API Management Service

- Overview of API Management
- Components of API Management System.
- API Apps vs API Management.
- Create managed APIs.
- Configure API management policies.
- Protect APIs with rate limits.
- Add caching to improve performance.

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

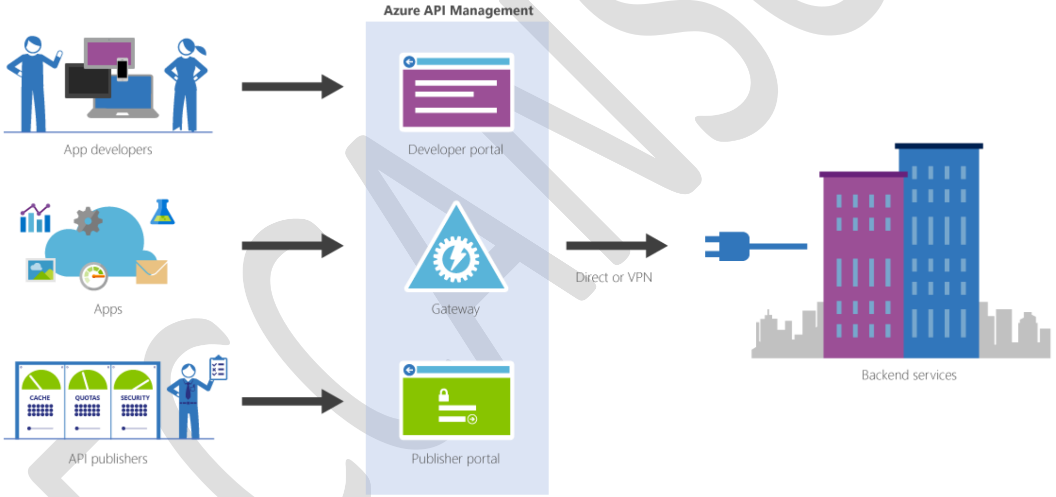
- Developers represent the user accounts in an API Management service instance.

- Developers can be created or invited to join by administrators, or they can sign up from the Developer portal.
- 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:~~

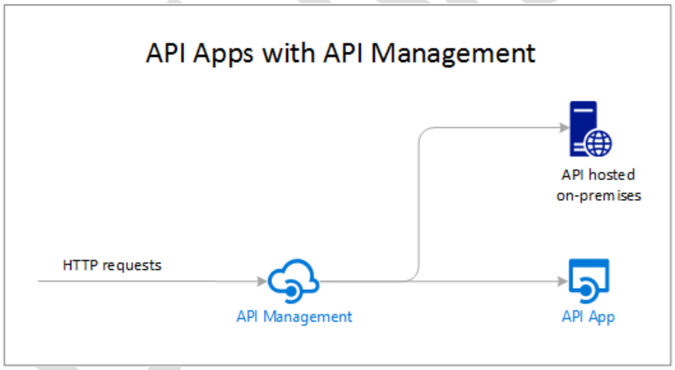
- ~~Define or import API schema.~~
- ~~Package APIs into products.~~

- ~~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 30+ 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

2. use Conference Demo API: **http://conferenceapi.azurewebsites.net**/?format=json

Import an API into our Our Management Service

3. Select DssAPIManager → Settings → API's → + Add API → API App
4. Dialog box → Browse → Select API App **/ Conference API**
5. API URL suffix = **conference** → Create
6. APIs → Select **Conference API** → Design Tab: You can see here all the operations supported by API App.
7. 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

8. APIs → Select **Conference API**→ Test tab: You can Test here all the operations supported by API App.
9. 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);
        };
    }
}
```

Mock an API response

Create a Test API

1. Select **APIs** from under **API MANAGEMENT**.
2. From the left menu, select **+ Add API**.
3. Select **Blank API** from the list.
4. Enter "*Test API*" for **Display name**.
5. Enter "*Unlimited*" for **Products**.
6. Select **Create**.

Add an Operation to the test API

7. Click **+ Add Operation**.
    a. URL: Get /test
    b. Display Name: Test Call
    c. Description: This is a demo for test operation
    d. Query tab: You can add query parameters. Besides providing a name and description, you can provide values that can be assigned to this parameter. One of the values can be marked as default (optional).
    e. Request tab: You can define request content types, examples and schemas.
    f. Response Tab:
        i. + Add response → 200 OK
        ii. Representations → +Add representation → Enter "application/json"
        iii. In the Sample text box enter {'Field1':'value1';'Field2':'value2'}
        iv. Save
8. Enable Response Mocking
    a. Test Operation → Design → Inbound processing → pencil icon

b. Mocking tab → Static responses for mocking behavior

c. API Management returns the following response = 200 OK, application/json

9. Test the mocked API

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

```xml
<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 />
    <base />
    <find-and-replace from="xyz" to="abc" />
  </inbound>
</policies>
```

Protect API with rate limit

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

```
<policies>
  <inbound>
    <rate-limit renewal-period="60" calls="10"/>
    <quota renewal-period="604800" calls="200"/>
    <base/>
  </inbound>
  . . .
</policies>
```

3. To Test the throttle,

   a. Navigate to Developer Portal = https://dssapimanager.portal.azure-api.net

   b. Register as Developer → Click on confirmation link in email for verification

   c. Navigate to PRODUCTS Menu and Subscribe to ContactsList

   d. 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);

     }

  }
```

Use client certificates to secure access to an API

1. Create Self-Signed Certificate:

```
pwd='Pa$$w0rd'

pfxFilePath='selfsigncert.pfx'

openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout privateKey.key -out selfsigncert.crt -subj
/CN=localhost
```

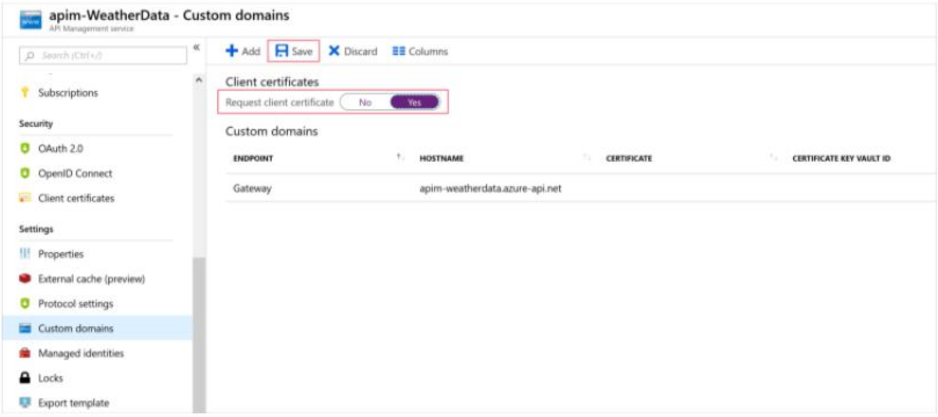
2. Convert to PEM format which the curl tool can use:

```
openssl pkcs12 -export -out $pfxFilePath -inkey privateKey.key -in selfsigncert.crt -password pass:$pwd

openssl pkcs12 -in selfsigncert.pfx -out selfsigncert.pem -nodes
```

3. Configure the gateway to request client certificates

API Management → Custom domains → Request Client certificate = Yes.



apim-WeatherData - Custom domains

API Management service

Client certificates

Request client certificate: No [✓ Yes]

Custom domains

ENDPOINT	HOSTNAME	CERTIFICATE	CERTIFICATE KEY VAULT ID
Gateway	apim-weatherdata.azure-api.net		

4. Get the thumbprint for the certificate

```
Fingerprint="$(openssl x509 -in selfsigncert.pem -noout -fingerprint)"

Fingerprint="${Fingerprint//:}"

echo ${Fingerprint#*=}
```

5. Copy the complete output from the final command to a text file. The output should be a hexadecimal string without any accompanying text and no colons

6. Edit inbound policy to only allow requests with a valid certificate

API → Select API → Inboud processing → +Add Policy

Replace the <inbound> node of the policy file with the following XML, substituting the thumbprint you copied earlier for desired-thumbprint:

```
<inbound>
  <choose>
    <when condition="@(context.Request.Certificate == null || context.Request.Certificate.Thumbprint != "desired-thumbprint")" >
      <return-response>
        <set-status code="403" reason="Invalid client certificate" />
      </return-response>
    </when>
  </choose>
  <base />
</inbound>
```

7. Call the gateway and pass the client certificate

   a) To test the API without the certificate, run the following command within the Cloud Shell.

   ```
   curl -X GET https://[api-gateway-name].azure-api.net/api/Weather/53/-1 \
     -H 'Ocp-Apim-Subscription-Key: [Subscription Key]'
   ```

   This command should return a 403 Client certificate error, and no data will be returned.

   b) In the Azure Cloud Shell, to test the API with the certificate, copy and paste the following cURL command, using the subscription key from the first exercise:

   ```
   curl -X GET https://[gateway-name].azure-api.net/api/Weather/53/-1 \
     -H 'Ocp-Apim-Subscription-Key: [subscription-key]' \
     --cert-type pem \
     --cert selfsigncert.pem
   ```

```
var cert = new X509Certificate2("d:\\demo.pfx", "1234");
var _clientHandler = new HttpClientHandler();
_clientHandler.ClientCertificateOptions = ClientCertificateOption.Manual;
//_clientHandler.SslProtocols = SslProtocols.Tls12;
_clientHandler.ClientCertificates.Add(cert);

var _client = new HttpClient(_clientHandler);
```

More: Configure certificate authentication in ASP.NET Core | Microsoft Docs

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>
```

5. 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 variable and an allowed subset 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
   a. Get from cache
   b. 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>
```

4. Test the Method –Test Tab.

   a. First Execute Get_All

   b. Delete a Contact by executing DeleteById

   c. 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.	No	none

	- **private** - downstream private caching is allowed. - **public** - private and shared downstream caching is allowed.		
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
   - **a.** TrackingId = "ABCD-EFGH-IJKL-LMNO"
   - **b. SystemTime = @(DateTime.Now.ToString())**
   - **c. 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>
```

Monitoring API's

1. Activity Log
2. Diagnostics Log
3. View Metrics
4. Setup Alerts

About Activity Logs

1. Activity logs provide insight into the operations that were performed on your API Management services.
2. Using activity logs, you can determine the "what, who, and when" for any write operations (PUT, POST, DELETE) taken on your API Management services.
3. Activity logs **do not** include read (GET) operations or operations performed in the classic Publisher Portal or using the original Management APIs.

About Diagnostics Logs

1. Diagnostic logs provide rich information about operations and errors that are important for auditing as well as troubleshooting purposes.
2. Diagnostics logs differ from activity logs. Activity logs provide insights into the operations that were performed on your Azure resources. Diagnostics logs provide insight into operations that your resource performed itself.

View metrics of your APIs

API Management emits metrics every minute, giving you near real-time visibility into the state and health of your APIs.

Setup Alert rule

You can configure to receive alerts based on metrics and activity logs. Azure Monitor allows you to configure an alert to do the following when it triggers:

- Send an email notification
- Call a webhook
- Invoke an Azure Logic App

Handling Revisions

When your API is ready to go and starts to be used by developers, you usually need to take care in making changes to that API and at the same time not to disrupt callers of your API.

It's also useful to let developers know about the changes you made.

To Add a new revision

1. EmployeeAPI → Revisions Tab → +Add revision, set Description → Create

Note: Notice that the **revision selector** (directly above the design tab) shows your current revision as **Revision 2**.

2. Clone any existing Method
3. We have now made a change to **Revision 2**. Use the **Revision Selector** near the top of the page to switch back to **Revision 1**.
4. Notice that your new operation does not appear in **Revision 1**.

Make your revision current and add a change log entry

1. Select the **Revisions** tab from the menu near the top of the page.
2. Open the context menu (**...**) for **Revision 2**.
3. Select **Make Current**. Check **Post to Public Change log for this API**, if you want to post notes about this change.
4. Select **Post to Public Change Log for this API**
5. Provide a description for your change that developers see, for example **Testing revisions. Added new "test" operation.**
6. **Revision 2** is now current.
7. Note changes: Developer portal → APIS → Select API → API Change History

Handling Multiple Versions

There are times when it is **impractical** to have **all callers** to your API **use exactly the same version**.

Sometimes you want to publish new or different API features to some users, while others want to stick with the API that currently works for them. When callers want to upgrade to a later version, they want to be able to do this using an easy to understand approach.

Add a new version

1. EmployeeAPI → context menu (…) → + Add Version
2. Version Schema = Path, Version identifier=v1, Products is Optional
3. Create
4. Browse the developer portal to see the version.

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.

Error Handling

Policies allowed in on-error

- choose
- set-variable
- find-and-replace
- return-response
- set-header
- set-method
- set-status
- send-request
- send-one-way-request
- log-to-eventhub
- json-to-xml
- xml-to-json

When an error occurs and control jumps to the on-error policy section, the error is stored in **context.LastError** property:

Properties of LastError: Source, Reason, Message, Scope, Section, Path, PolicyId

Following are predefined error policies

1. rate-limit
2. quota
3. jsonp
4. ip-filter
5. check-header
6. validate-jwt

Step1: Add the following to any operation.

```
<on-error>
  <set-header exists-action="override" name="http-error-reason">
    <value>@(context.LastError.Reason)</value>
```

```
   <!-- for multiple headers with the same name add additional value elements -->
  </set-header>
  <set-body>@(context.LastError.Message)</set-body>
  <base />
 </on-error>
```

Step2: Test the method with incorrect subscription-id:

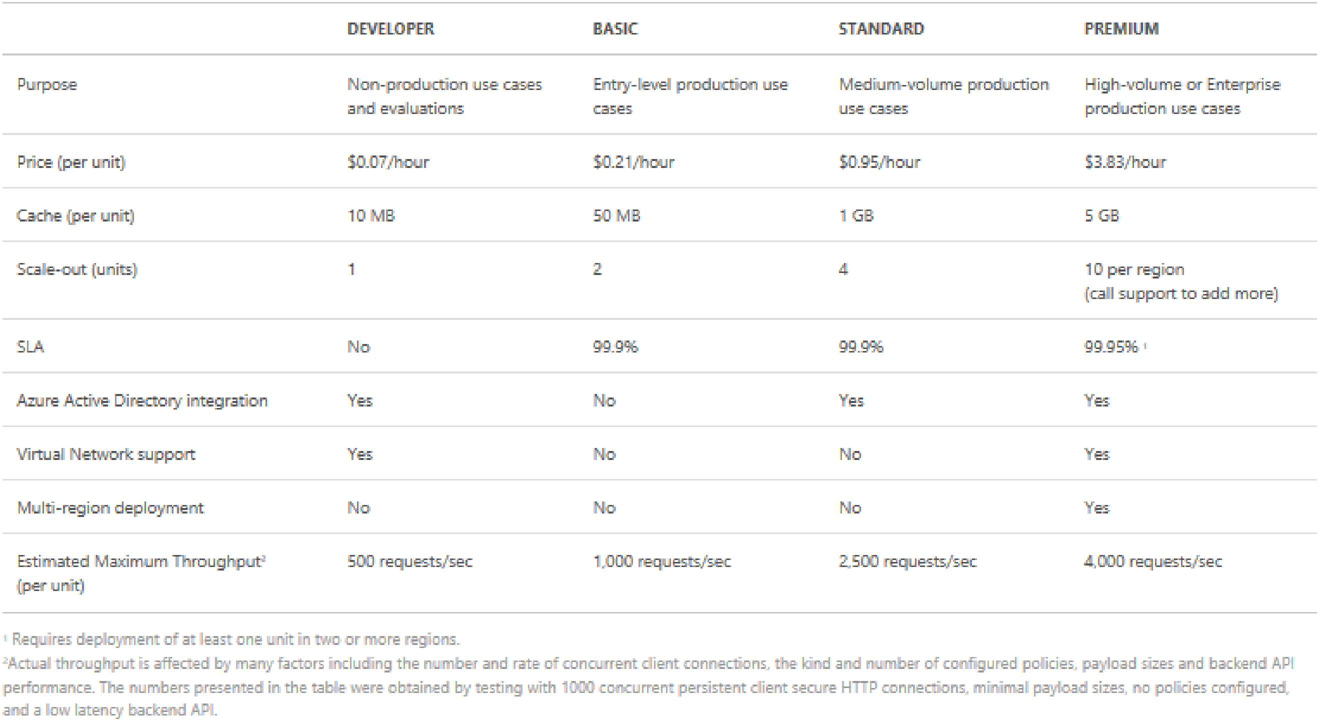
Example 2: Logging Error to Skackify

```
<!-- The policy defined in this file shows how to add an error logging policy to send errors to Stackify for logging. -->
<!-- You must specify the following named values: -->
 <!-- your-stackify-api-key  -->
 <!-- stackify-api-key  - Get this from your Stackify account -->
 <!-- environment-name - This will be send to Stackify for the environment filter, (Production, Dev, Test, etc) -->
 <!-- app-name - The Application name that will be sent to Stackify -->
<!-- Copy the following snippet into the on-error section. -->
<policies>
 <inbound>
  <base />
 </inbound>
 <backend>
  <base />
 </backend>
 <outbound>
  <base />
 </outbound>
 <on-error>
  <trace source="OnError">
   On Error
  </trace>
  <send-one-way-request mode="new">
   <set-url>https://api.stackify.com/Log/Save</set-url>
   <set-method>POST</set-method>
   <set-header name="Content-Type" exists-action="override">
    <value>value</value>
   </set-header>
   <set-header name="X-Stackify-PV" exists-action="override">
    <value>V1</value>
   </set-header>
   <set-header name="X-Stackify-Key" exists-action="override">
    <value>{{stackify-api-key}}</value>
   </set-header>
   <set-body>
    @{
    return new JObject(
    new JProperty("Environment","{{environment-name}}"),
    new JProperty("ServerName", context.Deployment.ServiceName),
    new JProperty("AppName", "{{app-name}}"),
    new JProperty("AppLoc", "/usr/local/stackify/stackify-agent"),
    new JProperty("Logger", "stackify-log-log4j12-1.0.12"),
    new JProperty("Platform", "java"),
    new JProperty("Msgs",
    new JArray(
    new JObject(
    new JProperty("Msg", context.LastError.Message),
    new JProperty("Th", "main"),
    new JProperty("EpochMs", (new DateTimeOffset(DateTime.Now)).ToUnixTimeSeconds() * 1000 ),
    new JProperty("Level", "error"),
    new JProperty("SrcMethod", context.LastError.Source)
    )))
    ).ToString();
    }
   </set-body>
  </send-one-way-request>
 </on-error>
</policies>
```

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.



	DEVELOPER	BASIC	STANDARD	PREMIUM
Purpose	Non-production use cases and evaluations	Entry-level production use cases	Medium-volume production use cases	High-volume or Enterprise production use cases
Price (per unit)	$0.07/hour	$0.21/hour	$0.95/hour	$3.83/hour
Cache (per unit)	10 MB	50 MB	1 GB	5 GB
Scale-out (units)	1	2	4	10 per region (call support to add more)
SLA	No	99.9%	99.9%	99.95% [1]
Azure Active Directory integration	Yes	No	Yes	Yes
Virtual Network support	Yes	No	No	Yes
Multi-region deployment	No	No	No	Yes
Estimated Maximum Throughput[2] (per unit)	500 requests/sec	1,000 requests/sec	2,500 requests/sec	4,000 requests/sec

[1] Requires deployment of at least one unit in two or more regions.

[2] Actual throughput is affected by many factors including the number and rate of concurrent client connections, the kind and number of configured policies, payload sizes and backend API performance. The numbers presented in the table were obtained by testing with 1000 concurrent persistent client secure HTTP connections, minimal payload sizes, no policies configured, and a low latency backend API.

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")
```

2. To obtain the public key (.pfx file)
    1. MMC → File → Add/Remove Snap-In → Certificates → Add → OK
    2. Include Private Key → . . . → Save PFX file.
3. 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.
4. 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)**
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

5. Select API Management App → Identities (Securities) → + Add
6. 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
7. 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

8. API Management Service → Groups → + Add AAD Group
9. Select the Custom Group from Azure AD → OK
10. 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

11. Visit the Sign-In page of the Developer portal
12. click **Azure Active Directory**.
13. 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

Delegated Authentication

Documentation:

https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-setup-delegation

Complete Sample Code can be downloaded from:

https://github.com/Azure/api-management-samples/tree/master/delegation

Get the Shared Signature:

1. API Management Service → Management API (Security) → Check Enable API Management REST API → Click Generate → Copy the Access token

Specify the Login URL:

2. API Management Service → Delegation (Security) → Delegation endpoint URL = <Custom Login Page URL> eg: http://localhost:52226/Login
3. Create an ASP.NET MVC application with LoginController (Sample Code for understanding only the basics).

```
using Newtonsoft.Json;
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using System.Web.Mvc;

namespace MyAPIDemoApp.Controllers
{
    public class LoginController : Controller
    {
        // GET: Logoin
        public ActionResult Index(string operation, string returnUrl, string salt, string sig)
        {
            //string s = $"{operation} - {returnUrl} - {salt} - {sig}";
            return View();
        }
```

```
public static string SerializeToJson<TAnything>(TAnything value)
{
    var settings = new JsonSerializerSettings { DateTimeZoneHandling = DateTimeZoneHandling.Utc };
    var formatting = Formatting.None;
    var writer = new StringWriter();

    var serializer = JsonSerializer.Create(settings);
    var jsonWriter = new JsonTextWriter(writer) { Formatting = formatting };

    serializer.Serialize(jsonWriter, value);
    return writer.GetStringBuilder().ToString();
}

public static TAnything DeserializeToJson<TAnything>(String value)
{
    var settings = new JsonSerializerSettings { DateTimeZoneHandling = DateTimeZoneHandling.Utc };
    var reader = new StringReader(value);

    var serializer = JsonSerializer.Create(settings);
    var jsonReader = new JsonTextReader(reader);

    return (TAnything)serializer.Deserialize(jsonReader, typeof(TAnything));
}
public class SsoUrl
{
    public string value;
}
[HttpPost]
public async Task<ActionResult> Index(FormCollection col)
{
    string token = "SharedAccessSignature
integration&201801251311&YYF9t4Cc/R9iGd5D1zeg8aK6RKQrHSleaDjdUV6eEv3gaXRZj2NhgdcWXncaBYbdWaPFBdn6ozx
Nb4PqFB8eRQ==";
```

```
        string baseUrl = "https://dssapimgmt.management.azure-api.net/";
        //Create a User
        using (var clientForNewUser = new HttpClient())
        {
            clientForNewUser.BaseAddress = new Uri(baseUrl);
            clientForNewUser.DefaultRequestHeaders.Add("Authorization", token);
            clientForNewUser.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("text/json"));

            //This is to provide unique id for each user
            string uid = Guid.NewGuid().ToString();

            //User Object required for adding a new user to API Management Service
            var ApimUser = new
            {
                firstName = uid + "-FUser",
                lastName = "LUser",
                email = $"User-{uid}@gmail.com",
                password = "Abcd@1234",
                state = "active"
            };

            var ApimUserJson = SerializeToJson(ApimUser);
            HttpResponseMessage response = await clientForNewUser.PutAsync("/users/" + uid + "?api-version=2014-02-14-
preview", new StringContent(ApimUserJson, Encoding.UTF8, "text/json"));
            if (response.IsSuccessStatusCode)
            {
                //User created successfully
                var clientToLoginUsingSSO = new HttpClient();
                clientToLoginUsingSSO.BaseAddress = new Uri("https://dssapimgmt.management.azure-api.net/");

                clientToLoginUsingSSO.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("text/json"));
                clientToLoginUsingSSO.DefaultRequestHeaders.Add("Authorization", token);
```

```
                    var resp = await clientToLoginUsingSSO.PostAsync("/users/" + uid + "/generateSsoUrl?api-version=2017-03-01",
new StringContent("", Encoding.UTF8, "text/json"));
                    if (resp.IsSuccessStatusCode)
                    {
                        HttpContent receiveStream = resp.Content;
                        var SsoUrlJson = await receiveStream.ReadAsStringAsync();
                        var su = DeserializeToJson<SsoUrl>(SsoUrlJson);
                        Response.Redirect(su.value);
                    }
                }
                else
                {
                    @ViewBag.message = "APIM REST Connection Error: " + response.StatusCode;
                }
            }
            return View();
        }
    }
}
```

Dealing with Secured Web API

How to protect a Web API backend with Azure Active Directory and API Management

- Create an Azure AD directory
- Create a Web API service secured by Azure Active Directory
- Add the code to the Web API project
- Publish the project to Azure
- Grant permissions to the Azure AD backend service application
- Import the Web API into API Management
- Call the API unsuccessfully from the developer portal
- Register the developer portal as an AAD application
- Configure an API Management OAuth 2.0 authorization server
- Enable OAuth 2.0 user authorization for the Calculator API
- Successfully call the Calculator API from the developer portal

- Configure a desktop application to call the API
- Configure a JWT validation policy to pre-authorize requests

To configure notifications and email templates

1. API Management Service → Notifications
2. API Management Service → Notification templates