0 Provider API configuration	11-9
11.4. Publish API definition changes to the API Manager	11-16
11.5. Verify the consumer application credentials and plan subscription	11-20
11.6. Test the inventory items API operation	11-26
11.7. Test the inventory API in the test client of the Developer Portal	11-32
11.8. Test the inventory API with the consumer application	11-39
Appendix A. Solutions	A-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

Bluemix®

DB™

IBM Bluemix™

Cloudant®

Express®

IMS™

DataPower®

IBM API Connect™

Notes®

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

LoopBack® and StrongLoop® are trademarks or registered trademarks of StrongLoop, Inc., an IBM Company.

Social® is a trademark or registered trademark of TWC Product and Technology, LLC, an IBM Company.

Other product and service names might be trademarks of IBM or other companies.

Exercises description

This course includes the following exercises:

- [Exercise 1, "Creating and publishing an API in API Designer"](#)
- [Exercise 2, "Defining an API that calls an existing SOAP service"](#)
- [Exercise 3, "Creating a LoopBack application"](#)
- [Exercise 4, "Defining LoopBack data sources"](#)
- [Exercise 5, "Implementing event-driven functions with remote and operation hooks"](#)
- [Exercise 6, "Assembling message processing policies"](#)
- [Exercise 7, "Declaring an OAuth 2.0 Provider and security requirement"](#)
- [Exercise 8, "Deploying an API implementation to a container runtime"](#)
- [Exercise 9, "Defining and publishing an API product"](#)
- [Exercise 10, "Subscribing and testing APIs"](#)
- [Exercise 11, "Troubleshooting the case study"](#)

In the exercise instructions, you can check off the line before each step as you complete it to track your progress.



Hint

If you are unable to complete an exercise, you can copy the model solution application from the **lab files** directory (/home/student/lab_files). See [Appendix A, "Solutions"](#) for instructions on how to access the solution code and application for each exercise.



Important

Online course material updates might exist for this course. To check for updates, see the Instructor wiki at: <http://ibm.biz/CloudEduCourses>

Exercise 1. Creating and publishing an API in API Designer

Estimated time

01:00

Overview

This exercise covers how to define an API interface from an existing API. You review the API operations, parameters, and data types from the API Designer web application. You also publish and test the API from the API Designer web application.

Objectives

After completing this exercise, you should be able to:

- Create an API definition in the API Designer
- Review the operations, properties, and data types in an API definition
- Publish an API to the API Connect development environment
- Start an API and the micro gateway
- Test an API with the API Explorer

Introduction

You can define APIs with one of two approaches:

- In an **interface-first** design, you define each API path, operation, request, and response message in an OpenAPI definition. You map the interface to an existing API implementation that is deployed in your architecture.
- In an **implementation-first** design, you build an API implementation as a collection of models, properties, relationships, and data sources. You generate a set of REST API operations from the models to an OpenAPI definition.

In this exercise, you build an OpenAPI definition based on the business requirements. The **savings plan** API provides a set of financial calculators to help customers estimate the potential return on their savings accounts. After you test a working API implementation, you independently design a set of API operations in the API Designer. You map the operations to the API implementation with an invoke message policy. Last, you test the API definition in the micro gateway.

Requirements

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.



Important

The exercises in this course use a set of lab files that might include scripts, applications, files, solution files, PI files, and other files. You can find the course lab files in the `/home/student/lab_files/` directory in the Xubuntu VM.

The exercises point you to the lab files as you need them.

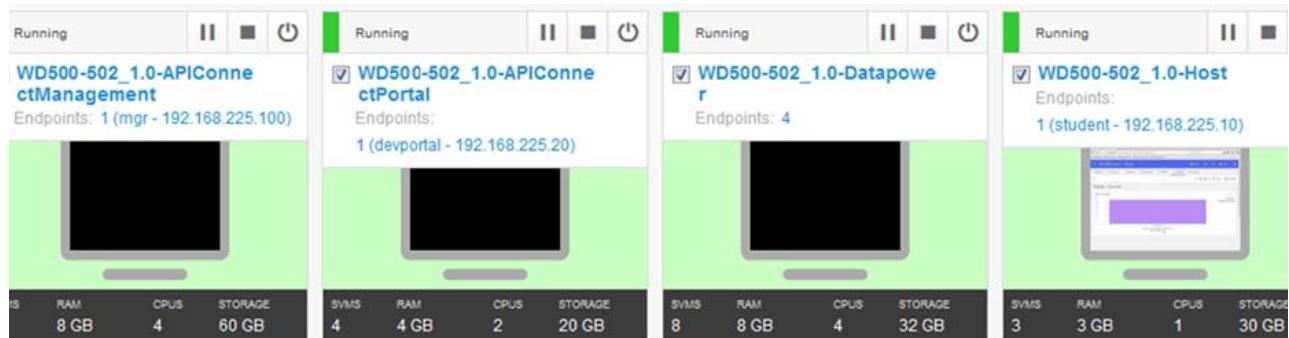
Exercise instructions

Before you begin

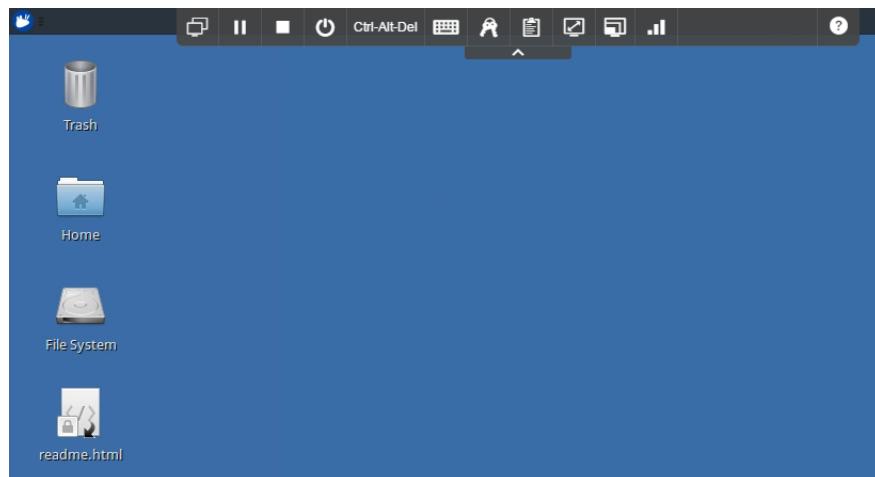
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



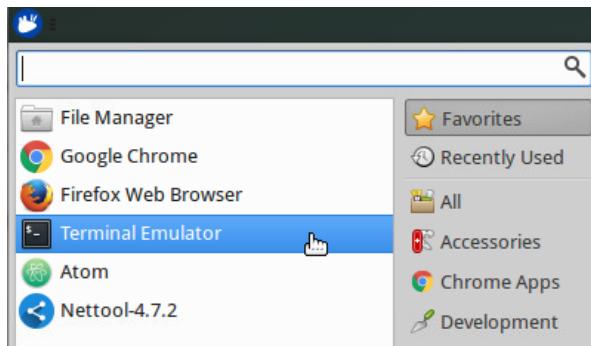
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

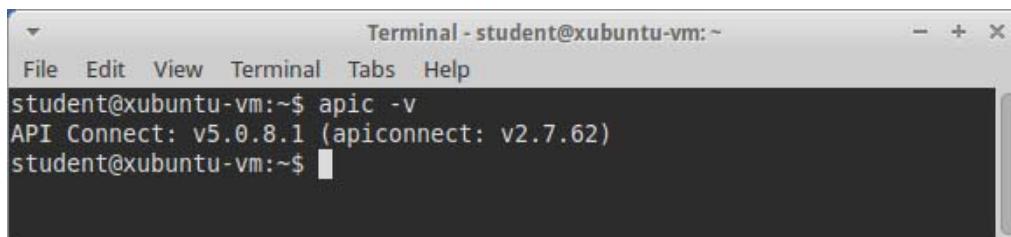
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

-
- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

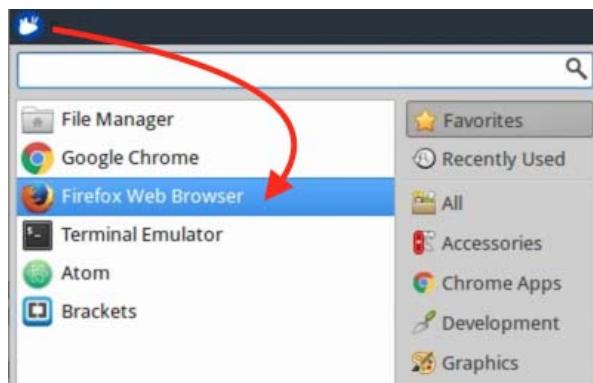
Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

1.1. Review the Savings Plan REST API sample application

The **Savings Plan** REST API projects a savings account balance. The **GET /Plans/estimate** operation calculates the savings account balance after several years of investment at a fixed interest rate. The operation assumes that you deposit the same amount at the end of each year.

In this section, review the operation of the Savings Plan API. Examine the HTTP request parameters for the API operation, and the structure of the API response message. You use this information to map your API definition to this service.

- __ 1. Open the **Savings Plan** REST API site.
 - __ a. Open a web browser from the main menu.



- __ b. Open the <http://savingsample.mybluemix.net> site.

Welcome to the Savings Plan REST API application

This application hosts a sample REST API that is written in the LoopBack API framework.

The **GET /Plans/estimate** operation calculates the savings account balance for a fixed interest rate and annual deposits over a number of years.

- __ 2. Test the **GET /Plans/estimate** API operation.
 - __ a. Scroll down the page.

___ b. In the test client, enter the following values:

- **Deposit:** 300
- **Interest rate:** 0.04 (to represent 4%)
- **Years to invest:** 20

How much will you deposit at the end of each year?
300

What is the annual interest rate?
0.04

How many years do you want to invest your savings?
20

Calculate

___ c. Click **Calculate**.

___ d. Review the results.

The balance is 8933.42 after investing 300 per year over 20 years at an annual interest rate of 4%.

[View API response message](#)



Information

The web page calls the REST API operations with the deposit, rate, and number of years that you typed into the form. The page takes the response from the API call and writes a result on the page.

In this example, you deposited \$300 into a savings account at the end of each year for 20 years. The account earns an annual interest rate of 4%, compounded annually. At the end of 20 years, the savings account balance is \$8,933.42.

___ 3. Review the raw HTTP request and response messages for the **GET /Plans/estimate** API operation.

___ a. Select **View API response message**.

___ b. Examine the API request message source.

← → ⌂ savingsample.mybluemix.net/api/Plans/estimate?deposit=300&rate=0.04&years ☆

```
{"balance":8933.42}
```



Information

The network path for the API operation is:

`http://savingsample.mybluemix.net/api/Plans/estimate`

The operation expects the input parameters as query parameters. As a GET method call, the HTTP request message body is empty. The API returns a response message as a JSON object.

__ 4. Test the **GET /Plans/estimate** API operation from your workstation with the cURL utility.

- __ a. Open a **Terminal Emulator** application window.
- __ b. Make an HTTP GET request to the `/Plans/estimate` path.

```
$ curl -v
"https://savingsample.mybluemix.net/api/Plans/estimate?deposit=300&rate=0
.04&years=20"
```

- __ c. Examine the result from the API operation call.

```
* Connected to savingsample.mybluemix.net (75.126.81.66) port 443 (#0)
> GET /api/Plans/estimate?deposit=300&rate=0.04&years=20 HTTP/1.1
> Host: savingsample.mybluemix.net
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Backside-Transport: OK OK
< Connection: Keep-Alive
< Transfer-Encoding: chunked
< Access-Control-Allow-Credentials: true
< Content-Type: application/json; charset=utf-8
< Date: Tue, 26 Dec 2016 02:46:49 GMT
< Etag: W/"13-TLAN43I7RBZKWr0BbInEvv
< Vary: Origin, Accept-Encoding
<
* Connection #0 to host savingsample.mybluemix.net left intact
{"balance":8933.42}
```



Information

The **cURL** utility is a widely customizable, third-party application for HTTP network testing. In this example, you sent an HTTP GET request to the **GET /Plans/estimate** API operation. The `-v` verbose parameter displays the headers in the HTTP request, and the headers and body from the HTTP response message.

In this course, you use a combination of command-line utilities and web application clients to test API operations. For more information about cURL utility, see: <https://curl.haxx.se/>

1.2. Create an API definition in the API designer

The API Designer is a web application that is hosted locally on your own workstation. This application is part of the API Connect toolkit. With API Designer, you can quickly import, define, test, and publish API definitions and implementations to your API Connect Cloud.

In this section, create an API definition with the API Designer web application. Define an API path that is named **/Plans/estimate** with one operation: a **GET** method on the same path.

— 1. Open the API Designer application.

— a. In the **terminal emulator** window, create a directory that is named **savings** in the home directory.

```
$ mkdir ~/savings
$ cd ~/savings
$ pwd
/home/student/savings
```

— b. Start the API Designer application.

```
$ apic edit
```



Note

If the API Designer does not start automatically in the browser, in the Firefox address area, type:

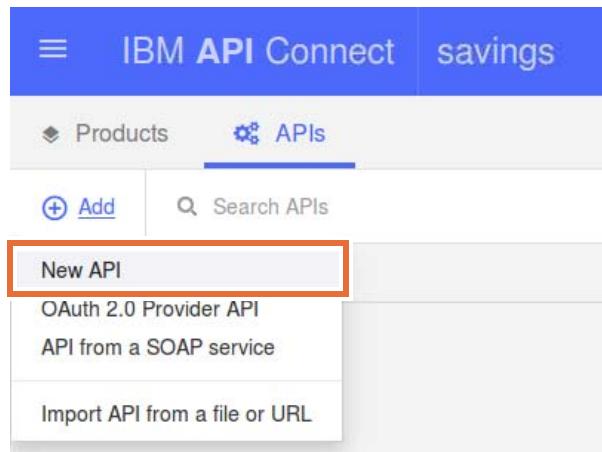
127.0.0.1:9000/#/design/apis

The API Designer should be displayed.

— 2. Create an OpenAPI API definition from scratch.

— a. From the API Designer main page, click the **API** tab.

— b. Select **Add > New API**.



— c. Enter the following properties for the API definition:

- Title: **savings**
- Name: **savings**
- Base Path: **/savings**
- Version: **1.0.0**

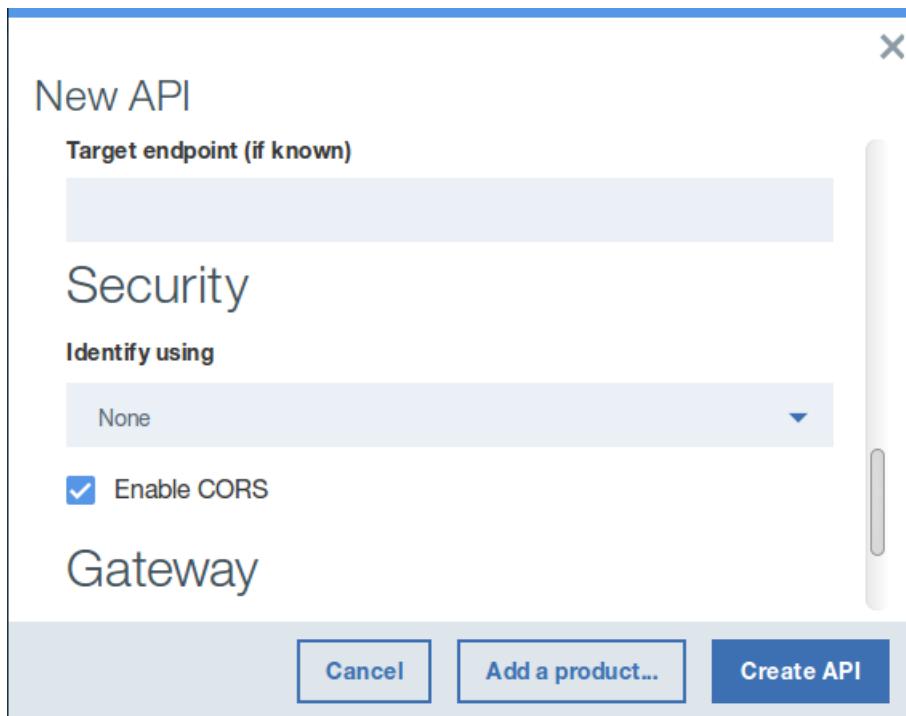
New API

Info	Title *
	savings
	Name *
	savings
	Base Path
	/savings
	Version *
	1.0.0

Additional properties ▾

- ___ d. Select the **additional properties** section and expand it.
- ___ e. Select Default in the **Create API using template** option.
- ___ f. Leave the **target endpoint** property empty.

- __ g. Change the **security** setting to identify by **none**.



- __ h. Select **Create API**.
- __ 3. Examine the **savings** API definition in the API Editor.
- __ a. Select the Info section.
 - __ b. Review the API description metadata: name, title, and version.
 - __ c. Enter a **description** that states Savings plan estimator to calculate growth from compound interest.

Info	
Title *	savings
Name *	savings
Version *	1.0.0
Description	Savings plan estimator to calculate growth from compound interest

- ___ d. Examine the **Host** and **Base Path** sections.

Host	Host *
	\$(catalog.host)
Base Path	Base Path
	/savings



Information

The **host** is the network address of the server that hosts the API definition. By default, API Designer inserts the runtime property **\$(catalog.host)**, the network endpoint at the API gateway that represents the API catalog. When you publish the API definition to the API Management server, the server replaces **\$(catalog.host)** with the actual network address.

The **base path** represents the network path immediately after the host name. For example, the address of the API gateway is `http://api.think.ibm`. In this case, the entry point into the savings API is `http://api.think.ibm/savings`.

- ___ e. Examine the **consumes** and **produces** sections.

Consumes application/json application/xml

Additional media types

Add media type

Produces application/json application/xml



Information

The **consumes** describes the media types that are used in the HTTP request message body and **produces** describes the media types that are used in the HTTP response message body. In this example, the API operations expect the message body in an **application/json** data format. Conversely, the API operations return response messages in an **application/json** format also.

The **consumes** and **produces** settings are the API-wide default values. You can override these settings at the operation level.

- ___ 4. Create a path that is named **/plans**.
 - ___ a. Select the **paths** section.
 - ___ b. Select the **plus (+)** icon to create a plan.
 - ___ c. Change the name of the path to **/Plans/estimate**.

The screenshot shows the 'Paths' section of the API Designer interface. A single path entry, '/Plans/estimate', is listed. To the right of the entry is a small trash can icon for deletion. At the bottom right of the list area is a blue button labeled 'Add Operation'.

- ___ 5. Define three parameters for the **/plans** path: **deposit**, **rate**, and **years**.
 - ___ a. Click in the **/Plans/estimate** path to expand it. Then, select the **GET /Plans/estimate** operation to open its property.
 - ___ b. In the **Description** field, enter **Calculate savings growth with annual compounding**.

The screenshot shows the properties for the 'GET /Plans/estimate' operation. At the top, it displays the method ('GET'), the path ('/Plans/estimate'), and a 'Deprecated' status with a trash can icon. Below this are several sections: 'Add Tag' (with a large empty input field), 'Summary' (with a large empty input field), 'Operation ID' (with a large empty input field), and 'Description' (containing the text 'Calculate savings growth with annual compounding').

- ___ c. Select the **Add parameter** link three times.

- ___ d. Enter the request parameter values according to the table.

Add Parameter +				
NAME	LOCATED IN	DESCRIPTION	REQUIRED	TYPE
deposit	Query	Annual deposits	<input checked="" type="checkbox"/>	number-float
rate	Query	Interest rate	<input checked="" type="checkbox"/>	number-float
years	Query	Years of saving	<input checked="" type="checkbox"/>	integer-int64

Table 1. GET /Plans/estimate API operation parameters

Name	Located in	Description	Required	Type
deposit	Query	Annual deposits	Yes	number-float
rate	Query	Interest rate	Yes	number-float
years	Query	Years of saving	Yes	integer-int64

- ___ 6. Define a return data type for the GET /Plans/estimate operation, named **plan-result**.
- ___ a. Select the **Definitions** section of the API editor.
 - ___ b. Select **plus (+)** to create a schema definition.
 - ___ c. Change the name of the schema definition type to **plan-result**.
 - ___ d. Set the description to **Calculated result from savings plan**.

Definitions

+

plan-result	<>	trash
Name *	plan-result	
Type	object	
Description	Edit Preview i	
Calculated result from savings plan		

- ___ e. In the **plan-result** definition properties, define the **balance** property according to the table.

*	PROPERTY NAME	DESCRIPTION	TYPE	EXAMPLE	ACTIONS
<input type="checkbox"/>	balance	Account balance	float ▾	8933.42	

Table 2. Plan-result schema type definition

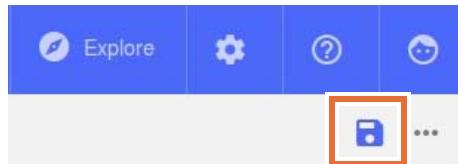
Property name	Description	Type	Example
balance	Account balance	float	8933.42

- ___ 7. Set the **response** from the **GET /Plans/estimate** operation to return the **plan-result** object.
- ___ a. Select the **paths** section.
 - ___ b. Expand the **GET /Plans/estimate** API operation.
 - ___ c. In the **responses** section, locate the status code **200** response.
 - ___ d. Delete the status code 200 by clicking the Trash icon.
 - ___ e. Click Add Response.
 - ___ f. Change the responses return type **status code** to 200, **description** to 200 OK, and **schema** to **plan-result**.

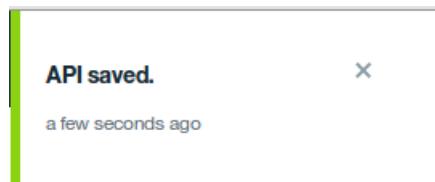
Responses

Add Response +		
STATUS CODE	DESCRIPTION	SCHEMA
200	200 OK	plan-result ▾

- ___ 8. Save the changes to the **savings** API definition.
- ___ a. In the upper-right section of the page, click the **save** option.



- __ b. Confirm that the editor saved the file successfully.



1.3. Invoke a GET operation in the existing API implementation

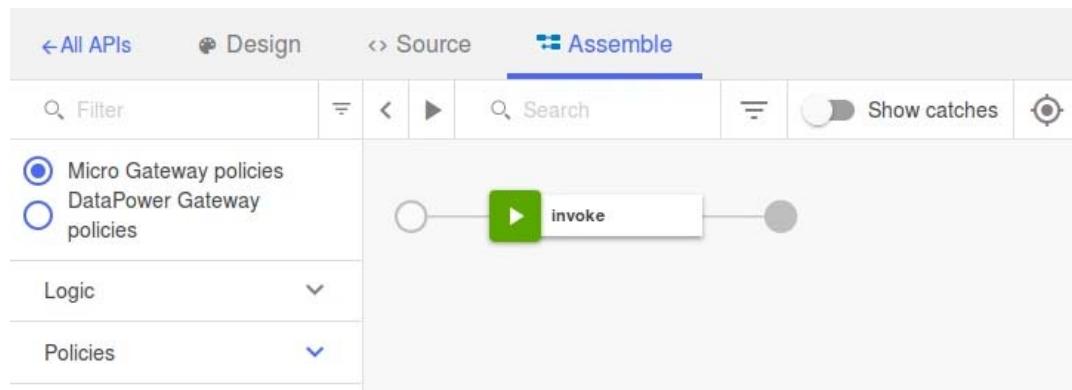
You created an API definition: the interface for the REST API. However, the interface describes the structure of the request and response messages only. It does not implement the actual API operation.

In this section, you send the API operations that you defined in the API definition to the existing **Savings Plan** REST API application. Modify the **invoke** properties to call the **GET /Plans/estimate** operation in the **Savings Plan** API. Map the response message to the **plan-result** object in the **savings** API definition.

- 1. Open the **Assemble** view in the API editor.
- a. In the **savings** API definition, click the **Assemble** tab.



- b. Examine the assemble view in the API editor.



Information

What is the **assemble** view?

The assemble view is a **graphical editor** that defines a **sequence of message processing policies**. The **API gateway transforms and routes API messages** based on these policies. The policies apply to all API operations in the API definition.

-
- 2. Change the **invoke** policy to call the **GET /Plans/estimate** API operation at:
`http://savingssample.mybluemix.net/api/`
 - a. Click the **invoke** policy in the pipeline to open the **properties** view.

- ___ b. Examine the **URL** field.



Information

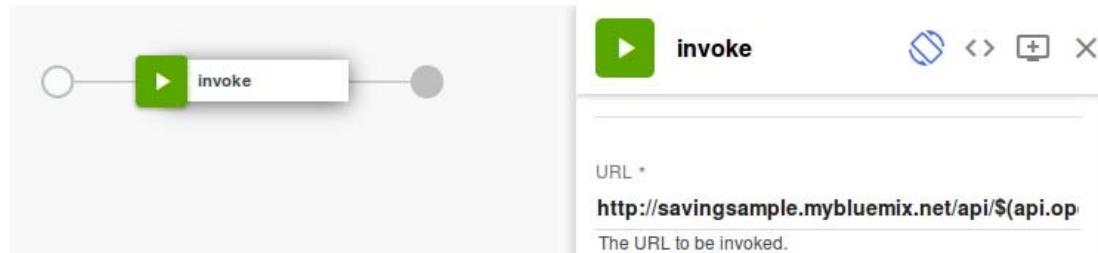
When the API gateway receives a request for the **GET /plans** API operation, it routes the request to the message processing policy. The **invoke** operation calls the implementation of the API, at `$(target-url)${request.path}`.

The **target-url** is an API property that represents the host name of the API implementation. This value is different from the **host** variable, which represents the entry point for the API: the API gateway.

The **request.path** property represents the web path from the original HTTP request message. In this example, the **request.path** is **/savings/plans**.

- ___ c. Modify the **URL** to:

```
http://savingsample.mybluemix.net/api/${api.operation.path}?${request.QueryString}
```



Information

The **api.operation.path** property is “/Plans/estimate”, while the **request.path** property is “/savings/Plans/estimate”. The **request.path** property includes the base path.

The **request.QueryString** property is “deposit=300&rate=0.04&years=20”. The query string does not include the question mark (?) character.

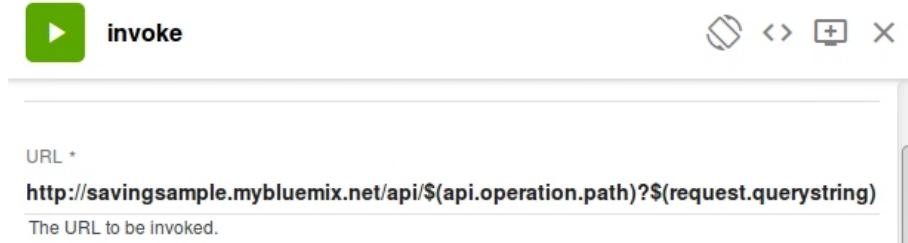
For more information, see the IBM Knowledge Center for API Connect and use the search string “API Connect context variables”.

-
- ___ d. Click **Toggle info panel between right and bottom**.



- ___ e. Confirm that the **URL** matches the value

`http://savingsample.mybluemix.net/api/${api.operation.path}?${request.QueryString}.`



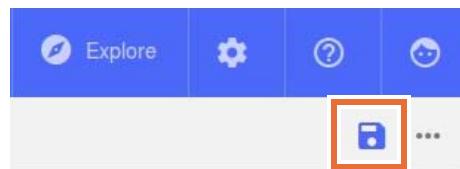
Note

In this exercise, you hardcode the network address for the API operation in the **invoke** policy. This approach works, but it is not flexible when you have different API paths in your API.

In later exercises, you learn how to substitute a runtime variable for the target URL with an API property.

-
- ___ 3. Save the changes to the **savings** API definition.

- ___ a. In the upper-right section of the page, click the **save** option.



- ___ b. Confirm that the editor saved the file successfully.

1.4. Test the API operation in the API Editor

The API Designer includes two test clients for your API operations:

- In the **assemble** view, you can quickly invoke any operation in the API definition that you opened. For example, you can test the **GET /Plans/estimate** operation from the **savings** API.
- In the **explore** view, you can review **any** API that you published to the micro gateway.

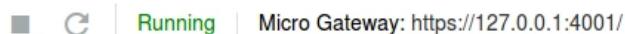
In this section, test the API operation from the test client in the **assemble** view. When you confirmed that your API works correctly, you can make further changes to the API definition and policy assembly.

— 1. Start the **Micro Gateway**.

— a. From the bottom of the API Designer web page, click **start**.

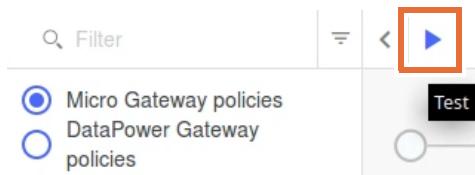


— b. Confirm that the **Micro Gateway** is in the **started** state.

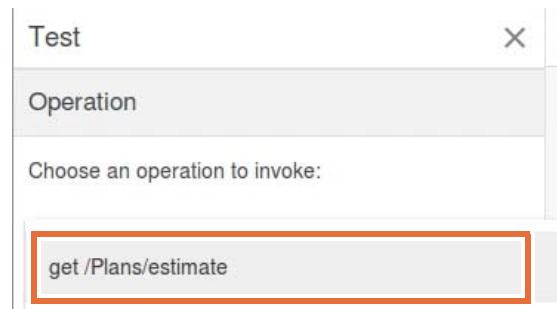


— 2. Test the **get /Plans/estimate** operation within the **assemble** view in the API Editor.

— a. In the **assemble** view, click the **test** icon. The icon is between the **filter** and **search** fields.



— b. Select the **get /Plans/estimate** operation in the test client.



- ___ c. Leave the request **Content-Type** and **Accept** headers to **application/json**.

Test X

Operation

Choose an operation to invoke:

Operation
get /Plans/estimate

Parameters

Content-Type
application/json

Accept
application/json

- ___ d. Enter the following API operation parameters:

- deposit: 300
- rate: 0.04
- years: 20

Accept
application/json

Annual deposits
deposit *
300

Generate

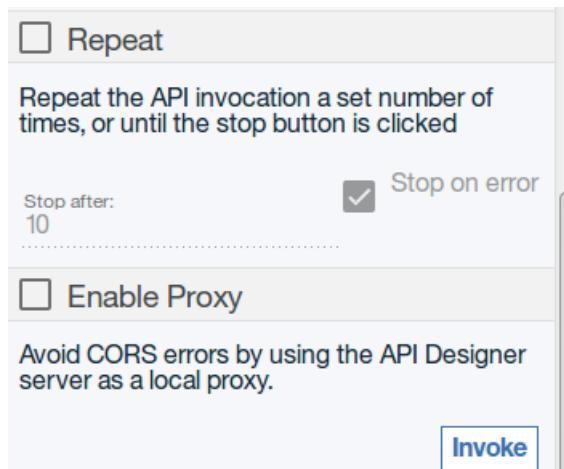
Interest rate
rate *
0.04

Generate

Years of saving
years *
20

Generate

- ___ e. After the parameters, you can repeat the API call multiple times. Leave the Repeat check box cleared and the Enable Proxy box cleared.



- ___ f. Click **Invoke**.
___ g. Examine the result from the API operation call.

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://127.0.0.1:4001/savings/Plans/estimate?deposit=300&rate=0.04&years=20>

Response time:
98ms



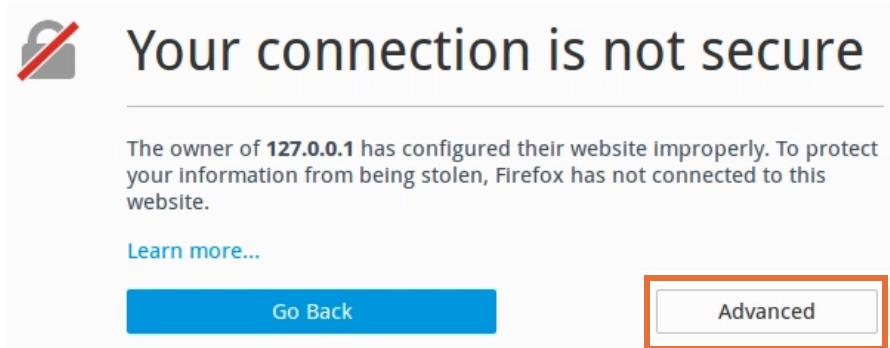
Information

Why did the test client API call fail?

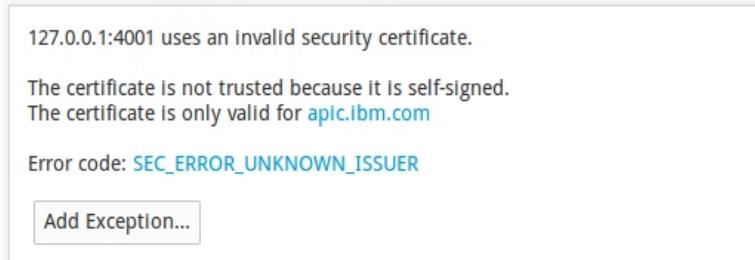
The API Designer web application made an HTTP request to the micro gateway at **localhost:4001**. The micro gateway uses a self-signed security certificate for SSL/TLS traffic. Web browsers do not accept self-signed security certificates.

To fix this issue, you must create a security exception for **localhost:4001** in your web browser. Accept the self-signed security certificate for the micro gateway with a security exception.

-
- ___ 3. Accept the **security exception** from the web browser.
 - ___ a. In the **Responses** section, click the link for the **Savings** API at the micro gateway.
 - ___ b. Click **Advanced**.



- ___ c. Click **Add Exception**.



__ d. Click **Confirm Security Exception**.



- __ 4. Invoke the **GET /Plans/estimate** API operation at the micro gateway again.
- __ a. In the **Assemble** view API test client, click **Invoke**.
 - __ b. Confirm that the micro gateway returns the correct **balance** value in the Response section.

Response

Status code:
200 OK

Response time:
162ms

Headers:

```
content-type: application/json; charset=utf-8
x-global-transaction-id: 2493533801
x-ratelimit-limit: 100
x-ratelimit-remaining: 98
```

Body:

```
{
  "balance": 8933.42
}
```

1.5. Define a POST API operation with a JSON request message

The **Savings Plan** REST API sample application also provides a **POST /Plans/estimate** operation. The operation accepts the **deposit**, **rate**, and **years** input parameters as a JSON object in the HTTP request message:

```
{ "deposit": 300, "rate": 0.04, "years": 20 }
```

In this section, create an API operation that is named **POST /Plans/estimate** in your **savings** OpenAPI definition. Define a schema object that is named **plan** with the **deposit**, **rate**, and **years** properties. Add an **invoke** policy to make a **POST** request to the API implementation at **savingsample.mybluemix.net**.

- ___ 1. Create a schema definition named **plan**.
 - ___ a. In the **savings** API definition, switch to the **Design** view.
 - ___ b. Select the **Definitions** section.
 - ___ c. Select **add (+)** to create a schema definition.
 - ___ d. Change the schema definition name to: **plan**

Name *	Type
plan	object

Description

Parameters for a savings plan estimate calculation

Edit Preview ⓘ

- ___ e. Change the description to: **Parameters for a savings plan estimate calculation**

- ___ f. In the **Properties** section, add the **deposit**, **rate**, and **years** properties.

Parameters for a savings plan estimate calculation					
Properties					Add Property
*	Property Name	Description	Type	Example	Actions
<input type="checkbox"/>	deposit	Annual deposit	float	300.00	< > 
<input type="checkbox"/>	rate	Interest rate	float	0.04	< > 
<input type="checkbox"/>	years	Duration in years	integer	20	< > 

Table 3. Plan definition properties

Property name	Description	Type	Example
deposit	Annual deposit	float	300.00
rate	Interest rate	float	0.04
years	Duration in years	integer	20

- ___ 2. Create the **POST /Plans/estimate** API operation.

- ___ a. Select the **paths** section.



The screenshot shows the 'Paths' section of the API Designer. It lists a single path: '/Plans/estimate'. Below it, there is a placeholder 'Path *' followed by another '/Plans/estimate'. To the right of this second entry is a blue button labeled 'Add Operation' with a plus sign icon above it. There is also a small trash can icon next to the second path entry.

- ___ b. In the **/Plans/estimate** API path, click **Add Operation**.

- ___ c. Select **POST**.

- ___ d. In the **POST /Plans/estimate** operation, enter **Calculate savings growth with annual compounding** as the description.

The screenshot shows the API Designer interface for creating a POST operation at the path '/Plans/estimate'. The 'Description' field is highlighted with a red box and contains the text 'Calculate savings growth with annual compounding'.

- ___ e. Add the **plan** object as the input parameter.

Parameters

Add Parameter +				
NAME	LOCATED IN	DESCRIPTION	REQUIRED TYPE	
plan	Body	savings plan estimate	<input checked="" type="checkbox"/>	<input type="text" value="plan"/>

Table 4. POST /Plans/estimate input parameters

Name	Located in	Description	Required	Type
plan	Body	Savings plan estimate details	Yes	plan

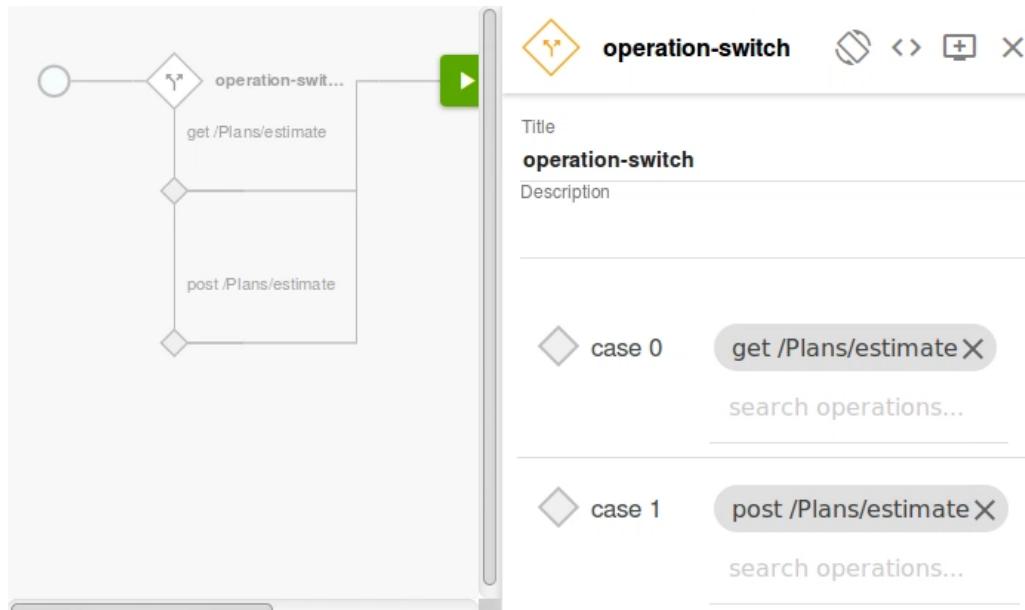
- ___ f. In the **200 OK** response, select **plan-result** as the response message schema type.

Responses		Add Response
Status Code	Description	Schema
200	200 OK	<input type="text" value="plan-result"/>

- ___ 3. Save the changes to the **savings** API definition.
- ___ 4. Create an **operation-switch** policy to handle two types of API operation request: **GET /Plans/estimate** and **POST /Plans/estimate**.
 - ___ a. Select the **Assemble** view.
 - ___ b. Add an **operation-switch** policy at the beginning of the message processing pipeline by selecting the **Operation Switch** from the palette, and then drop it between the start and invoke icons on the free-form area.



- ___ c. In the **properties** editor for the **operation-switch** policy, set the **case 0** branch to: **get /Plans/estimate**
- ___ d. Select **Add case**.
- ___ e. Set the **case 1** branch to: **post /Plans/estimate**

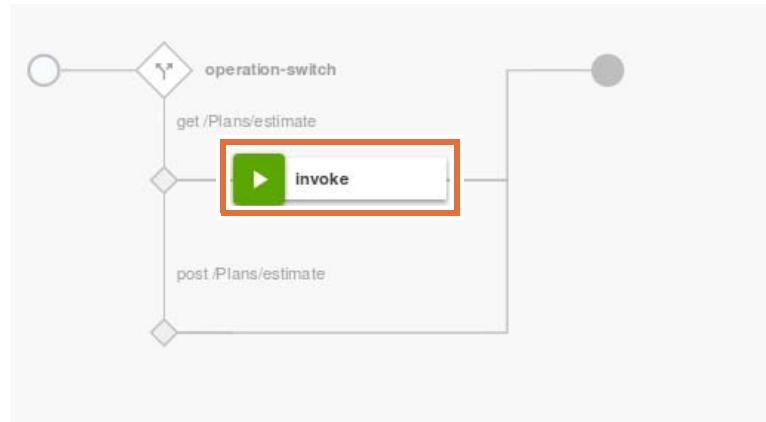


- ___ f. Close the operation-switch properties editor.
- ___ 5. Add the **invoke** policy to the **get /Plans/estimate** case.
- ___ a. Select the existing **invoke** policy in the assembly palette.

- ___ b. Drag the existing **invoke** policy over the **get /Plans/estimate** case.



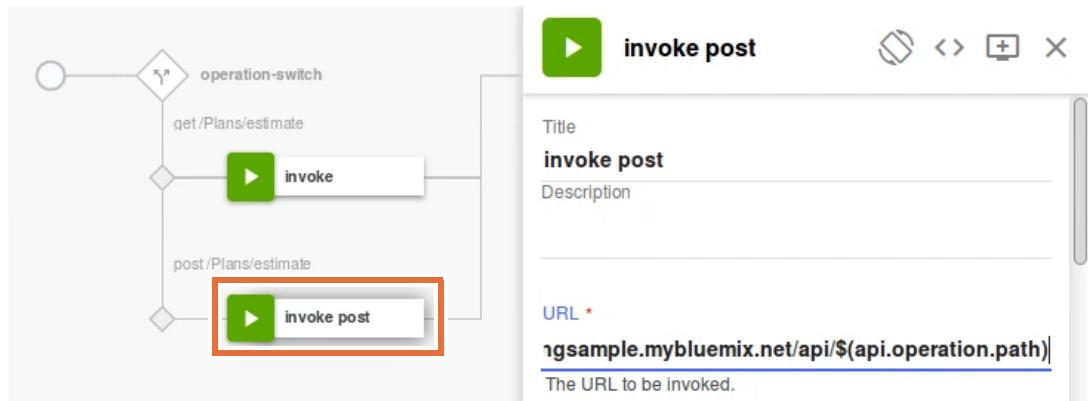
- ___ c. Drop the existing **invoke** policy in the highlighted section in the **get /Plans/estimate** case.



- ___ 6. Add an **invoke** policy in the **post /Plans/estimate** case.

- ___ a. From the policy palette, select the **invoke** policy.
- ___ b. Drag the **invoke** policy icon into the **post /Plans/estimate** case. This case is the second entry in the **operation-switch** construct.
- ___ c. In the **properties** editor, change the title to: **invoke post**

- ___ d. Set the URL to: `http://savingsample.mybluemix.net/api/${api.operation.path}`



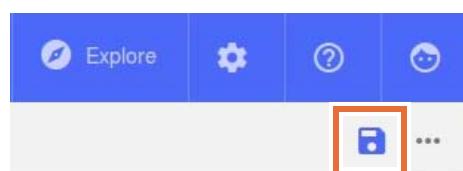
- ___ e. Scroll down to the **method** section.
___ f. Set the invoke method to **POST**.



- ___ g. Scroll down the properties page.
___ h. Clear the **Stop on error** check box.



- ___ 7. Save the changes to the **savings** API definition.
___ a. In the upper-left corner of the page, click the **save** option.



- ___ b. Confirm that the editor saved the file successfully.

1.6. Test the savings API in the Explore view

The **Explore** view is a test client within API Designer. You can review documentation and sample code on each operation in your API.

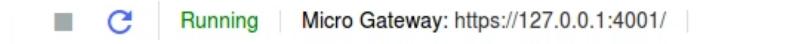
In this section, review the documentation on the **GET** and **POST /Plans/estimate** API operations. Test the operation from the built-in test client. Examine and test the sample code on your workstation.

- 1. Restart the **Micro Gateway**.

- a. From the bottom of the API Designer web page, click **restart**.



- b. Confirm that the Micro Gateway is in the **running** state.



- 2. Review the **savings** API definition in the **Explore** view.

- a. From the API Designer page, select the **Explore** view.



- b. Examine the **Explore** view.

savings 1.0.0		savings 1.0.0		Contact Information	
Operations		Description	Savings plan estimator to calculate growth from compound interest	Thomas Watson thomas@think.ibm www.ibm.com	
GET /Plans/estimate	POST /Plans/estimate				
Definitions					
plan					
plan-result					
		GET /Plans/estimate			
		Description			
		Calculate savings growth with annual compounding			
		Parameters			
		Name	Located	Require Schema	
		deposit	in	d	
		deposit	query	true	number
				<pre>curl --request GET \ --url 'https://127.0.0.1:4001/savings/Plans/estimate?deposit=REPLACE_THIS_VALUE&rate=REPLACE_THIS_VALUE&years=REPLACE_THIS_VALUE' \ --header 'accept: application/json' --header 'content-type: application/json'</pre>	



Information

The **Explore** view consists of three columns:

- The **left** column lists the APIs that you published on the micro gateway. In this example, the **savings** API definition includes two operations, and two schema data types that the operations use.
- The **middle** column lists **documentation** and **details** on the API, API operations, and schema data types.
- The **right** column lists code examples and an interactive test client for API operations.

___ 3. Test the **get /Plans/estimate** operation.

- ___ a. Select the **Get /Plans/estimate** operation from the **Savings 1.0.0** API.
- ___ b. Review the documentation for the API operation.
- ___ c. In the right column, scroll down to the test client.
- ___ d. Click **Try it**.

GET https://localhost:4001/savings/Plans/estimate

Examples Try it

___ e. Enter the following values for the test client:

- Deposit: 500
- Rate: 0.05
- Years: 20

Parameters	
deposit *	Generate
500	<input type="button" value="▼"/>
rate *	Generate
0.05	<input type="button" value="▼"/>
years *	Generate
20	<input type="button" value="▼"/>
Call operation	

___ f. Click **Call operation**.

**Note**

If you receive a **no responses received** error, repeat the steps to **accept a security exception in the web browser** from section 1.4 in this exercise.

- ___ g. Confirm that the API implementation returns a status of **200 OK**, with a balance of **16532.98**.

The screenshot shows the API Designer interface with a request and response pane. At the top right is a blue button labeled "Call operation".
Request:
GET https://localhost:4001/savings/Plans/estimate?deposit=500&rate=0.05&years=20
Headers:
Content-Type: application/json
Accept: application/json
Response:
Code: 200 OK
Headers:
content-type: application/json; charset=utf-8
x-global-transaction-id: 2963619361
x-ratelimit-limit: 100
x-ratelimit-remaining: 99
JSON Response:
{
 "balance": 16532.98
}

- ___ 4. Test the **POST /Plans/estimate** API operation.
- ___ a. Select the **POST /Plans/estimate** operation.
- ___ b. Click **Try it**.

- ___ c. In the test client, click **Generate** to create a sample JSON object in the request message:

The screenshot shows a portion of the API Designer interface. At the top, there's a dropdown menu labeled "Accept" with "application/json" selected. Below it, under "Parameters", there's a field labeled "plan *". To the right of this field is a blue "Generate" button, which is highlighted with a red rectangular box. Below the "plan" field, there's a code snippet in JSON format:

```
{
  "deposit": 300,
  "rate": 0.04,
  "years": "20"
}
```

At the bottom right of the interface is a large blue "Call operation" button.

- ___ d. Click **Call operation**.
- ___ e. Confirm that the API implementation returns a status of **200 OK**, with a balance value of **8933.42**.

The screenshot shows the results of a test call. At the top, there's a blue "Call operation" button. Below it, the "Request" section shows the following details:

POST https://localhost:4001/savings/Plans
/estimate
Headers:
Content-Type: application/json
Accept: application/json

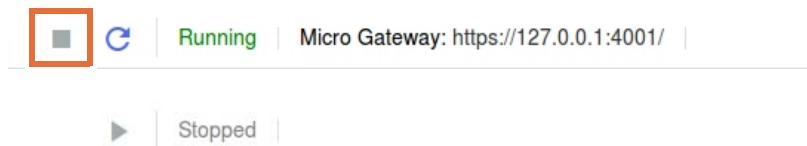
Below that, the "Response" section shows:

Code: 200 OK
Headers:
content-type: application/json; charset=utf-8
x-global-transaction-id: 4003442871
x-ratelimit-limit: 100
x-ratelimit-remaining: 99

At the bottom, there's a code snippet in JSON format:

```
{
  "balance": 8933.42
}
```

- ___ 5. Click the icon to stop the Micro Gateway.



- ___ 6. Close the API Designer in the browser.
___ 7. Click Ctrl+C in the terminal to stop the editor process.

End of exercise

Exercise review and wrap-up

In the first part of the exercise, you created an OpenAPI definition in the API Designer web application. You defined the API paths, operations, request, and response messages that describe the saving sample REST API application. You also reviewed a copy of the API application that is hosted on IBM Bluemix.

In the second part of the exercise, you defined a second API operation for the saving sample REST API. The first example used HTTP query parameters, and the second example embeds input parameters as a JSON object in the message body. You specified the data type schema as an OpenAPI schema type definition.

In the last part of the exercise, you reviewed and tested the API definition in the **Explore** view of the API Designer application.

Exercise 2. Defining an API that calls an existing SOAP service

Estimated time

01:00

Overview

With API Connect, you can define an API from existing enterprise services. In this exercise, you define an API that calls an existing SOAP service. You define API paths and methods that map to SOAP web service operations, and map SOAP message types to API data types. You then test the SOAP API in the API Explorer.

Objectives

After completing this exercise, you should be able to:

- Create an API definition with the API Designer web application
- Define API operations that map to a SOAP web service
- Map SOAP message types to API data types
- Test the SOAP API on the DataPower gateway of the developer toolkit
- Publish the API to API Manager
- Test the SOAP API in the API Manager test client (optional)

Introduction

For existing API implementations, you can **expose the API operations** at the **API gateway**. In this scenario, you create an API definition: a document that specifies a list of API paths, methods, and expected request and response messages.

In this scenario, your organization already developed a set of SOAP web services. SOAP is a mature, remote service standard that is common with large enterprise architectures. The goal of this exercise is to take the operations from the existing SOAP service and make it available at the API gateway.

In effect, you build a SOAP web service proxy for an existing service. This configuration does not convert a SOAP request to an alternative format. The concept of a SOAP-to-REST API bridge is covered in a later exercise in this course.

Requirements

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

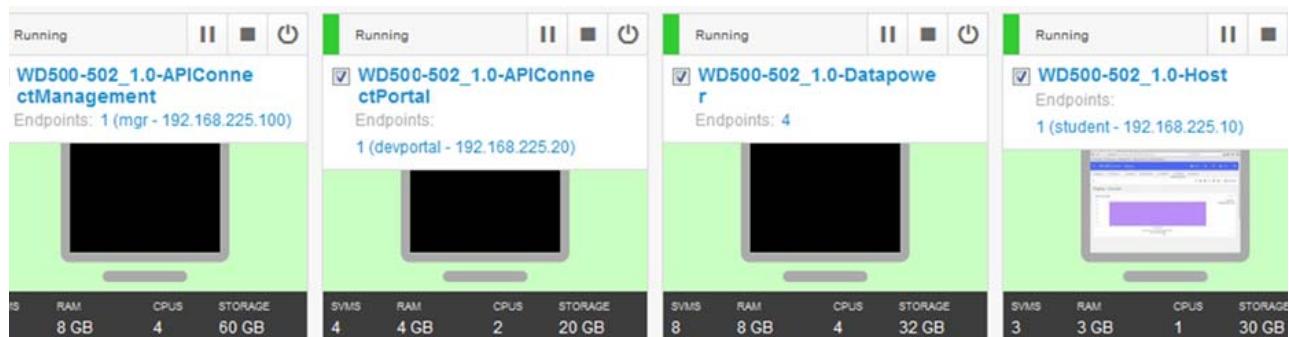
Exercise instructions

Before you begin

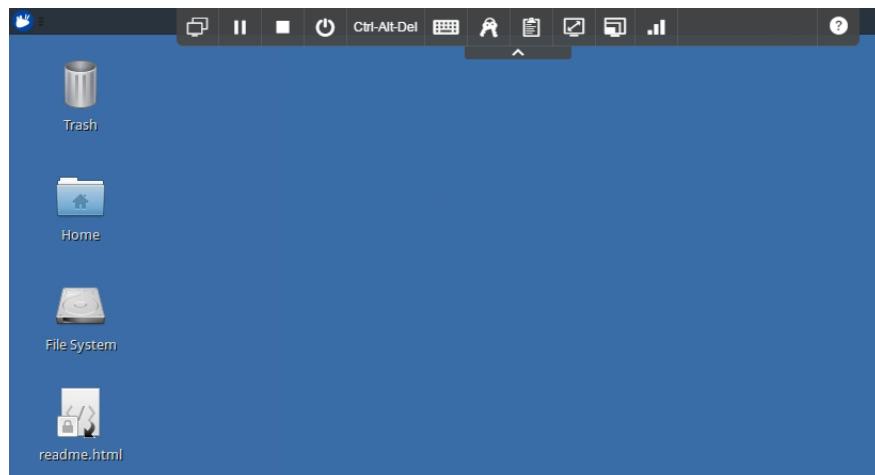
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



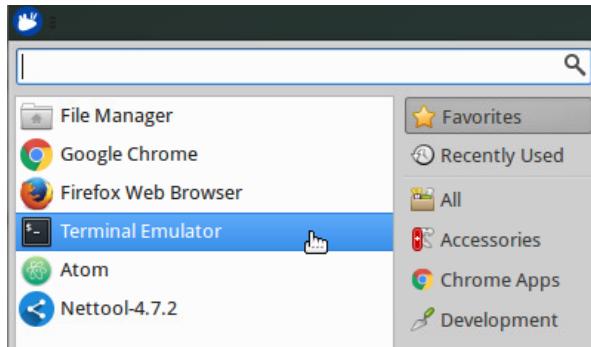
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

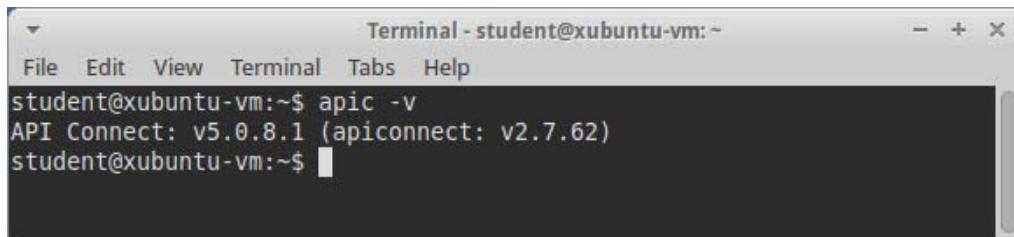
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

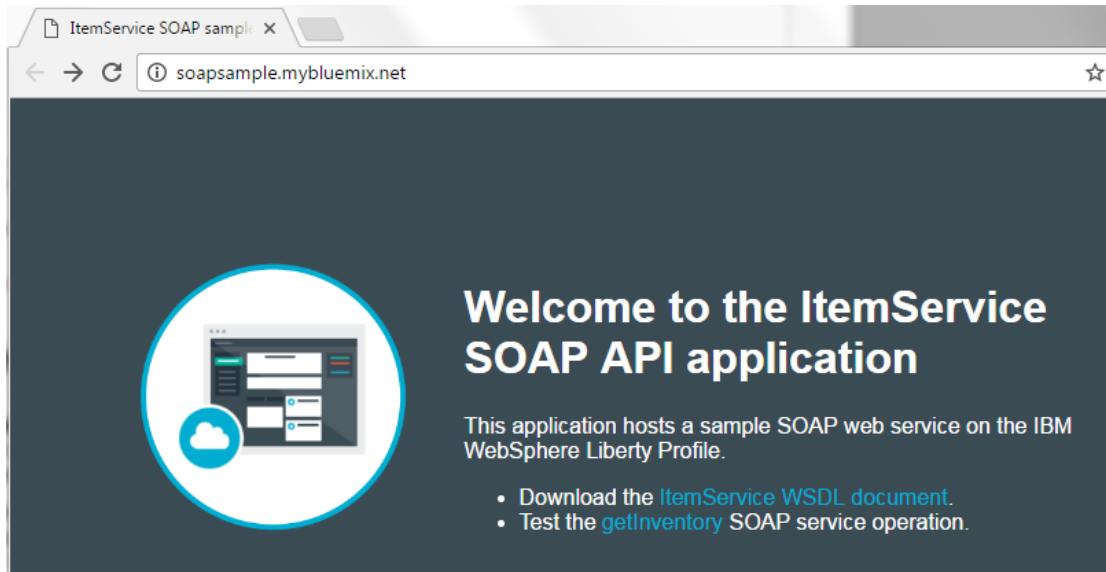
2.1. Review the existing SOAP web service

The **ItemService** application maintains a list of vintage IBM products from its corporate history. The application hosts a remote service that returns a list of items in the store inventory. Unlike services on API Connect, the ItemService remote service is built as a **SOAP web service**.

SOAP is an application protocol for remote service invocation over HTTP connections. SOAP services store the operation command and data in a custom XML data format that is known as a **SOAP envelope**. This design is in contrast to modern REST APIs, which uses HTTP methods as the command and store application data in the HTTP message.

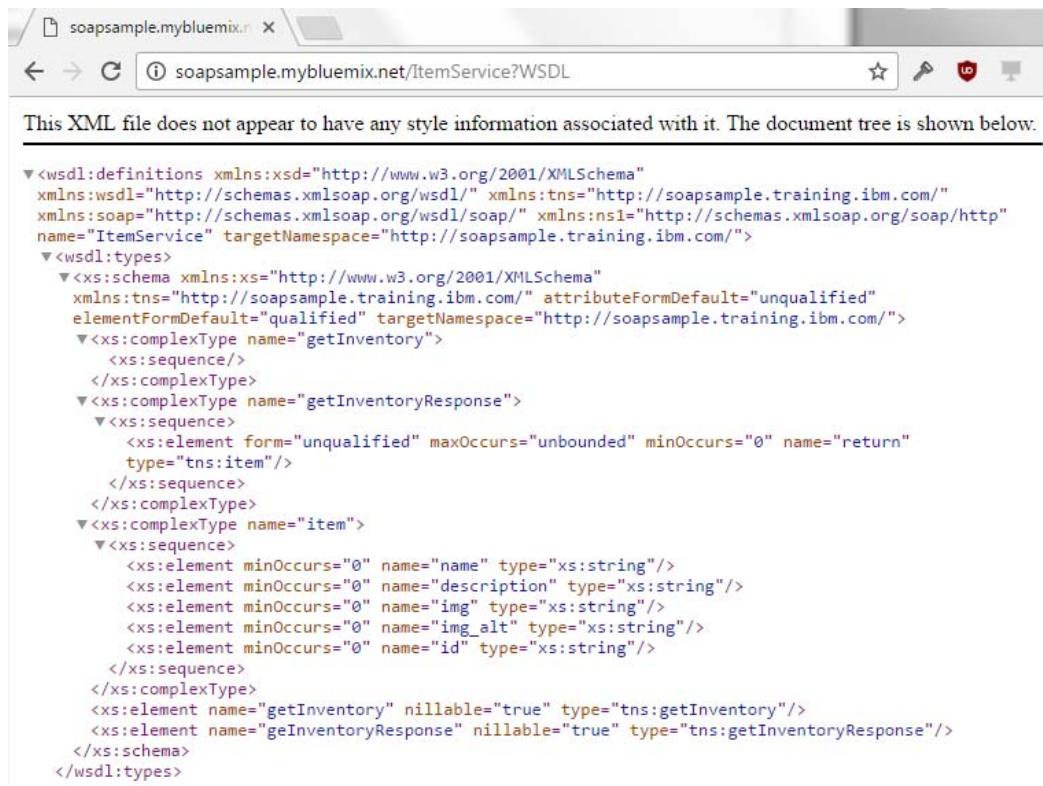
In this section, you test and verify the **ItemService** SOAP service. You retrieve and review a copy of the SOAP service interface, in the form of a Web Services Description Language (WSDL) document.

- ___ 1. Open the **ItemService** website.
 - ___ a. Open the <http://soapsample.mybluemix.net> page in a web browser.



- ___ b. Click the **WSDL document** link on the main page.

- __ c. Examine the contents of the **ItemService** WSDL document.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<wsdl:definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://soapsample.training.ibm.com/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    name="ItemService" targetNamespace="http://soapsample.training.ibm.com/">
  <wsdl:types>
    <xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://soapsample.training.ibm.com/" attributeFormDefault="unqualified"
      elementFormDefault="qualified" targetNamespace="http://soapsample.training.ibm.com/">
      <xss:complexType name="getInventory">
        <xss:sequence/>
      </xss:complexType>
      <xss:complexType name="getInventoryResponse">
        <xss:sequence>
          <xss:element form="unqualified" maxOccurs="unbounded" minOccurs="0" name="return"
            type="tns:item"/>
        </xss:sequence>
      </xss:complexType>
      <xss:complexType name="item">
        <xss:sequence>
          <xss:element minOccurs="0" name="name" type="xs:string"/>
          <xss:element minOccurs="0" name="description" type="xs:string"/>
          <xss:element minOccurs="0" name="img" type="xs:string"/>
          <xss:element minOccurs="0" name="img_alt" type="xs:string"/>
          <xss:element minOccurs="0" name="id" type="xs:string"/>
        </xss:sequence>
      </xss:complexType>
      <xss:element name="getInventory" nillable="true" type="tns:getInventory"/>
      <xss:element name="getInventoryResponse" nillable="true" type="tns:getInventoryResponse"/>
    </xsschema>
  </wsdl:types>
```



Information

What is the purpose of the Web Services Description Language (WSDL) document?

The purpose of the WSDL document is two-fold: to describe the service interface, and to specify the network endpoint and protocol bindings for the web service.

The **service interface** describes all the details that a client application requires to call the web service. In this document, the **schema** section lists two XML data structures: the request and response messages for the service operations. The **item** complex type describes the structure of the item model object: five fields that describe the details of an item in the store inventory.

The **portType** section lists the names of the **operations** in the web service. In this example, **ItemService** has one operation, named **getInventory**. The expected request message is defined in an XML element named **getInventory**. The response message is an XML element named **getInventoryResponse**.

The rest of the document describes how to connect and call the SOAP service over the network.

The **service** section lists the service endpoint as:

<http://soapsample.mybluemix.net/ItemService>

The **bindings** section explains how to construct an HTTP request message for the service, and how to interpret the HTTP response message from the same service.

For more information about the WSDL specification, see: <https://www.w3.org/TR/wsdl>

- ___ 2. Test the **getInventory** SOAP service operation.
 - ___ a. Return to the **ItemService** main page.
 - ___ b. Click **Test getInventory SOAP operation**.
 - ___ c. Review the **SOAP request** and **response** messages.

ItemService SOAP API Sample

This test client invokes the **getInventory** operation from the SOAP service.

SOAP request:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><soapenv:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:tns="http://soapsample.training.ibm.com/"><soapenv:Body>
<tns:getInventory></tns:getInventory></soapenv:Body></soapenv:Envelope>
```

SOAP response:

HTTP status code 200

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body><ns2:geInventoryResponse
  xmlns:ns2="http://soapsample.training.ibm.com/"><return><ns2:name>Dayton
  Meat Chopper</ns2:name><ns2:description>Punched-card tabulating machines and
  time clocks were not the only products offered by the young IBM. Seen here
  in 1930, manufacturing employees of IBM's Dayton Scale Company are
  assembling Dayton Safety Electric Meat Choppers.</ns2:description>
<ns2:img>images/items/meat-chopper.jpg</ns2:img><ns2:img_alt>Meat
  Chopper</ns2:img_alt><ns2:id>1</ns2:id></return><return><ns2:name>Hollerith
  Tabulator and Sorter</ns2:name><ns2:description>This equipment is
```



Information

The **ItemService** website includes a test client for the SOAP service. This page sends an SOAP request message to the `http://soapsample.mybluemix.net/ItemService` endpoint.

The first half of the page displays the SOAP request message. The HTTP request message stores an XML document in the SOAP envelope format. In the SOAP message body, the `<tns:getInventory />` XML element represents the name of the SOAP operation.

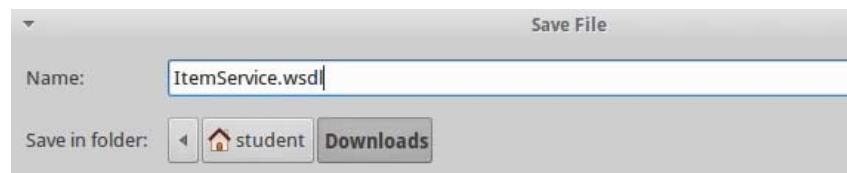
The second half of the page displays the response from the SOAP service. The HTTP status code of **200** indicates that the SOAP service processed the request successfully. The HTTP response message stores another XML document in the SOAP envelope format.

In this example, the **getInventory** service returned the names and descriptions of **items** in the inventory.

- ___ 3. Download a copy of the **ItemService** WSDL document.
- ___ a. Return to the ItemService main page.
- ___ b. Right-click the **ItemService WSDL document** link.
- ___ c. Click **Save link as...**.



- ___ d. Rename the document to: **ItemService.wsdl**

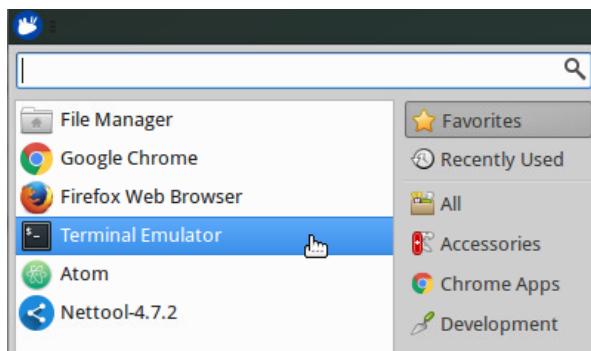


- ___ e. Click **Save**.

2.2. Create a SOAP API definition from a WSDL document

In this section, you generate an OpenAPI 2.0 API definition from an existing SOAP service. Import the WSDL document into the API Designer. Examine the API definition and message processing policies for the API.

- ___ 1. Create a directory to save development artifacts.
 - ___ a. Open a **Terminal Emulator** application window.



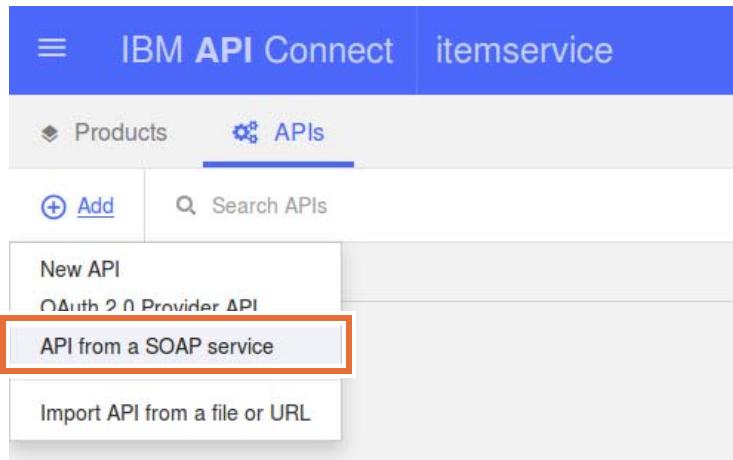
- ___ b. In the home directory, create a directory that is named **itemservice**.


```
$ mkdir ~/itemservice
```
- ___ c. Change directory to **itemservice**.


```
$ cd ~/itemservice
$ pwd
/home/student/itemservice
```
- ___ 2. Start the API Designer web application.
 - ___ a. Run the **apic edit** command.


```
$ apic edit
```
- ___ 3. Create an API definition that is named **ItemService** from the ItemService WSDL document.
 - ___ a. In the API Designer main page, switch to the **API** tab.

- ___ b. Select Add > API from a SOAP service.



- ___ c. In the New API from WSDL page, click Upload file.

New API from WSDL

Upload file

Load from URL

- ___ d. Select the **ItemService.wsdl** document that you saved in an earlier step.
 ___ e. Select **Show operations** from the “New API from WSDL” page.

New API from WSDL

ItemService.wsdl [Change file](#)

1 SOAP service found

ItemService

[Show operations](#)

getInventory

[Back](#)

[Cancel](#)

[Add a product...](#)

[Done](#)



Information

The **New API from WSDL** parsed through the WSDL document and found one SOAP service named **ItemService**. The service defines one operation, named **getInventory**.

- ___ f. Select the **ItemService** SOAP service.
- ___ 4. Create a **product** to hold the ItemService API definition.
- ___ a. Click **Add a product**.

1 SOAP service found

<input checked="" type="checkbox"/> ItemService	Show operations
getInventory	
Back	Cancel Add a product... Done

- ___ b. Change the **name** and **title** of the product to: **soap-services**
- ___ c. Set the version to **1.0.0**.
- ___ d. Clear the **Publish this product to a catalog** check box.

New API from WSDL

Product template	Create product using template Default
Info	Title * soap-services
	Name * soap-services
Version *	1.0.0
Publishing	<input type="checkbox"/> Publish this product to a catalog

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

- ___ e. Click **Done**.

- ___ 5. Examine the **itemservice** API definition in the API editor.
 ___ a. In the API editor, select the **Base Path** section.

The screenshot shows the API editor interface with the 'Design' tab selected. On the left, a sidebar lists various API configuration sections: Info, Schemes, Host, Base Path (which is currently selected and highlighted in blue), Consumes, Produces, Lifecycle, Policy Assembly, Security Definitions, Security, and Extensions. The main panel displays the 'Base Path' configuration, where the base path is set to '/ItemService'. Below this, the 'Consumes' section lists 'text/xml' as a media type, with a small 'x' icon indicating it can be removed. The 'Produces' section lists 'application/json' and 'application/xml', with 'application/xml' having a checked checkbox. There is also a link to 'Add media type'.



Information

The **Base Path** section describes the network endpoint on the API gateway for requests to the ItemService SOAP API.

Examine the **Consumes** and **Produces** sections of the API definition. The ItemService API expects HTTP requests with a **text/xml** media type. It returns HTTP responses with an **application/xml** media type.

In effect, the itemservice API receives and sends SOAP messages. This message type is in contrast to REST APIs, which use JavaScript Object Notation (JSON) as the data type.

- ___ 6. Remove the **client ID** security requirement from the **itemservice** API definition.
- ___ a. Select the **Security Definitions** section of the itemservice API.

The screenshot shows the 'itemservice' API definition in the API Designer. The left sidebar has a 'Security Definitions' tab selected. The main panel shows a single security definition named 'clientID (API Key)'. The details for this definition include:

- Name: clientID
- Parameter name: X-IBM-Client-Id
- Located In: Header
- Description: (empty)

Information

By default, all API definitions that you create in the API Designer include the **API Key** security requirement. This requirement forces every API caller to supply a valid **client ID** value. That is, application developers must register their application in the Developer Portal.

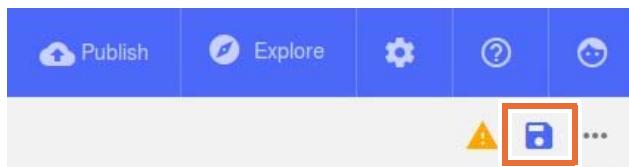
To allow unregistered applications access to the API, clear the **API Key** security requirement in the API definition.

- ___ b. Select the **Security** section.
- ___ c. Clear the **client ID** security requirement.

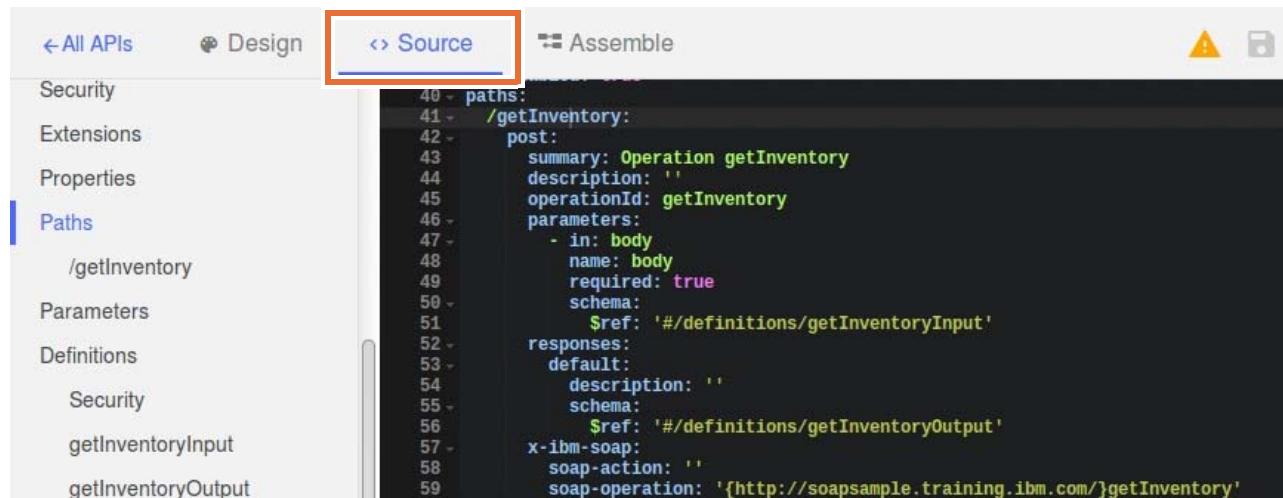
The screenshot shows the 'Security' section of the API definition. It contains a list of security requirements:

- Option 1: clientID (API Key) (highlighted with a red box)

- ___ d. Save the changes to the API definition.



- ___ 7. Review the message processing policies in the **itemservice** API definition.
- ___ a. Select the **Paths** section.
 - ___ b. Switch to the **Source** tab.



The screenshot shows the API management interface with the 'Source' tab highlighted. The left sidebar lists sections like Security, Extensions, Properties, Paths, Parameters, Definitions, and two definitions: getInventoryInput and getInventoryOutput. The main area displays the API definition code for the /getInventory endpoint:

```

40 - paths:
41 -   /getInventory:
42 -     post:
43 -       summary: Operation getInventory
44 -       description: ''
45 -       operationId: getInventory
46 -       parameters:
47 -         - in: body
48 -           name: body
49 -           required: true
50 -           schema:
51 -             $ref: '#/definitions/getInventoryInput'
52 -       responses:
53 -         default:
54 -           description: ''
55 -           schema:
56 -             $ref: '#/definitions/getInventoryOutput'
57 -         x-ibm-soap:
58 -           soap-action: ''
59 -           soap-operation: '{http://soapsample.training.ibm.com/}getInventory'

```

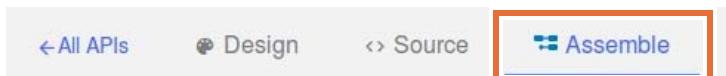
Information

The **itemservice** API defines one API operation: **POST /getInventory**. In the SOAP request message, this operation expects an input message named **getInventoryInput**. It returns a SOAP response message with an output message of **getInventoryOutput**.

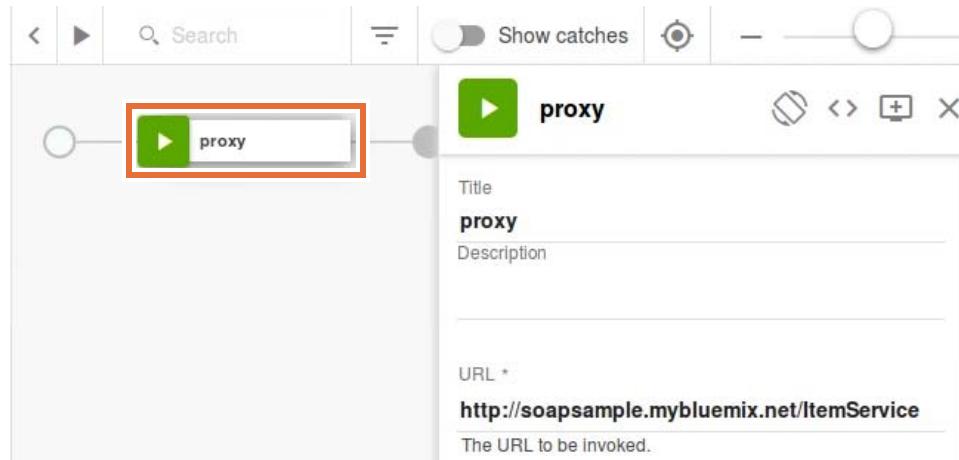
The **definitions** section of the API definition specifies the structure of the **getInventoryInput** and **getInventoryOutput** XML elements. You saw the message structures earlier from the WSDL document.

This API operation also defines a **soap-operation** header with the value **{http://soapsample.training.ibm.com}getInventory**. In this case, the entry in the parentheses is not a website address. It is the XML namespace for the **itemservice** SOAP service.

- ___ 8. Examine the message processing policies for the **itemservice** API definition.
- ___ a. Click the **Assemble** tab.



- __ b. Select the **proxy** policy in the assembly flow.



Information

The **assemble** view displays the **message processing policies** for all operations in the **itemservice** API. At run time, the API gateway enforces these policies on every request message that clients send to the API.

The **itemservice** API definition includes one policy: a **proxy** action. The purpose of this policy is to forward the HTTP request message to the SOAP service implementation at <http://soapsample.mybluemix.net/ItemService>.

- __ c. Click the **filter policies** icon.



- __ d. Switch to the **Micro Gateway policies**.

- __ e. Examine the error in the message processing policy pipeline.

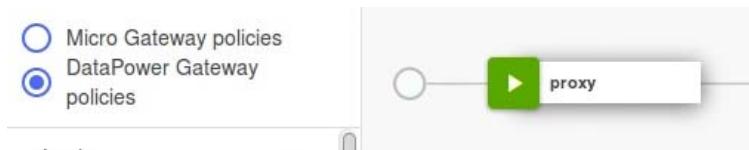


**Note**

The **proxy** policy is available on the DataPower Gateway only. You cannot proxy a SOAP service with the **Micro Gateway**.

Switch the policy type back to **DataPower Gateway policies** before you continue with this exercise.

-
- ___ f. Switch back to the **DataPower Gateway policies**.



- ___ g. If necessary, save the changes to the API definition.

2.3. Test the API on the embedded DataPower gateway

In this part, you test an API that contains DataPower policies on an embedded DataPower Docker container. To use this feature, you must be running the API Connect toolkit v5.0.7 or later, and the Docker software must be installed on the workstation. When you test an API definition that includes DataPower policies, the Docker DataPower image is automatically downloaded and started on your workstation.

- 1. The itemservice API must be open in the assembly editor of the API Designer.
- 2. Start the gateway from the terminal window.
 - a. From a terminal window, select **File > Open Terminal**.
The terminal emulation window opens in the `~/itemservice` directory.
 - b. In the terminal, type: `apic start`
 - c. After a few moments, you see the message `Service itemservice-gw starting`, use "apic services" to obtain port details.



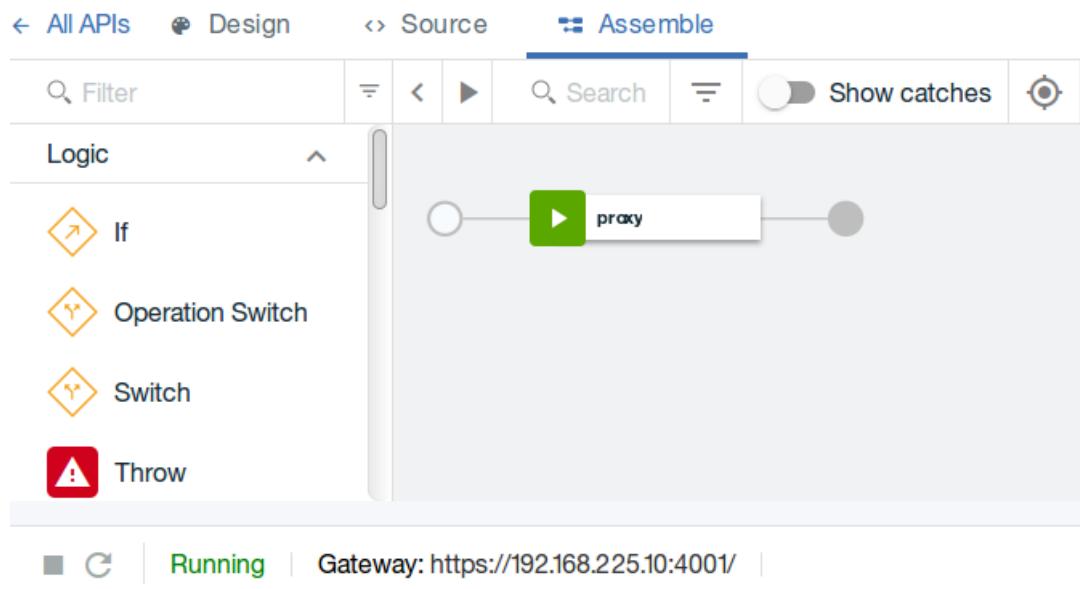
Note

The first time that you issue the `apic start` command in the terminal emulator window that contains an API definition that uses DataPower policies, it takes some time for the DataPower image to be downloaded. In some cases, the delay is in the order of about 5 minutes.

- d. Type: `apic services`
- e. The terminal displays the message: `Service itemservice-gw running on port 4001.`

```
Terminal - student@xubuntu-vm: ~/itemservice
File Edit View Terminal Tabs Help
student@xubuntu-vm:~/itemservice$ apic start
Service itemservice-gw starting, use "apic services" to obtain port details.
student@xubuntu-vm:~/itemservice$ apic services
Service itemservice-gw running on port 4001.
student@xubuntu-vm:~/itemservice$
```

- f. You also see in the API Designer that the gateway and the application are now running.

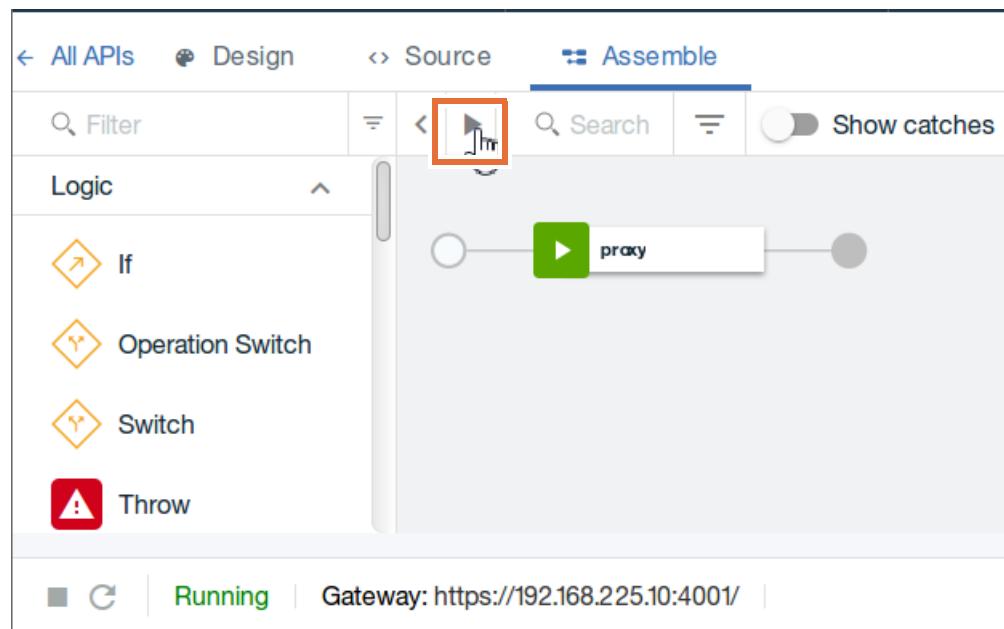


- g. Type `docker ps` in the terminal window.

You see that the embedded DataPower Docker container is running.

```
Terminal - student@xubuntu-vm:~/itemservice
File Edit View Terminal Tabs Help
student@xubuntu-vm:~/itemservice$ apic start
Service itemservice-gw starting, use "apic services" to obtain port details.
student@xubuntu-vm:~/itemservice$ apic services
Service itemservice-gw running on port 4001.
student@xubuntu-vm:~/itemservice$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
MMAND               ibm-apiconnect-toolkit/datapower-api-gateway:1.1.8   "/bin/drouter"
                   8 minutes ago      Up 8 minutes       0.0.0.0:32771->80/tcp, 0.0.0.0:32770->5554/tcp, 0.0.0.0:32769->9090/tcp, 0.0.0.0:4001->9443/tcp
itemservice         datapower-api-gateway_1
f943df03efc7       ibm-apiconnect-toolkit/datapower-mgmt-server-lite:1.1.8   "node lib/server.js"
                   8 minutes ago      Up 8 minutes       0.0.0.0:32768->2443/tcp
itemservice         datapower-mgmt-server-lite_1
student@xubuntu-vm:~/itemservice$
```

- ___ 3. Test the itemservice API from the test option of the API Designer.
 - ___ a. In the Assembly editor, click the Test icon.



- ___ b. Select the **post/getInventory** operation in the test client.



- ___ c. Scroll down to the body area in the test client.

- ___ d. Click **Generate** to create some data in the body of the message.



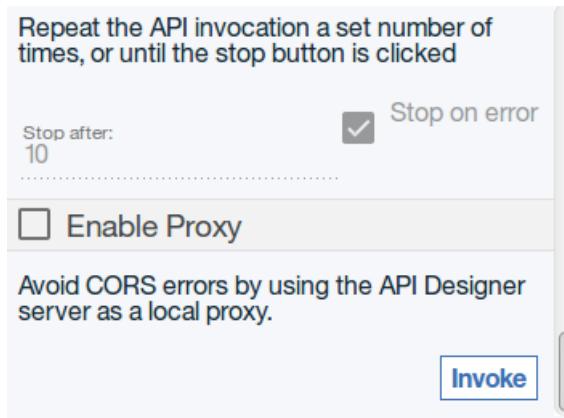
- ___ e. The body area is populated with data.

The screenshot shows the 'Design' tab of the API designer. The title bar says 'Test'. The main area displays the generated XML body:

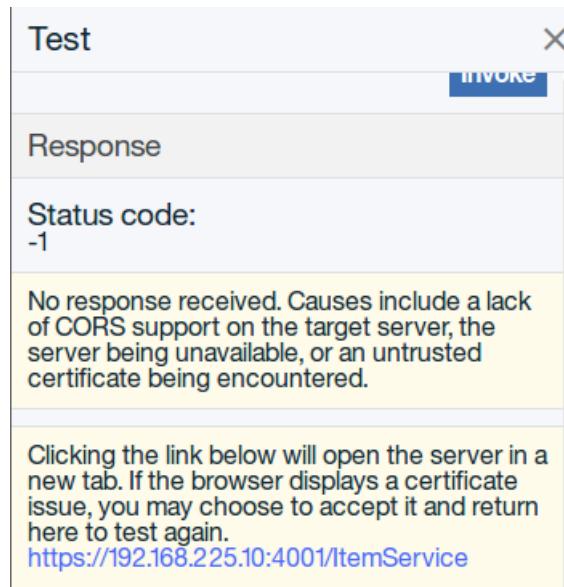
```
body *
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org
  /soap/envelope/">
  <soapenv:Header>
    <wsse:Security
      xmlns:wsse="http://docs.oasis-open.org
      /wss/2004/01/oasis-200401-wss-wssecurity-
      secext-1.0.xsd"
      xmlns:wsu="http://docs.oasis-open.org
      /wss/2004/01/oasis-200401-wss-wssecurity-
      utility-1.0.xsd">
        <wsse:UsernameToken>
```

At the bottom, there is a status bar with icons for 'Running' and 'Gateway: https://192.168.225.10:4001/'.

- ___ f. Scroll to the bottom of the test client.
Then, click **Invoke**.



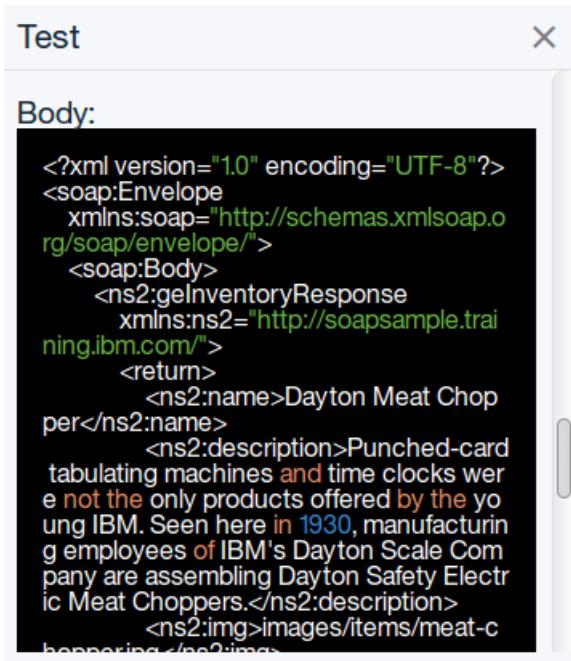
- ___ g. The first time that you invoke the API operation, you might see an error that is related to CORS (Cross-Origin Resource Sharing).



Click the link in the test area.

- ___ h. Add a security exception in the browser for the link.
___ i. Click **Invoke** to rerun the post operation.

- ___ j. The result is displayed in the test window.



The screenshot shows a 'Test' window with a 'Body:' section containing a SOAP response message. The message is as follows:

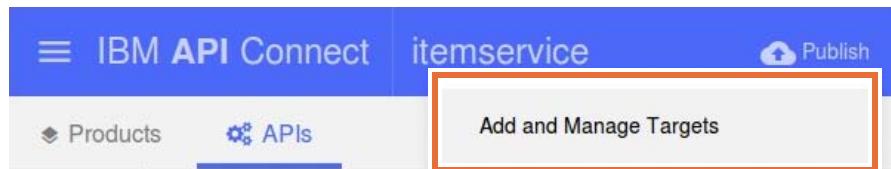
```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns2:gelInventoryResponse
            xmlns:ns2="http://soapsample.training.ibm.com/">
            <return>
                <ns2:name>Dayton Meat Chopper</ns2:name>
                <ns2:description>Punched-card tabulating machines and time clocks were not the only products offered by the young IBM. Seen here in 1930, manufacturing employees of IBM's Dayton Scale Company are assembling Dayton Safety Electric Meat Choppers.</ns2:description>
                <ns2:img>images/items/meat-choppering</ns2:img>
            </return>
        </ns2:gelInventoryResponse>
    </soap:Body>
</soap:Envelope>
```

- ___ k. Close the test window.
- ___ 4. Stop the gateway.
- ___ a. From the same terminal window that you started the gateway, type: apic stop
The itemservice-gw is stopped.
- ___ b. The ibm-apiconnect-toolkit/datapower-api-gateway and
ibm-apiconnect-toolkit/datapower-mgmt-server-lite containers are stopped.

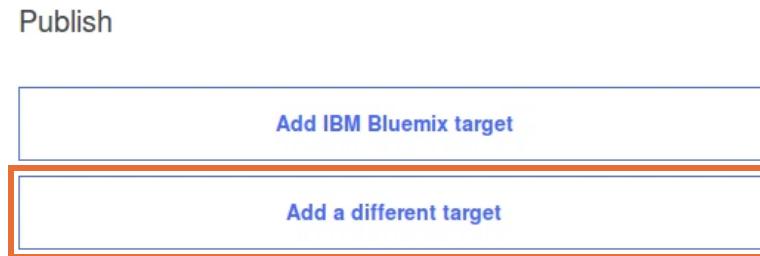
2.4. Publish the SOAP API definition

You can test an API definition that includes DataPower policies on API Manager. You must publish the API to the API Manager. In this section, publish the API product and definition from the API Designer web application.

- ___ 1. Sign in to the API Manager server.
 - ___ a. In the API Designer, click **Publish**.
 - ___ b. Click **Add and manage targets**.



- ___ c. In the Publish page, click **Add a different target**.



- ___ d. Log in to the API Manager server:
 - Host address: **mgr.think.ibm**
 - User name: **student@think.ibm**
 - Password: **Passw0rd!**

Sign in to IBM API Connect

API Connect host address		
<input type="text" value="mgr.think.ibm"/>		
Username		
<input type="text" value="student@think.ibm"/>		
Password		
<input type="password" value="....."/>		
<input type="button" value="Back"/>	<input type="button" value="Cancel"/>	<input style="background-color: #0070C0; color: white; font-weight: bold; border: 1px solid #0070C0; padding: 2px 10px; border-radius: 5px; font-size: 10pt; text-decoration: none; margin-right: 10px;" type="button" value="Sign in"/>

- ___ e. Click **Sign in**.

- __ 2. Define a publish target for the **soap-service** product and **itemservice** API.
- __ a. Select the **Sales** organization and **Sandbox** catalog.

Select an organization and catalog

Organization
Sales

Search

Sandbox

- __ b. Scroll down the page and click **Save**.
- __ 3. Publish the **soap-service** product to API Manager.
- __ a. Click **Publish**.
- __ b. Select the **mgr.think.ibm** publish target.

itemservice

Publish

Other Orgs

Catalog: Sandbox (sb)
Org: Sales (sales)
Server: mgr.think.ibm

Add and Manage Targets

- __ c. In the Publish window, leave the options empty to stage and publish the product and the API definition.

Publish

This target only has a catalog and no application. Application will not be published.

- Stage only
- Select specific products

Cancel **Publish**

- __ d. Click **Publish**.
- __ e. You see a message: "Successfully published product".

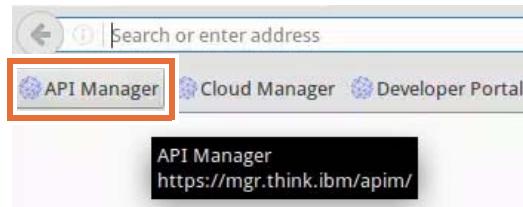
2.5. Test the API in the API Manager (optional)

This part is optional, since you already tested the API on the embedded DataPower gateway.

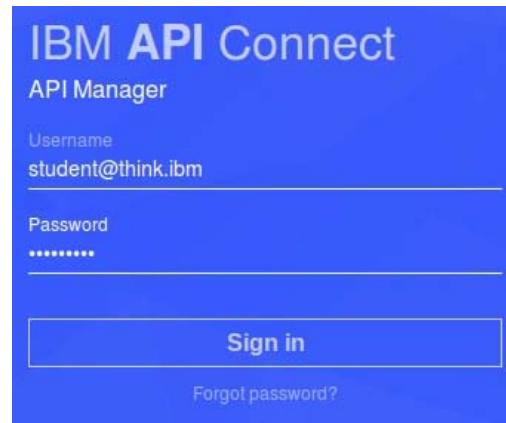
You published the SOAP API definition to the API Manager in the last section of the exercise. In turn, the API Manager published the **itemservice** API definition to the DataPower Gateway.

To test an API, you must use the test client in the API Manager web server.

- ___ 1. Log in to the API Manager website.
 - ___ a. Open a web browser window.
 - ___ b. Click the **API Manager** bookmark to open <https://mgr.think.ibm/apim>.



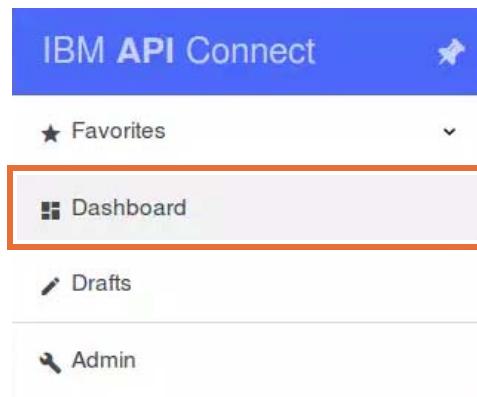
- ___ c. Log in to the API Manager
 - User name: **student@think.ibm**
 - Password: **Passw0rd!**



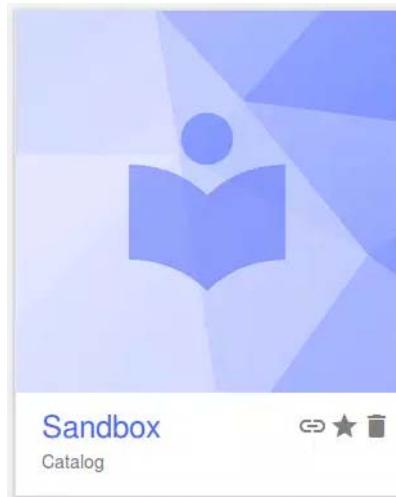
- ___ d. Click **Sign in**.
- ___ 2. Examine the published **itemservice** API definition.
 - ___ a. Click the **menu** icon from the upper-left section of the page.



- __ b. Select **Dashboard**.



- __ c. Select the **Sandbox** catalog.



- __ 3. Examine the **soap-services** API product in the editor.

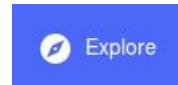
- __ a. Select the **soap-service** API product.

A screenshot of the IBM API Connect Product Editor. The top navigation bar shows "IBM API Connect" and "Sand...". Below the bar, there are tabs for "Dashboard", "Products" (which is underlined in blue), "Approvals", "Community", "Analytics", and "Settings". A search bar labeled "Search products" is present. The main content area displays a table with one row. The row contains the title "soap-services" and the state "Published 8 minutes ago". The entire row is highlighted with a red border.

- ___ b. Confirm that the **soap-services** product is published, along with the **itemservice** API definition.

The screenshot shows the product details for 'soap-services' version 1.0.0. At the top right, there's a green 'Published' button with a timestamp of '8 minutes ago'. Below it, there's an 'Offline / Online' toggle switch which is turned on (blue). To the right of the switch is a bar chart icon. Under the 'APIs' section, the 'itemservice 1.0.0' API is listed with its status as 'Published' and a blue bar chart icon. In the 'Plans' section, there's a 'Default Plan' with 0 subscribers, indicated by a blue bar chart icon.

- ___ 4. Open the **getInventory** SOAP operation in the **Explore** view.
___ a. Click **Explore** from the upper section of the page.



- ___ b. Select the **Sandbox** catalog.
___ c. In the list of deployed APIs, select the **itemservice 1.0.0** API.
___ d. Select the **getInventory** API operation.

The screenshot shows a navigation bar with 'Connect' and 'Sand...' buttons. Below is a 'Drafts' section. Under 'Catalogs', there is a list with 'Sandbox' highlighted with a red rectangle.

- ___ 5. Call the **POST** API operation.
___ a. In the left column, click **getInventory**.
___ b. In the right column, scroll down the page in the **POST https://api.think.ibm/sales/sb/ItemService** test client.
___ c. Click **Try it**.

- __ d. Click **Generate** to create the SOAP request message.

POST <https://api.think.ibm/sales/sb/ItemService> - Operation `getInventory`

Type Headers Examples Try it

Content-Type

text/xml

Accept

application/xml

Parameters

body*

Generate

- __ e. Click **Call operation**.



Note

At the first invocation of the service, your web browser blocks the service call. You must add a security exception before the client can make the Cross-Origin Resource Sharing (CORS) script call.

Request

```
POST https://api.think.ibm/sales/sb/ItemService
Headers:
Content-Type: text/xml
Accept: application/xml
SOAPAction:
```

Response

Code: 0

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://api.think.ibm/sales/sb/ItemService>

- ___ f. Select the **https://api.think.ibm/sales/sb/ItemService** API gateway endpoint.
 - ___ g. Add a **security exception** in your web browser for the link.
- ___ 6. Test the **POST getInventory** API operation again.
- ___ a. Click **Call operation**.
 - ___ b. Confirm that the API gateway returns a SOAP response message with the name and description of inventory items.

The screenshot shows the API Designer interface with a successful SOAP response. The request section shows the following headers:

```
Content-Type: text/xml
Accept: application/xml
SOAPAction:
```

The response section shows the following details:

Response

Code: 200 OK
 Headers:
 content-language: en-US
 content-type: text/xml; charset=ISO-8859-1
 x-global-transaction-id: 196c556559c5838d002ff7f0

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/
  <soap:Body>
    <ns2:getInventoryResponse
      xmlns:ns2="http://soapsample.training.ibm.c
      <return>
        <ns2:name>Dayton Meat Chopper</ns2:n
        <ns2:description>Punched-card tabulating
        <ns2:img>images/items/meat-chopper.jp
        <ns2:img_alt>Meat Chopper</ns2:img_alt
        <ns2:id>1</ns2:id>
      </return>
    </return>
```

- ___ 7. Sign out of API Manager.
- ___ 8. Close the API Designer in the browser.
- ___ 9. Click Ctrl+C in the terminal to stop the editor process.

End of exercise

Exercise review and wrap-up

The first part of the exercise described how to create an OpenAPI definition for an existing SOAP service. Specifically, you generated a SOAP API definition based on an existing SOAP service interface: the WSDL document.

In an API Connect solution, you must deploy an SOAP API to a DataPower API Gateway.

The second part of the exercise tested the SOAP API definition on the API Connect toolkit DataPower Docker container. You then published and tested the SOAP API definition to the API Management server. You optionally reviewed and tested the SOAP API operation in the Explore view in the API Manager website.

Exercise 3. Creating a LoopBack application

Estimated time

01:00

Overview

In this exercise, you build a LoopBack application to implement an API. You generate the application scaffold with the apic command-line utility, and define the model and properties with the API Designer web application.

Objectives

After completing this exercise, you should be able to:

- Create an API definition with the API Designer web application
- Create an application scaffold with the apic command-line utility
- Create LoopBack models and properties

Introduction

The LoopBack framework is a framework to build REST APIs in the Node.js programming language. This open source framework provides codeless API composition, isomorphic models, and a set of Node.js modules that you can use independently or together to quickly build applications that expose REST APIs.

An application interacts with data sources through the LoopBack model API, available locally within Node.js, remotely over REST, and through native client APIs for iOS, Android, and HTML5. Using these APIs, apps can query databases, store data, upload files, send emails, create push notifications, register users, and perform other actions that data sources and services provide.

In this exercise, you examine the sample API implementation for the note application: an API to create, retrieve, update, and delete a set of text notes. Identify the structure of a LoopBack application. Define and build model objects with the command-line utility and API Designer web application. Review the REST APIs with the API Explorer.

Requirements

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

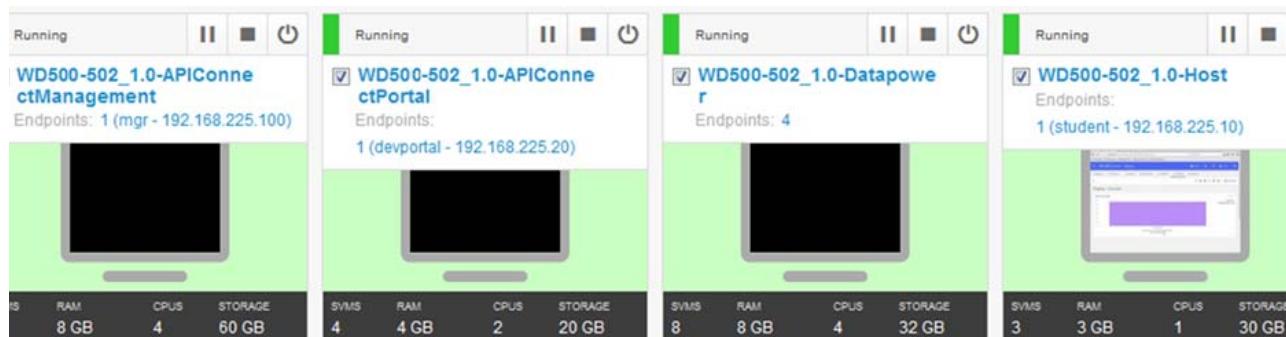
Exercise instructions

Before you begin

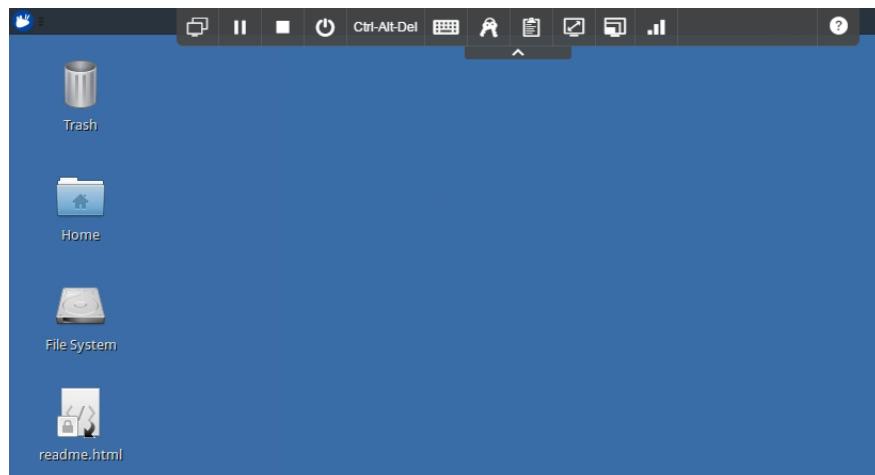
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



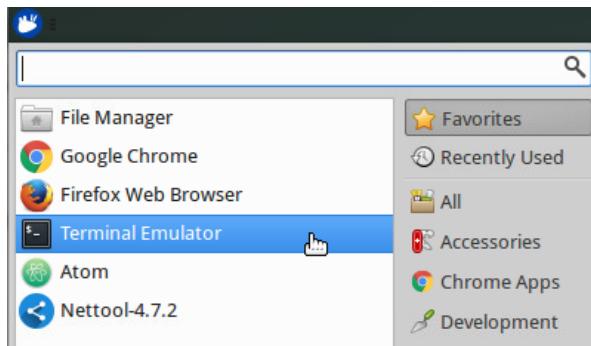
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

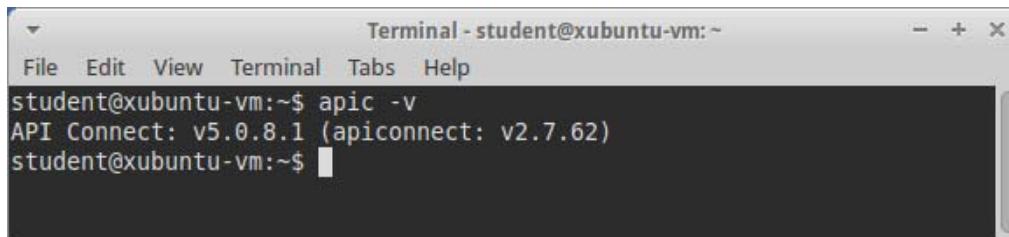
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

3.1. Verify your IBM API Connect toolkit installation

In this part, you confirm that the student workstation installed the IBM API Connect toolkit and its prerequisite software properly.



Note

The lab exercise case study and instructions were written and tested for the product versions that are listed in the instructions. If you run this exercise on your own API Connect toolkit installation, your results might not match the instructions that are written in this guide.



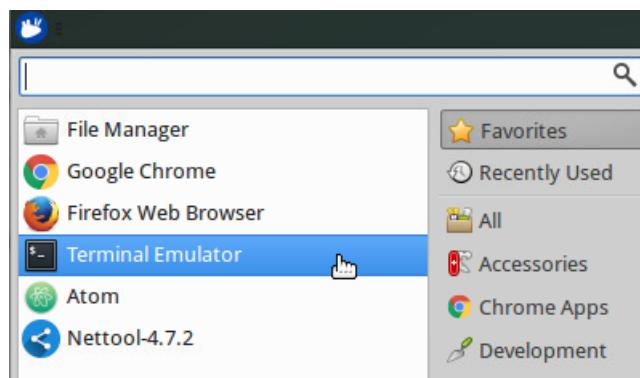
Information

The following instructions are commands that you enter in a Linux terminal emulator application. The line that starts with a dollar sign (\$) represents the user prompt.

```
$ whoami  
student
```

You type in the command after the prompt. The next line in the instruction represents the expected output from the command.

- ___ 1. Open a terminal emulator application.
 - ___ a. Select the Xubuntu start menu at the upper-right section of your display.
 - ___ b. In the **Favorites** section, select **Terminal Emulator**.



- ___ 2. Verify the version of “node” that is installed on your system.

```
$ which node  
/usr/local/bin/node  
$ node -v  
v6.11.3
```



Information

The `which node` command returns the application that the operating system is using. The `node -v` command returns the version of the Node.js runtime environment that is installed in your workstation.

- ___ 3. Verify the version of the “npm” Node package manager application.

```
$ npm -v  
3.10.10
```

- ___ 4. Verify the installation of the `apiconnect` global package.

```
$ npm list --global --json apiconnect  
{  
  "dependencies": {  
    "apiconnect": {  
      "version": "2.7.62",  
      "from": "apiconnect@apic-v5.0.8.1",  
      "resolved": "https://registry.npmjs.org/apiconnect/-/apiconnect-2.7.62.tgz"  
    }  
  }  
}
```



Information

The `npm list` command reads the metadata information from a package that is installed in your workstation. In this example, your global npm repository has a copy of the IBM API Connect toolkit, version 2.7.62.

3.2. Create a LoopBack application with the apic command-line utility

The LoopBack application is a Node.js application that implements a REST API based on a set of models. Generate a sample LoopBack application with the apic command-line utility.

- ___ 1. Create a directory to hold the contents of the LoopBack application.

- ___ a. Open a terminal window.
 - ___ b. Create two directories: `samples` and `samples/notes`

```
$ cd ~
$ mkdir samples samples/notes
$ cd samples/notes
```

- ___ c. Verify that your current working directory is the `sample/notes` directory.

```
$ pwd
/home/student/samples/notes
```

- ___ 2. Create a copy of the “notes” sample application.

- ___ a. Run the LoopBack application generator in the `apic` command-line utility.

```
$ apic loopback
```

- ___ b. Select **notes** as the application name.

- ___ c. Select the **notes (A project containing a basic working example, including a memory database)** project as the application starting point.

? What's the name of your application (notes)? **notes**

? Which version of LoopBack would you like to use? 3.x current (Use arrow keys)

 2.x (long term support)

> 3.x (current)

? What kind of application do you have in mind? (Use arrow keys)

 empty-server (An empty LoopBack API, without any configured models or data sources)

 hello-world (A project containing a controller, including a single vanilla Message and a single remote method)

> **notes (A project containing a basic working example, including a memory database)**

- ___ d. Wait until the generator finishes creating the application.

3.3. Examine the structure of a LoopBack application

Before you test the sample “notes” LoopBack application, explore the structure of the application scaffolding. Review the API definition files, the configuration settings, the model properties, and the model source code.

- ___ 1. Review the structure of the application.
 - ___ a. List the contents of the root directory in the “**notes**” application.

```
$ ls -F
common/
definitions/
node_modules/
package.json
server/
```



Information

The application generator created the directory structure for a LoopBack application:

- The **common** directory stores resources that the client and server application uses.
- The **definitions** directory stores the API interface and API product definition files for the LoopBack application. The interface definitions follow the OpenAPI specification, an open source specification that defines REST API structures. The product definition file is a format that is specific to IBM API Connect.
- The **server** directory stores the configuration settings, boot scripts, and data source settings for the server application.

Every LoopBack application is a Node.js application. Therefore, the **notes** application includes a **node_modules** directory to store local Node packages. The **package.json** application manifest stores metadata on the application, including name, version, package dependencies, and software licensing information.

- ___ b. Examine the contents of the **common** directory.

```
$ ls -F common
models/
$ ls -F common/models/
note.js  note.json
```



Information

At the core of a LoopBack application are the models: JavaScript configuration and code that represent the data models in the API. The LoopBack framework creates a set of data-centric create, retrieve, update, and delete operations for each model. In a later exercise, you explore how LoopBack connectors persist model data through data sources.

- ___ c. Examine the contents of the `server` directory.

```
$ ls -F server
boot/
datasources.json
middleware.development.json
server.js
config.json
middleware.json
model-config.json
```



Information

The `server` directory contains a set of configuration files that control the behavior of the LoopBack framework.

- The `datasources.json` file defines a list of LoopBack data sources. A data source is an object that provides programmatic access to a remote data source: a database, a remote service, or email server.
- The `middleware.json` file defines configuration settings for Express middleware modules. Express is a Node.js framework for web applications. LoopBack is built upon the Express framework.
- The `middleware.development.json` file defines environment properties for Express middleware modules.
- The `server.js` script is the main entry point to the LoopBack application. This script loads the LoopBack framework with the configuration files.
- The `config.json` file sets general parameters for the LoopBack application, such as the base path for the REST APIs.
- The `model-config.json` file maps each LoopBack module to a specific data source. The application stores the models in the `common/models` directory.

In a later exercise, you install and configure data sources with server configuration files. You also map LoopBack models that you create to data sources.

- ___ d. Examine the contents of the `server/boot` directory.

```
$ ls -F server/boot
authentication.js
root.js
$ more server/boot/root.js
'use strict';

module.exports = function(server) {
  // Install a '/' route that returns server status
  var router = server.loopback.Router();
  router.get('/', server.loopback.status());
  server.use(router);
};
```



Information

The **boot scripts** directory contains Node scripts that customize the LoopBack application behavior. In this example, the **root.js** script defines a default route that returns the LoopBack object status. In practice, you open this route at run time to confirm that the LoopBack application is running.

- ___ 2. Review the LoopBack “**note**” model configuration and source code.

- ___ a. Examine the contents of the `common/models` directory.

```
$ ls common/models
note.js  note.json
```

- ___ b. Review the contents of the `note.json` model property file.

```
$ more common/models/note.json
{
  "name": "Note",
  "properties": {
    "title": {
      "type": "string",
      "required": true
    },
    "content": {
      "type": "string"
    }
  }
}
```

- ___ c. Review the contents of the `note.js` model script file.

```
$ more common/models/note.js
module.exports = function(note) {
};
```



Information

Two files define the properties and behavior of a LoopBack model:

- The **model configuration file** declares the name, properties, and relationships in the model object. You create a configuration file in the JavaScript object notation (JSON) format.
- The **model script file** defines any additional behavior beyond the standard create, retrieve, update, and delete operations that the LoopBack framework provides for every model object. You develop a model script file as a Node.js JavaScript anonymous function.

In this application, the **note.json** model configuration defines “Note” as the name of the model object. The “Note” model defines two properties: “title” and “content”. Both properties store information in the JSON string format. The “title” is a required property.

The **note.js** model script file defines an empty anonymous function. By default, every LoopBack model object inherits its behavior from the model base class. Therefore, you do not need to implement any custom code to create data-centric REST API operations. You can implement remote methods or override the default operations with custom code in this script file.

___ 3. Review the in-memory database data source.

___ a. Examine the in-memory data source that is defined in the **server/datasources.json** configuration file.

```
$ more server/datasources.json
{
  "db": {
    "name": "db",
    "connector": "memory"
  }
}
```



Information

The **datasources.json** configuration file declares the name of each data source in the LoopBack application. Recall that a data source is a JavaScript object that represents an external data store. Examples include relational databases, non-relational data stores, and remote services.

In this example, the LoopBack application creates a data source that is named `db`. The data source maps to the “memory” LoopBack connector: an in-memory data store that persists model object data while the application is running for testing purposes.

- __ b. Examine the model mapping to data sources in the `server/model-config.json` configuration file.

```
$ more server/model-config.json
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  "User": {
    "dataSource": "db"
  },
  "AccessToken": {
    "dataSource": "db",
    "public": false
  },
  "ACL": {
    "dataSource": "db",
    "public": false
  },
  "RoleMapping": {
    "dataSource": "db",
    "public": false,
    "options": {
      "strictObjectIDCoercion": true
    }
  },
  "Role": {
    "dataSource": "db",
    "public": false
  },
  "Note": {
    "dataSource": "db"
  }
}
```



Information

The `model-config.json` configuration file maps LoopBack models in your application to a data source. In this example, the application mapped the “Note” model object to the “db” data source. Recall that the `datasources.json` configuration file created an in-memory database that is named `db`.

You can map each model to one data source. If you do not want to persist model data, or link a model to a data source, leave out a model mapping in the `model-config.json` file.

The “`_meta`” and “`mixins`” sections of the configuration file are preset values that the LoopBack framework code expects. Leave this section unchanged.

___ 4. Hide the API operations for the `user` model.

___ a. Open the `model-config.json` file with the Mousepad application.

```
$ mousepad server/model-config.json
```

___ b. Add a property named `public` to the `user` object.

___ c. Set the value of the `public` property to `false`.

```
"User": {
  "dataSource": "db",
  "public": false
}
```

The screenshot shows a window titled “*model-config.json - Mousepad”. The menu bar includes File, Edit, View, Text, Document, Navigation, and Help. The main content area displays the JSON configuration file:

```
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  "User": {
    "dataSource": "db",
    "public": false
  }
}
```

___ d. Save the changes to the `model-config.json` file.

___ e. Close the Mousepad application.

- ___ f. In the terminal emulator window, enter `apic loopback:refresh` to update the list of operations in the API definition file.

```
$ cd ~/samples/notes  
$ pwd  
/home/student/samples/notes  
$ apic loopback:refresh
```



Questions

Why did you hide the API operations for the `user` model?

The `user` model is a built-in object in the LoopBack framework. With the model, you can configure role-based access control to your API. For this exercise, you configure `notes` as a publicly accessible service with no security restrictions. Therefore, you can safely hide the `user` API operations.

For more information about the User REST API, see the LoopBack documentation:

<http://loopback.io/doc/en/lb3/User-REST-API.html>

3.4. Review the API definition in the API Designer web application

The **API definition** declares the interface of an API. The definition file specifies the API routes, methods, request, and response messages for each API operation. In this role, the API definition lists a set of operations that application developers can call.

The second role of the API definition file is to configure message processing rules and environment variables. The message processing policies specify how an API gateway transforms and routes API request and response messages. Environment variables set application server hosts.

Every API that you create in IBM API Connect must include at least one API definition file.

In this section, review the API definition with the API Designer web application. Examine the interface definition.

- 1. Start the API Designer web application.
 - a. From the terminal, verify that the current directory is the base of the sample “notes” application.


```
$ pwd
/home/student/samples/notes
```
 - b. Run the apic edit command.


```
$ apic edit
```
 - c. Wait until the API Designer application opens in your web browser.
- 2. Examine the layout of the API Designer web application.

TITLE	LAST MODIFIED	TYPE
notes 1.0.0 [notes.yaml]	2 minutes ago	REST



Information

The first two sections of the API Designer list the API definitions and products that you created in the `definitions` directory.

- A **product** groups one or more API definitions in IBM API Connect. You must add an API definition to a product before you publish the API.
- An **API definition** represents the interface for a set of API operations. IBM API Connect declares APIs in the **OpenAPI specification** format.

-
- ___ 3. Open the `notes.yaml` API definition document.
 - ___ a. Click the **API** tab in the API Designer web application.
 - ___ b. Select **notes 1.0.0**.
 - ___ 4. Review the “notes” API definition on the API Editor Design tab.

The screenshot shows the IBM API Connect interface. At the top, it displays "IBM API Connect" and "notes - notes 1.0.0". Below the header, there are tabs: "All APIs" (disabled), "Design" (selected), "Source", and "Assemble". On the left, a sidebar menu includes "Info", "Schemes", "Host", "Base Path", "Consumes", "Produces", "Lifecycle", "Policy Assembly", and "Security Definitions". The main content area is titled "Info" and contains fields for "Title *" (set to "notes") and "Name *" (set to "notes").



Information

In the **API Editor** view in API Designer, you can review and edit the API definition in one of three options:

- The **Design** tab opens the API definition document in a web form. The Design tab helps you enter information with the correct syntax.
- The **Source** tab opens the API definition file in the original text format. The API definition file uses the YAML (Yet Another Markup Language) text format. The fields of the API definition file follow the OpenAPI 2.0 specification. If you are familiar with the OpenAPI specification, switch to this view.
- The **Assemble** tab opens the message processing policy section of the API definition document in a graphical editor. You can also view the “assembly” section of an API definition document on the **Source** tab.

For more information about the structure of an API definition file, see the OpenAPI specification at: <https://github.com/OAI/OpenAPI-Specification>

- ___ a. Examine the **Info** section of the “Notes” API definition.

The **Info** section stores the name, version, and description of the API. This section also holds metadata on the API, such as licensing restrictions, external sources of documentation, and contact information.

- ___ b. Examine the **Host** and **Base Path** sections.

The screenshot shows the API definition interface with the sidebar menu open. The 'Host' section is selected, displaying the value '\$(catalog.host)'. The 'Base Path' section is also visible, showing the value '/api'.

Information

The **host** property stores the host name or IP address of the server that hosts the published APIs. In an API Connect solution, the host variable is the address of the API gateway. The **\$catalog.host** value is an environment variable that IBM API Connects assigns to the API definition when you publish your API.

The **Base Path** property sets the parent URL path for all API operations. In this example, the base path is set to `/api`.

- ___ c. Examine the **Consumes** and **Produces** sections.

The screenshot shows the API definition interface with the sidebar menu open. The 'Consumes' section is selected, showing 'application/json' checked under 'Consumes'. The 'Produces' section is also visible, showing 'application/json' checked under 'Produces'.



Information

The **Consumes** and **Produces** sections declare the media type in the API request message body and response message body. These two sections define how to encode an API request message, and what data formats to expect in the response message.

In this example, your application must send request messages with an `application/json` data format in the message body. If the API operation returns a response, the API implementation returns data in the `application/json` format.

The **Consumes** and **Produces** sections set the default media type for all API operations. You can override the media type for a particular API operation in the **Paths** section.

- ___ d. Examine the **Paths** section. Then, expand the `/Notes` path.

Method	Path	Status	Action
POST	/Notes	Deprecated	<input type="checkbox"/> Delete
PATCH	/Notes	Deprecated	<input type="checkbox"/> Delete
PUT	/Notes	Deprecated	<input type="checkbox"/> Delete
GET	/Notes	Deprecated	<input type="checkbox"/> Delete



Information

The **Paths** section lists the resource paths in the API. Each API operation consists of a resource path and an HTTP method. For example, the `/notes` path represents a **notes** data model object on the server.

- To create an instance of the **notes** object, send an HTTP **POST** request to `/notes`.
- To update the fields of a **notes** object, send an HTTP **PUT** or **PATCH** request to `/notes`.
- To retrieve the contents of a particular **notes** object, send an HTTP **GET** request to `/notes`.

Select each one of the four API operations to examine the expected request message, response message, HTTP status code, and URL parameters.

- ___ e. Review the **Parameters** section.

The screenshot shows the LoopBack interface. On the left, there's a sidebar with a navigation menu. The 'Parameters' option is highlighted with a blue bar at the top. Below it are 'Definitions', 'Note', 'x-any', and 'Tags'. The main content area is titled 'Parameters' and displays the message 'No parameters defined'.



Information

The **Parameters** section defines schemes for passing input parameters to API operations. For example, query parameters appear after a question mark (?) in the URL path. In this example, none of the “Notes” API operations use parameters.

- ___ f. In the **Definitions** section, select **Note** to review its properties.

The screenshot shows the LoopBack interface. On the left, there's a sidebar with a navigation menu. The 'Definitions' option is highlighted with a blue bar at the top. Below it are 'Parameters', 'Paths', and a list of API paths: '/Notes', '/Notes/replaceOrCreate', '/Notes/upsertWithWhere', '/Notes/{id}/exists', '/Notes/{id}', '/Notes/{id}/replace', '/Notes/findOne', '/Notes/update', '/Notes/count', and '/Notes/change-stream'. The main content area is titled 'Definitions' and shows a single entry for 'Note'. It includes fields for 'Name' (set to 'Note'), 'Type' (set to 'object'), and 'Description'. Below this, under 'Properties', there are two entries: 'title' (checkbox checked, type 'string') and 'content' (checkbox unchecked, type 'string').



Information

The **Definitions** section describes the structure of data objects that you send to or receive from API operation calls. For example, when you call GET /items, you retrieve a collection of “Notes” objects from the server. The “Notes” object consists of three properties: **title**, **content**, and **id**. The title and content properties are string data types, while the id property is a double data type. The id field is generated and not meant to be writable. The default settings between Loopback v2 and v3 might differ. For more information, see: <https://loopback.io/doc/en/lb3/Model-definition-JSON-file.html>

A one-to-one correlation exists between the structure of your LoopBack model objects and the data type definitions in this section of the API definitions file.

- ___ 5. In the API Editor view, click the **Source** tab.

The screenshot shows the IBM API Connect interface for a "notes - notes 1.0.0" API. The top navigation bar includes "IBM API Connect", the API name, and a "Publish" button. Below the navigation is a toolbar with tabs: "All APIs", "Design", "Source" (which is highlighted with a red box), and "Assemble". The left sidebar contains sections like "Extensions", "Properties", "Paths", "Analytics", "Parameters", "Definitions", and "Note". The main content area displays the API's OpenAPI specification. A vertical scrollbar is visible on the right side of the code editor. The code itself is a JSON-like structure for the "/Notes" endpoint, defining operations like POST, PUT, and PATCH, along with their parameters and responses.

```

paths:
  /Notes:
    post:
      tags:
        - Note
      summary: Create a new instance of the model
      operationId: Note.create
      parameters:
        - name: data
          in: body
          description: Model instance data
          required: false
          schema:
            description: Model instance data
            $ref: '#/definitions>Note'
      responses:
        '200':
          description: Request was successful
          schema:
            $ref: '#/definitions>Note'
          deprecated: false
    patch:
      tags:
        - Note
      summary: Patch an existing model instance or
      operationId: Note.patchOrCreate
      parameters:
        - name: data
          in: body
          description: Model instance data
          required: false
          schema:
            description: Model instance data
  
```



Information

The **Source** tab displays the API definition in its original format. The definition file follows a structured text format that is named YAML, or “Yet Another Markup Language”. If you are familiar with the different sections of the OpenAPI specification, then the source format might be easier to view.

If you change values on the **Source** tab, the **Design** tab also reflects those changes.

Keep in mind that the YAML file format and the OpenAPI specification are two separate topics. Not all YAML files are OpenAPI specifications. The YAML file format is an increasingly popular alternative to XML-based configuration files.

- ___ 6. Open the `notes-product.yaml` API product document.
 - ___ a. Select **All APIs** to return to the main page of the API Designer.
 - ___ b. Select **Products**.
 - ___ c. On the **Products** tab, click **Notes 1.0.0 [notes-product.yaml]**.

The screenshot shows the IBM API Connect interface. At the top, there's a dark header bar with the IBM API Connect logo and the word "notes". Below it is a navigation bar with four tabs: "Products" (which is selected and highlighted in blue), "APIs", "Models", and "Data Sources". In the center, there's a search bar with a magnifying glass icon and the placeholder text "Search products". Below the search bar, there's a button labeled "Add +". Underneath, there's a table-like structure with a single row. The first column is labeled "TITLE" and contains the text "notes 1.0.0 [notes-product.yaml]". This entire row is highlighted with a red box.

- ___ d. Examine the **Info** section.

The screenshot shows the API Designer interface with the "Design" tab selected. On the left, there's a sidebar with several options: "All Products", "Design" (selected and highlighted in blue), "Source", "Explore", and "Settings". Under "Design", there are sub-options: "Info", "Contact", "License", "Terms of Service", "Visibility", "APIs", and "notes 1.0.0". The "Info" option is currently selected and highlighted with a blue box. To its right, there's a detailed "Info" section with the following fields and values:

Title	notes
Name	notes
Version	1.0.0



Information

The **info** section lists the name, version, description, license, and terms of service for the API product. This section has the same structure as the one in an API definition.

- ___ e. Examine the **API** section.

The screenshot shows the API Designer interface. On the left, there's a sidebar with tabs: 'Visibility', 'APIs' (which is highlighted with a blue border), 'Plans', and 'Default Plan'. The main panel is titled 'APIs' and contains a single entry: 'notes 1.0.0'. Below this entry is a small note: 'notes 1.0.0'.



Information

In this example, the “notes” API product contains exactly one API: the “notes” API definition that you examined earlier.

The remaining sections of an API product are **visibility**, **APIs**, and **plans**. Visibility determines whether application developers can view or subscribe to APIs after you publish the product. The API section lists the API definitions that you associated with the API product. The plans state the service level that application developers expect when they call API operations in a product.

You examine the concepts of products, API visibility, and plan subscriptions in a later exercise.

- ___ f. Select **All Products** to return to the API Designer main page.

The screenshot shows the API Designer interface with the 'All Products' tab selected in the top navigation bar. The top bar also includes 'Design', 'Source', and 'Explore' buttons. The left sidebar is titled 'Info' and contains links to 'Contact', 'License', and 'Terms of Service'. The main panel is also titled 'Info' and shows two fields: 'Title' with the value 'notes' and 'Name' with the value 'notes'.

3.5. Examine the LoopBack model and data source with API Designer

In the API Designer main page, the last two tabs help you develop your LoopBack application without command-line tools or coding:

- The **models** section lists the LoopBack models that you created in the `common/models` directory of your application.
- The **data sources** section lists the data sources that you defined in the `servers/datasources.json` configuration file.

In this section, review the options available on these tabs.

- 1. Examine the “**Note**” LoopBack model configuration in the API Editor.
 - a. From the API Designer main page, click the **Models** tab.
 - b. In the list of models, select **Note**.
 - c. Examine the **name**, **data source**, and **properties** fields of the “Note” LoopBack model.

The screenshot shows the 'Models' tab in the API Designer. At the top, there's a navigation bar with 'All Models' (with a back arrow), 'Explore', 'Settings', and a refresh icon. Below the navigation is a search bar with placeholder 'Name' and a dropdown menu currently set to 'Note'. There are sections for 'Plural' (set to 'Note'), 'Base Model' (empty), 'Data Source' (set to 'db'), and checkboxes for 'Public' (checked) and 'Strict' (unchecked). A large button at the bottom right has a plus sign and a '+' icon. Below this is a table titled 'Properties' with columns: Required, Property Name, Is Array, Type, ID, Index, and Description (optional). Two properties are listed: 'content' (checkbox unchecked, string type, ID checked, Index checked) and 'title' (checkbox checked, string type, ID unchecked, Index unchecked).

Required	Property Name	Is Array	Type	ID	Index	Description (optional)
<input type="checkbox"/>	content	<input type="checkbox"/>	string	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/>	title	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>	



Information

The **model** page lists the structure of the LoopBack model in your application. In this example, the name of the model is: `Note`

The base model field is blank, indicating that the “Note” model extends the base model class. The “Note” model is bound to the “db” LoopBack data source.

The two property fields are **content** and **title**. Both properties are JavaScript strings. The title property is a mandatory field.

-
- ___ 2. Examine the “memory” LoopBack data source configuration in the API Editor.
 - ___ a. Select **All models** to return to the API Designer main page.
 - ___ b. Click the **Data Sources** tab.
 - ___ c. Open the “db” LoopBack data source.

The screenshot shows the 'Data Sources' tab in the LoopBack API Designer. On the left, there's a sidebar with a back arrow labeled 'All Data Sources' and a save icon. The main area has a form with the following fields:

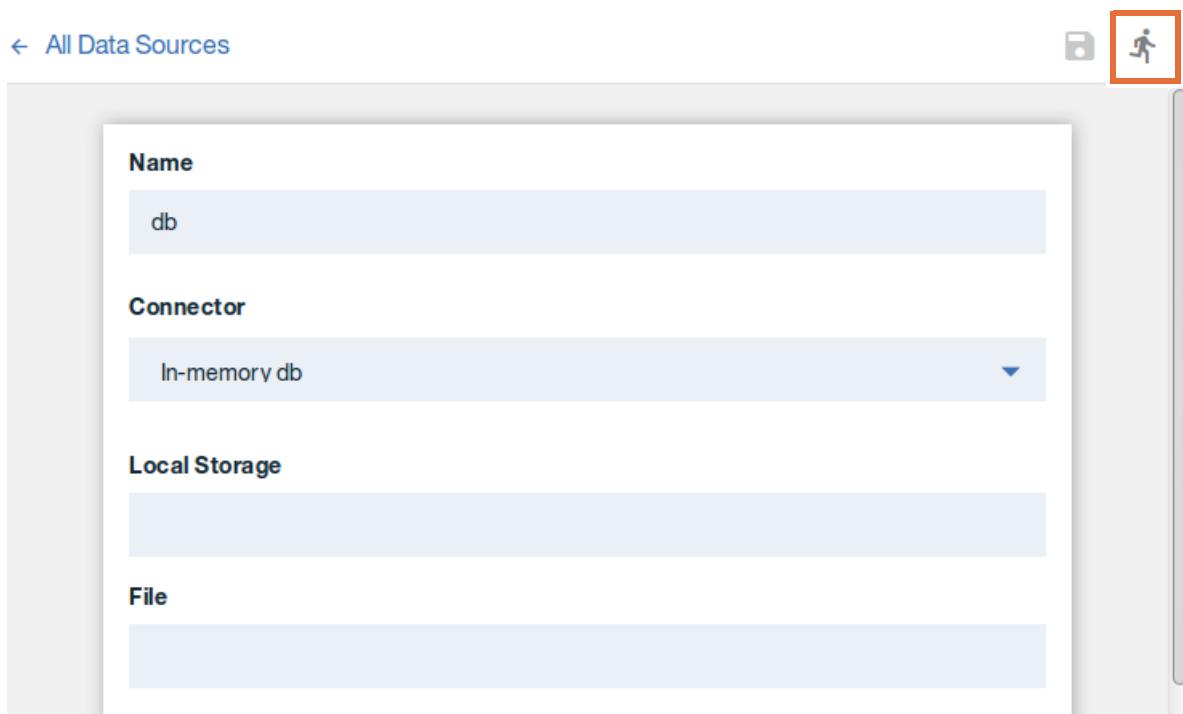
- Name:** db
- Connector:** In-memory db (with a dropdown arrow)
- Local Storage:** (empty input field)
- File:** (empty input field)



Information

The **data source** page lists the details for the “db” in-memory data source that the application defined in the `server/datasources.json` configuration file. This data source uses the in-memory database connector to store model data. The connector itself is part of the LoopBack framework: you do not need to download any additional packages.

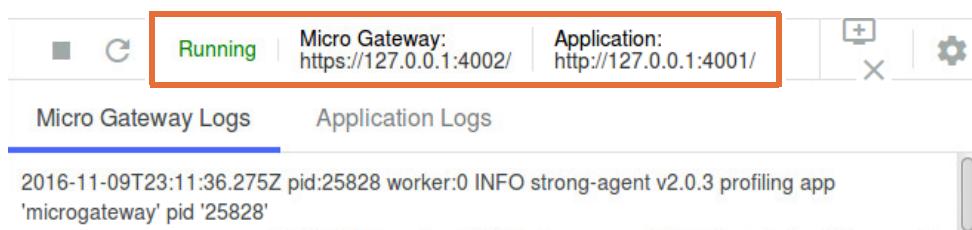
- ___ 3. Test the connection to the in-memory data source.
- ___ a. With the **db** connector open in the editor, click the Test Data Source Connection icon.



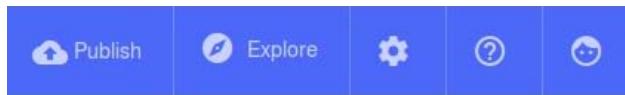
- ___ b. You see a message "Data source connection test succeeded".
- ___ c. Click **All Data Sources**.
- ___ 4. Start the **notes** application.
- ___ a. From the application control at the bottom of the API Designer page, click **Start**.



- ___ b. Wait until the **Micro Gateway** and **Application** are both started.



- ___ 5. Open the **Explore** tab in the API Designer web application.
- ___ a. From the toolbar, click **Explore** to examine the API.



- ___ b. Examine the API Explorer web interface.

A screenshot of the API Explorer interface. On the left, there's a sidebar titled 'Operations' with a dropdown menu set to 'by.name'. Below it is a list of operations: Note.create, Note.patchOrCreate, Note.replaceOrCreate__put_Notes, Note.find, Note.replaceOrCreate__post_Notes_rep..., Note.upsertWithWhere, Note.exists__get_Notes_{id}_exists, Note.exists__head_Notes_{id}, Note.findById, and Note.replaceById__put_Notes_{id}. The main area shows the 'notes 1.0.0' API definition. The 'Note.create' operation is highlighted. The middle column contains its documentation: 'Create a new instance of the model and persist it into the data source.' and the right column shows the REST method 'POST https://localhost:4002/api/Notes'. Below it are other operations: 'Note.patchOrCreate' (PATCH https://localhost:4002/api/Notes), 'Note.replaceOrCreate__put_Notes' (PUT https://localhost:4002/api/Notes), and 'Note.find' (GET https://localhost:4002/api/Notes).

Information

The **Explore** tab in the API Designer web application consists of three columns:

- The left column lists the API operations by name from the API definition. In this example, the **notes 1.0.0** API definition declares a set of operations such as `Note.create` and `Note.find` on the `/Notes` resource path.
- The middle column displays the documentation for the API operation that you selected.
- The right column provides interactive testing tools for the selected API. The REST operations are displayed.

- ___ 6. Test the **POST /notes** API operation from the Explore page.
- ___ a. Click the `POST https://localhost:4002/api/Notes` operation in the right column.

POST https://localhost:4002/api/Notes

Examples Try it

Example request

curl

```
curl --request POST \
--url https://localhost:4002/api/Notes \
--header 'accept: application/json' \
--header 'content-type: application/json' \
--header 'x-ibm-client-id: default' \
--header 'x-ibm-client-secret: SECRET' \
--data '{"title": "kadag", "content": "horvibbe"}'
```

- You see an example for calling the POST operation with `curl`.
- ___ b. Click **Try it**.
- ___ c. Scroll down on the right column until the **data** section appears.
- ___ d. Type the values in the data area:

```
{
  "title": "Buy eggs and milk"
}
```

Accept

application/json

Parameters

data	Generate
<pre>{ "title": "Buy eggs and milk" }</pre>	Generate

Call operation

The identifier (`id`) field is not included, and the LoopBack connector assigns a unique identifier when it creates a model instance.

- ___ e. Click **Call operation**.
- ___ f. Confirm that the API operation completes successfully.

```

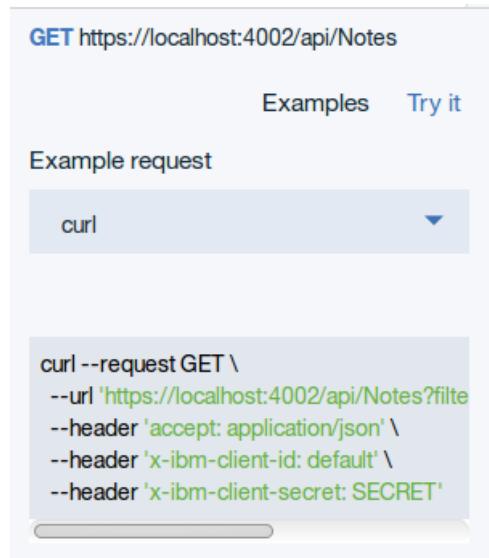
Request
POST https://localhost:4002/api/Notes
Headers:
Content-Type: application/json
Accept: application/json
X-IBM-Client-Id: default
X-IBM-Client-Secret: SECRET

Response
Code: 200 OK
Headers:
content-type: application/json;
charset=utf-8
x-ratelimt-limit: 100
x-ratelimt-remaining: 99

{
  "title": "Buy eggs and milk",
  "id": 1
}

```

- ___ 7. Review the data that the LoopBack connector saves in memory.
- ___ a. On the **Explore** tab of the API Designer, click the `GET https://localhost:4002/api/Notes` operation in the right column.



- ___ b. Click **Try it**.
- ___ c. Click **Call operation**.

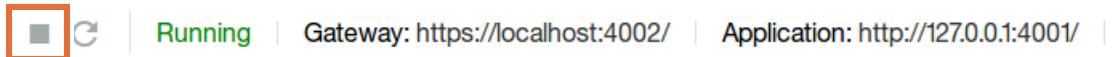
- ___ d. Confirm that the API operation completes successfully and returns the same values that were posted earlier.

Response

```
Code: 200 OK
Headers:
content-type: application/json;
charset=utf-8
x-ratelimt-limit: 100
x-ratelimt-remaining: 99

[
{
  "title": "Buy eggs and milk",
  "id": 1
}]
```

- ___ 8. Stop the servers.



End of exercise

Exercise review and wrap-up

In the first part of the exercise, you created a copy of the sample “Notes” LoopBack application with the `apic` command-line utility. You reviewed the API definition, models, and data source with the API Designer.

In the second part of the exercise, you started and tested the LoopBack application with the Explore page in the API Designer web application. You called API operations from the web-based test client.

Exercise 4. Defining LoopBack data sources

Estimated time

01:00

Overview

In this exercise, you bind the model to relational and non-relational databases with data sources. You define relationships between models. Finally, you test API operations with the API Explorer.

Objectives

After completing this exercise, you should be able to:

- Install and configure the MySQL connector
- Install and configure the MongoDB connector
- Generate models and properties from a data source
- Define relationships between models
- Test an API with the API Explorer

Introduction

The Node.js runtime environment consists of an interpreter for the JavaScript programming language, and a library to build server-side web applications. By its design, the library is minimalist. You install the software packages that you choose to build your application.

The LoopBack framework provides a robust set of libraries to quickly build REST APIs based on model objects. In the previous exercise, you examined how to generate, customize, and test a LoopBack application that saved its data in memory.

In this exercise, you build a second application that persists its model data to data stores: MySQL and MongoDB. The MySQL database is a relational database that holds its data in tables. The MongoDB database is a document-based non-relational database: it stores its data in documents that do not conform to a set schema.

Requirements

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

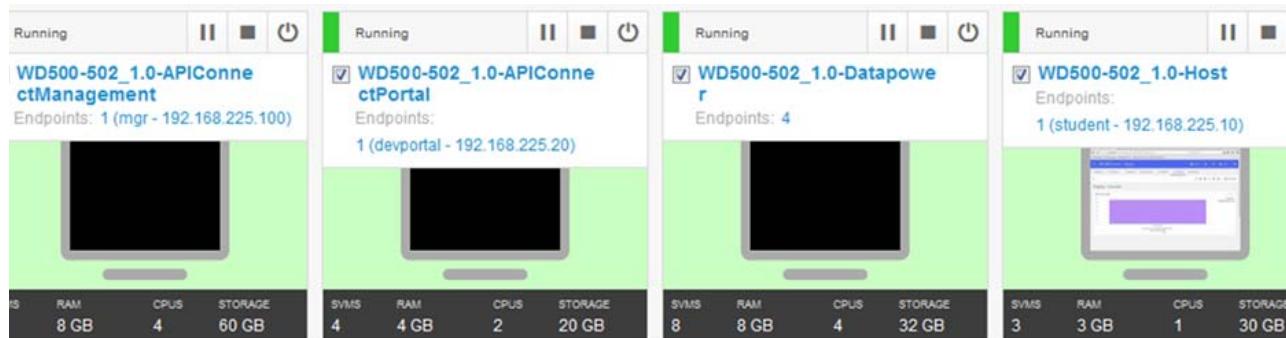
Exercise instructions

Before you begin

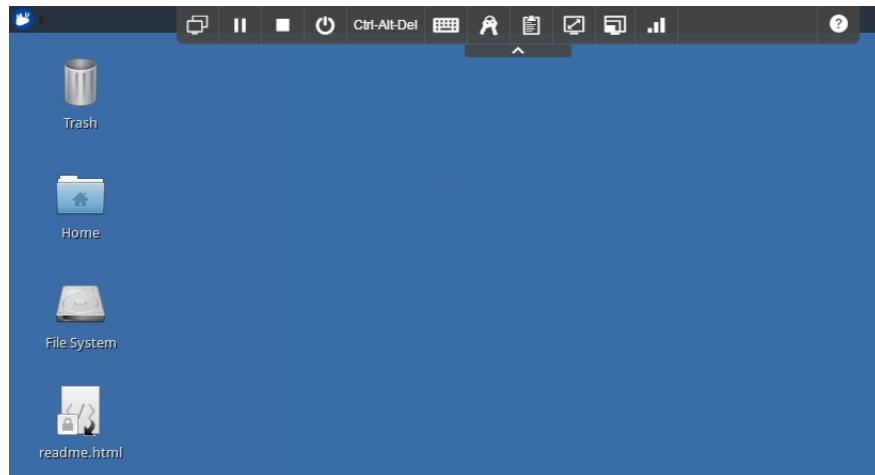
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



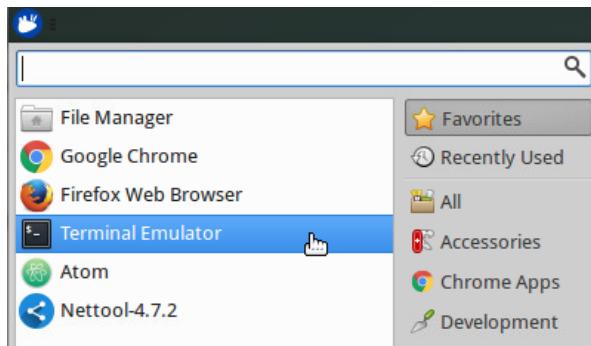
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

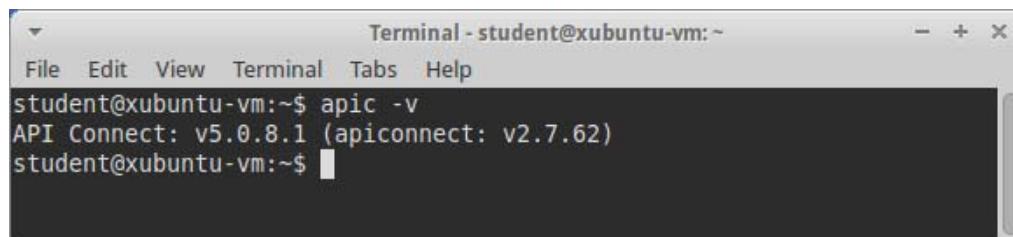
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

4.1. Create the inventory application and MySQL data source

In this section, create the directory structure and configuration files for the inventory LoopBack application. This application manages a set of descriptions for memorable items from IBM's history. The LoopBack framework provides REST API access to the inventory items through the model class.

- ___ 1. Open the terminal emulator application.
 - ___ a. From the start menu, click **Terminal Emulator**.
- ___ 2. Create a directory for the application, named **inventory**, in the student home directory.
 - ___ a. In the home directory, create a directory that is named **inventory**.


```
$ mkdir ~/inventory
$ cd inventory
$ pwd
/home/student/inventory
```
- ___ 3. Generate the directory structure and configuration files for an empty server LoopBack application.
 - ___ a. Run the **apic loopback** command in the **inventory** directory.
 - ___ b. Create an **empty-server** LoopBack application.


```
$ apic loopback
? What's the name of your application (inventory)? inventory
? Which version of LoopBack would you like to use? 3.x
  2.x (long term support)
> 3.x (current)
? What kind of application do you have in mind? (Use arrow keys)
> empty-server (An empty LoopBack API, without any configured models or
data sources)
    hello-world (A project containing a controller, including a single
    vanilla Message and a single remote method)
    notes (A project containing a basic working example, including a memory
    database)
```
- ___ c. Confirm that the “apic” utility generated the LoopBack application successfully.
- ___ 4. Install the LoopBack MySQL connector into the inventory application.
 - ___ a. Install the **loopback-connector-mysql** npm package.


```
$ npm install --save loopback-connector-mysql
```

- ___ b. Examine the **dependencies** section of the project manifest file.

```
$ more package.json
...
"dependencies": {
  "compression": "^1.0.3",
  "cors": "^2.5.2",
  "helmet": "^1.3.0",
  "loopback": "^3.0.0",
  "loopback-boot": "^2.6.5",
  "loopback-connector-mysql": "^5.2.0",
  "serve-favicon": "^2.0.1",
  "strong-error-handler": "^2.0.0"
},
...
```

- ___ c. Confirm that the manifest file lists the **loopback-connector-mysql** npm package.



Information

The LoopBack **connector** is a software package that creates a **data source**: a Node.js object that manages access from the LoopBack framework to a data store.

Before you begin, you must download and install the correct connector for your data source. Use the “npm” utility to retrieve the most recent version of the LoopBack connector.

- ___ 5. Define a LoopBack data source that is named `mysql-connection` with the “apic” command-line utility.

- ___ a. Create a **data source** object with the **apic** command-line utility.

```
$ apic create --type datasource
```

- ___ b. Enter the following properties for a data source that is named **mysql-connection**:

- Data source name: `mysql-connection`
- Connector: MySQL (supported by StrongLoop)
- Connection String url to override other settings: Leave blank (Enter)
- Connector-specific configuration:
 - o Host: `mysql.think.ibm`
 - o Port: 3306
 - o User: student
 - o Password: `Passw0rd!`
 - o Database: think

```
? Enter the data-source name: mysql-connection
? Select the connector for mysql-connection:
  > MySQL (supported by StrongLoop)
Connector-specific configuration:
? Connection String url to override other settings: <leave blank>
? host: mysql.think.ibm
? port: 3306
? user: student
? password: Passw0rd!
? database: think
```

- ___ c. Review the changes to the `server/datasources.json` configuration file.

```
$ more server/datasources.json
{
  "mysql-connection": {
    "host": "mysql.think.ibm",
    "port": 3306,
    "url": "",
    "database": "think",
    "password": "Passw0rd!",
    "name": "mysql-connection",
    "user": "student",
    "connector": "mysql"
  }
}
```



Information

The `apic create` command adds a data source definition to your LoopBack application. The data source relies on the LoopBack connector: an npm package that implements the data source object. You can modify the settings directly in the `server/datasources.json` configuration file.

- ___ 6. Examine the **mysql-connection** data source from the API Designer web application.

- ___ a. Open the API Designer application from the inventory directory.

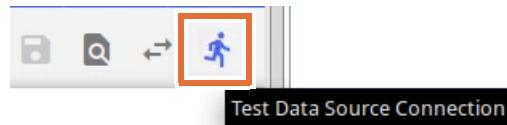
```
$ pwd
/home/student/inventory
$ apic edit
```

- ___ b. From the API designer main page, click the **Data Sources** tab.

- ___ c. Select the **mysql-connection** data source.

The screenshot shows the IBM API Connect interface with the title bar 'IBM API Connect inventory'. Below the title bar, there are tabs for 'Products', 'APIs', 'Models', and 'Data Sources', with 'Data Sources' being the active tab. Underneath the tabs is a search bar with the placeholder 'Search data sources'. A large list area is titled 'Name' and contains a single item: 'mysql-connection', which is highlighted with a mouse cursor.

- ___ d. Review the **mysql-connection** data source settings.
___ e. Click the **Test Data Source Connection** at the upper-right side of the toolbar.



- ___ f. Confirm that the data source test completes successfully.

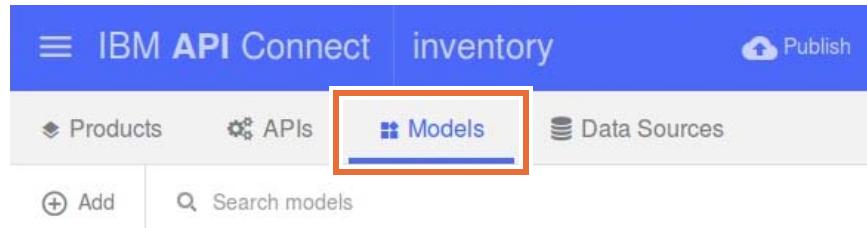


- ___ g. Click **All Data Sources** to return to the landing page.

4.2. Create the item LoopBack model

The **item** model object represents an entry from the **inventory** MySQL database. In this section, define the properties in the **item** model. Map the **item** table in the **inventory** MySQL database to the LoopBack model.

- ___ 1. Create a LoopBack model named **item**.
 - ___ a. Click **Models** from the main toolbar.



- ___ b. Click **Add** to create a model object.
- ___ c. Enter **item** as the model name.

 A screenshot of a modal dialog box titled 'New LoopBack Model'. It contains a single input field labeled 'Name' with the value 'item'. At the bottom right are two buttons: 'Cancel' and a larger blue 'New' button.

- ___ d. Click **New**.
- ___ 2. Set the data source to **mysql-connection**.
 - ___ a. In the item model page, select the **data source** field.

- ___ b. Select **mysql-connection** from the menu.

The screenshot shows the configuration interface for a LoopBack model named 'item'. The 'Data Source' field is highlighted with a red box, containing the value 'mysql-connection'. Below it, there are two options: 'Public' (checked) and 'Strict' (unchecked).

Name	item
Plural	
Base Model	PersistedModel
Data Source	mysql-connection
	<input checked="" type="checkbox"/> Public
	<input type="checkbox"/> Strict

- ___ c. Enable the **Public** option.
___ d. Save your changes.
- ___ 3. Define the **name**, **description**, **img**, **img_alt**, **price**, and **rating** properties.
- ___ a. In the item model page, scroll down to the **Properties** section.

- ___ b. Select the plus sign (+) to add a property.

Required	Property Name	Is Array	Type	ID	Index	Description (optional)
<input checked="" type="checkbox"/>		<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>	item name

- ___ c. Enter the following details for the property:

- Required: **yes**
- Property name: **name**
- Type: **string**
- Description: **item name**

Required	Property Name	Is Array	Type	ID	Index	Description (optional)
<input checked="" type="checkbox"/>	name	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>	item name

- ___ d. Enter the remaining five properties for the **item** model.

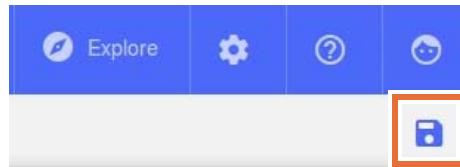
Table 5. **Item** model properties

Required	Property name	Type	Description
Yes	name	string	item name
Yes	description	string	item description
Yes	img	string	location of item image
Yes	img_alt	string	item image title
Yes	price	number	item price
No	rating	number	item rating

__ e. Confirm that your properties list matches the table.

Properties							
Required	Property Name	Is Array	Type	ID	Index	Description (optional)	
<input checked="" type="checkbox"/>	name	<input type="checkbox"/>	string ▾	<input type="checkbox"/>	<input type="checkbox"/>	item name	
<input checked="" type="checkbox"/>	description	<input type="checkbox"/>	string ▾	<input type="checkbox"/>	<input type="checkbox"/>	item description	
<input checked="" type="checkbox"/>	img	<input type="checkbox"/>	string ▾	<input type="checkbox"/>	<input type="checkbox"/>	location of item image	
<input checked="" type="checkbox"/>	img_alt	<input type="checkbox"/>	string ▾	<input type="checkbox"/>	<input type="checkbox"/>	item image title	
<input checked="" type="checkbox"/>	price	<input type="checkbox"/>	number ▾	<input type="checkbox"/>	<input type="checkbox"/>	item price	
<input type="checkbox"/>	rating	<input type="checkbox"/>	number ▾	<input type="checkbox"/>	<input type="checkbox"/>	item rating	

__ 4. Save the changes to your **item** model.



4.3. Discover models from a MySQL data source

The model discovery tools automate the process of generating LoopBack models and defining properties that represent tables from a relational database. In this section, you examine one of two features in the API Designer data sources editor:

- The **discover models** feature scans through the tables in a relational database and generates LoopBack models and properties based on the table schema.
- The **update schema** feature works in the reverse direction: it updates the database schema based on the LoopBack model structure.

With the model discovery tools, you can quickly create models and update their data structure to match changes in the underlying database.

- ___ 1. Close the API Designer web application.
 - ___ a. Close API Designer.
 - ___ b. In the **terminal emulator** window, press Ctrl+C to exit the API Designer application.
- ___ 2. Back up and remove the **item** model from the application.
 - ___ a. Change directory to **~/inventory/common/models**.


```
$ cd ~/inventory/common/models
$ pwd
/home/student/inventory/common/models
```
 - ___ b. Change the name of **item.js** and **item.json** to **item.js.backup** and **item.json.backup**.


```
$ mv item.js item.js.backup
$ mv item.json item.json.backup
$ ls -f
item.js.backup item.json.backup
```
 - ___ c. Change directory to **~/inventory/server**.

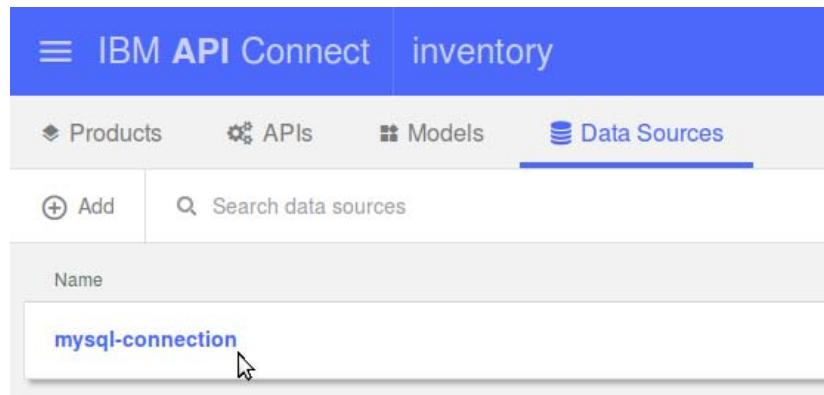

```
$ cd ~/inventory/server
$ pwd
/home/student/inventory/server
```
 - ___ d. Open **model-config.json** in a text editor.


```
$ mousepad model-config.json
```

- __ e. Delete the model mapping from the **item** model to the **mysql-connection** data source.

```
*model-config.json - Mousepad
File Edit View Text Document Navigation Help
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  "item": {
    "dataSource": "mysql-connection",
    "public": true
  }
}
```

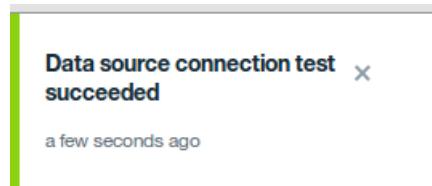
- __ f. Save the **model-config.json** file.
- __ g. Close the text editor.
- __ 3. Open the **mysql-connection** data source.
- __ a. Change directory to the **inventory** application directory.
- ```
$ cd ~/inventory
$ pwd
/home/student/inventory
```
- \_\_ b. Start the API Designer web application.
- ```
$ apic edit
```
- __ c. From the API Designer main page, click **Data Sources**.
- __ d. Open the **mysql-connection** entry.



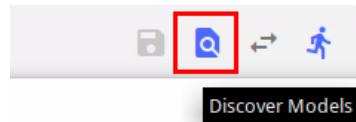
- ___ 4. Test the data source connection.
 - ___ a. Click **Test Data Source Connection** in the upper-left section of the page.



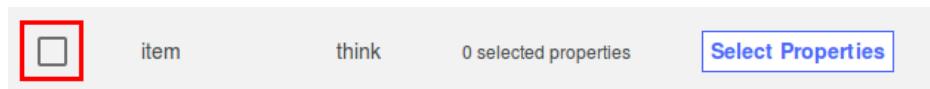
- ___ b. Confirm that the data source is connected to the database successfully.



- ___ 5. Generate a sample model from the think database with the **mysql-connection** data source.
 - ___ a. Click **Discover Models** from the data source toolbar.



- ___ b. The connector retrieves a list of tables from the data source connection. Confirm that the **item** table appears in the list.
- ___ c. To select properties to include in the LoopBack model, select the **item** table check box.



- __ d. Examine the list of properties in the **item** table.

Select Properties: Item

Select the Properties to Review. Required fields are automatically selected and can not be unchecked

<input checked="" type="checkbox"/>	Property Name	Type	Required	ID
<input checked="" type="checkbox"/>	id	Number	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	name	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	description	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	img	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	imgAlt	String	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	price	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>

[Cancel](#)

[Select](#)



Information

The **Select Properties** wizard gives you the option to choose which database column it stores as a LoopBack model property.

- The **Discover Models** wizard must create model properties for every column that the database marked as required. You cannot clear these properties.

- __ e. Click **Select**.

- ___ f. Confirm that the wizard selected seven properties to create from the data source.

Discover Models from: server.mysql-connection

Table Name	Schema	selected properties
<input checked="" type="checkbox"/> item	think	7 selected properties

Select the tables you want to generate models for

Cancel **Generate**

- ___ g. Click **Generate** to create the **item** LoopBack model.
- ___ 6. Examine the generated **item** model.
- ___ a. In the API Designer, switch to the models page.
 - ___ b. Open the **item** model.
 - ___ c. Confirm that the **item** model is a **PersistedModel** object that is mapped to the **mysql-connection** data source.

← All Models

Name	item
Plural	
Base Model	PersistedModel
Data Source	mysql-connection
<input checked="" type="checkbox"/> Public	<input type="checkbox"/> Strict

- ___ d. Scroll down to the list of properties at the end of the page.

- ___ e. Confirm that seven properties appear in the list.

Properties

Required	Property Name	Is Array	Type	ID	Index	Description (optional)	
<input checked="" type="checkbox"/>	description	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		
<input checked="" type="checkbox"/>	id	<input type="checkbox"/>	number	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
<input checked="" type="checkbox"/>	img	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		
<input type="checkbox"/>	imgAlt	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		
<input checked="" type="checkbox"/>	name	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		
<input checked="" type="checkbox"/>	price	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		
<input type="checkbox"/>	rating	<input type="checkbox"/>	string	<input type="checkbox"/>	<input type="checkbox"/>		



Note

When you created the **item** model in the API Designer, you did not define a property that is named "id" in the model.

By default, if you do not define any "id" property and you set the `idInjection` property in the model configuration file to **true**, the LoopBack framework creates an identifier that is named "id". The database generates a unique identifier value when you create a record.

-
- ___ 7. Close the API Designer web application.
- ___ a. Close API Designer.
 - ___ b. In the **terminal emulator** window, press Ctrl+C to exit the API Designer application.
- ___ 8. Restore the **item** model that you defined in the API Designer.
- ___ a. Change directory to **~/inventory/common/models**.
- ```
$ cd ~/inventory/common/models
$ pwd
/home/student/inventory/common/models
```
- \_\_\_ b. Delete the generated **item.js** and **item.json** model files.
- ```
$ rm item.js item.json
```

- __ c. Change the name of **item.js.backup** and **item.json.backup** to **item.js** and **item.json**.

```
$ mv item.js.backup item.js
$ mv item.json.backup item.json
$ ls -f
item.js item.json
```

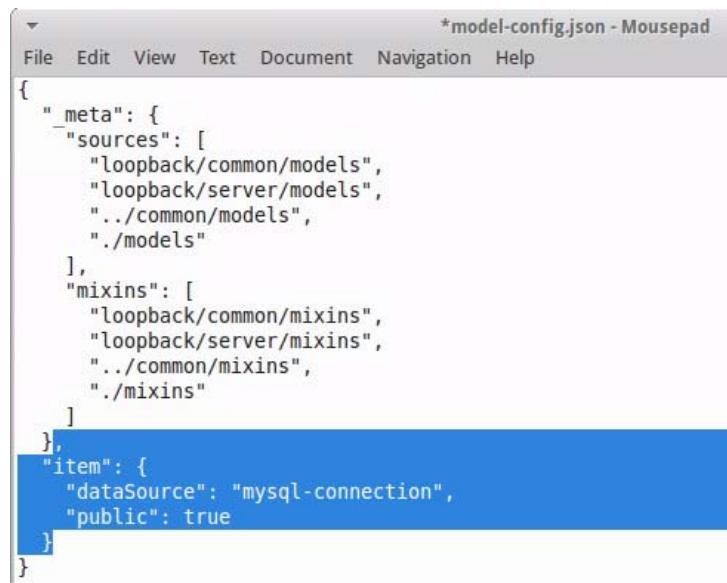
- __ 9. Change the model name to “item” in the **server/model-config.json** configuration file.

- __ a. Change directory to **~/inventory/server**.

```
$ cd ~/inventory/server
$ pwd
/home/student/inventory/server
```

- __ b. Open the **model-config.json** file in a text editor.

- __ c. Change the name of the model to “item”.



```
*model-config.json - Mousepad
File Edit View Text Document Navigation Help
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  "item": {
    "dataSource": "mysql-connection",
    "public": true
  }
}
```

- __ d. Save and close **model-config.json**.

- __ 10. Update the API definition with the changes to the “item” model object.

- __ a. From the terminal emulator window, run: **apic loopback:refresh**

```
$ cd ~/inventory
$ pwd
/home/student/inventory
$ apic loopback:refresh
```

4.4. Test the inventory items model

In this section, start a local copy of the inventory LoopBack application with the API Designer web application. Examine the API operations for the **item** model in the Explore view. Test the API operations and retrieve model data through the **mysql-connection** data source.

- 1. Start API Designer.

- a. From the **terminal emulator** window, navigate to the **inventory** application directory.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

- b. Start the API Designer web application.

```
$ apic edit
```

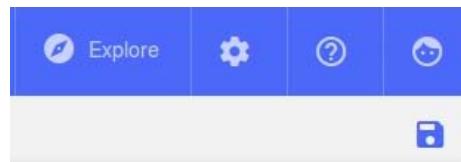
- 2. Start the **inventory** LoopBack application.

- a. At the bottom of the API Designer page, click **Start the servers**.
 - b. Wait until the application and Micro Gateway are started.



- 3. Open the Explore page in the API Designer web application.

- a. From the toolbar at the start of the page, click **Explore**.



- 4. Retrieve a list of inventory items with the GET /items operation.

- a. Examine the list of API operations for the **item** model.

A screenshot of the API Designer Explore page. The top navigation bar includes a search bar, a dropdown for "Operations" set to "by name", and a REST tab. The main content area shows the "inventory 1.0.0" service. The "item" model is listed with its operations:

- item.create**: Create a new instance of the model and persist it into the data source.
- item.patchOrCreate**: Patch an existing model instance or insert a new one into the data source.
- item.replaceOrCreate__put_items**
- item.find**



Information

The **Explore** page lists the API definitions from the current API product. In this example, the application hosts the **inventory** API, at version **1.0.0**.

A list of data-centric LoopBack model operations appear after the API name. Each API operation consists of the model name and a method, such as create or find. The corresponding REST operations include an HTTP method (verb) and a resource path. For more information about purpose of each API operation, see: <http://loopback.io/doc/en/lb3/LoopBack-core-concepts.html>

- b. Select the **item.find** API operation.
- c. Examine the three-column view for the API operation.

The screenshot shows the LoopBack Explore page with the following details:

- Left Column:** A list of API operations: item.create, item.patchOrCreate, item.replaceOrCreate__put_items, item.find, item.replaceOrCreate__post_items_replace, item.upsertWithWhere, item.exists__get_items_{id}_exists, item.exists__head_items_{id}, item.findById.
- Middle Column - item.find:**
 - Description:** Find all instances of the model matched by filter from the data source.
 - Security:**

X-IBM-Client-Id	apiKey located in header	clientIdHeader
X-IBM-Client-Secret	apiKey located in header	clientSecretHeader
 - Parameters:** filter string, optional in query
- Right Column:**
 - HTTP Method:** GET https://localhost:4002/api/items
 - Example Request:** curl
 - Code Example:**

```
curl --request GET \
--url 'https://localhost:4002/api/items?filter=REPL...' \
--header 'accept: application/json' \
--header 'x-ibm-client-id: default' \
--header 'x-ibm-client-secret: SECRET'
```

At the bottom of the interface, there are status indicators: Running, Gateway: https://localhost:4002, Application: http://127.0.0.1:4001, and standard window control buttons.



Information

When you select an API operation from the left column, the middle column displays the API operation details. In this example, the **item.find** operation takes an optional parameter that is named **filter** as a **query** parameter.

On the right column, the Explore page displays sample code that you can implement to call the API operation. The example displays the command parameters for **cURL**, a popular command-line utility for making HTTP requests.

- ___ d. Scroll down in the right column to see the example `curl` request and example response.

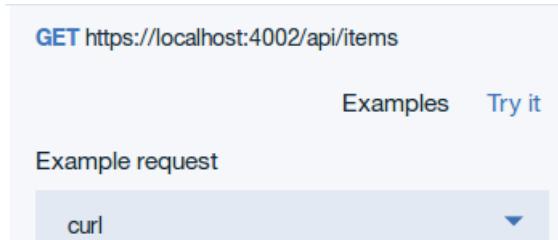
The screenshot shows the LoopBack API Explorer interface. In the top section, there is a code editor containing a `curl` command:

```
curl --request GET \
--url 'https://localhost:4002/api/items?filter=REPL...' \
--header 'accept: application/json' \
--header 'x-ibm-client-id: default' \
--header 'x-ibm-client-secret: SECRET'
```

Below the code editor, the text "Example response" is followed by a JSON object:

```
[  
  {  
    "name": "Bessie Armstrong",  
    "description": "Ba rahik bohevzh afouhsuw vari ba...",  
    "img": "udzegu",  
    "img_alt": "igfu",  
    "price": 47.97,  
    "rating": 69.0231737,
```

- ___ e. Scroll back to the **GET** operation.



- ___ f. Click **Try it**.



Note

If you see an API definition of `$(catalog.host)`, your LoopBack application did not substitute the correct host name for the API operation. Make sure that the **inventory** application is started before you open the **Explore** page.

- ___ g. Click **Call operation**.

- ___ 5. Examine the result from the **GET** API operation call.

- ___ a. Review the request and response headers from the API call.

Request

```
GET https://localhost:4002/api/items
Headers:
Content-Type: application/json
Accept: application/json
X-IBM-Client-Id: default
X-IBM-Client-Secret: SECRET
```

Response

```
Code: 200 OK
Headers:
content-type: application/json; charset=utf-8
x-ratelimit-limit: 100
x-ratelimit-remaining: 99
```

- b. Scroll down further and review the list of items in the message body.

```
[
  {
    "description": "Punch-card tabulating machines and...",
    "img": "images/items/meat-chopper.jpg",
    "img_alt": "Dayton Meat Chopper",
    "name": "Dayton Meat Chopper",
    "price": 4599.99,
    "rating": 0,
    "id": 1
  },
  ...
]
```



- ___ 6. Check the number of items in the **inventory** database with the **GET /items/count** API operation.
 - ___ a. In the **Explore** page, select the **item.count** operation from the left column.
 - ___ b. In the right column, scroll down to the API operation test client for the **http://localhost:<port>/api/items/count** operation.

The screenshot shows the API Explorer interface for a LoopBack application. At the top, it displays the URL **GET https://localhost:4002/api/items/count**. Below this are two buttons: **Examples** and **Try it**. A dropdown menu labeled **curl** is selected under the **Example request** section. The curl command shown is:

```
curl --request GET \
--url 'https://localhost:4002/api/items/count?where' \
--header 'accept: application/json' \
--header 'x-ibm-client-id: default' \
--header 'x-ibm-client-secret: SECRET'
```

- ___ c. Click **Try it**.
- ___ d. Click **Call operation**.
- ___ e. Confirm that the result returns a **count** of **12** items.

The screenshot shows the API Explorer interface displaying the results of the API call. It has two main sections: **Request** and **Response**.

Request:

```
GET https://localhost:4002/api/items/count
Headers:
Content-Type: application/json
Accept: application/json
X-IBM-Client-Id: default
X-IBM-Client-Secret: SECRET
```

Response:

```
Code: 200 OK
Headers:
content-type: application/json; charset=utf-8
x-ratelimit-limit: 100
x-ratelimit-remaining: 99

{
  "count": 12
}
```

- ___ 7. Close the API Designer and the API application.
 - ___ a. Close the web browser.

- __ b. In the **terminal emulator** window, press Ctrl+C to exit the API Designer application.

```
Express server listening on http://127:0.0.1:9000
^C
$
```



Information

In this section, you created the **item** model object to represent a row from the **item** table in the **think** MySQL database. You connected the **item** model to the relational database through the **mysql-connection** data source.

You tested the connection between the **item** model and the **mysql-connection** data source through the LoopBack API.

4.5. Create a MongoDB data source and the review model

In this section, you define a second model object, named **ratings**, which represents user reviews on your **inventory** API. Since people write reviews of varying lengths, it makes sense to use a document-based database instead of a relational database to save the reviews.

The **MongoDB** database is an example of a document-centric, non-relational database. Instead of a rigid database table with defined columns, MongoDB stores each record as an arbitrary file of data.

At the end of this section, you connect the **ratings** LoopBack model object to a MongoDB data source. The data source sends and retrieves user reviews from a MongoDB database.

- ___ 1. Open a terminal emulator window.
- ___ 2. Define a **MongoDB** data source in the **inventory** application.

- ___ a. Stop the local copy of the inventory application.

```
$ apic stop
Stopped inventory
Stopped inventory-gw
```

- ___ b. Verify that the current directory is the **inventory** application directory.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

- ___ c. Run the **apic** data source command.

```
$ apic create --type datasource
```

- ___ d. Create a data source that is named **mongodb-connection** with the **MongoDB** LoopBack connector.

- Data source name: **mongodb-connection**
 - Connector: **MongoDB (supported by StrongLoop)**
 - Connector-specific configuration:
 - Host: **mongo.think.ibm**
 - Port: **27017**
 - User: **<leave blank>**
 - Password: **<leave blank>**
 - Database: **think**
 - Install loopback-connector-mongodb: **Y**

```

? Enter the data-source name: mongodb-connection
? Select the connector for mongodb-connection:
  > MongoDB (supported by StrongLoop)
Connector-specific configuration:
? Connection String url to override other settings (eg:
mongodb://username:password@hostname:port/database):
? host: mongo.think.ibm
? port: 27017
? user:
? password:
? database: think
? Install loopback-connector-mongodb@^1.4 (Y/n) Y

```



Information

When the `apic create` command defines a data source, it checks whether you have the correct LoopBack connector for the data source. If you did not install the connector, you can select **yes** to the `install loopback-connector-mongodb` prompt. This command is the same as running the `npm install` command:

```
npm install --save loopback-connector-mongodb
```

For more information about the MongoDB LoopBack connector configuration and settings, see: <http://loopback.io/doc/en/lb2/MongoDB-connector.html>

- ___ 3. Create a LoopBack model object that is named **review** to represent reviews that are stored in the MongoDB database.
 - ___ a. Run the `apic` command to create a LoopBack model.


```
$ apic create --type model
```
 - ___ b. Define a model that is named **review**, with the following properties:
 - Model name: **review**
 - Attach model to data source: `mongodb-connection` (`mongodb`)
 - Base model class: `PersistedModel`
 - Expose review via the REST API? **N**
 - Common model or server only? **common**

```

? Enter the model name: review
? Select the data-source to attach review to:
  > mongodb-connection (mongodb)
? Select models base class:
  > PersistedModel
? Expose review via the REST API? (Y/n): N
? Common model or server only?
> common

```



Note

Remember to answer **No** in the **expose review via the REST API** question. You do not expose the **review** model in the **inventory** API. You write custom code in a later exercise that modifies **review** records in a remote method.

- c. Add a property that is named **date**, of type **date**, to the **review** model.

Enter an empty property name when done.

```

? Property name: date
? Property type:
  > date
? Required? Y
? Default value [leave blank for none]: <leave blank>

```

- d. Enter the remaining properties according to the table.

Table 6. Review model properties

Property name	Property type	Required	Default value
date	date	Yes	<leave blank>
reviewer_name	string	No	<leave blank>
reviewer_email	string	No	<leave blank>
comment	string	No	<leave blank>
rating	number	Yes	<leave blank>

- e. Leave the **property name** blank after you create the last property.

Let's add another review property.

Enter an empty property name when done.

? Property name:

Done running LoopBack generator

Updating swagger and product definitions

Created /home/student/inventory/definitions/inventory.yaml swagger description

4.6. Define a relationship between the item and review models

In the LoopBack framework, you can define relationships between two sets of models in your application. When you create a relationship, you impose a set of behaviors on a pair of model objects. For example, an online product review does not exist on its own. Every review applies to exactly one item in the inventory database.

To model the relationship in the **inventory** LoopBack application, you create a relationship that states **an item has many reviews**. Use the “apic” command-line utility to define the relationship, and save the information in the model configuration files.

- ___ 1. Define a one-to-many relationship between the **item** and **review** models.

- ___ a. Start the **apic** LoopBack relationship command.

```
$ apic loopback:relation
```

- ___ b. Create a model relationship named **reviews**: a one-to-many relationship from the **item** model to the **review** model.
 - Create a relationship from: **item**
 - Relation type: **has many**
 - Create a relationship with: **review**
 - Property name for the relation: **reviews**
 - Custom foreign key: <leave blank>
 - Require a through model: **No**

? Select the model to create the relationship from:

```
> item
```

? Relation type:

```
> has many
```

? Choose a model to create a relationship with:

```
> review
```

? Enter the property name for the relation: **reviews**

? Optionally enter a custom foreign key: <leave blank>

? Require a through model? **No**

- ___ c. Open the **item.json** model definition file.

```
$ more common/models/item.json
```

- ___ d. Review the relationship definition section in **item.json**.

```
"validations": [],
"relations": {
  "reviews": {
    "type": "hasMany",
    "model": "review",
    "foreignKey": ""
  }
},
"acl": [],
"methods": {}
```



Note

The relationships that you create in your LoopBack application are **logical relationships**. The underlying MySQL and MongoDB databases *do not* have a relationship between its records.

From the point of view of the **item** table in the MySQL database, **reviews** is a property that stores an identifier value. This identifier is a number that corresponds to a document in the **review** MongoDB database.

The API operations in your **inventory** LoopBack application enforce the **reviews** relationship. It controls how users can create, update, and delete **item** and **review** model objects.

- ___ 2. Review the **review** model in the API Designer.

- ___ a. Open the API Designer web application.

\$ apic edit

- ___ b. Open the **inventory** API definition.

The screenshot shows the IBM API Connect interface with the title bar "IBM API Connect" and "inventory". Below the title bar, there are four tabs: "Products", "APIs" (which is selected), "Models", and "Data Sources". Under the "APIs" tab, there is a button "Add" and a search bar "Search APIs". A table below has a single row with the title "inventory 1.0.0 [inventory.yaml]". This row is highlighted with a red rectangle.

- ___ c. Examine the **Paths** section of the API definition.

- ___ d. Select the `/items/{id}/reviews/{fk}` path.

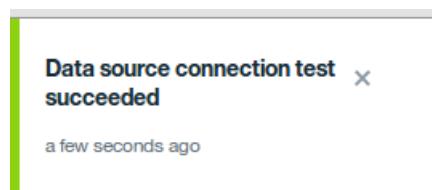
The screenshot shows the API Designer interface. On the left, there is a sidebar titled "Paths" with a list of URLs. One URL, `/items/{id}/reviews/{fk}`, is highlighted with a red box. To the right, the main panel displays the selected path: `/items/{id}/reviews/{fk}`. Below this, under "Parameters", it says "No parameters defined". Under "Operations", there are three entries: "GET /items/{id}/reviews/{fk}" (blue button), "DELETE /items/{id}/reviews/{fk}" (red button), and "PUT /items/{id}/reviews/{fk}" (orange button).



Information

When you created the **reviews** relationship, the LoopBack framework created three API operations that your clients use to create, retrieve, and modify **reviews** that belong to the **item** model.

- ___ e. Click **All APIs**.
- ___ 3. Test the mongodb-connection.
 - ___ a. Click **Data Sources**.
 - ___ b. Select mongodb-connection.
 - ___ c. Click Test Data sources.



The mongodb-connection tests successfully.

- ___ 4. Close and exit the API Designer application.

End of exercise

Exercise review and wrap-up

The first part of the exercise examined how to generate a LoopBack application with the API Connect Toolkit. The `apic loopback` command creates a starting point for an empty LoopBack API application in the workstation. You defined two business model in LoopBack: an “item” record and a “review” document.

In the second part of the exercise, you configured LoopBack data types for two databases: a MySQL relational database and a document-based MongoDB database. You defined a relationship that linked the “item” MySQL table with “review” documents in the MongoDB database.

Exercise 5. Implementing event-driven functions with remote and operation hooks

Estimated time

00:45

Overview

This unit explores how LoopBack data sources retrieve and save model data from data stores. You learn how to install LoopBack connectors in your application, and how to map model objects to data sources.

Objectives

After completing this exercise, you should be able to:

- Define a remote hook in a LoopBack application
- Define an operation hook in a LoopBack application
- Test remote and operation hooks in the assembly editor test client

Introduction

In the previous exercise, you created the **inventory** LoopBack application. You generated a set of API operations based on two model objects: **item** and **review**. The **item** model represents details of items in the **think** MySQL database. The **review** model stores user-submitted reviews of items for sale. A non-relational, document-centric MongoDB database stores the reviews.

In this exercise, you extend the **inventory** model with remote hooks: preprocessing and post-processing functions that change the behavior of API operation calls.

You also create two **operation** hooks: event listeners that the LoopBack framework triggers when it accesses or modifies persisted model objects.

Requirements

Before you start this exercise, you must complete the **data sources** exercise in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

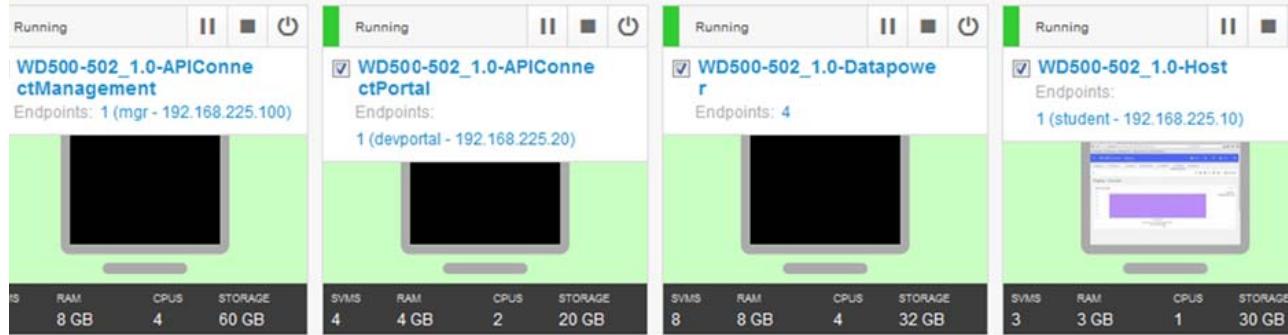
Exercise instructions

Before you begin

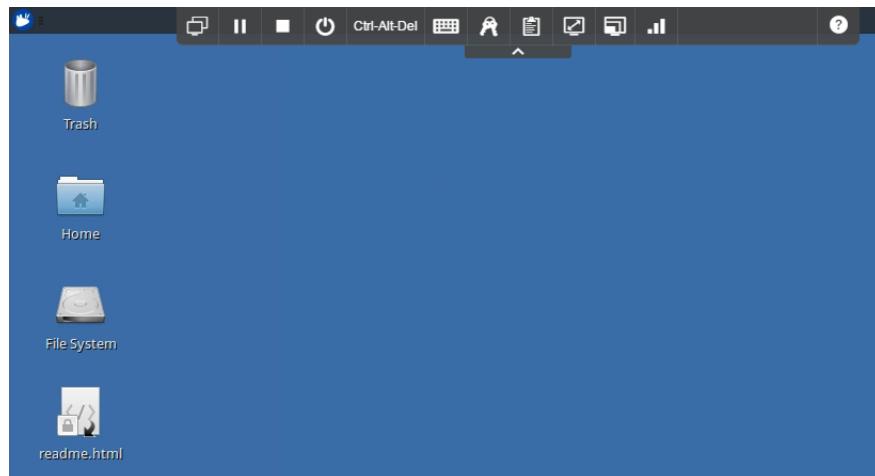
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



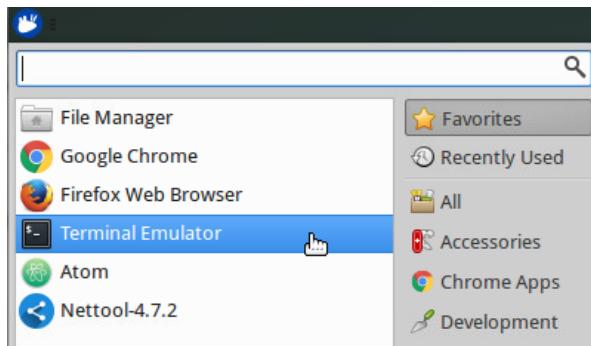
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

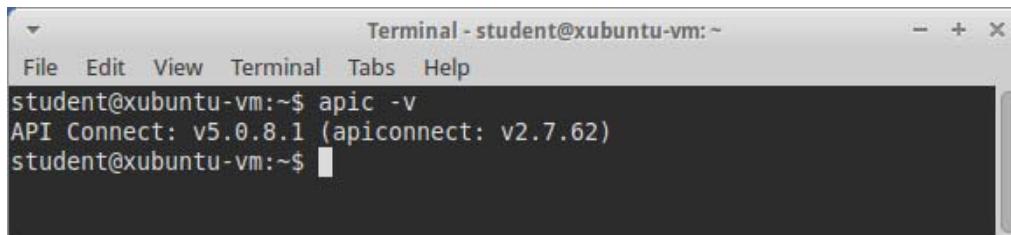
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

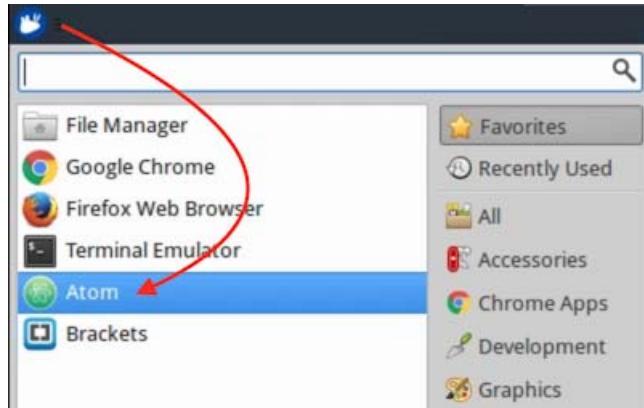
The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

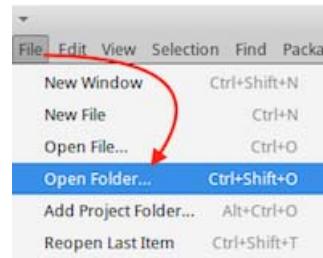
5.1. Modify the API base path

By default, LoopBack applications host API operations with the `/api` base path. Change the base path to: `/inventory`

- ___ 1. Open the **inventory** LoopBack application in the **Atom** text editor.
 - ___ a. Open the application favorites menu.
 - ___ b. Select the **Atom** text editor.

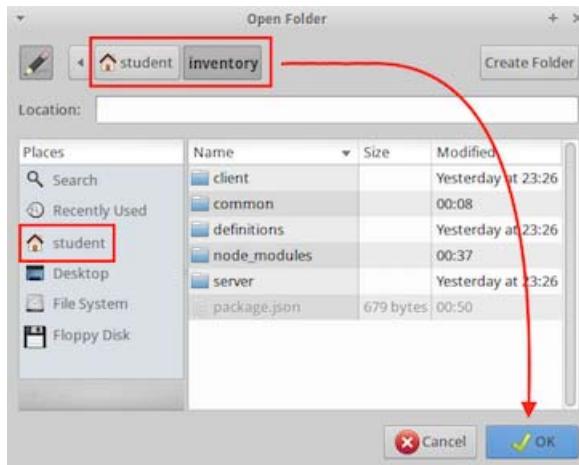


- ___ c. From the **Atom** menu bar, click **File > Open Folder**.



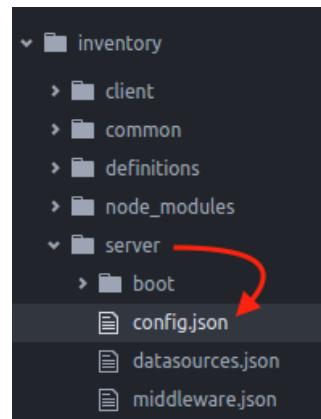
- ___ d. Select the **student** folder in the Places menu.

- __ e. Navigate to the **inventory** folder.



- __ f. Click **OK**.

- __ 2. Edit the API root in the **server/config.json** configuration path to: **"/inventory"**
- From the folder tree menu, expand the **server** folder.
 - Open the **config.json** file to view the source.

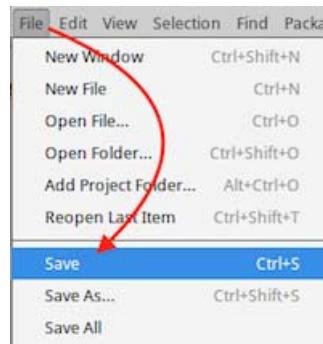


- __ c. In the **"restApiRoot"** property, change the value from **"/api"** to: **"/inventory"**

```

1
2 "restApiRoot": "/inventory",
3   "host": "0.0.0.0",
4   "port": 3000,
5   "remoting": {
6     "context": false,
7     "rest": {
8       "normalizeHttpPath": false,
9       "xml": false
10     },
11     "json": {
12       "strict": false,
13       "limit": "100kb"
14     }
15   }
16 }
```

- ___ 3. Save your changes.



- ___ 4. Update the API definition with the modified base path value.

- ___ a. In a **terminal emulator** window, confirm that the current directory is the **inventory** application directory.

```
$ pwd
/home/student/inventory/
```

- ___ b. Run the apic utility LoopBack **refresh** command.

```
$ apic loopback:refresh
Updating swagger and product definitions
Created /home/student/inventory/definitions/inventory.yaml swagger
description
```



Information

What are the “swagger and product definitions”?

The “swagger definition” refers to the **OpenAPI definition** for the inventory API application. The Swagger specification is the predecessor for the OpenAPI 2.0 specification. The term “swagger” is synonymous with “OpenAPI definition”.

An API product is a collection of OpenAPI definitions. You can create and edit product definitions in the API Designer web application.

When you run the `apic loopback:refresh` command, the apic utility scans through the LoopBack models and server configuration files. It updates the API definition and API product to make the interface files consistent with the implementation.

- ___ 5. Confirm the modified base path in the inventory API definition file.

- ___ a. Open the API Designer web application in the inventory application directory.

```
$ apic edit
```

- __ b. Open the **inventory** API definition.

The screenshot shows the IBM API Connect interface. The top navigation bar has tabs for 'Products', 'APIs' (which is selected), 'Models', and 'Data Sources'. Below the navigation is a search bar with the placeholder 'Search APIs'. A large button labeled '+ Add' is visible. The main content area is titled 'Title' and contains the text 'inventory 1.0.0 [inventory.yaml]'. This text is highlighted with a red rectangular box.

- __ c. Confirm that the **base path** entry is updated to `/inventory`.

The screenshot shows the API Designer interface. The top navigation bar has tabs for 'All APIs', 'Design' (which is selected), 'Source', and 'Assemble'. On the left, there are sections for 'Schemes', 'Host', and 'Base Path'. The 'Base Path' section is highlighted with a red rectangular box. It shows the current value as 'Base Path /inventory'. Below the 'Base Path' section is another section labeled 'Consumes'.

- __ d. Close the web browser.
__ e. Press **Ctrl+C** in the **terminal emulator** window to exit the API Designer web application.

5.2. Create an operation hook

An **operation hook** is a function that listens to an entire class of operations: create, read, update, or delete operation. Whereas remote hooks run before or after a **specific** named operation, operation hooks listen to a **class of operations** to a model that is persisted in a data source.

- The **access** hook is triggered whenever the LoopBack framework queries a data source for models, for example, when you call any API operation that runs the **create**, **retrieve**, **update**, or **delete** method in the `PersistedModel` object.
- The **before save** hook is triggered before any operation that **creates** or **updates** a `PersistedModel` object.
- The **after save** hook is triggered after any operation that **creates** or **updates** a `PersistedModel` object.
- The **before delete** and **after delete** hooks are triggered before LoopBack removes a model from a data source. Specifically, these hooks run when you call the `deleteAll`, `deleteById`, and `prototype.delete()` functions in the `PersistedModel` object.
- The **loaded** hook is triggered after a LoopBack connector retrieves model data, but before it creates a model object from the data. You can use a loaded hook to transform or decrypt raw data from a connector before LoopBack builds a model instance.
- The **persist** hook listens to operations that save data to the data source. While the **before save** hook listens to changes to a LoopBack model object, the **persist** hook runs before the LoopBack framework saves, or persists, a modified model object to the data source.

In this section, you define an **access** hook to listen to items that are retrieved from a data source. You also monitor updates to the item model object with a **before save** hook. For example, you can log whether the LoopBack framework created or updated a persisted model object.

- ___ 1. Open the `common/models/item.js` model script file.
 - ___ a. In the **Atom** text editor, expand the `common/models` directory.
 - ___ b. Open the `item.js` script from the `inventory/common/models` project folder.



```
item.js
1 'use strict';
2
3 module.exports = function(Item) {
4
5 };
6
```

- ___ 2. Create an **access** hook in the `item` model.
 - ___ a. Open the `item.js` model script file in the editor.

- __ b. Define a listener function that observes the **access** event.

```
Item.observe('access', function logQuery(ctx, next){  
  console.log('Accessed %s matching %j',  
    ctx.Model.modelName, ctx.query.where);  
  next();  
});
```



The screenshot shows a code editor window with a dark theme. The file is named 'item.js'. The code is identical to the one provided above, defining an observe event for 'access' on the Item model. The code editor has syntax highlighting for JavaScript, with keywords like 'use strict', 'module.exports', and 'function' in blue, and variable names in green.



Information

The LoopBack framework calls the `logQuery` function when an API operation queries information from a data source. The function writes the name of the model and the lookup parameters in the console log.

___ 3. Create an **after save** hook in the **item** model.

___ a. In the **item.js** script file, add a second listener function that observes the **after save** event.

```
Item.observe('after save', function(ctx, next) {
  if (ctx.instance) {
    console.log(
      'Saved %s#%s',
      ctx.Model.modelName, ctx.instance.id);
  } else {
    console.log(
      'Updated %s matching %j',
      ctx.Model.plural modelName, ctx.where);
  }
  next();
});
```

```
item.js
1 'use strict';
2
3 module.exports = function(Item) {
4   Item.observe('access', function logQuery(ctx, next){
5     console.log('Accessed %s matching %j',
6       ctx.Model.modelName, ctx.query.where);
7     next();
8   });
9
10  Item.observe('after save', function(ctx, next) {
11    if (ctx.instance) {
12      console.log(
13        'Saved %s#%s',
14        ctx.Model.modelName, ctx.instance.id);
15    } else {
16      console.log(
17        'Updated %s matching %j',
18        ctx.Model.plural modelName, ctx.where);
19    }
20    next();
21  });
22};
```



Information

The LoopBack framework runs the **after save** operation hook when an API operation creates or updates a record in the data source. The *ctx.instance* object represents the newly created model object. If the data source did not create a model instance, the log records the existing object that the framework updated.

___ 4. Save the **item.js** model script file.

5.3. Test the operation hook

To test the **access** operation hooks, run the **inventory** API application with the API Designer web application. Test API operations and review the entries in the application log.

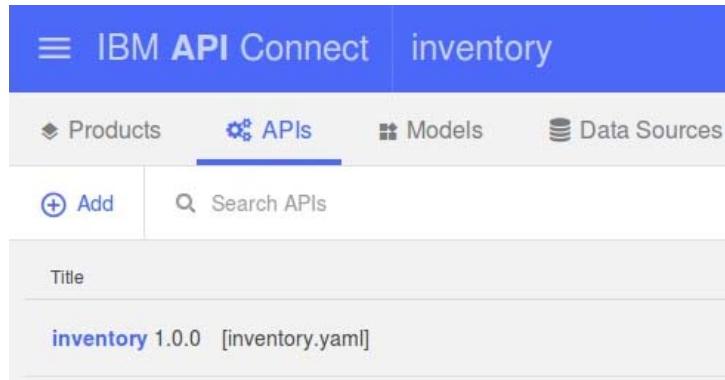
- ___ 1. Start the API Designer web application.
 - ___ a. In a **terminal emulator** window, confirm that the current directory is in the inventory application.


```
$ pwd
/home/student/inventory/
```
 - ___ b. Start the API Designer application.


```
$ apic edit
```
- ___ 2. Start the **inventory** API application.
 - ___ a. From the API Designer web application, click the **start** icon at the end of the page.
 - ___ b. Make sure that the API application and micro gateway are started.

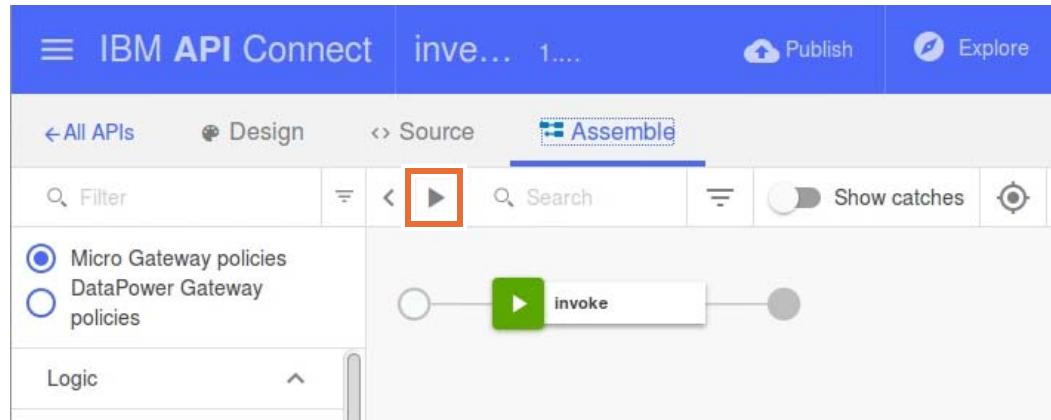


- ___ 3. Run the **item.count** API operation from the Assemble view test client.
 - ___ a. In the API Designer main page, click the **API** tab.
 - ___ b. Open the **inventory** API definition.

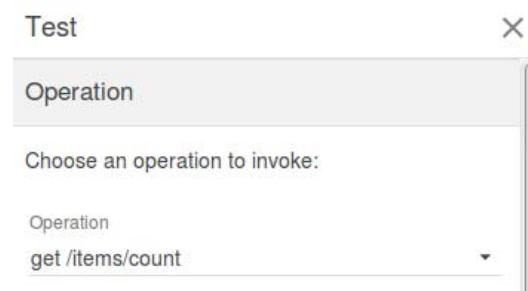


- ___ c. Switch to the **Assemble** tab.

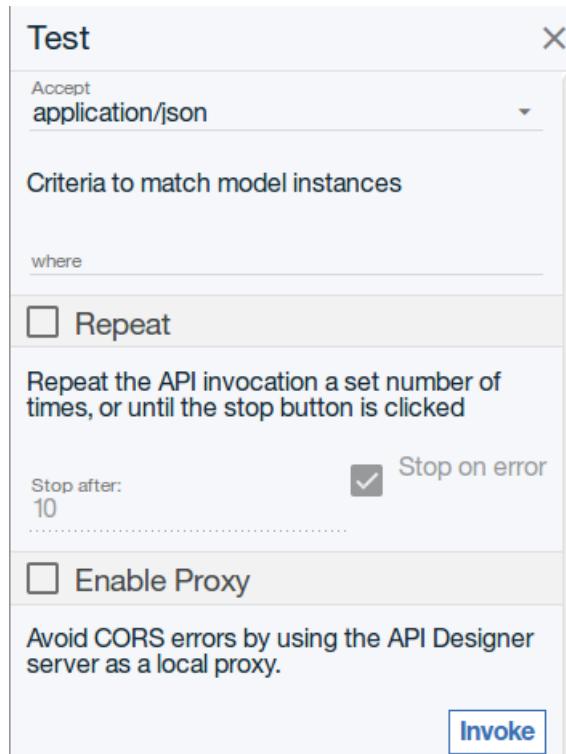
- __ d. In the assembly view, open the test client feature. The **test client** icon is a triangle (>) immediately to the left of the search field.



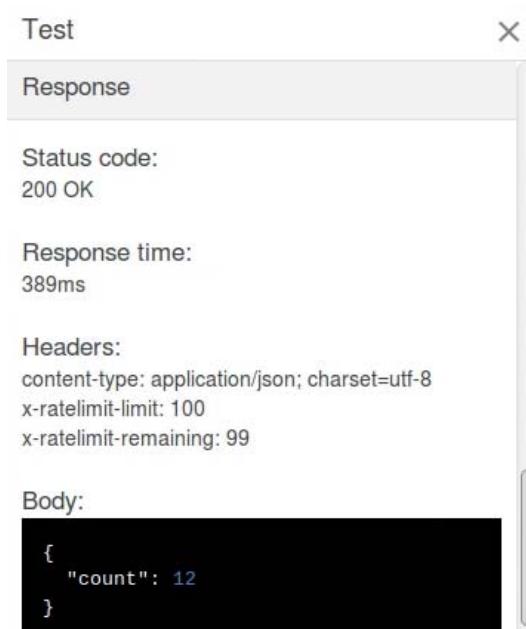
- __ e. Select the **get /items/count** operation to test.



- ___ f. At the end of the test client page, click **invoke**.



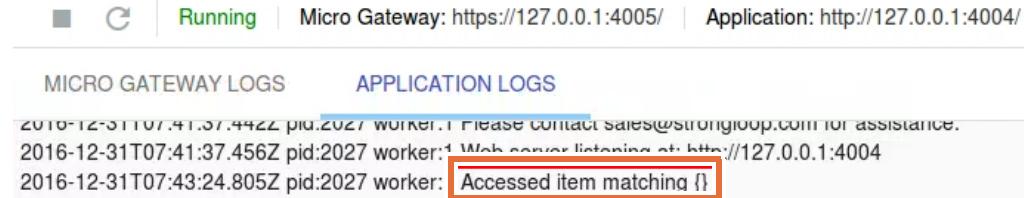
- ___ g. Confirm that the operation returned a result of **12** item records in the count.



- ___ 4. Review the output from the operation hook in the application log.
 - ___ a. In the application and micro gateway bar, click the **settings** (gear) icon.

Gateway: <https://localhost:4002/> | Application: <http://127.0.0.1:4001/> | 

- ___ b. Click the **application logs** tab to view the console output from the **inventory** application.
- ___ c. Click the **console** (monitor +) icon to maximize the log view.



- ___ d. Confirm that the **access** operation hook logged the query against the **item** database.



Note

If you do not see anything in the application logs in the API Designer, you can also see the log output by opening a terminal window. Then, from the inventory directory, type: `apic logs`

```
Terminal - student@xubuntu-vm:~/inventory
File Edit View Terminal Tabs Help
student@xubuntu-vm:~/inventory$ apic logs
Tailing logs for `inventory'
[Wed Oct 11 13:25:21 2017] com.ibm.diagnostics.healthcenter.loader INFO: Node Application Metrics 3.0.2.201706071954 (Agent Core 3.2.1)
[Wed Oct 11 13:25:21 2017] com.ibm.diagnostics.healthcenter.mqtt INFO: Connecting to broker localhost:1883
strong-supervisor attaching dashboard at /appmetrics-dash
2017-10-11T17:25:21.68Z pid:22555 worker:0 INFO supervisor starting (pid 22555)
2017-10-11T17:25:21.68Z pid:22555 worker:0 INFO supervisor reporting metrics to `internal`:
2017-10-11T17:25:21.71Z pid:22555 worker:0 INFO supervisor size set to undefined
2017-10-11T17:25:22.02Z pid:22569 worker:1 [Wed Oct 11 13:25:22 2017] com.ibm.diagnostics.healthcenter.loader INFO: Node Application Metrics 3.0.2.201706071954 (Agent Core 3.2.1)
2017-10-11T17:25:22.08Z pid:22569 worker:1 [Wed Oct 11 13:25:22 2017] com.ibm.diagnostics.healthcenter.mqtt INFO: Connecting to broker localhost:1883
2017-10-11T17:25:22.09Z pid:22569 worker:1 strong-supervisor attaching dashboard at /appmetrics-dash
2017-10-11T17:25:23.71Z pid:22569 worker:1 Web server listening at: http://localhost:4001
2017-10-11T17:26:12.19Z pid:22569 worker:1 Accessed item matching {}
```

- ___ 5. Test the **item.find** API operation with a **where** clause of "id": 1.
- ___ a. In the test client, change the operation to: get /items



- ___ b. In the **filter** parameter, enter `{"where": {"id":1}}` as the query parameter.

Filter defining fields, where, include, order, offset, and limit
filter
`{"where": {"id":1}}`

- ___ c. Click **Invoke**.
- ___ d. Confirm that the **access** operation hook logged the data source retrieval and query parameter.

Response

Status code:
200 OK

Response time:
282ms

■ Running | Micro Gateway: https://127.0.0.1:4005/ | Application: http://127.0.0.1:4004/

MICRO GATEWAY LOGS	APPLICATION LOGS
2016-12-31T07:41:57.456Z pid:2027 worker:1 Web server listening at: http://127.0.0.1:4004	2016-12-31T07:43:24.805Z pid:2027 worker:1 Accessed item matching {"id":1}
	2016-12-31T08:02:34.648Z pid:2027 worker:1 Accessed item matching {"id":1}

5.4. Create a remote hook

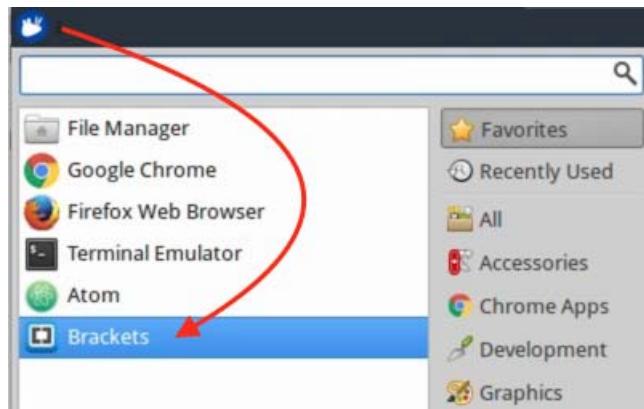
A **remote hook** is a custom JavaScript function that runs before or after the LoopBack application completes an API operation call.

In this section, you write a function that the LoopBack framework runs after an API operation. When a user submits a review for an item, the remote hook takes an average of all review scores, and updates the value in the database.

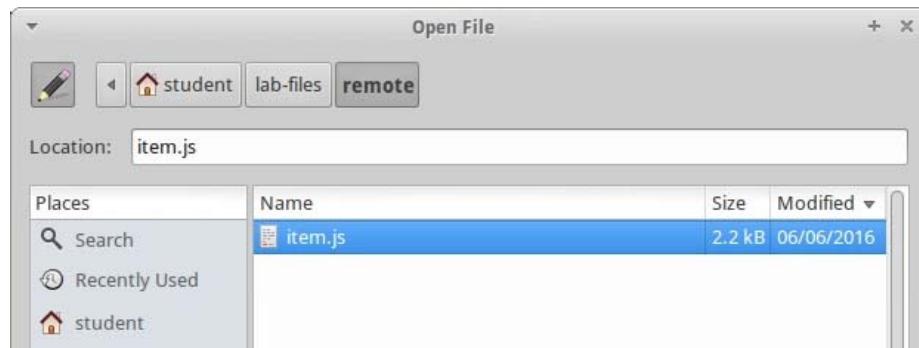
For more information about remote hooks, see the LoopBack documentation:

<http://loopback.io/doc/en/lb3/Remote-hooks.html>

- ___ 1. Open the `common/models/item.js` model script file.
- ___ 2. Copy the remote hook implementation from the solution file.
- ___ a. Open the **Brackets** text editor from the application menu.

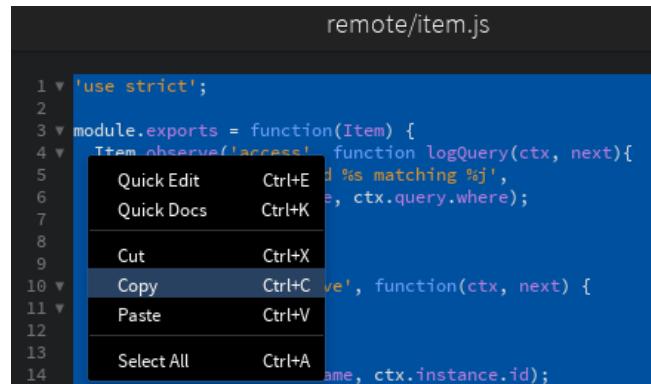


- ___ b. In the Brackets editor, expand the `lab-files/remote/` folder.
- ___ c. Select the `item.js` script.



- ___ d. In the menu bar, click **Edit > Select All** to highlight the contents of the `item.js` script.

- __ e. Press Ctrl+C to copy the contents of the `item.js` script into the system clipboard.

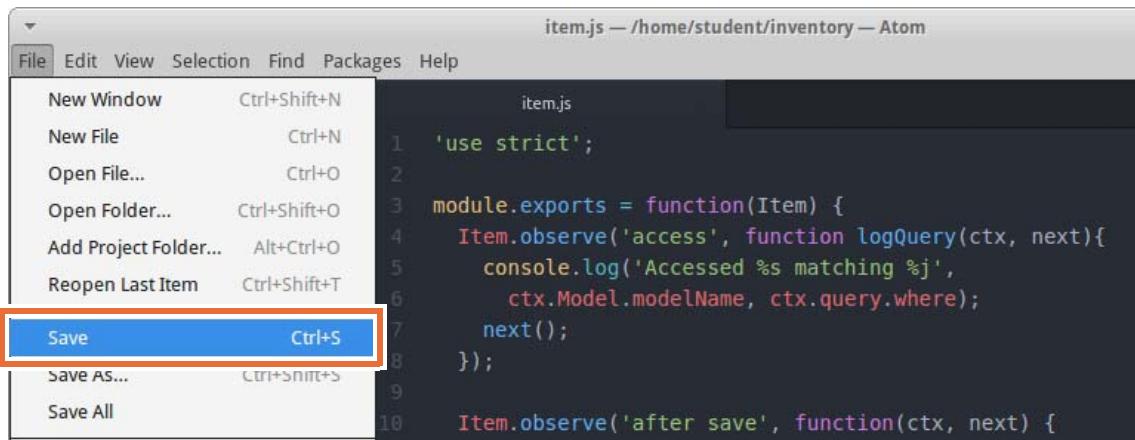


```
remote/item.js

1 'use strict';
2
3 module.exports = function(Item) {
4   Item.observe('access', function logQuery(ctx, next){
5     Quick Edit    Ctrl+E d %s matching %j',
6     Quick Docs   Ctrl+K e, ctx.query.where);
7
8     Cut          Ctrl+X
9     Copy         Ctrl+C ve', function(ctx, next) {
10    Paste        Ctrl+V
11
12    Select All   Ctrl+A ame, ctx.instance.id);
13
14 }
```

- __ f. In the **Atom** text editor, remove the contents of the `item.js` script file.
 __ g. Click **Edit > Paste** from the menu bar in the Atom editor.
 3. Save the changes to the `item.js` model script file.

- __ a. In the **Atom** text editor, click **File > Save**.



__ 4. Examine the remote hook implementation in the Atom editor.

__ a. Review the contents of the `item.js` script file.

```

          item.js

20      next();
21  });
22
23  Item.afterRemote('prototype.__create_reviews', function (ctx, remoteMethodOutput) {
24    var itemId = remoteMethodOutput.itemId;
25
26    console.log("calculating new rating for item: " + itemId);
27
28    var searchQuery = {include: {relation: 'reviews'}};
29
30    Item.findById(itemId, searchQuery, function findItemReviewRatings(err, findResult) {
31      var reviewArray = findResult.reviews();
32      var reviewCount = reviewArray.length;
33      var ratingSum = 0;
34
35      for (var i = 0; i < reviewCount; i++) {
36        ratingSum += reviewArray[i].rating;
37      }
38
39      var updatedRating = Math.round((ratingSum / reviewCount) * 100) / 100;
40

```



Information

A **remote hook** declaration has two components: the operation on the model, and script code that LoopBack runs when the user calls the model operation.

In this example, the `Item.afterRemote` listens to the `prototype.__create_reviews` event. After the LoopBack framework creates a *review* model instance, the framework triggers the anonymous function.

```

Item.afterRemote(
  'prototype.__create_reviews',
  function (ctx, remoteMethodOutput, next) {
...
}

```

The anonymous function holds the logic for the remote hook. The function accepts three input variables:

- The context variable `ctx` captures the environment settings of the LoopBack environment. It holds the original request message for the create review API operation.
- The `remoteMethodOutput` variable captures the response from the intercepted API call. In this example, it is the create review API operation response.

- When your remote method finishes its work, it starts the *next* callback handler to pass control back to the LoopBack framework. You must call the *next* function; otherwise, the LoopBack API application does not know whether the event handler successfully completed its work.

b. Examine the code within the anonymous function:

```
module.exports = function (Item) {
  ...
  Item.afterRemote(
    'prototype.__create_reviews',
    function (ctx, remoteMethodOutput, next) {
      var itemId = remoteMethodOutput.itemId;
      console.log("calculating new rating for item: " + itemId);

      var searchQuery = {include: {relation: 'reviews'}};
      Item.findById(
        itemId,
        searchQuery,
        function findItemReviewRatings(err, findResult) {
          var reviewArray = findResult.reviews();
          var reviewCount = reviewArray.length;
          var ratingSum = 0;

          for (var i = 0; i < reviewCount; i++) {
            ratingSum += reviewArray[i].rating;
          }

          var updatedRating =
            Math.round((ratingSum / reviewCount) * 100) / 100;
          console.log("new calculated rating: " + updatedRating);

          findResult.updateAttribute(
            "rating",
            updatedRating,
            function (err) {
              if (!err) {
                console.log("item rating successfully updated");
              } else {
                console.log("error updating rating for item: " + err);
              }
            }
          );
          next();
        });
    });
};
```



Information

In this example, the `Item.afterRemote('prototype.__create__reviews', ...)` function listens to the create reviews operation. After the operation completes successfully, LoopBack calls the anonymous function that is stated as the second parameter of the `Item.afterRemote` call.

- ___ c. Examine the `findById` function call in the remote hook function implementation.

```
var searchQuery = {include: {relation: 'reviews'}};
Item.findById(
  itemId,
  searchQuery,
  function findItemReviewRatings(err, findResult) {
    ...
});
```



Information

The `Item.findById` function returns the `item` model with an identifier that matches the value of `itemId`. The search query includes the relationship that is named `reviews`. The purpose of the search query is to capture a list of `review` models that correspond to the `item` in question.

The `findById` function returns the search results to the `findResult` input parameter.

- ___ d. Examine the first half of the `findItemReviewRatings` function.

```
function findItemReviewRatings(err, findResult) {
  var reviewArray = findResult.reviews();
  var reviewCount = reviewArray.length;
  var ratingSum = 0;

  for (var i = 0; i < reviewCount; i++) {
    ratingSum += reviewArray[i].rating;
  }
  var updatedRating =
    Math.round((ratingSum / reviewCount) * 100) / 100;
```



Information

The `findItemReviewRatings` function processes the search results from the `Item.findById` function.

In the first half of this function, the `findResults.reviews()` call returns an array of `review` model object- that belongs to the `item` model. In other words, the function returns all reviews for the item in question. The section beginning with the `for` loop calculates the average review rating for this `item`.

- __ e. Review the remaining half of the **findItemReviewRatings** function.

```
findResult.updateAttribute(  
    "rating",  
    updatedRating,  
    function (err) {  
        if (!err) {  
            console.log("item rating successfully updated");  
        } else {  
            console.log("error updating rating for item: "+ err);  
        }  
    }  
};  
next();
```



Information

In the last step, you calculated and saved the average rating value in the `updatedRating` variable. The `findResult.updateAttribute` function updates the `item.rating` property.

The last command in the `findItemReviewRatings` function is the call to the `next` function. This function notifies the LoopBack framework that the remote hook completed its operation. You must add a call to the `next` function at the end of a remote hook.

- __ 5. Close the **Atom** and **Brackets** text editors.

5.5. Test the remote hook

To test the remote hook, create a review for an item in the inventory application. Use the assembly view test client to generate a sample review for the item record with an **id** value of 1.

The test operation also triggers the **after save** operation hook. The log function records the item record that is modified in the application log.

- 1. Open the API Designer web application.
 - a. Start the API Designer web application from the **inventory** application directory.
 - b. Open the **inventory** API definition.
 - c. Switch to the **assemble** view.
 - d. Open the test client.
- 2. Restart the micro gateway and inventory application.
 - a. In the application bar, click **restart**.
 - b. Confirm that the application is in the running state.

 | Running | Gateway: https://localhost:4002/ | Application: http://127.0.0.1:4001/ |

- 3. Create a review for the **item** record with an identifier value of 1.
 - a. In the test client, select the **post /item/{id}/reviews** operation.



- ___ b. In the **data** input parameter, click **Generate** to create sample values for the review.

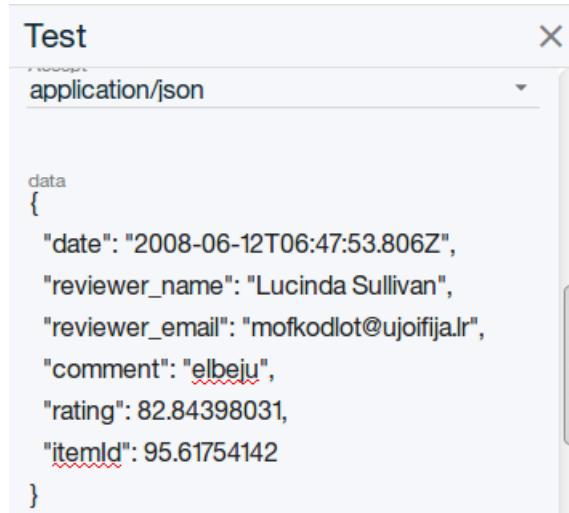


```
Test
data
{
  "date": "Thu Jun 17",
  "reviewer_name": "Annie Powell",
  "reviewer_email": "eb@vo.tp",
  "comment": "ediagoru",
  "rating": 66.44525652,
  "id": 8072839267614720,
  "itemId": 54.25891453
}
```



Note

In the API Designer Toolkit V5.0.8.1, you must remove the id field and value from the sample data, since Loopback generates the id.



```
Test
Content-Type: application/json

data
{
  "date": "2008-06-12T06:47:53.806Z",
  "reviewer_name": "Lucinda Sullivan",
  "reviewer_email": "mofkodlot@ujofija.lr",
  "comment": "elbeju",
  "rating": 82.84398031,
  "itemId": 95.61754142
}
```

- ___ c. Set the **item id** to **1**. This parameter is the identifier for the item record that has a relationship to the review.

item id
Id *
1

- ___ d. Click **Invoke**.

- ___ e. Confirm that the operation created the item review.



The screenshot shows a 'Test' window with the following details:

- Status code: 200 OK
- Response time: 295ms
- Headers:
 - content-type: application/json; charset=utf-8
 - x-ratelimit-limit: 100
 - x-ratelimit-remaining: 99
- Body:


```
{
  "date": "2001-06-17T04:00:00.000Z",
  "reviewer_name": "Annie Powell",
  "reviewer_email": "eb@vo.tp",
  "comment": "ediagoru",
  "rating": 66.44525652,
```

- ___ f. Confirm that the **after save** operation hook logged the action in the application log.

```
2016-12-31T08:31:33.727Z pid:3264 worker:1 Accessed item matching {"id":1}
2016-12-31T08:31:33.763Z pid:3264 worker:1 calculating new rating for item: 1
2016-12-31T08:31:33.764Z pid:3264 worker:1 Accessed item matching {"id":1}
2016-12-31T08:31:33.793Z pid:3264 worker:1 new calculated rating: 35.47
2016-12-31T08:31:33.833Z pid:3264 worker:1 Saved item#1
2016-12-31T08:31:33.834Z pid:3264 worker:1 item rating successfully updated
```

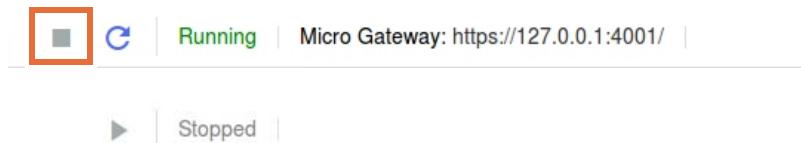


Information

When you call the **POST /item/1/reviews** operation, the **remote hook** intercepts the call and looks up the reviews that belong to the item. The function calculates the average review rating, and it updates the **Item.rating** property with the value.

The **after save** operation hook also intercepts the update operation on the **item** model object. It wrote the entry “**Saved item#1**” in the application log.

- ___ 4. Stop the Micro Gateway.



End of exercise

Exercise review and wrap-up

The first part of the exercise examined the role of operation hooks: event listeners that act on access or modifications to data source records that back the model objects.

The second part of the exercise examined the role of remote hooks: event listeners that act before or after the LoopBack framework calls the implementation for an API operation.

Exercise 6. Assembling message processing policies

Estimated time

02:00

Overview

This exercise explains how to create message processing policies. You define a sequence of policies in the assembly view of the API Designer. You define an API that exposes an existing SOAP service as a REST API. You also define an API that transforms responses from an existing service into a defined message format.

Objectives

After completing this exercise, you should be able to:

- Create an API with JSON object definitions and paths
- Configure an API to call an existing SOAP service
- Import an existing API definition into the source editor
- Map data from multiple API calls into an aggregate response
- Define and call a gateway script with an API assembly
- Test DataPower gateway policies in the API Manager explore view

Introduction

Message processing policies are a set of processing actions that you apply to API operation request and response messages. You assemble message processing policies as a sequence of actions in the **assembly view** of the **API Designer** web application.

The purpose of message-processing policies is to process requests to an API operation at the HTTP message level. You can transform, validate, or process a request message before it reaches the API implementation. You can also add logic constructs that route messages based on the content of the request message.

The API gateway enforces the message processing policies that you define in the API definition file. In effect, the message policies apply to all operations that you defined in the API definition.

The types of message processing policies that you can apply depend on your choice of API gateway. A number of policies run only on DataPower API Gateways.

In this exercise, you define two more API definitions: **financing** and **logistics**. In the financing API, you define a policy that converts REST API operations to SOAP API service requests. In the

logistics API, you define a sequence of processing policies that aggregates data from multiple sources.

Requirements

Before you start this exercise, you must complete the **data sources** and **remote** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

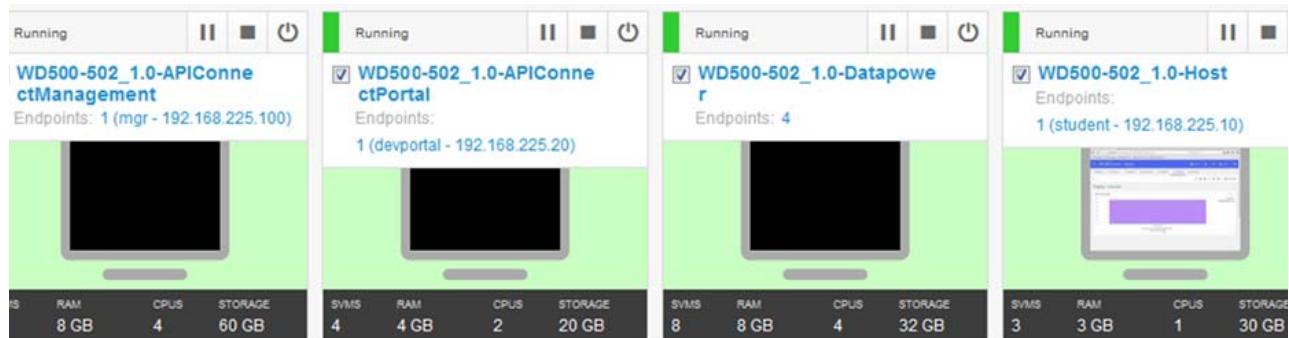
Exercise instructions

Before you begin

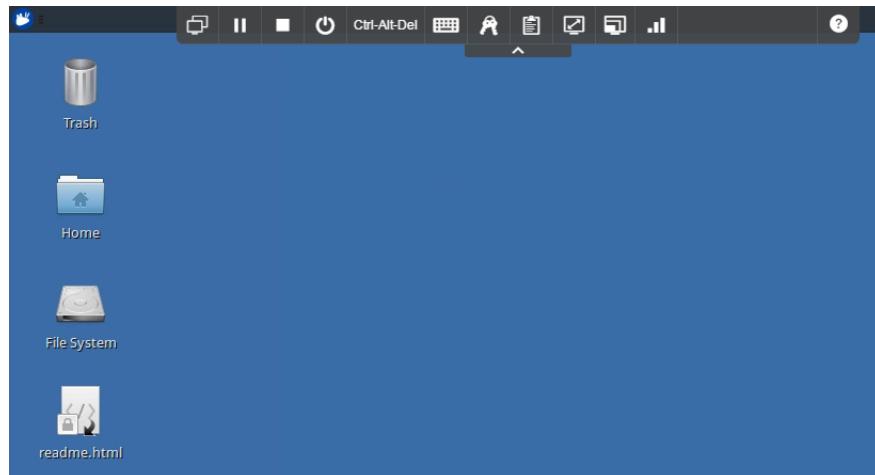
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



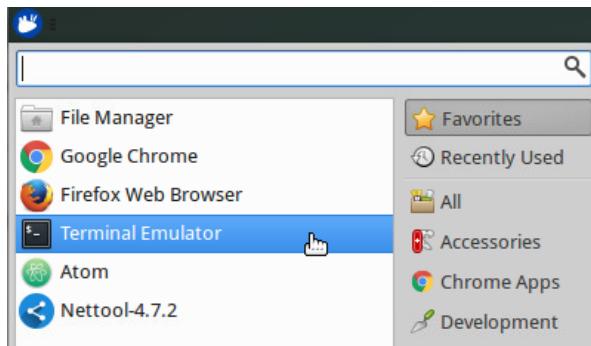
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

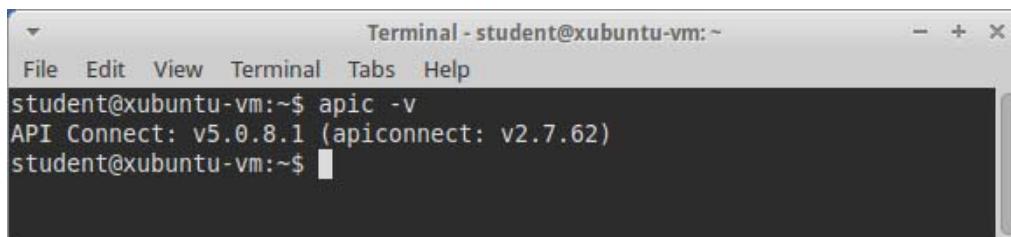
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

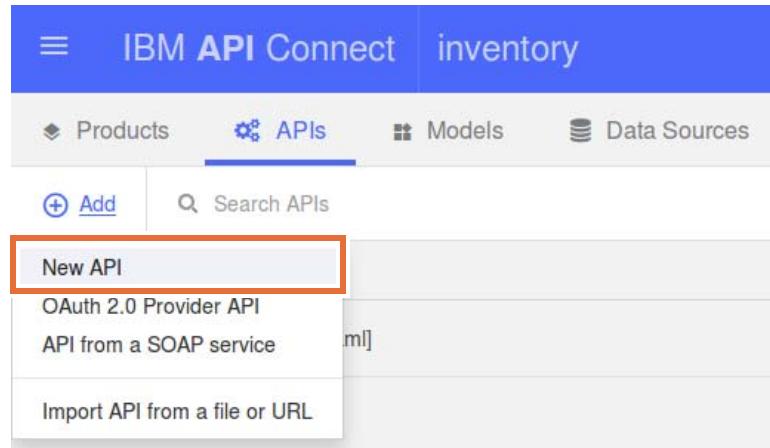
6.1. Create the financing API definition

In this section, you define a second API definition in your **inventory** application: the **financing** API. Map the API definition to a different base path from the inventory API. Configure the financing API to accept HTTPS connections from API clients. Set the request and response message body type to: **application/json**

- ___ 1. Open the API Designer web application.
 - ___ a. Open a terminal emulator window.
 - ___ b. Change the current directory to: `~/inventory`

```
$ cd ~/inventory
$ pwd
/home/student/inventory/
```
 - ___ c. Run the API Designer web application.


```
$ apic edit
```
- ___ 2. Create an API definition named **financing**.
 - ___ a. In the API Designer main page, switch to the **API** view.
 - ___ b. Select **Add > New API** to create an API definition.



___ c. Enter the following details for the **financing** API definition:

- Name: **financing**
- Title: **financing**
- Base path: **/financing**
- Version: **1.0.0**

New OpenAPI from scratch

Info	Title *
	financing
Name *	financing
Base Path	/financing
Version *	1.0.0
Additional properties ▾	
<input type="button" value="Cancel"/> <input type="button" value="Add a product..."/> <input type="button" value="Create API"/>	

___ d. Click **Create API**.

___ 3. Add a description for the financing API definition.

___ a. In the editor for the financing API definition, select the **info** category.

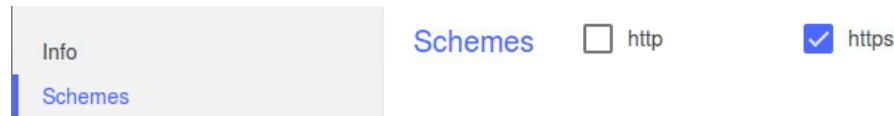
___ b. Change the description to: Operations for calculating financing payments

Info Schemes Host Base Path Consumes Produces Lifecycle Policy Assembly	Info Title financing Name financing Version 1.0.0 Description Operations for calculating financing payments
---	--

___ 4. In the **financing** API, accept **HTTPS** connections for the **Schemes**. Set the message body **consumes** and **produces** headings to: application/json

___ a. Select the **schemes** category.

- ___ b. Confirm that the scheme is set to **https**.



- ___ c. Select the **Base Path** category.
 ___ d. Confirm that the Base Path is set to: /financing
 ___ e. In the **consumes** and **produces** sections, set each of these sections to **application/json**.

Base Path
Base Path: /financing

Consumes application/json application/xml
Additional media types
Add Media Type

Produces application/json application/xml
Additional media types
Add Media Type

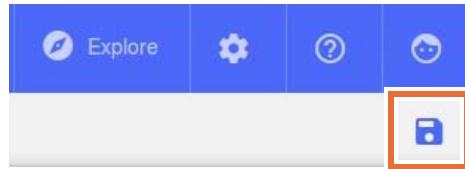
- ___ 5. Remove the client ID header security requirement.
- ___ a. Select the **Security** section of the API definition.
 ___ b. Clear the **clientIdHeader (API Key)** security requirement.

Security ⊕

Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.

Option 1	<input type="checkbox"/> clientIdHeader (API Key)	✖
----------	---	--

- ___ 6. Save the changes in the editor.



Note

In the last step, you defined an API named **financing** that accepts API requests at the **/financing** base path. The API accepts **HTTPS** requests with a message body payload of type **"application/json"**. By default, all API operations return responses of type **"application/json"**.

In the next section, you define a JSON data model named **paymentAmount**. The API operations in the **financing** API use the **paymentAmount** object as an input parameter.

- ___ 7. Create an object definition named **paymentAmount**.

- ___ a. In the **financing** API, select the **Definitions** section.

- ___ b. Select the plus sign (+) to add a definition.
 ___ c. Click **new-definitions-1** to open the definition in the editor.
 ___ d. Change the name of the definition to: **paymentAmount**

Name	Type
paymentAmount	object

- ___ e. Select the sample property named **new-property-1**.

___ f. Enter the following property values:

- Property name: `paymentAmount`
- Description: `Monthly payment amount`
- Type: `float`
- Example: `199.99`

Properties					Add Property
*	Property Name	Description	Type	Example	Actions
<input type="checkbox"/>	<code>paymentAmount</code>	<code>Monthly payment amount</code>	float ▾	<code>199.99</code> ▾	< > 

___ 8. Save the changes to the financing API.

6.2. Create an API operation in the financing API

In the previous section, you defined a definition object named **paymentAmount**. This object represents the financing payment that the API caller enters as an input parameter.

In this section, create an API operation that calculates the financing costs. This operation takes a **paymentAmount** data type as an input parameter.

- ___ 1. Define an API path that is named: **/calculate**
 - ___ a. In the **financing** API definition, select the **paths** section.
 - ___ b. Select the plus sign (+) to create a path.
 - ___ c. Set the **path** to **/calculate**.

The screenshot shows the 'Paths' section of an API definition. It contains a single path entry: '/calculate'. To the right of the path, there are two buttons: 'Add Operation' and 'Add Parameter'. A '+' icon is located at the top right of the list.

Information

The base path for the financing API is **/financing**. Combined with the API operation path, the URL path for the **calculate** resource is <https://<hostname>:<port>/financing/calculate>.

- ___ 2. Define a **GET /calculate** operation that calculates a finance rate. Pass the amount to finance as a query parameter in the GET /calculate operation.
 - ___ a. In the **/calculate** path, select the **GET /calculate** operation to edit the options for the resource.
 - ___ b. Set the **operation ID** to **get.financingAmount**.

The screenshot shows the configuration for a 'GET /calculate' operation. At the top, it displays 'GET /calculate'. Below this, there are sections for 'Summary' and 'Operation ID', which is set to 'get.financingAmount'. There is also a 'Description' field at the bottom.



Information

An operation ID is an HTTP header field that uniquely identifies the API operation.

After an application calls the **GET /calculate** API operation, the API gateway routes the request through one or more application servers. To track which API operation the client called, the gateway adds an operation ID header in the request message.

- ___ c. Click the **Add parameter** link *three times* to add three parameters.

Parameters						Add Parameter
Name	Located In	Description	Required	Type		
api_parameter-1	Query		<input type="checkbox"/>	string	<input type="button" value="Delete"/>	
api_parameter-2	Query		<input type="checkbox"/>	string	<input type="button" value="Delete"/>	
api_parameter-3	Query		<input type="checkbox"/>	string	<input type="button" value="Delete"/>	

- ___ d. Set the parameters according to the table of values.

Table 7. GET /calculate operation parameters

Name	Located in	Description	Required	Type
amount	Query	amount to finance	Yes	number-float
duration	Query	length of term in months	Yes	integer-int32
rate	Query	interest rate	Yes	number-float

Parameters

Add Parameter

Name	Located In	Description	Required	Type
amount	Query	amount to finance	<input checked="" type="checkbox"/>	number-float
duration	Query	length of term in mont	<input checked="" type="checkbox"/>	integer-int32
rate	Query	interest rate	<input checked="" type="checkbox"/>	number-float

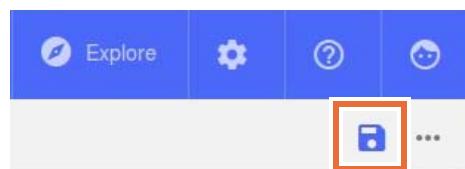
- ___ e. Scroll down to the responses section of the **GET /calculate** API operation.
 ___ f. In the response setting for status code **200**, leave the description as **200 OK**.
 ___ g. Set the response message to the **paymentAmount** model definition that you created earlier.

Responses

Add Response

Status Code	Description	Schema
200	200 OK	<u>paymentAmount</u>

- ___ 3. Save the API definition.



6.3. Invoke a SOAP web service with a message processing policy

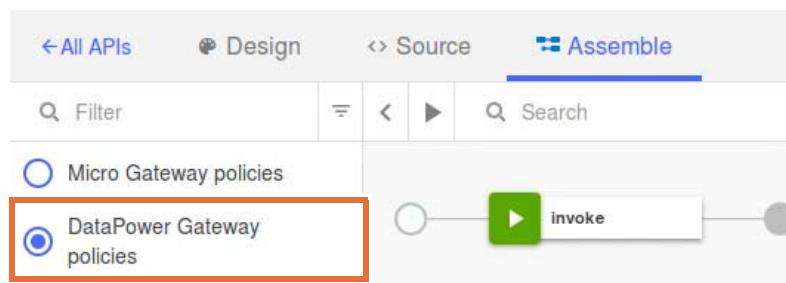
In the last section, you created an API operation that is named **GET /calculate** in the **financing** API. The operation takes three query parameters: the amount to finance, the length of the financing term in months, and the interest rate. The operation returns a JSON object of type **paymentAmount**: the monthly payment cost.

In this section, you assemble a message processing policy that calculates the payment amount. You invoke a remote SOAP web service to perform the calculation. You attach the message policy to a REST API operation. In effect, you create a bridge that converts a REST API operation to a SOAP web service invocation.

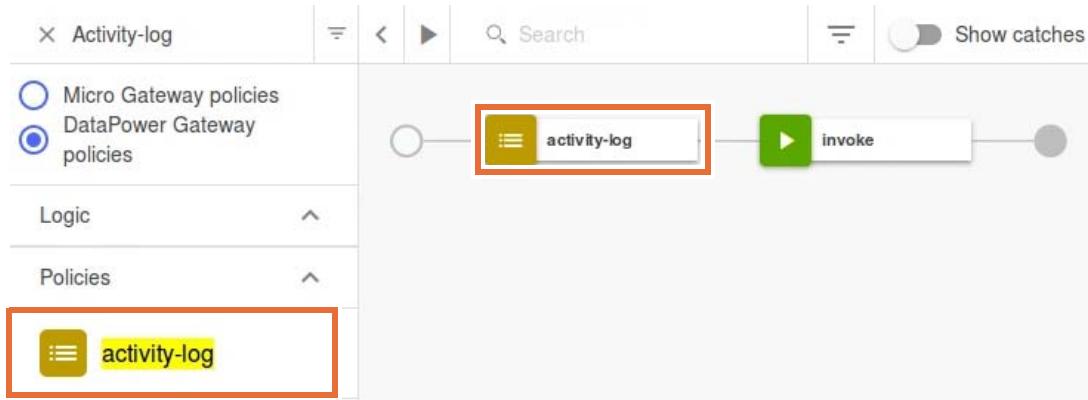
- ___ 1. Open the **Assemble** view for the **financing** API definition.



- ___ 2. Select the **DataPower Gateway policies**.

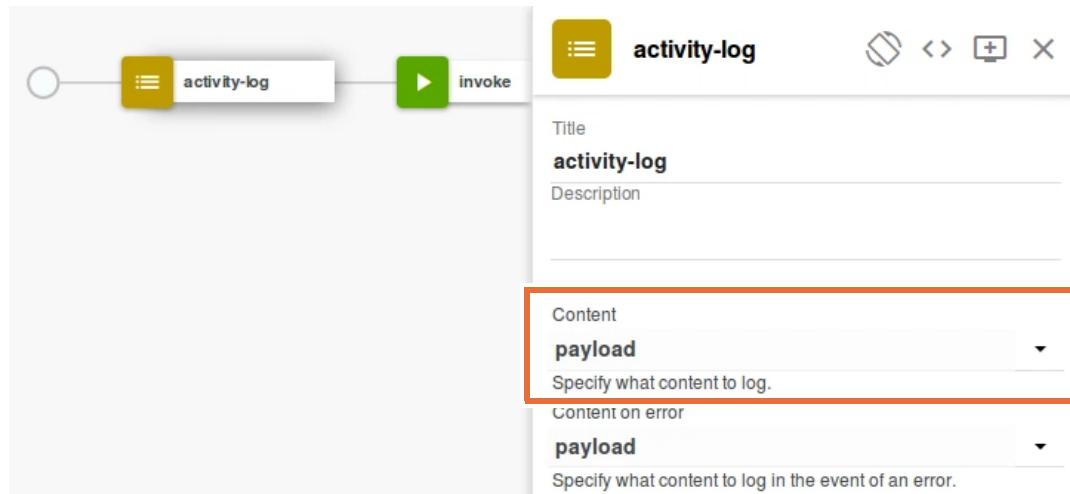


- ___ 3. Add an **activity-log** policy to record the request message payload to the API gateway logs.
 - ___ a. In the policy list, scroll down to the **activity-log** policy.
 - ___ b. Drag the policy and place it **in front** of the **invoke** policy.

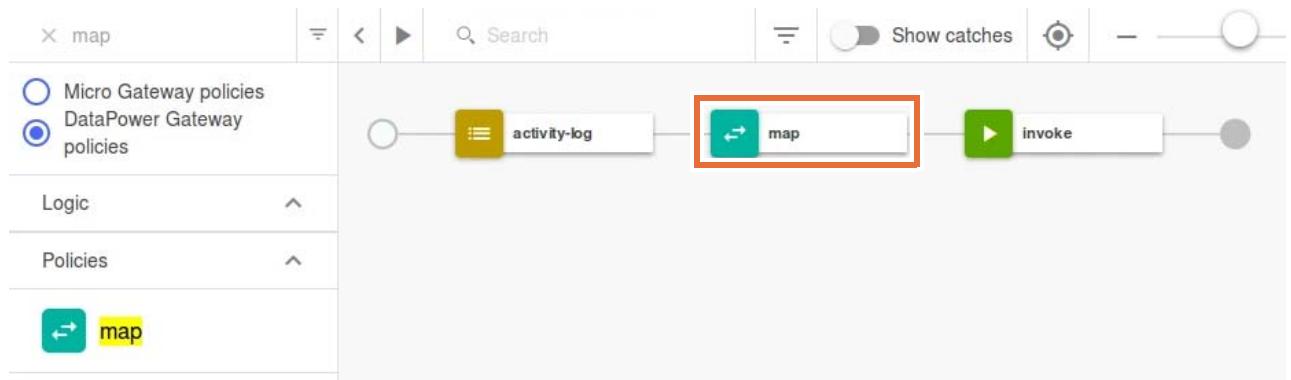


- ___ c. Select the **activity-log** policy in the message processing pipeline. The configuration options for the policy open in a view.

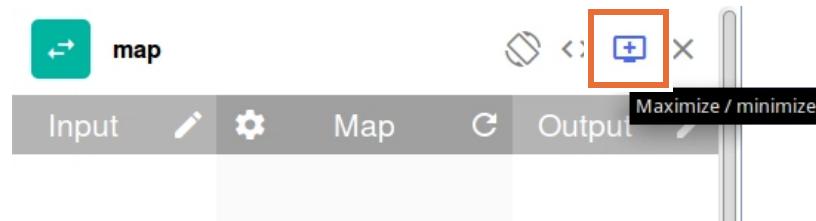
- ___ d. Change the **content** setting for the **activity-log** to **payload**.



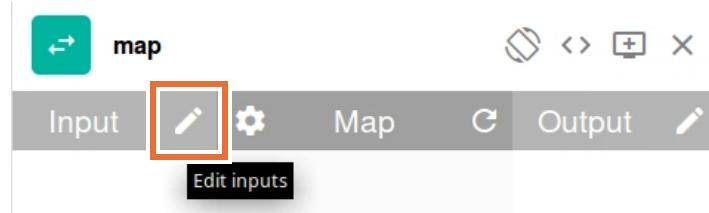
- ___ e. Close the activity-log properties window to return to the message processing policy pipeline.
- 4. Map the input parameters from the **GET /calculate** API request to the SOAP request message.
- ___ a. Select the **map** policy from the **Policies** palette.
- ___ b. In the policy pipeline, add a **map** policy between the **activity-log** and **invoke** steps.



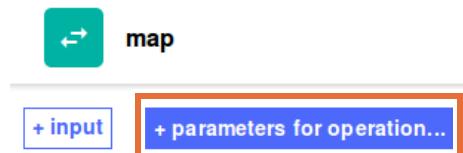
- ___ c. Open the properties for the **map** policy action.
- ___ d. Click the plus sign (+) icon to expand the **map** properties window.



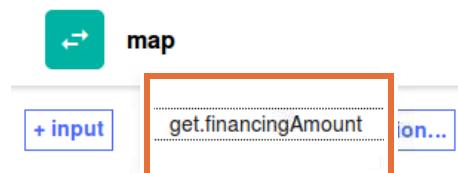
- ___ 5. Create three input parameters for the **amount**, **duration**, and **rate** query parameters.
- ___ a. In the **map** properties window, click the **edit** pencil icon in the **Input** column.



- ___ b. In the **input** editor, click **+ parameters for operation**.



- ___ c. Click the **get.financingAmount** operation to create a set of input parameters based on the request message from the operation.



- ___ d. Set the **definition** for the **request.parameters.amount** and **request.parameters.rate** parameters to **float**.

Context variable	Name
<code>request.parameters.amount</code>	<code>amount</code>

Content type	Definition *
<code>none</code>	<code>float</code>

Context variable	Name
<code>request.parameters.duration</code>	<code>duration</code>

Content type	Definition *
<code>none</code>	<code>integer</code>

Context variable	Name
<code>request.parameters.rate</code>	<code>rate</code>

Content type	Definition *
<code>none</code>	<code>float</code>

+ input
+ parameters for operation...
Done

- ___ e. Confirm that the input parameters match the following table.

Table 8. Financing API calculate operation input parameters

Context variable	Name	Content type	Definition
<code>request.parameters.amount</code>	<code>amount</code>	<code>none</code>	<code>float</code>
<code>request.parameters.duration</code>	<code>duration</code>	<code>none</code>	<code>integer</code>
<code>request.parameters.rate</code>	<code>rate</code>	<code>none</code>	<code>float</code>

- ___ f. Click **Done**.



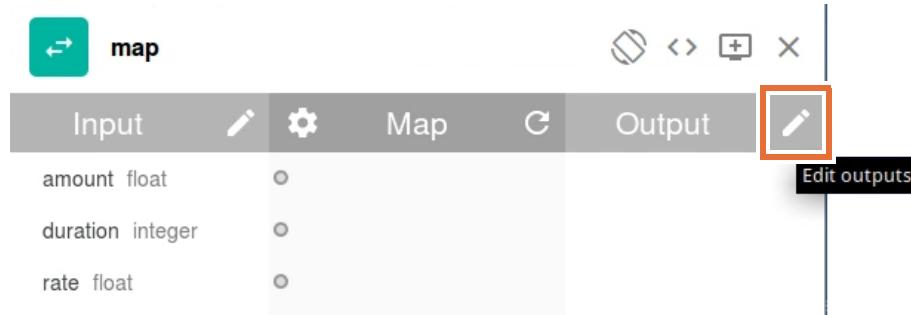
Note

To avoid typing errors, you copy the output parameters of the map policy action from a sample schema file.

The `schema_financingSoap.yaml` file contains an OpenAPI definition of the SOAP request message. This definition describes the XML elements and attributes of the SOAP message.

__ 6. Edit the **Output** parameters of the **map** policy.

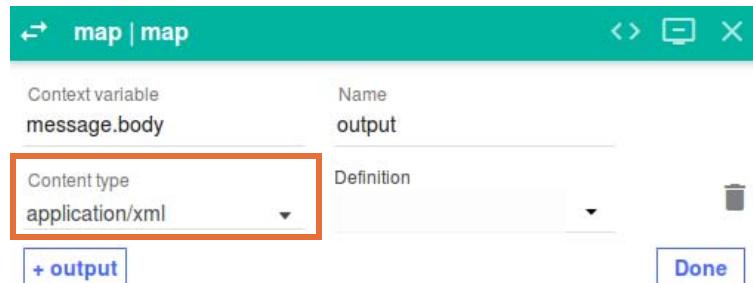
__ a. In the **map** policy editor, click the **edit outputs** icon from the **Output** column.



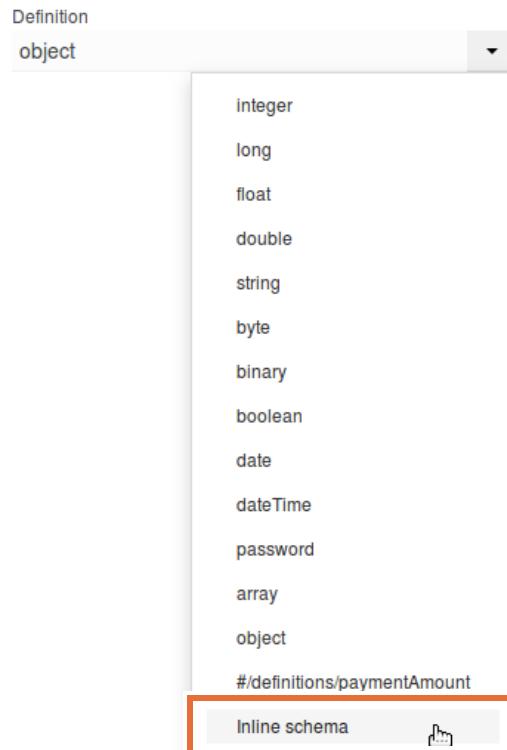
__ b. Click **+ output** to add an output parameter.

__ c. Leave the context variable to **message.body**.

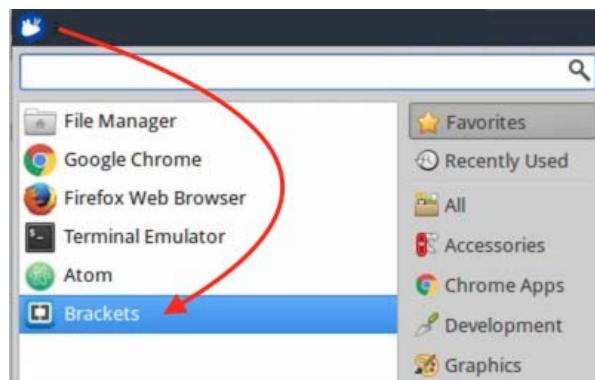
__ d. Set the **Content type** to **application/xml**.



- ___ e. In the **Definition** field, scroll down and select **inline schema**.

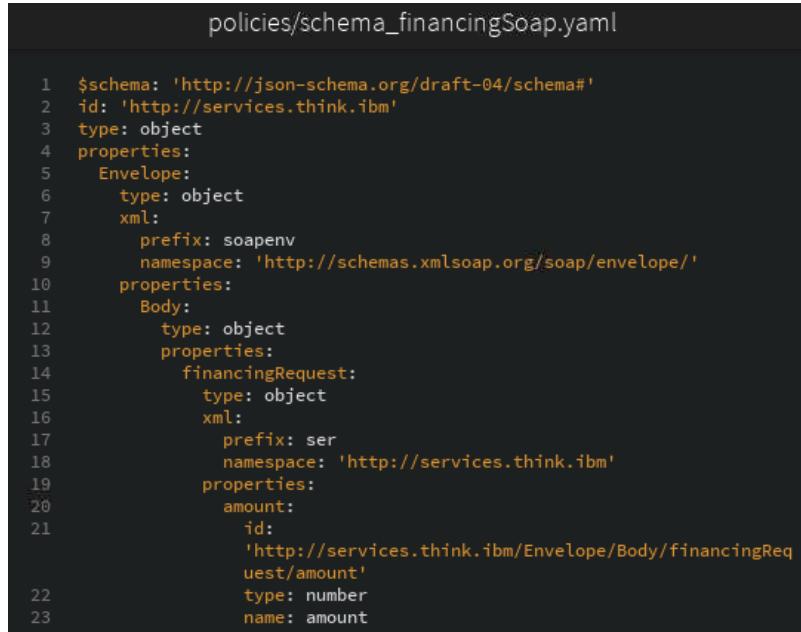


- ___ f. Leave the inline schema editor open in your web browser.
- ___ 7. Copy the output parameters for the **map** policy from the `schema_financingSoap.yaml` file.
- ___ a. Open the **Brackets** text editor.



- ___ b. Expand the `lab_files/policies` folder in the directory column.
- ___ c. Open the `schema_financingSoap.yaml` file.

- ___ d. Copy the contents of the `schema_financingSoap.yaml` document to the system clipboard.



```

1 $schema: 'http://json-schema.org/draft-04/schema#'
2 id: 'http://services.think.ibm'
3 type: object
4 properties:
5   Envelope:
6     type: object
7     xml:
8       prefix: soapenv
9       namespace: 'http://schemas.xmlsoap.org/soap/envelope/'
10    properties:
11      Body:
12        type: object
13        properties:
14          financingRequest:
15            type: object
16            xml:
17              prefix: ser
18              namespace: 'http://services.think.ibm'
19            properties:
20              amount:
21                id:
22                  'http://services.think.ibm/Envelope/Body/financingRequest/amount'
23                type: number
24                name: amount

```

- ___ e. Switch back to your web browser.
 ___ f. In the **inline schema** window for the **map output** parameter editor, paste the contents of the `schema_financingSoap.yaml` file.

Provide a schema



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

```

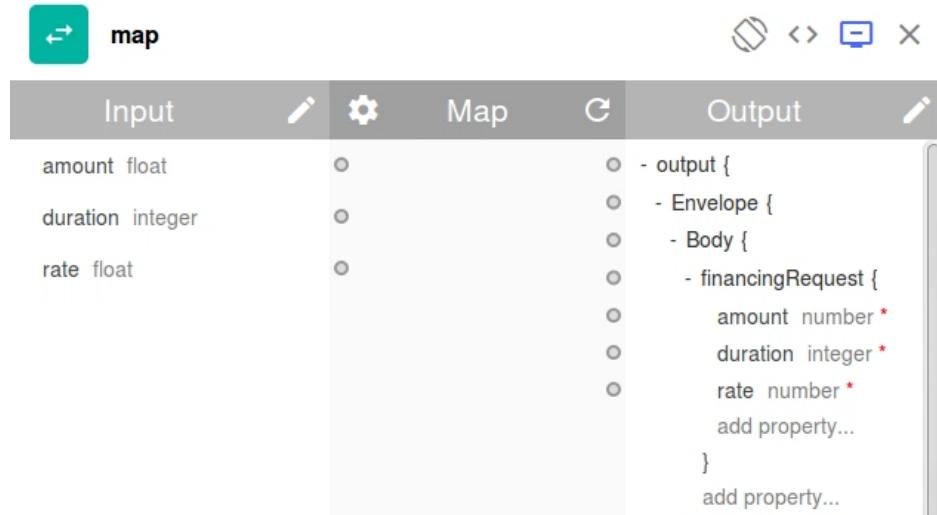
1 $schema: 'http://json-schema.org/draft-04/schema#'
2 id: 'http://services.think.ibm/Envelope/Body/financingRequest/rate'
3 type: number
4 name: rate
5 additionalProperties: false
6 required:
7   - amount
8   - duration
9   - rate
10  name: financingRequest
11  additionalProperties: false
12  required:
13    - financingRequest
14  name: Body
15  additionalProperties: false
16  required:
17    - Body
18  name: Envelope
19  additionalProperties: false
20  required:
21    - Envelope
22  title: output

```

Done **Cancel**

- ___ g. Click **Done** in the inline schema editor.
 ___ h. Click **Done** in the **output** parameters editor.

- ___ 8. Review the **map** policy action.



Information

On the left side, the input to the **GET /calculate** API operation consists of three query variables: **amount**, **duration**, and **rate**. On the right side, the SOAP web service accepts three parameters in the XML SOAP request message.

Your task is to map the three input parameters to the XML elements of the same name in the output map column. At run time, the API gateway creates a SOAP web service request and copies the values in the message body.

- ___ 9. Map the **amount**, **duration**, and **rate** API input parameters to the SOAP web service request.
- ___ a. Click the **node (circle)** that follows the **amount** input parameter.
 - ___ b. Click the **node (circle)** at the **amount** parameter in the Output column.

- ___ c. Repeat the process to connect the **duration** and **rate** parameters to the output parameters of the same name.



- ___ 10. Close the **map** policy editor.
- ___ 11. Configure the **invoke** policy to call the SOAP web service.
- Select the **invoke** policy.
 - Change the **URL** field to: <https://services.think.ibm:1443/financing>

The screenshot shows the 'invoke' policy editor interface. At the top, there is a green play button icon and the word 'invoke'.

Below the title and description fields, there is a 'URL *' field containing the value <https://services.think.ibm:1443/financing>.

A tooltip below the URL field says: 'The URL to be invoked.'

- ___ c. Set the **HTTP method** to **POST**.

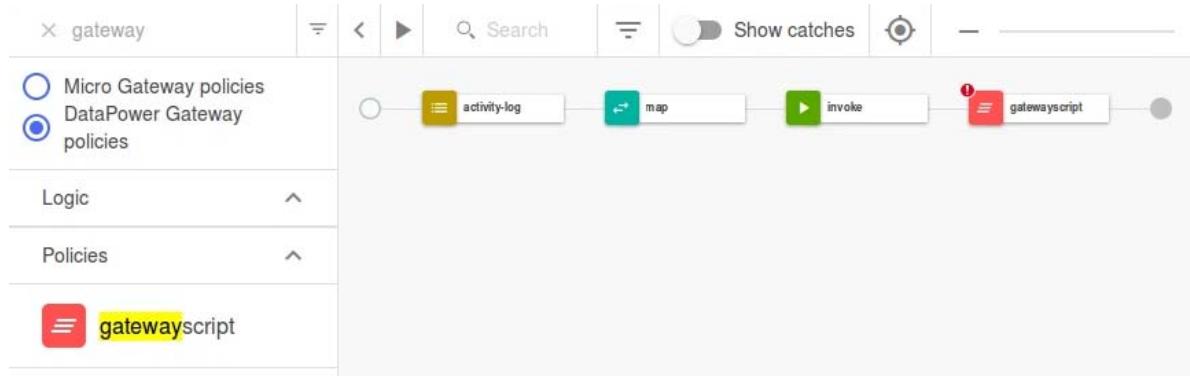
HTTP Method *

POST

The HTTP method to use for the invocation. If omitted or set to 'Keep', then the method from the incoming request will be used.

- ___ d. Close the **invoke** policy editor.

- __ 12. Configure a **gatewayscript** policy to set the 'content-type' header to: 'application/xml'
- __ a. Drag a **gatewayscript** policy *after* the **invoke** policy in the pipeline.



- __ b. Open the **gatewayscript** policy editor.
- __ c. Enter the following code:

```
session.name('_apimgmt').setVar('content-type', 'application/xml');
```



- __ d. Close the gatewayscript editor.
- __ 13. Add an **xml-to-json** policy to convert the SOAP service response to a JSON message.
- __ a. Select the **xml-to-json** policy from the **policies** palette.
- __ b. Add an **xml-to-json** policy *after* the **gatewayscript** policy.



- __ 14. Save the changes to the **financing** API.



Information

In the first half of this exercise, you created an API definition named **financing**. The financing API defines one REST API operation: **GET /calculate**. To call the API operation, you send an HTTP **GET** request with the following URL:

https://<hostname>:<port>/financing/calculate?amount=<amount>&duration=<duration>&rate=<rate>

You also defined a message processing policy that transforms the REST API operation into a SOAP web service call.

- The **activity-log** policy saves a copy of the REST API operation request into the API gateway logs.
- The **map** policy copies the **amount**, **duration**, and **rate** query parameters into a SOAP request message. These three input parameters map to XML elements of the same name.
- The **invoke** policy makes a SOAP service request, and captures the response message.
- The **gatewayscript** policy takes the SOAP response message and adds an HTTP header that is named `content-type` with a value of `application/xml`.
- The **xml-to-json** policy takes the SOAP response message body and converts the value into a JSON message.

6.4. Create the logistics API definition

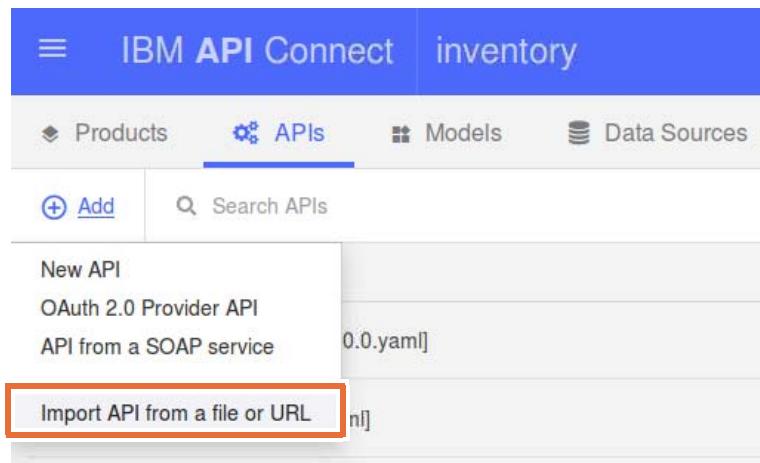
In the second half of this exercise, you explore the logic constructs in message processing policies. Specifically, you create a policy that routes an API operation request based on the contents of the request message.

You define a third API named **logistics**. This API provides two REST APIs:

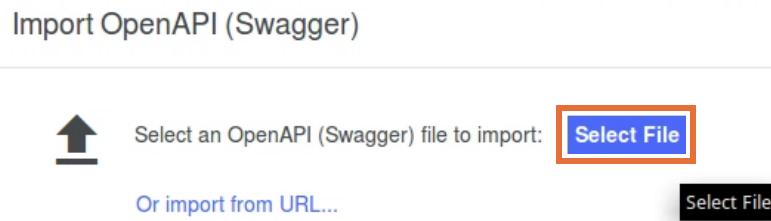
- The **GET /stores** operation returns the address of a store location based on the postal code (ZIP code) that you provide.
- The **GET /shipping** operation returns two shipping rate quotations. The service combines, or aggregates, the responses from two REST API calls.

In the interest of time, you import a predefined OpenAPI definition file, **logistics_1.0.0.yaml**. You complete the API definition by creating assembly flows for each API operation.

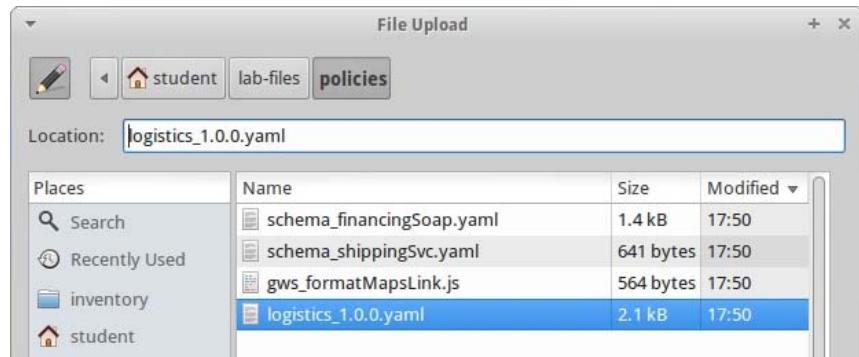
- ___ 1. Open the main page in the API Designer web application.
 - ___ a. Click **All APIs** to return to the main page.
 - ___ b. Select **Add > Import API from a file or URL** to define an API.



- ___ c. In the **Import OpenAPI (Swagger)** wizard, click **Select File** to import a local copy of the API definition file.



- ___ d. Select **logistics_1.0.0.yaml** from the `/home/student/lab_files/policies` directory.



- ___ e. Click **Open** to return to the import wizard.

- ___ f. Click **Import**.

Import OpenAPI (Swagger)



Select an OpenAPI (Swagger) file to import: **logistics_1.0.0.yaml**

[Or import from URL...](#)

Add a product

[Cancel](#)

Import



Note

The **Import** wizard in **API Designer** copies the API definition file into the root directory of your LoopBack application. However, the product and API definition files belong in the `definitions` folder.

Move the `logistics_1.0.0.yaml` file into the `definitions` folder. Run the `apic loopback:refresh` command to update the location of the file. Restart the **API Designer** web application.

- ___ 2. Exit **API Designer**.

- ___ a. Close the **API Designer** web browser page.
- ___ b. In the terminal emulator window from which you started API Designer, press **Ctrl+C** to stop the application.

___ 3. Move the `logistics_1.0.0.yaml` file into the `definitions` directory.

___ a. Move `logistics_1.0.0.yaml` to the `~/inventory/definitions/` directory.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
$ mv logistics_1.0.0.yaml definitions
```

___ b. Confirm that the `logistics_1.0.0.yaml` file appears in the `definitions` directory listing.

```
$ ls -f definitions
financing_1.0.0.yaml logistics_1.0.0.yaml inventory.yaml
inventory-product.yaml
```

___ c. Refresh the API definition files in the `inventory` application.

```
$ apic loopback:refresh
```

___ 4. Start the API Designer web application.

```
$ apic edit
```

___ 5. Remove the client ID header security requirement.

___ a. In the **API Designer** web application, open the **logistics** API.

___ b. Select the **Security** section of the API definition.

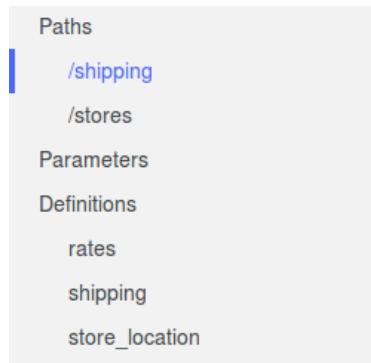
___ c. Clear the **clientID (API Key)** security requirement.

The screenshot shows the IBM API Connect interface in Design mode. The left sidebar has tabs for All APIs, Design (which is selected), Source, and Assemble. Under the Design tab, there are sub-tabs: Lifecycle, Policy Assembly, Security Definitions, and Security (which is also highlighted with a red box). The main content area is titled 'Security' and contains a box stating: 'Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.' Below this is a section for 'Option 1' with a checkbox labeled 'clientID (API Key)', which is also highlighted with a red box. A trash icon is visible next to the checkbox.

___ 6. Review the **API paths** in the **logistics** API definition.

___ a. Select the **Design** view.

- __ b. Select the **Paths** section.



- __ c. Select the **GET /shipping** API operation.

The image shows a configuration screen for a 'GET /shipping' API operation. At the top, it displays 'GET /shipping'. Below this, there is a 'Summary' section with the text 'Calculate shipping costs to a destination zip code'. Underneath, there is a 'Description' section. To the right, there is a 'Parameters' section with a 'Add Parameter' button. A table lists a single parameter: 'zip' (Name), 'Query' (Located In), 'Destination zip code' (Description), 'Required' (checkbox checked), and 'string' (Type). There is also a delete icon for this parameter row.

Name	Located In	Description	Required	Type
zip	Query	Destination zip code	<input checked="" type="checkbox"/>	string

The **GET /shipping** API operation calculates the shipping costs based on the destination postal code. The client sends the ZIP code as a query parameter.

- __ d. Open the **GET /stores** API operation.

The screenshot shows the configuration of a GET /stores API operation. At the top, it displays the method (GET), path (/stores), and a trash can icon for deletion. Below this, there is a tag section with 'stores' and an 'Add Tag' button. The summary is described as 'Locate store near zip code'. The operation ID is left blank. The description is also left blank. In the parameters section, there is a table with columns: Name, Located In, Description, Required, and Type. A single parameter 'zip' is listed, defined as a query parameter of type string, marked as required, and checked. There is also a 'Add Parameter' button.

The **GET /stores** API operation returns the location of the closest store. The client sends the ZIP code as a query parameter for this operation.

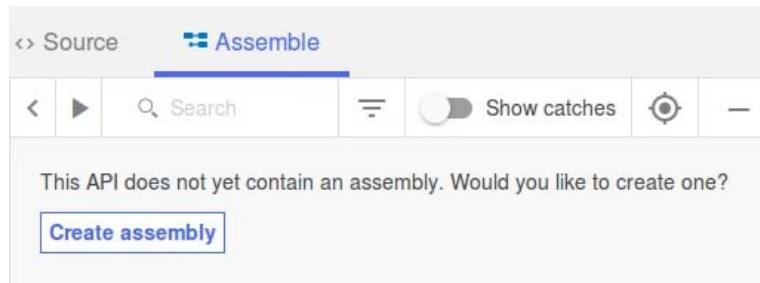
6.5. Define message processing policies for the logistics API operations

In this section, you define a set of message processing policies to retrieve the result of the two operations in the **logistics** API: **GET /shipping** and **GET /stores**.

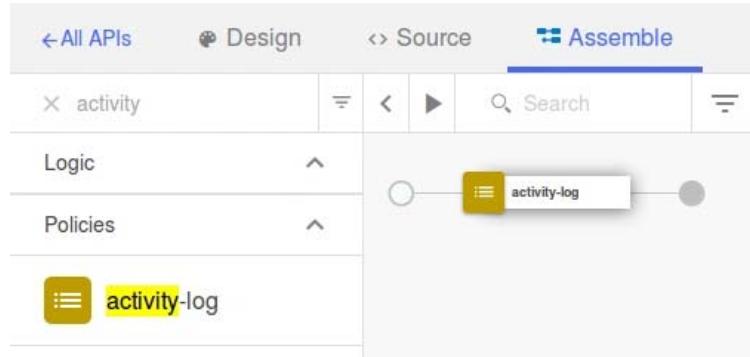
The **GET /shipping** API operation calls two more APIs to calculate the shipping cost. The **GET /stores** API calls a geolocation API with a gateway script to return a map link.

To distinguish between the two different types of API operation requests, use an **operation-switch** policy to route incoming requests based on the resource path.

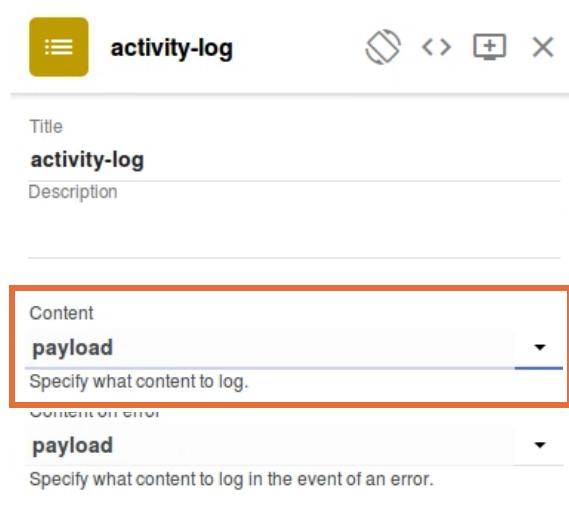
- 1. Open the **Assemble** view.
 - a. In the API Designer, switch to the **Assemble** view for the **logistics** API definition.
 - b. Click **Create assembly**.



- 2. Capture the API operation request payload with an **activity-log** policy.
 - a. Add an **activity-log** policy to the pipeline.



- __ b. Set the **Content** field to **payload** in the activity-log properties.

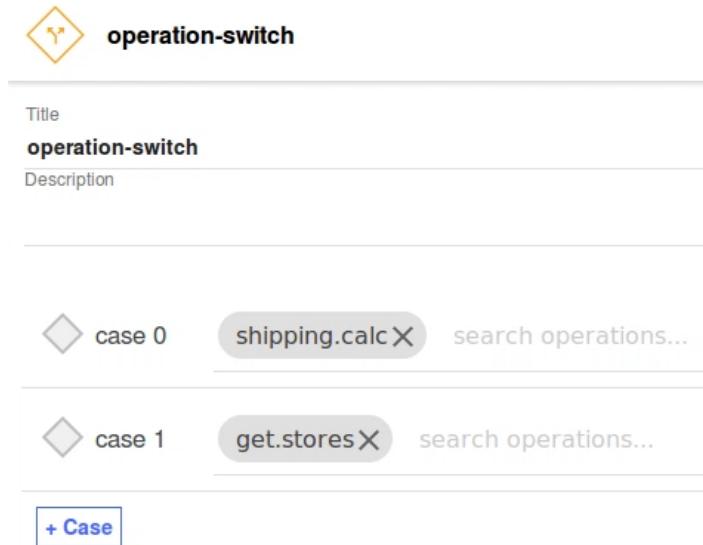


- __ 3. Add an **operation-switch** policy to distinguish between requests to **GET /shipping** and **GET /stores**.
- __ a. In the **Logic** palette, select the **operation-switch** construct.
 - __ b. Add an **operation-switch** policy after the **activity-log** policy.



- __ c. Select the **operation-switch** policy to open the Properties view.
- __ d. Select the **search operation** field in the **case 0** branch.
- __ e. Select **shipping.calc**.
- __ f. Select **+ Case** to create a case for the operation switch.

- ___ g. In **case 1**, select **get.stores** as the operation.

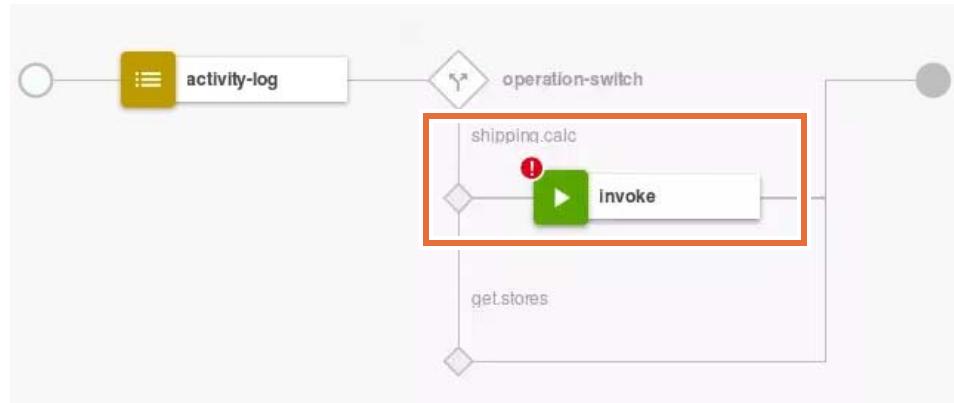


- ___ h. Close the properties view.
___ 4. Save the changes to the API definition.

6.6. Define the message policies for the GET /shipping API operation

The **logistics** API defines two API operations: **GET /shipping** and **GET /stores**. In this section, define a sequence of message processing policies that calls an external web service to calculate the shipping costs.

- __ 1. Add an **invoke** policy to calculate the shipping cost with the **xyz** shipping company.
 - __ a. Add an **invoke** policy in the **case 0** branch of the **operation-switch** policy.



- __ b. Open the properties view for the **invoke** policy.

__ c. Enter the following service details:

- Title: `invoke_xyz`
- URL: `$(shipping_svc_url)?company=xyz&from_zip=90210&to_zip={zip}`

Title
`invoke_xyz`

Description

URL *

`$(shipping_svc_url)?company=xyz&from_zip=90210&to_zip={zip}`

The URL to be invoked.

Stop on error

Defines whether the flow stops when a particular error is thrown during the policy execution. Errors not specified here will trigger a catch flow.

Response object variable
`xyz_response`

The name of a variable that will be used to store the response data from the request. This can then be referenced in other actions, such as 'Map'.

__ d. Close the Invoke properties view.



Note

The `$(shipping_svc_url)` is an environment-specific parameter that stores the network address for the shipping cost calculator service. The message processing policy calls the remote service, with the `zip` query parameter from the original `GET /shipping` API operation request.

- __ 2. Add a second **invoke** policy to calculate the shipping cost with the **cek** shipping company.
- __ a. In the **case 0** path, add an **invoke** policy to the *right* of the **invoke_xyz** policy.

- __ b. Enter the following properties in the **invoke** policy:

- Title: **invoke_cek**
- URL: **`$(shipping_svc_url)?company=cek&from_zip=90210&to_zip={zip}`**

The screenshot shows the 'invoke_cek' properties view. At the top is a green play button icon followed by the text 'invoke_cek'. Below this is a 'Title' field containing 'invoke_cek'. Underneath is a 'Description' section. The 'URL *' field contains the value '\$(shipping_svc_url)?company=cek&from_zip=90210&to_zip={zip}'. A note below the URL field says 'The URL to be invoked.'

- Clear **Stop on error**.
- Response object variable: **cek_response**

The screenshot highlights two sections: 'Stop on error' (checkbox) and 'Response object variable' (set to 'cek_response'). A note below the response variable says 'The name of a variable that will be used to store the response data from the request. This can then be referenced in other actions, such as 'Map'.'

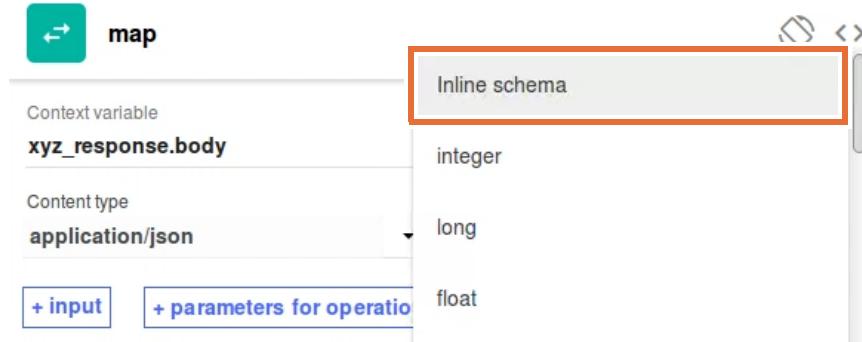
- __ c. Close the Invoke properties view.
- __ 3. Combine the results from the **invoke_xyz** and **invoke_cek** service calls into one response message.
- __ a. Add the **map** policy after the second invoke policy in the **case 0** branch.



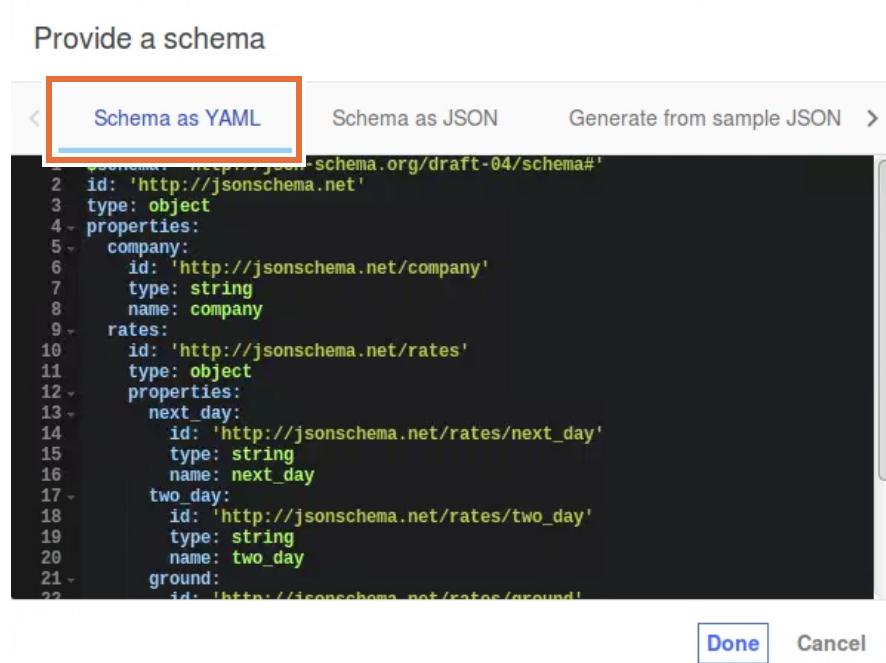
- __ b. Open the **map** policy **Properties** view.
- __ c. Click the **edit** icon in the **input** column.
- __ d. Click **+input** to add an input source.

__ e. Specify the following input message details:

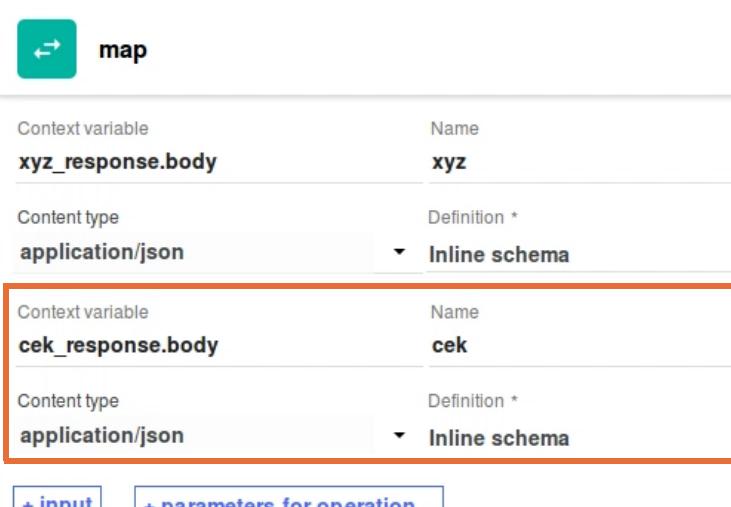
- Context variable: `xyz_response.body`
- Name: `xyz`
- Content type: `application/json`
- Definition: **Inline schema**

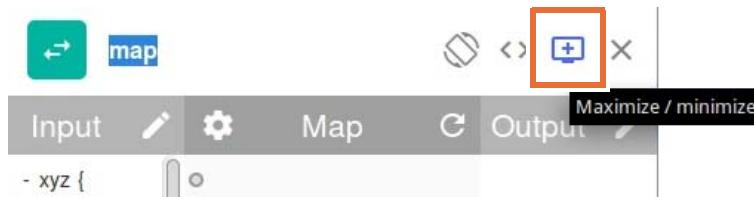


- __ f. Open `lab_files/policies/schema_shippingSvc.yaml` in the **Brackets** text editor.
 __ g. Copy the contents of `schema_shippingSvc.yaml`.
 __ h. Paste the clipboard contents into the **schema as YAML** window.



- __ i. Click **Done**.

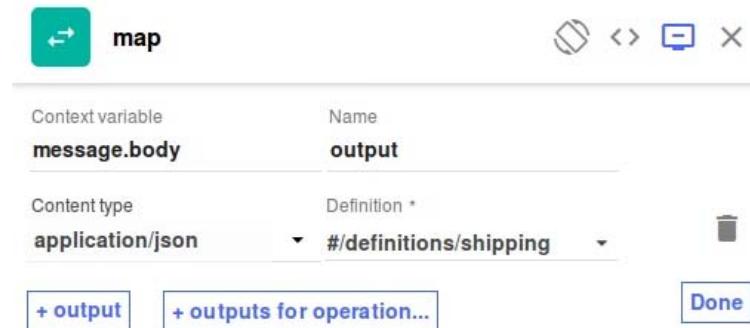
- ___ j. Add a second **Input** message in the **map** policy with the following properties:
- Context variable: `cek_response.body`
 - Name: `cek`
 - Content type: `application/json`
 - Definition: **inline schema**
- 
- ___ k. Copy the contents of the same `lab_files/policies/schema_shippingSvc.yaml` file into the **inline schema** window.
- ___ l. Click **Done**.
- ___ m. Click **Done**.
- ___ 4. Define an output message for the **GET /shipping** API operation in the **Output** column of the **map** policy.
- ___ a. Locate the **maximize icon (monitor +)** at the upper-right section of the **map** policy editor.
- ___ b. Click the **maximize** icon to expand the editor to the full width of the browser window.



- ___ c. Click the **edit (pencil)** icon in the **Output** column.
- ___ d. Click **+output** to add an output message.

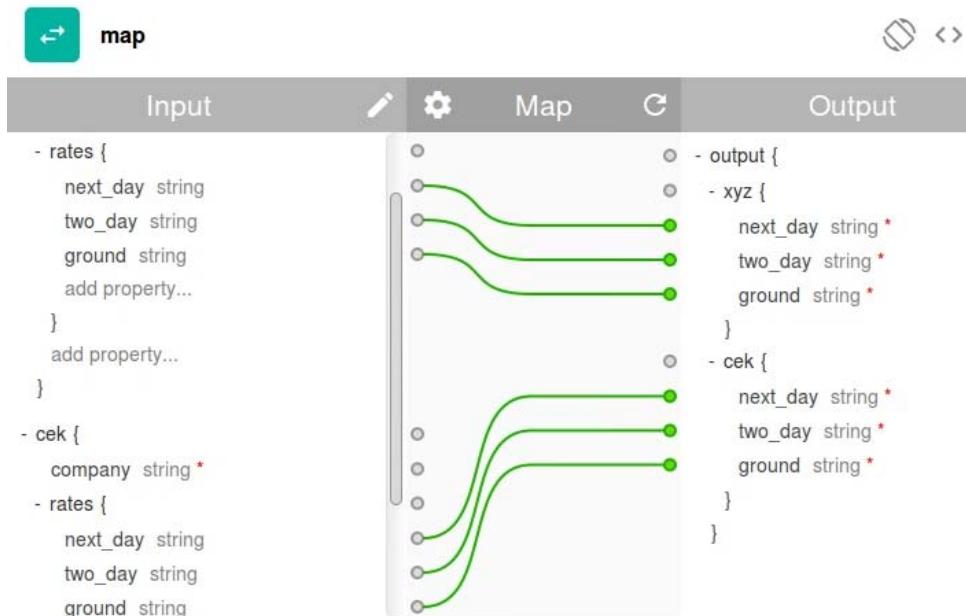
__ e. Enter the following details in the output message:

- Context variable: `message.body`
- Name: `output`
- Content type: `application/json`
- Definition: `#/definitions/shipping`



__ f. Click **Done**.

- __ 5. Map the responses from the `invoke_xyz` and `invoke_cek` calls to the **GET /shipping** response message.
- __ a. In the **map** policy editor, the `xyz_response.body` and `cek_response.body` messages appear in the **input** column. The final **output** message combines these results into one message.
 - __ b. Draw a line from each of the fields in the `xyz` input message to the `xyz` section of the output message.
 - __ c. Draw a line from each of the fields in the `cek` input message to the `cek` section of the output message.



- __ d. Close the **map** policy editor.
 - __ 6. Save the changes in the **logistics** API definition.
-



Information

In case zero (0), the **GET /shipping** API operation makes two external service calls. The two **invoke** policies save a shipping cost quote for the **xyz** and **cek** companies. The **map** policy copies the results from the two service calls into one response message.



In the next section, you define a set of message processing policies to handle case one (1), the **GET /stores** API operation.

6.7. Define the message policies for the GET /stores API operation

In this section, you define a sequence of message processing policies for the **GET /stores** API operation. Call a remote geolocation service to find a store location based on the client's postal code. Add a **gateway script** policy to format the contents of the API response message.

- ___ 1. Add an **invoke** policy to call a geolocation service.
 - ___ a. In the **GET /stores** case in the **operation-switch** policy, add an **invoke** policy.



- ___ b. Open the properties view for the **invoke** policy.
- ___ c. Enter the following details for the **invoke** policy:
 - Title: **invoke_geolocate**
 - URL:
`http://nominatim.openstreetmap.org/search?postalcode={zip}&format=json`

invoke_geolocate	
Title	invoke_geolocate
Description	
URL *	<code>http://nominatim.openstreetmap.org/search?postalcode={zip}&format=json</code>
The URL to be invoked.	

- Clear **Stop on error**.
- Response object variable: **geocode_response**

Stop on error

Defines whether the flow stops when a particular error is thrown during the policy execution. Errors not specified here will trigger a catch flow.

geocode_response

The name of a variable that will be used to store the response data from the request. This can then be referenced in other actions, such as 'Map'.



Information

The **Nominatim** service geocodes addresses for the Open Street Map project. You provide a part of an address, and the service returns the latitude and longitude coordinates for the location.

- d. Close the properties editor for the **invoke_geolocate** policy.
- 2. Add a **gatewayscript** policy to return a map link for the latitude and longitude coordinates that you received in the **invoke** policy.
 - a. Add a **gatewayscript** policy after the **invoke** policy in the **case 1** operation-script path.
 - b. Enter the following settings in the **gatewayscript** properties view:
 - Title: **format-maps-link**
 - c. Enter the following script into the policy:

```
// Require API Connect functions
var apic = require('local:///isp/policy/apim.custom.js');

// Save the geocode service response body to a variable
var mapsApiResponse = apic.getvariable('geocode_response.body');

// Get location attributes from the response message
var location = mapsApiResponse[0];

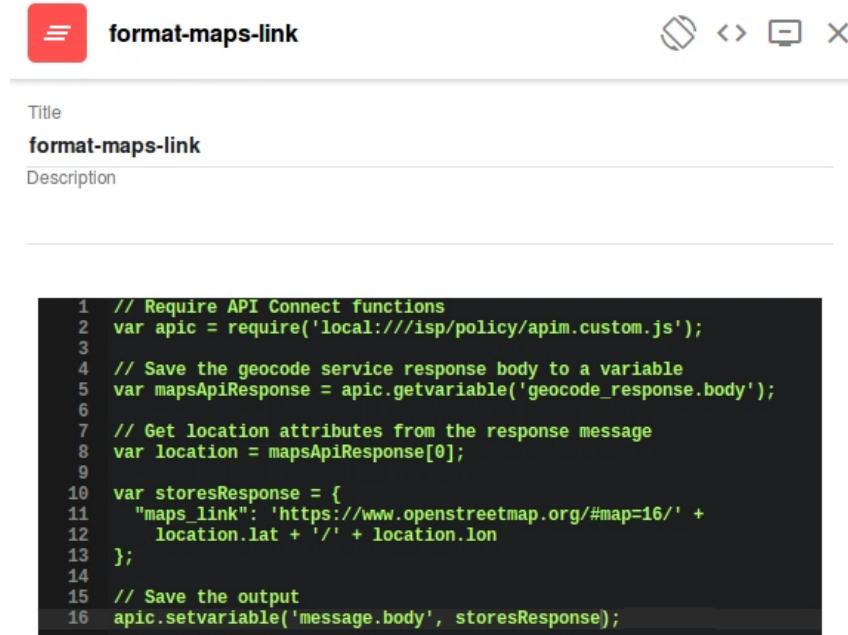
var storesResponse = {
  "maps_link": 'https://www.openstreetmap.org/#map=16/' +
    location.lat + '/' + location.lon
};

// Save the output
apic.setvariable('message.body', storesResponse);
```



Note

You can also copy the gateway script source code from the **formatMapsLink.js** script in the **~/lab_files/policies/** directory.



```

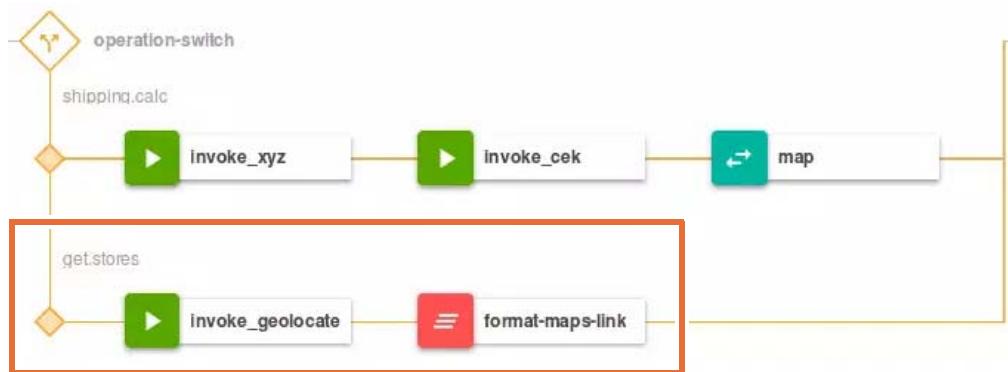
1 // Require API Connect functions
2 var apic = require('local:///isp/policy/apim.custom.js');
3
4 // Save the geocode service response body to a variable
5 var mapsApiResponse = apic.getvariable('geocode_response.body');
6
7 // Get location attributes from the response message
8 var location = mapsApiResponse[0];
9
10 var storesResponse = {
11   "maps_link": 'https://www.openstreetmap.org/#map=16/' +
12     location.lat + '/' + location.lon
13 };
14
15 // Save the output
16 apic.setvariable('message.body', storesResponse);

```

- ___ d. Close the **gatewayscript** editor.
 ___ 3. Save the logistics API definition.

Information

In the case one branch of the operation switch construct, the **GET /stores** API operation retrieves the map coordinates based on the US postal code (ZIP code) from the API request. The gateway script policy takes the latitude (location.lat) and longitude (location.lon) values to build an Open Street Map link.



6.8. Test the financing and stores API policy assemblies in the DataPower gateway

In this section, you publish the two API policy assemblies in the sandbox environment on the DataPower gateway. Unlike previous lab exercises, you must test these policies on a DataPower gateway. The micro gateway in the local workstation does not support the gateway script policy.

- 1. Add the **financing** and **logistics** API definitions to the **inventory** API product.
 - a. Switch to the **Products** tab in the API Designer main page.
 - b. Open the **inventory 1.0.0** product.

The screenshot shows the IBM API Connect interface with the 'Products' tab selected. Below the tabs, there is a search bar labeled 'Search products'. A product entry for 'inventory 1.0.0 [inventory-product.yaml]' is listed, with its entire title being highlighted by a red rectangular box.

- c. Select the **APIs** section.
- d. Click the plus (+) icon to add API definitions to the product.
- e. Select the **financing** and **logistics** API definitions to include in the product.

Select APIs

Select the APIs to include in this product. Any APIs removed from this list will also be removed from the plans in this product.

Search APIs

<input checked="" type="checkbox"/>	financing	1.0.0
<input checked="" type="checkbox"/>	inventory	1.0.0
<input checked="" type="checkbox"/>	logistics	1.0.0

Cancel

Apply

- f. Click **Apply**.
- 2. Save the changes to the **inventory** product definition.

- ___ 3. Define a publish target for the sandbox environment on the API Management server.
- ___ a. In the API Designer toolbar, click **Publish**.
 - ___ b. Select **Add and manage targets** to create a publish target.
 - ___ c. In the publish wizard, click **Add a different target**.

Publish



- ___ d. Enter the following credentials to log in to the API management server:
 - API Connect host address: `mgr.think.ibm`
 - User name: `student@think.ibm`
 - Password: `Passw0rd!`

Sign in to IBM API Connect

The dialog box contains the following fields:

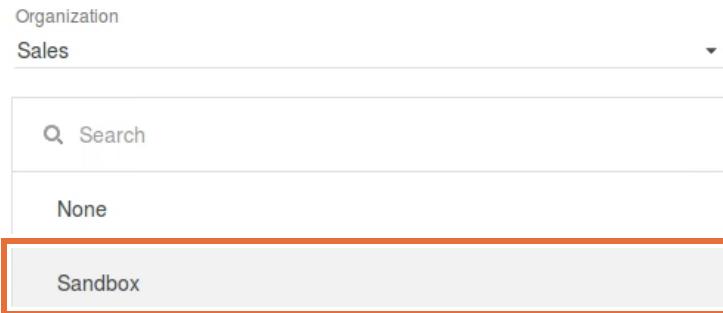
- API Connect host address: mgr.think.ibm
- Username: student@think.ibm
- Password: (redacted)

Buttons at the bottom: Back, Cancel, Sign in (highlighted).

- ___ e. Click **Sign in**.

- ___ f. Select the **Sales** organization and the **Sandbox** catalog.

Select an organization and catalog

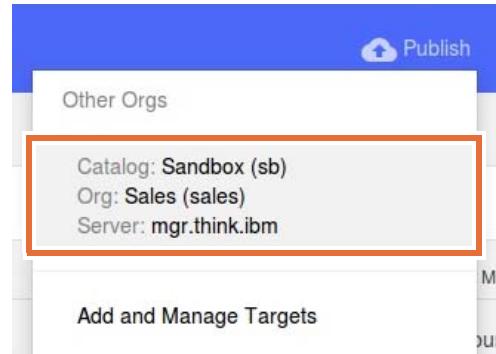


- ___ g. Click **Next**.
___ h. In the Select an App page, select **None**.

Select an App



- ___ i. Click **Save**.
___ 4. Publish the financing and logistics API definitions to the API Management server.
___ a. Click the **Publish** link again from the API Designer toolbar.
___ b. Select the **Sandbox** catalog publish target.



- ___ c. In the Publish window, leave the two options cleared.

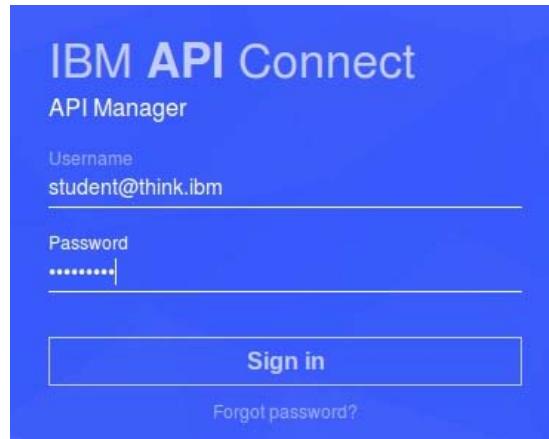
Publish

This target only has a catalog and no application. Application will not be published.

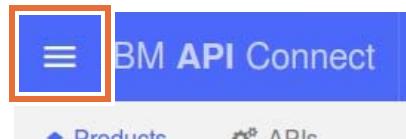
- Stage only
- Select specific products

[Cancel](#) [Publish](#)

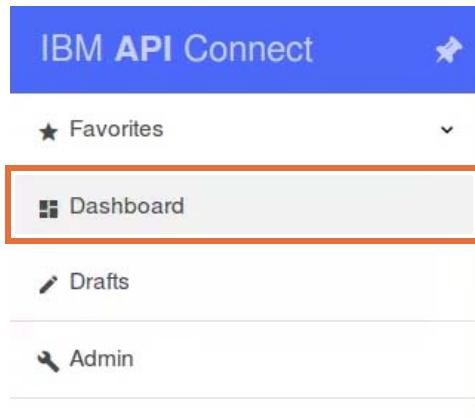
- ___ d. Click **Publish**.
- ___ e. Confirm that the **inventory** product was successfully published.
- ___ 5. Review the published product in the API Manager web page.
- ___ a. Open <https://mgr.think.ibm/apim/> in a web browser window.
- ___ b. Log in with the credentials:
- User name: **student@think.ibm**
 - Password: **Passw0rd!**



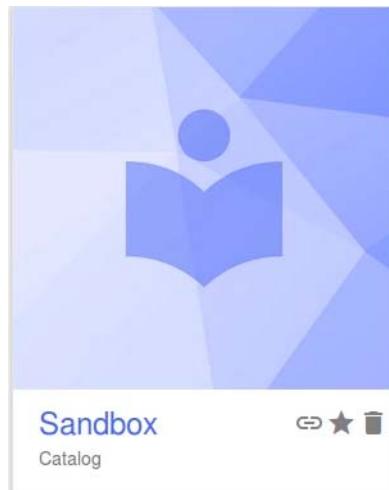
- ___ c. Select the API Manager menu at the upper-left section of the page.



- ___ d. Select the **Dashboard** view.



- ___ e. Select the **Sandbox** catalog from the dashboard.



- ___ f. Confirm that the **inventory** product is in the list of published products.

The screenshot shows the 'Products' tab selected in the navigation bar. Below the header, there is a search bar labeled 'Search products'. The main content area displays a table with one row. The row contains the product title 'inventory', its version 'inventory:1.0.0', and status information indicating it was 'Published' 4 minutes ago.

Title	State
inventory inventory:1.0.0	Published 4 minutes ago

- ___ 6. Open the test client in the API Manager server.

- ___ a. Click the **Explore** link in the API Manager sandbox page.

- __ b. In the Explore menu, select the **Sandbox** catalog.



- __ 7. Test the **financing** API from API Manager.
- __ a. From the explore page, select the **get.financingAmount** operation in the **financing 1.0.0** API definition.

Sandbox catalog <https://mgr.think.ibm/orgs/sales/catalogs/sb>

ItemService 1.0.0 inventory 1.0.0 financing 1.0.0 Search Operations by name	financing 1.0.0 REST Description Operations for calculating financing payments get.financingAmount Security Parameters amount number, required in query amount to finance duration integer, required in query length of term in months rate number, required in query interest rate
---	---

- ___ b. The test client appears on the third client on the right. Click **Try it** for the `GET https://api.think.ibm/sales/sb/financing/calculate` REST operation.

The screenshot shows a REST API test client interface. At the top, there is a blue button labeled "GET https://api.think.ibm/sales/sb/financing/calculate". Below this, there are two buttons: "Examples" and "Try it", with "Try it" being highlighted with a red border. Underneath these buttons is a section titled "Example request" with a dropdown menu set to "curl". Below the dropdown is a code block containing a curl command:

```
curl --request GET \
--url 'https://api.think.ibm/sales/sb/financing/calculate'
--header 'accept: application/json'
```

- ___ c. Type the following values for the **get.finacingAmount** REST API:

- Amount to finance: 300
- Length in terms of months: 24
- Interest rate: 0.04

The screenshot shows a form for the "get.finacingAmount" REST API. It has three input fields: "amount *", "duration *", and "rate *". Each field has a "Generate" link to its right. The "amount" field contains the value "300". The "duration" field contains the value "24". The "rate" field contains the value "0.04". At the bottom of the form is a large blue button labeled "Call operation".

- ___ d. Click **Call operation**.

- __ e. Confirm that the financing operation returns the SOAP response that is packaged as a JSON object in the HTTP response message.

```

Request
GET https://api.think.ibm/sales/sb/financing
/calculate?amount=300&duration=24&rate=0.04
Headers:
Content-Type: application/json
Accept: application/json

Response
Code: 200 OK
Headers:
content-type: application/json
x-global-transaction-id:
196c556559e4f52701460310

{
  "soapenv:Envelope": {
    "@xmlns": {
      "soapenv": "http://schemas.xmlsoap.org/soap/envelope/"
    },
    "soapenv:Body": {
      "@xmlns": {
        "soapenv": "http://schemas.xmlsoap.org/soap/envelope/"
      }
    }
  }
}

```

- __ 8. Test the **logistics** API shipping operation from the API Manager.

- __ a. Select the **shipping.calc** operation in the **logistics** API definition.

Sandbox catalog <https://mgr.think.ibm/orgs/sales/catalogs/sb>

The screenshot shows the API Manager interface with the following details:

- Sandbox catalog:** <https://mgr.think.ibm/orgs/sales/catalogs/sb>
- API Definition:** logistics 1.0.0 (REST)
- Operations:**
 - inventory 1.0.0
 - financing 1.0.0
 - logistics 1.0.0 (selected)
 - shipping.calc
 - get.stores
- Search:** Search bar with placeholder "Search".
- Operations Filter:** Operations dropdown set to "by name".
- shipping.calc Operation Details:**
 - Description:** Calculate shipping costs to a destination zip code
 - Parameters:**

zip	string, required in query
Destination zip code.	
- Security:** Security section for the API.

- __ b. In the test client, click **Try it** for the GET <https://api.think.ibm/sales/sb/logistics/shipping> operation.
- __ c. Type **98121** as the **zip** input parameter.

GET <https://api.think.ibm/sales/sb/logistics/shipping>

Examples Try it

Type Headers

Accept

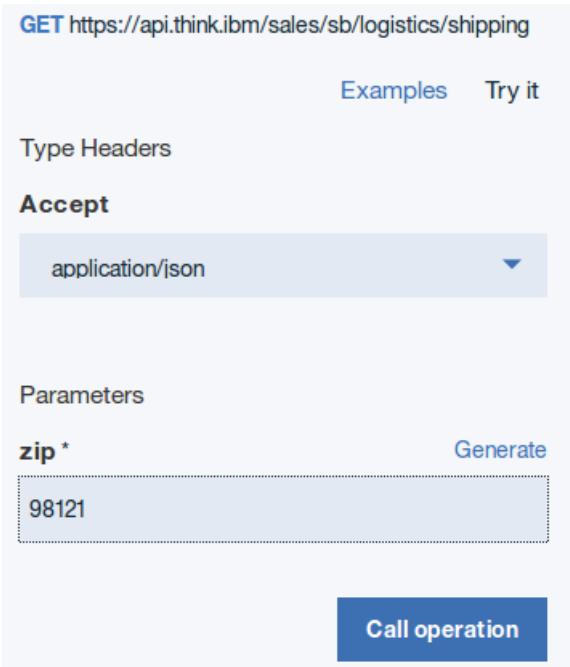
application/json ▾

Parameters

zip * Generate

98121

Call operation



Click **Call operation**.

- __ d. Confirm that the GET /shipping operation combined the results from the xyz and cek shipping companies into a single result in the response message.

Request

```
GET https://api.think.ibm/sales/sb/logistics
/shipping?zip=98121
Headers:
Content-Type: application/json
Accept: application/json
```

Response

```
Code: 200 OK
Headers:
content-type: application/json
x-global-transaction-id:
196c556559e4f74d01462c40
```

```
{
  "xyz": {
    "next_day": "39.36",
    "two_day": "26.93",
    "ground": "20.72"
  },
  "cek": {
    "next_day": "35.78",
    "two_day": "24.48",
    "ground": "18.83"
  }
}
```

- __ 9. Test the **logistics** API store location operation from the API Manager.

- __ a. Select the **get.stores** operation in the **logistics** API definition.

Sandbox catalog <https://mgr.think.ibm/orgs/sales/catalogs/sb>

The screenshot shows the API Manager interface with the following details:

- Sandbox catalog:** <https://mgr.think.ibm/orgs/sales/catalogs/sb>
- API Definition:** logistics 1.0.0
- Operations Tab:** The 'get.stores' operation is highlighted.
- Parameters Section:**
 - zip:** string, required in query

- ___ b. In the test client, click **Try it** for the `GET https://api.think.ibm/sales/sb/logistics/stores` operation.
- ___ c. Enter `98121` as the store location zip input parameter.

GET https://api.think.ibm/sales/sb/logistics/stores

Examples Try it

Type Headers

Accept

application/json ▾

Parameters

zip * Generate

98121

Call operation

- ___ d. Click **Call operation**.
- ___ e. Confirm that the stores API operation returned the map coordinates that correspond to the zip code of 98121.

Request

GET https://api.think.ibm/sales/sb/logistics/stores?zip=98121

Headers:

Content-Type: application/json

Accept: application/json

Response

Code: 200 OK

Headers:

content-type: application/json

x-global-transaction-id: 196c556559e4f9730139e292

```
{
  "maps_link": "https://www.openstreetmap.org/#map="
}
```

End of exercise

Exercise review and wrap-up

The first part of the exercise examined how to build a REST-to-SOAP bridge at the API gateway. You mapped the parameters from the financing API operation into an SOAP XML message. You set the media type to **text/xml**, and converted the SOAP response message to a JSON message.

The second part of the exercise examined the logic constructs in the assembly flow. You defined an **operation-switch** policy to create two subflows: one for the **GET /shipping** operation, and one for the **GET /stores** operation.

In the GET /stores operation, you combined the shipping rates from two separate remote services calls into one single REST API response. In the GET /stores operation, you looked up the map coordinates for a store location based on a United States postal code.

Exercise 7. Declaring an OAuth 2.0 Provider and security requirement

Estimated time

01:00

Overview

In this exercise, you examine two of the three parties in an OAuth 2.0 flow: the OAuth 2.0 Provider API and the API resource server. You define an OAuth 2.0 Provider API to authorize access and issue tokens. In the case study application, you declare an OAuth 2.0 security constraint that enforces access control with the OAuth 2.0 Provider API.

Objectives

After completing this exercise, you should be able to:

- Define an OAuth 2.0 Provider API in the API Designer web application
- Configure the client ID and client secret security definition
- Declare and enforce an OAuth 2.0 security definition with the API Designer web application

Introduction

In the case study, the **inventory** API provides a set of operations that display items in the store and reviews on items. This exercise explains how to secure the **inventory** API with OAuth 2.0 authorization. You configure an OAuth 2.0 Provider: the security server that verifies client identity and access rights to the **inventory** API operations. The OAuth 2.0 Provider also issues and manages **access tokens**: a time-limited key that allows a client access to API resources.

The OAuth 2.0 Provider is a special type of API that you define in the API Designer. When you publish the OAuth 2.0 Provider API to the API gateway, the gateway acts as the security server.

Requirements

Before you start this exercise, you must complete the **data sources**, **remote**, and **policies** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

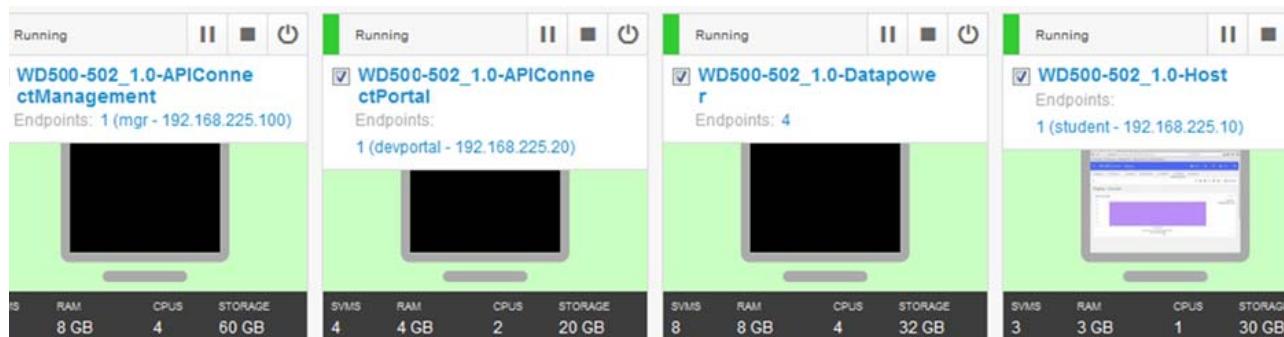
Exercise instructions

Before you begin

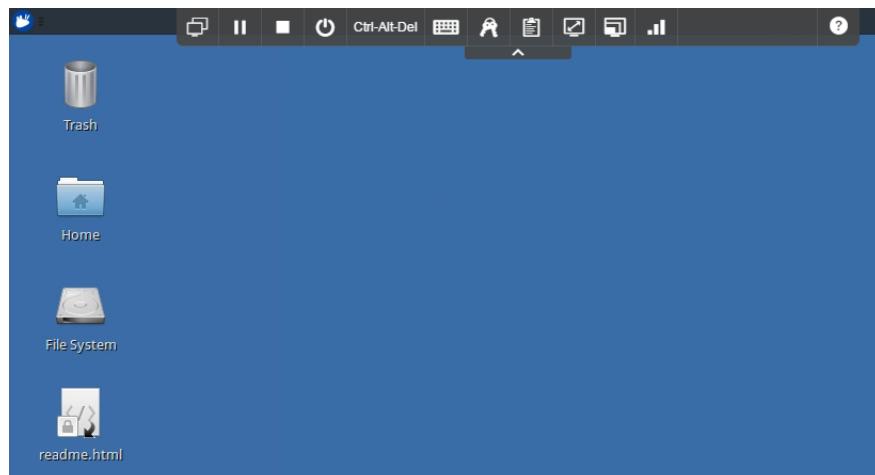
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



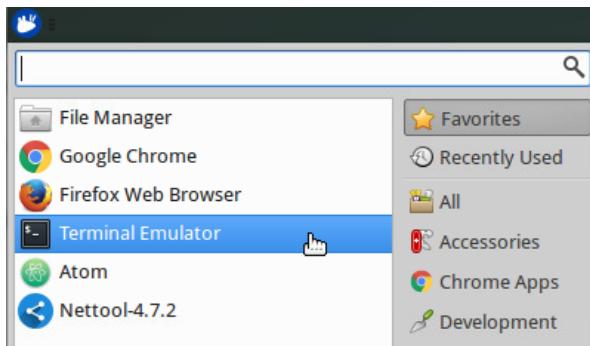
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

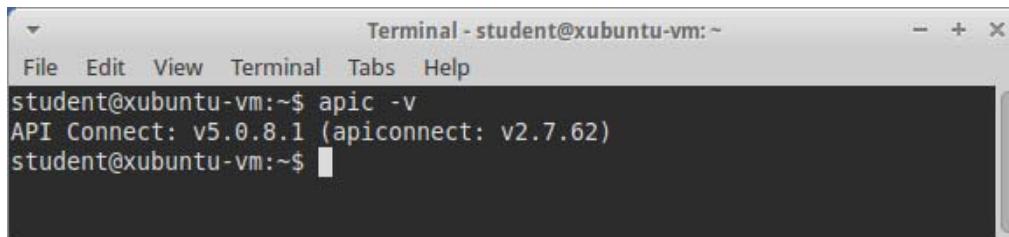
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

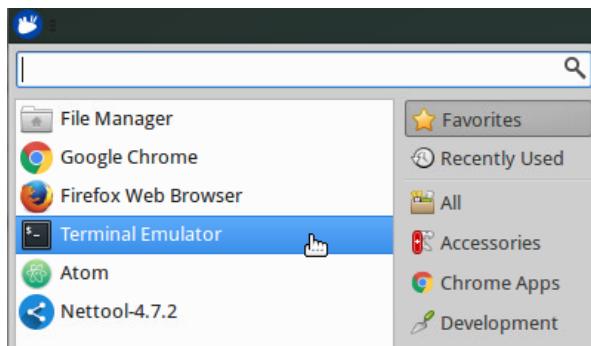
Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

7.1. Create an OAuth 2.0 Provider API

The **OAuth 2.0 Provider API** is a preset API definition: it defines the **/authorize** and **/token** API operations according to the message flow in the OAuth 2.0 specification.

In this section, define an **OAuth 2.0 Provider API** definition in API Designer. The Designer application creates a predefined API definition that includes the **/authorize** and **/token** API operations. When you deploy an OAuth 2.0 Provider API, the API gateway implements the API operations that the OAuth 2.0 message flow requires.

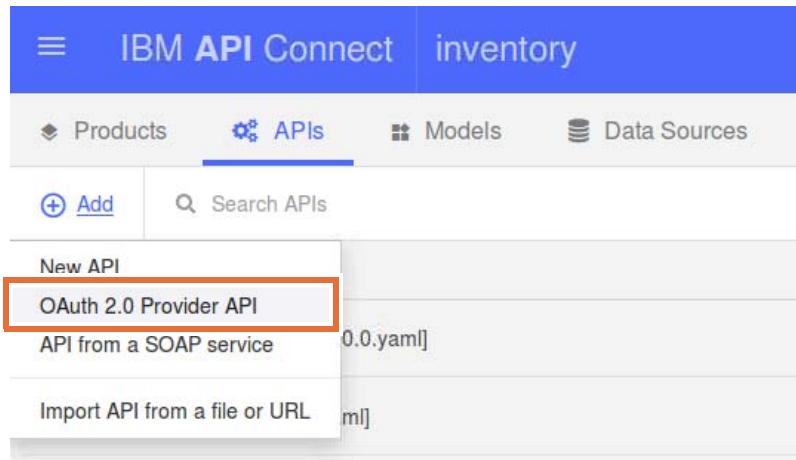
- ___ 1. Start the API Designer web application.
 - ___ a. Open a **Terminal Emulator** application window.



- ___ b. Change directory to the **inventory** application.
- ___ c. Start the **API Designer** from the command-line utility.


```
$ cd ~/inventory
$ pwd
/home/student/inventory
$ apic edit
```
- ___ 2. Add an **OAuth 2.0 Provider API** named “**oauth**”.
 - ___ a. Click the **API** tab.

- __ b. Click **Add > OAuth 2.0 Provider API** from the menu.



- __ c. Enter the following details for the OAuth provider:

- Title: **oauth-provider**
- Name: **oauth-provider**
- Base path: **/oauth20**
- Version: **1.0.0**

New OAuth 2.0 Provider API

Info	Title *
	<input type="text" value="oauth-provider"/>
	Name *
	<input type="text" value="oauth-provider"/>
	Base Path
	<input type="text" value="/oauth20"/>
	Version *
	<input type="text" value="1.0.0"/>

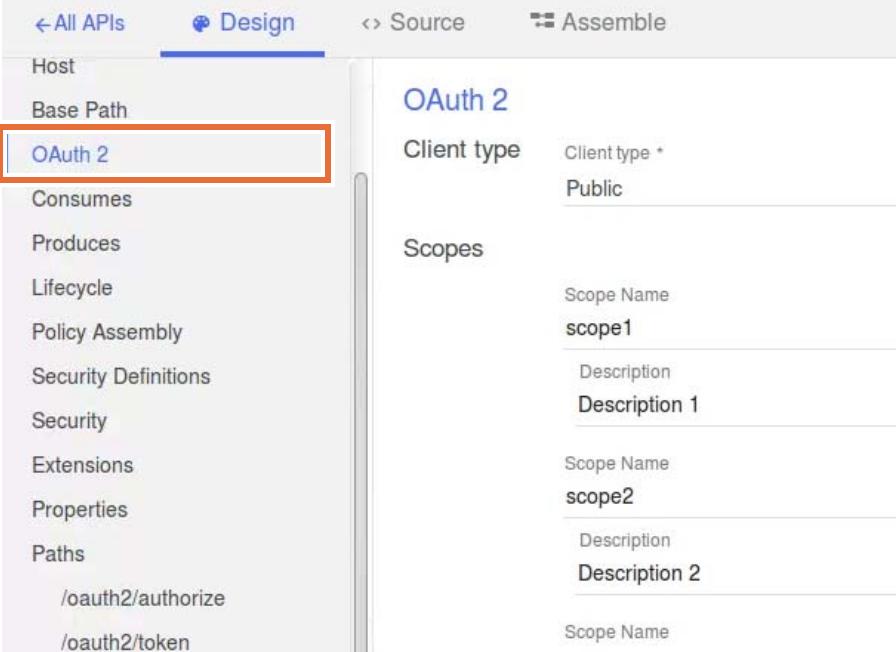
Cancel Add a product... Create API



Remember to change the **Base Path** field to: **/oauth20**

- __ d. Click **Create API**.

- ___ 3. Examine the API definition sections in the **oauth-provider** API.



The screenshot shows the 'Design' tab selected in the top navigation bar. On the left, a sidebar lists various API sections: Host, Base Path, OAuth 2 (which is highlighted with a red box), Consumes, Produces, Lifecycle, Policy Assembly, Security Definitions, Security, Extensions, Properties, Paths, /oauth2/authorize, and /oauth2/token. The main panel on the right is titled 'OAuth 2'. It contains a 'Client type' field set to 'Public' and a 'Scopes' section. The 'Scopes' section lists two entries: 'scope1' (Scope Name: scope1, Description: Description 1) and 'scope2' (Scope Name: scope2, Description: Description 2). There is also a partially visible entry for 'scope3'.

Information

When you create an **OAuth 2.0 Provider API**, the API definition includes an extra section: **OAuth 2**. This section specifies which type of OAuth 2.0 message flow to support in the API.

The **OAuth 2.0 Provider API** also defines the two API operations according to the OAuth 2.0 specification: **/oauth2/authorize**, and **/oauth2/token**.

In the first step, the **/oauth2/authorize** operation verifies the identity of the client, and determines whether that client has access rights to a certain resource. If it allows the client access to the resource, the **authorize** operation returns an **authorization code**.

In the second step, the client sends the **authorization code** to the **/oauth2/token** operation. If the authorization code is valid, the **token** operation exchanges the authorization code for an **access token**.

When the client application calls an API that is secured according to the OAuth 2.0 scheme, it sends the **access token** as part of the call.

- ___ 4. Configure the **OAuth message flow** settings in the **OAuth 2** API definition section.

- ___ a. In the navigation bar, select the **OAuth 2** section.

- ___ b. Set the **client type** as **confidential**.

The screenshot shows a configuration interface for an OAuth 2 client. At the top, it says "OAuth 2". Below that, under "Client type", there is a dropdown menu with "Confidential" selected. A blue circular icon with a white "i" is visible on the left.

The **client type** states whether the OAuth 2.0 Provider should treat the client that is calling the API as a **public** or **confidential** resource. A **public** client cannot guarantee whether it can keep the user name and password secret. A **confidential** client protects the user name and password that it sends to the API provider.

For example, a JavaScript based web application that is running in a web browser is a **public** client. A malicious user can inspect the application code, and extract the client credentials. A web application that is hosted on a server is a **confidential** client. The application stores the client credentials on the server, and the user cannot inspect the source code on the server.

- ___ c. Modify the following values for the **scope1**:

- Name: **inventory**
- Description: **Access to the inventory API**

The screenshot shows a configuration interface for OAuth 2 scopes. It has a header "Scopes" with a plus sign icon. Below it, there is a table with two rows. The first row is for "Scope Name" with the value "inventory". The second row is for "Description" with the value "Access to the inventory API". There is also a trashcan icon next to the description row.

- ___ d. Select the trashcan icon for the **scope2** and **scope 3 entries** to delete the remaining scopes.

The screenshot shows the same configuration interface for OAuth 2 scopes. The "Scopes" table now includes a third row for "Scope Name" with the value "scope2" and a "trashcan" icon next to it. The "Description" row for "inventory" still exists.

The **scope** is a user-defined property that the API provider expects from the client. In this example, you configure the **oauth-provider** API provider to secure access to the **inventory** API. When a client application requests access to an operation in the inventory API, it must state a scope that is named **inventory** in the **/authorize** operation request.

- ___ e. Set the **grant type** to support the **resource owner password credentials** grant type. Select **Password** and clear the remaining options.

Grants

- Implicit
- Password
- Application
- Access Code



Information

The OAuth 2.0 message flow has four variations:

- The **Implicit** grant redirects the client application to the **/authorize** operation to authenticate and authorize a web server to access resources on the client's behalf. The **authorize** operation runs a callback operation on the web server with the access token.
- The **Password** grant type is similar to the **implicit** grant type, but it does not redirect the user to a separate login page. The users enter their credentials in the client application. The client application sends the user name and password to the **/authorize** operation.
- The **Application** grant type uses the **client's credentials**. In this scenario, the client application itself has its own user name and password. The client application calls the **/authorize** operation and requests access to resources.
- The **Access Code** is similar to the **implicit** grant type, but it adds a call to the **/token** operation. After the user authorizes the client to access resources on the server, the **/authorize** operation returns an authorization code. The client application must exchange the authorization code for an **access token** with the **/token** operation.

- ___ f. Set the **Identity extraction** scheme to **Basic** authentication.

Identity extraction	Collect credentials using
	<u>Basic</u>



Information

The **identity extraction** scheme specifies how the **/authorize** operation finds the client credentials. In this case, the **/authorize** operation takes the **user name** and **password** values from the **HTTP basic authentication** header in the request message.

- ___ g. Set the **authentication** scheme to: **authentication URL**
- ___ h. Set the authentication URL value to: **<https://services.think.ibm:1443/auth>**

Authentication

Authenticate application users using

Authentication URL

Authentication URL

<https://services.think.ibm:1443/auth>



Information

The **authentication** scheme determines how the **/authorize** operation verifies the user name and password. In this example, the **oauth-provider** API sends an HTTP request to an external service. If that service returns an HTTP status code of **200 OK**, the **/authorize** operation successfully authenticates the client.

- ___ i. Set the authorization policy to **Authenticated**.

Authorization [Authorize application users using](#)
[Authenticated](#)



Information

With an **authorization** setting of **authenticated**, the **/authorize** operation allows any authenticated client access to any resource.

- ___ j. Enable **refresh tokens**.

Tokens	Access tokens
Time to live (seconds)	
3600	
<input checked="" type="checkbox"/> Enable refresh tokens	



Information

To prevent against replay attacks, access tokens have an expiry date. To extend the expiry date of an access token, the client application sends a **refresh token** to the **/token** service.

- ___ k. Disable the **Enable revocation** setting.



- ___ 5. Review the OAuth configuration for the **oauth-provider** API definition.

OAuth 2

Client type

Client type

Confidential

Scopes



Scope Name

inventory

Description

Access to the inventory API



Grants

 Implicit Password Application Access Code

Identity extraction

Collect credentials using

Basic

Authentication

Authenticate application users using

Authentication URL

Authentication URL

https://services.think.ibm:1443/auth

TLS Profile

Authorization

Authorize application users using

Authenticated

Tokens

Access tokens

Time to live (seconds)

3600

 Enable refresh tokens

Count

2048



Time to live (seconds)

2682000

 Enable revocation

- ___ 6. Modify the **/oauth2/authorize** path to **/authorize**.
 - ___ a. Select the **/oauth2/authorize** path in the **paths** section of the API definition.
 - ___ b. Change the path to **/authorize**.

Paths

/authorize	✖
------------	---

Path *

/authorize

Parameters

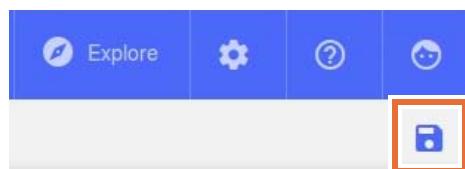
- ___ 7. Modify the **/oauth2/token** path to **/token**.
 - ___ a. Select the **/oauth2/token** path in the **paths** section of the API definition.
 - ___ b. Change the path to **/token**.

/oauth2/token	✖
---------------	---

Path *

/token

- ___ 8. Save the changes to the **oauth-provider** API definition.



- ___ 9. Add the **oauth-provider** API definition to the **inventory** API product.
 - ___ a. Click **All APIs** to return to the main page.
 - ___ b. Click **Products**.

- ___ c. Open the **inventory 1.0.0** product.

The screenshot shows the IBM API Connect interface with a blue header bar. Below it is a navigation bar with tabs: Products (selected), APIs, Models, and Data Sources. Underneath the navigation bar is a search bar labeled 'Search products'. A large table below has a single row with the title 'inventory 1.0.0 [inventory-product.yaml]'. This row is highlighted with a thick red border.

- ___ d. Select the **APIs** section.
 ___ e. Click the plus (+) button beside the APIs section.
 ___ f. In the **Select APIs** window, select **oauth-provider 1.0.0**.

Select APIs

Select the APIs to include in this product. Any APIs removed from this list will also be removed from the plans in this product.

Search APIs

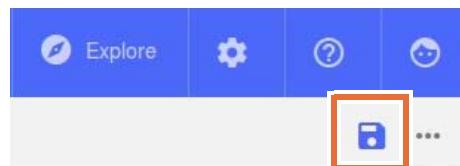
<input checked="" type="checkbox"/>	financing	1.0.0
<input checked="" type="checkbox"/>	inventory	1.0.0
<input checked="" type="checkbox"/>	logistics	1.0.0
<input checked="" type="checkbox"/>	oauth-provider	1.0.0

Cancel

Apply

- ___ g. Click **Apply**.

- ___ 10. Save the changes to the **inventory** API product.



- ___ 11. Click **All Products**.

7.2. Configure OAuth 2.0 authorization in the inventory API

In the previous section, you defined an OAuth 2.0 Provider API. The **oauth-provider** API defines two services that restrict access to API operations in the **inventory** scope. The authorization service checks the user name and password encoded in the HTTP Basic authentication header.

In this section, you specify a **security definition** in the **inventory** API definition. The OAuth 2.0 security definition specifies the grant type and scope that the **oauth-provider** API expects.

- ___ 1. Switch to the **inventory** API definition.
 - ___ a. Click **APIs** to list all the APIs.
 - ___ b. Open the **inventory** API definition.

The screenshot shows the API Manager's 'APIs' tab. At the top, there are tabs for 'Products', 'APIs' (which is selected), 'Models', and 'Data Sources'. Below the tabs is a search bar labeled 'Search APIs'. A large list of APIs is displayed, each with a title and a link to its YAML file. The 'inventory' API is highlighted with a red box and has a cursor icon pointing at it. Other APIs listed include 'financing', 'logistics', and 'oauth-provider'.

Title	Version	YAML File
financing	1.0.0	[financing_1.0.0.yaml]
inventory	1.0.0	[inventory.yaml]
logistics	1.0.0	[logistics_1.0.0.yaml]
oauth-provider	1.0.0	[oauth-provider_1.0.0.yaml]

- ___ 2. Verify that the **Base Path** is set to **/inventory**.
 - ___ a. Select the **Base Path** category in the API editor.
 - ___ b. Confirm that the Base Path is set to **/inventory**.

The screenshot shows the API editor's configuration panel. On the left, there is a sidebar with categories: 'Info', 'Schemes', 'Host', and 'Base Path' (which is selected). On the right, there are two main sections: 'Base Path' and 'Base Path'. The 'Base Path' section contains a text input field with the value '/inventory'.

Base Path	Base Path
/inventory	/inventory

- ___ 3. Create an **OAuth security definition**.
 - ___ a. Navigate to the **security definitions** section.

- ___ b. Click the **plus (+)** icon and select **OAuth** from the menu.



- ___ c. Scroll down the page to the **oauth-1** security definition.
 ___ d. Enter the OAuth security definition details to match the **oauth-provider** API.
- Name: **oauth**
 - Description: **OAuth authorization settings for inventory API**
 - Flow: **password**
 - Token URL: **<https://api.think.ibm/sales/sb/oauth20/token>**

 A screenshot of a configuration form for the "oauth (OAuth)" security definition. The form fields are:

- Name ***: oauth
- Description**: OAuth authorization settings for inventory API
- Flow**: Password
- Token URL**: https://api.think.ibm/sales/sb/oauth20/token

- ___ 4. Add the **inventory** scope to the **oauth** security definition.
 ___ a. In the **oauth** security definition, click the **plus (+)** icon in the **Scopes** section.



- ___ b. Enter the following details for the OAuth security scope:
- Scope name: **inventory**
 - Description: **Access to inventory API operations**

Scope Name	inventory
Description	Access to inventory API operations

- ___ 5. Apply the security definition to the operations in the **inventory** API.
- ___ a. Scroll down to the **Security** section of the **inventory** API definition.
- ___ b. Select the **oauth (OAuth)** security definition to apply it to all API operations.

Security

Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.

Option 1 oauth (OAuth) inventory
 clientIdHeader (API Key)
 clientSecretHeader (API Key)

Information

The **oauth** security definition specifies how client applications invoke operations in the **inventory** API definition. Specifically, a client application must fulfill the requirements that are set in the **oauth** security definition to call an **inventory** API operation.

In this example, the client application must send an API operation request with the **inventory** scope. The **inventory** API expects a **password** OAuth 2.0 grant type. To access the API, clients retrieve an access token from the **token service** from the **oauth-provider** API, at <https://api.think.ibm/sales/sb/oauth20/token>.

- ___ 6. Review the security settings at the API operation level.
- ___ a. Navigate to the **Paths** section of the **inventory** API definition.
- ___ b. Select the **/items** path.

- __ c. Expand the **GET /items** API operation.

The screenshot shows the API definition interface. On the left, a sidebar lists various paths under 'Paths', with '/items' selected. On the right, the expanded '/items' operation is shown with three methods: PUT, PATCH, and GET. The GET method is highlighted with a blue background. Below the methods, there is a summary: 'Find all instances of the model matched by filter from the data source.' and an 'Operation ID' field containing 'item.find'. A 'Tags' section shows 'item x Add Tag'.

- __ d. Scroll down to the **Security** section of the **GET /items** API operation.
 __ e. To review the operation-level security options, clear **Use API security definitions**.
 __ f. Review the security settings.

The screenshot shows the 'Security' section for the GET /items operation. It includes a checkbox labeled 'Use API security definitions' and a list of security options: 'oauth (OAuth)', 'clientIdHeader (API Key)', and 'clientSecretHeader (API Key)'. Each option has an associated checkbox and a delete icon.

- __ g. Select **Use API security definitions** after you reviewed the options.

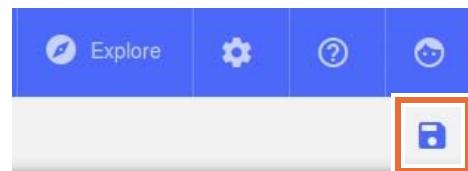
The screenshot shows the 'Security' section for the GET /items operation, identical to the previous one, but with the 'Use API security definitions' checkbox now checked.

Information

By default, every operation in the API definition inherits the security configuration from the **Security** section of the document. If you want to override these settings, you can change the list of enabled security definitions in the **Security** section of a specific API operation.

Leave the **Use API security definitions** to follow the default settings for the entire API.

- __ 7. Save the changes to the inventory API definition.



End of exercise

Exercise review and wrap-up

The first part of the exercise explained how to create and publish an OAuth 2.0 Provider: a set of API operations that authorize access to protected resources. You create the **oauth-provider** API definition that defined two operations: the **authorize** and **token** services.

The second part of the exercise covered how to secure API resources with an OAuth 2.0 message flow. You defined an OAuth 2.0 security requirement on all operations in the **inventory** API.

Exercise 8. Deploying an API implementation to a container runtime

Estimated time

00:45

Overview

In this exercise, you deploy and publish a LoopBack API application to a swarm of Docker container runtime environments. A swarm manager administers a cluster of Docker containers. At run time, the swarm manager routes API requests to a Docker cluster member that runs an instance of your API and its runtime dependencies.

Objectives

After completing this exercise, you should be able to:

- Test a local copy of a LoopBack API application
- Examine the Dockerfile that is used to deploy a Loopback API
- Build a Docker image by using the docker command with the Dockerfile
- Verify that the API runs on the Docker image
- Push the image to a local registry
- Initialize the Docker swarm
- Use the Docker stack command to deploy a swarm service
- Test that the API runs on the swarm service

Introduction

You defined, developed, and secured the **inventory** API with the API Connect Toolkit. You have two options to deploy the application:

- Deploy the API application to the IBM Cloud
- Deploy the API application to a containerized runtime in your own enterprise network

In this exercise, you learn how to deploy the API application to a Docker containerized runtime. For this exercise, the Docker cluster members are hosted in the Xubuntu host. In a real-world configuration, the cluster members are hosted on different hosts in your network.

**Note**

You can deploy the LoopBack API application to any server environment that hosts Node.js applications, such as a Node.js Cloud Foundry application buildpack or a Docker container. However, you cannot control the API applications from the API Management server.

Requirements

Before you start this exercise, you must complete the **data sources**, **remote**, **policies**, and **authorization** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

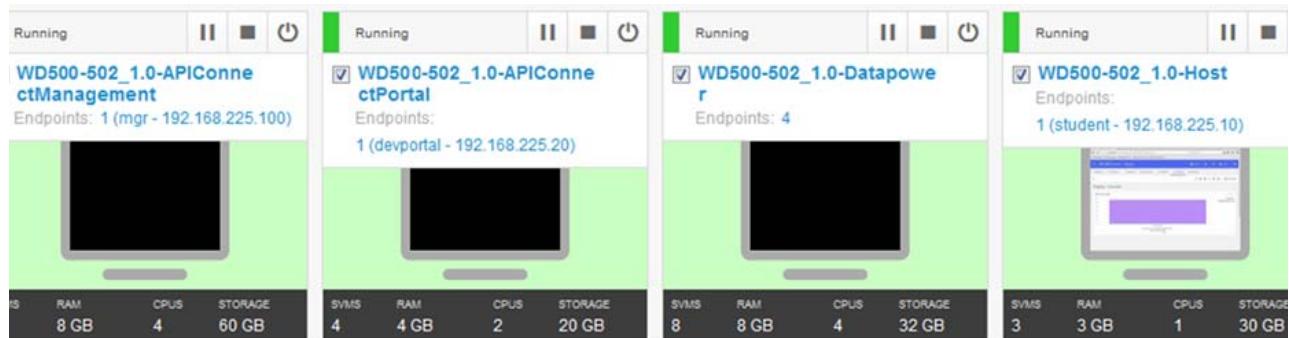
Exercise instructions

Before you begin

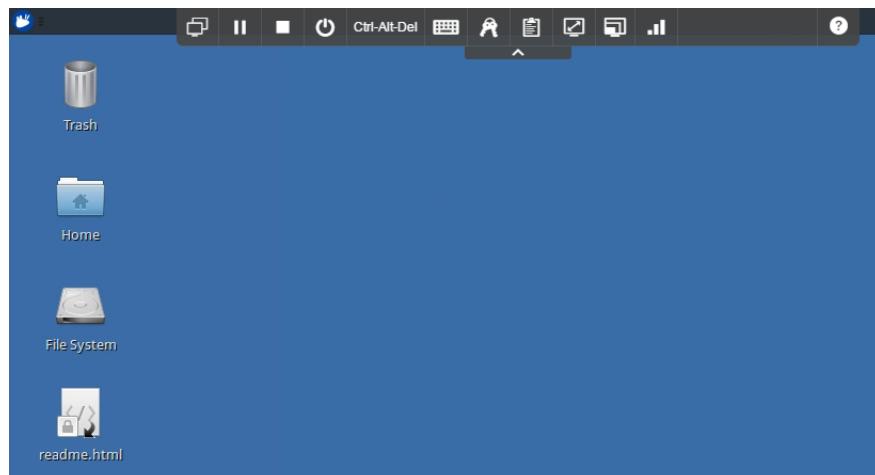
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



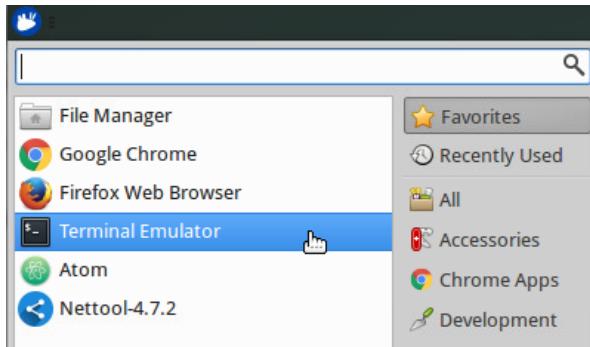
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

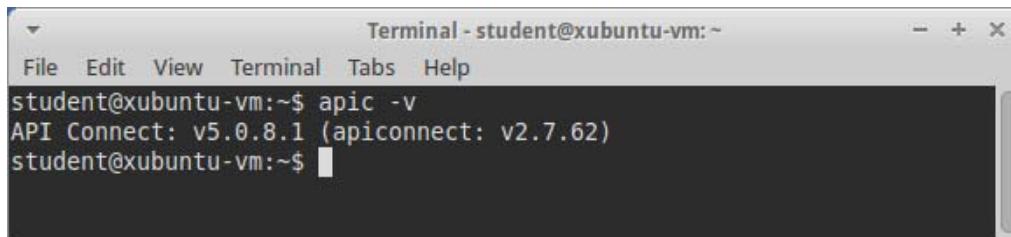
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

-
- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

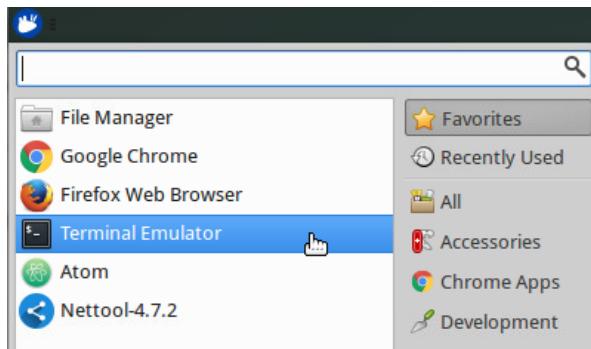
Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

8.1. Test a local copy of the LoopBack API application

In a previous exercise, you created the **inventory** LoopBack API application. You designed and implemented two model objects: **items** and **reviews**. The LoopBack framework created API paths and operations that map to create, retrieve, update, and delete functions on **items** and **reviews**.

Before you deploy and publish the **inventory** LoopBack API application, make sure that the application runs correctly on your workstation.

- ___ 1. Start the **inventory** application.
- ___ a. Open the **Terminal Emulator** application.



- ___ b. Change directory to the **inventory** application.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

- ___ c. Start **inventory** as a Node application.

```
$ npm start
> inventory@1.0.0 start /home/student/inventory
> node .
Web server listening at: http://0.0.0.0:3000
```

- ___ d. Wait until the web server is started and listening.



Information

By default, the LoopBack application listens on **port 3000** when you start the application locally on your workstation. You can override this setting in the **server/config.json** configuration file.

- ___ 2. Test the **GET /inventory/items** API operation with the cURL utility.
- ___ a. In the **Terminal Emulator** application, click **File > Open Tab**.

- ___ b. Make an HTTP GET request to the `/inventory/items` path.

```
$ curl -k -I http://0.0.0.0:3000/inventory/items
HTTP/1.1 200 OK
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
X-XSS-Protection: 1; mode=block
X-Frame-Options: DENY
X-Download-Options: noopen
X-Content-Type-Options: nosniff
Content-Type: application/json; charset=utf-8
Content-Length: 10476
ETag: W/"28ec-0eqf3uQDRW98EMjbal7CbQ"
Date: Mon, 28 Nov 2016 18:49:17 GMT
Connection: keep-alive
```



Information

The **cURL** application is a highly customizable command-line utility that makes HTTP and HTTPS requests to web servers. In this step, you called the `/inventory/items` API path on the local copy of the **inventory** LoopBack application.

The `-k` option instructs cURL to ignore security warnings from TLS certificate errors. By default, the cURL application does not accept self-signed security certificates. You ignore this warning in development and testing.

The `-I` or `--head` option instructs cURL to display the HTTP response message header, but not the message body. You can omit this parameter to see the list of inventory items from the service.

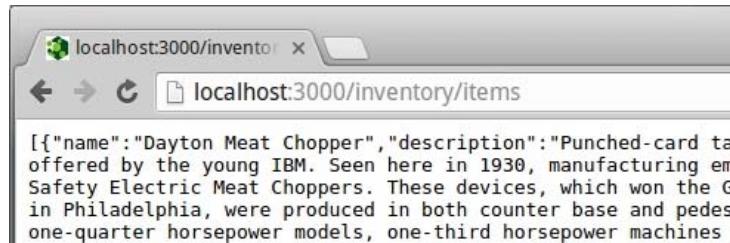


Attention

If you **did not receive** an HTTP status code of **200 OK**, the **inventory** LoopBack API is not working correctly. Make sure that you correct any errors before you continue with this exercise.

- ___ 3. Review the contents of the `/inventory/items` API operation.
- ___ a. Open a web browser.
- ___ b. Enter `http://localhost:3000/inventory/items` in the address bar.

- ___ c. Confirm that a list of inventory items appears in the web page, starting with the **Dayton Meat Chopper**.



[{"name": "Dayton Meat Chopper", "description": "Punched-card tabulating machines and time clocks were not the only products offered by the young IBM. Seen here in 1930, manufacturing employees of IBM's Dayton Scale Company are assembling Dayton Safety Electric Meat Choppers.

- ___ 4. Stop the **inventory** application.
- ___ a. Switch to the **Terminal Emulator** tab with the LoopBack API application.
- ___ b. Press Ctrl+C to stop the Node application.

8.2. Set up the Docker image configuration

With the **Docker swarm** technology, you can publish Loopback applications to a distributed, clustered containerized environment.

- A Docker container **image** is an executable package that includes everything that is needed to run it: code, runtime, system libraries, and settings.

If you decide to use a Docker container to host your LoopBack application, you must use Docker software to build the Docker image. For the purposes of this exercise, Docker and docker-compose software are preinstalled on the workstation.

1. Build the Docker image.

- a. Open a **Terminal Emulator** application window.
 b. Ensure that you are in the inventory directory.

```
$ cd ~/inventory
```

- c. Copy the Dockerfile from the `~/lab_files/containers` folder.

```
$ cp ~/lab_files/containers/Dockerfile ./
```

- d. Review the Dockerfile in the inventory folder.

```
$ cat Dockerfile
```

```
# Build with:
```

```
# docker build -t apic-inventory-image .
```

```
# Create a small alpine linux distribution as the base image
```

```
FROM alpine:3.6
```

```
RUN apk add --update nodejs nodejs-npm && npm install npm@4.4.4 -g
```

```
RUN which node; node -v
```

```
WORKDIR /inventory
```

```
# Copy the files in the host current dir to the Docker WORKDIR
```

```
ADD . /inventory
```

```
RUN pwd;ls
```

```
# Make port available outside the container
```

```
EXPOSE 3000
```

```
#CMD to start
```

```
CMD [ "npm", "start" ]
```

```
# Run the image after the build with the command:
```

```
# docker run --add-host mysql.think.ibm:192.168.225.10 --add-host mongo.think.ibm:192.168.225.10 -p 3000:3000 apic-inventory-image
```

- __ e. Build the Docker image by using the Dockerfile.

```
$ docker build -t apic-inventory-image .
```

```
Sending build context to Docker daemon 97.5 MB
Step 1/8 : FROM alpine:3.6
--> 76da55c8019d
Step 2/8 : RUN apk add --update nodejs nodejs-npm && npm install npm@4.4.4
-g
--> Running in 129df0759f46
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.6/main/x86_64/APKINDEX.tar.gz
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.g
z
(1/9) Installing ca-certificates (20161130-r2)
...
Step 7/8 : EXPOSE 3000
--> Running in 9cadad965d66
--> 09de2c5ce14f
Removing intermediate container 9cadad965d66
Step 8/8 : CMD npm start
--> Running in 77c82583fcc8
--> e743a15b9563
Removing intermediate container 77c82583fcc8
Successfully built e743a15b9563
```



Note

The Docker build command includes a period (.) at the end for the path that the build command uses. The path option specifies that all the files in the local directory are sent to the Docker daemon when the build command is run.



Information

You might receive an error when running the build command, such as

```
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.g
z
ERROR: http://dl-cdn.alpinelinux.org/alpine/v3.6/community: temporary
error (try again later)
```

If so, create an entry in /etc/default/docker:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --dns 192.168.225.10"
```

Restart the Docker daemon with the command:

```
sudo service docker restart
```

Rerun the docker build command.

- ___ f. Display the list of Docker images.

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
apic-inventory-image  latest    e743a15b9563  7 minutes
ago                 136 MB
alpine              3.6      76da55c8019d  4 weeks ago
3.97 MB
```



Note

In the lab environment, the **apic-inventory-image** listens to commands on port **3000**.

- ___ 2. Run the application on the Docker image.

- ___ a. In the terminal emulator, type:

```
$ docker run --add-host mysql.think.ibm:192.168.225.10 --add-host
mongo.think.ibm:192.168.225.10 -p 3000:3000 apic-inventory-image
```

```
> inventory@1.0.0 start /inventory
> node .
```

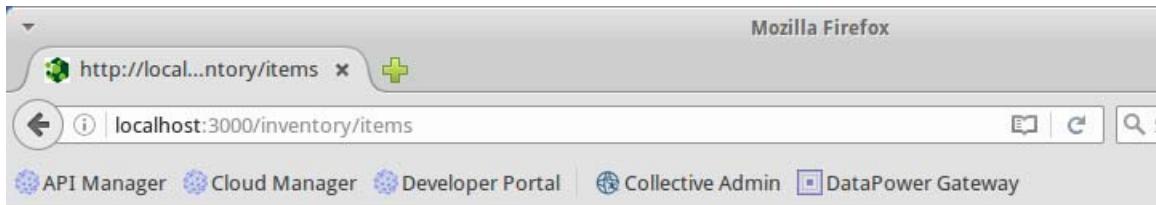
Web server listening at: http://localhost:3000

- ___ 3. Review the contents of the **/inventory/items** API operation.

- ___ a. Open a web browser.

- ___ b. Enter **http://localhost:3000/inventory/items** in the address bar.

- ___ c. Confirm that a list of inventory items appears in the web page, starting with the **Dayton Meat Chopper**.



```
[{"name": "Dayton Meat Chopper", "description": "Punched-card tabulating machines and time clocks were made by IBM. Seen here in 1930, manufacturing employees of IBM's Dayton Scale Company are assembling Dayton Scale models which won the Gold Medal at the 1926 Sesquicentennial International Exposition in Philadelphia, were produced in styles (5000 and 6000 series, respectively). They included one-quarter horsepower models, one-third horsepower (Models 6213F), one-half horsepower types (Styles 5117, 6117F and 6217F) and one horsepower choppers (Styles 5118, 6118F and 6218F) ranging in price from $180 to $375. Three years after this photograph was taken, the Dayton Scale Company became an IBM Manufacturing Company in 1934.", "img": "images/items/meat-chopper.jpg", "img_alt": "Dayton Meat Chopper"}
```

- ___ 4. Stop the running the Docker container.

- ___ a. Open another terminal emulator window or tab.
___ b. In the terminal emulator, type:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
65bce9862bd9	apic-inventory-image	"npm start"	17 minutes ago
	Up 17 minutes	0.0.0.0:3000->3000/tcp	compassionate_knuth

- ___ c. In the terminal emulator, type:

```
$ docker stop <container ID>
```



Note

If only one Docker container exists, you can use the first few characters of the container ID as a shorthand notation.

8.3. Load the inventory image to a local Docker registry

The Docker Registry 2.0 implementation is used for storing and distributing Docker images. Instead of using a public registry, such as Docker Hub, a local Docker registry is preinstalled on the Xubuntu student workstation. The local registry was installed with the `docker pull registry` command.

- ___ 1. Start the local registry.

- ___ a. In the **terminal emulator** application, type:

```
$ docker run -d -p 5001:5000 --restart always --name registry registry:2
```

- ___ b. In the terminal emulator, type:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            NAMES
STATUS              PORTS              NAMES
ac0a3432a66c        registry:2        "/entrypoint.sh /e..."   18 seconds ago      0.0.0.0:5001->5000/tcp   registry
```

- ___ 2. Tag the image that you created earlier so that it points to the local registry.

- ___ a. In the **terminal emulator** application, type:

```
$ docker tag apic-inventory-image localhost:5001/my-inventory-image
```

- ___ b. Push the image to the local registry:

```
$ docker push localhost:5001/my-inventory-image
```

The push refers to a repository [localhost:5001/my-inventory-image]

bf6d866db1ce: Pushed

fa3779030202: Pushed

4868daef6aa0: Pushed

5bef08742407: Pushed

latest: digest:

sha256:39c4f22e3f28837f60c85c685f2e4cec9113f6782616f6ac03a2825230d3d882

size: 1158

- ___ c. Pull the image back from the local registry:

```
$ docker pull localhost:5001/my-inventory-image
```

Using default tag: latest

latest: Pulling from my-inventory-image

Digest:

sha256:39c4f22e3f28837f60c85c685f2e4cec9113f6782616f6ac03a2825230d3d882

Status: Image is up to date for localhost:5001/my-inventory-image:latest

- ___ d. Display the list of Docker images:

```
$ docker images
```

localhost:5001/my-inventory-image	latest
-----------------------------------	--------

e743a15b9563	4 hours ago	136 MB
--------------	-------------	--------

8.4. Deploy the LoopBack application on the Docker image to a swarm

In this step, you use docker-compose to deploy your inventory image to a Docker swarm.

- ___ 1. Review the `docker-compose.yml` file.
 - ___ a. In the **terminal emulator** application, type:

```
$ ping mgr.think.ibm
PING mgr.think.ibm (192.168.225.100) 56(84) bytes of data
64 bytes from mgr.think.ibm (192.168.225.100) icmp_seq=1 ttl=64
```

- ___ 2. Open a **terminal** window to the **inventory** application directory.
 - ___ a. Open a **Terminal Emulator** application window.
 - ___ b. Navigate to the **inventory** directory.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

8.5. Deploy the LoopBack application on the Docker image to a swarm

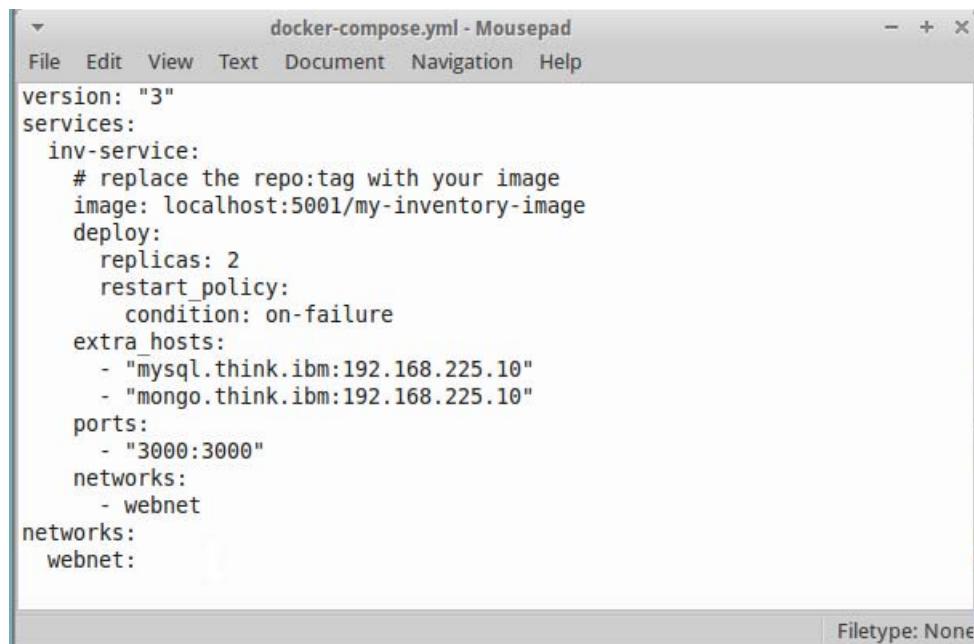
In this step, you use docker-compose to deploy your inventory image to a Docker swarm.

- 1. Copy the `docker-compose.yml` file from the `lab_files` directory to the inventory directory.
 - a. In the **terminal emulator** application, type:

```
$ cp ~/lab_files/containers/docker-compose.yml ./
```

- 2. Review the `docker-compose.yml` file.
 - a. In the **terminal emulator** application, type:

```
$ mousepad docker-compose.yml
```
 - b. Ensure that the name of your image is in the `image:` line in the file.



The screenshot shows a window titled "docker-compose.yml - Mousepad" containing a YAML configuration file for Docker Compose. The file defines a service named "inv-service" with the following specifications:

```
version: "3"
services:
  inv-service:
    # replace the repo:tag with your image
    image: localhost:5001/my-inventory-image
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failure
    extra_hosts:
      - "mysql.think.ibm:192.168.225.10"
      - "mongo.think.ibm:192.168.225.10"
    ports:
      - "3000:3000"
    networks:
      - webnet
networks:
  webnet:
```

The status bar at the bottom right of the editor window indicates "Filetype: None".

- c. Close the editor.

___ 3. Initialize the Docker swarm.

```
$ docker swarm init --advertise-addr 192.168.225.10
Swarm initialized: current node (7m8qit3gff8ovwd7pzoepy83c) is now a
manager.
To add a worker to this swarm, run the following command:
  docker swarm join \
    --token
SWMTKN-1-53ejlky1dk7cucw0zt1xhonvoossvc99k1nq4g2n3ml1v6qseuj-0xs5ctx9nu173
gg10qg4d7hly \
  192.168.225.10:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

___ 4. Deploy the service to a Docker stack by using the `docker-compose.yml` file.

```
$ docker stack deploy -c docker-compose.yml mystack
Creating network mystack_webnet
Creating service mystack_inv-service
```

___ a. List the Docker services.

```
$ docker service ls
ID                  NAME                MODE            REPLICAS  IMAGE
ns2z64c7a16m      mystack_inv-service  replicated   2/2
localhost:5001/my-inventory-image:latest
```

___ b. Inspect the service.

```
$ docker service inspect --pretty <service_ID>
ID:      ns2z64c7a16mjx2mo0bmyal7m
Name:    mystack_inv-service
Labels:
  com.docker.stack.namespace=mystack
Service Mode:Replicated
  Replicas:2
Placement:
ContainerSpec:
  Image:
    localhost:5001/my-inventory-image:latest@sha256:39c4f22e3f28837f60c85c685
    f2e4cec9113f6782616f6ac03a2825230d3d882
  Resources:
    Networks: j3jcud6w7kdpxiyyb325r12s1
    Endpoint Mode:vip
  Ports:
    PublishedPort 3000
      Protocol = tcp
      TargetPort = 3000
```

8.6. Test the LoopBack application on the Docker swarm

In this step, you test your inventory image that is running on a Docker swarm.

- ___ 1. Verify the local workstation host name.

- ___ a. In the **terminal emulator** application, verify that the Xubuntu host workstation can reach the API Management server.

```
$ ping inventory.think.ibm
PING smtp.think.ibm (192.168.225.10) 56(84) bytes of data.
64 bytes from smtp.think.ibm (192.168.225.10): icmp_seq=1 ttl=64
time=0.057 ms
...

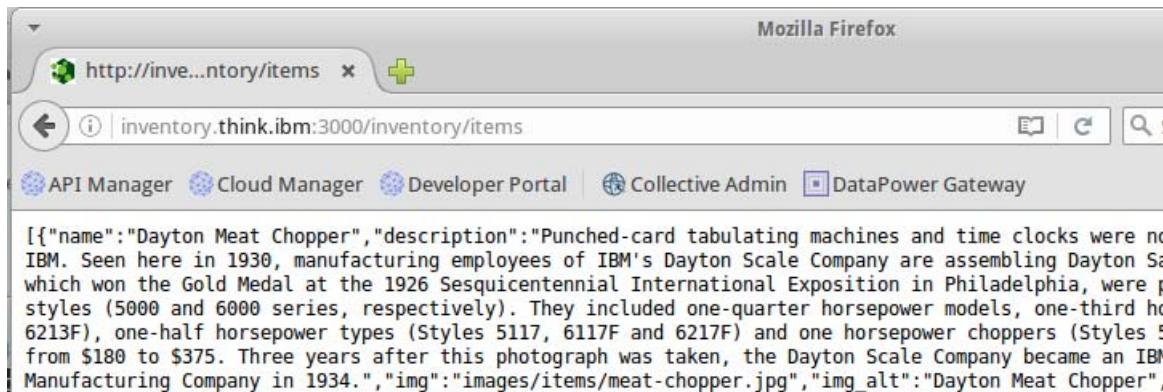
```

- ___ 2. Open the Firefox browser.

- ___ a. In the **Firefox** browser address, type:

`http://inventory.think.ibm:3000/inventory/items`

- ___ b. The browser displays the application that is running on one of the instances of the Docker swarm.



- ___ 3. Display the details of the running service.

- ___ a. In the **terminal emulator** browser address, type:

```
$ docker service ps mystack_inv-service
ID                  NAME                IMAGE
NODE              DESIRED STATE   CURRENT STATE          ERROR    PORTS
zxbowaktqts3  mystack_inv-service.1
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 24 minutes ago
1z3cmnb18cje  mystack_inv-service.2
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 24 minutes ago
```

- __ b. Scale the service that is running on the swarm:

```
$ docker service scale mystack_inv-service=5
mystack_inv-service scaled to 5
```

- __ c. Display the updated service details:

```
$ docker service ps mystack_inv-service
ID          NAME          IMAGE
NODE        DESIRED STATE  CURRENT STATE      ERROR  PORTS
zxbowaktqts3  mystack_inv-service.1
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 53 minutes ago
1z3cmnb18cje  mystack_inv-service.2
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 53 minutes ago
wavh8b8z8pik  mystack_inv-service.3
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 3 minutes ago
qw8ppgd01t38  mystack_inv-service.4
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 3 minutes ago
yd1hjonwx0ic  mystack_inv-service.5
localhost:5001/my-inventory-image:latest  xubuntu-vm  Running
Running 3 minutes ago
```

8.7. Delete the service that is running on the Docker swarm

In this step, you delete the service from the Docker swarm and leave the swarm.

- 1. Remove the service from the swarm.
 - a. In the **terminal emulator** application, type:

```
$ docker service rm mystack_inv-service  
mystack_inv-service
```

- 2. Stop the local Docker registry.
 - a. In the **terminal emulator** application, type:

```
$ docker stop registry  
registry
```

You rerun the Loopback application on Docker in a later exercise.

End of exercise

Exercise review and wrap-up

The first part of the exercise reviews the inventory API application that you built in the previous exercise. You confirmed that the LoopBack application works in your workstation environment.

In the second part of the exercise, you set up the Loopback application on a Docker image by using a Dockerfile. Next, you loaded the inventory image to a local Docker registry.

In the last part of the exercise, you initialized a Docker swarm and deployed a service to the swarm.

Exercise 9. Defining and publishing an API product

Estimated time

00:45

Overview

This exercise examines how to publish APIs with plans and products. You create a product and a plan, and deploy the product to the API Manager in the API Designer.

Objectives

After completing this exercise, you should be able to:

- Modify the invoke URL for the inventory application to route to the Docker image
- Create a product in the API Designer
- Modify the product properties and add the APIs to the product
- Define an API plan
- Define a publish target
- Stage a product to a catalog

Introduction

In a previous exercise, you published the implementation of an API: the **inventory** LoopBack application. In this exercise, you package and publish the **financing**, **logistics**, **inventory**, and **oauth-provider** API definitions to the API Management server.

- An **API definition** lists the paths and operations in an API. For each operation, the API definition specifies the possible request, response, and fault message. API definitions also contain metadata about an API, including version, description, security configuration, environment properties, and message processing policies.
 - An **OpenAPI** definition file is the design-time artifact that represents the API definition.
- A **product** combines one or more **API definitions** into a bundle. The product itself includes metadata, version, and licensing information.
 - A Product document is the design-time artifact that represents an API product. Unlike the OpenAPI definition file, this document is not part of a standard.
- A **plan** is a contract between the API provider and the API consumer. It specifies the rate of API calls over a defined time period. A plan is defined in a section of a product.

When you publish a product, you make the API definitions in the product available for use. As part of the publish process, the API gateway configures network endpoints according to the API definition. The gateway enforces security constraints and processes messages according to policies in the API definition.

The product, API definition, and plans also appear in the Developer Portal. Application developers who want to use or consume APIs can retrieve the API definition from the portal.

Requirements

Before you start this exercise, you must complete the **data sources**, **remote**, **policies**, **authorization**, and **container** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

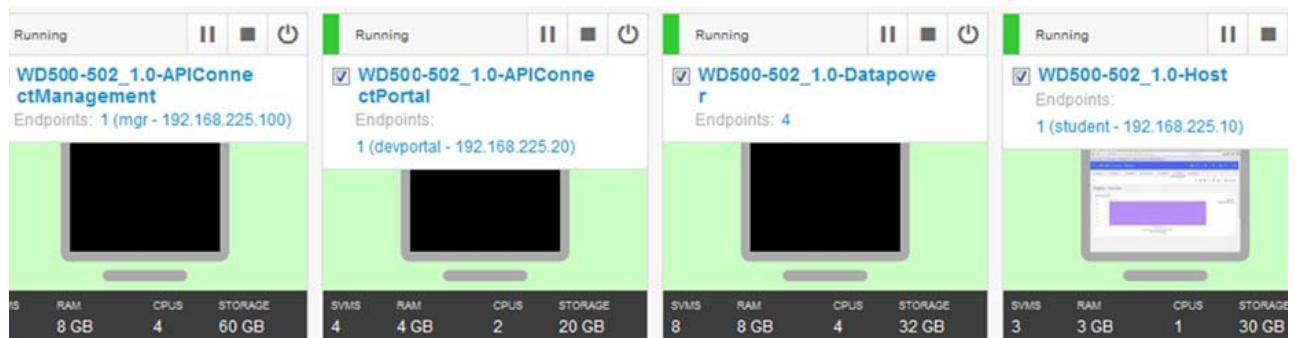
Exercise instructions

Before you begin

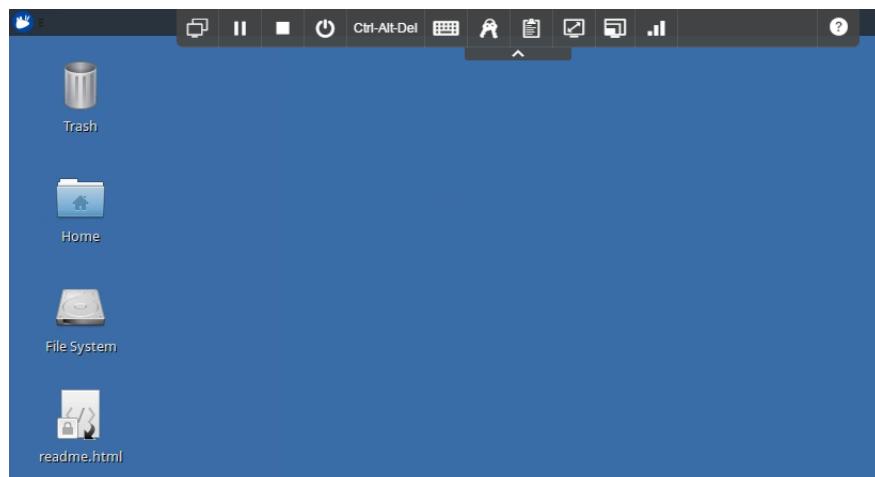
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



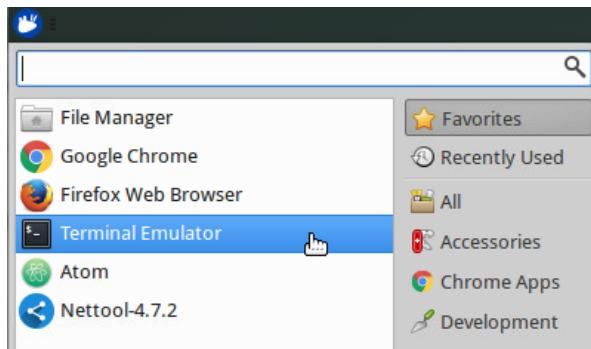
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

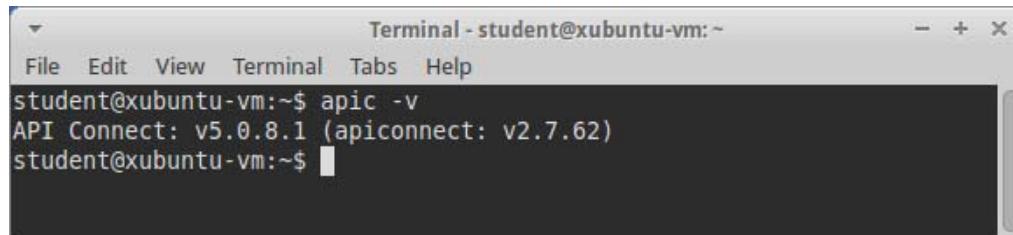
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

9.1. Change the invoke URL so that it routes to the Docker image and port number

In this section, you change the value of the invoke URL for the inventory API so that it routes to the Docker image and port number. Save the application server as a property in the API definition.

- 1. Open the API Designer application.
 - a. Open a **Terminal Emulator** application window.
 - b. Change directory to the **inventory** application directory.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```
 - c. Start the API Designer application.

```
$ apic edit
```
- 2. Open the **inventory** API definition.
 - a. Switch to the **API** tab in the API Designer.
 - b. Click the **inventory** API.
- 3. Create an API property named **app-server**.
 - a. Click **Properties** in the Design view.
 - b. Click the **plus** (+) sign to create an API property.
 - c. Type **app-server** in the property name.

__ d. Enter the following values:

- Description: **Host name and port number**
- Default: **http://inventory.think.ibm:3000**

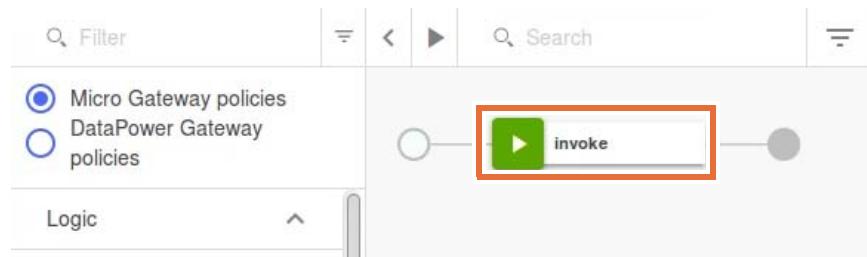
The screenshot shows the 'app-server' property configuration in the catalog. The 'Property Name *' field contains 'app-server'. The 'Description' field contains 'Host name and port number'. A button labeled 'Add value +' is present. Below it, a table shows a single entry: 'Default' under Catalog and 'http://inventory.think.ibm:3000' under Value. The 'Value' field is highlighted with a red box.

Catalog	Value
Default	http://inventory.think.ibm:3000

- __ 4. Save the changes to the **inventory** API definition.
- __ 5. Open the **Assemble** tab in the **inventory** API definition editor.
- __ a. In the API definition editor for the **inventory** API, click **Assemble**.



- __ 6. Examine the default message processing policies.
- __ a. Select the **invoke** policy in the assemble view.



- __ b. Review the **invoke URL** in the **Properties** view.



Information

The default message processing policy is to **call the API application that implements the API operation**. The API application endpoint consists of three parts:

- The **runtime-url** is the **host name** of the application server that listens to API operation requests.
- The **request.path** is the **API path**, for example: `/inventory/items`
- The **request.search** is the query parameters that limit the results.

In the following steps, you modify the message processing policy to call the Docker image URL and port number for the deployed **inventory** API application.

- __ 7. Change the policy type to **DataPower Gateway policies**.



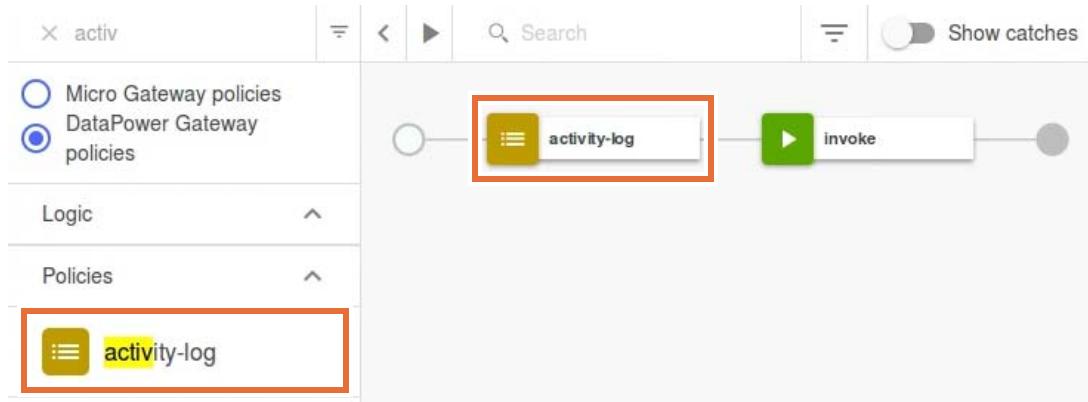
- ___ 8. Change the **invoke** operation to send the HTTP request to the Docker image, at the network address that is stored in the **app-server** property.
 - ___ a. In the **properties** view for the Invoke policy, change the URL field to:
`$(app-server)${request.path}${request.search}`



Important

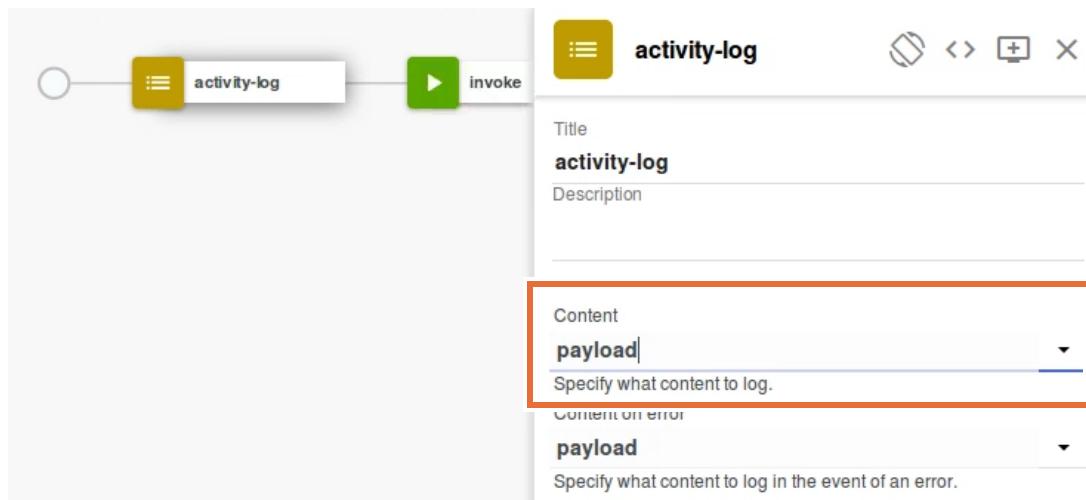
Make sure that you change the **invoke** policy to call the **\$(app-server)** network address.

- ___ 9. Close the invoke properties view.
- ___ 10. Capture the API operation request message in an **activity log**.
 - ___ a. Search for the **activity-log** policy in the palette.
 - ___ b. Drag the **activity-log** policy and place it to the left of the **invoke** policy.



- ___ c. Select the **activity-log** to open the properties view.

- __ d. Select **payload** in the **Content** field.

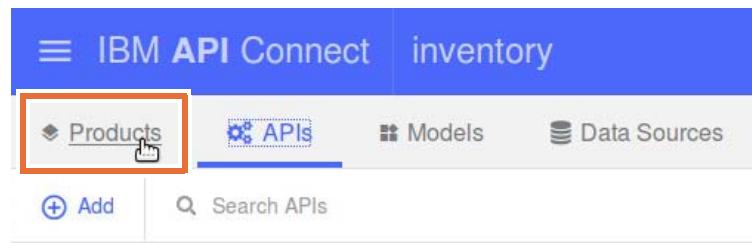


- __ e. Close the activity-log properties view.
__ 11. Save the changes in the inventory API definition.

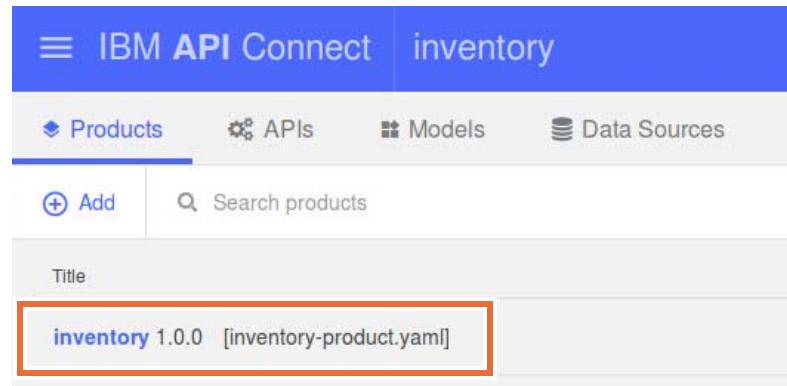
9.2. Modify the product properties and add the APIs to a product plan

Before you publish an API definition to the API Connect Cloud, you must add the API into a **product**. Recall that a product is a collection of API definitions. In this section, you define the membership and metadata for the **think** product.

- 1. Modify the **inventory** product.
 - a. Click the **All APIs** tab.
 - b. In the API Designer main page, click the **Products** tab.



- c. Open the **inventory** product.



Information

When you generate a LoopBack API application, the “apic” utility creates one API definition and one API product in the **definitions** directory. You can update the details of the product, or add the API definition to another product. However, you can add an API definition to one product only.

___ 2. Edit the product information and contact details.

- Title: **think**
- Name: **think**
- Description: The ****think**** product is an online store of memorabilia from IBM's history.
- Contact name: **Thomas Watson**
- Contact email: **thomas@think.ibm**
- Contact URL: **http://www.ibm.com**

Info

Title *

think

Name

think

Version *

1.0.0

Description [Edit](#) [Preview](#) [i](#)

The ****think**** product is an online store of memorabilia from IBM's history.

___ 3. Specify the MIT license and terms of service.

___ a. Select the **License** section of the product.

- ___ b. Enter the following license and terms of service:
- License name: **The MIT License (MIT)**
 - License URL: <https://opensource.org/licenses/MIT>
 - Terms of service: *paste the contents of ~/lab_files/publish/license.txt*

License	Name
	The MIT License (MIT)
	URL
	https://opensource.org/licenses/MIT

Terms of Service	Terms of Service
	Copyright (c) 2017+ IBM

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge,

___ 4. Set the product visibility to **Authenticated users**.

- ___ a. Select the **visibility settings** section.
- ___ b. Change the **Visible to** field to **Authenticated users**.
- ___ c. Leave the **Subscribable by** field to **Authenticated users**.

Visibility

Visible to: ⓘ

Authenticated users

All authenticated developers in consumer organizations who have signed up for this developer portal can see this product

Subscribable by: ⓘ

Authenticated users



Information

The **visibility** setting determines whether an application developer can see an API product on the Developer Portal. The **subscribable** setting determines which type of application developer can subscribe to the product.

- ___ 5. Review the API definitions that belong to the product.
 - ___ a. Select the API section of the product.

APIs

inventory	1.0.0
financing	1.0.0
logistics	1.0.0
oauth-provider	1.0.0

The product includes four APIs: **financing**, **logistics**, **inventory**, and **oauth-provider**. The version number for each API definition appears next to the name.

- ___ 6. Rename the default API plan to **silver**.
 - ___ a. Select the **default plan** entry in the **plans** section.

__ b. Enter the following plan details:

- Title: **Silver Plan**
- Name: **silver**
- Description: **Limited access to APIs**

The screenshot shows the configuration of a 'Silver Plan'. It includes fields for Title ('Silver Plan'), Name ('silver'), Description ('Limited access to APIs'), and Billing Model ('None'). Below these, there's a 'Rate limits (calls / time interval)' section set to '100 / 1 Hour' with an 'Enforce hard limit' checkbox. The entire configuration is enclosed in a light gray border.

Title	Silver Plan		
Name	silver		
Description	Limited access to APIs		
Billing Model	None		
Rate limits (calls / time interval) +			
100	/ 1	Hour	<input type="checkbox"/> Enforce hard limit

__ c. Leave the rate limit as **100 requests per 1 hour**.

__ d. Select all four APIs in the list.

The screenshot shows a list of '4 APIs included' with checkboxes next to each item. All four items are checked: 'inventory 1.0.0', 'financing 1.0.0', 'logistics 1.0.0', and 'oauth-provider 1.0.0'. A red box highlights the entire list. To the right of the list is a 'Show selected only' checkbox.

4 APIs included	
<input checked="" type="checkbox"/>	inventory 1.0.0
<input checked="" type="checkbox"/>	financing 1.0.0
<input checked="" type="checkbox"/>	logistics 1.0.0
<input checked="" type="checkbox"/>	oauth-provider 1.0.0

- __ 7. Add a second API plan, named **gold**.
- __ a. Click the **plus (+)** icon in the Plans section.
- __ b. Select the **New Plan 1** entry.
- __ c. Enter the following plan details:
- Title: **Gold Plan**
 - Name: **gold**
 - Description: **Unlimited access to these APIs for approved users**
 - Rate limit: **Unlimited**
 - Approval: Select **Require subscription approval**

The screenshot shows the configuration page for the 'gold' API plan. It includes sections for Title ('gold'), Description ('Unlimited access to these APIs for approved users'), Billing Model ('None'), Rate limits ('Unlimited'), Burst limits ('No burst limits defined'), and Approval ('Require subscription approval' checked).

gold

Description
Unlimited access to these APIs for approved users

Billing Model
None

Rate limits (calls / time interval) ⊕
Unlimited

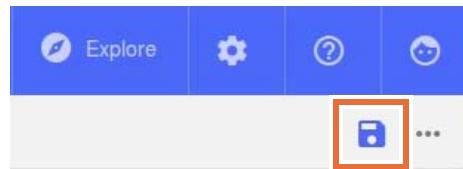
Burst limits (calls / time interval) ⊕
No burst limits defined

Approval
 Require subscription approval

- __ d. Make sure that the **gold** plan applies to all four API definitions in the **think** product.

The screenshot shows a configuration panel for an API product. At the top, under 'Approval', there is a checked checkbox labeled 'Require subscription approval'. Below this, it says '4 APIs included' and there is an unchecked checkbox labeled 'Show selected only'. A list of four APIs is shown, each with a checked checkbox and a dropdown arrow: 'inventory 1.0.0', 'financing 1.0.0', 'logistics 1.0.0', and 'oauth-provider 1.0.0'.

- __ 8. Save your changes to the product.



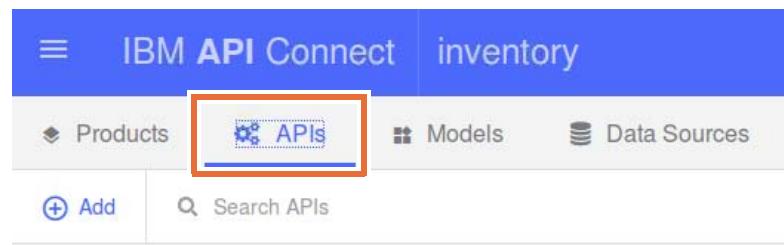
9.3. Enable API key security in the financing and logistics API

Earlier in this exercise, you disabled the API key check in the **financing** and **logistics** APIs to test its operations. Before you publish the **think** product, enable **API key** security in the APIs. When you call an operation in the financing and logistics API, you must provide a valid client ID and client secret value.

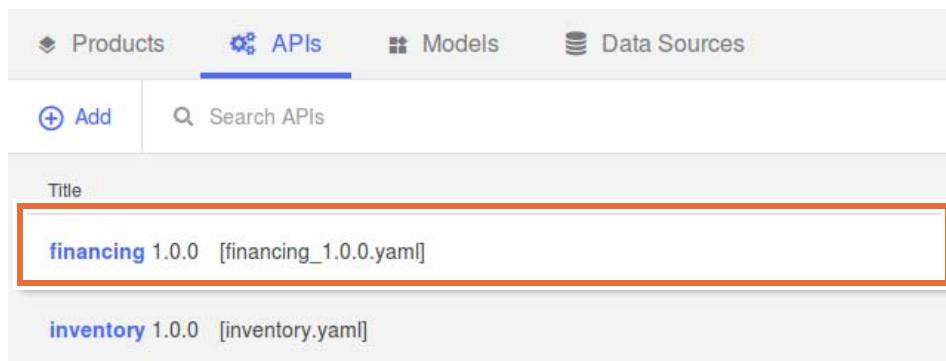
- ___ 1. Switch to the API view in the API Designer web application.
 - ___ a. Click **All Products** to return to the product list.



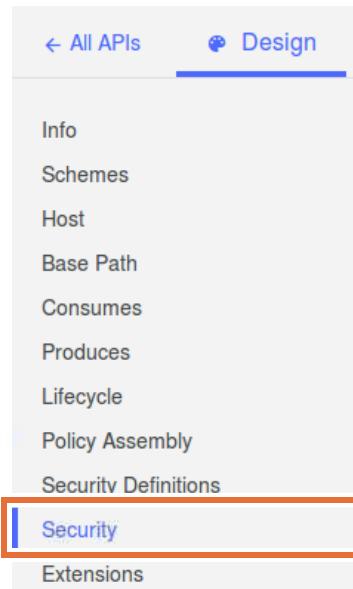
- ___ b. Click the **APIs** tab to open the list of API definitions.



- ___ 2. Open the **financing** API definition.
 - ___ a. Select **financing 1.0.0** from the API list.



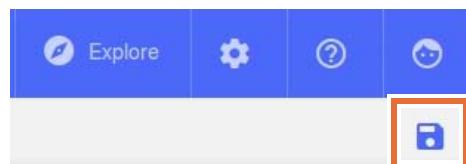
- ___ 3. Enable **API key** security in the financing API.
- ___ a. In the **financing** API definition, select the **security** section in the Design view.



- ___ b. Select the **clientIdHeader (API Key)** security requirement.

The screenshot shows the 'Security' configuration page. It includes a descriptive text box: 'Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.' Below this, there is a list of security requirements. The first item, 'Option 1' with the checked checkbox 'clientIdHeader (API Key)', is highlighted with a red box. There is also a trash icon next to the requirement.

- ___ 4. Save the changes to the **financing** API.



- ___ 5. Open the **logistics** API definition.
- ___ a. Select **All APIs** to open the list of API definitions.



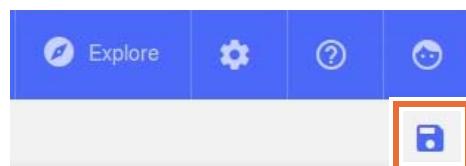
- __ b. Select **logistics 1.0.0** from the API list.

The screenshot shows a navigation bar with 'Products', 'APIs' (selected), 'Models', and 'Data Sources'. Below is a search bar with '+ Add' and 'Search APIs'. The main area lists three APIs: 'financing 1.0.0 [financing_1.0.0.yaml]', 'inventory 1.0.0 [inventory.yaml]', and 'logistics 1.0.0 [logistics_1.0.0.yaml]'. The 'logistics' entry is highlighted with a red box.

- __ 6. Enable **API key** security in the logistics API.
 __ a. Select the **security** section.
 __ b. Enable the **clientID (API Key)** security requirement.

The screenshot shows a left sidebar with 'Security Definitions' containing 'Extensions', 'Properties', 'Paths', and '/shipping'. The 'Security' tab is selected and highlighted with a red box. The main panel is titled 'Security' with the sub-instruction: 'Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.' It shows 'Option 1' with a checked checkbox next to 'clientID (API Key)', which is also highlighted with a red box. A trash icon is visible to the right of the option.

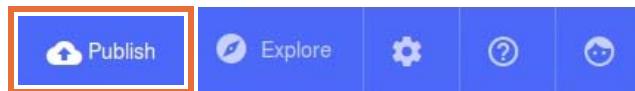
- __ 7. Save the changes to the **logistics** API.



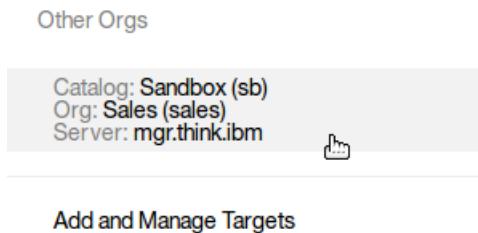
9.4. Publish an API product and API definitions

To make the API products, plans, and definitions available for use, you must publish the files to the API Management server. Open the API Designer web application and publish the product to your API Connect Cloud.

- 1. Sign in to the API Management server as a publish target.
 - a. Open another tab on the browser, and sign on to the API Manager with these credentials:
 - Host address: `mgr.think.ibm`
 - User name: `student@think.ibm`
 - Password: `Passw0rd!`
 - Click **Sign in**.
- 2. Click the other tab on the browser to return to the API Designer.
 - a. In the API Designer page, click **Publish**.



- b. Select the **Sandbox** catalog, **Sales** organization, and **mgr.think.ibm** server that was created previously.



Information

When you publish an API product to the API Connect Cloud, you specify three environment parameters: the **organization**, the **catalog**, and the **app**.

- The **organization** represents a department or team within your company. The members within the organization develop, test, publish, and support the API implementation.
- The **catalog** represents a collection of products in an organization. You separate products and APIs for testing before you make them available to developers.
- The **app**, in this context, is the **application that implements the API definition**. In a previous exercise, you defined the **inventory** API application on the API Manager.

- ___ 3. Publish the API product to the API Connect Cloud.
- ___ a. In the **Publish** page, leave all the options cleared.

Publish

This target only has a catalog and no application. Application will not be published.

Stage only

Select specific products

Cancel **Publish**

- ___ b. Click **Publish**.

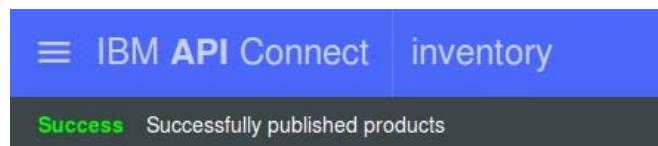
Information

What is the effect of selecting each **publish** option?

- The **publish application** option deploys the **inventory** API application to the server. If you select this option, you overwrite the application that you deployed in an earlier exercise. You also assign a new **app-id** value to the application.
- The **Stage or publish products** option pushes the API products and API definitions to the Management server.
 - When you select **stage only**, you copy the API product and all of its associated API definitions to the API Management server. However, an **organization owner** must log in to the API Manager and change the lifecycle state from **staged** to **published**.
 - When you choose **select specific products**, you can exclude certain API products in the stage step, publish step, or both.

In this exercise, you publish the **API products** and all API definitions that belong to the product: the **inventory**, **financing**, **logistics**, and **oauth-provider** APIs. You do not publish the **inventory** API application. You also **staged** and **published** the product in one step.

- ___ 4. Confirm that API Designer successfully published the product to API Manager.



- ___ 5. Close the **API Designer** tab in the browser.

9.5. Review the product in the API Manager server

The **API Management server**, or **API Manager**, is the central control and administration server in your API Connect Cloud. When you publish an API product, plan, definition, or application, you send your API artifacts to the API Manager. In turn, the API Manager sends the product, plan, and definition files to the API gateway and Developer Portal. The API Manager deploys the API application to the Liberty collective.

In this section, you review the **think** API product in the API Manager. Confirm the publish state of your four API definitions: financing, inventory, logistics, and oauth-provider. Remove any previous versions of the inventory API definition that you uploaded for testing in earlier exercises.

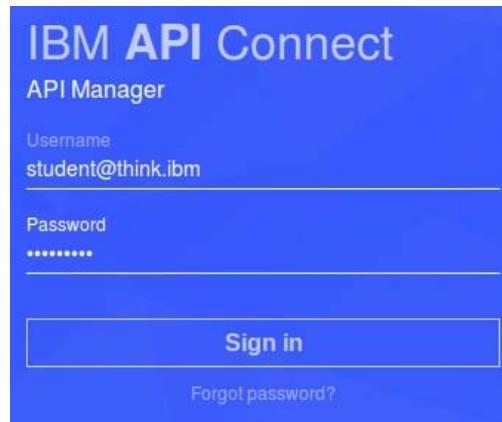
- ___ 1. The API Manager is already open in the browser.



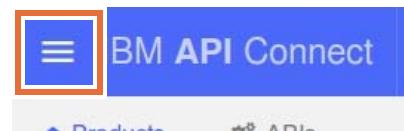
Note

If you closed the browser, open the browser to the API Manager website.

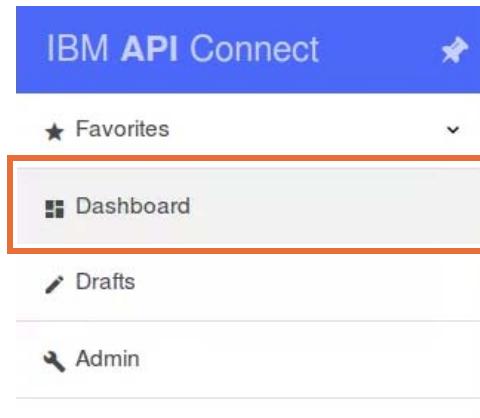
- Open `http://mgr.think.ibm/apim/` in a web browser.
- Log in to the API Manager with the following credentials:
 - User name: `student@think.ibm`
 - Password: `Passw0rd!`



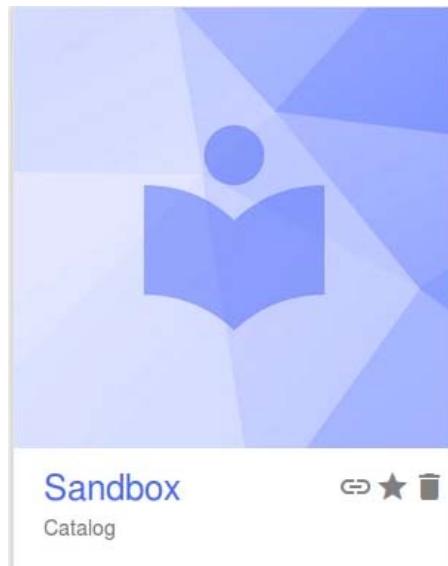
- ___ a. Open the API Manager menu from the upper-left section of the page.



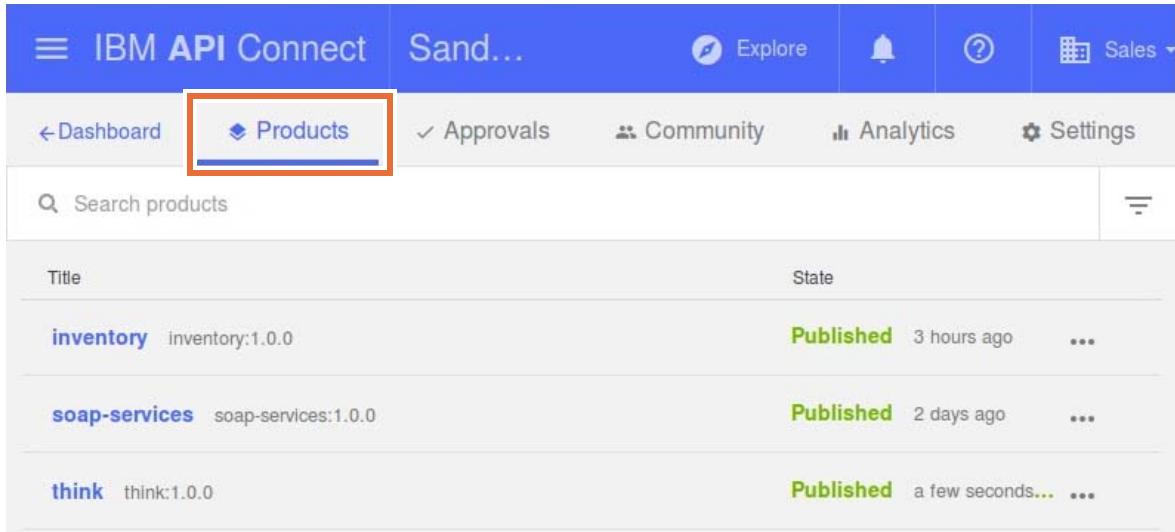
- __ b. Select the **Dashboard** entry.



- __ c. Open the **Sandbox** catalog from the dashboard.



- __ 2. Examine the published products in the sandbox catalog.
 __ a. Click the **Products** tab.



The screenshot shows the IBM API Connect interface with the 'Products' tab highlighted by a red box. The main content area displays three published API products:

Title	State
inventory inventory:1.0.0	Published 3 hours ago
soap-services soap-services:1.0.0	Published 2 days ago
think think:1.0.0	Published a few seconds... ...

Information

The **products** view lists all of the API products that you published from the API Designer:

- In the **policies** lab, you published the **financing** and **logistics** API definitions for testing in the DataPower gateway. The **inventory** product holds the older versions of the API definitions.
- In the **existing** services lab, you published a SOAP API definition that is named **savings** in the **soap-services** product.
- In this exercise, you published an updated version of the **inventory** product. You renamed the product to the **think** product.

Before you continue testing, you retire and remove the **inventory** product from the sandbox catalog. The **think** product contains a copy of the financing and logistics API definitions already.

- __ 3. Remove the **inventory** product that you tested in an earlier exercise.
- At the **inventory1.0.0** API product entry, click the ellipses (...) link.
 - Select **Retire**.





Questions

What is the difference between **deprecating** and **retiring** an API product?

When you deprecate a product, application developers cannot subscribe to the product. However, developers that are already subscribed to the product can continue to use the API.

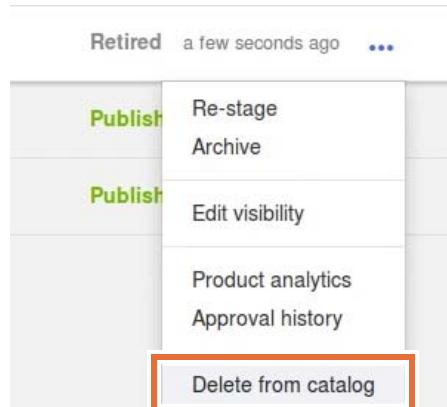
When you retire a product, API Connect makes all existing subscriptions inactive. The product is still published on the Management server, but it is no longer available for use.

- ___ c. Review the confirmation window.

Retiring inventory 1.0.0...

Are you sure you want to retire this product? All subscriptions to plans in this product will be removed.

- ___ d. Click **OK**.
- ___ e. After the **inventory** product is retired, click **Delete from catalog** from the menu.



- ___ f. Click **OK** to confirm product deletion.

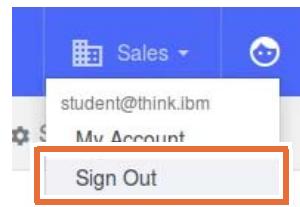
- ___ 4. Confirm that you published two products in the API Manager: soap-services, and think.

The screenshot shows the IBM API Connect dashboard. At the top, there is a banner message: "Success (version) has been removed from Sandbox" with a timestamp "a few seconds ago". Below the banner, there is a link to "Dashboard". A search bar is present with the placeholder "Search products". The main content area displays a table of products:

Title	State
soap-services soap-services:1.0.0	Published 2 days ago ...
think think:1.0.0	Published a few seconds... ...

- ___ 5. Log out of the API Manager.

- ___ a. Click the **user icon** at the upper-right section of the API Manager web page.
___ b. Click **Sign Out** from the menu.



End of exercise

Exercise review and wrap-up

In the first part of the exercise, you modified the invoke URL to route to the address and port number of the Docker image. Next, you examined the concept of an API product: a collection of API definitions that are grouped for deployment. You also defined an API plan: a set of non-functional requirements that govern the rate and limit of API calls from the consumer.

The final part of the exercise examined how to publish the API product, plan, and definitions to the API Management server.

Exercise 10. Subscribing and testing APIs

Estimated time

01:00

Overview

In this exercise, you learn about the application developer experience in the Developer Portal. You create a developer account and add an application. You also review the client ID and client secret values, subscribe to an API plan, and test operations from an API product.

Objectives

After completing this exercise, you should be able to:

- Self-register an application developer account
- Add an API consumer application in a developer account
- Review and reset the client ID and client secret values
- Test an API operation in the Developer Portal
- Test an API operation with a consumer application

Introduction

In the previous exercises in this course, you assumed the role of the API developer: the provider of the API. The API developer defines, implements, secures, and tests the API application.

In this exercise, you focus on the application developer role: the consumer of the API. The application developer registers and develops an application that uses the operations in the published API.

You create an account on the Developer Portal: the repository of information for APIs that are published in the API Connect Cloud. You register your client application in your account. The Developer Portal provides you with the client ID and client secret: metadata that uniquely identifies your application when you make API calls. You test the **financing** and **logistics** API operations from the test client in the Developer Portal. In a final step, you test a JavaScript client application with the OAuth secured **/inventory/items** API operation.

Requirements

Before you start this exercise, you must complete the **data sources**, **remote**, **policies**, **authorization**, **containers**, and **publish** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

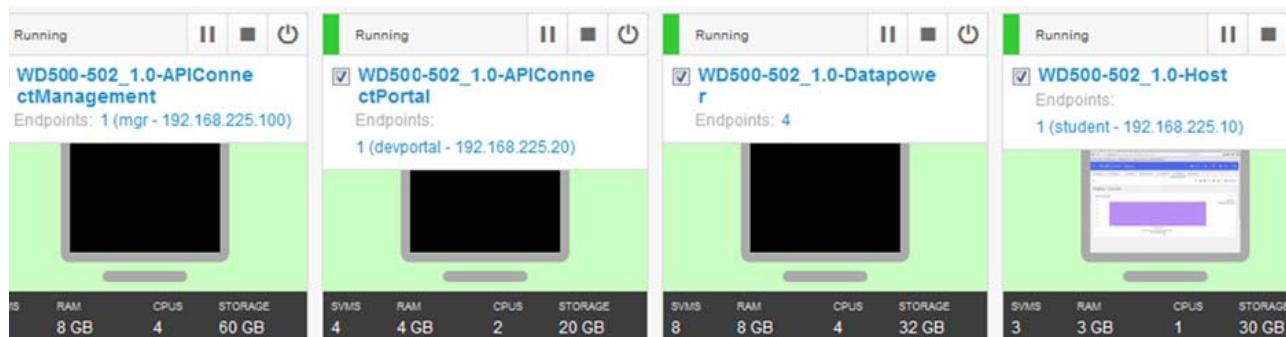
Exercise instructions

Before you begin

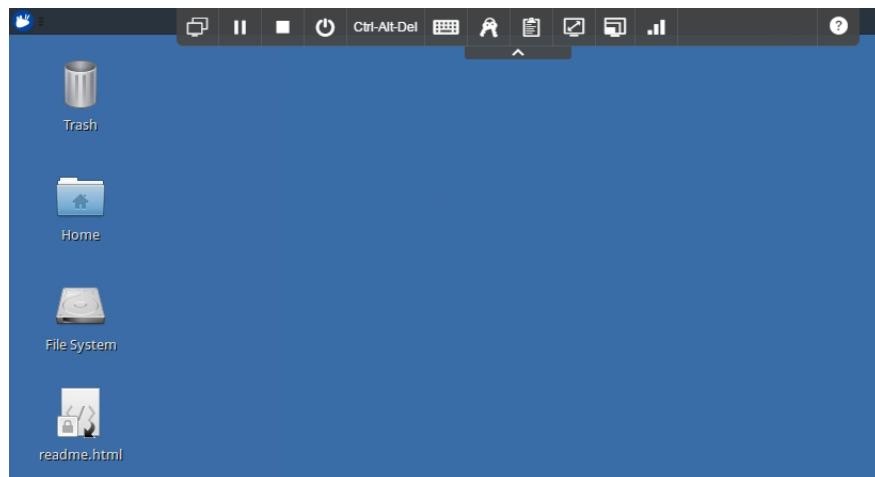
You complete the instructions in this exercise on the remote lab environment. Before you begin your exercise, make sure that all four virtual machines in the IBM API Connect Cloud are running.

When you complete the exercise, you can suspend the lab environment to save your current state.

- ___ 1. In the IBM Remote Lab Platform, make sure that all four virtual machines are started.
 - ___ a. Outside of the Xubuntu host virtual machine, examine the console for the four virtual machines that you use in this course.
 - ___ b. If the four virtual machines (Developer Portal, Management server, DataPower Gateway server, and Xubuntu Host) are not started, start the server.
 - ___ c. Make sure that all four virtual machines are started.



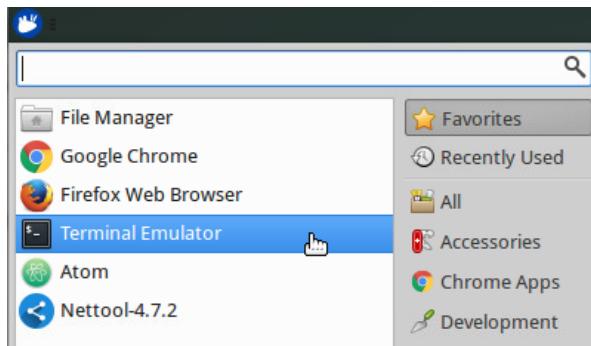
- ___ 2. Open the student workstation on the Xubuntu Host virtual machine.
 - ___ a. Click the picture of the desktop in the Xubuntu Host pane.
 - ___ b. A remote desktop connection opens in a web browser window. Wait until the connection opens.



**Optional**

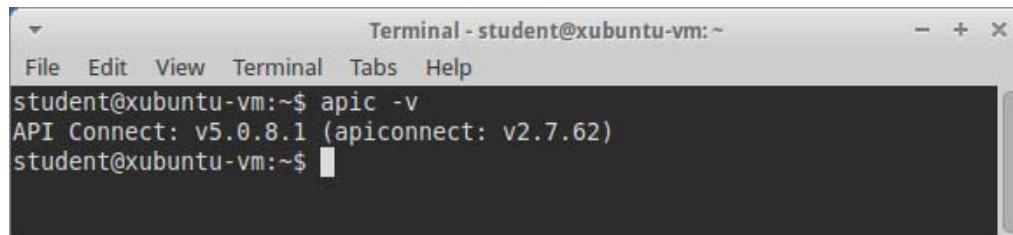
The following instructions verify the version of the IBM API Connect Toolkit that is installed on the Xubuntu virtual machine. If you completed this step in an earlier exercise, skip this step and continue with the current exercise.

- ___ 3. Verify the API Connect Toolkit version from the “apic” command-line utility.
 - ___ a. Open the Xubuntu start menu, which is at the upper-left area of the desktop.
 - ___ b. Open a Terminal Emulator application window.



- ___ c. From the command shell, check the version of the “apic” command-line utility.
- ___ d. Verify the API Connect Toolkit version number.

```
$ apic -v
API Connect: v5.0.8.1 (apiconnect: v2.7.62)
```

**Information**

The “API Connect” version number indicates which IBM API Connect release is used. In this example, the API Connect Toolkit is bundled with version 5.0.8.1.

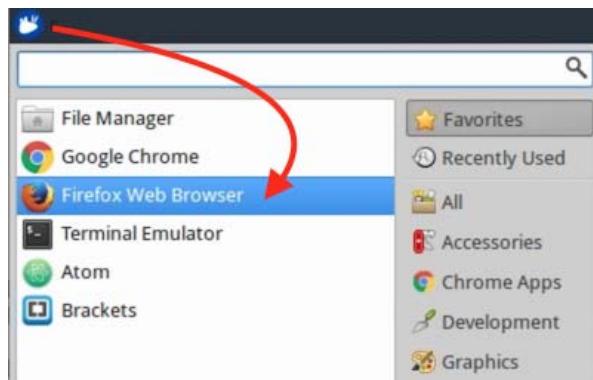
Keep in mind that the “`apic -v`” command does not check the version number of your API gateway, API Management server, or Developer Portal. It is the administrator’s responsibility to make sure that all four components in an API Connect installation are kept up-to-date.

10.1. Register a client application in the Developer Portal

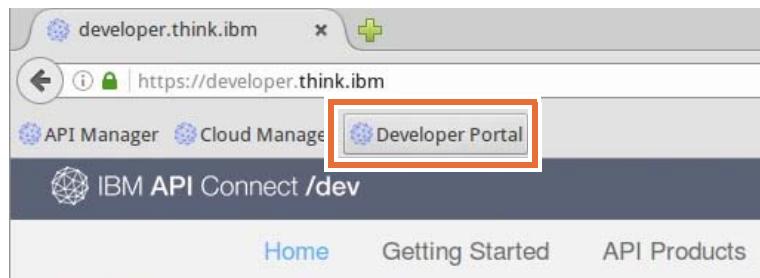
In this section, you complete this first step by creating a developer account on the Developer Portal. The portal provides a **self-registering** system: application developers can sign up for an account without requiring an authorization from the organization owner.

When you defined the **financing**, **logistics**, and **inventory** APIs, you set the API settings as **subscribable** by **authenticated users**. In this scenario, the applications that call operations from these three APIs must first subscribe to the APIs in the Developer Portal. To subscribe to an API, you must have a developer account on the Portal.

- ___ 1. Open the Developer Portal page.
 - ___ a. Open a web browser from the start menu.



- ___ b. Open the Developer Portal page at <https://developer.think.ibm/sales/sb/>.



- ___ 2. Log in to the Developer Portal with the developer@consumer.ibm account.
 - ___ a. Click the **Log in** link.

 A screenshot of a user login form titled "User login". It includes three buttons: "Create new account", "Log in" (which is highlighted with a black border), and "Request new password".

- ___ b. Enter the user credentials for the application developer account.
- User name: `developer@consumer.ibm`
 - Password: `Passw0rd!`

User login

[Create new account](#) **Log in** [Request new password](#)

Username *

Enter your developer.think.ibm username.

Password *

Enter the password that accompanies your username.

Log in

- ___ c. Click **Log in**.
- ___ 3. Register a client application with the `developer@consumer.ibm` account.
- ___ a. Click **Apps** from the navigation bar.
 - ___ b. The account has no registered client applications. Click **Create new App** to register one.

IBM API Connect /dev

developer@consumer.ibm Developer Consumer

Home Getting started API Products **Apps** Blogs Forums Support

No applications have been found.

+ Create new App

- ___ c. In the Register application page, enter **Inventory client application** in the title.

- ___ d. In the Description field, enter: **A web application to browse items for sale, and to submit reviews.**

Register application

Title *

Description

A web application to browse items for sale, and to submit reviews.

OAuth Redirect URI

The URL authenticated OAuth flows for this application should be redirected to.

- ___ e. Click **Submit**.



Information

The Developer Portal assigns a unique identifier, a **client ID**, to the client application that you registered. When you call APIs that are hosted on the API Connect Cloud, you must send the **client ID** in the HTTP request message header.

The Portal also assigns a secret passphrase, a **client secret**, for the client ID. As an extra layer of security, you send the **client ID** and **client secret** in the message header for API calls.

Like a password, you must take precautions to protect the client secret value. You should send a client secret value over a secured TLS/SSL connection.

- ___ 4. Save the **client ID** and **client secret** values in the Notepad application.

- ___ a. In the apps page, select the **Show Client Secret** check box.
- ___ b. Copy the value into your clipboard.



- ___ c. Paste the value into your Notepad application.



Note

You can view your client secret value one time only. If you did not copy down this value, you must reset it in the apps page.

- ___ d. In the client credentials section, select **Show** in the **Client ID** field.

- ___ e. Copy the value and paste it into the Notepad application.

Client secret:
T5IR7jh1ul2vO7xF0wO7fK6lO8iK8IN8hQ4oE4mF8tW8eP7hU3

Client id:
1d22980e-e8e7-4abc-9688-7f373423c858

- ___ 5. Verify the **client secret** value in the Inventory client application app page.

- ___ a. Copy the **client secret** value into the clipboard.

- ___ b. In the Client Credentials section, click **Verify**.

The screenshot shows a 'Client Credentials' form. It has two input fields: 'Client ID' and 'Client Secret'. Below each field is a 'Reset' button. To the right of the 'Client Secret' field is a 'Verify' button, which is highlighted with a red rectangular border.

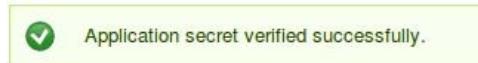
- ___ c. Paste the value into the client secret field.

Verify application secret

The screenshot shows a 'Verify application secret' form. It has a single input field labeled 'Secret *' containing the value 'T5lR7jH1uI2vO7xF0wO7fK6lO8iK8lN8hQ4oE4mF8tW8eP7hU3'. Below the input field is a 'Submit' button.

- ___ d. Click **Submit**.

- ___ e. Confirm that the Developer Portal verified the client secret value.



10.2. Subscribe to a plan for the inventory API product

The **Inventory client application** must subscribe to an API plan to use an API. In this section, review the two plans in the inventory API product. Subscribe to a plan before testing the API operations to which the plan applies.

- 1. Review the **plans** for the inventory API product.
 - a. Click **API Products** from the navigation bar.
 - b. Select the **think (v1.0.0)** product.

The screenshot shows a navigation bar with links: Home, Getting started, API Products (which is highlighted in blue), Apps, Blogs, Forums, and Support. Below the navigation bar, there is a list of API products. The first item is 'think (1.0.0) (4 APIs included)', which is highlighted with a red box. To the left of the product name is a blue circular icon containing a stylized stack of books or documents. To the right of the product name is a rating section with five stars and the text 'No votes yet'.



Note

The Developer Portal looks up the list of published APIs every 10 – 15 minutes. If you published the **think** API product within this time frame, the product might not appear in the API products list.

If you do not see the **think** product in the list, wait a few minutes and refresh the page.

-
- c. In the API product page, scroll down to the **plans** section.
 - d. Examine the **gold** and **silver** plans.

The screenshot shows the 'think 1.0.0' API product page. On the left, there is a sidebar with a tree view of APIs: 'APIs' (selected), 'inventory', 'financing', 'logistics', and 'oauth'. The main area is titled 'Plans' and contains a table with four columns: 'Plan Name', 'Gold', 'Silver', and 'Rate Limit'. The table has five rows corresponding to the APIs listed in the sidebar. Each row has a dropdown arrow next to the API name. The 'Gold' column shows 'unlimited' for all APIs. The 'Silver' column shows '100 per hour' for all APIs. At the bottom of the table are two 'Subscribe' buttons.

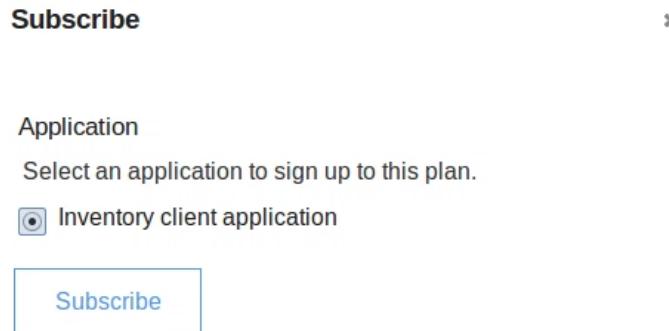
Plan Name	Gold	Silver
inventory 1.0.0	unlimited	100 per hour
financing 1.0.0	unlimited	100 per hour
logistics 1.0.0	unlimited	100 per hour
oauth-provider 1.0.0	unlimited	100 per hour

**Note**

The **gold** plan does not restrict the number of API calls from the client application. However, an organization owner or lifecycle administrator must approve any subscriptions to the **gold** plan.

The **silver** plan limits the client application to 100 API calls in each hour. This plan does not require human approval. Application developers can subscribe to the **silver** plan and use it immediately.

- ___ e. Click **Subscribe** in the **silver** plan.
- ___ f. In the Subscribe window, select the **Inventory client application**.



- ___ g. Click **Subscribe**.
- ___ h. Confirm that you successfully subscribed to the plan.

Successfully subscribed to this plan.

- ___ 2. Review the plan subscriptions for the **Inventory client application**.
 - ___ a. Select the **Apps** section.
 - ___ b. Open the **Inventory client application**.

The screenshot shows the IBM API Management interface with the following navigation bar:

Home Getting started API Products Apps Blogs Forums Support 🔍

[+ Create new App](#)

A card for the **Inventory client application** is displayed, featuring a mobile phone icon and the following text:

Inventory client application
A web application to browse items for sale, and to submit reviews.

- ___ c. Confirm that the **think (v1.0.0) silver** plan appears in the Subscriptions section.

The screenshot shows the 'Inventory client application' configuration page. At the top, there's a 'Description' section with a note: 'A web application to browse items for sale, and to submit reviews.' Below it is a 'Credentials' section with a 'Default' profile. The 'Client ID' field contains a long string of characters, and there are 'Show' and 'Edit' buttons. The 'Client Secret' field is grayed out, with 'Verify' and 'Reset' buttons next to it. In the bottom right corner of this section, there's a blue pencil icon followed by the word 'Edit'. Below this is a 'Subscriptions' section containing a single item: 'think (1.0.0) (silver)'. This item is highlighted with a red rectangular border. To the right of the subscription name are 'View Details' and 'Unsubscribe' buttons. The entire screenshot is framed by a thin gray border.

10.3. Test a subscribed API in the test client

In the last section, you subscribed the Inventory client application to the silver plan from the think (v1.0.0) API product.

In this section, test two APIs from the **think** product in the Developer Portal test client. Test the **GET /shipping** and **GET /stores** operations from the **logistics** API. Examine and test the **GET /calculate** operation from the **financing** API.

- 1. Open the test client with the **think 1.0.0** API product.
 - a. Select the **API Products** section of the Developer Portal.
 - b. In the **think 1.0.0** product, select the **logistics** entry.

The screenshot shows the 'think 1.0.0' product page in the IBM Developer Portal. On the left, there's a sidebar with 'APIs' and a list of products: 'inventory', 'financing', 'logistics' (which is highlighted with a red box), and 'oauth-provider'. On the right, there's a circular icon with three stacked bars, a rating of 5 stars, and the text 'No votes yet'. Below that is a 'Description' section with the text: 'The think product will provide really awesome APIs to your application.'

- c. Examine the test client.

The screenshot shows the test client interface for the logistics 1.0.0 API. The left sidebar lists operations: 'Inventory 1.0.0', 'financing 1.0.0', 'logistics 1.0.0', 'Operations', and 'Definitions'. Under 'Operations', 'GET /shipping' is selected and highlighted with a blue box. The main area shows the '/shipping' endpoint for 'GET /shipping'. It includes a 'Summary' section with the description 'Calculate shipping costs to a destination zip code', a 'Security' section, and a 'Parameters' section with a 'zip' parameter described as 'Destination zip code.'. To the right, there's a dark panel with 'Example Request' containing a curl command, and a 'Subscribe' button.

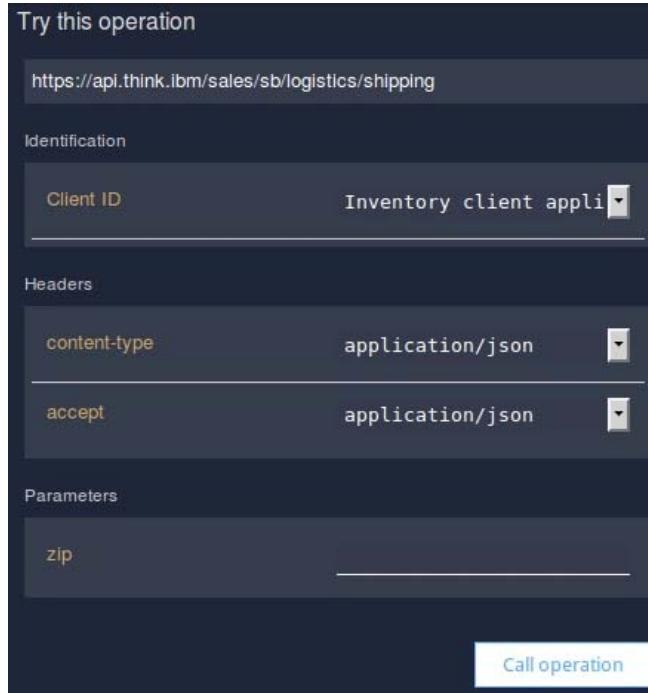


The test client in the Developer Portal has a similar structure to the **explore** view in the API Designer application and API Manager website. The client is divided into three columns:

- The **left** column lists the operations and schema definitions in the API product.
- The **middle** column provides documentation on the selected operation.

- The **right** column lists client code examples and the test client application.
-

2. Test the **GET /shipping** operation in the **logistics** API.
 - a. Select the **GET /shipping** operation from the left column.
 - b. In the rightmost column, scroll down to the **Try this operation** section.



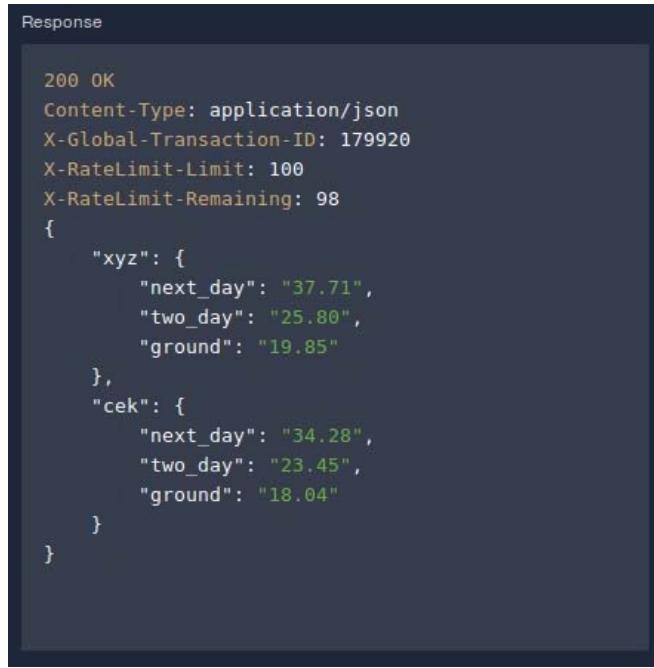
Note

The test client automatically adds the **client ID** and **client secret** from your registered application. In this case, the test client calls the **GET /shipping** operation with the client credentials of your **Inventory client application**.

This test client is useful in learning about API operations without building a sample application.

-
- c. Enter a postal code of **90210** in the **zip** field.
 - d. Click **Call operation**.

- ___ e. Confirm that the operation returns a status of **200 OK**.



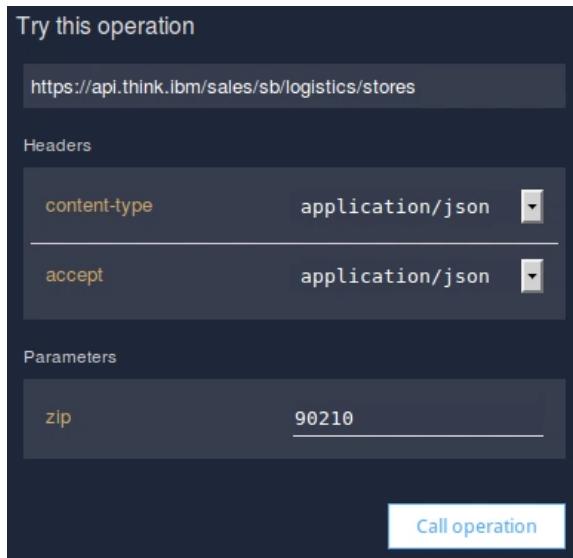
```

Response

200 OK
Content-Type: application/json
X-Global-Transaction-ID: 179920
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 98
{
  "xyz": {
    "next_day": "37.71",
    "two_day": "25.80",
    "ground": "19.85"
  },
  "cek": {
    "next_day": "34.28",
    "two_day": "23.45",
    "ground": "18.04"
  }
}

```

- ___ 3. Test the **GET /stores** operation in the **logistics** API.
- Select the **GET /stores** operation from the left column.
 - In the right column, scroll down to the **try this operation** section.
 - Enter **90210** in the **zip** input parameter.



Try this operation

https://api.think.ibm/sales/sb/logistics/stores

Headers

content-type	application/json
accept	application/json

Parameters

zip	90210
-----	-------

Call operation

- ___ d. Click **Call operation**.

- __ e. Confirm that the operation returns a status of **200 OK**.

```

Request
GET https://api.think.ibm/sales/sb/logistics/stores?zip=90210
content-type: application/json
accept: application/json

Response
200 OK
Content-Type: application/json
X-Global-Transaction-ID: 187264
{
  "maps_link": "https://www.openstreetmap.org/#map=16/34.1047478/-118.4162066"
}

```



Optional

Copy the maps link to display a map of the area in the postal code that you entered.

- __ 4. Test the **GET /calculate** operation from the **financing** API definition.
- __ a. Select **GET /calculate** from the left column.
 - __ b. Scroll down to the **try this operation** section in the right column.
 - __ c. Enter the following values into the test client:
 - Amount: 300
 - Duration: 24
 - Rate: 0.04

The screenshot shows the IBM API Explorer interface. On the left, the 'APIs' sidebar lists 'inventory 1.0.0', 'financing 1.0.0' (selected), 'logistics 1.0.0', and 'oauth-provider 1.0.0'. The 'Operations' section for 'financing 1.0.0' is expanded, showing the 'GET /calculate' operation selected (highlighted with a red box). The main panel displays the 'try this operation' details:

- Headers**:

content-type	application/json
accept	application/json
- Parameters**:

amount	300
duration	24
rate	0.04

A large red box highlights the 'GET /calculate' operation in the sidebar and the parameter input fields in the main panel.

- ___ d. Click **Call operation**.
- ___ e. Confirm that the operation returned a status code of **200 OK**.

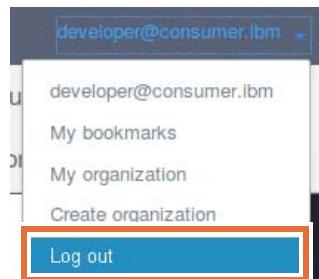


The screenshot shows a terminal window titled "Response". The output is a JSON object representing a SOAP envelope. The status line says "200 OK". The content type is "application/json". The X-Global-Transaction-ID header is "187121". The JSON structure includes "soapenv:Envelope" and "soapenv:Body" fields, both containing XML content with namespaces like "http://schemas.xmlsoap.org/soap/envelope/" and "http://services.think.ibm".

```
Response

200 OK
Content-Type: application/json
X-Global-Transaction-ID: 187121
{
    "soapenv:Envelope": {
        "@xmlns": {
            "soapenv": "http://schemas.xmlsoap.org/soap/envelope/",
            "ser": "http://services.think.ibm"
        },
        "soapenv:Body": {
            "@xmlns": {
                "soapenv": "http://schemas.xmlsoap.org/soap/envelope/",
                "ser": "http://services.think.ibm"
            }
        }
    }
}
```

- ___ 5. Log out of the Developer Portal.
 - ___ a. Select `developer@consumer.ibm` at the upper-right section of the Developer Portal web page.
 - ___ b. Click **Log out** from the menu.



- ___ 6. Close the web browser.

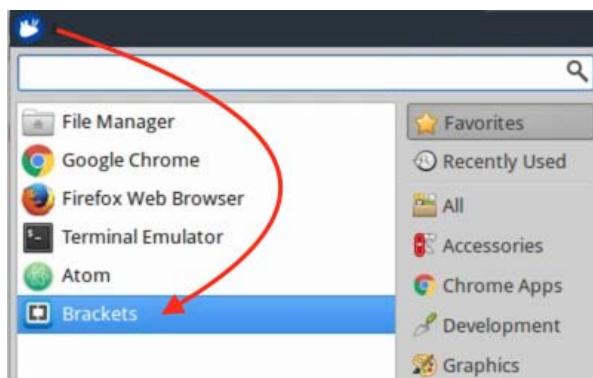
10.4. Update the client application with the client ID and client secret

In a previous exercise, you defined two security requirements on the **inventory** API definition:

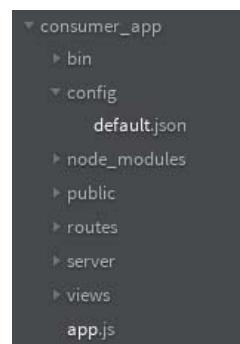
- All operations require an **API key**. The client must provide a valid **client ID** and **client secret** value in the HTTP request header when it calls an API operation.
- All operations require **OAuth authentication**. In a two-step process, the client application must retrieve a valid **authorization code** from the **oauth-provider** API. After authorization, the client must exchange the code for an **access token**. The client must send the access token with every API operation call.

In this section, you add the **client ID** and **client secret** values to a configuration file in a sample JavaScript client application. You test the **GET /inventory/items** API operation with the web client that performs an OAuth authorization and token exchange.

- ___ 1. Open the **Brackets** text editor.
 - ___ a. Select **Brackets** from the application menu.



- ___ 2. Copy the **client ID** and **client secret** values into the consumer application.
 - ___ a. In the navigation column, select the `~/lab_files/devportal/consumer_app/config/default.json` file.



- __ b. Copy and paste the **client ID** and **client secret** values from the Notepad to the **default.json** configuration file.



```
1  "Application": {  
2      "client_id": "1d22980e-e8e7-4abc-9688-7f373423c858",  
3      "client_secret":  
4          "T5lR7jHluI2v07xF0wO7fK6l08iK8lN8hQ4oE4mF8tW8eP7hU3"  
5  }  
6  ▼  "API-Server": {  
7      "protocol": "https",  
8      "host": "api.think.ibm",  
9      "org": "sales",  
10     "catalog": "sb"  
11 },
```



Note

To copy a value, select the text in the Notepad application and press Ctrl+C.

To paste a value, highlight the text in the text editor and press Ctrl+V.

- __ c. Save the changes to the **default.json** file.



10.5. Test a subscribed API with a sample application

- __ 1. Start the Loopback application in the runtime environment.

- __ a. Open the **terminal emulator** application to the inventory directory.

```
$ cd ~/inventory/
```

- __ b. Start the Docker image that runs the inventory API.

```
$ docker run --add-host mysql.think.ibm:192.168.225.10 --add-host mongo.think.ibm:192.168.225.10 -p 3000:3000 apic-inventory-image
```

```
> inventory@1.0.0 start /inventory
> node .
```

Web server listening at: http://localhost:3000

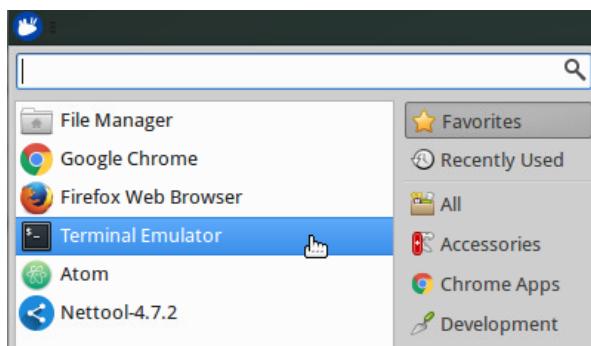


Note

You test the client application with a single instance of the inventory application that runs in the Docker runtime. Optionally, you can start Docker in swarm mode with a cluster of Docker servers that run the inventory application in the runtime environment.

- __ 2. Start the **consumer** application from another terminal emulator application.

- __ a. Open the **Terminal Emulator** application from the favorites menu.



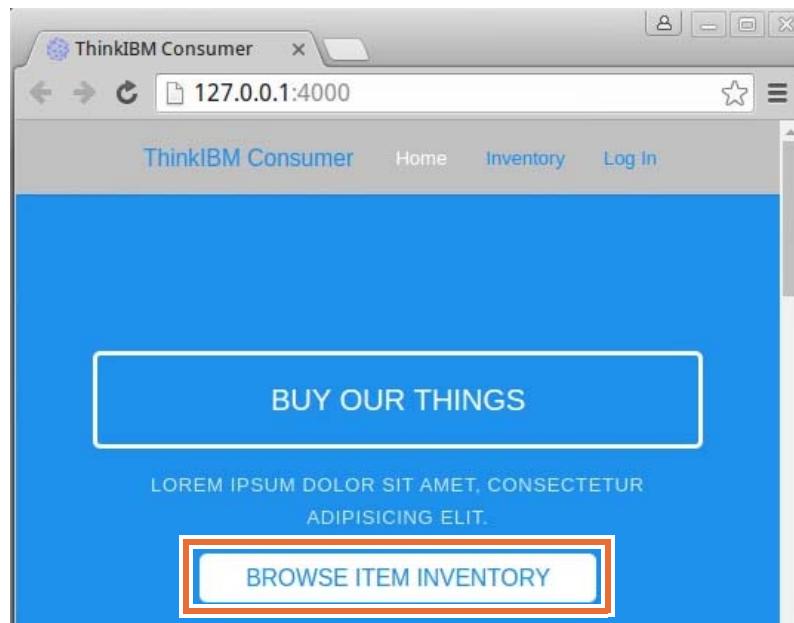
- __ b. Navigate to the `consumer_app` directory.

```
$ cd ~/lab_files/devportal/consumer_app/
$ pwd
/home/student/lab_files/devportal/consumer_app/
```

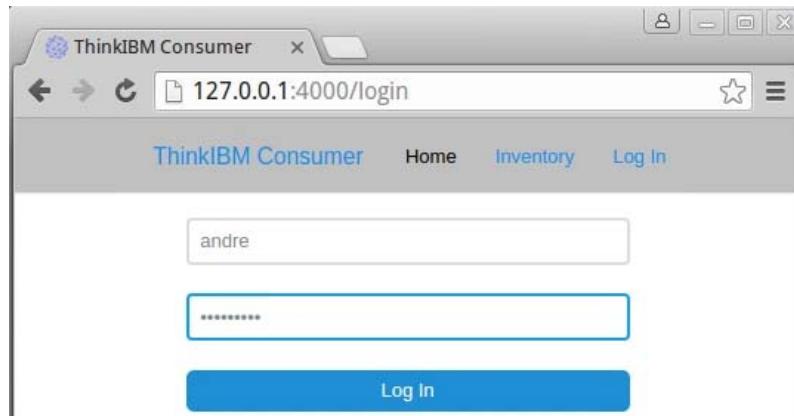
- __ c. Start the consumer application.

```
$ npm start
```

- ___ 3. Test the `GET /inventory/items` API operation from the consumer application.
 - ___ a. When you start the consumer application, a web browser opens to the main page. Click **BROWSE ITEM INVENTORY**.



- ___ b. Enter any value in the **user name** and **password** fields. The sample authentication service that you configured in the OAuth authorization exercise accepts any value as credentials.





Information

The user login page is part of the OAuth 2.0 authentication message flow. In the first step, a user selects the **browse item inventory** link on the main page. Since the **GET /inventory/items** operation is secured by the OAuth authentication, a valid **resource owner** must log in and **grant permission** to access the **/inventory/items** resource.

You can view the output in the terminal emulator to see the output from the OAuth message flow:

- 1) The user requests access to the **/inventory/items** resource.

```
GET /inventory 302 10.850 ms - 56
GET /login 200 124.067 ms - 1622
```

- 2) The consumer application redirects the request to an OAuth login page. The resource owners enter their user name and password. The client application sends the credentials to the **https://api.think.ibm/sales/sb/oauth20/authorize** API operation to verify the credentials.

Initializing OAuth client...

```
Authorization URL: https://api.think.ibm/sales/sb/oauth20/authorize
Token URL: https://api.think.ibm/sales/sb/oauth20/token
Client ID: d53ce870-3d74-4c0c-b46c-0571679b6e2b
Client Secret: W5wS7tYlnA2vG0qX2oD2eA8rC5wM1xJ4yU7iT7tT3wO8fL7xE3
Auth Grant Type: password
Redirect URI: (undefined)
Scope: inventory
```

- 3) The sample authentication service accepts the credentials. In turn, the **/oauth20/authorize** API operation returns an authorization code to the client application.
- 4) The client application exchanges the authorization code for an **access token** with a call to **https://api.think.ibm/sales/sb/oauth20/token**. The token service returns a time-limited token to the client application.

... authorize and token calls successful.

Using OAuth Token:

```
AAIkZDUzY2U4NzAtM2Q3NC00YzBjLWI0NmMtMDU3MTY3OWI2ZTJiAWzKPm9YYtgOPFa4PJSWD0quFHo
RE8gi90xHnfmwvUSs3caZ2o50a6ECUIDkslrSREOgucsWIM09oFPzMMcGvw
```

```
POST /login 302 267.674 ms - 64
```

- 5) On subsequent calls to **/inventory/items**, the client application sends the access token in the HTTP request message header.

MY OPTIONS:

```
{ "method": "GET", "url": "https://api.think.ibm/sales/sb/inventory/items?filter[order]=name%20ASC", "strictSSL": false, "headers": { "X-IBM-Client-Id": "d53ce870-3d74-4c0c-b46c-0571679b6e2b", "X-IBM-Client-Secret": "W5wS7tYlnA2vG0qX2oD2eA8rC5wM1xJ4yU7iT7tT3wO8fL7xE3", "Authorization": "Bearer" }
```

```
AAIkZDUzY2U4NzAtM2Q3NC00YzBjLWI0NmMtMDU3MTY3OWI2ZTJiAWzKPM9YYtgOPFa4PJSWD0quFHo
RE8gi90xHnfmwvUSs3caZ2o50a6ECUIDkslrSREOgucsWIM09oFPzMMCgvw" } }
```

GET /inventory 200 456.900 ms - 10785

-
4. Confirm that the client application displays inventory items in the page.

The terminal window shows the following log output:

```
File Edit View Terminal Tabs Help
GET /stylesheets/pure/layouts/main.css 304 4.411 ms - -
GET /stylesheets/pure/layouts/login.css 304 4.696 ms - -
GET /javascripts/calc-shipping.js 304 4.858 ms - -
Using OAuth Token: AAIkZDUzY2U4NzAtM2Q3NC00YzBjLWI0NmMtMDU3MT
tgOPFa4PJSWD0quFHoRE8gi90xHnfmwvUSs3caZ2o50a6ECUIDkslrSREOguc
POST /login 302 267.674 ms - 64
MY OPTIONS:
[{"method":"GET","url":"https://api.think.ibm/sales/sb/invento
er]-name%20ASC","strictSSL":false,"headers":{"X-IBM-Client-ID
0c-b46c-0571679b6e2b","X-IBM-Client-Secret":"W5wS7ty1nA2vG0qX
iT7tT3w08fL7xE3","Authorization":"Bearer AAIkZDUzY2U4NzAtM2Q3
3MTY3OWI2ZTJiAWzKPM9YYtgOPFa4PJSWD0quFHoRE8gi90xHnfmwvUSs3caZ
guucsWIM09oFPzMMCgvw"}}
GET /inventory 200 456.900 ms - 10785
GET /stylesheets/pure/pure-min.css 304 3.803 ms - -
GET /stylesheets/pure/grids-responsive-min.css 304 5.595 ms -
GET /stylesheets/font-awesome/font-awesome.css 304 5.875 ms -
GET /stylesheets/pure/layouts/main.css 304 5.250 ms - -
GET /stylesheets/pure/layouts/inventory.css 200 9.658 ms - 59
GET /javascripts/inventory-sort.js 200 9.741 ms - 647
GET /images/items/608-calculator.jpg 200 11.231 ms - 35920
GET /images/items/collator.jpg 200 10.222 ms - 26927
GET /images/items/meat-chopper.jpg 200 10.602 ms - 54219
GET /images/items/electric-card-punch.jpg 200 7.887 ms - 5316
```

The browser window shows a product listing for a 'Calculator'. The product image is a black electronic calculator with a numeric keypad and function keys. Below the image, the product name 'Calculator' is displayed, followed by its price '\$5199.99', and a five-star rating icon.



Troubleshooting

If you do not see a list of inventory items, review [Exercise 11, "Troubleshooting the case study"](#) for instructions on how to correct configuration issues in your API Connect Cloud.

End of exercise

Exercise review and wrap-up

The first part of the exercise covered the client application registration process in the Developer Portal. To use APIs that are published in API Connect, the client application developer must subscribe to API. The Developer Portal issues a client ID and client secret: credentials that uniquely identify a client application when it calls API operations.

The second part of the exercise focused on testing the operations of the **inventory** API. In the previous exercises in this course, you built a LoopBack API application that returns item records from the inventory database. In subsequent exercises, you secured the API with OAuth 2.0 authorization. You configured a client web application with the client credentials and tested the inventory API in the last step.

Exercise 11. Troubleshooting the case study

Estimated time

01:15

Overview

In this exercise, you review and apply troubleshooting steps in the exercise case study. You apply the steps to verify the operation of the inventory application and correct the issues that you identify.

Objectives

After completing this exercise, you should be able to:

- Verify the key components in the exercise case study
- Troubleshoot and correct common issues with the case study
- Isolate implementation issues in your own API Connect Cloud solution

Introduction

In the previous exercise, you tested the **logistics**, **financing**, and **inventory** APIs in the Developer Portal test client. You also tested the OAuth 2.0 message flow with the **consumer app**, a client-side JavaScript client.

Requirements

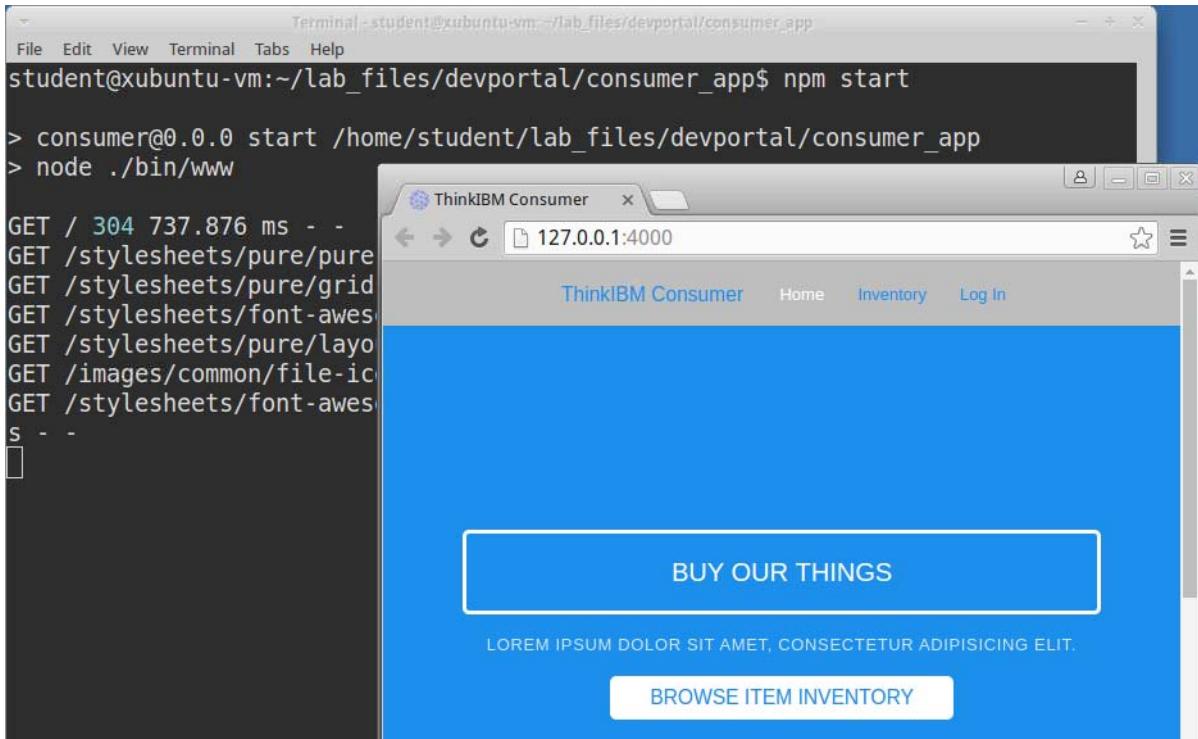
Before you start this exercise, you must complete the **data sources**, **remote**, **policies**, **authorization**, **containers**, **publish**, and **Developer Portal** exercises in this course.

You must complete this lab exercise in the student development workstation: the Xubuntu host environment. This virtual machine is preconfigured with the Node runtime environment, the “npm” Node package manager, and the IBM API Connect toolkit.

Exercise instructions

11.1. Review the inventory items API operation

In the **Developer Portal** exercise, you used the consumer client application to test the **GET /inventory/items** operation. If you encounter a runtime error when you test the operation, you must determine which part of the API operation caused the issue.



The consumer_app makes three API calls when you click **Log in**:

- Call #1: Consumer app > API gateway **/oauth20/authorize** operation
- Call #2: Consumer app > API gateway **/oauth20/token** operation
- Call #3: Consumer app > API gateway **/inventory/items** > > **Inventory LoopBack API**

In *call #1*, the consumer application calls the **/oauth20/authorize** operation from the OAuth 2.0 Provider API. The authorize operation redirects your web browser to a login page. You enter any value for the user name and password, and the authorization URL returns HTTP status code 200 to the OAuth 2.0 Provider. In turn, the authorize API operation returns an OAuth authorization bearer token to the consumer application.

In *call #2*, the consumer application sends the authorization bearer token to the **/oauth20/token** operation at the OAuth 2.0 Provider API. After the token API operation verifies that the bearer token is valid, it returns an OAuth 2.0 access token to the consumer application.

In *call #3*, the consumer application calls the **/inventory/items** API operation with the access token.

11.2. Test the OAuth authorize and token API operations

When you call an operation in the inventory API, OAuth 2.0 Provider must authenticate the resource owner identity and authorize your request.

The OAuth 2.0 message flow consists of two steps:

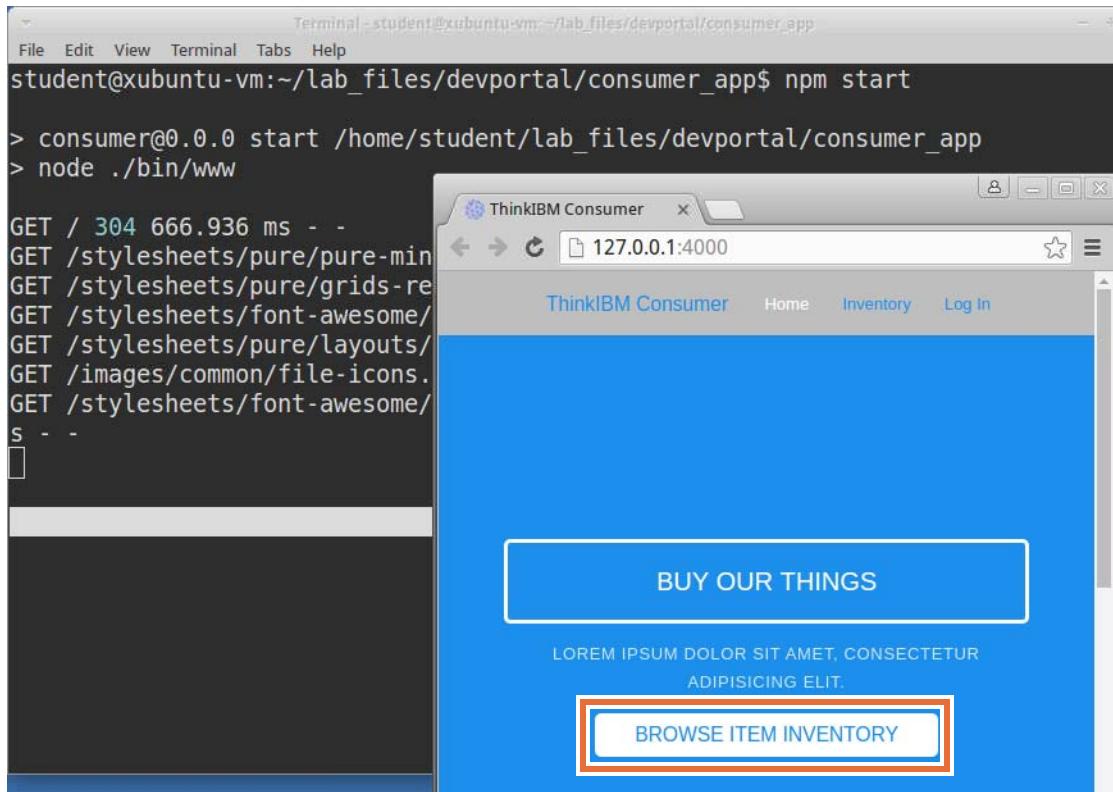
- In step 1, the consumer app sends the resource owner user name and password to the **/oauth/authorize** operation. If the user is authenticated, the operation returns an **authorization bearer token**.
- In step 2, the consumer app sends the authorization bearer token to the **/oauth20/token** operation. The token operation **exchanges** a valid bearer token for an **access token**.

In this section, make sure that the **authorize** and **token** API operations are hosted at the API gateway. Test the operations before you proceed to the next section.

- 1. Test the **/oauth20/authorize** and **/oauth20/token** operations with the **consumer** application.
 - a. Open a terminal emulator window.
 - b. Change directory to the consumer client application, at:
`~/lab_files/devportal/consumer_app`
`$ cd ~/lab_files/devportal/consumer_app`
`$ pwd`
`/home/student/lab_files/devportal/consumer_app`
 - c. Start the **consumer app**.
`$ npm start`
 - d. Confirm that the consumer app started successfully.
`> consumer@0.0.0 start /home/student/lab_files/devportal/consumer_app`
`> node ./bin/www`

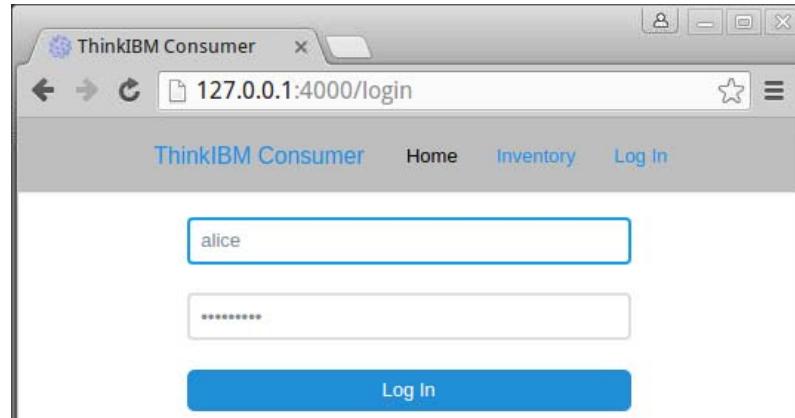
`GET / 304 666.936 ms --`
`GET /stylesheets/pure/pure-min.css 304 10.010 ms --`
`GET /stylesheets/pure/grids-responsive-min.css 304 9.816 ms --`
`GET /stylesheets/font-awesome/font-awesome.css 304 11.010 ms --`
`GET /stylesheets/pure/layouts/main.css 304 13.720 ms --`
`GET /images/common/file-icons.png 304 0.344 ms --`
`GET /stylesheets/font-awesome/fonts/fontawesome-webfont.woff?v=4.0.3 304 3.944 ms --`

__ e. In the consumer application web page, click **BROWSE ITEM INVENTORY**.



__ f. In the resource owner login page, enter the following values:

- User name: **alice**
- Password: **Passw0rd!**

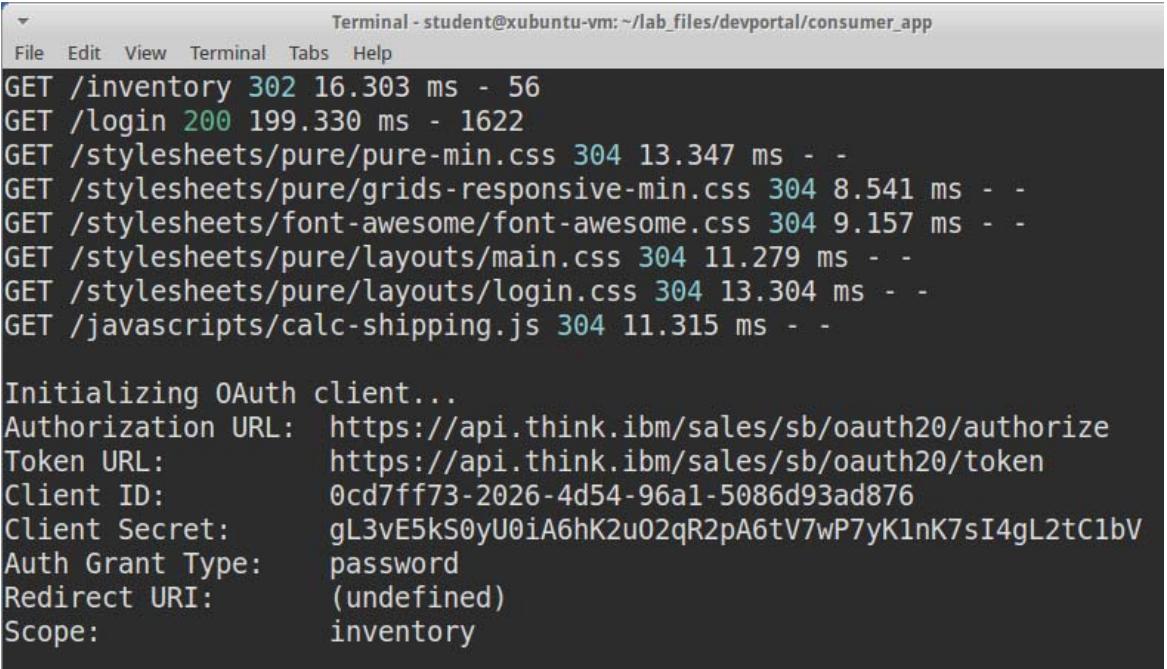


__ g. Click **Log in**.

__ 2. Review the results from the **authorize** and **token** API calls.

__ a. Review the console log entries in the terminal window.

```
Initializing OAuth client...
Authorization URL: https://api.think.ibm/sales/sb/oauth20/authorize
Token URL: https://api.think.ibm/sales/sb/oauth20/token
Client ID: 0cd7ff73-2026-4d54-96a1-5086d93ad876
Client Secret: gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV
Auth Grant Type: password
Redirect URI: (undefined)
Scope: inventory
```



The screenshot shows a terminal window titled "Terminal - student@xubuntu-vm: ~/lab_files/devportal/consumer_app". The window contains two sets of text. The top set is configuration information for an OAuth client, identical to the one listed above. The bottom set is a log of HTTP requests made by the application, showing various CSS and JavaScript files being loaded. The log includes status codes like 302, 200, and 304, along with their respective response times in milliseconds.

```
File Edit View Terminal Tabs Help
GET /inventory 302 16.303 ms - 56
GET /login 200 199.330 ms - 1622
GET /stylesheets/pure/pure-min.css 304 13.347 ms - -
GET /stylesheets/pure/grids-responsive-min.css 304 8.541 ms - -
GET /stylesheets/font-awesome/font-awesome.css 304 9.157 ms - -
GET /stylesheets/pure/layouts/main.css 304 11.279 ms - -
GET /stylesheets/pure/layouts/login.css 304 13.304 ms - -
GET /javascripts/calc-shipping.js 304 11.315 ms - -
Initializing OAuth client...
Authorization URL: https://api.think.ibm/sales/sb/oauth20/authorize
Token URL: https://api.think.ibm/sales/sb/oauth20/token
Client ID: 0cd7ff73-2026-4d54-96a1-5086d93ad876
Client Secret: gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV
Auth Grant Type: password
Redirect URI: (undefined)
Scope: inventory
```



Information

The console displays the configuration settings and the results from calling the **authorize** and **token** services at the OAuth 2.0 Provider API.

- The **authorization URL** and **token URL** must match the values that you defined in the **oauth-provider_1.0.0.yaml** API definition.
- The **client ID** and **client secret** values must match the values that you verified in the Developer Portal.
- The **authorization grant type** is set to `password`, the **scope** is `"inventory"`, and the redirect URI is not specified.

If you sent the correct client ID and secret values to the **authorize** operation, the server sends back an authorization bearer token. The consumer application calls the **token** operation to exchange the bearer token for an **access token**.



Troubleshooting

If you see a `"client_id unauthorized"` error message, the **OAuth provider API** rejected the login attempt from the **consumer** application.

... cannot retrieve OAuth access token.

Reason:

```
{"status":401,"body":{"error":"invalid_client","error_description":"client_id unauthorized"}}
```

The screenshot shows a browser window with the URL `127.0.0.1:4000/login`. The page displays an error message: **Error: Cannot retrieve OAuth access token.** Below it, the reason is given as a JSON object: `{"status":401,"body": {"error": "invalid_client", "error_description": "client_id unauthorized"}}`.

Below the error message, there is a section titled **During OAuth authorize** which lists the following parameters:

- Authorization URL: `https://api.think.ibm/sales/sb/oauth20/authorize`
- Token URL: `https://api.think.ibm/sales/sb/oauth20/token`
- Client ID: `0cd7ff73-2026-4d54-96a1-5086d93ad876`
- Client Secret: `L3vE5kS0yU0iA6hK2u02qR2pA6tV7wP7yKlnK7sI4gL2tC1bV`
- Auth Grant Type: `password`
- Redirect URI: `(undefined)`
- Scope: `inventory`

Two scenarios can cause this error:

- If the **client ID** and **client secret** values *do not match* the values in the Developer Portal, the **authorize** operation rejects the request. Verify the credentials for the client application again.
- If the **client ID** and **client secret** values *do match* the Developer Portal settings, make sure that you subscribed the application to the **Think** product in the Portal.

The screenshot shows two parts of the IBM Developer Portal interface.

The top part shows the details for the **Inventory client application**. It includes a mobile device icon, the application name, an **Update** button, and a **Description** field containing the text: "A web application to browse items for sale, and to submit reviews."

The bottom part shows the **Subscriptions** section. It lists a single subscription: `think (1.0.0) (silver)`. This item is highlighted with a red border. To the right of the subscription, there are **View Details** and **Unsubscribe** buttons.



Troubleshooting

If you see an **"API not found for requested URI"** error message, the **consumer** application cannot reach the `/oauth20/authorize` or `/oauth20/token` operations. Verify the **base path** and **path** settings in the **oauth-provider** API definition.

... cannot retrieve OAuth access token.

```
Reason: {"status":404,"body":{"httpCode":"404","httpMessage":"Not Found","moreInformation":"API not found for requested URI"}}
```

The screenshot shows a browser window with the URL `127.0.0.1:4000/login`. The page displays an error message: **Error: Cannot retrieve OAuth access token.** Below it, the **Reason:** is shown as a JSON object: `{"status":404,"body":{"httpCode":"404","httpMessage":"Not Found","moreInformation":"API not found for requested URI"}}`.

Below the error message, there is a section titled **During OAuth authorize** which lists the following configuration parameters:

Authorization URL:	<code>https://api.think.ibm/sales/sb/oauth20/authorize</code>
Token URL:	<code>https://api.think.ibm/sales/sb/oauth20/token</code>
Client ID:	<code>0cd7ff73-2026-4d54-96a1-5086d93ad876</code>
Client Secret:	<code>gL3vE5k50yU0iA6hK2u02qR2pA6tV7wP7yK1nK7sI4gL2tC1bV</code>
Auth Grant Type:	<code>password</code>
Redirect URI:	<code>(undefined)</code>
Scope:	<code>inventory</code>



Important

If you see a **Using Access Token** message, you configured the OAuth authorization correctly.

... authorize and token calls successful.

Using Access Token:

```
AAEkMGNkN2ZmNzMtMjAyNi00ZDU0LTk2YTEtNTA4NmQ5M2FkODc2M37UsgOHiltIvtMl7_wp3oXm23ZhLQ
SCs5ZA05n7QgwGxbcozbdGmY2V4oUvnznkR9fxnjLbZ8IU3dwNqJ-4hQ
```

You can review the following exercise sections, or you can skip ahead to [Section 11.6, "Test the inventory items API operation,"](#) on page 11-26.

11.3. Verify the OAuth 2.0 Provider API configuration

In this section, you verify the OAuth 2.0 Provider API configuration with the API Designer web application. You configured the **/oauth20/authorize** and **/oauth20/token** operations and the OAuth 2.0 security requirement in [Exercise 7, "Declaring an OAuth 2.0 Provider and security requirement"](#).



Important

If you received an **"API not found for the requested URI"** error, either the API gateway is not online, or the **/oauth20/authorize** and **/oauth20/token** operations are not available. Complete the steps in this section to resolve the error.

- ___ 1. Verify that the Xubuntu environment can reach the API gateway.

- ___ a. Open a terminal emulator window.
- ___ b. Send a test message to the API gateway.

```
$ ping api.think.ibm
```

- ___ c. Confirm that the API gateway responds to the `ping` command.

```
PING dp.think.ibm (192.168.225.52) 56(84) bytes of data.  
64 bytes from dp.think.ibm (192.168.225.52): icmp_seq=1 ttl=195  
time=0.541ms
```

- ___ d. Press Ctrl+C to exit the `ping` command.



Note

If you cannot reach the **api.think.ibm** server, check the status of the DataPower gateway in your remote lab environment. Make sure that the virtual machine is running.

- ___ 2. Verify that the API gateway accepts HTTP requests on the **/oauth20/authorize** operations.

- ___ a. Open a terminal emulator window.

- ___ b. Send a test message to the **/oauth20/authorize** operation.

```
$ curl -k https://api.think.ibm/sales/sb/oauth20/authorize
```

- ___ c. Confirm that the **authorize** operation returns a message of **401 unauthorized**.

```
{ "httpCode": "401", "httpMessage": "Unauthorized",  
"moreInformation": "Client Id is missing." }
```



Troubleshooting

If the `curl` command receives a status code of **404 Not Found**, check the **path** and **base path** setting for the **authorize** operation. The **oauth-provider** API in the **inventory** directory defines the **authorize** and **token** operations.

___ 3. Verify the **oauth-provider** API definition.

___ a. Change directory to the **inventory** application.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

___ b. Run **API Designer**.

```
$ apic edit
```

___ c. Open the **oauth-provider** API definition in the editor.

The screenshot shows the API Designer interface with the 'APIs' tab selected. The list of APIs includes:

- financing** 1.0.0 [financing_1.0.0.yaml]
- inventory** 1.0.0 [inventory.yaml]
- logistics** 1.0.0 [logistics_1.0.0.yaml]
- oauth-provider** 1.0.0 [oauth-provider_1.0.0.yaml] (highlighted with a red box)

___ d. Make sure that the **Base Path** is set to `/oauth20`.

The screenshot shows the API Designer interface with the 'Design' tab selected. The 'Host' section shows the 'Base Path' field set to `/oauth20`, which is highlighted with a red box.

___ e. Verify the **OAuth 2** settings.

- Client type: **Confidential**
- Scope name: **inventory**
- Grants: **Password**
- Identity extraction: **Basic**
- Authenticate application users using: **Authentication URL**
- Authentication URL: **<https://services.think.ibm:1443/auth>**
- Authorize application users using: **Authenticated**

OAuth 2

Client type	Client type Confidential	▼
Scopes	Scope Name inventory	⊕
Description Access to the inventory API		
Grants	<input type="checkbox"/> Implicit <input checked="" type="checkbox"/> Password <input type="checkbox"/> Application <input type="checkbox"/> Access Code	▼
Identity extraction	Collect credentials using Basic	▼
Authentication Authenticate application users using		
Authentication URL https://services.think.ibm:1443/auth		
TLS Profile		
Authorization Authorize application users using		
Authenticated		
Tokens Access tokens		
Time to live (seconds) 3600		
<input checked="" type="checkbox"/> Enable refresh tokens		
Count 2048		
Time to live (seconds) 2682000		
<input checked="" type="checkbox"/> Enable revocation		

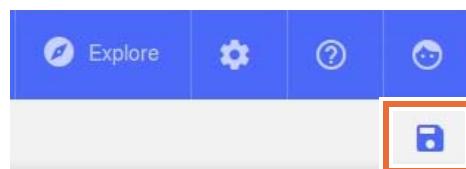
- ___ 4. Verify the paths for the **authorize** and **token** operations.
- ___ a. In the **oauth-provider** API definition, select the **paths** category.
 - ___ b. Select the **/authorize** path.
 - ___ c. Make sure that the **/authorize** path lists two operations: **GET /authorize** and **POST /authorize**.

The screenshot shows the API Designer interface with the 'Design' tab selected. On the left, a sidebar lists categories: Produces, Lifecycle, Policy Assembly, Security Definitions, Security, Extensions, Properties, Paths, /authorize, /token, Parameters, and Definitions. The '/authorize' item is highlighted with a red box. On the right, under the 'Paths' section, the path '/authorize' is listed. Below it, the 'Parameters' section shows 'No parameters defined'. Under the path, there are two operations: 'GET /authorize' and 'POST /authorize', both highlighted with a red box.

- ___ d. Select the **/token** path.
- ___ e. Make sure that the **/token** path lists one operation: **POST /token**.

The screenshot shows the API Designer interface with the 'Design' tab selected. On the left, a sidebar lists categories: Security, Extensions, Properties, Paths, /authorize, /token, Parameters, and Definitions. The '/token' item is highlighted with a red box. On the right, under the 'Paths' section, the path '/token' is listed. Below it, the 'Parameters' section shows 'No parameters defined'. Under the path, there is one operation: 'POST /token', highlighted with a red box.

- ___ 5. If you made any corrections to the **oauth-provider** API, save your changes.



__ 6. Verify the OAuth security definition in the **inventory** API.

__ a. Click **All APIs**.

__ b. In the **API** list, select **inventory**.

Title	Last Modified
financing 1.0.0 [financing_1.0.0.yaml]	7 hours ago
inventory 1.0.0 [inventory.yaml]	7 hours ago
logistics 1.0.0 [logistics_1.0.0.yaml]	7 hours ago
oauth-provider 1.0.0 [oauth-provider_1.0.0.yaml]	7 hours ago

__ c. Select **Security Definitions** from the list of API properties.

__ d. Locate the **OAuth** security definition in the **Security Definitions** section.

__ e. Verify the **OAuth** security settings.

- Name: oauth
- Description: OAuth authorization settings for inventory API
- Flow: Password
- Token URL: <https://api.think.ibm/sales/sb/oauth20/token>
- Scope name: inventory
- Scope description: Access to inventory API operations

The screenshot shows the 'Design' tab selected in the top navigation bar. On the left, a sidebar lists various API metadata sections like Info, Schemes, Host, etc., with 'Security Definitions' highlighted by a red box. The main panel displays the configuration for the 'oauth (OAuth)' security definition. It includes fields for Name (* oauth), Description (OAuth authorization settings for inventory API), Flow (Password), Token URL (https://api.think.ibm/sales/sb/oauth20/token), and Scopes (Scope Name: inventory, Description: Access to inventory API operations). A red box highlights the entire configuration area.

__ f. Select **Security**.

__ g. Confirm that the **OAuth** security requirement and the **inventory** scope are both **enabled**.

The screenshot shows the 'Policy Assembly' tab selected in the top navigation bar. On the left, a sidebar lists 'Security Definitions' and 'Security' (highlighted by a red box). The main panel shows the 'Security' configuration with a descriptive text: 'Define security requirements for the API. Multiple alternative sets can be defined, any one of which can be satisfied to access the API.' Below this, 'Option 1' is selected, showing three checked requirements: 'oauth (OAuth)', 'inventory', 'clientIdHeader (API Key)', and 'clientSecretHeader (API Key)'. A red box highlights the 'oauth (OAuth)' and 'inventory' checkboxes.

__ 7. Review the security settings for the **/inventory/items** API operation.

__ a. In the **inventory** API definition, select the **/items** path.

- __ b. Select the **GET /items** operation.

The screenshot shows the API management interface. On the left, a sidebar lists various API paths under 'Paths'. The path '/items' is highlighted with a red box. On the right, the main panel shows the details for the '/items' endpoint. The path is listed as '/items' with an 'Add Operation' button. Below it, the 'Parameters' section indicates 'No parameters defined'. Under the operations section, there are four entries: 'POST /items', 'PUT /items', 'PATCH /items', and 'GET /items'. The 'GET /items' entry is also highlighted with a red box.

- __ c. Scroll down to the **Security** settings for the **GET /items** operation.
 __ d. Make sure that the **Use API security definitions** setting is **enabled**.

The screenshot shows the security settings for the '/items' endpoint. In the 'Properties' section on the left, the '/items' path is highlighted with a red box. In the 'Responses' section on the right, a table shows a single row for status code 200 with description 'Request was succ' and schema 'array'. In the 'Security' section, there is a checkbox labeled 'Use API security definitions' which is checked and highlighted with a red box.

- __ 8. If you made any corrections to the **inventory** API, save your changes.



11.4. Publish API definition changes to the API Manager

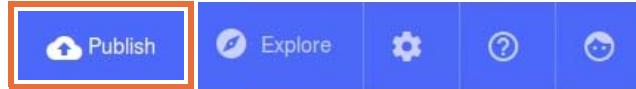
If you saved changes to the **oauth-provider** or **inventory** API definitions, you must publish your changes to the API Manager server.



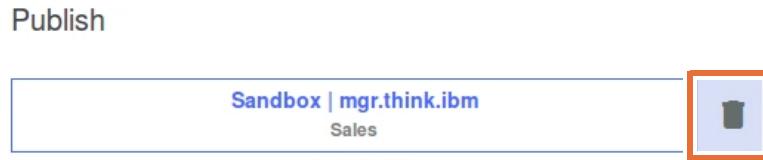
Important

If you *did not make any corrections*, skip ahead to [Section 11.5, "Verify the consumer application credentials and plan subscription,"](#) on page 11-20.

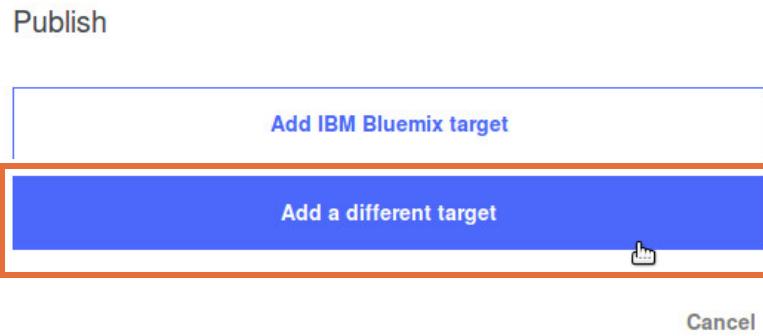
- ___ 1. Sign in to the API Management server as a publish target.
 - ___ a. In the API Designer page, click **Publish**.



- ___ b. Select **Add and manage targets**.
- ___ c. Delete the **Sandbox** catalog target that you created in a previous exercise.



- ___ d. In the Publish wizard, click **Add a different target**.



- ___ e. Sign in to the API Connect Management server with the following credentials:
- Host address: **mgr.think.ibm**
 - User name: **student@think.ibm**
 - Password: **Passw0rd!**

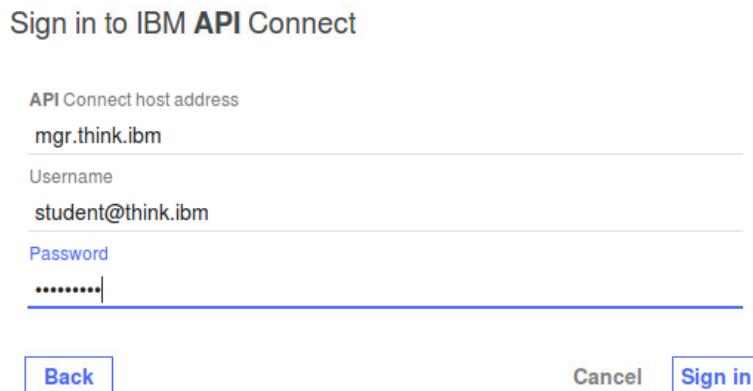
Sign in to IBM API Connect

API Connect host address
mgr.think.ibm

Username
student@think.ibm

Password

[Back](#) [Cancel](#) [Sign in](#)



- ___ f. Click **Sign in**.
- ___ 2. Select the API Connect organization, catalog, and API application.
- ___ a. In the “Select an organization and catalog” page, select the **Sales** organization.
 - ___ b. Select the **Sandbox** catalog.

Select an organization and catalog

Organization
Sales

Search

None

Sandbox



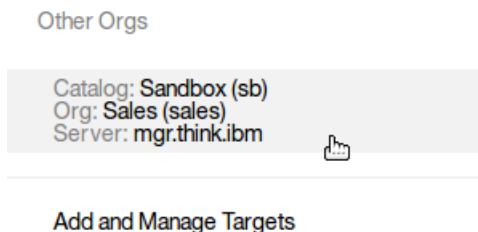
- ___ c. Click **Next**.

- ___ d. In the “Select an app” page, choose **None**.

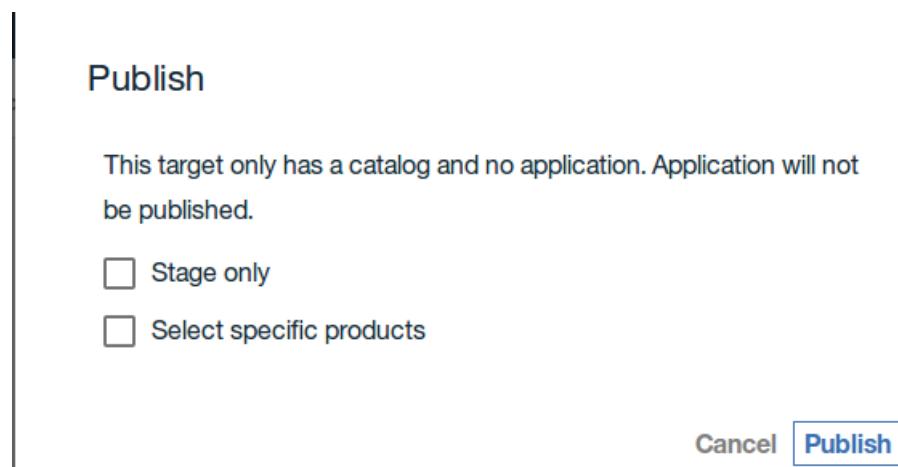
Select an App



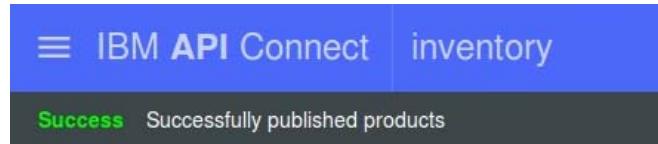
- ___ e. Click **Save**.
- ___ 3. Publish the API product to the API Connect Cloud.
- ___ a. Click **Publish** to open the publish target menu.
- ___ b. Select the publish target with the following characteristics:
- Catalog: **Sandbox (sb)**
 - Org: **sales (sales)**
 - Server: **mgr.think.ibm**



- ___ c. In the **Publish** page, clear all options.



- ___ d. Click **Publish**.
- ___ 4. Confirm that API Designer successfully published the product to API Manager.



- ___ 5. Close the API Designer application.



Important

Before you proceed to the next section of the exercise, test the corrections that you published to the API Connect Cloud.

Run the **consumer** application and test the OAuth 2.0 authorize and token operations according to the steps in [Section 11.2, "Test the OAuth authorize and token API operations," on page 11-4](#).

11.5. Verify the consumer application credentials and plan subscription

In this section, you verify the consumer application **client ID** and **client secret** values. The two values act as the user name and password for the consumer application: it authenticates the identity of the application that calls the inventory API.

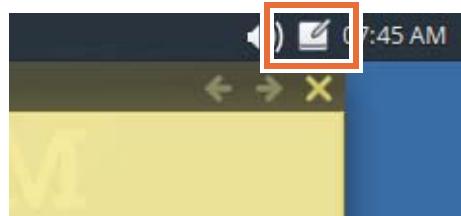
You registered the consumer application and saved the client ID and client secret values in [Exercise 10, "Subscribing and testing APIs"](#).



Important

If you received a “Client ID unauthorized” error in the consumer application, complete the steps in this section to verify the client ID and client secret values for the application. In addition, check the plan subscription for the application in the Developer Portal.

- 1. Review the Client ID and Client secret values that you saved in the **publish** exercise.
 - a. Open the Notepad application in the system menu bar.



- b. Locate the **Client ID** and **Client secret** values that you saved in the Notepad application. You must pass the values that correspond to the application that you registered in the Developer Portal.

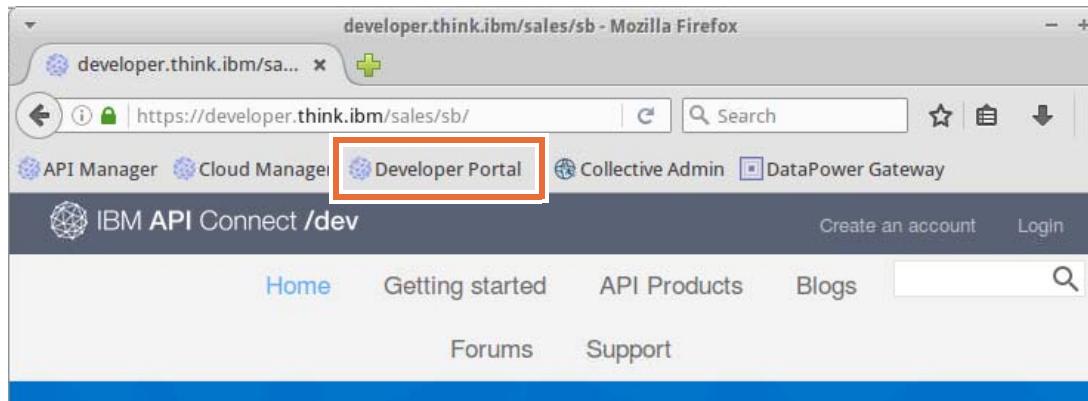
```
Client ID:  
0cd7ff73-2026-4d54-96a1-5086d93ad876  
  
Client secret:  
gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV
```



Note

The Developer Portal generates random values for the **Client ID** and the **Client secret**. Your Client ID and Client secret values are different from the examples that are in this exercise.

- ___ 2. Open the consumer application entry in the Developer Portal.
- ___ a. In the Mozilla Firefox browser, open the Developer Portal at:
<https://developer.think.ibm/sales/sb>



- ___ b. Click **Log in** from the upper-right section of the page.
- ___ c. Enter the following credentials:
 - User name: **developer@think.ibm**
 - Password: **Passw0rd!**

The screenshot shows the "User login" page of the IBM API Connect /dev portal. The header includes "IBM API Connect /dev", "Home", "Getting started", and "API Products". Below the header is a "User login" form. It features three input fields: "Username *", "Password *", and a "Log in" button. The "Username *" field contains "developer@consumer.ibm" and has a placeholder "Enter your developer.think.ibm/sales/sb username.". The "Password *" field contains "*****" and has a placeholder "Enter the password that accompanies your username.". The "Log in" button is located at the bottom left of the form.

- ___ d. Click **Log in**.
- ___ e. Open the list of **apps** that you registered on the portal.

- __ f. Select the **Inventory client application**.

The screenshot shows the IBM API Connect /dev interface. At the top, there is a navigation bar with links for Home, Getting started, API Products, Apps (which is highlighted with a red box), Blogs, Forums, and Support. Below the navigation bar, there is a search bar and a button for '+ Create new App'. The main content area displays a card for the 'Inventory client application'. This card features a circular icon with a smartphone, the text 'Inventory client application' (also highlighted with a red box), and a description: 'A web application to browse items for sale, and to submit reviews.' A red box also highlights the entire card.

- __ 3. Verify that the **Client ID** matches the value that you saved in the Notepad application.
- __ a. In the **Credentials** section, select **Show** in the **Client ID** field.
 - __ b. Confirm that the **Client ID** matches the value that you saved in Notepad.

The screenshot shows the IBM API Connect /dev interface with the 'Credentials' section open. The 'Credentials' tab is selected, and there is a link to 'Add credentials'. Below the tabs, there is a table with one row labeled 'Default'. The 'Client ID' field contains the value '0cd7ff73-2026-4d54-96a1-5086d93ad876', which is highlighted with a red box. To the right of this field is a checkbox labeled 'Show' (which is checked) and a 'Reset' button. Below the 'Client ID' row is another row for 'Client Secret', which contains a grayed-out input field, a 'Verify' button, and a 'Reset' button. A red box highlights the 'Client ID' field and its associated controls.

- __ 4. Verify that the **client secret** matches the value that you saved in the Notepad.
- __ a. In the **Inventory client application** page, click **Verify** in the **Client secret** field.

- ___ b. Copy the client secret from the Notepad application and paste it into the web page.

Verify application secret

The screenshot shows a 'Verify application secret' form with a red box around the 'Secret *' input field containing the value 'gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV'. To the right, a browser window displays the same value in a yellow-highlighted text area, with a context menu open showing 'Copy' selected.

- ___ c. Click **Submit**.
 ___ d. Confirm that the Developer Portal verified the client secret value.

The screenshot shows a success message: 'Application secret verified successfully.' Below it is the 'Inventory client application' page with tabs for 'Analytics', 'Notification settings', and 'Delete'.

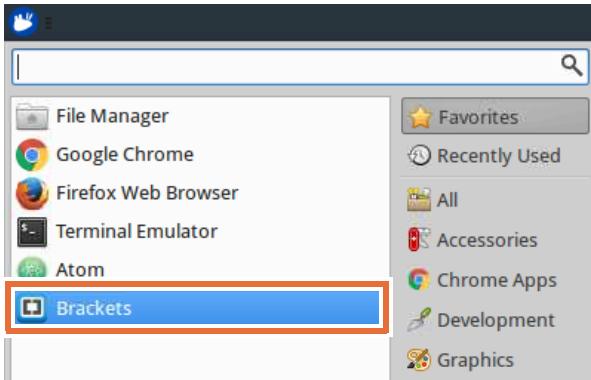
- ___ 5. Verify that the **Inventory client application** is subscribed to the **think API** product.
 ___ a. Review the list of plan subscriptions for the **Inventory client application**.
 ___ b. Make sure that the **think (1.0.0) (silver)** plan appears in the list.

The screenshot shows the 'Inventory client application' details page with a 'Update' button and a 'Description' section stating 'A web application to browse items for sale, and to submit reviews.'

The screenshot shows the 'Subscriptions' section with a red box around the 'think (1.0.0) (silver)' plan entry, which includes 'View Details' and 'Unsubscribe' buttons.

- ___ 6. Log out of the Developer Portal.

- __ 7. Make sure that the **client_ID** and **client_secret** values in the **consumer app** configuration file match the values in the Developer Portal.
 - __ a. Open the **Brackets** editor.



- __ b. Open the consumer application configuration file, at `~/lab_files/devportal/consumer_app/config/default.json`.
- __ c. Make sure that the **client_ID** and **client_secret** properties match the values that you saved in the **Notepad** application.

The screenshot shows the Brackets code editor with the file `devportal/consumer_app/config/default.json` open. The left sidebar shows a file tree with the following structure:

```

lab_files/
  > authorization
  > collectives
  > datasources
  > devportal
    > consumer_app
      > bin
      > config
        default.json
      > public
      > routes
      > server
      > views
      app.js
      npm-debug.log
      npm-shrinkwrap.json
      package.json
  > public
  > routes
  > server
  > views
  app.js
  npm-debug.log
  npm-shrinkwrap.json
  package.json
  
```

The main editor area displays the JSON content of the `default.json` file. A red rectangle highlights the section where the `client_id` and `client_secret` values are defined. The JSON code is as follows:

```

1  {
2   "Application": [
3     "client_id": "0cd7ff73-2026-4d54-96a1-5086d93ad876",
4     "client_secret":
5       "gL3vE5kS0yU0iA6hK2u02qR2pA6tV7wP7yK1nK7sI4gL2tC1bV"
6   },
7   "API-Server": {
8     "protocol": "https",
9     "host": "api.think.ibm",
10    "org": "sales",
11    "catalog": "sb"
12  },
13  "APIs": [
14    "inventory": {
15      "base_path": "/inventory",
16      "require": [
17        "client_id",
18        "client_secret",
19        "oauth"
20      ],
21    }
22  }
  
```

- __ 8. If you made any corrections to the `config/default.json` file, save your changes.
- __ 9. Close the Brackets editor.



Important

Before you proceed to the next section of the exercise, test the corrections that you published to the API Connect Cloud.

Run the **consumer** application and test the OAuth 2.0 authorize and token operations according to the steps in [Section 11.2, "Test the OAuth authorize and token API operations," on page 11-4](#).

11.6. Test the inventory items API operation

In the last section of the troubleshooting exercise, you confirmed that the **authorize** and **token** operations ran successfully. The **consumer** application is authorized to call any operation in the **inventory** API.

In the terminal emulator window, the application log displays an attempt to call the **GET /inventory/items** operation.

```

Initializing OAuth client...
Authorization URL: https://api.think.ibm/sales/sb/oauth20/authorize
Token URL: https://api.think.ibm/sales/sb/oauth20/token
Client ID: 0cd7ff73-2026-4d54-96a1-5086d93ad876
Client Secret: gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV
Auth Grant Type: password
Redirect URI: (undefined)
Scope: inventory

... authorize and token calls successful.

Using Access Token:
AAEkMGNkN2ZmNzMtMjAyNi00ZDU0LTk2YTEtNTA4NmQ5M2FkODc2ECZX7vwMRhp-hakDrvX7xxaQLeE
oeBEmvhHN-JL-gDWBVmQNnvjfF_9r3QNw7DBYCa8rRJRi0GgXL7-69OhRKg
POST /login 302 603.616 ms - 64

Sending API request...
Method: GET
URL:
https://api.think.ibm/sales/sb/inventory/items?filter[order]=name%20ASC
Headers:
X-IBM-Client-Id: 0cd7ff73-2026-4d54-96a1-5086d93ad876
X-IBM-Client-Secret: gL3vE5kS0yU0iA6hK2uO2qR2pA6tV7wP7yK1nK7sI4gL2tC1bV
Authorization: Bearer
AAEkMGNkN2ZmNzMtMjAyNi00ZDU0LTk2YTEtNTA4NmQ5M2FkODc2_NeRQVlrJ9zCQ711u4t8tx1Gpc-
y-audX-tXK85ZWBU8PfgsfIBJHH4Mp1QZlubP7pBOeeAcQPMbY2wHc0F3xQ

```

After the **consumer** application receives an **access token**, it sends an HTTP **GET** request to the **/inventory/items** operation. The full path for the API operation is <https://api.think.ibm/sales/sb/inventory/items>. The HTTP query parameter of **filter[order]=name%20ASC** returns the list of items in ascending alphabetical order.

The consumer application sends three HTTP headers in the API request:

- The **X-IBM-Client-Id** and **X-IBM-Client-Secret** headers contain the client ID and client secret for the **consumer** application. The two headers satisfy the API key and secret security requirements that you defined in the **inventory** API definition.
- The **Authorization** header stores the OAuth access token value. You must send a valid access token for all API operations that you secured with the **OAuth** security requirement.

The API request flows through the architecture that you built from **Exercise 4** to **Exercise 9**.

1. The **consumer** application calls the **/inventory/items** operation at **api.think.ibm**, the DataPower API Gateway.
2. The **Docker container instance** runs a copy of the **inventory** LoopBack application. It processes the **GET /inventory/items** operation call, and returns a list of inventory items.



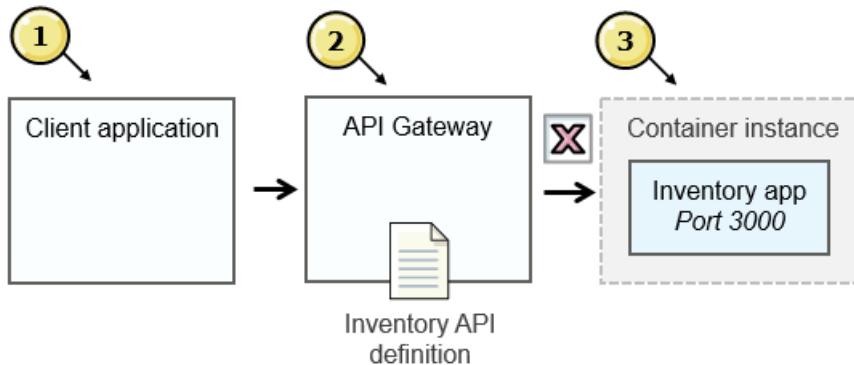
Troubleshooting

If you receive an "**Inventory API call failed**" message with a status code of **500**, the API gateway failed to establish a backside connection.

```
... API request failed on server.  
Status code:      500  
Message:          Inventory API call failed  
More information: Failed to establish a backside connection
```



The failure occurs between the API gateway and the Docker container runtime instance.



You must verify the **invoke** policy in the **inventory** API definition. In addition, check that the Docker container that runs the application is started.

- Make sure that the **invoke** policy calls `$(app-server)$(request.path)$(request.search)`.
 - Make sure that the `$(app-server)` environment property resolves to `http://inventory.think.ibm:3000`.
 - Make sure that the Docker container is started.
-

___ 1. Open the **inventory** API definition.

___ a. In the terminal emulator, change directory to the **inventory** application.

```
$ cd ~/inventory
$ pwd
/home/student/inventory
```

___ b. Start the API Designer application.

```
$ apic edit
```

___ c. In the **APIs** listing, open the **inventory** API definition.

Title	Last Modified
financing 1.0.0 [financing_1.0.0.yaml]	7 hours ago
inventory 1.0.0 [inventory.yaml]	7 hours ago
logistics 1.0.0 [logistics_1.0.0.yaml]	7 hours ago
oauth-provider 1.0.0 [oauth-provider_1.0.0.yaml]	7 hours ago

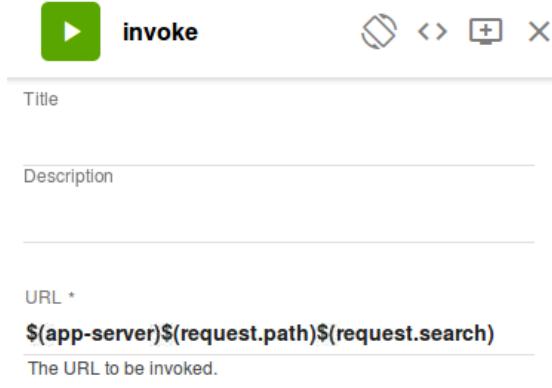
___ 2. Make sure that the **invoke** policy calls the application on the Docker container.

___ a. In the **inventory** API definition, click the **Assemble** tab.

___ b. Confirm that two policies appear in the canvas: **activity-log** and **invoke**.



- ___ c. Select the **invoke** policy.
- ___ d. Verify the properties for the **invoke** policy.
 - URL: **`$(app-server)$(request.path)$(request.search)`**



- ___ e. Close the **invoke** property editor.
- ___ 3. Make sure that the **\$(app-server)** environment property is set to **http://inventory.think.ibm:3000**.
- ___ a. Click the **Design** view.

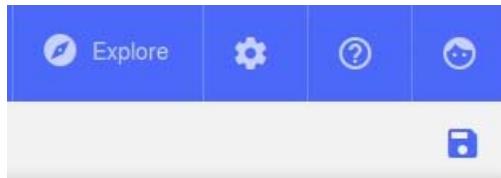


- ___ b. Select the **Properties** section.
- ___ c. Select **app-server** to open the property details.

- ___ d. Make sure that the **Default** value is set to `http://inventory.think.ibm:3000`.

Catalog	Value
Default	<code>http://inventory.think.ibm:3000</code>

- ___ 4. If you made any corrections to the **inventory** API, save your changes.



- ___ 5. Ensure that the Docker container is started.

- ___ a. Open a terminal emulator window.
___ b. Confirm that the Docker container is started.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED
STATUS              PORTS              NAMES
8e228c99c6da      apic-inventory-image   "npm start"   14 seconds
ago                Up 13 seconds      0.0.0.0:3000->3000/tcp friendly_swartz
```

- ___ c. If it is not started, start the Docker container.

```
$ docker run --add-host mysql.think.ibm:192.168.225.10 --add-host
mongo.think.ibm:192.168.225.10 -p 3000:3000 apic-inventory-image

> inventory@1.0.0 start /inventory
> node .
Web server listening at: http://localhost:3000
```

**Important**

If you *made any corrections* to the **inventory** API definition, you must publish your updates to the API Manager. Complete the publish instructions in [Section 11.4, "Publish API definition changes to the API Manager,"](#) on page 11-16.

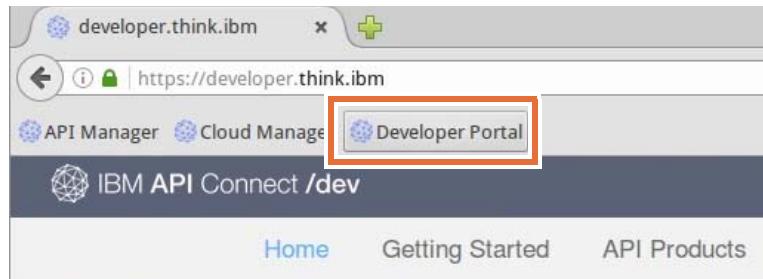
If you *did not make any changes*, proceed to the next section of this exercise.

11.7. Test the inventory API in the test client of the Developer Portal

In the last section, you verified that the **consumer** application passed **OAuth 2.0 authorization** successfully. You also verified the message processing policies for the **inventory** API at the API gateway.

In this section, test the inventory APIs from the **think** product in the Developer Portal test client. Test the **GET /items** operation from the **inventory** API.

- ___ 1. Open the Developer Portal page.
 - ___ a. Open a web browser from the start menu.
 - ___ b. Open the Developer Portal page at <https://developer.think.ibm/sales/sb/>.



- ___ 2. Log in to the Developer Portal with the **developer@consumer.ibm** account.
 - ___ a. Click the **Login** link.
 - ___ b. Enter the user credentials for the application developer account.
 - User name: **developer@consumer.ibm**
 - Password: **Passw0rd!**Click **Log in**.
- ___ 3. Open the test client with the **think 1.0.0** API product.
 - ___ a. Select the **API Products** section of the Developer Portal.

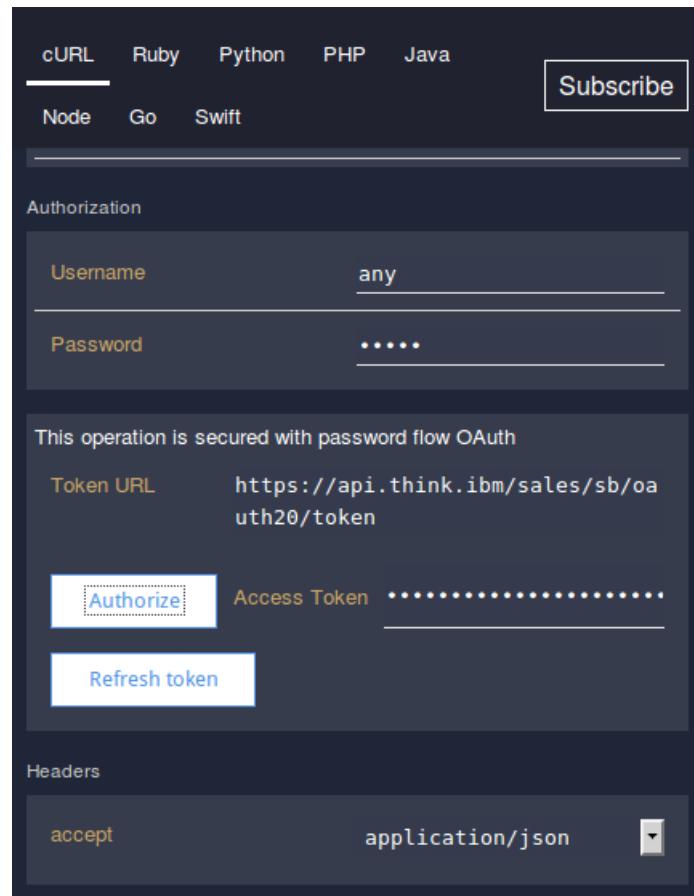
- ___ b. In the think 1.0.0 product, select the **Inventory** entry.

- ___ 4. Test the **GET /items** operation in the **inventory** API.

- ___ a. Select the **GET /items** operation from the left column.
 ___ b. In the rightmost column, scroll down to the **Try this operation** section.

- ___ c. The Inventory client application is already prefilled in the Client ID.
 ___ d. Copy the client secret from the Notes application (Ctrl+C) and paste it (Ctrl+V) into the value for the client secret in the test client.
 ___ e. Type any values into the Username and Password fields in the test client.

__ f. Click **Authorize** in the test client.



__ g. Click **Call Operation**.

h. You see the response 200 OK in the response area.

The screenshot shows a REST test client interface. At the top, there are tabs for cURL, Ruby, Python, PHP, Java, Node, Go, and Swift, with cURL selected. To the right is a 'Subscribe' button. Below the tabs, the request URL is shown as 'GET https://api.think.ibm/sales/sb/inventory/items'. The request headers include 'Authorization: Bearer', 'X-IBM-Client-Id: d53ce870-3d74-4c0c-b46c-0571679b6e2b', 'X-IBM-Client-Secret:', 'accept: application/json', and 'Content-Type: application/json'. The response section shows the status '200 OK' and the response body, which includes headers like 'Content-Type: application/json; charset=utf-8', 'X-Global-Transaction-ID: 196c556559ee52720193f1e0', 'X-RateLimit-Limit: name=rate-limit,100', 'X-RateLimit-Remaining: name=rate-limit,98', and a JSON array of items: [{"name": "Dayton Meat Chopper", "description": "Punched-card tabulating machines and time"}].

You have successfully tested the inventory API in the test client.

i

Information

An error might be displayed when you click **Authorize** for the access token.

[cURL](#) [Ruby](#) [Python](#) [PHP](#) [Java](#) [Node](#) [Go](#) [Swift](#) [Subscribe](#)

Authorization

Username	any
Password	*****

This operation is secured with password flow OAuth

Token URL <https://api.think.ibm/sales/sb/oauth20/token>

[Authorize](#) [Access Token](#)

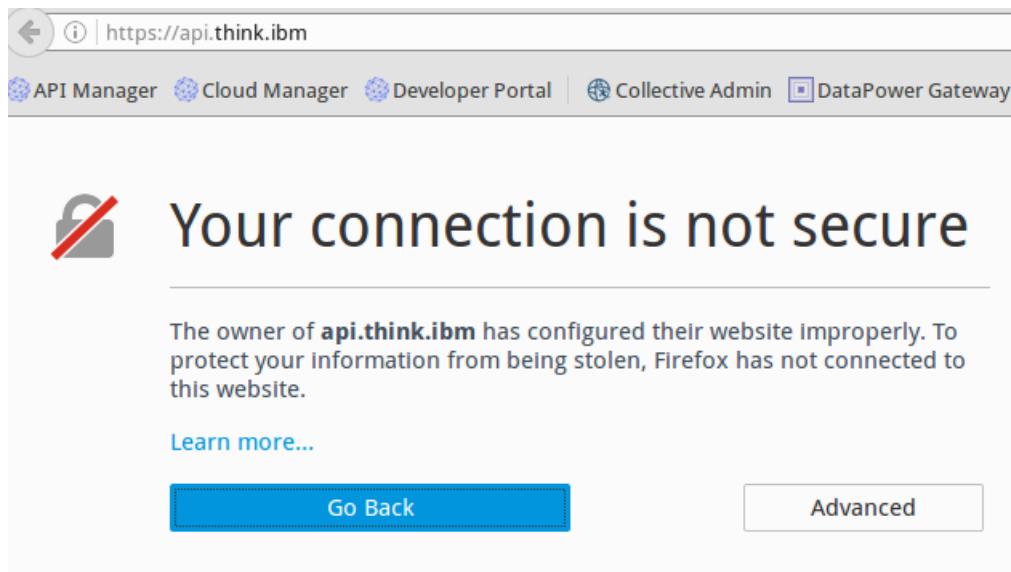
Error. An error has occurred. See browser developer tools for more details.

- 1) Open the browser developer tools.

- 2) The browser developer tools display an error with the self-signed certificate.

Headers	Cookies	Params	Response	Timings	Security
An error occurred:					
api.think.ibm uses an invalid security certificate.					
The certificate is not trusted because it is self-signed.					
The certificate is only valid for API Connect					
Error code: SEC_ERROR_UNKNOWN_ISSUER					

- 3) In another browser tab, type `https://api.think.ibm` in the address area. Then, click **Advanced**.

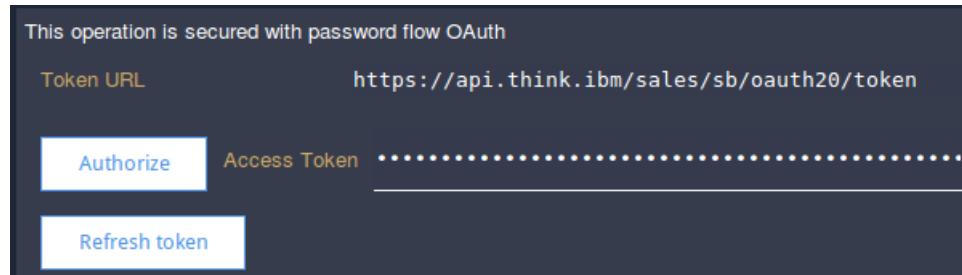


- 4) Click **Confirm Security Exception**.



- 5) Return to the Developer Portal by clicking the tab in the browser.

- 6) Click the **Authorize** tab in the Developer Portal to retry the GET /items operation. You see that the test client of the Developer Portal inserts something into the access token field.



- 7) Continue the test of the GET /items operation in the Developer Portal.

-
- 5. When you have successfully tested the inventory GET /items operation, sign out of the Developer Portal and close the browser.

11.8. Test the inventory API with the consumer application

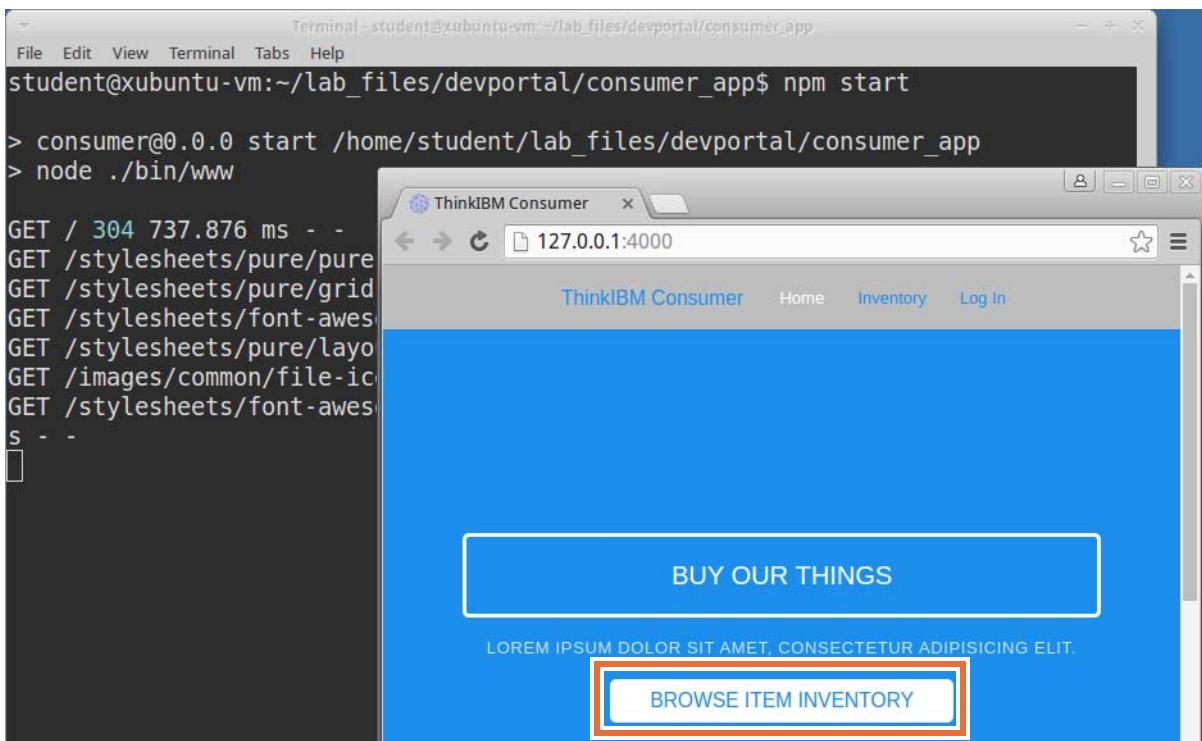
In the last section, you verified that the **inventory** application works on the test client of the Developer Portal.

Complete an end-to-end test with the **consumer** application.

- ___ 1. Open a terminal emulator window.
- ___ 2. Run the consumer application.
 - ___ a. Open a terminal emulator window.
 - ___ b. Change directory to `~/lab_files/devportal/consumer_app`.

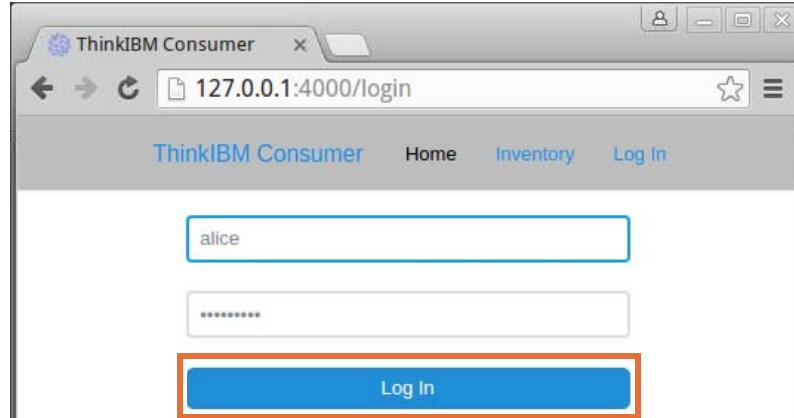

```
$ cd ~/lab_files/devportal/consumer_app
$ pwd
/home/student/lab_files/devportal/consumer_app/
```
 - ___ c. Run the application.


```
$ npm start
```
- ___ 3. Test the `/inventory/items` API operation with the **consumer** application.
 - ___ a. Click **BROWSE ITEM INVENTORY**.



___ b. Log in with the following credentials:

- User name: **alice**
- Password: **Passw0rd!**



___ c. Click **Log in**.

___ d. Confirm that the application displays a list of **inventory items** that are sorted in **ascending alphabetical order**.

```

Terminal - student@xubuntu-vm:~/lab_files/devportal/consumer
File Edit View Terminal Tabs Help
GET /stylesheets/pure/Layouts/main.css 304 4.411 ms -
GET /stylesheets/pure/layouts/login.css 304 4.696 ms -
GET /javascripts/calc-shipping.js 304 4.858 ms -
Using OAuth Token: AA1kZDUzY2U4NzAtM2Q3NC00YzBjLWI0NmMtMDU3MT
tgOPFa4PJSW0quFHoRE8gi90xHnfmwvUss3caZ2o50a6ECUIDkslrSRE0gu
POST /login 302 267.674 ms - 64
MY OPTIONS:
{"method":"GET","url":"https://api.think.ibm/sales/sb/inven
er=name%20ASC","strictSSL":false,"headers":{"X-IBM-Client
-0c-b46c-0571679b6e2b","X-IBM-Client-Secret":"W5wS7tY1nA2vG0
i7tt3w08fL7xE3","Authorization":"Bearer AA1kZDUzY2U4NzAtM2
3MTY30WI2ZTJiAwzPm9YYtgOPFa4PJSW0quFHoRE8gi90xHnfmwvUss3c
gucsWIM090fPzMgCgvw"}}
GET /inventory 200 456.900 ms - 10785
GET /stylesheets/pure/pure-min.css 304 3.803 ms -
GET /stylesheets/pure/grids-responsive-min.css 304 5.595 ms
GET /stylesheets/font-awesome/font-awesome.css 304 5.875 ms
GET /stylesheets/pure/layouts/main.css 304 5.250 ms -
GET /stylesheets/pure/layouts/inventory.css 200 9.658 ms -
GET /javascripts/inventory-sort.js 200 9.741 ms - 647
GET /images/items/608-calculator.jpg 200 11.231 ms - 35920
GET /images/items/collator.jpg 200 10.222 ms - 26927
GET /images/items/meat-chopper.jpg 200 10.602 ms - 54219
GET /images/items/electric-card-punch.jpg 200 7.887 ms - 53

```

___ 4. Test the **GET /inventory/items/{id}** operation.

___ a. Select the **Calculator** item in the inventory page.

- ___ b. Review the details for the item.

The terminal window shows the following log output:

```

Terminal - student@xubuntu: ~
File Edit View Terminal Tabs Help
GET /images/items/electric-card-punch.j...
GET /images/items/summary-punch.j...
GET /images/items/hollerith-tabula...
GET /images/items/01-typewriter.j...
GET /images/items/a-typewriter.j...
GET /images/items/803-proof.jpg 304 ...
GET /images/items/selectric.jpg 304 ...
GET /images/items/tape-controlled.j...
GET /item/8 200 427.611 ms - 5459...
GET /stylesheets/pure/pure-min.cs...
GET /stylesheets/pure/grids-respo...
GET /stylesheets/font-awesome/fon...
GET /stylesheets/pure/layouts/main...
GET /images/items/608-calculator.j...
GET /javascripts/calc-shipping.js
GET /stylesheets/pure/layouts/item...
GET /javascripts/calc-financing.j...
GET /item/8 304 285.010 ms - ...
GET /stylesheets/pure/pure-min.cs...
GET /stylesheets/pure/grids-respo...
GET /stylesheets/font-awesome/fon...
GET /stylesheets/pure/layouts/main...
GET /stylesheets/pure/layouts/item...
GET /javascripts/calc-shipping.js

```

The web browser window displays the product details for the "Calculator" item. It includes a thumbnail image of the IBM 608 calculator, its name, a star rating, its price (\$5199.99), and a "Calculate Monthly Payment" button. To the right, there is a "Calculate Shipping" section with a "Zip Code" input field containing "98121" and a "Calculate" button. Below the product details, there is a "Product Description" section with text about the IBM 608 calculator.

- ___ 5. Test the **GET /logistics/shipping** operation.

- ___ a. In the **Calculate shipping** form, enter **98121** in the Zip Code field.

The screenshot shows the "Calculate Shipping" section of the application. The "Zip Code" input field contains the value "98121", which is highlighted with a red rectangular border. Below the input field is a blue "Calculate" button.

- ___ b. Click **Calculate**.

- __ c. Review the estimated shipping options from the **logistics API**.

The terminal window on the left shows a series of GET requests being processed by a server. The web browser window on the right displays a product page for a 'Calculator'. The product image is a black and white photograph of a vintage-style calculator. The product title is 'Calculator' and it has a rating of 5 stars. The price is listed as '\$5199.99'. Below the product details is a blue button labeled 'Calculate Monthly Payment'. To the right of the product information is a sidebar titled 'Calculate Shipping' which lists various shipping options with their costs:

Carrier	Service	Cost
XYZ	Next Day	\$39.36
XYZ	Two Day	\$26.93
XYZ	Ground	\$20.72
CEK	Next Day	\$35.78
CEK	Two Day	\$24.48
CEK	Ground	\$18.83

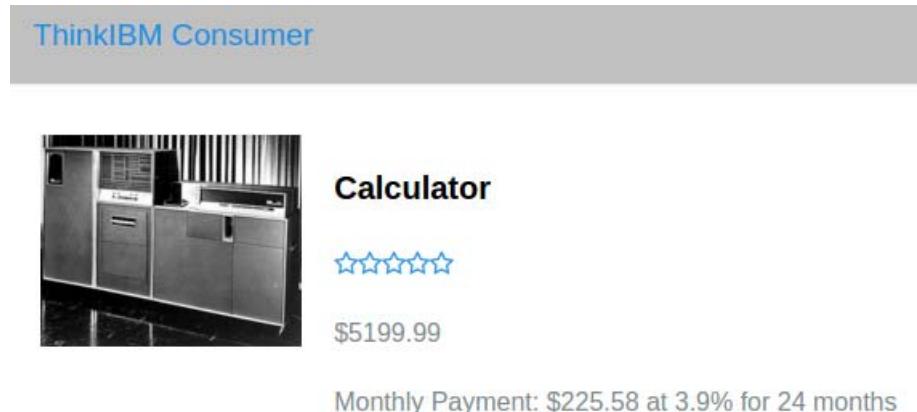
At the bottom of the sidebar, there is a link 'Pickup from nearest store'.

- __ 6. Click **Calculate Monthly Payment**.



[Calculate Monthly Payment](#)

- ___ 7. The monthly payment is displayed.



- ___ 8. Close the web browser.
___ 9. Press Ctrl+C in the terminal emulator to stop the **consumer** application.

End of exercise

Exercise review and wrap-up

In the first part of the exercise, you verified the **OAuth 2.0 authorization** flow. You reviewed the settings for the **/oauth20/authorize** and **/oauth20/token** API operations.

In the second part of the exercise, you verified the **/inventory/items** API operation by calling the **inventory** API that was running in a Docker container.

Appendix A. Solutions

The **lab files** directory (`/home/student/lab_files`) contains source code and configuration files for the WD500 lab exercises. Each subdirectory in the lab files directory represents an exercise in the course.

Within each exercise subdirectory, the **complete** folder contains a copy of the model solution for the lab. For example, the solution for Exercise 1, “Creating and publishing an API in API Designer” is located in `/home/student/lab_files/interface/complete`.

If you are unable to complete a particular lab exercise, back up your current work. Copy the model solution application project into your home directory. For complete instructions on how to set up a model solution for a lab exercise, refer to the `README.md` file in the **complete** directory.

Table 9. Model solution file for course exercises

Exercise name	Completed exercise solution
Exercise 1, "Creating and publishing an API in API Designer"	<code>~/lab_files/interface/complete</code>
Exercise 2, "Defining an API that calls an existing SOAP service"	<code>~/lab_files/existing/complete</code>
Exercise 3, "Creating a LoopBack application"	<code>~/lab_files/loopback/complete</code>
Exercise 4, "Defining LoopBack data sources"	<code>~/lab_files/datasources/complete</code>
Exercise 5, "Implementing event-driven functions with remote and operation hooks"	<code>~/lab_files/remote/complete</code>
Exercise 6, "Assembling message processing policies"	<code>~/lab_files/policies/complete</code>
Exercise 7, "Declaring an OAuth 2.0 Provider and security requirement"	<code>~/lab_files/authorization/complete</code>
Exercise 8, "Deploying an API implementation to a container runtime"	<i>Not applicable</i>
Exercise 9, "Defining and publishing an API product"	<code>~/lab_files/publish/complete</code>
Exercise 10, "Subscribing and testing APIs"	<i>Not applicable</i>
Exercise 11, "Troubleshooting the case study"	<i>Not applicable</i>



IBM Training



© Copyright International Business Machines Corporation 2017.