**Overview of API Management**

API Management helps organizations **publish APIs to external partner and internal developers** to unlock the potential of their data and services.

You can use Azure API Management as a Gateway service to take any backend and launch a full-fledged API program based on it.

We can quickly create consistent and modern API gateways for existing back-end services hosted anywhere, secure and protect them from abuse and overuse, and get insights into usage and health. Plus, automate and scale developer onboarding to help get your API program up and running.

**Benefits of API Management**

* Work with any host, API, and scale.
* Attract more developers.
* Secure and optimize your APIs.
* Gain insights into your APIs.

To use API Management, administrators create APIs. Each API consists of one or more **operations**, and each **API** can be added to one or more **products**. To use an API, developers **subscribe** to a **product** that contains that API, and then they can call the API's operation, subject to any usage **policies** that may be in effect.

## **APIs and operations**

APIs are the foundation of an API Management service instance. Each API represents a set of operations available to developers. Each API contains a reference to the back-end service that implements the API, and its operations map to the operations implemented by the back-end service. Operations in API Management are highly configurable, with control over URL mapping, query and path parameters, request and response content, and operation response caching. Rate limit, quotas, and IP restriction policies can also be implemented at the API or individual operation level.

**Products:**

* Products are how APIs are provided to developers.
* Products in API Management have one or more APIs, and are configured with a title, description, and terms of use.
* Products can be **Open** or **Protected**. Protected products must be subscribed to before they can be used, while open products can be used without a subscription. Subscription approval is configured at the product level and can either require administrator approval, or be auto-approved.
* When a product is ready for use by developers it can be **published**.
* Groups are used to manage the visibility of products to developers on the developer portal.

**Groups:**

API Management has the following immutable system groups.

* **Administrators** - Azure subscription administrators are of this group. Administrators manage API Management service instances, creating the APIs, operations, and products that are used by developers. members
* **Developers** - Authenticated developer portal users fall into this group. Developers are the customers that build applications using your APIs. Developers are granted access to the developer portal and build applications that call the operations of an API.
* **Guests** - Unauthenticated developer portal users, such as prospective customers visiting the developer portal of an API Management instance fall into this group. They can be granted certain read-only access, such as the ability to view APIs but not call them.

Note: In addition to these system groups, administrators can create custom groups or leverage external groups in associated Azure Active Directory tenants.

**Policies:**

* Policies are a powerful capability of API Management 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.
* Popular statements include format conversion from XML to JSON and call rate limiting to restrict the amount of incoming calls from a developer, and many other policies are available.

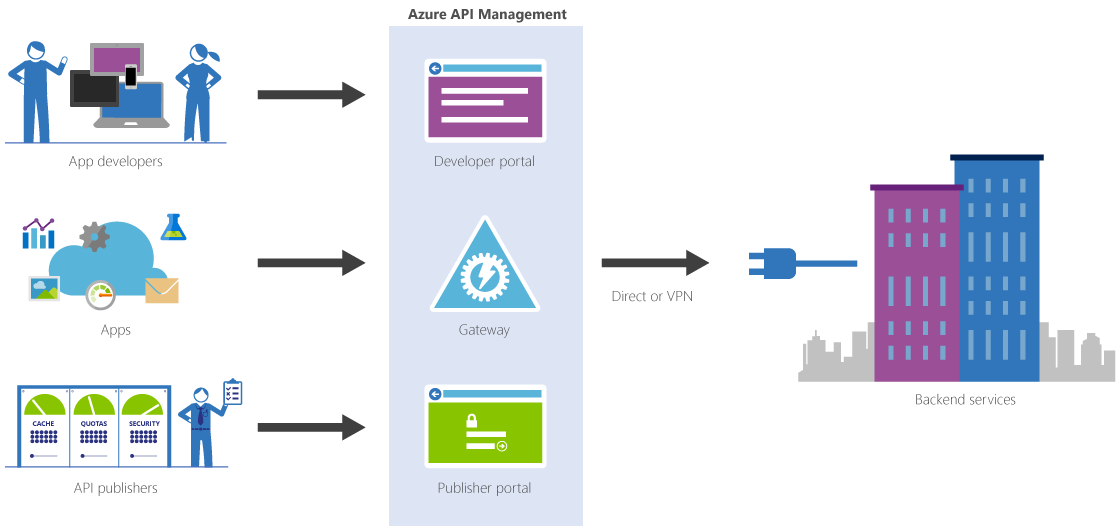
**Developers**

* Developers represent the user accounts in an API Management service instance.
* Developers can be created or invited to join by administrators, or they can sign up from the [Developer portal](https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts#developer-portal).
* Each developer is a member of one or more groups, and can be subscribe to the products that grant visibility to those groups.
* When developers subscribe to a product they are granted the primary and secondary key for the product. This key is used when making calls into the product's APIs.

**Developer portal**

* The developer portal is where developers can learn about your APIs, view and call operations, and subscribe to products.
* You can customize the look and feel of your developer portal by adding custom content, customizing styles, and adding your branding.

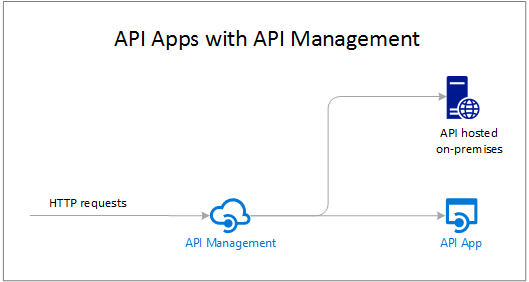
**Components of API Management System**



1. The **API gateway** is the endpoint that:
   * Accepts API calls and routes them to your backends.
   * Verifies API keys, JWT tokens, certificates, and other credentials.
   * Enforces usage quotas and rate limits.
   * Transforms your API on the fly without code modifications.
   * Caches backend responses where set up.
   * Logs call metadata for analytics purposes.
2. ~~The~~ **~~publisher/Azure portal~~** ~~is the administrative interface where you set up your API program. Use it to:~~
   * ~~Define or import API schema.~~
   * ~~Package APIs into products.~~
   * ~~Set up policies like quotas or transformations on the APIs.~~
   * ~~Get insights from analytics.~~
   * ~~Manage users.~~
3. The **developer portal** serves as the main web presence for developers, where they can:
   * Read API documentation.
   * Try out an API via the interactive console.
   * Create an account and subscribe to get API keys.
   * Access analytics on their own usage.

**API Apps Vs Azure API Management**

* **API Management is about managing APIs**. You put an API Management front end on an API to monitor and throttle usage, manipulate input and output, consolidate several APIs into one endpoint, and so forth. The APIs being managed can be hosted anywhere.
* **API Apps is about hosting APIs**. The service includes features that facilitate developing and consuming APIs, but it doesn't do the kinds of monitoring, throttling, manipulating, or consolidating that API Management does. If you don't need API Management features, you can host APIs in API apps without using API Management.



**Note**: Some features of API Management and API Apps have similar functions. For example, both can automate CORS support. When you use the two services together, you would use API Management for CORS since it functions as the front end to your API apps.

**Create Managed API's**

**Create an API Management Instance**

1. Azure Portal 🡪 +New 🡪 Web + Mobile 🡪 API management
2. Name=DssAPIManager, Organization name = "Deccansoft", Pricing Tier="Developer" 🡪 Create.

**This takes around 40+ minutes to show Status = "Online"**

Note: The Developer Tier is for development, testing, and pilot API programs where high availability is not a concern. In the Standard and Premium tiers, you can scale your reserved unit count to handle more traffic. The Standard and Premium tiers provide your API Management service with the most processing power and performance.

**Either**

1. Build a WebAPI App using Visual Studio and Publish to Azure as App Service API App

OR

1. use Conference Demo API: [**http://conferenceapi.azurewebsites.net**/?format=json](http://conferenceapi.azurewebsites.net/?format=json)

**Import an API into our Our Management Service**

1. Select DssAPIManager 🡪 Settings 🡪 API's 🡪 + Add API 🡪 API App
2. Dialog box 🡪 Browse 🡪 Select API App **/ Conference API**
3. API URL suffix = **conference** 🡪 Create
4. APIs 🡪 Select **Conference API** 🡪 Design Tab: You can see here all the operations supported by API App.
5. APIs 🡪 Select **Conference API** 🡪 Settings 🡪 Web service URL = http://conferenceapi.azurewebsites.net

Note: Same steps can be performed from **Publisher Portal** also.

**Subscribe and Testing API Operations**

1. APIs 🡪 Select **Conference API**🡪 Test tab: You can Test here all the operations supported by API App.
2. We can also navigate to Developer portal for testing.

**C# Code for testing the API**

using System;

using System.Net.Http;

using System.Web;

static class Program

{

static void Main()

{

var client = new HttpClient();

var queryString = HttpUtility.ParseQueryString(string.Empty);

client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", "78ea0f8c92884622b1ada0101fc2bf57");

var request = new HttpRequestMessage()

{

RequestUri = new Uri("https://dssdemoapimgmt.azure-api.net/conference/**topics?day=1**")

};

using (var response = client.**SendAsync**(request).Result)

{

Console.WriteLine(response.Content.ReadAsStringAsync().Result);

};

}

}

**Policies in Azure API Management**

* In Azure API Management (APIM), policies are a powerful capability of the system that allow the publisher to **change the behavior of the API through configuration.**
* 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.

**Understanding policy configuration**

* The policy definition is a simple XML document that describes a sequence of inbound and outbound statements. A list of statements is provided to the right and statements applicable to the current scope are enabled and highlighted.
* Clicking an enabled statement will add the appropriate XML at the location of the cursor in the definition view.
* Each policy statement is designed for use in certain scopes and policy sections.

The series of specified policy statements is executes in order for a request and a response

<policies>

<inbound>

<!-- statements to be applied to the request go here -->

</inbound>

<backend>

<!-- statements to be applied before the request is forwarded to the backend service go here -->

</backend>

<outbound>

<!-- statements to be applied to the response go here -->

</outbound>

<on-error>

<!-- statements to be applied if there is an error condition go here -->

</on-error>

</policies>

**Note:** If there is an error during the processing of a request, any remaining steps in the inbound, backend, or outbound sections are skipped and execution jumps to the statements in the on-error section.

**Order of execution of Policies**

**Policy scopes: Product ->** global (All APIs) -> API (All Operations) -> Specific Operation

In the example below, the cross-domain statement would execute before any higher policies which would in turn, be followed by the find-and-replace policy.

<policies>

<inbound>

<cross-domain />

<find-and-replace from="xyz" to="abc" />

<base />

</inbound>

</policies>

**Protect API with rate limit**

1. DssAPIManager 🡪 Products 🡪 + Add
   1. Display name = Free Trail,
   2. Description = "Subscriber will be able to run **10 calls/minute** up to a maximum of **200 calls/week** after which access is denied.
   3. State="Published"
   4. Check Requires subscription
   5. Click on Select API and select Employee 🡪 Select
   6. Create
2. DssAPIManager 🡪 Products 🡪 Free Trail 🡪
   1. Access control 🡪 + Add group 🡪 Check Developers 🡪 Select
   2. **Policies** 🡪 Place cursor after <inbound> 🡪 Click +Limit call rate per subscription and +Set usage quota per subscription
   3. Edit the XML as below 🡪 Save

<policies>

<inbound>

<rate-limit renewal-period="60" calls="10"/>

<quota renewal-period="604800" calls="200"/>

<base/>

</inbound>

. . .

</policies>

1. To Test the throttle,
   1. Navigate to Developer Portal = <https://dssapimanager.portal.azure-api.net>
   2. Register as Developer 🡪 Click on confirmation link in email for verification
   3. Navigate to PRODUCTS Menu and Subscribe to ContactsList
   4. Navigate to APIS Menu 🡪 Select Contacts List 🡪 Try Get Method and notice that it fails with the message "429 Too Many Requests" after 5 requests. Wait for 1 min and try again.

**Restrict Call IP's**

Example: To limit **inbound** requests and accept only those from an IP address or range of IP address:

<ip-filter action="allow | forbid">

<address>123.456.321.123</address>

<address-range from="123.456.321.123" to="123.456.321.255"/>

</ip-filter>

**Check Http header:**

<check-header name="User-Agent" ignore-case="true" failed-check-error-message="User-Agent is missing or invalid" failed-check-httpcode="401">

<value>Mobile</value>

<value>Desktop</value>

<!-- if there are multiple possible allowed values, then add additional value elements -->

</check-header>

**Cross Domain Policies:**

Below policy is for testing the API from Developer Portal:

<cors>

<allowed-origins>

<origin>https://demo-apimgmt.developer.azure-api.net/</origin>

</allowed-origins>

<allowed-methods>

<method>GET</method>

<method>POST</method>

</allowed-methods>

<allowed-headers>

<header>\*</header>

</allowed-headers>

</cors>

<cors>

<allowed-origins>

<origin>\*</origin>

<!-- allow any -->

<!-- OR a list of one or more specific URIs (case-sensitive) -->

<origin>deccansoft.com</origin>

<!-- URI must include scheme, host, and port. If port is omitted, 80 is assumed for http and 443 is assumed for https. -->

</allowed-origins>

<allowed-methods>

<!-- allow any -->

<method>POST</method>

</allowed-methods>

<allowed-headers>

<!-- allow any -->

<header>\*</header>

</allowed-headers>

</cors>

**Protect API with JWT Token**

To generate and verify JWT Token: <http://JWT.io>

To generate base64 string: <https://www.base64encode.org/>

Use the below policy

|  |
| --- |
| **<validate-jwt** failed-validation-error-message="Unauthorized" failed-validation-httpcode="401" header-name="**Authorization**" require-scheme="Bearer">  <issuer-signing-keys>  **<!-- This is base 64 encrypted key -->**  <key>dGhpcyBpcyBhIHNlY3JldCBrZXkgZm9yIGVuY3J5cHRpb24=</key>  </issuer-signing-keys>  <audiences>  <audience>Audience 1</audience>  <audience>Sandeep</audience>  </audiences>  <issuers>  <issuer>Issuer 1</issuer>  <issuer>Issuer 2</issuer>  </issuers>  <required-claims>  <claim name="claim1" match="any">  <value>value1</value>  <value>value2</value>  </claim>  </required-claims>  </validate-jwt> |
| Add the reference to the NuGet Package **Microsoft.IdentityModel.Tokens and System.IdentityModel.Tokens.Jwt;**  using System;  using System.Text;  using System.Net.Http;  using System.Web;  using System.IdentityModel.Tokens.Jwt;  using Microsoft.IdentityModel.Tokens;  static class Program  {  static void Main()  {  // Define const Key this should be private secret key stored in some safe place  string key = "This is my key, and is also secret";  // Create Security key using private key above:  // not that latest version of JWT using Microsoft namespace instead of System  var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));  var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);//.HmacSha256Signature);  // Finally create a Token  var header = new JwtHeader(credentials);  //Some PayLoad that contain information about the customer  var payload = new JwtPayload  {  { "exp", 1714202031}, //This is a mandatory payload  { "iss", "Issuer 1" },  { "aud", "Audience 1"},  { "claim1", "value1"},  };  var secToken = new JwtSecurityToken(header, payload);  var handler = new JwtSecurityTokenHandler();  // Token to String so you can use it in your client  var tokenString = handler.WriteToken(secToken);  //And finally when you received token from client you can either validate it or try to read  //var token = handler.ReadJwtToken(tokenString);  //Console.WriteLine(token.Payload.First().Value);  var client = new HttpClient();  var queryString = HttpUtility.ParseQueryString(string.Empty);  client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", "ff6dea10b5264540846466ffc94e1a02");  var request = new HttpRequestMessage()  {  RequestUri = new Uri("https://dssdemoapimgmt.azure-api.net/conference/speakers")  };  request.Headers.Add("Authorization", "Bearer " + tokenString);  var response = client.SendAsync(request).Result;  Console.WriteLine(response.StatusCode);  }  } |

**Transformation Policies**

Let's see how to transform API so it does not reveal a private backend info. For example, you might want to hide the info about technology stack that is running on the backend. You might also want to hide original URLs that appear in the body of API's HTTP response and instead redirect them to the APIM gateway.

This section shows how to hide the HTTP headers that you do not want to show to your users. In this example, the following headers get deleted in the HTTP response:

* **X-Powered-By**
* **X-AspNet-Version**

1. API Management service 🡪 **EmployeeAPI 🡪 All Operations 🡪 Design**
2. Outbound Processing 🡪 Edit 🡪 Code editor
3. Place cursor after <**outbound**>
4. Click Transformation Policy 🡪 Click twice Set HTTP header 🡪 replace as below

<outbound>

<set-header name="X-Powered-By" exists-action="delete" />

<set-header name="X-AspNet-Version" exists-action="delete" />

</outbound>

1. Test the Response.

**Transformation to find and replace text from response:**

<outbound>

<find-and-replace from="Name" to="EmployeeName"/>

</outbound>

**Policy Expressions**

Policy expressions syntax is C# 6.0. Each expression has access to the implicitly provided [context](https://docs.microsoft.com/en-us/azure/api-management/api-management-policy-expressions#ContextVariables) variable and an allowed [subset](https://docs.microsoft.com/en-us/azure/api-management/api-management-policy-expressions#CLRTypes) of .NET Framework types.

1. **Single statement expressions** are enclosed in @(expression), where expression is a well-formed C# expression statement.
2. **Multi-statement expressions** are enclosed in @{expression}. All code paths within multi-statement expressions **must end with a return statement**.

@(true)

@((1+1).ToString())

@("Hi There".Length)

@(Regex.Match(context.Response.Headers.GetValueOrDefault("Cache-Control",""), @"max-age=(?<maxAge>\d+)").Groups["maxAge"]?.Value)

@(context.Variables.ContainsKey("maxAge") ? int.Parse((string)context.Variables["maxAge"]) : 3600)

@{

string value;

if (context.Request.Headers.TryGetValue("Authorization", out value))

{

return Encoding.UTF8.GetString(Convert.FromBase64String(value));

}

else

{

return null;

}

}

API Reference:

<https://docs.microsoft.com/en-us/azure/api-management/api-management-policy-expressions#PolicyExpressionsExamples>

**Note**: Expressions can be used as attribute values or text values in any of the API Management policies, unless the policy reference specifies otherwise.

**Example: To add custom header and query parameters:**

<policies>

<inbound>

<set-header name="x-product-name" exists-action="override">

<value>@(context.Product.Name)</value>

</set-header>

<set-query-parameter name="x-username-location" exists-action="override">

<value>@(context.User.Id)</value>

<value>@(context.Deployment.Region)</value>

</set-query-parameter>

<base />

</inbound>

</policies>

**To dynamically set backend service URL:**

<policies>

<inbound>

<choose>

<when condition="@(context.Request.Url.Query.GetValueOrDefault("version") == "2013-05")">

<set-backend-service base-url="http://dssapidemo.azurewebsites.net/api/8.2/" />

</when>

<when condition="@(context.Request.Url.Query.GetValueOrDefault("version") == "2014-03")">

<set-backend-service base-url="http://dssapidemo.azurewebsites.net/api/9.1/" />

</when>

</choose>

<base />

</inbound>

</policies>

**Using Variables and expressions**

This example shows how to use the Validate JWT policy to pre-authorize access to operations based on token claims.

<set-variable name="signingKey" value="insert signing key here" />

<choose>

<when condition="@(context.Request.Method.Equals("patch",StringComparison.OrdinalIgnoreCase))">

<validate-jwt header-name="Authorization">

<issuer-signing-keys>

<key>@((string)context.Variables["signingKey"])</key>

</issuer-signing-keys>

<required-claims>

<claim name="edit">

<value>true</value>

</claim>

</required-claims>

</validate-jwt>

</when>

<when condition="@(new [] {"post", "put"}.Contains(context.Request.Method,StringComparer.OrdinalIgnoreCase))">

<validate-jwt header-name="Authorization">

<issuer-signing-keys>

<key>@((string)context.Variables["signingKey"])</key>

</issuer-signing-keys>

<required-claims>

<claim name="create">

<value>true</value>

</claim>

</required-claims>

</validate-jwt>

</when>

<otherwise>

<validate-jwt header-name="Authorization">

<issuer-signing-keys>

<key>@((string)context.Variables["signingKey"])</key>

</issuer-signing-keys>

</validate-jwt>

</otherwise>

</choose>

**Add Caching to Improve Performance**

Operations in API Management can be configured for response caching. Response caching can significantly reduce API latency, bandwidth consumption, and web service load for data that does not change frequently.

1. Select the method on which you want to provide caching.
2. Inbound processing 🡪 Code editor 🡪 Include two caching policies
   1. Get from cache
   2. Store to cache
3. Edit the XML as below

<policies>

<inbound>

<cache-lookup vary-by-developer-groups="false" vary-by-developer="false" must-revalidate="false" downstream-caching-type="public">

<vary-by-header>Accept</vary-by-header>

<!-- should be present in most cases -->

<vary-by-header>Accept-Charset</vary-by-header>

<!-- should be present in most cases -->

<vary-by-header>Authorization</vary-by-header>

<vary-by-query-parameter>parameter name</vary-by-query-parameter>

<!-- optional, can repeated several times -->

</cache-lookup>

<base/>

</inbound>

<outbound>

<base/>

<cache-store caching-mode="cache-on" duration="3600"/>

</outbound>

</policies>

1. Test the Method –Test Tab.
   1. First Execute Get\_All
   2. Delete a Contact by executing DeleteById
   3. Again, call Get\_All and not that its listing deleted contact also as the data is fetched from cache.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Required** | **Default** |
| allow-private-response-caching | When set to true, allows caching of requests that contain an Authorization header. | No | false |
| downstream-caching-type | This attribute must be set to one of the following values. - **none** - downstream caching is not allowed. - **private** - downstream private caching is allowed. - **public** - private and shared downstream caching is allowed. | No | none |
| must-revalidate | When downstream caching is enabled this attribute turns on or off the must-revalidate cache control directive in gateway responses. | No | true |
| vary-by-developer | Set to true to cache responses per developer key. | Yes |  |
| vary-by-developer-groups | Set to true to cache responses per user role. | Yes |  |

This this snippet into the outbound section. Note that cache duration is set to the max-age value provided in the Cache-Control header received from the backend service or to the default value of **5 min** if none is found:

<cache-store duration="@{

var header = context.Response.Headers.GetValueOrDefault("Cache-Control","");

var maxAge = Regex.Match(header, @"max-age=(?<maxAge>\d+)").Groups["maxAge"]?.Value;

return (!string.IsNullOrEmpty(maxAge))?int.Parse(maxAge):300;

}"

/>

**Managing Properties / Named Values**

Each API Management service instance has a properties collection of key/value pairs that are global to the service instance.

These properties can be used to manage constant string values across all API configuration and policies.

1. API Management service 🡪 **Named values 🡪 +Add**
2. Add Following Name Value Pairs
   1. TrackingId = "ABCD-EFGH-IJKL-LMNO"
   2. **SystemTime = @(DateTime.Now.ToString())**
   3. **ClientAuthorized = @(context.Request.Headers.ContainsKey("Authorization") ? "Yes" : "No")**
3. **User the property in Policies**

<set-header exists-action="override" name="Authorized">

<value>**{{ClientAuthorized}}**</value>

</set-header>

<set-header exists-action="append" name="TrackingId">

<value>**{{TrackingId}}**</value>

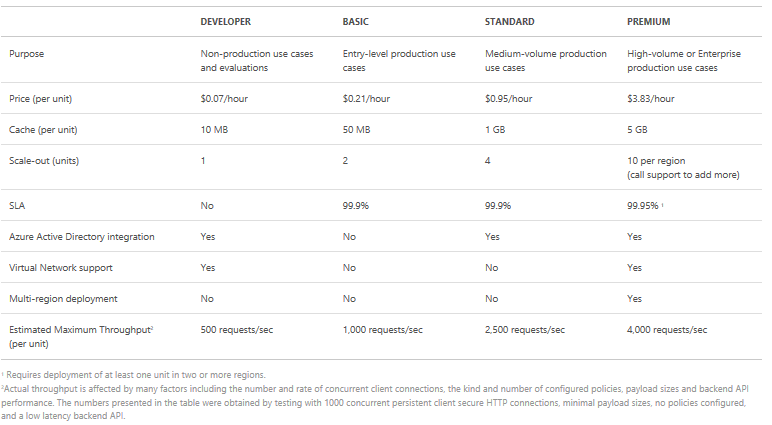
</set-header>

**Customize the Developer Portal**

1. **Developer portal** 🡪 **Expand icon on left side of the screen** 🡪 **Styles**
2. In Change variable values to customize developer portal appearance = "**headings-color**". This controls the color of the text
3. Pickup an appropriate color
4. Select Publish from the lower left to preview locally
5. Select **Publish customizations** to make the changes publicly available.

**Upgrade and scale**

* Customers can scale an API Management (APIM) instance by adding and removing **units**.
* A **unit** is composed of dedicated Azure resources and has a certain load-bearing capacity expressed as a **number of** **API calls per minute**.
* Capacity and price of each unit depends on the **tier** in which the unit exists. You can choose between four tiers: **Developer**, **Basic**, **Standard**, **Premium**.
* If you need to increase capacity for a service within a tier, you should add a unit. If the tier that is currently selected in your APIM instance does not allow adding more units, you need to upgrade to a higher-level tier.



### Use the Azure portal to upgrade and scale

1. Navigate to your APIM instance in the Azure portal.
2. Select **Scale and pricing**.
3. Pick the desired tier.
4. Specify the number of **units** you want to add. You can either use the slider or type the number of units.  
   If you choose the **Premium** tier, you first need to select a region.
5. Press **Save**

**Configure a custom domain name**

There is a number of endpoints to which you can assign a custom domain name. Currently, the following endpoints are available:

* **Proxy** (default is: <apim-service-name>.azure-api.net),
* **Portal** (default is: dssapimgmtdemo.portal.azure-api.net),
* **Management** (default is: <apim-service-name>.management.azure-api.net),
* **SCM** (default is: <apim-service-name>.scm.azure-api.net).

Commonly, customers update **Proxy** (this URL is used to call the API exposed through API Management) and **Portal** (the developer portal URL).

**Management** and **SCM** endpoints are used internally by APIM customers and thus are less frequently assigned a custom domain name.

1. **Generate a client certificate: Execute the following command in the same PowerShell window opened earlier**

**New-SelfSignedCertificate** -Type Custom -KeySpec Signature `

-Subject "CN=\*.bestazuretraining.com" -KeyExportPolicy Exportable `

-HashAlgorithm sha256 -KeyLength 2048 `

-CertStoreLocation "Cert:\CurrentUser\My" `

-Signer $cert -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.3.2")

1. To obtain the public key (.pfx file)
   1. MMC 🡪 File 🡪 Add/Remove Snap-In 🡪 Certificates 🡪 Add 🡪 OK
   2. Include Private Key 🡪 . . . 🡪 Save PFX file.
2. Map it to APIM
   1. In the **Custom domain name**, specify the name you want to use. For example, api.bestazuretraining.com.   
      Wildcard domain names (for example, \*.domain.com) are also supported.
   2. In the **Certificate**, specify a valid .PFX file that you want to upload.
   3. If the certificate has a password, enter it in the **Password** field.
3. Save.

Note: The process of assigning the certificate may take 15 minutes or so.

**Authentication with AAD**

**Register an Azure AD Application**

1. Azure Active Directory 🡪 App Registration 🡪 New Application Registration
2. Name = APIManagementADApp, Application type=Web app/ API, Sign-On URL = <Sign-In URL of Developer Portal of API Management App>
3. Create
4. Select the New App
   1. Properties 🡪 Copy Application ID to Clipboard / Notepad
   2. Keys 🡪 Generate the New Secret and copy the same to Notepad
   3. Reply URL = <Sign-In URL of Developer Portal>**-aad (eg** [**https://dssdemoapimgmt.portal.azure-api.net/signin-aad**](https://dssdemoapimgmt.portal.azure-api.net/signin-aad)**)**
   4. Required permissions 🡪 Click Windows Azure Active Directory 🡪 Check Application permissions, Read directory data 🡪 Check Delegated Permissions, Sign in and read user profile 🡪 Save 🡪 Grant Permissions 🡪 Yes

**Register Azure AD Application with API management Service**

1. Select API Management App 🡪 Identities (Securities) 🡪 + Add
2. Provider type = Azure Active Directory, Client ID = <Copy Application ID from Notepad>, Client Secret = <Copy secret from notepad>, Allowed tenants = <Azure AD tenant id> eg: myazureaccount.onmicrosoft.com
3. Add

Note: Multiple domains can be specified in the **Allowed Tenants** section. Before any user can log in from a different domain than the original domain where the application was registered, a global administrator of the different domain must grant permission for the application to access directory data. To grant permission, the global administrator should go to https://<URL of your developer portal>/aadadminconsent (for example, <https://contoso.portal.azure-api.net/aadadminconsent>), type in the domain name of the Active Directory tenant they want to give access to and click Submit.

**Add an external Azure Active Directory Group to API Management Service**

1. API Management Service 🡪 Groups 🡪 + Add AAD Group
2. Select the Custom Group from Azure AD 🡪 OK
3. API Management Service 🡪 Products 🡪 Select Unlimited 🡪 Access control 🡪 +Add group 🡪 Check the new AD group 🡪 Select

**How to log in to the Developer portal using an Azure Active Directory account**

1. Visit the Sign-In page of the Developer portal
2. click **Azure Active Directory**.
3. Provide the Username and Password for the account create in Azure AD using default domain name.

Reference: <https://docs.microsoft.com/en-in/azure/api-management/api-management-howto-aad>

**Authentication with AAD B2C:**

<https://docs.microsoft.com/en-in/azure/api-management/api-management-howto-aad-b2c>