API Economy: building your APIs with Azure API Management and API App

AUTHORS – Chun Liu

Table of Contents

Summary

Azure API Management (APIM) helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services. Businesses everywhere are looking to extend their operations as a digital platform, creating new channels, finding new customers and driving deeper engagement with existing ones. API Management provides the core competencies to ensure a successful API program through developer engagement, business insights, analytics, security, and protection.

This document is a lab manual which consists of step by step tutorials to show you how to use APIM to manage the web APIs of an Azure API App. The lab is focused on **configuring** Azure API Management and API App rather than **developing** the web APIs, although it involves an exercise of building an API App with ASP.NET Core 2.0.

Objectives

As a result of working through this lab, you will be better able to:

- Understand the concepts and capabilities of Azure API App and API Management
- Understand how to configure and manage APIs with API Management
- Complete tasks with the Publisher and Developer portals of API Management
- Customize the behavior of APIs with policies of API Management

Prerequisites

There are no specific prerequisites to complete the lab. However, you would be able to complete the lab more efficiently if you have experiences in the following areas.

- Familiar with Azure portal and Azure Active Directory
- Familiar with Visual Studio and ASP.NET
- Basic understanding of Azure App Service and API Management

Exercise 1: Build and deploy the Web APIs

In this exercise, we will create an ASP.NET Core 2.0 Web API project, and deploy it to an Azure API App. The Web API that we build is based on the TodoApi that is described in the tutorial, Create a web API with ASP.NET Core and Visual Studio for Windows.

As the lab is focused on the configuration and management of Azure API Management and API App rather than the coding of Web APIs, also for the sake of time, we will get the completed code from the GitHub repository in the following exercises. If you want to know how to create the project from scratch, or the details of the code, please refer to the above link.

1.1 Clone the GitHub repository and build the solution locally

Please follow the steps below to get the code of the Web API from GitHub and build it in Visual Studio 2017.

1. In the lab machine, click the **Command Prompt** icon on Windows task bar to launch the Command Prompt.

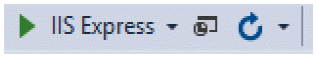
2. In the Command Prompt window, switch the work directory to **C:\Labfiles** by typing the following command:

```
cd c:\labfiles
```

3. Type the following command to clone the repository from GitHub:

```
git clone https://github.com/chunliu/apimlab.git
```

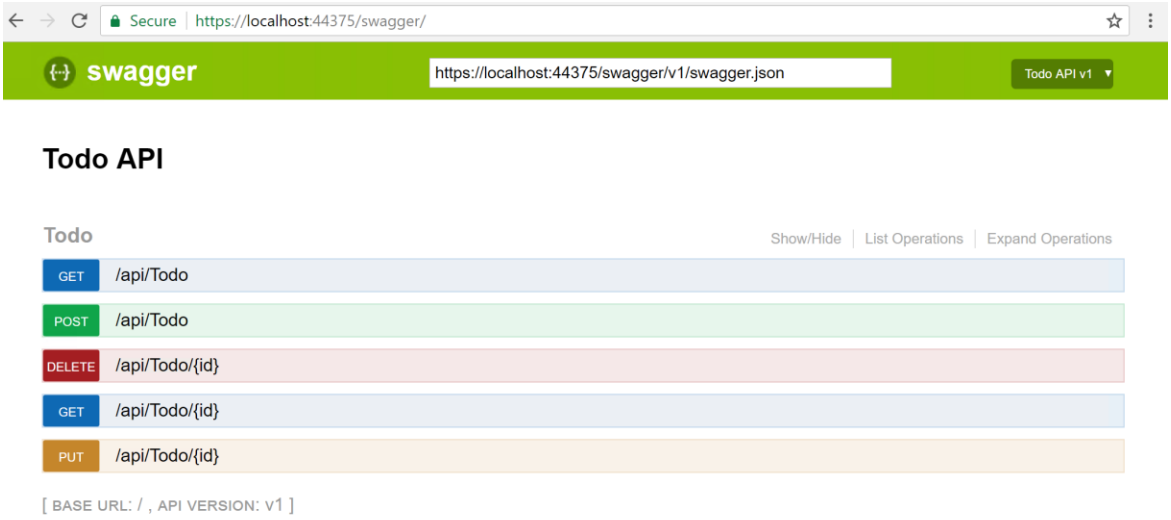
4. When the clone completes, the output in Command Prompt looks like something similar with the following screenshot, and a new folder, **apimlab**, is created in C:\Labfiles.

5. Close the **Command Prompt** window.
6. Open Windows Explorer and locate to the folder: **C:\Labfiles\apimlab\src\TodoApi**.
7. Double click **TodoApi.sln** to launch the project in Visual Studio 2017.
8. In Visual Studio 2017, press **F5** or click **IIS Express** on the toolbar 

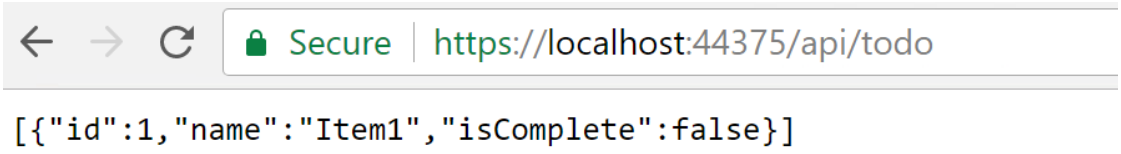
to launch the Web API.

> Visual Studio 2017 will have to restore all the Nuget packages used by the solution and build it. The build could take several seconds to minutes to complete.

9. When the Web API is built and launched, you would see something similar with the following screenshot in the browser. This is the Open API/Swagger help page of our web APIs.



10. If you browse to the URL: **https://localhost:44375/api/todo**, you would see a JSON response of a todo item list, like the following screenshot.



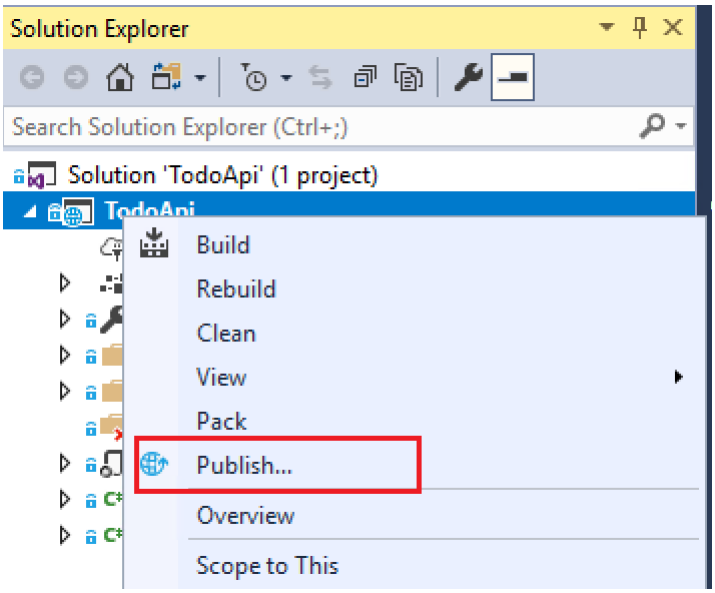
If you can see both the swagger page and the JSON response, the Web API has been built successfully on your lab machine, and you have completed the exercise 1.1 successfully.

You can also browse each of operations on the swagger page to see its definition.

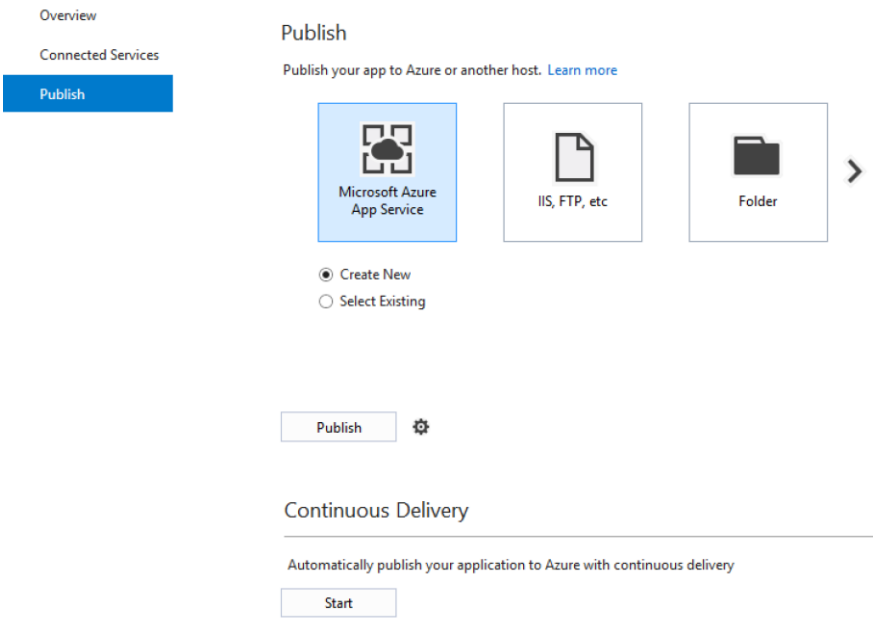
1.2 Publish the Todo API to an Azure API App

When the code can be built and launched successfully on your lab machine, it is ready to be deployed to an Azure API App. Please follow the steps below to publish it from Visual Studio 2017.

1. Close all browser windows.
2. In Visual Studio 2017, if it is still in the debugging mode, stop the debugging by clicking the stop button on the toolbar  .
3. In the **Solution Explorer** window of Visual Studio, **right click** on the project name TodoApi and click **Publish** on the popup menu, as shown in the following screenshot.



4. On the publish configuration page, select **Microsoft Azure App Service** and **Create New** option, and then click **Publish** button, as shown below.

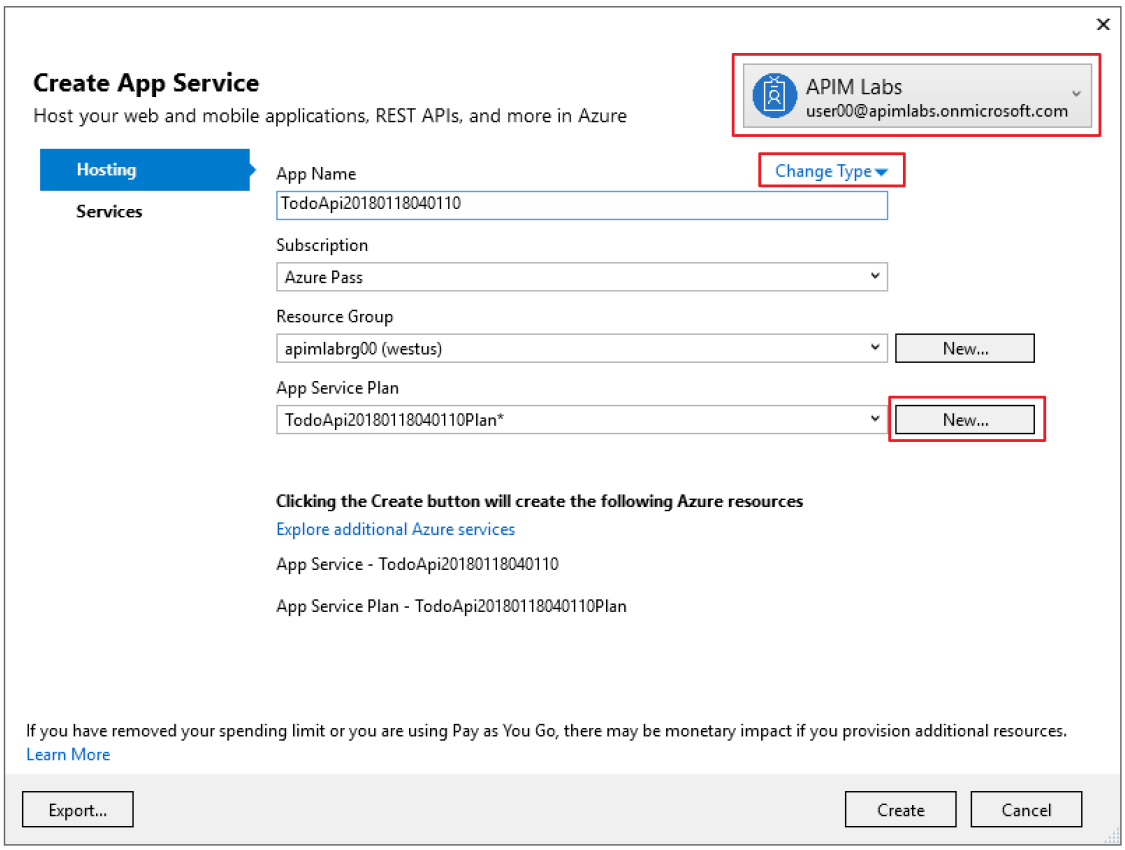


5. On the popup window, click **Add an account** button on the upper left corner, and click **Add an account** from the dropdown list. Sign in with the assigned user name and password.

6. Click **Change Type** and choose **API App**.

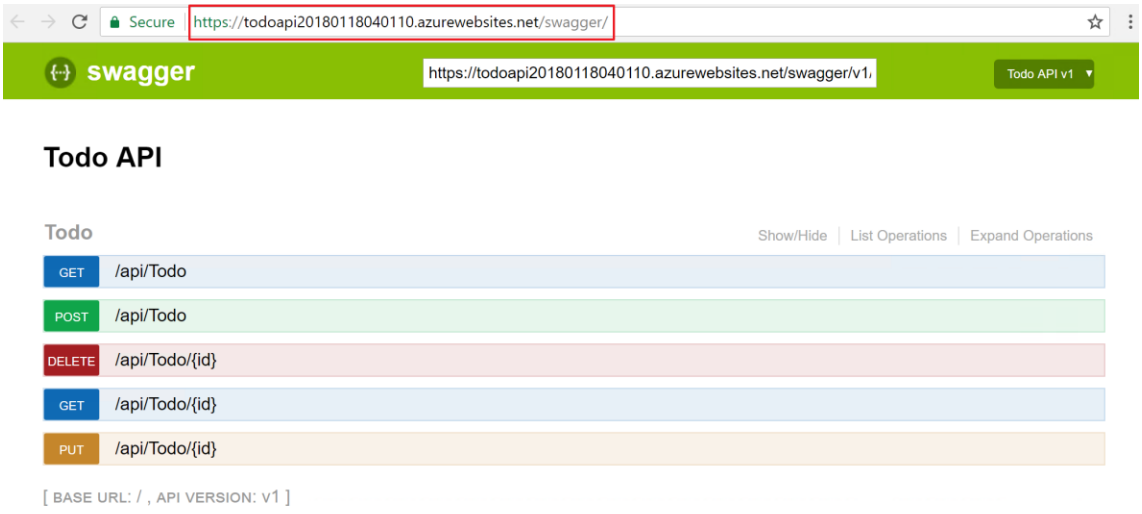
7. **Select the existing resource group** for the Resource Group option, and **accept the default values** for App Name and App Service Plan, as shown in the following screenshot.

> Please use the screenshot below only as a reference. The actual value of user account, App Name, Subscription, Resource Group and App Service Plan that you get will be different from what is shown on the screenshot.



8. Click **New** button besides the name of App Service Plan, change the **Location** to **West US** and the **Size** as **S1(1 core, 1.75 GB RAM).** Click OK.

9. Click **Create** button. Visual Studio would take several seconds to minutes to create the App Service and the API App. When the API App is created, it will publish the code to the API App. When the publishing completes, you would see the swagger page of the Web API as shown in the following screenshot. Notice the URL of the page is now pointing to an Azure website. **Please keep the swagger page open in the browser** as we need to use it later.



10. When the deployment completes, close Visual Studio 2017 on your lab machine.

1.3 Test the Todo API with Postman

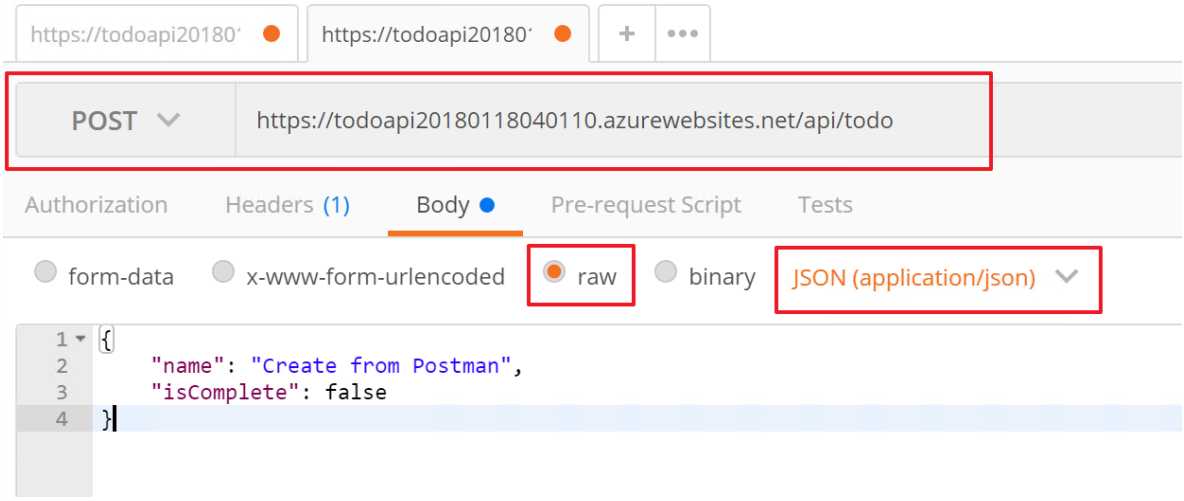
To make sure the Todo API works as expected, please follow the steps below to test it with Postman.

1. Open Postman by clicking its icon  on the status bar.
2. In **Enter request URL** textbox, input the URL of get todo list operation, which is like: **https://<your api app name>.azurewebsites.net/api/todo**.

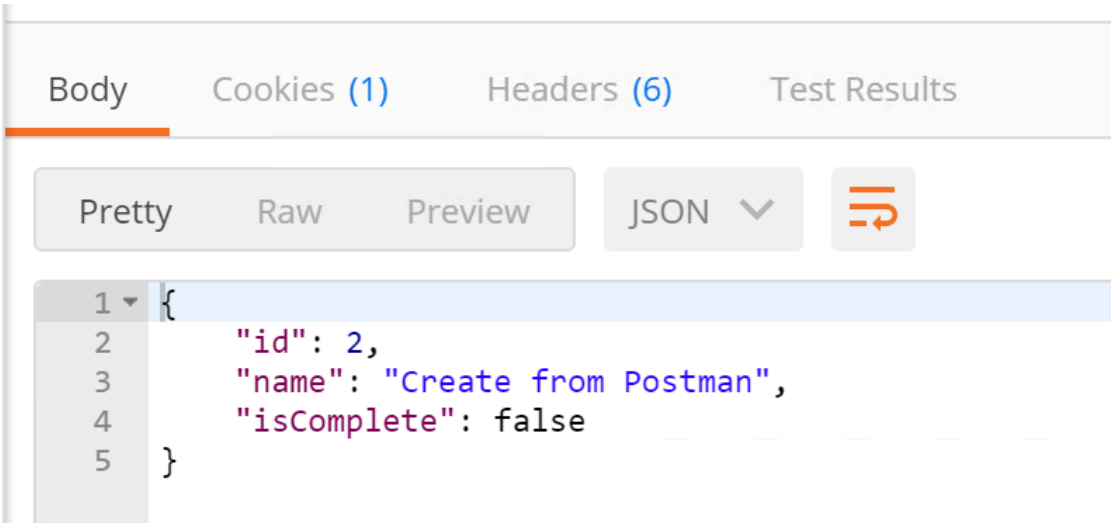
3. Click **Send**. If the Todo API works, you will see the following response.

4. Add a new tab in Postman, and in this new tab window, change the HTTP verb to **Post**. Input the same URL as Step 2.
5. Click **Body** tab under the URL textbox. Select **Raw**, and change the content type to **JSON (application/json)**. In the request body textbox, input the following data.

```
{
    "name": "Create from Postman",
    "isComplete": false
}
```



6. Click **Send**. You will see the following response.



You have completed all tasks of the Exercise 1 successfully.

Exercise 2: Import API to API Management

In this exercise, we will import our API to API Management, use Publisher portal to manage the API and Operations, and use Developer portal to test them.

Creating a new APIM instance usually takes between 20 and 30 minutes. For the sake of time of our lab, we will not create a new instance from scratch in this exercise. Rather, we will use a pre-provisioned APIM instance. If you'd like to know how to create an APIM instance from scratch, please refer to the links below.

- Create a new Azure API Management service instance - Portal
- Create a new Azure API Management service instance - PowerShell

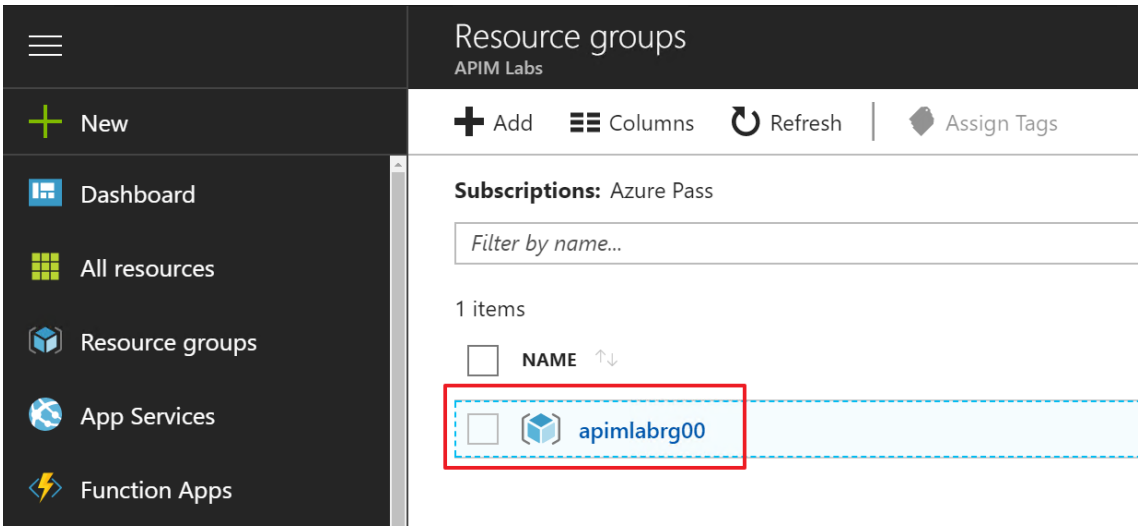
Please use the screenshots in the following exercise only as a reference. The actual value that you see in your lab machine will be different from the values shown in the screenshots.

2.1 Open Publisher portal and get familiar with it

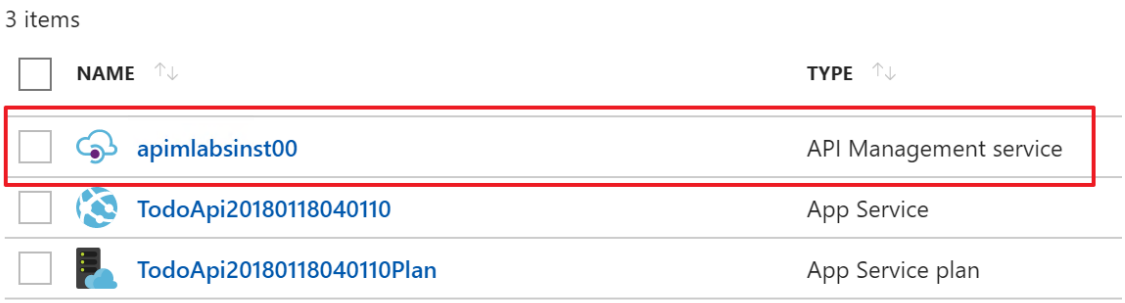
Azure APIM supports adding APIs manually or importing them from their definitions. In this exercise, we will import our API with Open API/Swagger definition since our API has already had the definition document. If you want to know more about adding API manually, please refer to Add an API manually.

Please follow the steps below to import the API.

1. Open a web browser and browse to https://portal.azure.com.
2. Sign in with the assigned user name and password.
3. In the Azure portal, click **Resource groups** on the left navigation menu, and you will see one resource group assigned to you, similar as the following screenshot. Click the **name of the resource group** to open the resource group page.

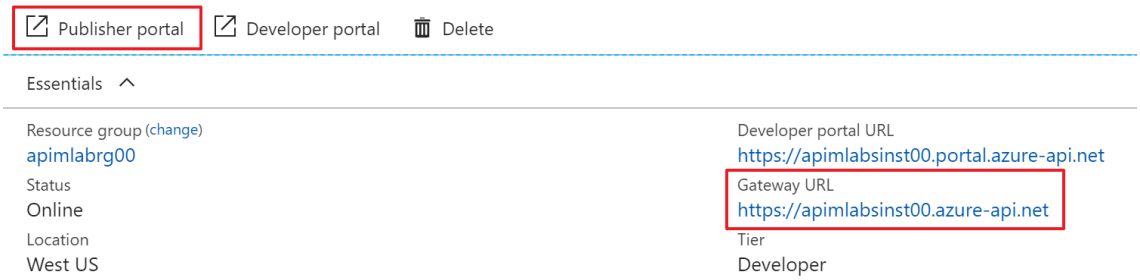


4. On the resource group page, you would see existing resources such as the APIM instance, App Service Plan and API App etc., similar as the following screenshot.



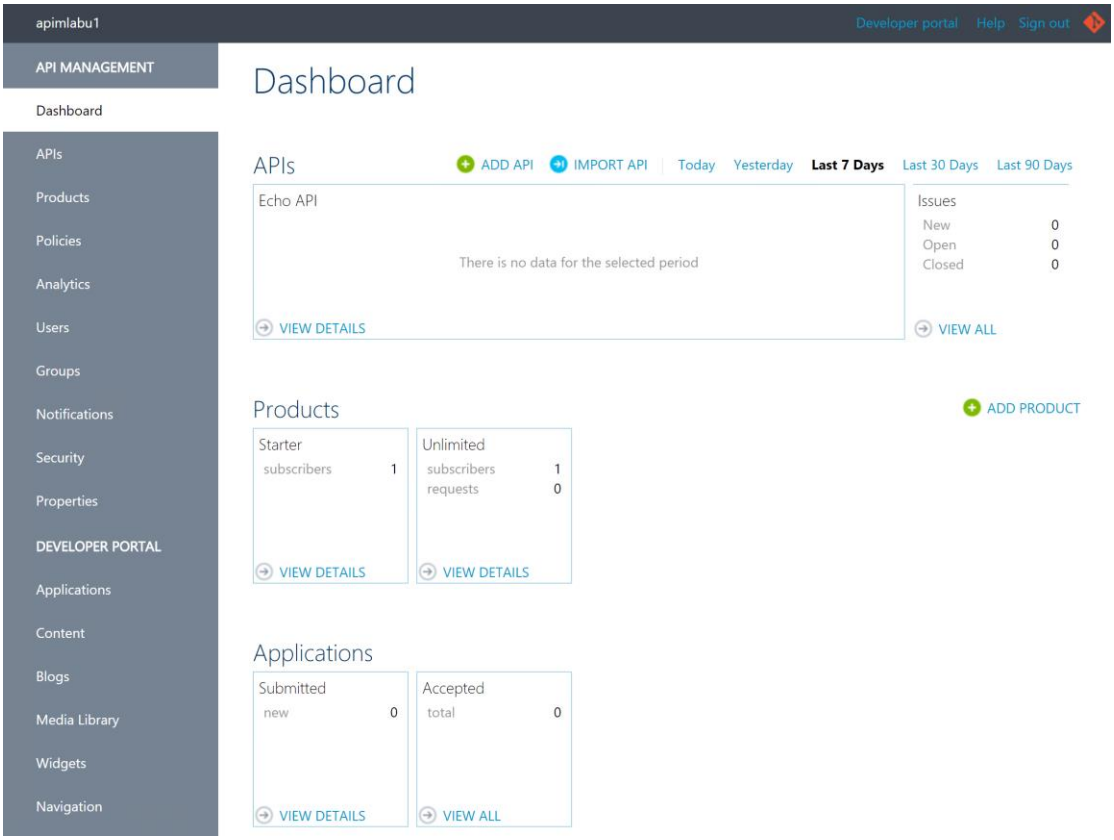
5. Click the name of the resource whose type is **API Management service** to open APIM page.

6. On top of the APIM page, you will see the links to **Publisher portal** and **Developer portal** similar as the screenshot below. The **Gateway URL** is the URL for client request of API calls.



7. Click **Publisher portal** link to open the publisher portal in a new browser tab.

8. The following screenshot shows the user interface of the publisher portal. With the publisher portal, you can manage your APIs and Products, create policies and perform analytics tasks etc. It is main place for us to manage our APIs in APIM.



9. Browse the links in the publisher portal to get familiar with it.

2.2 Import the API

To import the API with Open API/Swagger document, APIM requires the document must have the following 3 attributes, which are not in the swagger document of our API by default. We need to add them to the document manually.

- **Host Name**
- **Base Path**
- **Schemes**

For details of the import restrictions of APIM, please see [API import restrictions and known issues](#).

Please follow the steps below to complete the exercise.

1. On the swagger page of our API, copy the URL of the swagger.json which looks like: **https://<name of your api app>.azurewebsites.net/swagger/v1/swagger.json**
2. Browse to the URL of swagger.json in a new browser tab. You will see something similar as the screenshot below.

3. **Right click** on the page in the browser and click **Save as**. Save it to the following location: **C:\labfiles\swagger.json**.

4. Open the saved swagger.json in a text editor. You can use any text editor to edit it, but to make things easier, we will use Visual Studio Code (VS Code). Click **Visual Studio Code** icon  on Windows task bar to launch it, and then open the saved swagger.json by clicking **File > Open File**.

5. In VS Code window, press **Alt+Shift+F** or right click and then click **Format Document** to format the file so that it can be edited easily.

6. Locate the "basePath" attribute, and add "host" and "schemes" attributes as shown below between "basePath" and "path".

```
    "basePath": "/",
    "host": "<name of your api app>.azurewebsites.net",
    "schemes": [
        "https"
    ],
```

   The value of "host" attribute should be the host name of the URL of your API App, something like "todoapi20180102090307.azurewebsites.net". **Don't include "http://" or "https://" in it**.

7. Save the file.

8. In the Publisher portal, click **APIs** on the left navigation panel, and then click **Import API**.

9. In the **Import API** page, configure the following settings, as shown in the screenshot below.

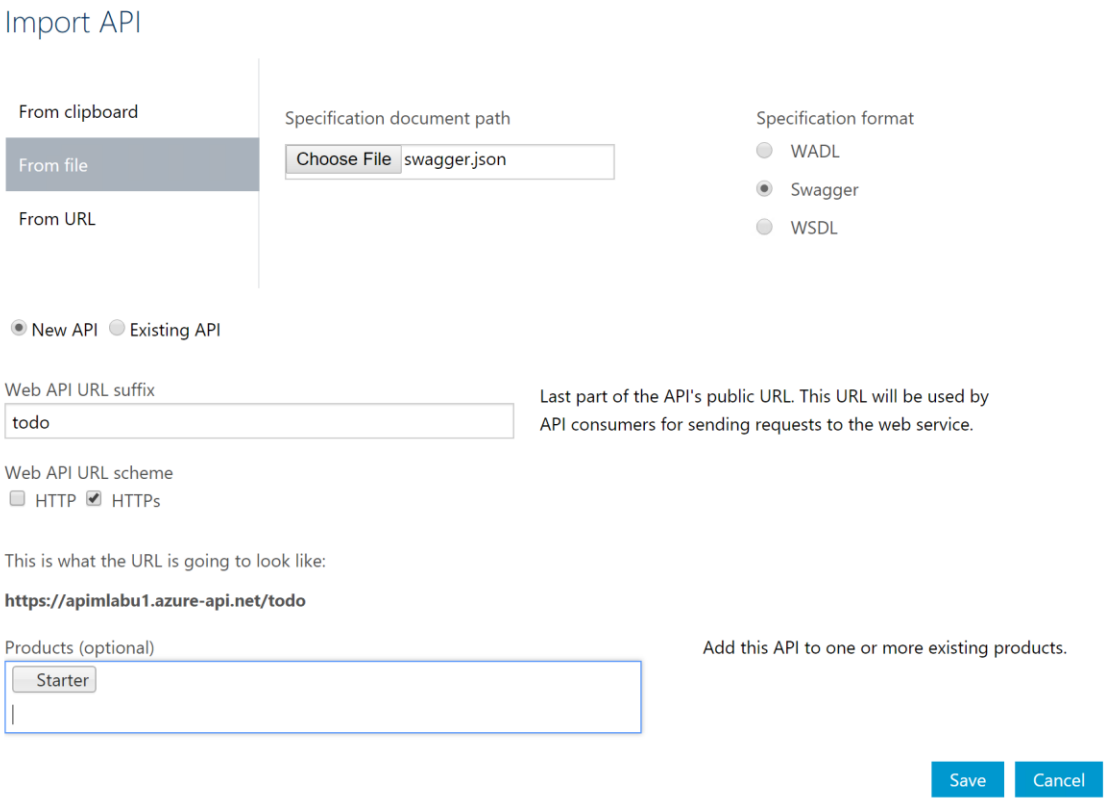
   a. Choose **From file**, and select the edited swagger.json.

   b. Select **Swagger** as the **Specification format**.

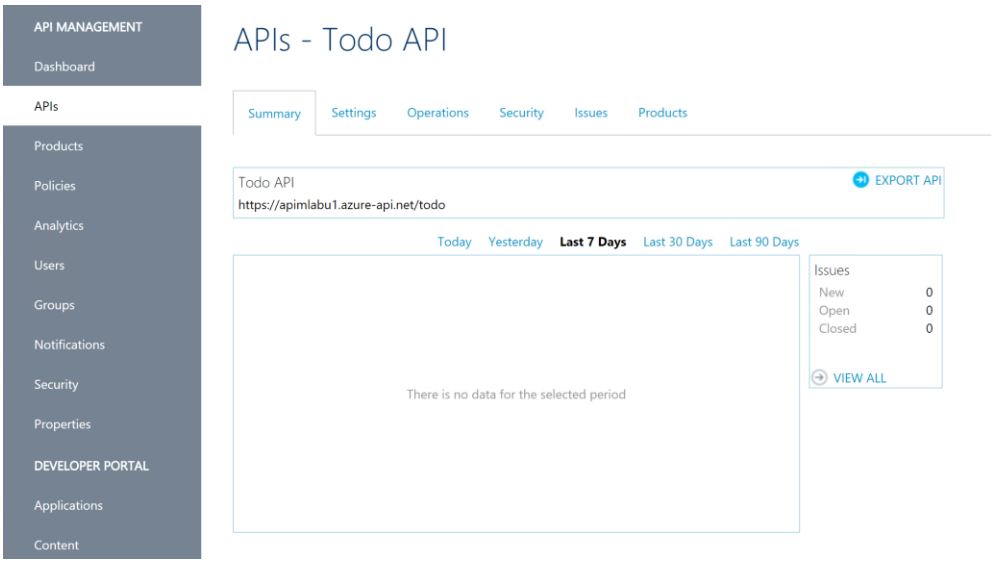
   c. Choose **New API** and give it a suffix such as **todo**.

d. Associate it with a product, for example **Starter**.

e. Click **Save**.



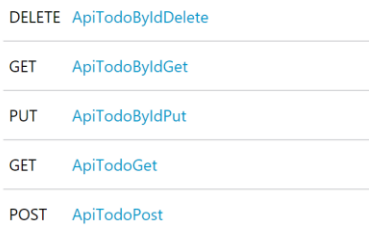
10. When the API is imported successfully, you will see something similar as the following screenshot in Publisher portal.



2.3 Configure Operations

When the API is imported, we can manage the configuration of operations further in Publisher portal.

1. On Todo API page, click **Operations** tab. You will see the following operations.

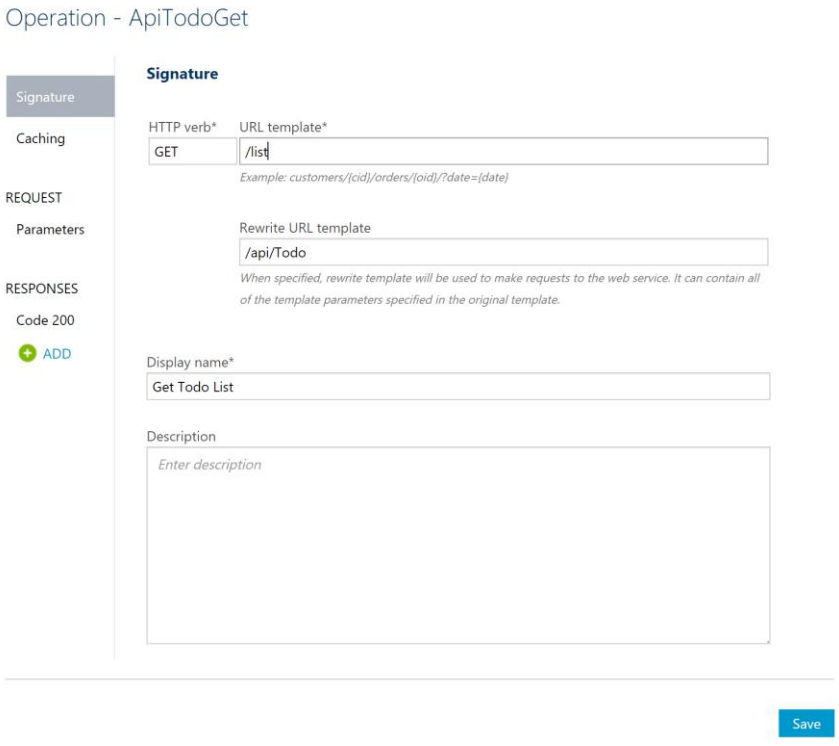


   The display names of the operations are generated by APIM based on the swagger document. The names are not very user friendly. Also, the default URL template used by the operations is not very intuitive as well. Let's change them.

2. Click on the operation, **ApiTodoGet**.

3. Change the **Signature** of this operation to the following, as shown in the screenshot below, and click **Save**. **Please don't change the HTTP verb**.

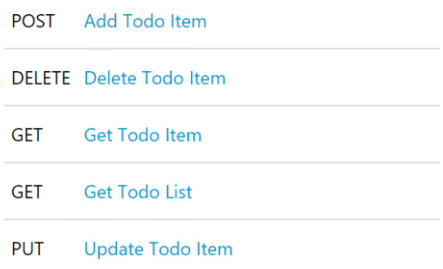
URL template:	/list
Rewrite URL template:	/api/todo
Display name:	Get Todo List



4. Repeat the step 3 for other operations, and change their **Signature** according to the values in the following table. Again, **please don't change the HTTP verb of the operations**.

Operation	URL template	Rewrite URL template	Display name
ApiTodoByIdGet	/list/{id}	/api/Todo/{id}	Get Todo Item
ApiTodoPost	/add	/api/Todo	Add Todo Item
ApiTodoByIdPut	/update/{id}	/api/Todo/{id}	Update Todo Item
ApiTodoByIdDelete	/delete/{id}	/api/Todo/{id}	Delete Todo Item

5. The operations look like the following after the configuration.



2.4 Test the API and Operations

To test the API and Operations, we will use the Developer portal of APIM. Developer portal is the place where client developers can learn and test your APIs. You can also test the API and Operations with tools like Postman.

1. In **Publisher portal**, right click on the link of **Developer portal** on the upper left side and **open it in a new tab**.

2. Switch to the browser tab of Developer portal. You will see the default layouts of the developer portal. The layouts and the style of the developer portal can be customized. However, the customization of the developer portal is not covered in this lab. If you want to know more about it, please see Customize the style of the Developer portal pages.

3. In the Developer portal, click **APIS** and then click **Todo API**.

4. On the Todo API page, you can see the document for each operation, such as the Request URL, Request Header, Request Body and sample code for some popular languages. This information could help client developers to understand the API and operations easily.

   If you browse the document for each operation, you would notice the request header, **Ocp-Apim-Subscription-Key**, is required by all operations. This is the subscription key for APIM to authorize the requests. All requests to APIM must have a valid subscription key.

5. Click **Get Todo List** operation and click **Try it** button. It will navigate to a page with a sample request. Review the request headers, and the HTTP request. Click **Send**.

6. When the response is returned from our API App, you will see a JSON array, something similar with the following screenshot.

7. Besides returning the result of our API App, APIM also generates the trace as we set the **Opc-Apim-Trace** header to true in the request. The trace could help developers to troubleshoot issues. If you want, you can download the trace with the URL shown in **Opc-Apim-Trace-Location** response header.

8. A simpler way to view the trace is to click the **Trace** tab. It will show traces of different phases of APIM, such as Inbound, Backend, Outbound and On error.

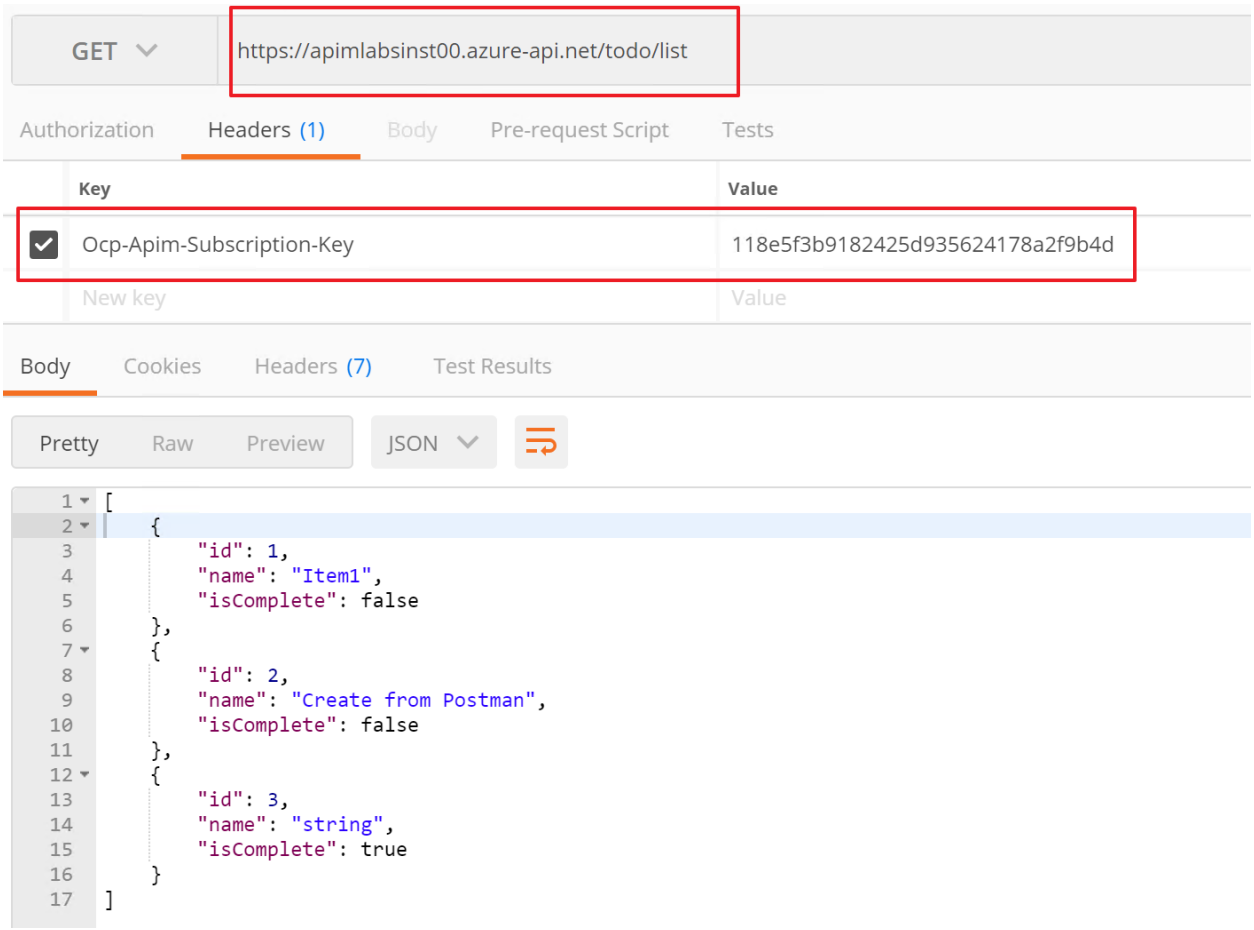
9. Optionally, you can test other operations in the Developer portal.

2.5 Test the Operations with Postman

To test the API Operations in Postman, we can repeat what we did in Exercise 1.3, but with the following changes.

Request URL:	<The operations URL of APIM. It looks **like https://<your apim name>.azure-api.net/todo/list**>
Request Header:	<The request from Postman must have a **Ocp-Apim-Subscription-Key** request header. You can get the value of this request header from the developer portal. >

The following screenshot shows an example of sending request from Postman.



When you finish testing the API and Operations, you have completed the Exercise 2.

Exercise 3: Secure the API App with Azure AD

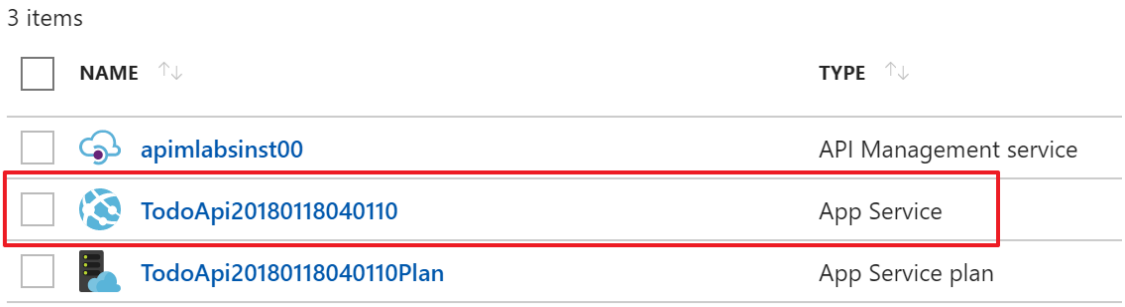
So far, we didn't enable any authentication/authorization on our API App. It means anyone can call our API directly as long as they know the URL of our API App. We want to protect our API App from being called out of APIM. The ideal state we want to achieve is that all calls must go through the APIM gateway no matter if the developers know the URL of our API App or not. To achieve it, we will protect our API App with OAuth 2.0 with Azure AD (AAD) and APIM.

3.1 Enable authentication on API App

With Azure App Service, enabling authentication/authorization for Web App or API App has become to be very easy. We don't even need to make any code change to enable it.

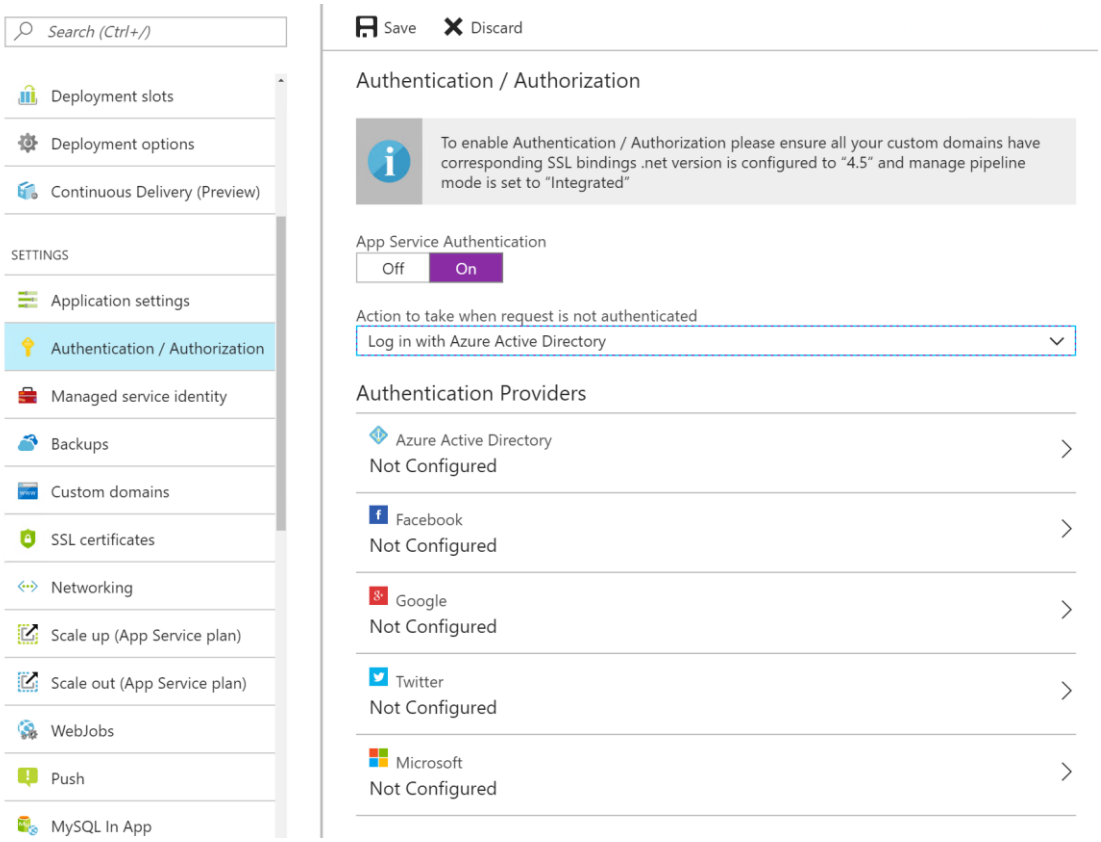
1. Sign in to the Azure portal in the browser, navigate to the resource group and open the resource group page. The steps are the same as what is described in exercise 2.1.

2. Click the name of the resource whose type is **App Service**. This should open the page of our API App.



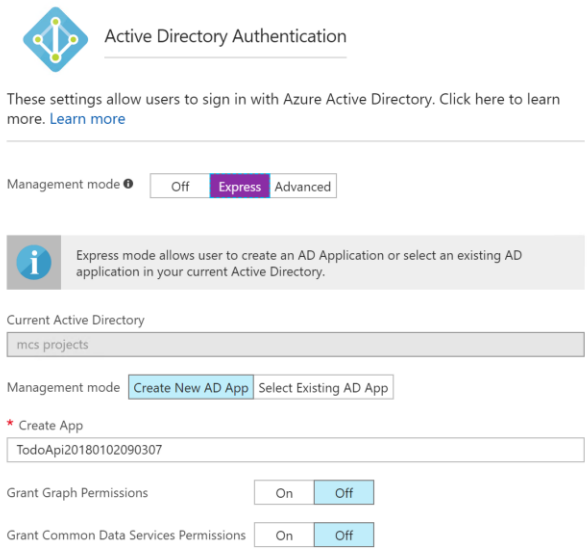
3. On the API App page, click **Authentication/Authorization** under **Settings**.

4. Turn **On** the App Service Authentication, and select **Log in with Azure Active Directory** for **Action to take when request is not authenticated**, as shown in the following screenshot.



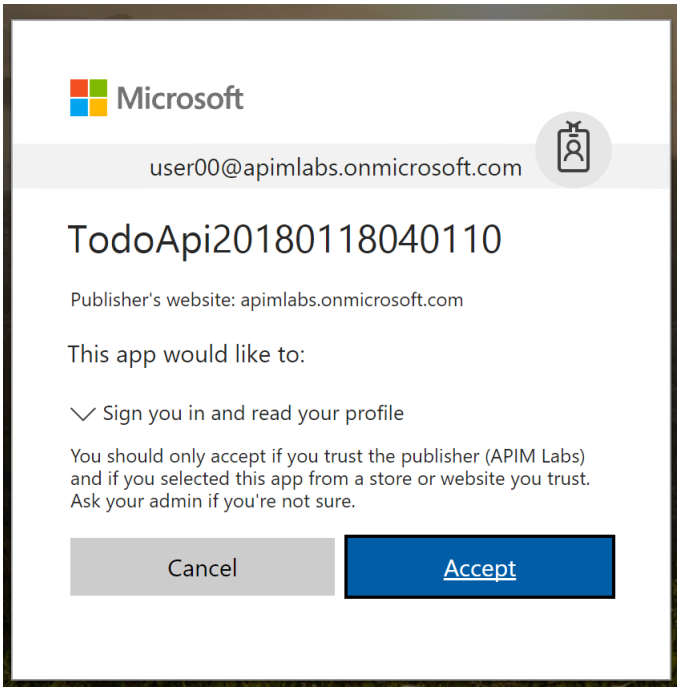
5. Click **Azure Active Directory** under **Authentication Providers**.

6. Select **Express** as the **Management mode**, as shown in the following screenshot, **don't change the default value of Create App** and click **OK**.

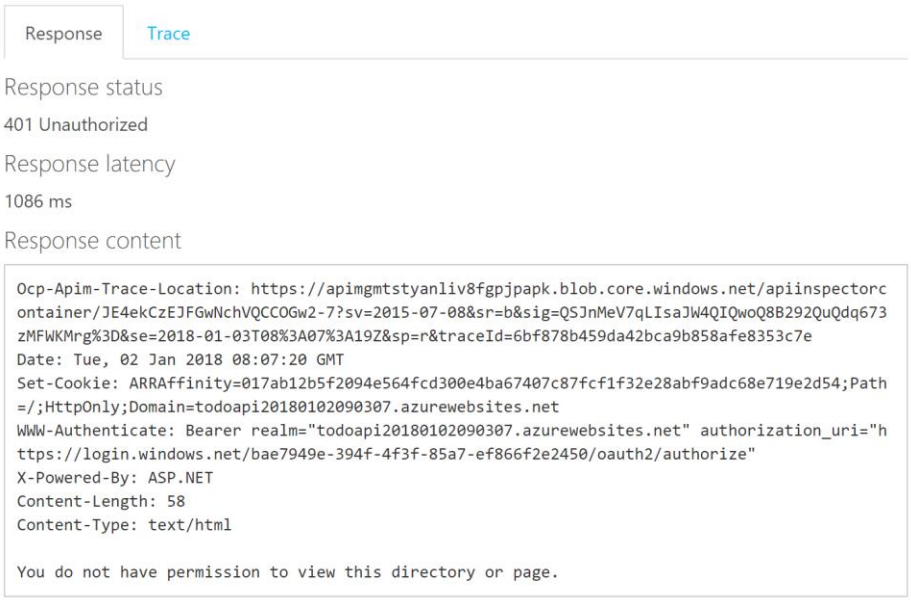


7. Click **Save** on the Authentication/Authorization page.

8. When the authentication is enabled, if you try to open the swagger page with the URL: **https://<your api app name>.azurewebsites.net/swagger**, you would see an authorization page of AAD, similar as the following screenshot. You will see the swagger page when you click **Accept**.



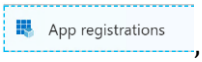
9. If you try the same test in exercise 2.4 and 2.5, you will get an HTTP 401 Unauthorized error as shown in the screenshot below.



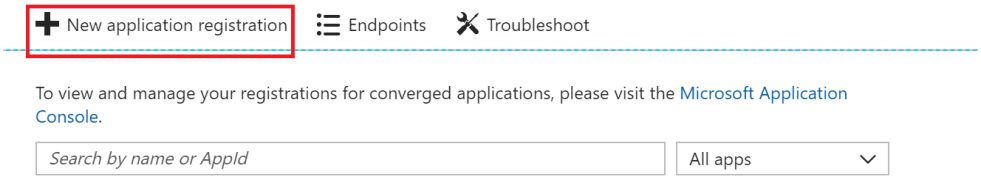
This is expected because our API App is now protected by the AAD. The request from developer portal is unauthorized by the AAD. Therefore, the request fails. For developer portal to be able to send authorized request, we need to register it as an app in the AAD. We will do it in the following exercise.

3.2 Register the developer portal as an AAD application

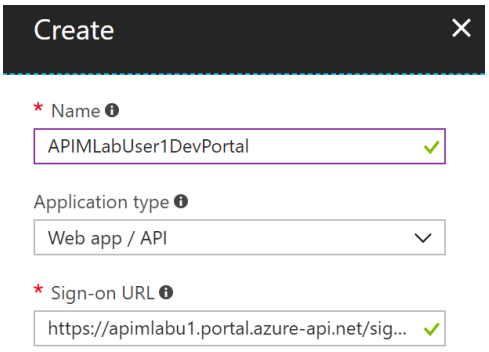
Please follow the steps below to register the developer portal in AAD.

1. In Azure portal, click **Azure Active Directory** on the left navigation panel.
2. Click **App Registrations** 

, and then click **New application registration** on top of the page, as shown in the screenshot below.



3. For **Name**, give it a unique name. As AAD is shared with all lab users, it's better to include the unique user id that you use in the name, such as APIMLab***User01***DevPortal.
4. For **Application Type**, select **Web app/API**.
5. For **Sign-on URL**, enter the URL of your API Management service and append /signin. For example, https://apimlabsinst00.portal.azure-api.net/signin. This is just a placeholder and we will change it to another URL later.

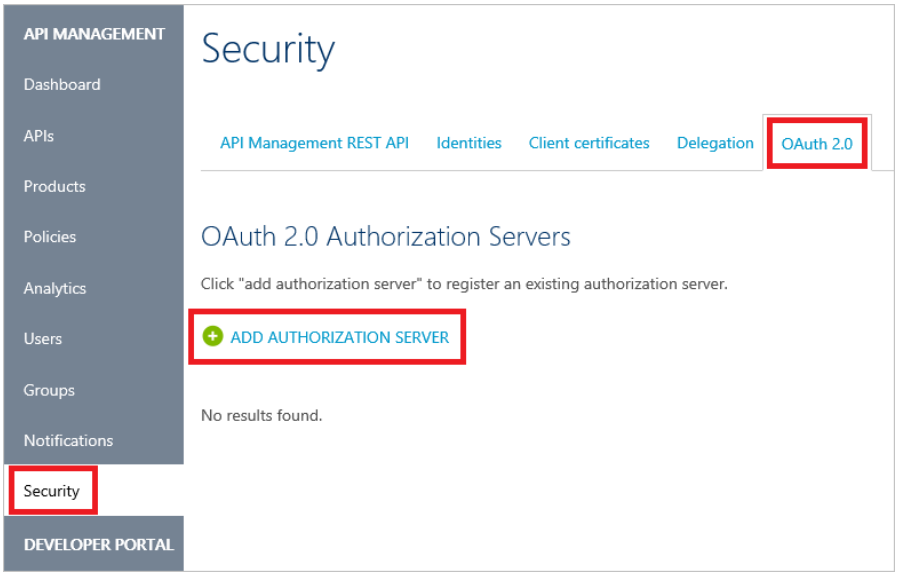


6. Click **OK**. Wait several seconds, and a new app will be shown on the **App Registration** page. If you cannot see it, enter the name in the **Search by name or AppId** textbox.

3.3 Configure an API Management OAuth 2.0 authorization server

Please follow the steps below to configure an OAuth 2.0 authentication server in APIM.

1. Click **Security** from the API Management menu on the left, click **OAuth 2.0**, and then click **Add authorization** server.

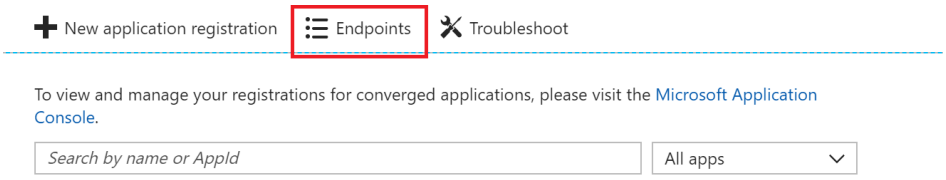


2. Finish the configuration with the following settings.

    a. Give it a **Name** such as **APIM Lab AAD**.

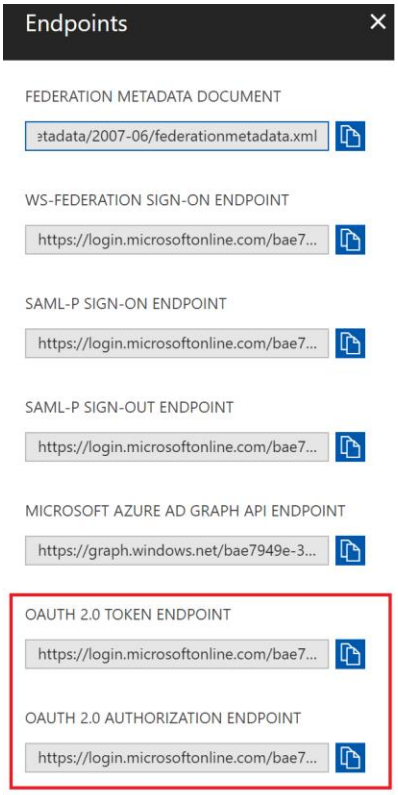
    b. For **Client registration page URL**, enter http://localhost as a placeholder since this feature is not used in this lab.

    c. For the **Authorization endpoint URL** and **Token endpoint URL**, you can get the value by clicking the **Endpoints** on the **App Registration** page in **Azure Portal**.



Copy the **OAuth 2.0 authorization endpoint** and paste it into the **Authorization endpoint URL** textbox.

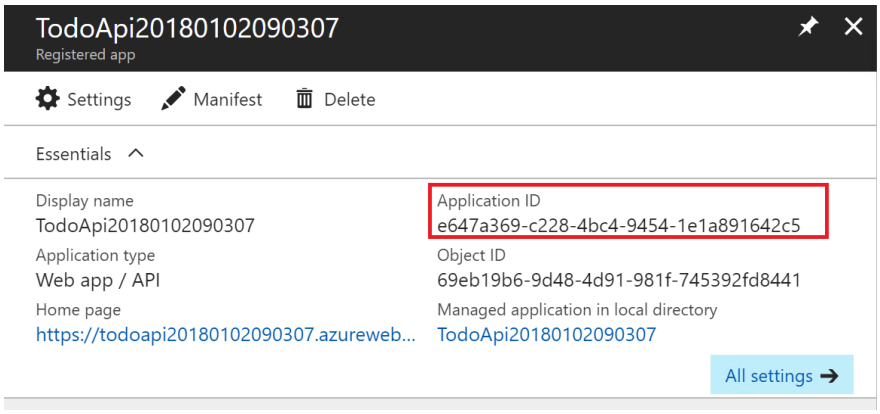
Copy the **OAuth 2.0 token endpoint** and paste it into the **Token endpoint URL** textbox.



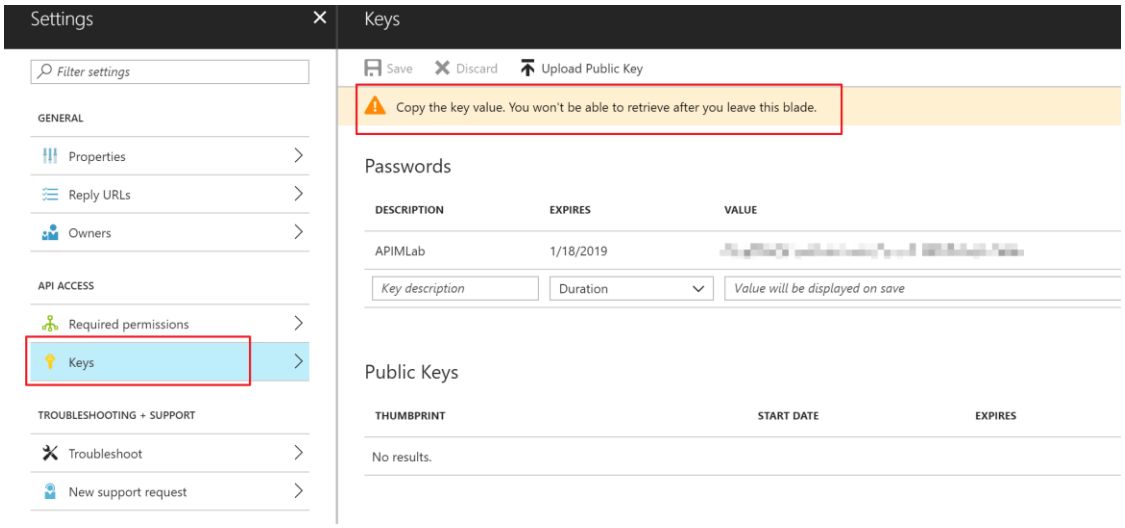
d. We need to register our API App as a resource so that the developer portal can access it with an access token. Add an additional body parameter named **resource** and for the value, use the AppId of our API App.

To get the AppId of our API App, in the Azure AD settings of **Azure portal**, click the name of your API App on the **App Registration** page, and copy the **Application Id**.

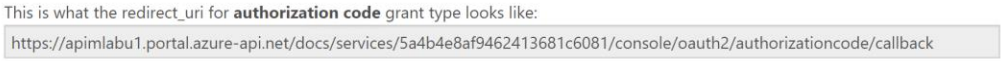
As the AAD is shared with all lab users, you would see many applications with the similar name on the **App Registration** page. Make sure you select your API App. A simple way to do it is to put your API App name in the **Search by name or AppId** textbox to filter the app list.



e. The value for **Client ID** is the **Application ID** of the Developer Portal App that we registered in AAD, and the value for **Client secret** is the **Key** of the Developer Portal App. If there is no key created for the dev portal app, you can create one by providing a **Description**, **Expiration** time and clicking **Save**. Make sure you copy the **Value** before navigating away from the page.

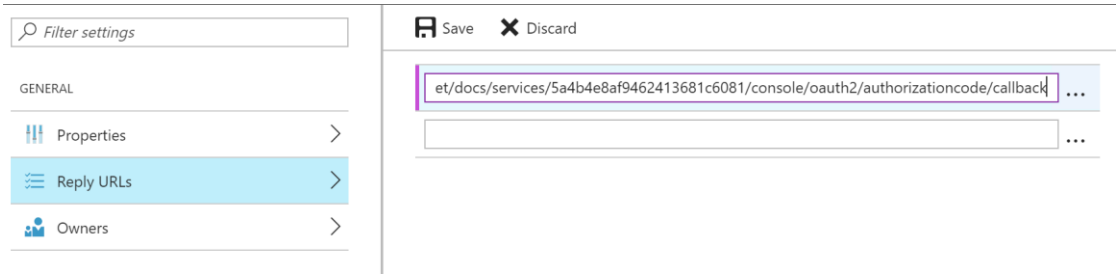


f. Copy the value of redirect_uri into clipboard, and click **Save**.



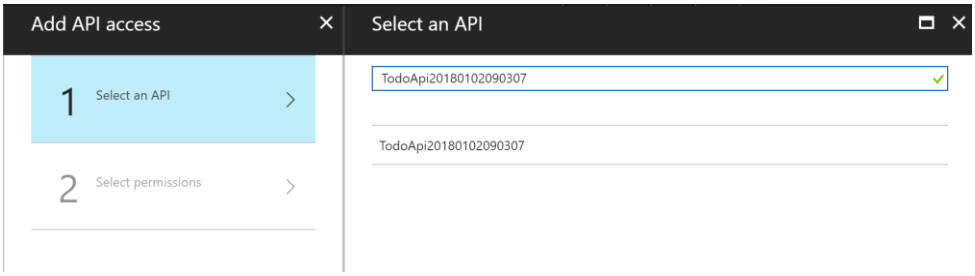
3. In **Azure portal**, go to AAD and open the **App Registration** page. Find the developer portal app that you registered, and click it.

4. Click **Reply URLs**, and replace the existing reply URL with the value of redirect_uri and click **Save**.

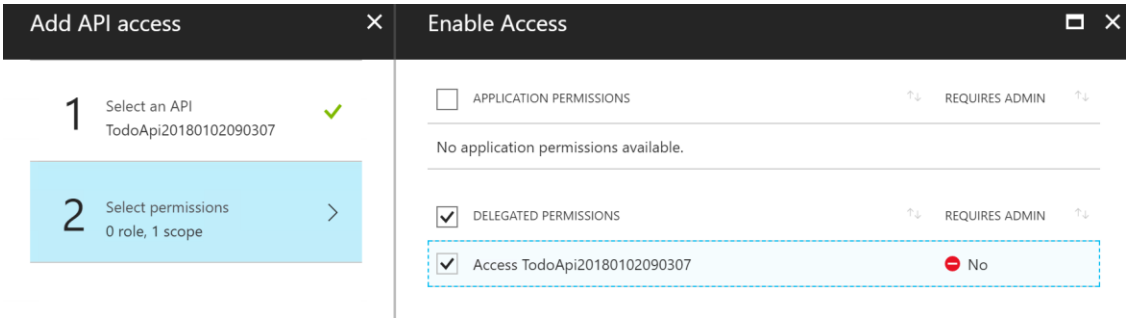


5. Click **Required permissions**, and then click **Add**.

6. Click **Select an API**, and search the name of your API App. Select it once found.



7. On **Select permissions** page, select **Access <API App Name>**. Click **Select**, and then click **Done**.



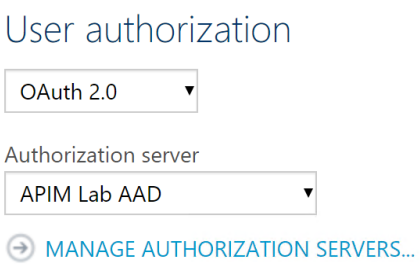
With the above steps, you have configured the OAuth 2.0 authentication server in APIM. Let's test it in the following exercise.

3.4 Test the OAuth 2.0 authentication in APIM

To test if the OAuth 2.0 authentication works in APIM, please follow the steps below.

1. In the Publisher portal, click **APIs**, then click **Todo API**, and then click **Security** tab.

2. For **User authorization**, select **OAuth 2.0**. For **Authorization server**, select the one that we created in exercise 3.3.



3. Switch to Developer portal, click **APIS**, then **Todo API**. Select **Get Todo List** operation, and click **Try it**.

4. Comparing to the UI before enabling the OAuth 2.0, there is a new option for authorization as shown in the following screenshot.

5. If we keep **APIM Lab AAD** as **No auth**, and click **Send**, we will get the **401 Unauthorized** error.

6. Click the dropdown of APIM Lab AAD and select **Authorization code**. Click **Accept** button on the popup window. Developer portal will get the access token from AAD and add it in the **Authorization** request header, as shown in the following screenshot.

7. Click **Send**. Now we can get the response back successfully.

8. Optionally, if you repeat the test of Exercise 2.5 with Postman, you will still see authorization error. That is because there is no **Authorization** request header in the request of Postman. You will have to manually add the Authorization header and copy its value from the developer portal. We will see how we can make Postman to send the request successfully without adding the Authorization request header in Exercise 4.

You have completed Exercise 3.

Exercise 4: Control the behavior of the API with policies

APIM policies are a powerful capability of the system that allow the publisher to change the behavior of the API through configuration. Policies are a collection of Statements that are executed sequentially on the request or response of an API.

Policies are applied inside the gateway which sits between the API consumer and the managed API. The gateway receives all requests and usually forwards them unaltered to the underlying API. However, a policy can apply changes to both the inbound request and outbound response.

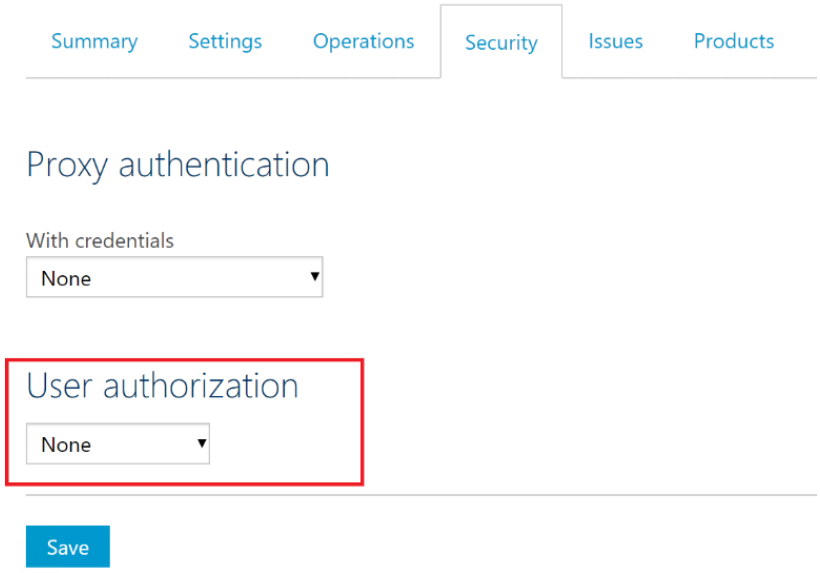
We will see how to use it in the following exercises.

4.1 Simplify the OAuth 2.0 authorization with policies

In Exercise 3, we protected our backend API App with AAD, configured the OAuth 2.0 authentication server, and enabled the OAuth 2.0 authorization on the API in APIM. With these configurations, the client developers would need to have AAD accounts or their apps need to be registered in the AAD before the apps can call our API. This is fine if the client developers are internal users and the apps are for internal use of an organization. However, there will be problems if the developers are external partners and the apps are for external use, since they won't have the AAD accounts.

To address this problem, we will use APIM policies to handle the OAuth 2.0 authorization between the APIM gateway and the backend API App. The APIM gateway will only use subscription key to authorize the client developers.

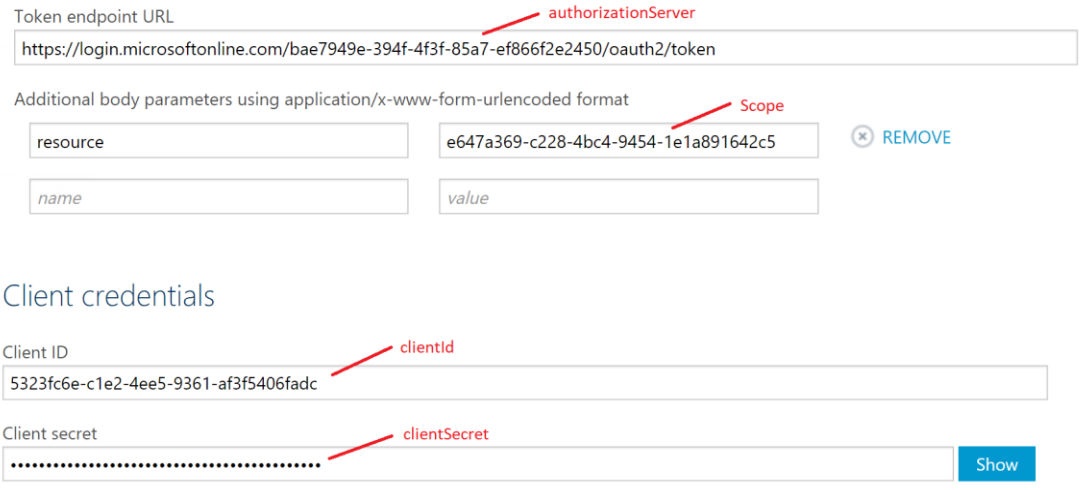
1. In the Publisher portal, click **APIs**, then **Todo API**, and then click **Security** tab. Change User authorization from **OAuth 2.0** to **None** and **Save**.



2. The inbound policy that we are going to create uses APIM properties to retrieve the AAD related information. To create the properties, click **Properties** and click **Add Property**. Add the following 4 properties.

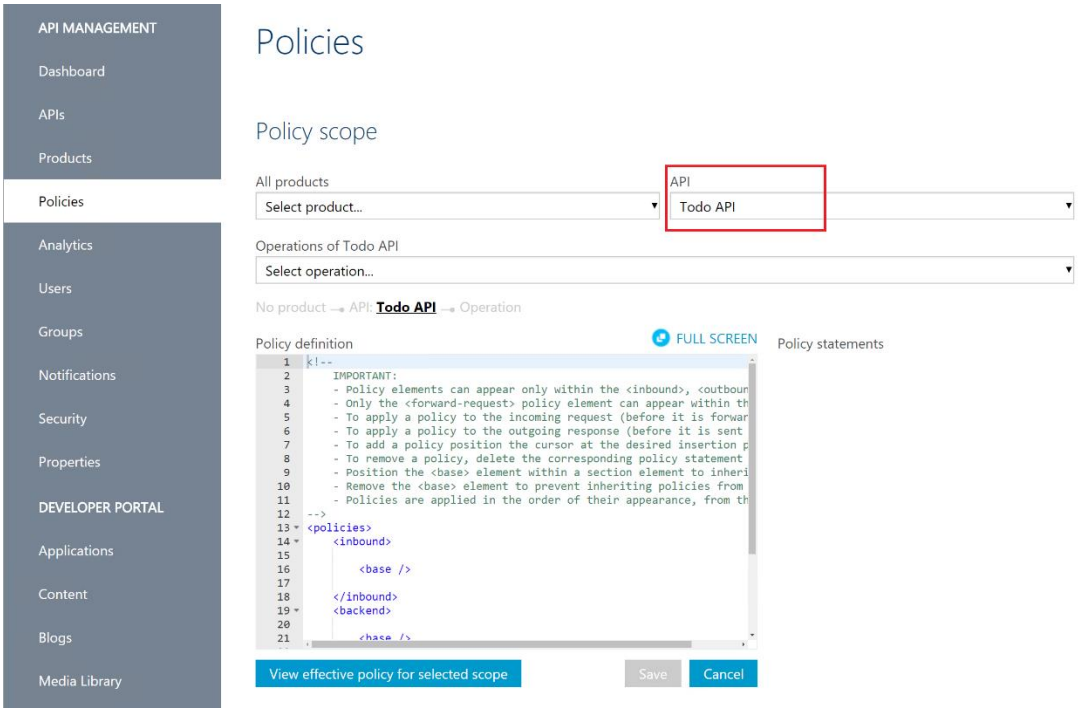
Property Name	**Value**
authorizationServer	*<The token endpoint of AAD>*
clientId	*<The Application ID of developer portal in AAD>*
clientSecret	*<The Key of developer portal in AAD, which is URL encoded.>*
scope	*<The Application ID of the API App>*

The value of the properties can be got from the OAuth 2.0 authentication server that we configured in Exercise 3, as shown in the following screenshots.



> Please note that the value of clientSecret property needs to be encoded with URL encoder. You can use some online tools to encode it, such as https://www.urlencoder.org/

3. Click **Policies**. Policies can be applied at Global, Product, API, or Operation scope. For our scenario, as each of the operations of the API will need to get the OAuth 2.0 access token from AAD, we will create an API scope policy so that it will applied to all operations of the API. Select **Todo API** for API and click **Add policy**, as shown in the screenshot below. You can click **Full Screen** to enlarge the editor window.



4. Copy and paste the following code of the inbound policy to the location between **<inbound></inbound>**, under **<base />**, and **Save**. You can also refer to ***C:\Labfiles\apimlab\src\APIMPolicies\OAuth20.xml*** for the completed code of the policy.

```
<!-- This snippet provides and example of using OAuth2 for authorization
between the gateway and a backend. It shows how to obtain an access token from
AAD and forward it to the backend. -->
<!-- Send request to AAD to obtain a bearer token -->
<!-- Parameters: authorizationServer - format
https://login.windows.net/TENANT-GUID/oauth2/token -->
<!-- Parameters: scope - a URI encoded scope value -->
<!-- Parameters: clientId - an id obtained during app registration -->
<!-- Parameters: clientSecret - a URL encoded secret, obtained during app
registration -->
<!-- Copy the following snippet into the inbound section. -->
<cache-lookup-value key="accessTokenCache" variable-name="accessToken" />
```

```
<choose>
    <when condition="@(!context.Variables.ContainsKey("accessToken"))">
        <send-request ignore-error="true" timeout="20" response-variable-
name="bearerToken" mode="new">
            <set-url>{{authorizationServer}}</set-url>
            <set-method>POST</set-method>
            <set-header name="Content-Type" exists-action="override">
                <value>application/x-www-form-urlencoded</value>
            </set-header>
            <set-body>
                @{
                    return
"client_id={{clientId}}&resource={{scope}}&client_secret={{clientSecret}}&gran
t_type=client_credentials";
                }
            </set-body>
        </send-request>

        <set-variable name="bearerTokenVar"
value="@((String)((IResponse)context.Variables["bearerToken"]).Body.As<JObject
>().ToString())" />
        <set-variable name="accessToken"
value="@(JObject.Parse((String)(context.Variables["bearerTokenVar"]))["access_
token"].ToString())" />
        <set-variable name="expiresIn"
value="@(JObject.Parse((String)(context.Variables["bearerTokenVar"]))["expires
_in"].ToString())" />
        <cache-store-value key="accessTokenCache"
value="@((String)context.Variables["accessToken"])"
duration="@(Int32.Parse((String)context.Variables["expiresIn"]) - 10)" />
    </when>
</choose>
<set-header name="Authorization" exists-action="override">
    <value>
        @("Bearer " + (String)context.Variables["accessToken"])
    </value>
</set-header>
<!--  Don't expose APIM subscription key to the backend -->
<set-header exists-action="delete" name="Ocp-Apim-Subscription-Key" />
```

5. Switch to the Developer portal, click **APIS**, then **Todo API**. Select **Get Todo List** operation and click **Try it**.

6. On the Try it page, you won't see **APIM Lab AAD** under **Authorization** anymore. Based on the experience of Exercise 3.1, if we click **Send**, we would expect to get a 401 Unauthorized error. However, we won't see the error this time.

7. Click **Send** button. Instead of getting a 401 error, we get the 200 OK and the result of the API is returned successfully.

8. To examine what happened at the backend, click **Trace** tab. You will see the inbound trace similar as the following screenshots.


```
cache-lookup-value (46 ms)
{
    "message": "Cache lookup resulted in a miss, variable will not be set.",
    "key": "2_accessTokenCache",
    "variableName": "accessToken"
}
```

```
send-request (2 ms)
{
    "message": "POST request to 'https://login.microsoftonline.com/bae7949e-394f-4f3f-85a7-ef866f2e2450/oauth2/token' has been se
nt, result stored in 'bearerToken' variable.",
    "statusCode": "OK",
    "statusReason": "OK",
    "headers": [
        {
            "name": "Pragma",
            "value": "no-cache"
        },
```

```
set-header (1 ms)
{
    "message": "Specified value was assigned to the header (see below).",
    "header": {
        "name": "Authorization",
        "value": "Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ing0Nzh4eU9wbHNNMUg3TlhrN1N4MTd4MXVwYyIsImtpZCI6Ing0Nzh4eU9w
bHNNMUg3TlhrN1N4MTd4MXVwYyJ9.eyJhdWQiOiJlNjQ3YTM2OS1jMjI4LTRiYzQtOTQ1NC0xZTFhODkxNjQyYzUiLCJpc3MiOiJodHRwczovL3N0cy53aW5kb3dzLm5l
dC9iYWU3OTQ5ZS0zOTRmLTRmM2YtODVhNy1lZjg2NmYyZTI0NTAvIiwiaWF0IjoxNTE0ODk3NTE0LCJuYmYiOjE1MTQ4OTc1MTQsImV4cCI6MTUxNDkwMTQxNCwiYWlvI
joiWTJOZ1lQZzdiYmJSbXUvYVhqd2VqWFpzNzNlNUFnQT0iLCJhcHBpZCI6IjUzMjNmYzZlLWMxZTItNGVlNS05MzYxLWFmM2Y1NDA2ZmFkYyIsImFwcGlkYWNyIjoiMS
IsImlkcCI6Imh0dHBzOi8vc3RzLndpbmRvd3MubmV0L2JhZTc5NDllLTM5NGYtNGYzZi04NWE3LWVmODY2ZjJlMjQ1MC8iLCJvaWQiOiJkMjIxYTM2MS05NTA4LTRiMzU
tYWViNC05NDc1ZDA5ZjAwNWEiLCJzdWIiOiJkMjIxYTM2MS05NTA4LTRiMzUtYWViNC05NDc1ZDA5ZjAwNWEiLCJ0aWQiOiJiYWU3OTQ5ZS0zOTRmLTRmM2YtODVhNy1l
Zjg2NmYyZTI0NTAiLCJ1dGkiOiJhZFloZVJYeXRrYUdmX2NRWEdOOUFBIiwidmVyIjoiMS4wIn0.rZ0qUwC-YcGHWS0wL6YTn0ZM8FtDvDEjcSdaxpwDclx0wqFa8qaWB
54B1mjryGLRUcMdaIiYBjCtFf0yTeXTfFy78965bZTkF0ls6RQLNu-XmUipIA_vSFjbms3rzOMn-d2TO4oWPyOPNYcVFxbuxtwYQembT1rqSgsj9B6YxSJhFKov7iV-Q7
4oXZg9d2ZhWvxVBkTchxtzEGioSo-AqrKIhQPX-htSkFuchixcPP4W7d7oTFSAxFum52SNk9DEBVWt_t9Qi4-LfhBoZlZlE3Cmzllv2-nfQ43al_DaA90ikMT3fSLS54A
Kt1ywavuOA0c_cCPFeNKtKVzSfnaibw"
    }
}
```


These traces show how the policy works:

- When a request is coming in, the policy will look up to the cache to see if a valid access token is cached.
- If not, it sends a request to AAD to get an access token and cache it.
- Then it adds the access token to the Authorization request header and send the request to the backend API App.

9. Repeat the test of Exercise 2.5. Now the Postman can send a request without Authorization header and get the successful response.

With this policy, client developers just need a valid subscription key to call the API via APIM gateway. They don't need to deal with the complexity of OAuth and AAD.

4.2 Transform the request and the response

APIM provide us many useful policies. With these policies, we can restrict the access, transform the request and response, cache the data to improve the performance and perform many other activities on our APIs. For the full list of policies, please refer to API Management policies.

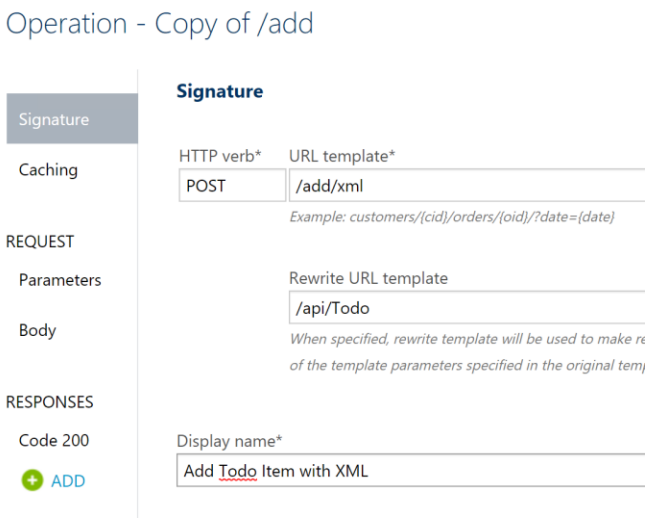
In this exercise, we will see how to create an Operation manually, and we will also leverage the transformation policies of APIM to transform the request and response. Hopefully, it will give you an idea on how to use the policies of APIM to achieve your goal.

1. In Publisher portal, click **APIs**, then **Todo API**. And then click **Operations** tab.

2. We will create a new operation to add a todo item but with XML request. To make it easy, we will create the operation based on the existing **Add Todo Item** operation. Click **Clone** link besides the **Add Todo Item** operation. A new operation named **Copy of /add** will be created.

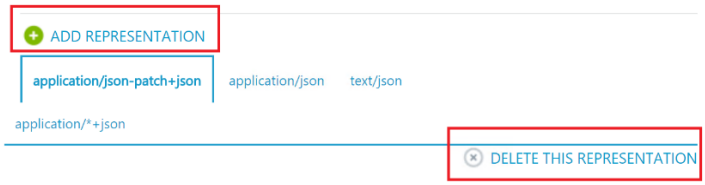
3. Click **Copy of /add**. Change its signature to the following and click **Save**.

URL template	/add/xml
Rewrite URL template	/api/todo
Display name	Add Todo Item with XML



4. Click **Body** under **Request**. Remove all existing representations by clicking **Delete This Representation**. And then add the following two representations by clicking **Add Representation**.

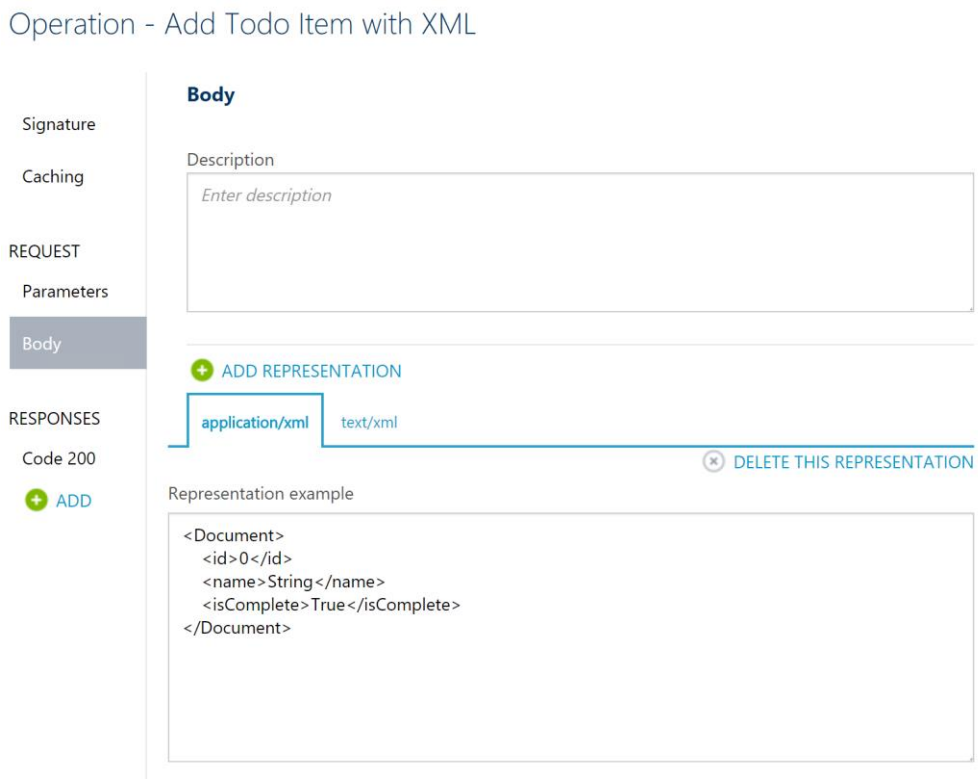
    - application/xml
    - text/xml



5. Copy and paste the following XML to the **Representation example** textbox and click **Save**. This will help the developers to better understand the request body.

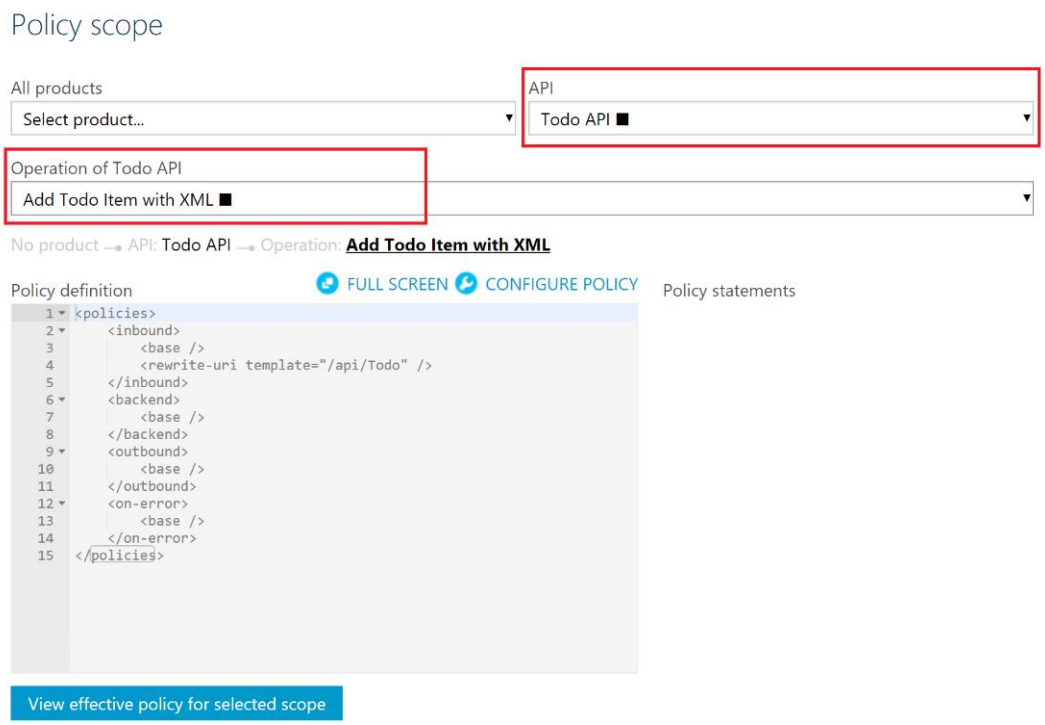
```
<Document>
    <id>0</id>
    <name>String</name>
    <isComplete>True</isComplete>
</Document>
```

6. The final body settings of the operation look like the following:

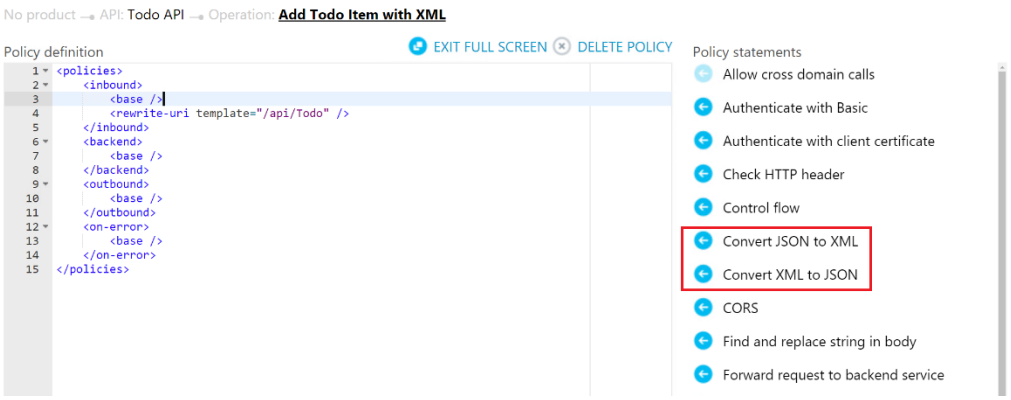


Now this operation can accept XML request. However, since our backend API App only accept JSON, we cannot forward the request directly to the backend API App. Otherwise, we will get a **HTTP 415 Unsupported Media Type** error. We need to convert the XML to JSON first.

7. Click **Policies**. Select **Todo API** for API and **Add Todo Item with XML** for Operation, as shown in the screenshot below.



8. Click **Configure Policy** to enable the policy editor, and then click **Full Screen**. The screen looks like the following:



9. What we need to implement for inbound policy is to convert the XML to JSON. There is a built-in policy statement in APIM which can help us to achieve it easily. In the policy editor window, **add a new line** in inbound between **<base />** and **<rewrite-uri template="/api/todo" />** (press Enter at the end of **<base />**).

10. From the **Policy statement** list on the right side, click **Convert XML to JSON**. It will insert the following policy statement.

```
<xml-to-json kind="javascript-friendly | direct" apply="always | content-type-xml" consider-accept-header="true | false"/>
```

11. Edit the above statement with the following values:

kind	direct
apply	content-type-xml
consider-accept-header	false

After editing, the statement will look like the following:

```
<xml-to-json kind="direct" apply="content-type-xml" consider-accept-
header="false"/>
```

When xml-to-json convert the XML, it will include the root element of the XML in the converted JSON. However, our backend API App doesn't accept the JSON with the root element. We need to remove it before forwarding the request to the backend.

12. Add a new line between **<xml-to-json … />** and **<rewrite-uri … />**. Click **Find and replace string in body** on the **Policy statement** list on the right side **twice** to add two of this statement.

13. Change the two **Find and replace string in body** statements as follows.

```
<find-and-replace from="{&amp;quot;Document&amp;quot;:" to="" />
<find-and-replace from="}}" to="}" />
```

We have finished configuring the inbound policy. Considering the client may only accept XML as the response, we need to configure the outbound policy to convert JSON back to XML.

14. In **<outbound></outbound>**, add a new line under **<base />**. And then click **Convert JSON to XML** on the Policy statement list. It will add the following statement.

```
<json-to-xml apply="always | content-type-json" consider-accept-header="true |
false"/>
```

15. Edit the statement and change it according to the following:

```
<json-to-xml apply="content-type-json" consider-accept-header="true"/>
```

16. Click **Save**. If everything is correct, the policies will be saved. Otherwise, you will see an error on the page. The final version of the policies looks like the following. You can also refer to ***C:\Labfiles\apimlab\src\APIMPolicies\xml2json.xml*** for the completed code of the policy.

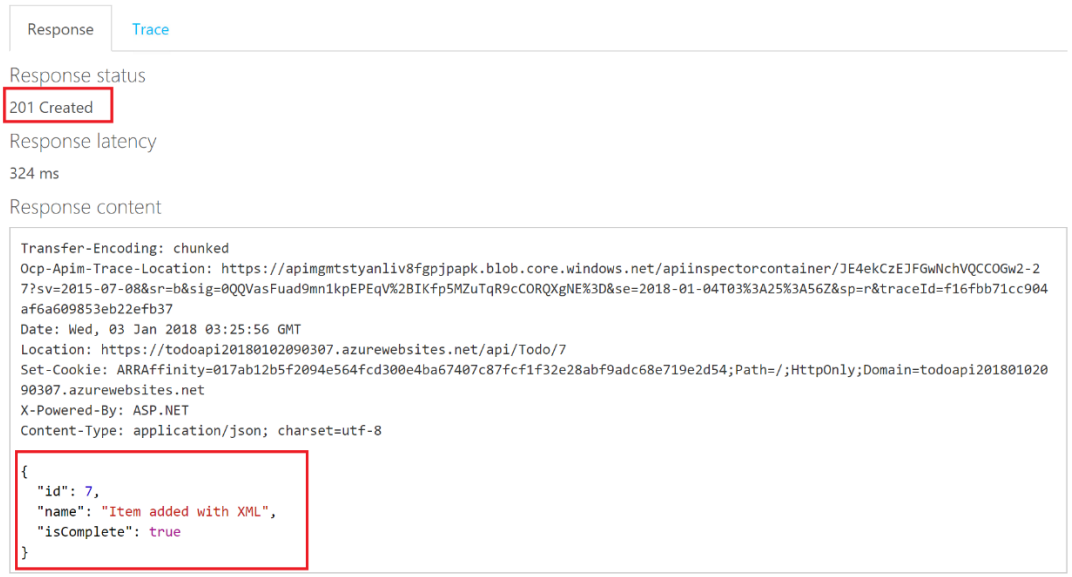
```
<policies>
    <inbound>
        <base />
        <xml-to-json kind="direct" apply="content-type-xml" consider-accept-header="false" />
        <find-and-replace from="{&amp;quot;Document&amp;quot;:" to="" />
        <find-and-replace from="}}" to="}" />
        <rewrite-uri template="/api/Todo" />
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
        <json-to-xml apply="content-type-json" consider-accept-header="true" />
    </outbound>
    <on-error>
        <base />
    </on-error>
</policies>
```

17. Now we can test the operation in the **Developer portal**. Switch to the Developer portal.

18. Click **APIS**, then **Todo API**. Click **Add Todo Item with XML** operation, and click **Try it** button.

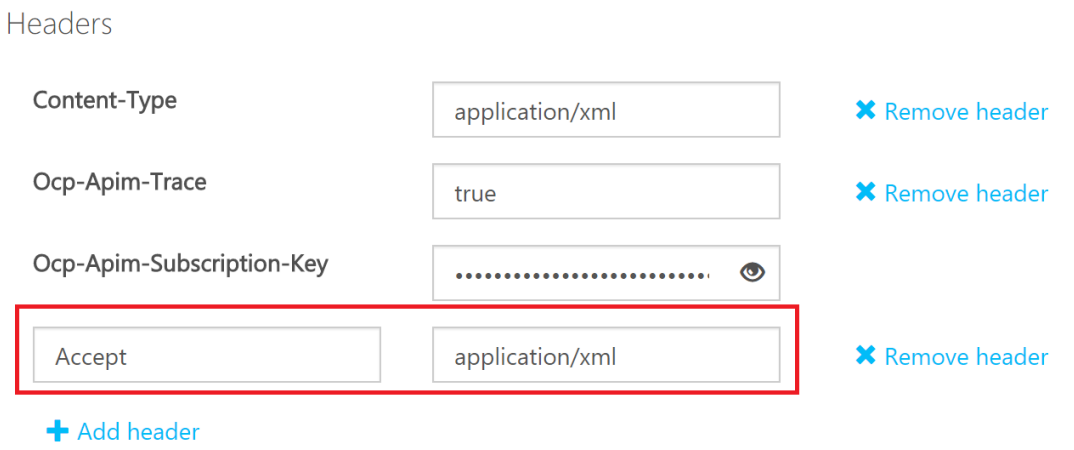
19. Notice that the value of **Content-Type** header is **application/xml**, and the request body is in XML format. Change the value of the **<name>** element in the request body from **String** to **Item added with XML**, and click **Send**.

20. The response you will see is something similar to the following screenshot.

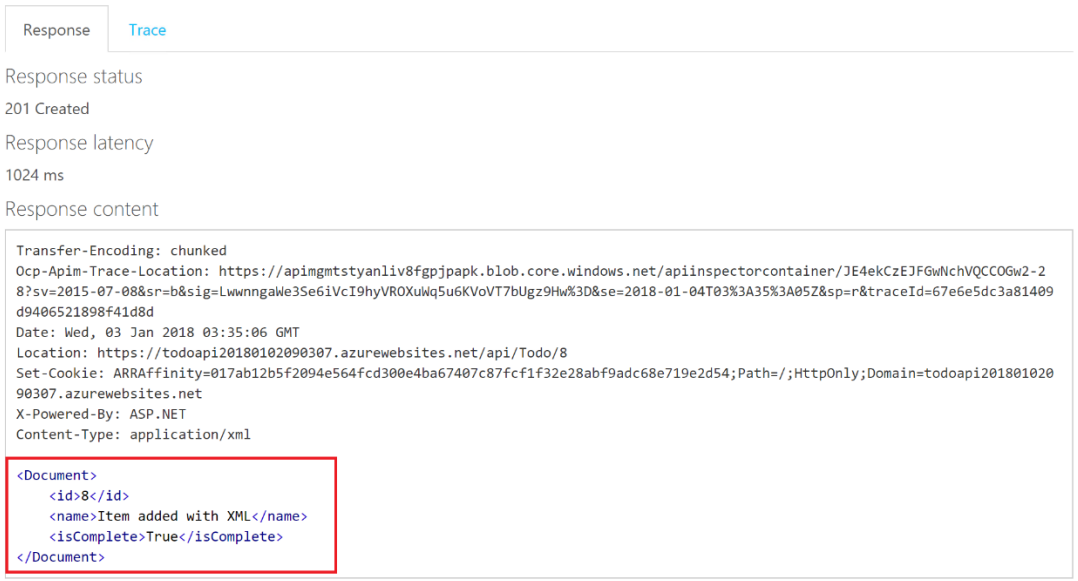


Note that the response is still in JSON format. That is because we configure the json-to-xml statement to respect the **Accept** header from client (with consider-accept-header="true" in the policy statement). If the request doesn't have an Accept header to indicate it can accept XML, the statement won't do the convert.

21. On the **Try it** page, add an **Accept** header and set the value to **application/xml**, as shown in the screenshot below. Click **Send**.



22. With the Accept header, the response is now in XML format, as shown in the following screenshot.



You have completed the Exercise 4.

Conclusion

In this lab, we have covered the following areas of API App and API Management.

- Build and deploy the API App
- Manage API and Operations with API Management

- Secure the API with Azure AD
- Control the behavior of APIs with APIM Policies

We hope you have got the better understanding and the hands-on experiences on the above areas.

There are a lot more regarding to APIM which are not covered by this lab, such as monitoring the APIs, revisions and versions, debugging and troubleshooting, customization of the developer portal, and operations in production etc. We will leave them to you to explore further, and API Management documentation is a good place to start.