Table of Contents

- Overview
  - What is API Management?
- Get started
  - Manage your first API
  - Protect your API with rate limits
  - Add caching to improve performance
  - Frequently asked questions
- How to
  - Develop APIs
    - Add and publish an API Product
    - Add operations
    - Create an API
    - Import an API definition
  - Secure your backend
    - Protect Web API backend with AAD
    - Connect via VPN or ExpressRoute
    - Mutual Certificate authentication
  - Configure Policies
    - Custom caching
    - Advanced monitoring
    - Advanced request throttling
    - Using external services
    - Policy reference index
    - Policies overview
    - Manage secrets using properties
  - Customize the developer experience
    - Customize the Developer Portal
    - Authentication with AAD
    - Delegated authentication

- E-mail notifications and templates
- Enable console OAuth support
- Customize using templates
- Manage in production
  - Manage groups
  - Deploy to multiple Azure regions
  - Log events to Azure Event Hubs
  - Set up DR using backup/restore
  - Trace calls with the API Inspector
  - Manage user accounts
  - Manage using automation
  - Configure using Git
- Reference
  - PowerShell
  - Classic PowerShell
  - REST
  - Policies
  - Templates
- Resources
  - Pricing
  - MSDN forum
  - Stack Overflow
  - Videos
  - Service updates
  - Whitepaper: Cloud-based API Management
  - API design guidance
  - API implementation guidance
  - Consume SOAP WCF services over HTTP
  - Connecting Event Hubs to API Management
  - API Management - Plays well with other Azure services
  - Provide RBAC-like access using the REST API
  - Set up PostMan to call API Management APIs

API Management plugin for SmartBear Ready! API

Manage your first API in Azure API Management

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • Glenn Gailey • Justin Yoo • v-aljenk • Jamie Strusz • ShawnJackson • Dene Hager

Overview

This guide shows you how to quickly get started in using Azure API Management and make your first API call.

What is Azure API Management?

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

Common scenarios include:

- **Securing mobile infrastructure** by gating access with API keys, preventing DOS attacks by using throttling, or using advanced security policies like JWT token validation.
- **Enabling ISV partner ecosystems** by offering fast partner onboarding through the developer portal and building an API facade to decouple from internal implementations that are not ripe for partner consumption.
- **Running an internal API program** by offering a centralized location for the organization to communicate about the availability and latest changes to APIs, gating access based on organizational accounts, all based on a secured channel between the API gateway and the backend.

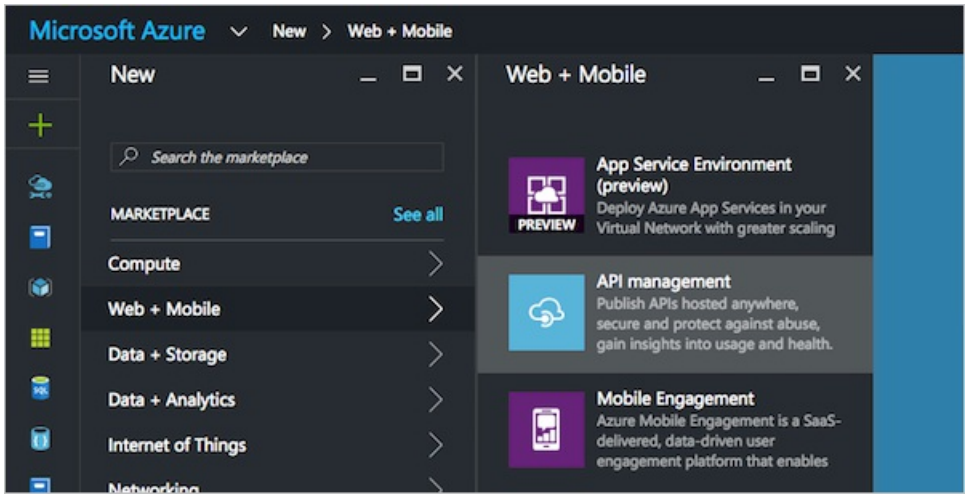
The system is made up of the following components:

- 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.
- The **publisher 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.
- 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.

Create an API Management instance

> **NOTE**
>
> To complete this tutorial, you need an Azure account. If you don't have an account, you can create a free account in just a couple of minutes. For details, see Azure Free Trial.

The first step in working with API Management is to create a service instance. Sign in to the Azure Portal and click **New**, **Web + Mobile**, **API Management**.

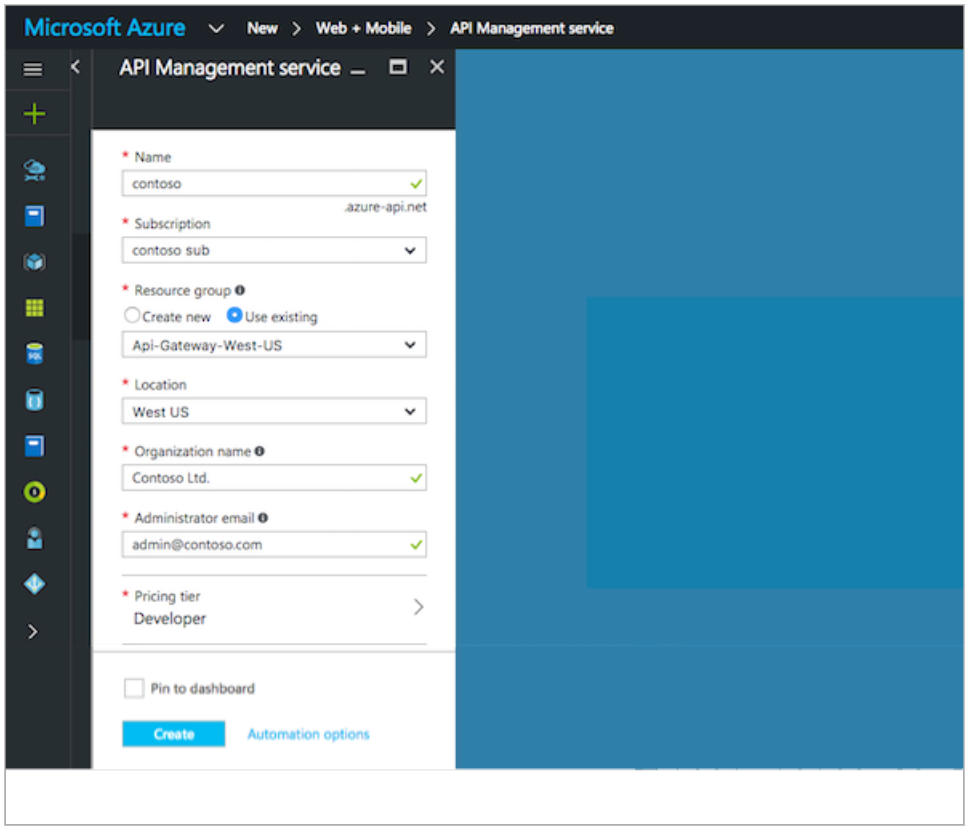


For **Name**, specify a unique sub-domain name to use for the service URL.

Choose the desired **Subscription**, **Resource group** and **Location** for your service instance.

Enter **Contoso Ltd.** for the **Organization Name**, and enter your email address in the **Administrator E-Mail** field.

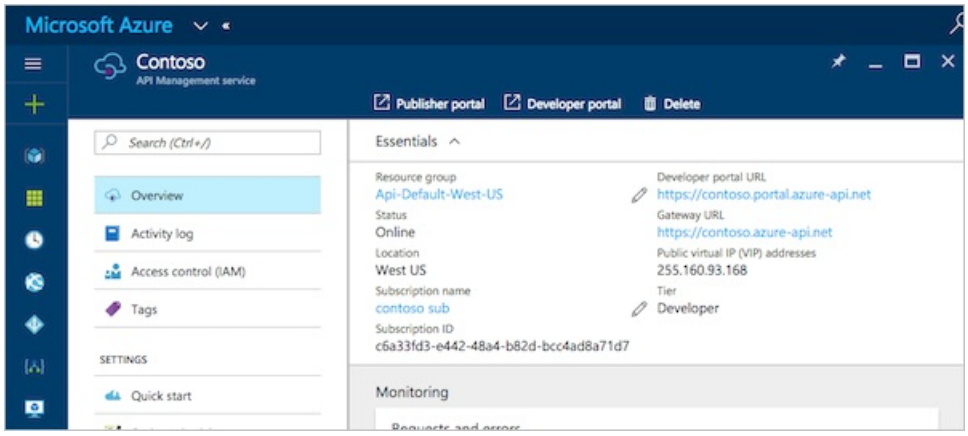
> **NOTE**
>
> This email address is used for notifications from the API Management system. For more information, see How to configure notifications and email templates in Azure API Management.



API Management service instances are available in three tiers: Developer, Standard, and Premium.

> **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. You can complete this tutorial by using any tier. For more information about API Management tiers, see API Management pricing.

Click **Create** to start provisioning your service instance.



Once the service instance is created, the next step is to create or import an API.

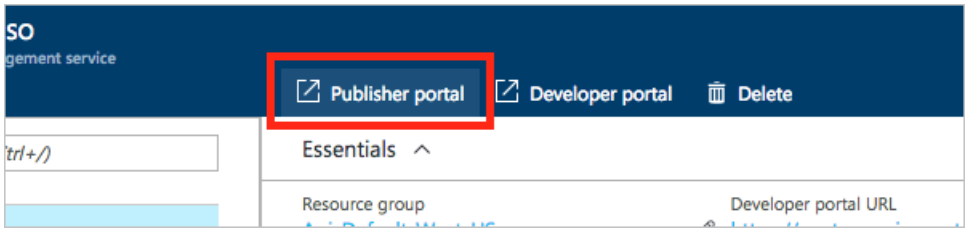
Import an API

An API consists of a set of operations that can be invoked from a client application. API operations are proxied to existing web services.

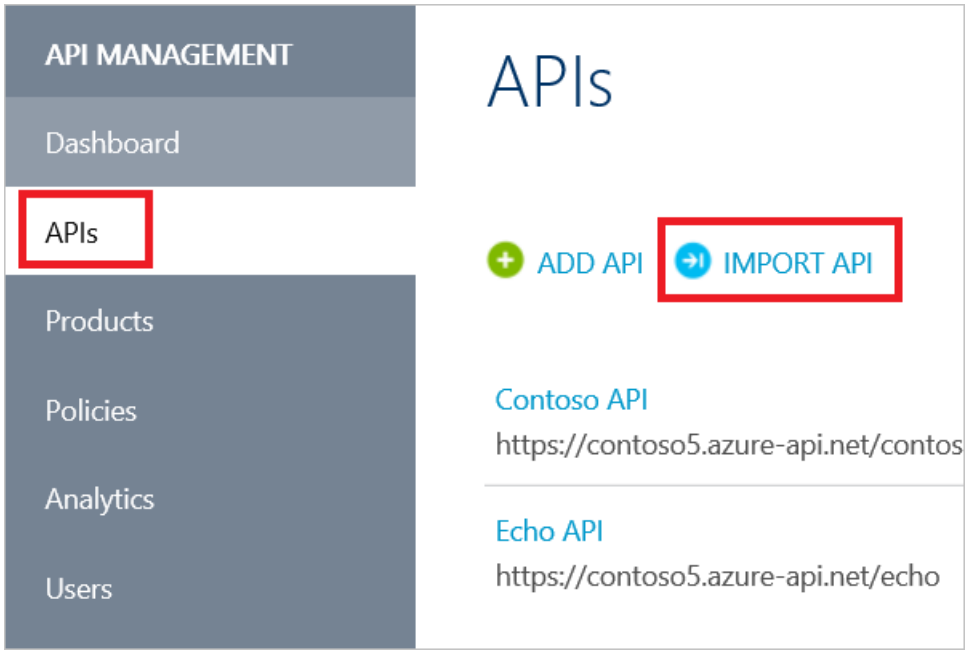
APIs can be created (and operations can be added) manually, or they can be imported. In this tutorial, we will import the API for a sample calculator web service provided by Microsoft and hosted on Azure.

> **NOTE**
>
> For guidance on creating an API and manually adding operations, see How to create APIs and How to add operations to an API.

APIs are configured from the publisher portal. To reach it, click **Publisher portal** from the service toolbar.

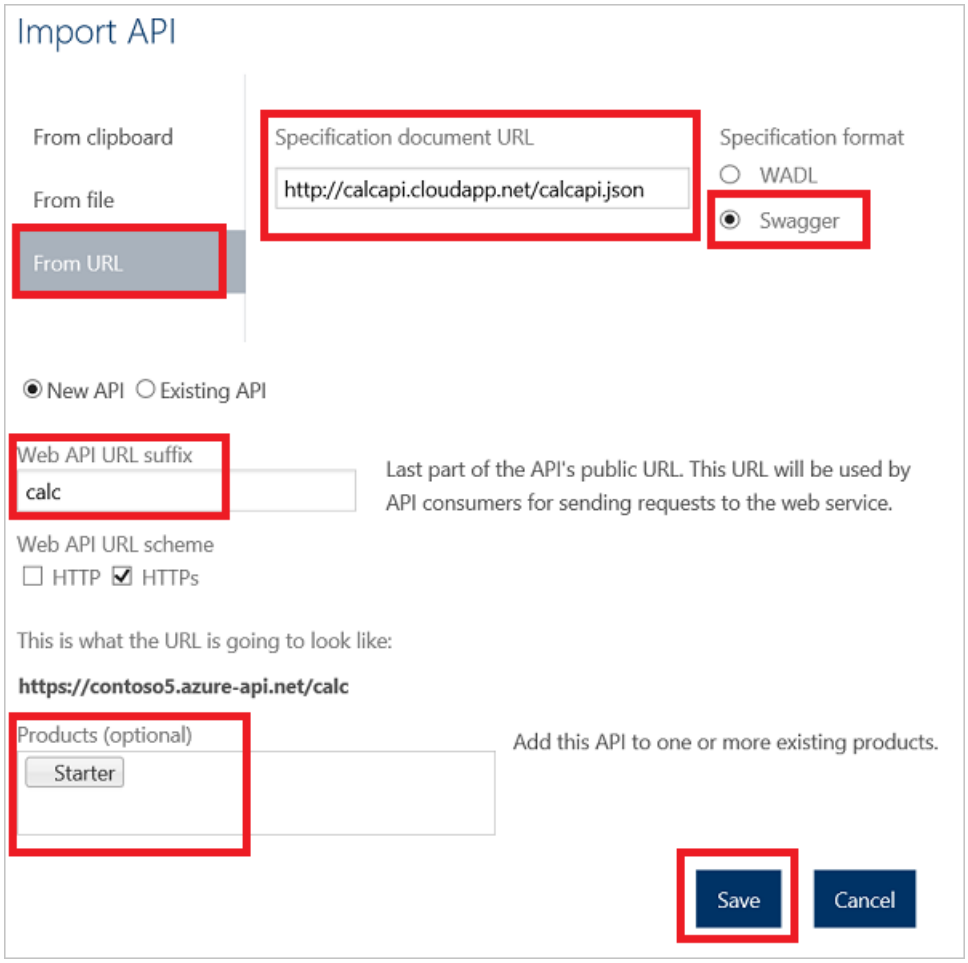


To import the calculator API, click **APIs** from the **API Management** menu on the left, and then click **Import API**.



Perform the following steps to configure the calculator API:

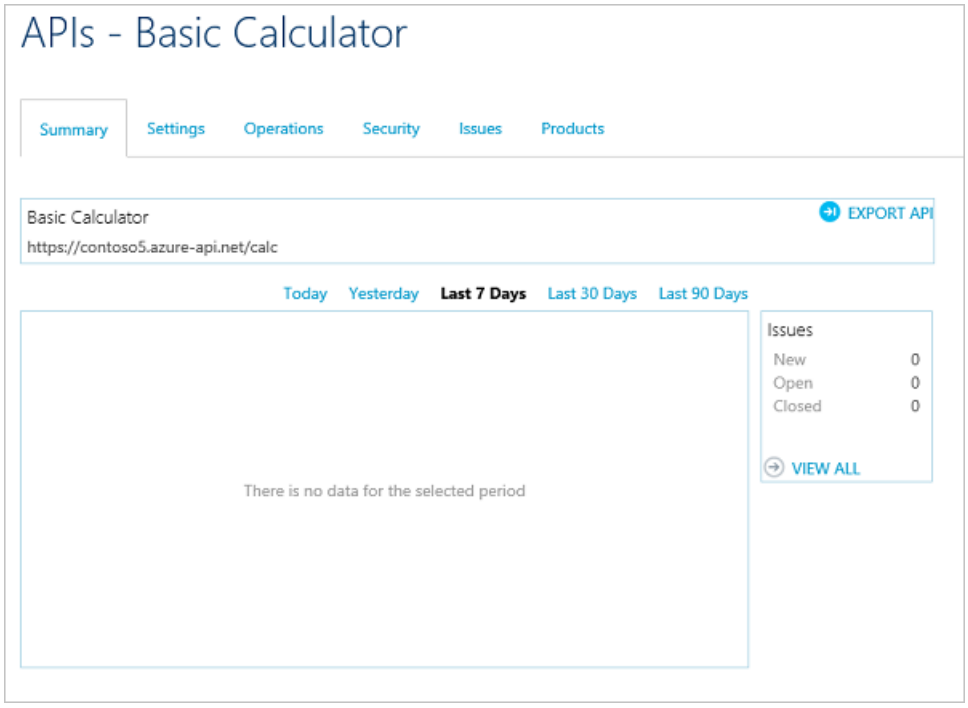
1. Click **From URL**, enter **http://calcapi.cloudapp.net/calcapi.json** into the **Specification document URL** text box, and click the **Swagger** radio button.
2. Type **calc** into the **Web API URL suffix** text box.
3. Click in the **Products (optional)** box and choose **Starter**.
4. Click **Save** to import the API.



NOTE

API Management currently supports both 1.2 and 2.0 version of Swagger document for import. Make sure that, even though Swagger 2.0 specification declares that `host`, `basePath`, and `schemes` properties are optional, your Swagger 2.0 document **MUST** contain those properties; otherwise it won't get imported.

Once the API is imported, the summary page for the API is displayed in the publisher portal.



The API section has several tabs. The **Summary** tab displays basic metrics and information about the API. The

Settings tab is used to view and edit the configuration for an API. The Operations tab is used to manage the API's operations. The **Security** tab can be used to configure gateway authentication for the backend server by using Basic authentication or mutual certificate authentication, and to configure user authorization by using OAuth 2.0. The **Issues** tab is used to view issues reported by the developers who are using your APIs. The **Products** tab is used to configure the products that contain this API.

By default, each API Management instance comes with two sample products:

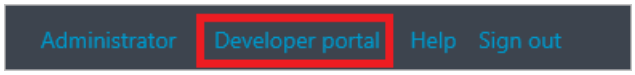
- **Starter**
- **Unlimited**

In this tutorial, the Basic Calculator API was added to the Starter product when the API was imported.

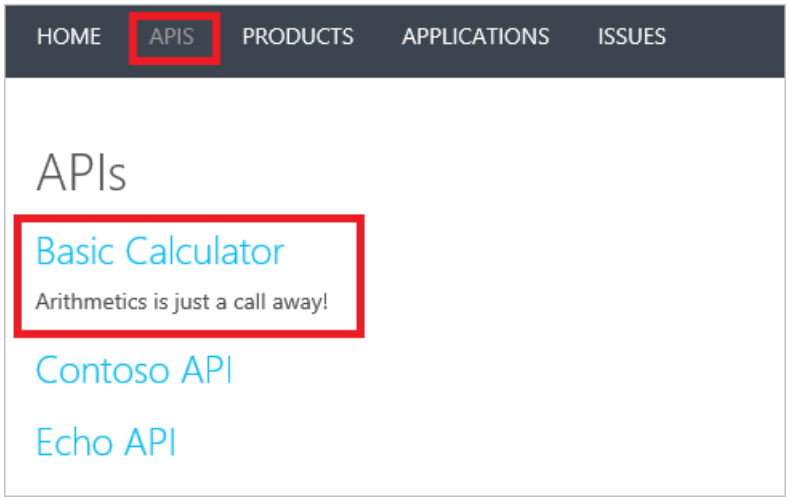
In order to make calls to an API, developers must first subscribe to a product that gives them access to it. Developers can subscribe to products in the developer portal, or administrators can subscribe developers to products in the publisher portal. You are an administrator since you created the API Management instance in the previous steps in the tutorial, so you are already subscribed to every product by default.

Call an operation from the developer portal

Operations can be called directly from the developer portal, which provides a convenient way to view and test the operations of an API. In this tutorial step, you will call the Basic Calculator API's **Add two integers** operation. Click **Developer portal** from the menu at the top right of the publisher portal.

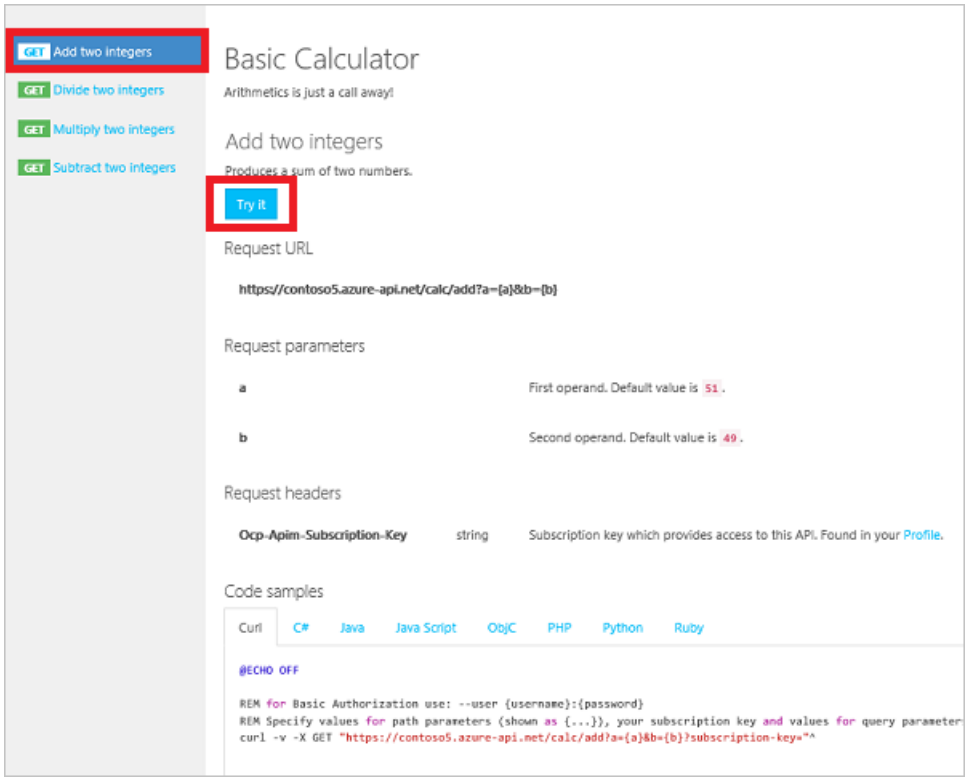


Click **APIs** from the top menu, and then click **Basic Calculator** to see the available operations.

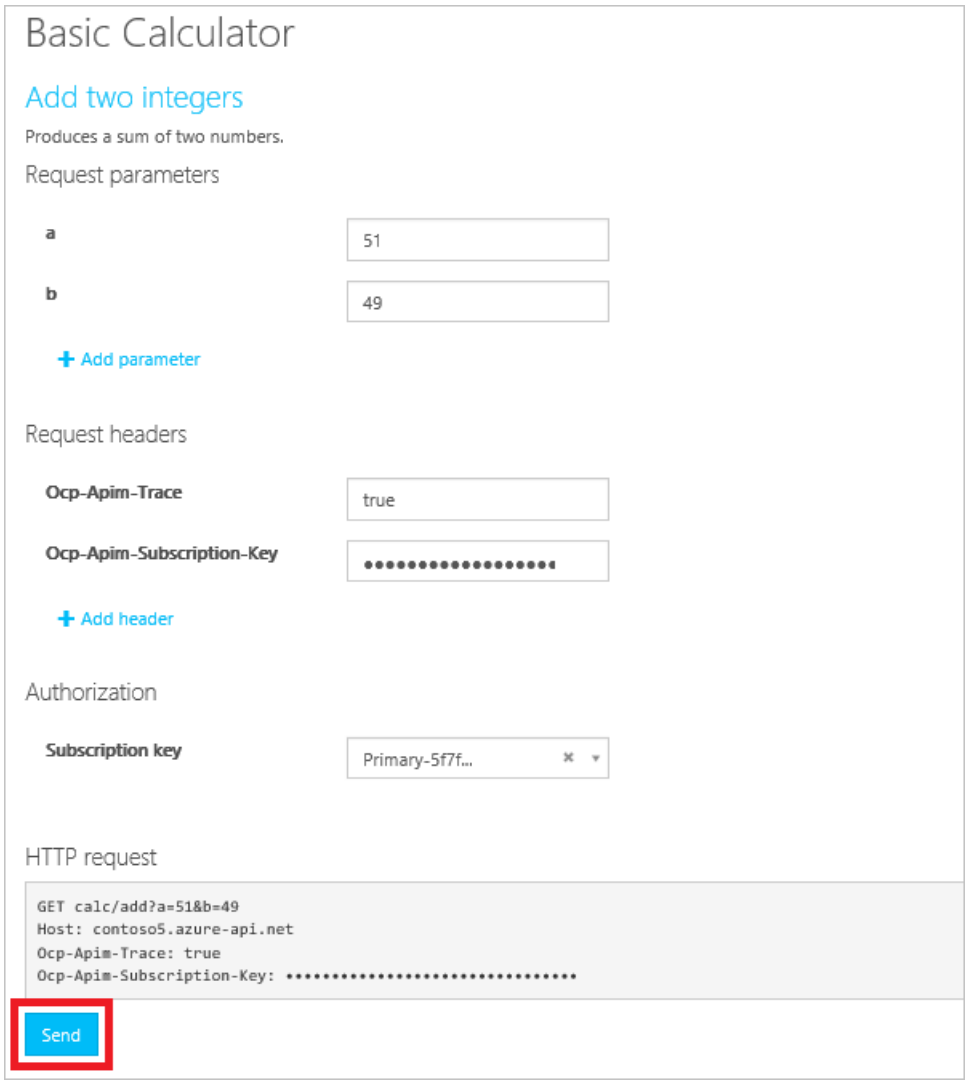


Note the sample descriptions and parameters that were imported along with the API and operations, providing documentation for the developers that will use this operation. These descriptions can also be added when operations are added manually.

To call the **Add two integers** operation, click **Try it**.



You can enter some values for the parameters or keep the defaults, and then click **Send**.



After an operation is invoked, the developer portal displays the **Response status**, the **Response headers**, and any **Response content**.

HTTP request

```
GET calc/add?a=51&b=49
Host: contoso5.azure-api.net
Ocp-Apim-Trace: true
Ocp-Apim-Subscription-Key: ••••••••••••••••••••••••••••••••
```

Send

Response status

200 OK

Response latency

1695 ms

Response headers

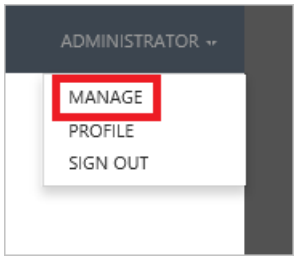
```
1.  Pragma: no-cache
2.  Ocp-Apim-Trace-Location: https://apimgmtf54evxijiwxvklmlc.blob.core.windows.net/apiinspec
    j1cdCDJzQ2-2?sv=2013-08-15&sr=b&sig=0VKCTQmOluzOMDlLnM9TnKGbVd9zRknma2u8oTU28xc%3D&se=201
    &traceId=ebf9e452bd68471286285f35126c9b89
3.  Cache-Control: no-cache
4.  Date: Fri, 22 May 2015 17:55:43 GMT
5.  Server: Microsoft-HTTPAPI/2.0
6.  X-AspNet-Version: 4.0.30319
7.  X-Powered-By: ASP.NET
8.  Content-Length: 124
9.  Content-Type: application/xml; charset=utf-8
10. Expires: -1
```

Response content

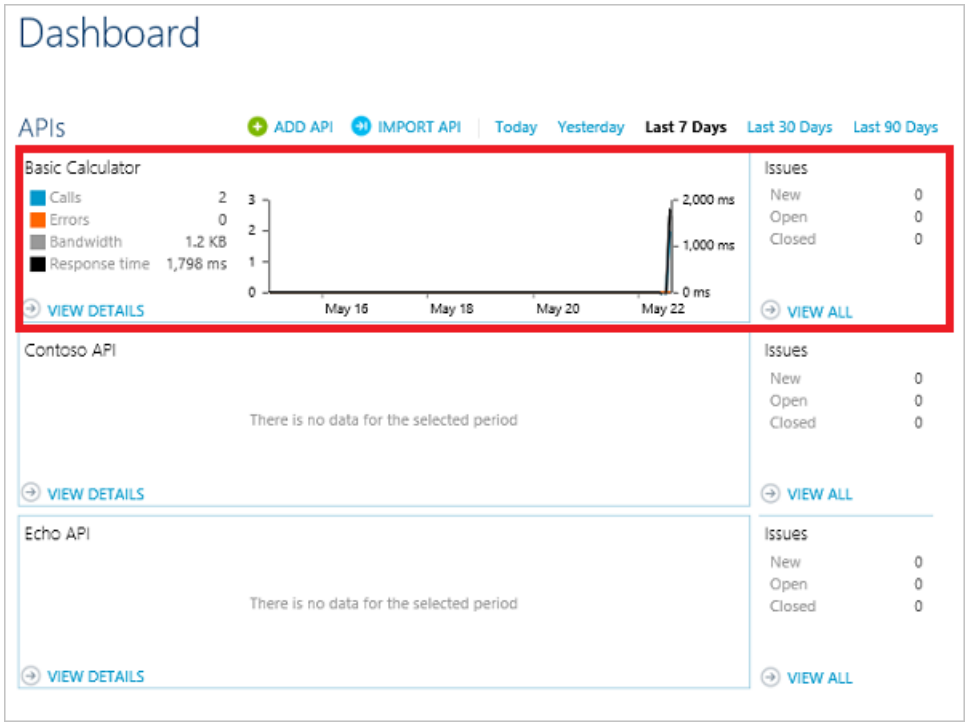
```
<result>
        <value>100</value>
        <broughtToYouBy>Azure API Management - http://azure.microsoft.com/apim/ </broughtToYouBy>
</result>
```

View analytics

To view analytics for Basic Calculator, switch back to the publisher portal by selecting **Manage** from the menu at the top right of the developer portal.



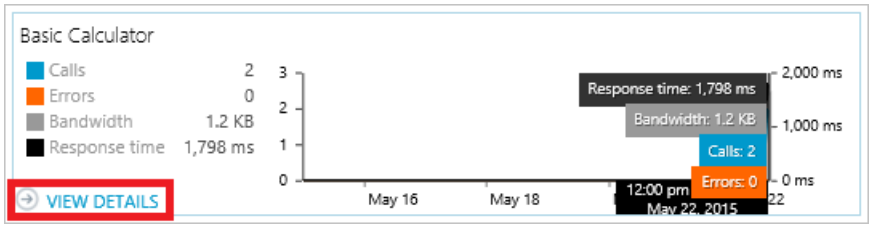
The default view for the publisher portal is the **Dashboard**, which provides an overview of your API Management instance.

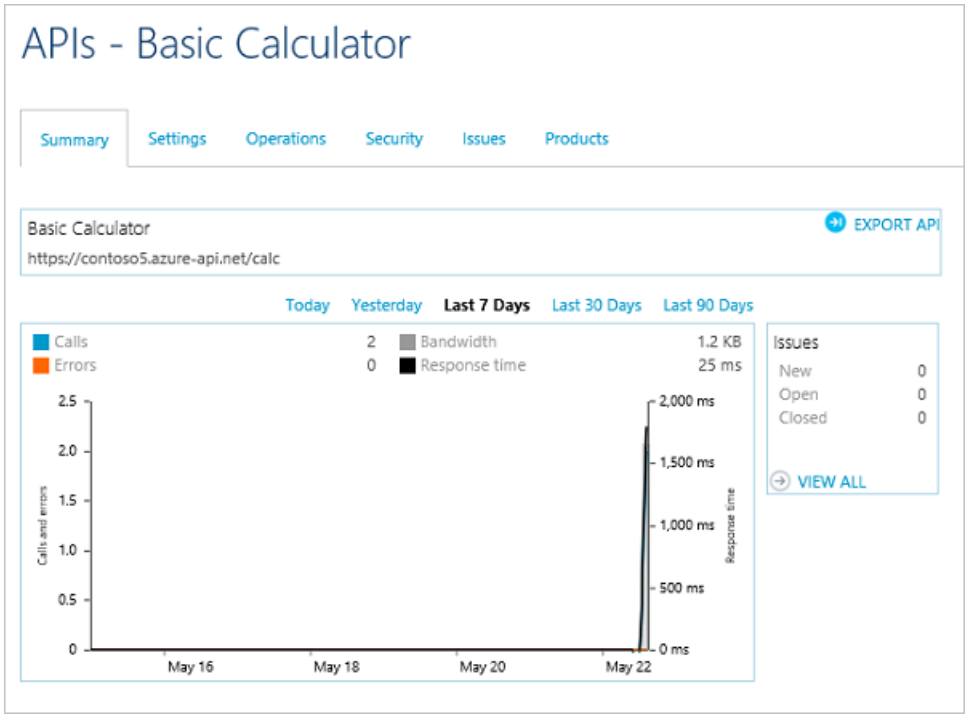


Hover the mouse over the chart for **Basic Calculator** to see the specific metrics for the usage of the API for a given time period.

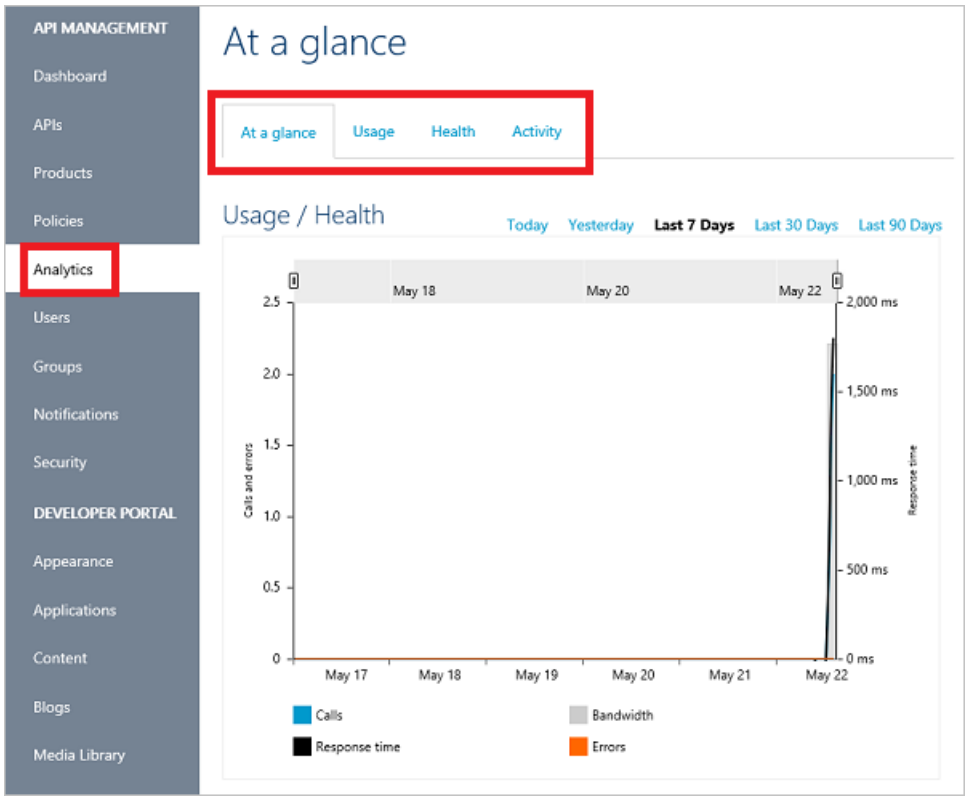
> **NOTE**
>
> If you don't see any lines on your chart, switch back to the developer portal and make some calls into the API, wait a few moments, and then come back to the dashboard.

Click **View Details** to view the summary page for the API, including a larger version of the displayed metrics.





For detailed metrics and reports, click **Analytics** from the **API Management** menu on the left.



The **Analytics** section has the following four tabs:

- **At a glance** provides overall usage and health metrics, as well as the top developers, top products, top APIs, and top operations.
- **Usage** provides an in-depth look at API calls and bandwidth, including a geographical representation.
- **Health** focuses on status codes, cache success rates, response times, and API and service response times.
- **Activity** provides reports that drill down on the specific activity by developer, product, API, and operation.

Next steps

- Learn how to Protect your API with rate limits.

Protect your API with rate limits using Azure API Management

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • Matt Barry

This guide shows you how easy it is to add protection for your backend API by configuring rate limit and quota policies with Azure API Management.

In this tutorial, you will create a "Free Trial" API product that allows developers to make up to 10 calls per minute and up to a maximum of 200 calls per week to your API using the Limit call rate per subscription and Set usage quota per subscription policies. You will then publish the API and test the rate limit policy.

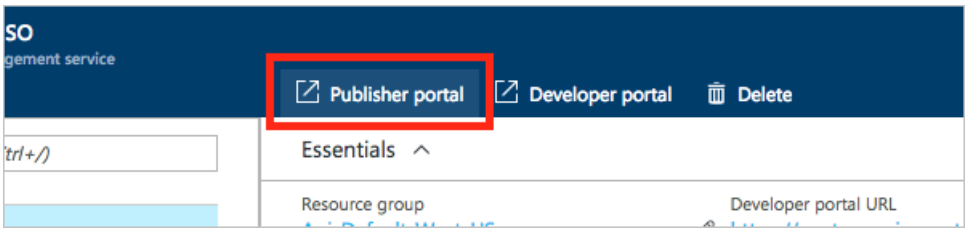
For more advanced throttling scenarios using the rate-limit-by-key and quota-by-key policies, see Advanced request throttling with Azure API Management.

To create a product

In this step, you will create a Free Trial product that does not require subscription approval.

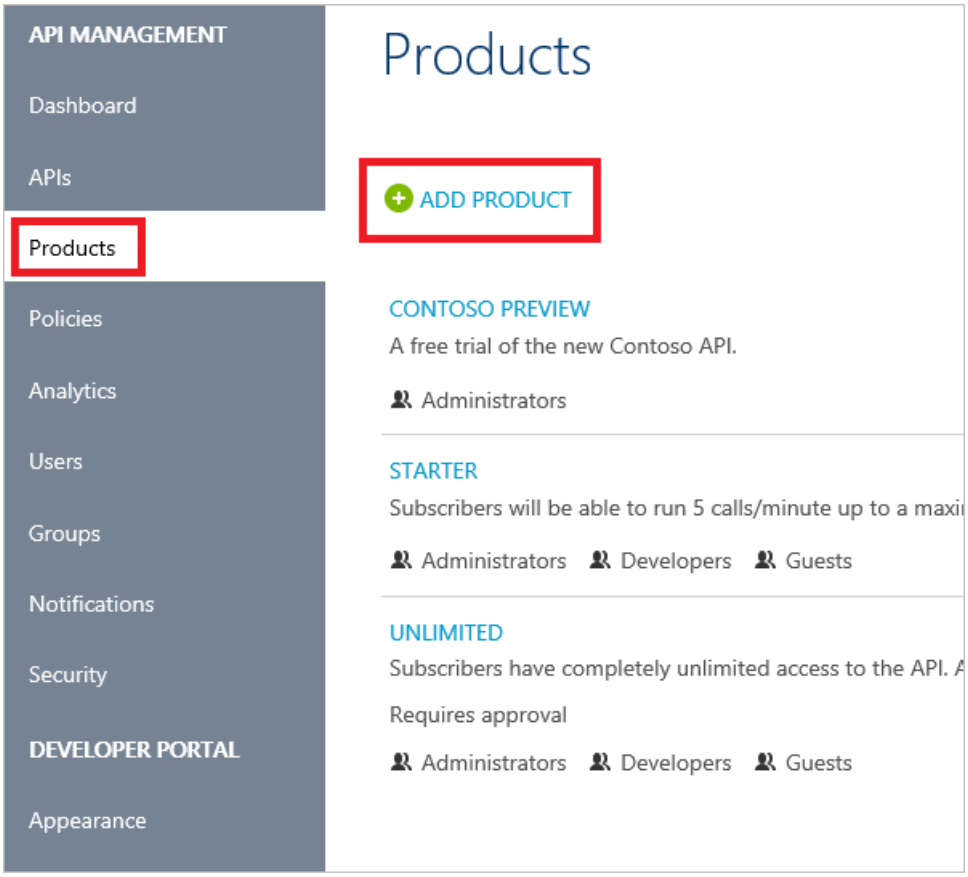
> **NOTE**
> If you already have a product configured and want to use it for this tutorial, you can jump ahead to Configure call rate limit and quota policies and follow the tutorial from there using your product in place of the Free Trial product.

To get started, click **Publisher portal** in the Azure Portal for your API Management service.

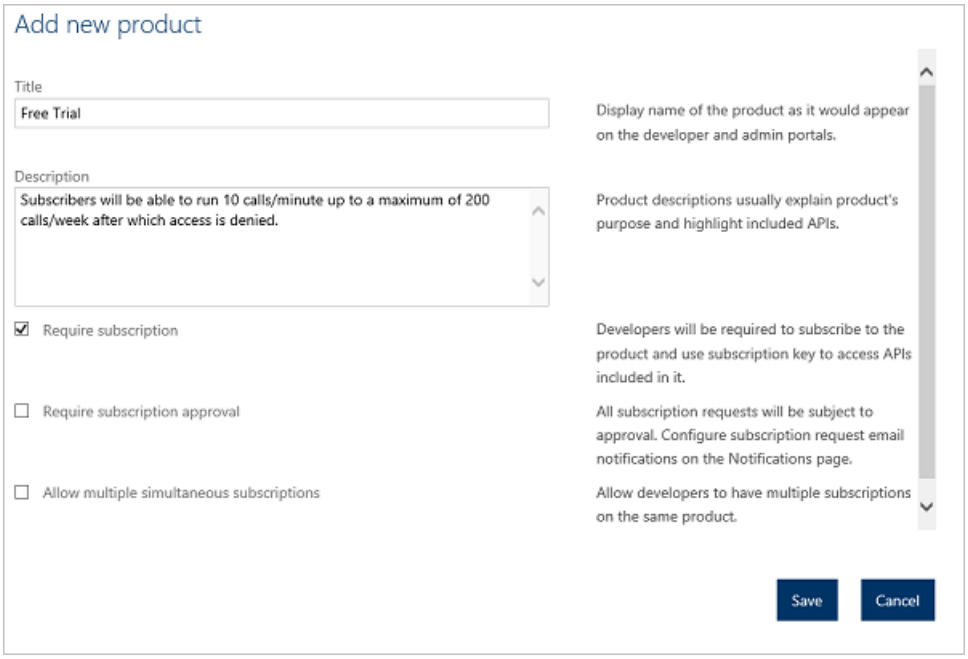


> If you have not yet created an API Management service instance, see Create an API Management service instance in the Manage your first API in Azure API Management tutorial.

Click **Products** in the **API Management** menu on the left to display the **Products** page.



Click **Add product** to display the **Add new product** dialog box.



In the **Title** box, type **Free Trial**.

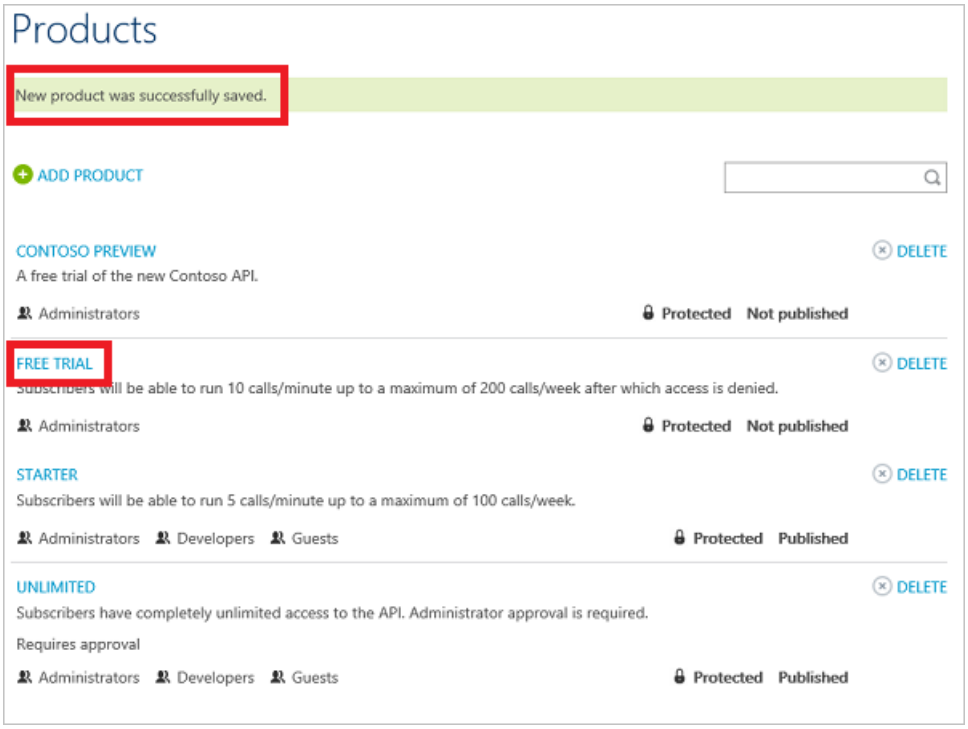
In the **Description** box, type the following text: **Subscribers will be able to run 10 calls/minute up to a maximum of 200 calls/week after which access is denied.**

Products in API Management can be protected or open. Protected products must be subscribed to before they can be used. Open products can be used without a subscription. Ensure that **Require subscription** is selected to create a protected product that requires a subscription. This is the default setting.

If you want an administrator to review and accept or reject subscription attempts to this product, select **Require subscription approval**. If the check box is not selected, subscription attempts will be auto-approved. In this example, subscriptions are automatically approved, so do not select the box.

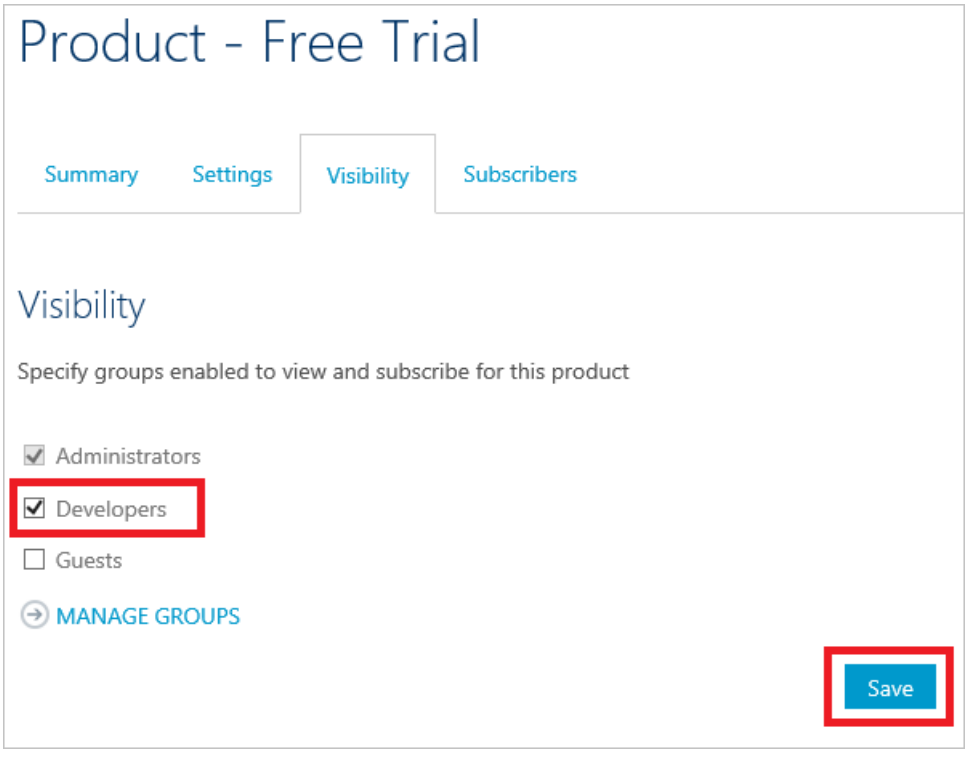
To allow developer accounts to subscribe multiple times to the new product, select the **Allow multiple simultaneous subscriptions** check box. This tutorial does not utilize multiple simultaneous subscriptions, so leave it unchecked.

After all values are entered, click **Save** to create the product.



By default, new products are visible to users in the **Administrators** group. We are going to add the **Developers** group. Click **Free Trial**, and then click the **Visibility** tab.

> In API Management, groups are used to manage the visibility of products to developers. Products grant visibility to groups, and developers can view and subscribe to the products that are visible to the groups in which they belong. For more information, see How to create and use groups in Azure API Management.



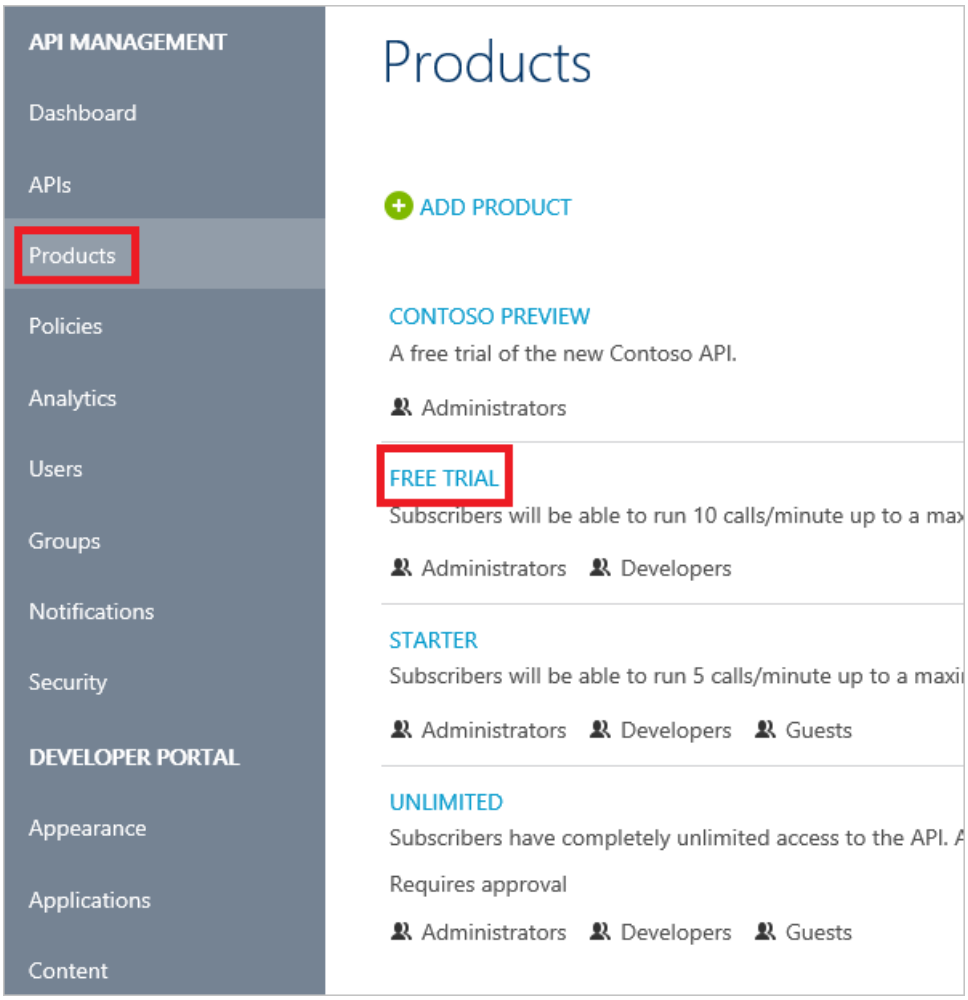
Select the **Developers** check box, and then click **Save**.

To add an API to the product

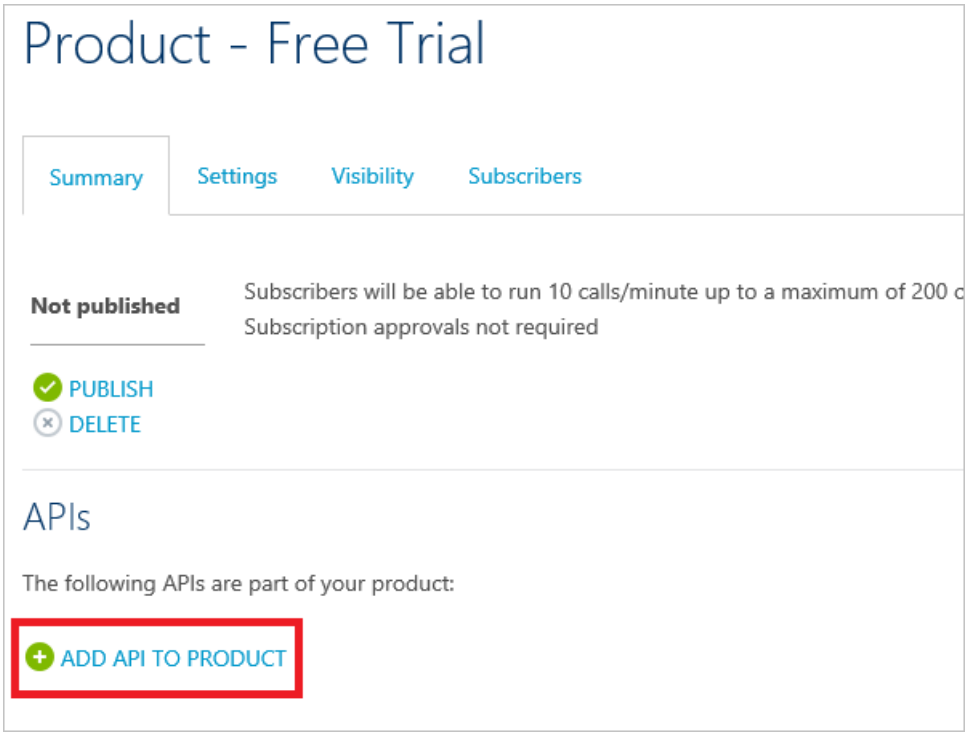
In this step of the tutorial, we will add the Echo API to the new Free Trial product.

> Each API Management service instance comes pre-configured with an Echo API that can be used to experiment with and learn about API Management. For more information, see Manage your first API in Azure API Management.

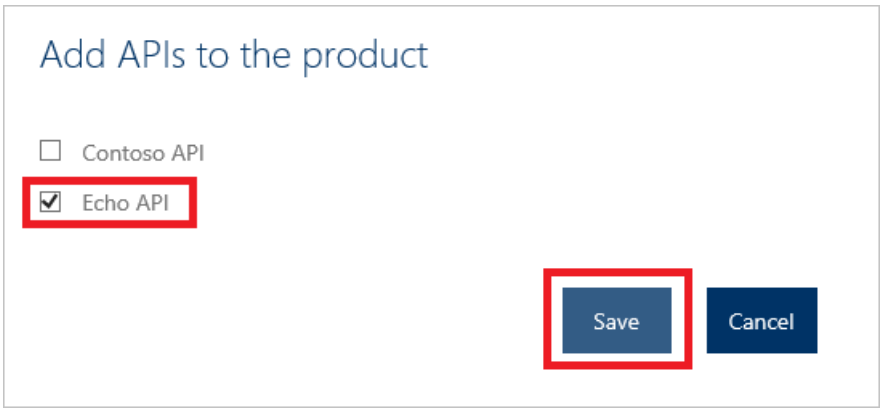
Click **Products** from the **API Management** menu on the left, and then click **Free Trial** to configure the product.



Click **Add API to product**.

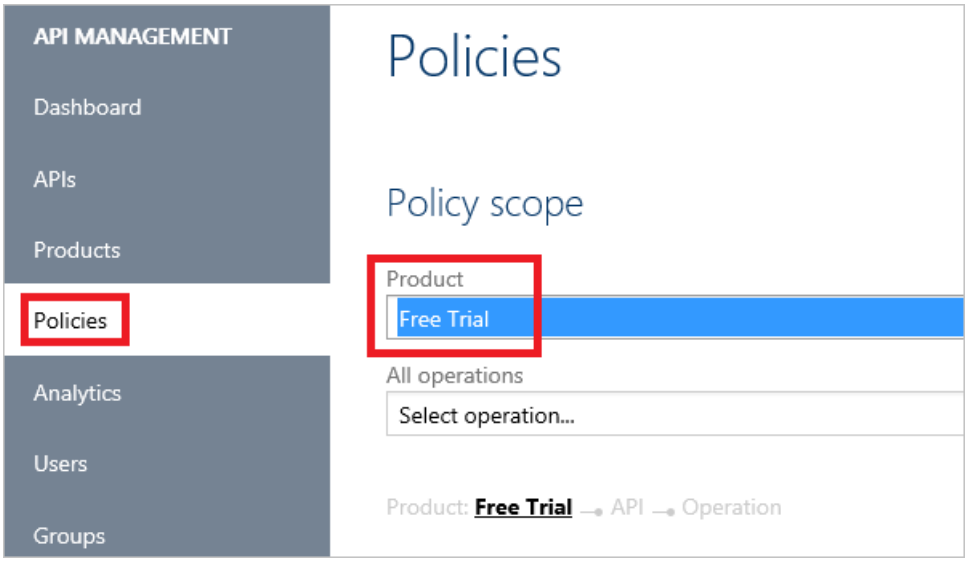


Select **Echo API**, and then click **Save**.

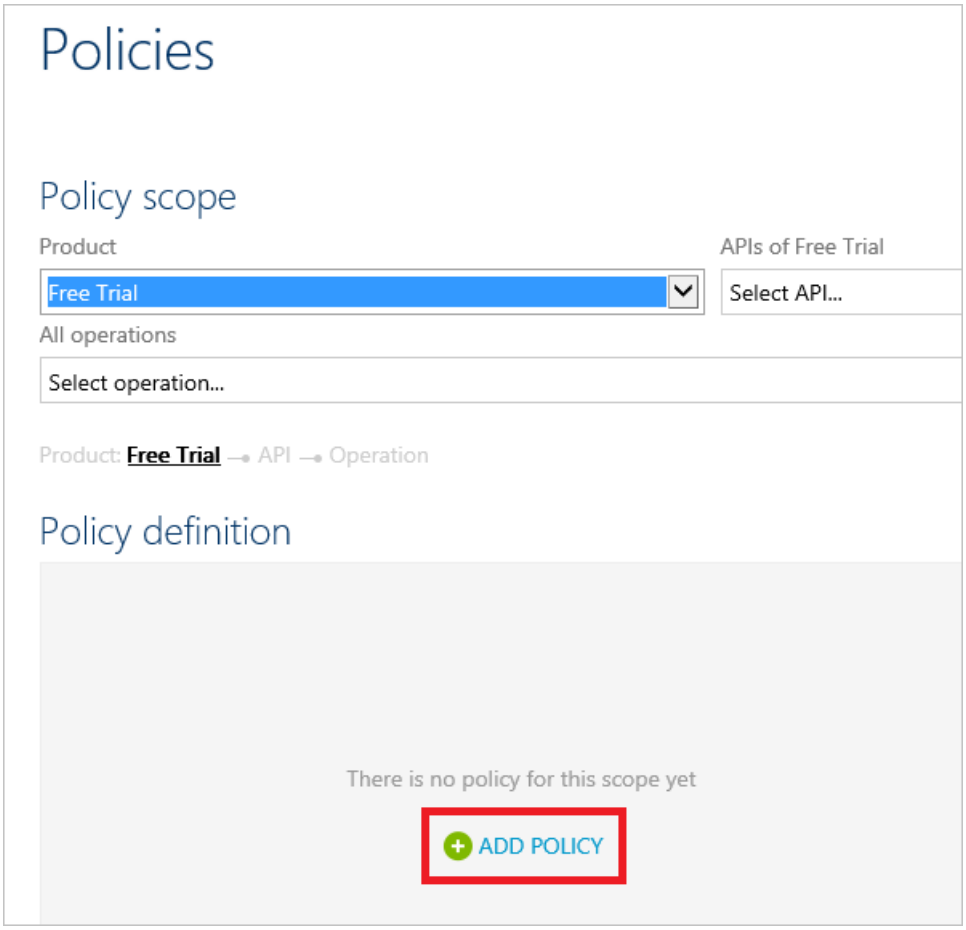


To configure call rate limit and quota policies

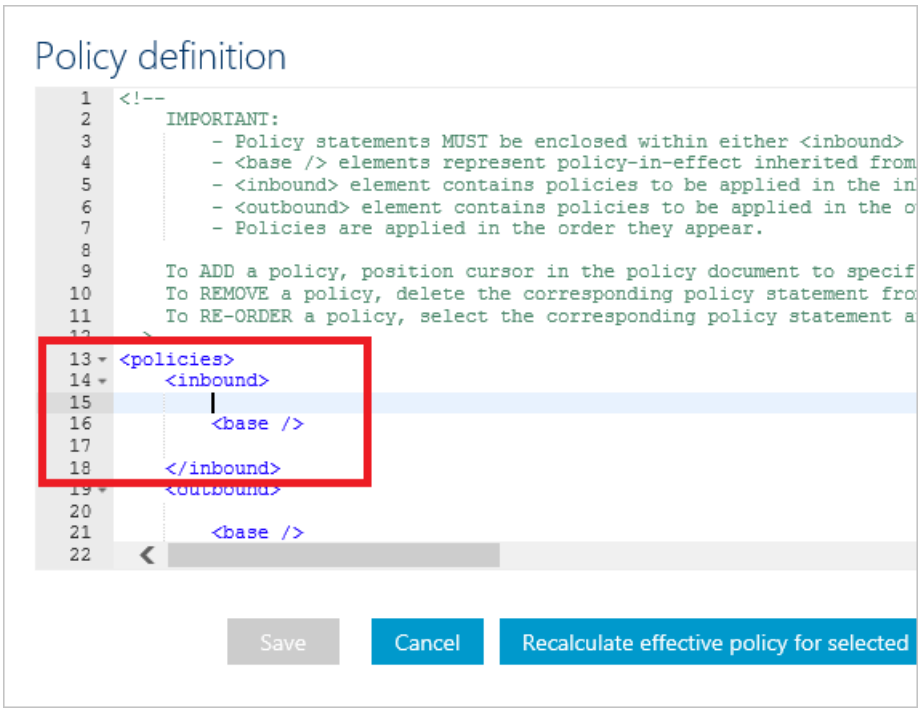
Rate limits and quotas are configured in the policy editor. Click **Policies** under the **API Management** menu on the left. In the **Product** list, click **Free Trial**.



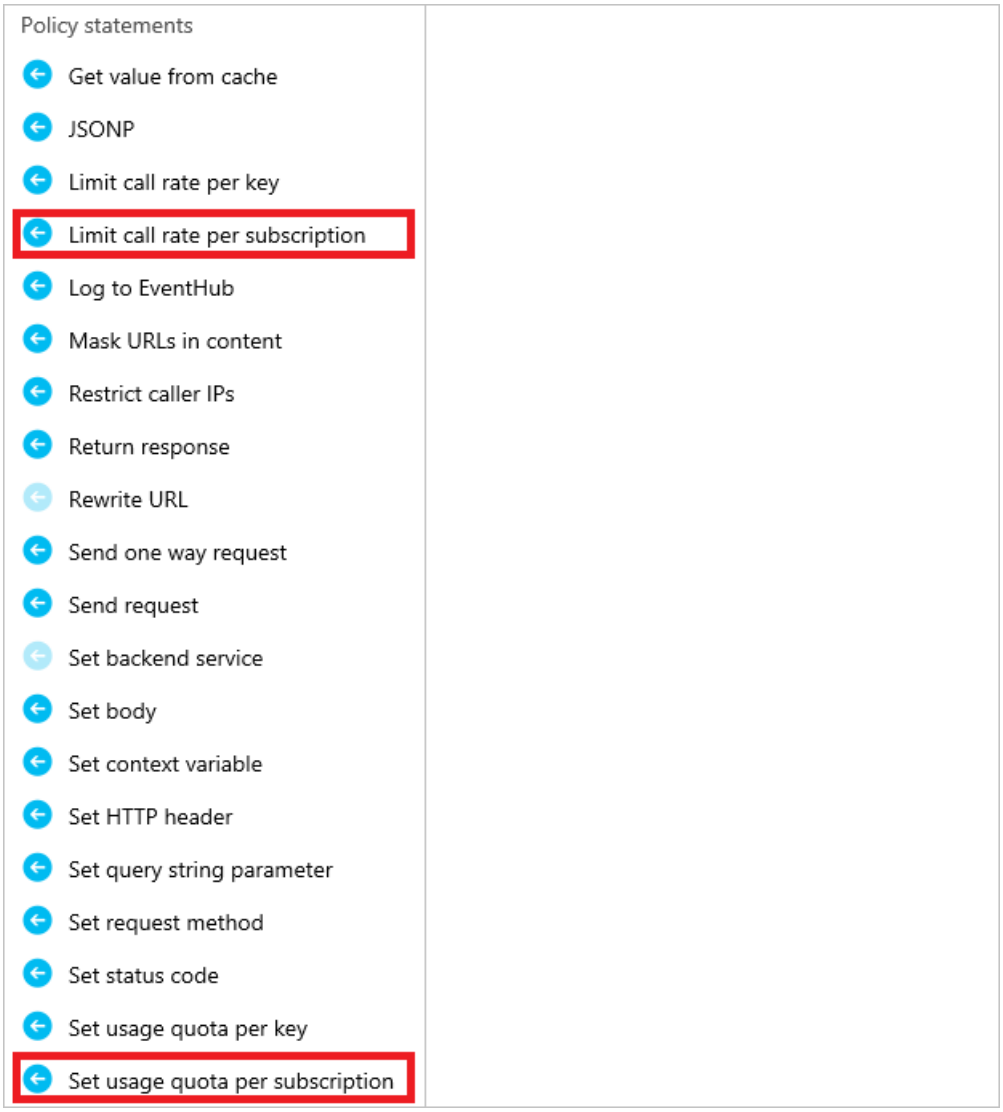
Click **Add Policy** to import the policy template and begin creating the rate limit and quota policies.



To insert policies, position the cursor into either the **inbound** or **outbound** section of the policy template. Rate limit and quota policies are inbound policies, so position the cursor in the inbound element.



The two policies we are adding in this tutorial are the Limit call rate per subscription and Set usage quota per subscription policies.



After the cursor is positioned in the **inbound** policy element, click the arrow beside **Limit call rate per subscription** to insert its policy template.

```
<rate-limit calls="number" renewal-period="seconds">
<api name="name" calls="number">
<operation name="name" calls="number" />
</api>
</rate-limit>
```

Limit call rate per subscription can be used at the product level and can also be used at the API and individual operation name levels. In this tutorial, only product-level policies are used, so delete the **api** and **operation** elements from the **rate-limit** element, so only the outer **rate-limit** element remains, as shown in the following example.

```
<rate-limit calls="number" renewal-period="seconds">
</rate-limit>
```

In the Free Trial product, the maximum allowable call rate is 10 calls per minute, so type **10** as the value for the **calls** attribute, and **60** for the **renewal-period** attribute.

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

To configure the **Set usage quota per subscription** policy, position your cursor immediately below the newly added **rate-limit** element within the **inbound** element, and then click the arrow to the left of **Set usage quota**

per subscription.

```
<quota calls="number" bandwidth="kilobytes" renewal-period="seconds">
<api name="name" calls="number" bandwidth="kilobytes">
<operation name="name" calls="number" bandwidth="kilobytes" />
</api>
</quota>
```

Because this policy is also intended to be at the product level, delete the **api** and **operation** name elements, as shown in the following example.

```
<quota calls="number" bandwidth="kilobytes" renewal-period="seconds">
</quota>
```

Quotas can be based on the number of calls per interval, bandwidth, or both. In this tutorial, we are not throttling based on bandwidth, so delete the **bandwidth** attribute.

```
<quota calls="number" renewal-period="seconds">
</quota>
```

In the Free Trial product, the quota is 200 calls per week. Specify **200** as the value for the **calls** attribute, and then specify **604800** as the value for the **renewal-period** attribute.

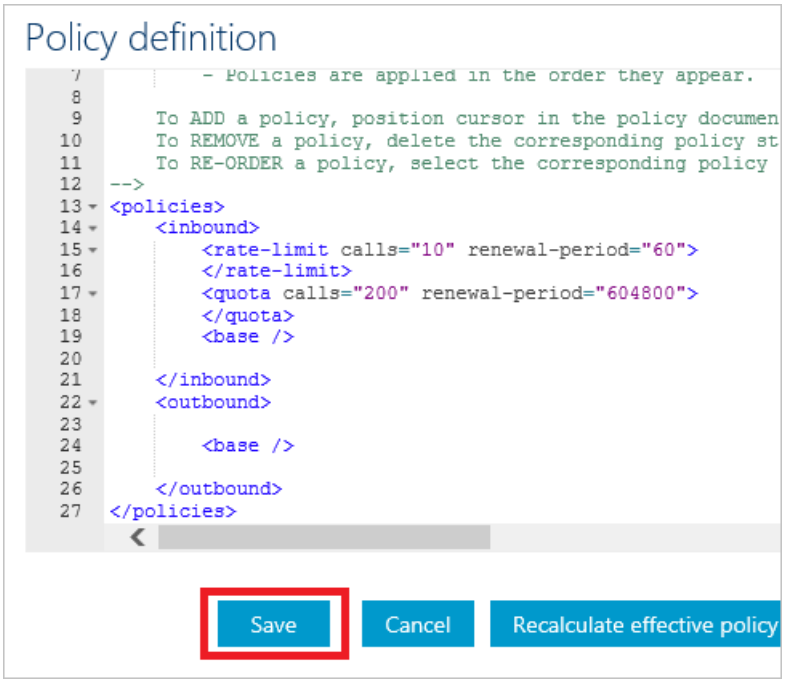
```
<quota calls="200" renewal-period="604800">
</quota>
```

> Policy intervals are specified in seconds. To calculate the interval for a week, you can multiply the number of days (7) by the number of hours in a day (24) by the number of minutes in an hour (60) by the number of seconds in a minute (60): 7 * 24 * 60 * 60 = 604800.

When you have finished configuring the policy, it should match the following example.

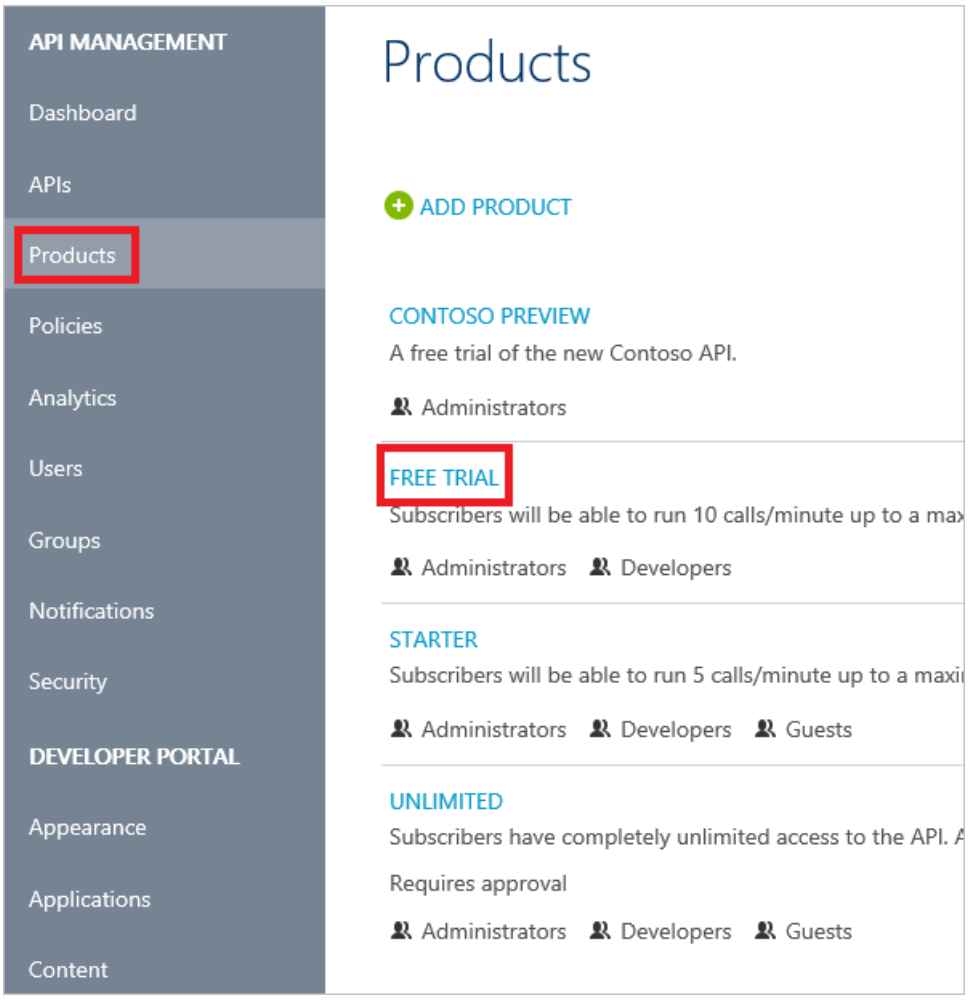
```
<policies>
    <inbound>
        <rate-limit calls="10" renewal-period="60">
        </rate-limit>
        <quota calls="200" renewal-period="604800">
        </quota>
        <base />

</inbound>
<outbound>

    <base />

    </outbound>
</policies>
```

After the desired policies are configured, click **Save**.

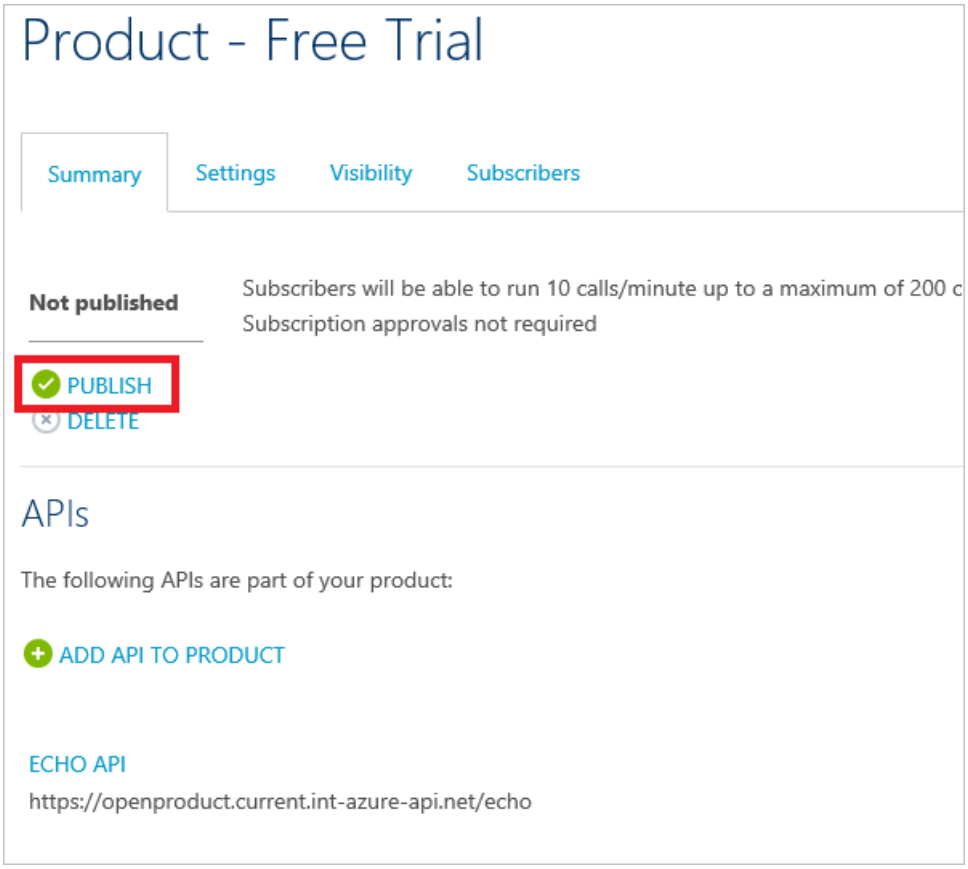


To publish the product

Now that the the APIs are added and the policies are configured, the product must be published so that it can be used by developers. Click **Products** from the **API Management** menu on the left, and then click **Free Trial** to configure the product.



Click **Publish**, and then click **Yes, publish it** to confirm.

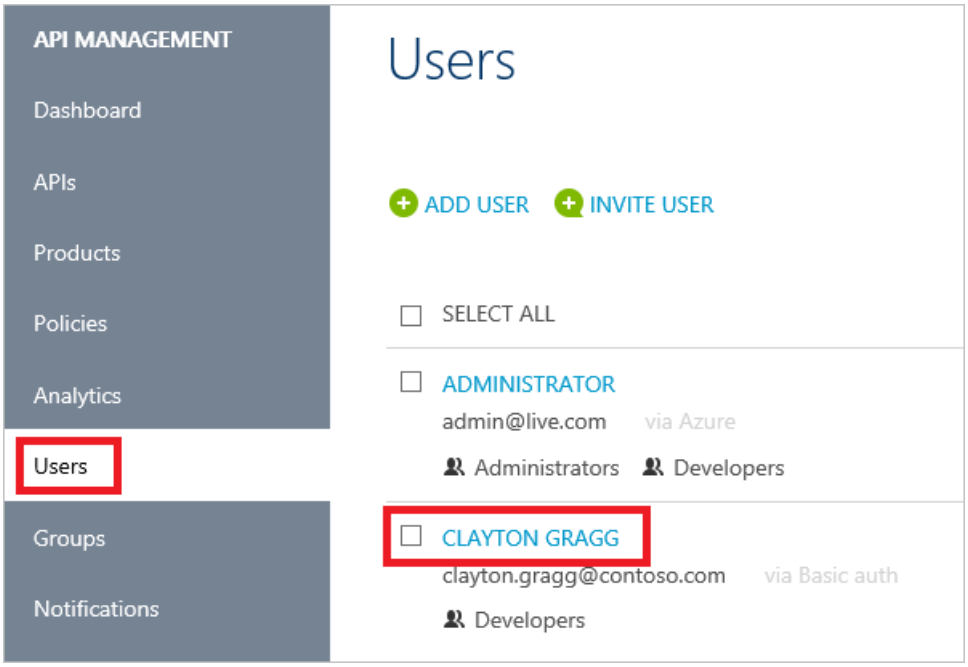


To subscribe a developer account to the product

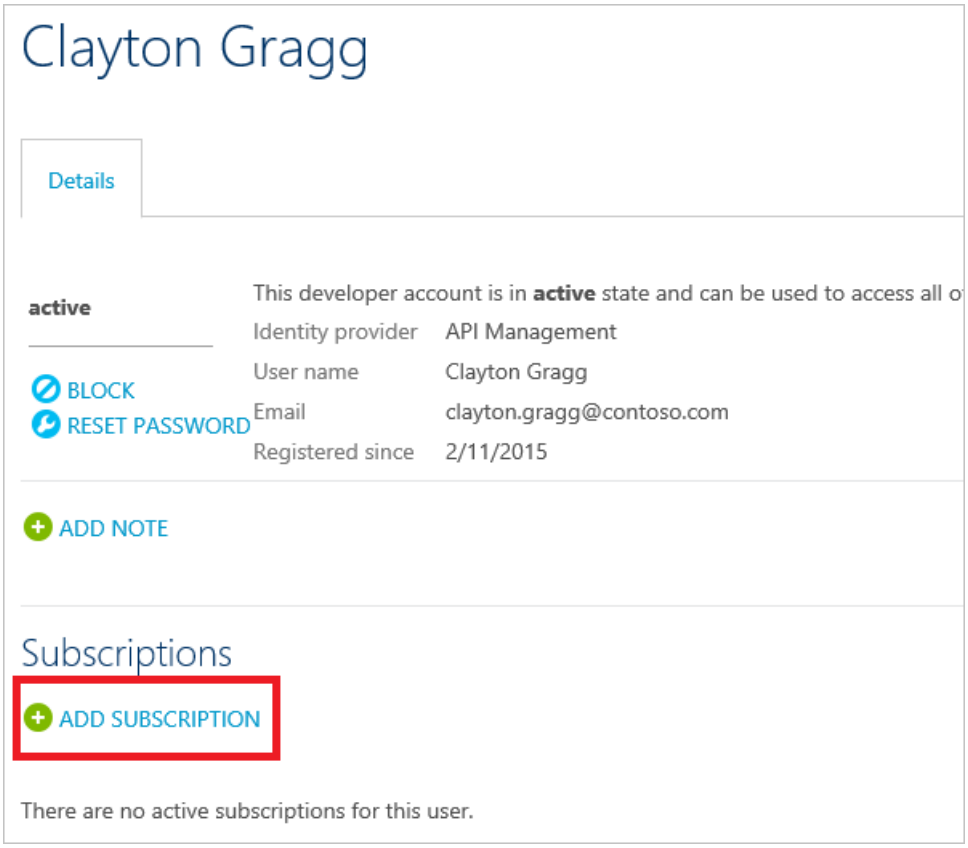
Now that the product is published, it is available to be subscribed to and used by developers.

> Administrators of an API Management instance are automatically subscribed to every product. In this tutorial step, we will subscribe one of the non-administrator developer accounts to the Free Trial product. If your developer account is part of the Administrators role, then you can follow along with this step, even though you are already subscribed.

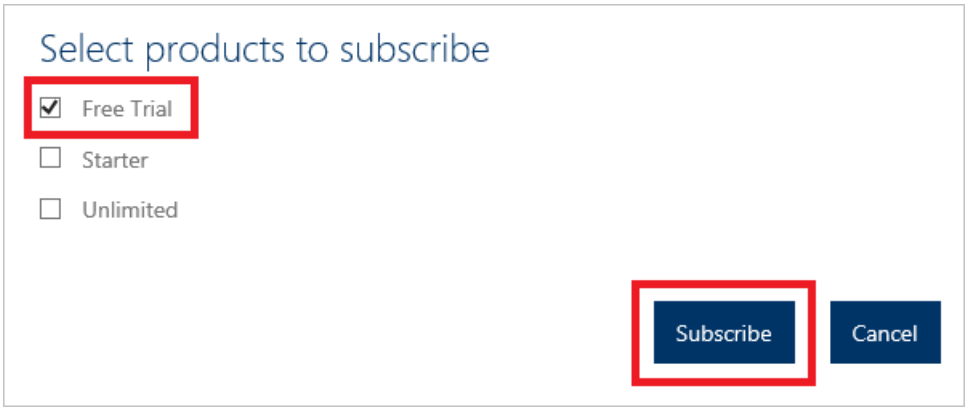
Click **Users** on the **API Management** menu on the left, and then click the name of your developer account. In this example, we are using the **Clayton Gragg** developer account.



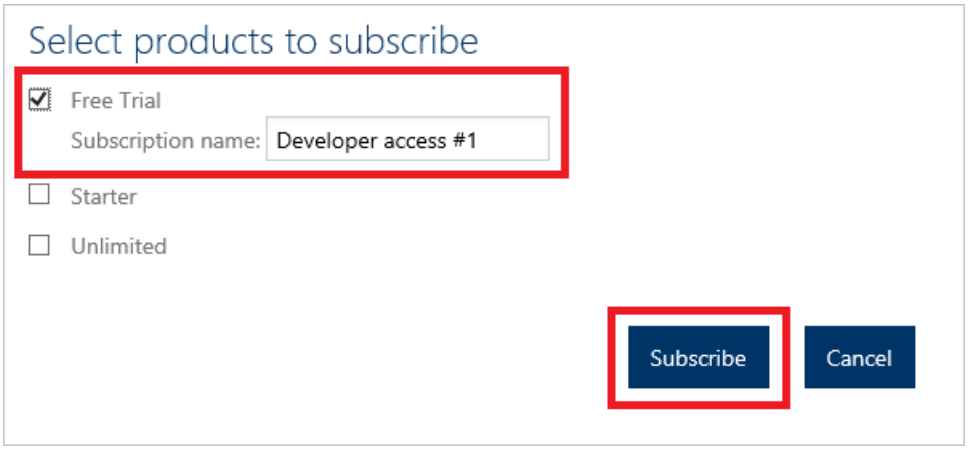
Click **Add Subscription**.



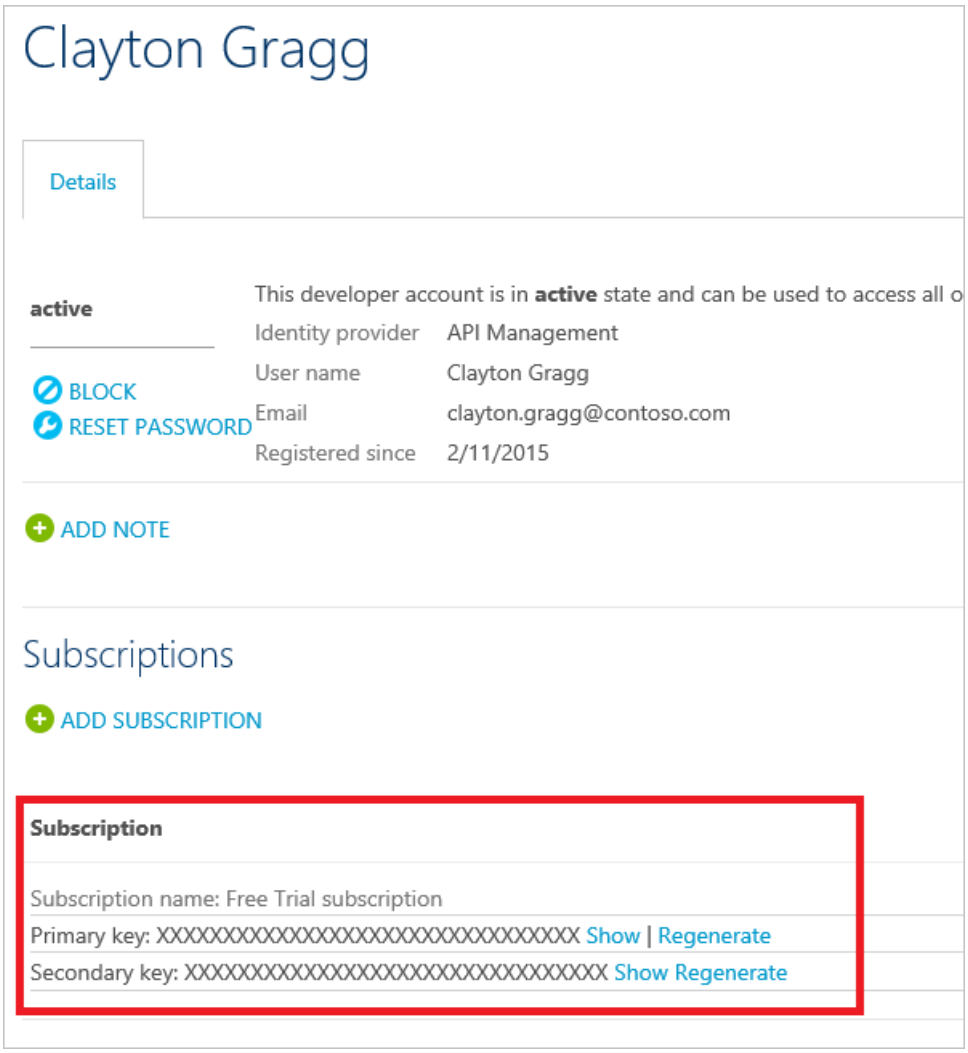
Select **Free Trial**, and then click **Subscribe**.



> **NOTE**
> In this tutorial, multiple simultaneous subscriptions are not enabled for the Free Trial product. If they were, you would be prompted to name the subscription, as shown in the following example.



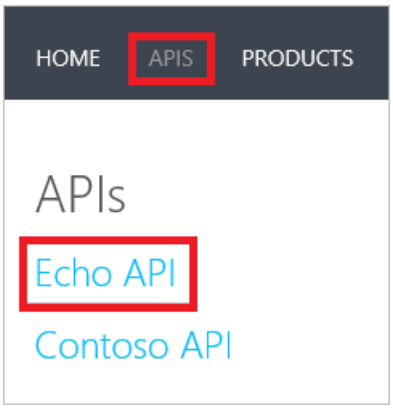
After clicking **Subscribe**, the product appears in the **Subscription** list for the user.



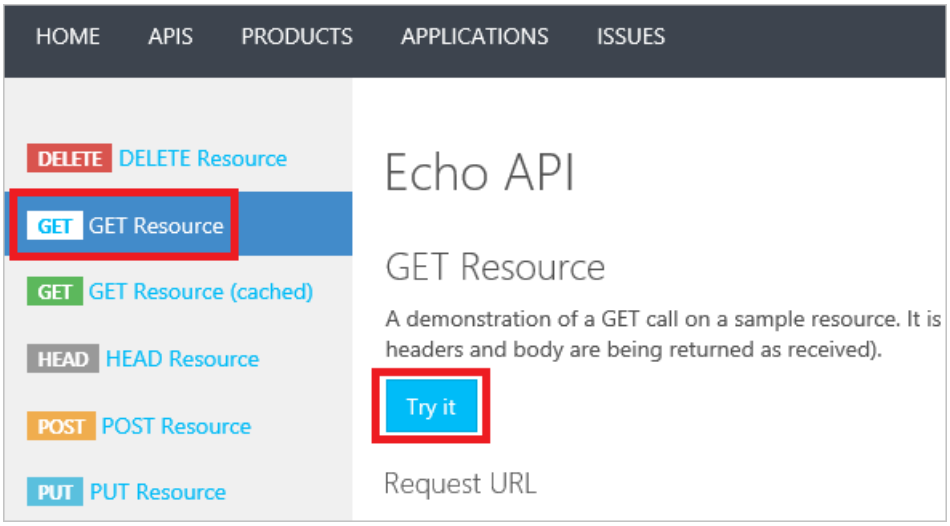
To call an operation and test the rate limit

Now that the Free Trial product is configured and published, we can call some operations and test the rate limit policy. Switch to the developer portal by clicking **Developer portal** in the upper-right menu.

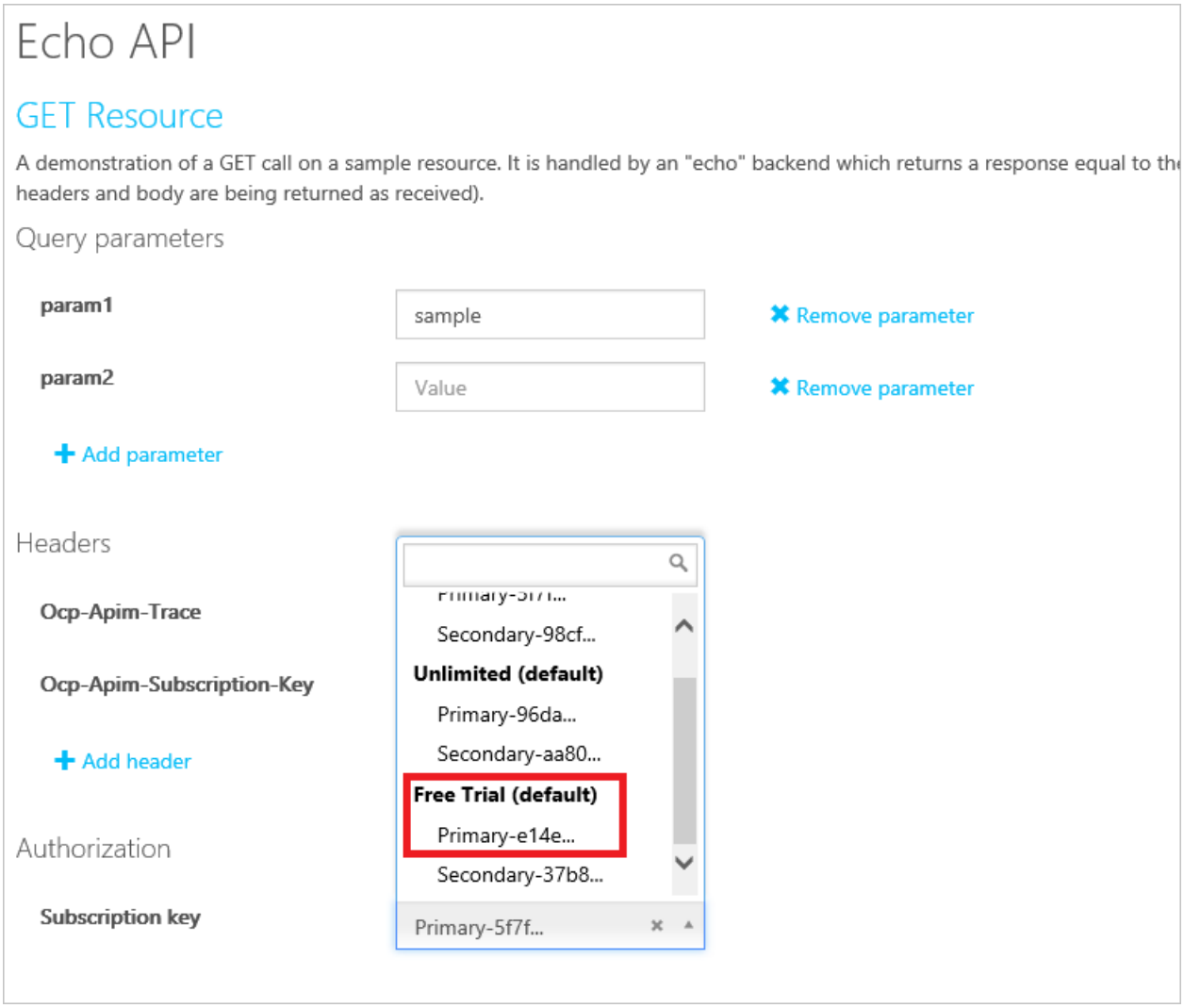
Click **APIs** in the top menu, and then click **Echo API**.



Click **GET Resource**, and then click **Try it**.

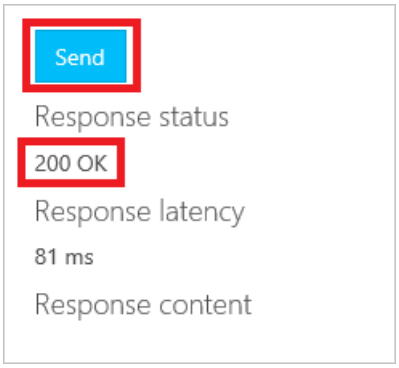


Keep the default parameter values, and then select your subscription key for the Free Trial product.

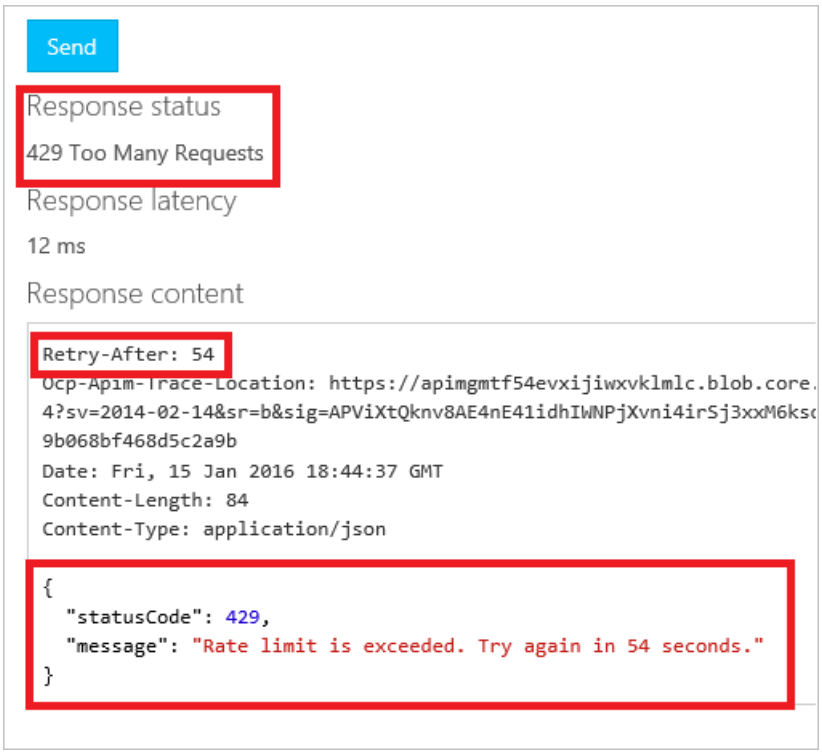


> **NOTE**
>
> If you have multiple subscriptions, be sure to select the key for **Free Trial**, or else the policies that were configured in the previous steps won't be in effect.

Click **Send**, and then view the response. Note the **Response status** of **200 OK**.



Click **Send** at a rate greater than the rate limit policy of 10 calls per minute. After the rate limit policy is exceeded, a response status of **429 Too Many Requests** is returned.



The **Response content** indicates the remaining interval before retries will be successful.

When the rate limit policy of 10 calls per minute is in effect, subsequent calls will fail until 60 seconds have elapsed from the first of the 10 successful calls to the product before the rate limit was exceeded. In this example, the remaining interval is 54 seconds.

Next steps

- Watch a demo of setting rate limits and quotas in the following video.

Add caching to improve performance in Azure API Management

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • Sigrid Elenga

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.

This guide shows you how to add response caching for your API and configure policies for the sample Echo API operations. You can then call the operation from the developer portal to verify caching in action.

> **NOTE**
> For information on caching items by key using policy expressions, see Custom caching in Azure API Management.

Prerequisites

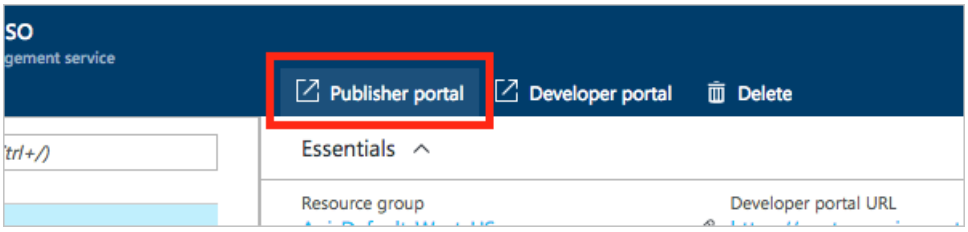
Before following the steps in this guide, you must have an API Management service instance with an API and a product configured. If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

Configure an operation for caching

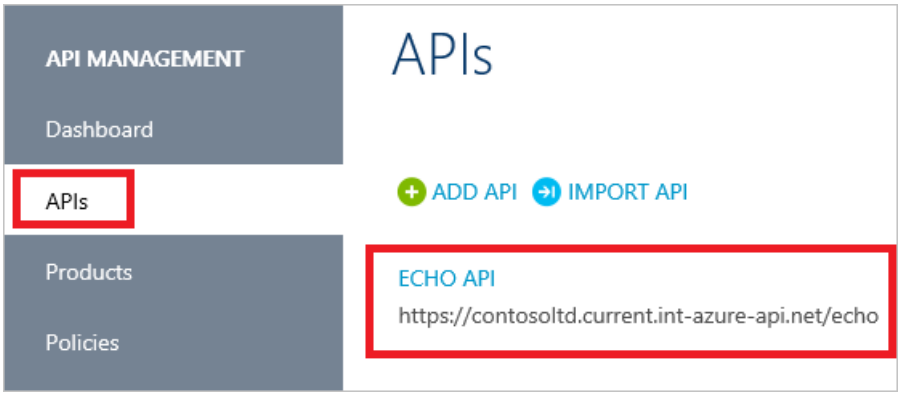
In this step, you will review the caching settings of the **GET Resource (cached)** operation of the sample Echo API.

> **NOTE**
> Each API Management service instance comes preconfigured with an Echo API that can be used to experiment with and learn about API Management. For more information, see Get started with Azure API Management.

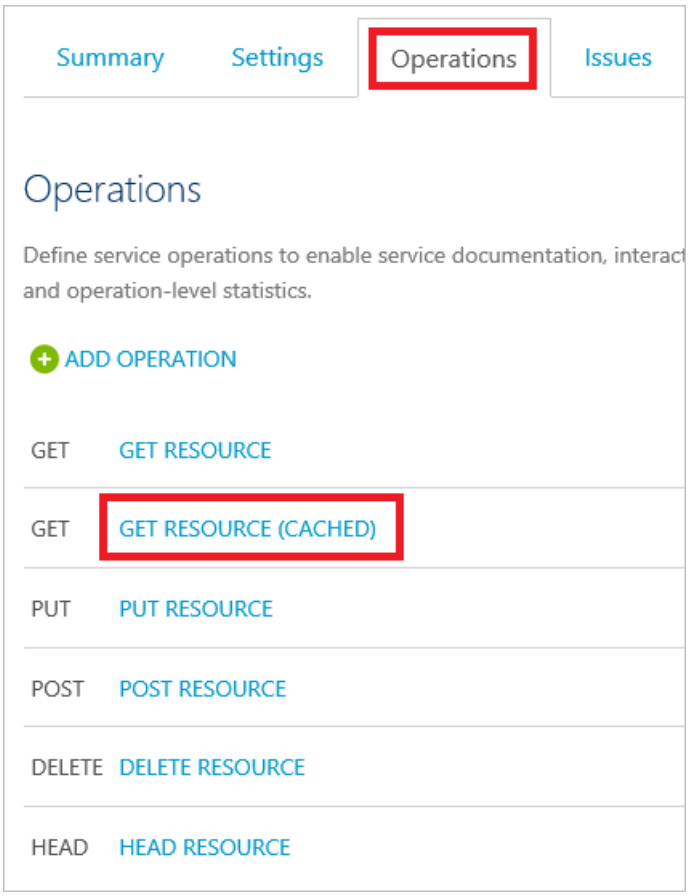
To get started, click **Publisher portal** in the Azure Portal for your API Management service. This takes you to the API Management publisher portal.



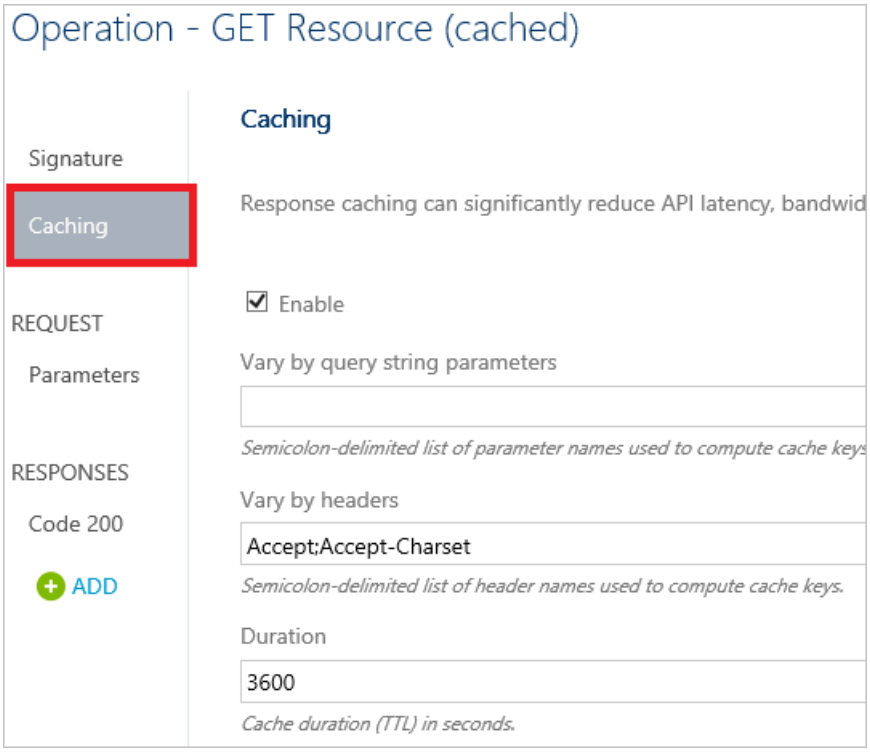
Click **APIs** from the **API Management** menu on the left, and then click **Echo API**.



Click the **Operations** tab, and then click the **GET Resource (cached)** operation from the **Operations** list.



Click the **Caching** tab to view the caching settings for this operation.



To enable caching for an operation, select the **Enable** check box. In this example, caching is enabled.

Each operation response is keyed, based on the values in the **Vary by query string parameters** and **Vary by headers** fields. If you want to cache multiple responses based on query string parameters or headers, you can configure them in these two fields.

Duration specifies the expiration interval of the cached responses. In this example, the interval is **3600** seconds, which is equivalent to one hour.

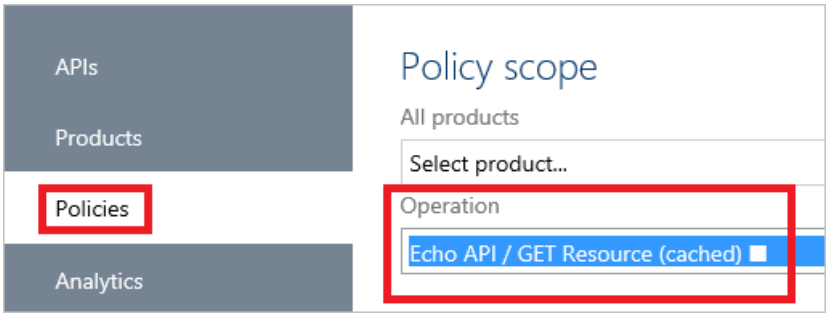
Using the caching configuration in this example, the first request to the **GET Resource (cached)** operation returns a response from the backend service. This response will be cached, keyed by the specified headers and query string parameters. Subsequent calls to the operation, with matching parameters, will have the cached response returned, until the cache duration interval has expired.

Review the caching policies

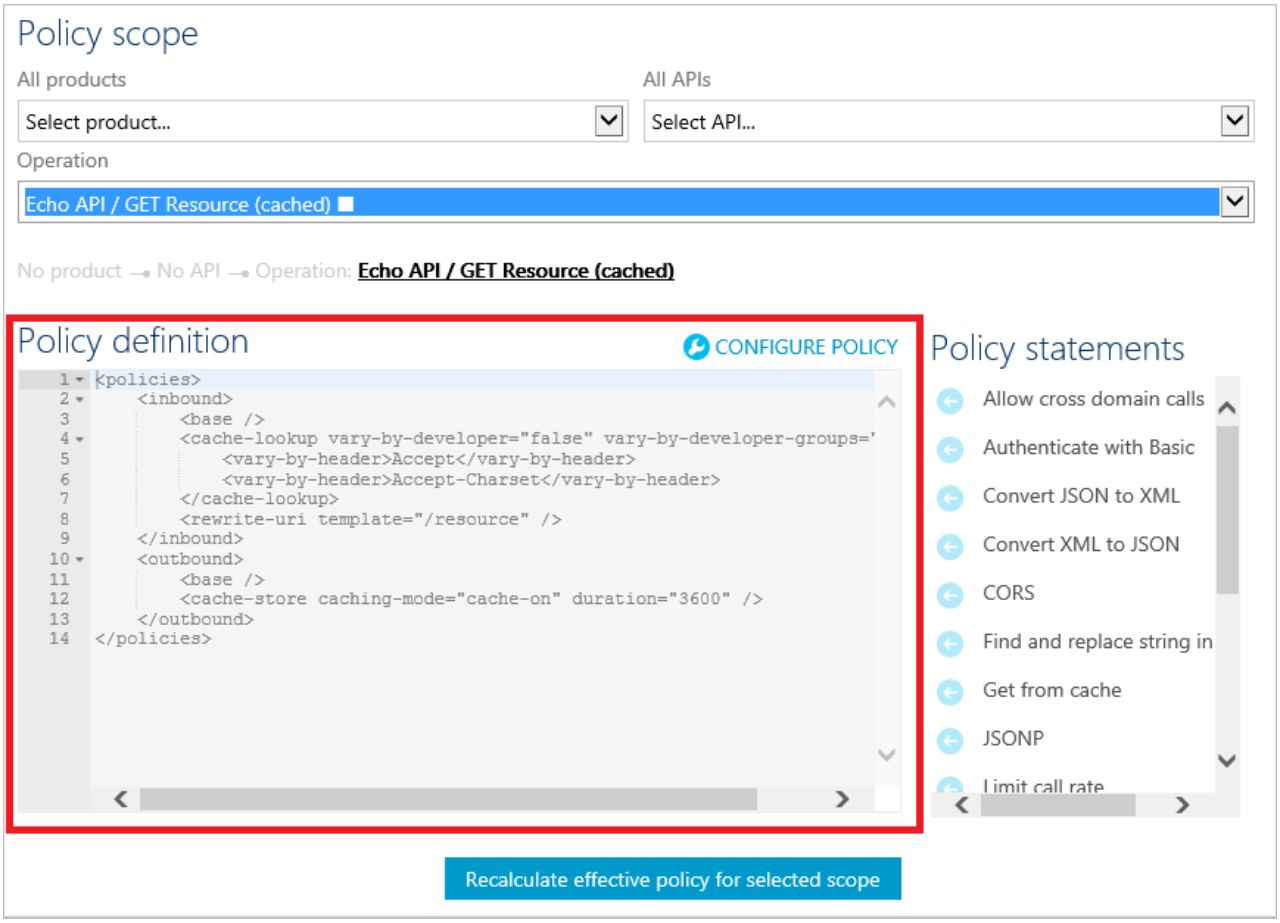
In this step, you review the caching settings for the **GET Resource (cached)** operation of the sample Echo API.

When caching settings are configured for an operation on the **Caching** tab, caching policies are added for the operation. These policies can be viewed and edited in the policy editor.

Click **Policies** from the **API Management** menu on the left, and then select **Echo API / GET Resource (cached)** from the **Operation** drop-down list.



This displays the policies for this operation in the policy editor.



The policy definition for this operation includes the policies that define the caching configuration that were reviewed using the **Caching** tab in the previous step.

```
<policies>
    <inbound>
        <base />
        <cache-lookup vary-by-developer="false" vary-by-developer-groups="false">
            <vary-by-header>Accept</vary-by-header>
            <vary-by-header>Accept-Charset</vary-by-header>
        </cache-lookup>
        <rewrite-uri template="/resource" />
    </inbound>
    <outbound>
        <base />
        <cache-store caching-mode="cache-on" duration="3600" />
    </outbound>
</policies>
```

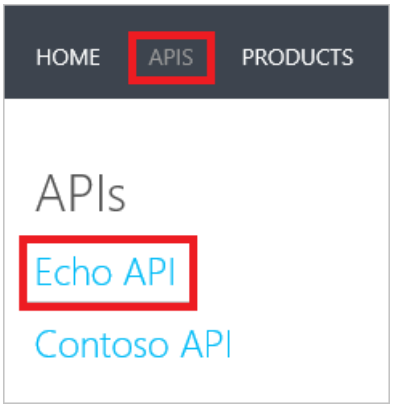
NOTE

Changes made to the caching policies in the policy editor will be reflected on the **Caching** tab of an operation, and vice-versa.

Call an operation and test the caching

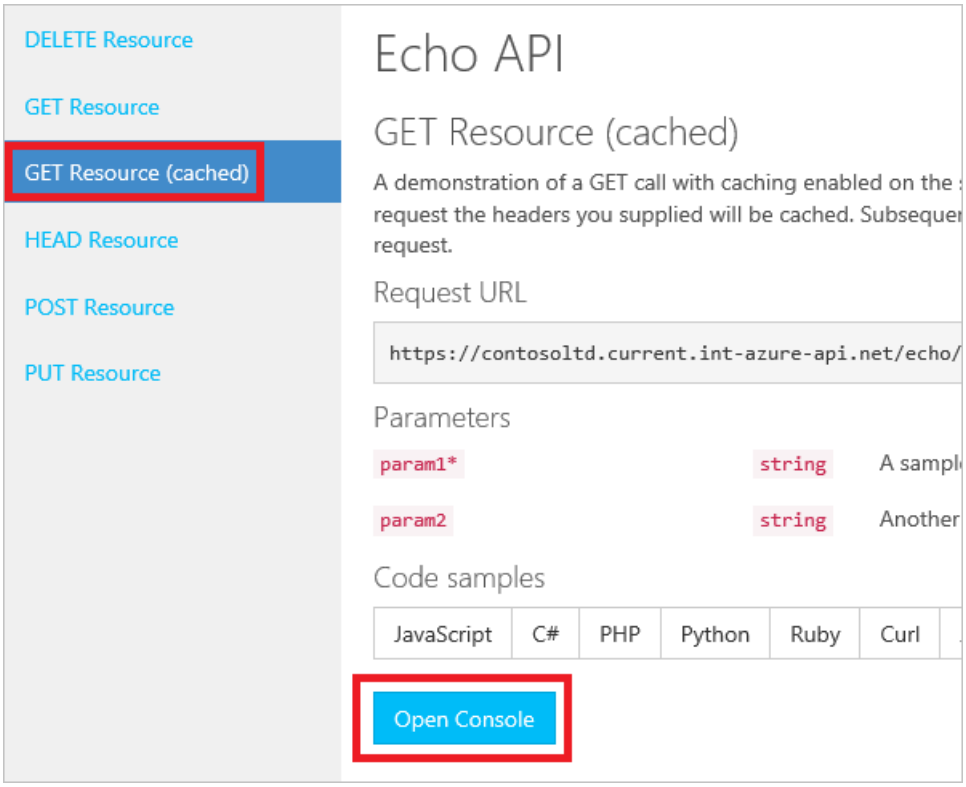
To see the caching in action, we can call the operation from the developer portal. Click **Developer portal** in the top right menu.

Click **APIs** in the top menu, and then select **Echo API**.

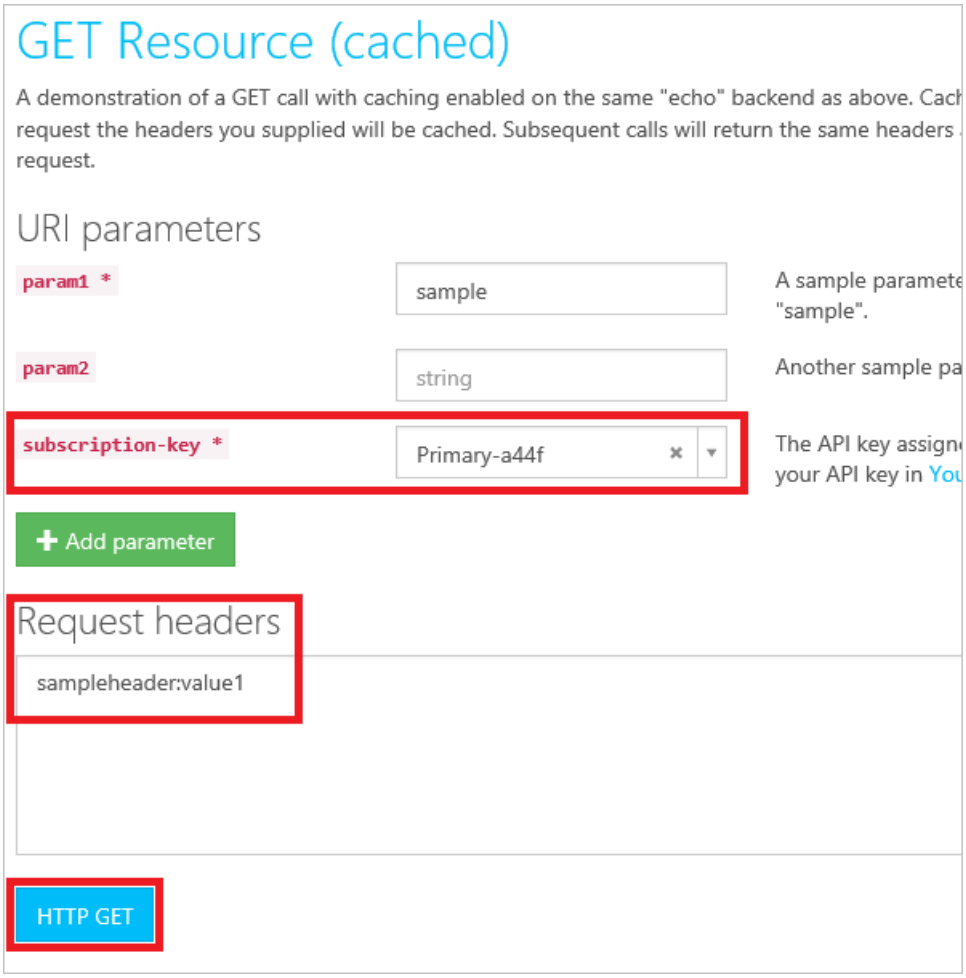


> If you have only one API configured or visible to your account, then clicking APIs takes you directly to the operations for that API.

Select the **GET Resource (cached)** operation, and then click **Open Console**.



The console allows you to invoke operations directly from the developer portal.



Keep the default values for **param1** and **param2**.

Select the desired key from the **subscription-key** drop-down list. If your account has only one subscription, it will already be selected.

Enter **sampleheader:value1** in the **Request headers** text box.

Click **HTTP Get** and make a note of the response headers.

Enter **sampleheader:value2** in the **Request headers** text box, and then click **HTTP Get**.

Note that the value of **sampleheader** is still **value1** in the response. Try some different values and note that the cached response from the first call is returned.

Enter **25** into the **param2** field, and then click **HTTP Get**.

Note that the value of **sampleheader** in the response is now **value2**. Because the operation results are keyed by query string, the previous cached response was not returned.

Next steps

- For more information about caching policies, see Caching policies in the API Management policy reference.
- For information on caching items by key using policy expressions, see Custom caching in Azure API Management.

Azure API Management FAQs

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

miaojiang • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson • MattFarm • ShawnJackson • Kristine Toliver • Vlad Vinogradsky

---

Get the answers to common questions, patterns, and best practices for Azure API Management.

Frequently asked questions

- How can I ask the Microsoft Azure API Management team a question?
- What does it mean when a feature is in preview?
- How can I secure the connection between the API Management gateway and my back-end services?
- How do I copy my API Management service instance to a new instance?
- Can I manage my API Management instance programmatically?
- How do I add a user to the Administrators group?
- Why is the policy that I want to add unavailable in the policy editor?
- How do I use API versioning in API Management?
- How do I set up multiple environments in a single API?
- Can I use SOAP with API Management?
- Is the API Management gateway IP address constant? Can I use it in firewall rules?
- Can I configure an OAuth 2.0 authorization server with AD FS security?
- What routing method does API Management use in deployments to multiple geographic locations?
- Can I use an Azure Resource Manager template to create an API Management service instance?
- Can I use a self-signed SSL certificate for a back end?
- Why do I get an authentication failure when I try to clone a GIT repository?
- Does API Management work with Azure ExpressRoute?
- Can I move an API Management service from one subscription to another?

How can I ask the Microsoft Azure API Management team a question?

You can contact us by using one of these options:

- Post your questions in our API Management MSDN forum.
- Send an email to apimgmt@microsoft.com.
- Send us a feature request in the Azure feedback forum.

What does it mean when a feature is in preview?

When a feature is in preview, it means that we're actively seeking feedback on how the feature is working for you. A feature in preview is functionally complete, but it's possible that we'll make a breaking change in response to customer feedback. We recommend that you don't depend on a feature that is in preview in your production environment. If you have any feedback on preview features, please let us know through one of the contact options in How can I ask the Microsoft Azure API Management team a question?.

How can I secure the connection between the API Management gateway and my back-end services?

You have several options to secure the connection between the API Management gateway and your back-end services. You can:

- Use HTTP basic authentication. For more information, see Configure API settings.
- Use SSL mutual authentication as described in How to secure back-end services by using client certificate authentication in Azure API Management.
- Use IP whitelisting on your back-end service. If you have a Standard or Premium tier API Management instance, the IP address of the gateway remains constant. You can set your whitelist to allow this IP address. You can get the IP address of your API Management instance on the Dashboard in the Azure portal.
- Connect your API Management instance to an Azure Virtual Network. For more information, see How to set up VPN connections in Azure API Management.

How do I copy my API Management service instance to a new instance?

You have several options if you want to copy an API Management instance to a new instance. You can:

- Use the backup and restore function in API Management. For more information, see How to implement disaster recovery by using service backup and restore in Azure API Management.
- Create your own backup and restore feature by using the API Management REST API. Use the REST API to save and restore the entities from the service instance that you want.
- Download the service configuration by using Git, and then upload it to a new instance. For more information, see How to save and configure your API Management service configuration by using Git.

Can I manage my API Management instance programmatically?

Yes, you can manage API Management programmatically by using:

- The API Management REST API.
- The Microsoft Azure ApiManagement Service Management Library SDK.
- The Service deployment and Service management PowerShell cmdlets.

How do I add a user to the Administrators group?

Here's how you can add a user to the Administrators group:

1. Sign in to the Azure portal.
2. Go to the resource group that has the API Management instance you want to update.
3. In API Management, assign the **Api Management Contributor** role to the user.

Now the newly added contributor can use Azure PowerShell cmdlets. Here's how to sign in as an administrator:

1. Use the `Login-AzureRmAccount` cmdlet to sign in.
2. Set the context to the subscription that has the service by using `Set-AzureRmContext -SubscriptionID <subscriptionGUID>`.
3. Get a single sign-on URL by using `Get-AzureRmApiManagementSsoToken -ResourceGroupName <rgName> -Name <serviceName>`.
4. Use the URL to access the admin portal.

Why is the policy that I want to add unavailable in the policy editor?

If the policy that you want to add appears dimmed or shaded in the policy editor, be sure that you are in the correct scope for the policy. Each policy statement is designed for you to use in specific scopes and policy sections. To review the policy sections and scopes for a policy, see the policy's Usage section in API Management policies.

How do I use API versioning in API Management?

You have a few options to use API versioning in API Management:

- In API Management, you can configure APIs to represent different versions. For example, you might have two different APIs, MyAPIv1 and MyAPIv2. A developer can choose the version that the developer wants to use.
- You also can configure your API with a service URL that doesn't include a version segment, for example, https://my.api. Then, configure a version segment on each operation's Rewrite URL template. For example, you

can have an operation with a URL template called /resource and a Rewrite URL template called /v1/Resource. You can change the version segment value separately for each operation.

- If you'd like to keep a "default" version segment in the API's service URL, on selected operations, set a policy that uses the Set backend service policy to change the back-end request path.

How do I set up multiple environments in a single API?

To set up multiple environments, for example, a test environment and a production environment, in a single API, you have two options. You can:

- Host different APIs on the same tenant.
- Host the same APIs on different tenants.

Can I use SOAP with API Management?

SOAP pass-through support is now available. Administrators can import the WSDL of their SOAP service, and Azure API Management will create a SOAP front end. Developer portal documentation, test console, policies and analytics are all available for SOAP services.

Is the API Management gateway IP address constant? Can I use it in firewall rules?

At the Standard and Premium tiers, the public IP address (VIP) of the API Management tenant is static for the lifetime of the tenant, with some exceptions. The IP address changes in these circumstances:

- The service is deleted and then re-created.
- The service subscription is suspended (for example, for nonpayment) and then reinstated.
- You add or remove Azure Virtual Network (you can use Virtual Network only at the Premium tier).

For multi-region deployments, the regional address changes if the region is vacated and then reinstated (you can use multi-region deployment only at the Premium tier).

Premium tier tenants that are configured for multi-region deployment are assigned one public IP address per region.

You can get your IP address (or addresses, in a multi-region deployment) on the tenant page in the Azure portal.

Can I configure an OAuth 2.0 authorization server with AD FS security?

To learn how to configure an OAuth 2.0 authorization server with Active Directory Federation Services (AD FS) security, see Using ADFS in API Management.

What routing method does API Management use in deployments to multiple geographic locations?

API Management uses the performance traffic routing method in deployments to multiple geographic locations. Incoming traffic is routed to the closest API gateway. If one region goes offline, incoming traffic is automatically routed to the next closest gateway. Learn more about routing methods in Traffic Manager routing methods.

Can I use an Azure Resource Manager template to create an API Management service instance?

Yes. See the Azure API Management Service QuickStart templates.

Can I use a self-signed SSL certificate for a back end?

Yes. Here's how to use a self-signed Secure Sockets Layer (SSL) certificate for a back end:

1. Create a Backend entity by using API Management.
2. Set the **skipCertificateChainValidation** property to **true**.
3. If you no longer want to allow self-signed certificates, delete the Backend entity, or set the **skipCertificateChainValidation** property to **false**.

Why do I get an authentication failure when I try to clone a Git repository?

If you use Git Credential Manager, or if you're trying to clone a Git repository by using Visual Studio, you might run into a known issue with the Windows credentials dialog box. The dialog box limits password length to 127

characters, and it truncates the Microsoft-generated password. We are working on shortening the password. For now, please use Git Bash to clone your Git repository.

Does API Management work with Azure ExpressRoute?

Yes. API Management works with Azure ExpressRoute.

Can I move an API Management service from one subscription to another?

Yes. To learn how, see Move resources to a new resource group or subscription.

How to create and publish a product in Azure API Management

11/15/2016 • 4 min to read • Edit on GitHub

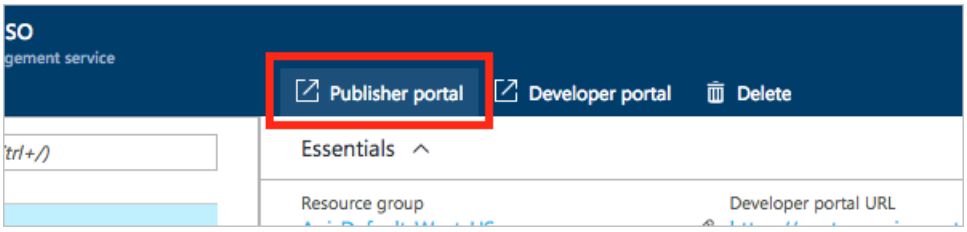
Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

In Azure API Management, a product contains one or more APIs as well as a usage quota and the terms of use. Once a product is published, developers can subscribe to the product and begin to use the product's APIs. The topic provides a guide to creating a product, adding an API, and publishing it for developers.

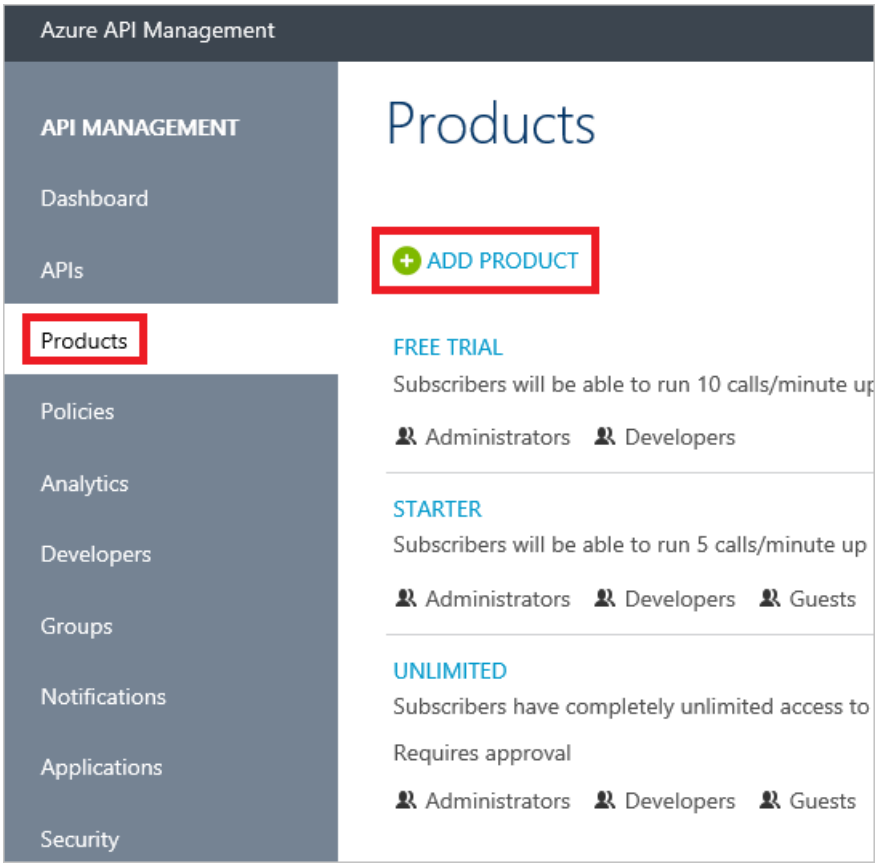
Create a product

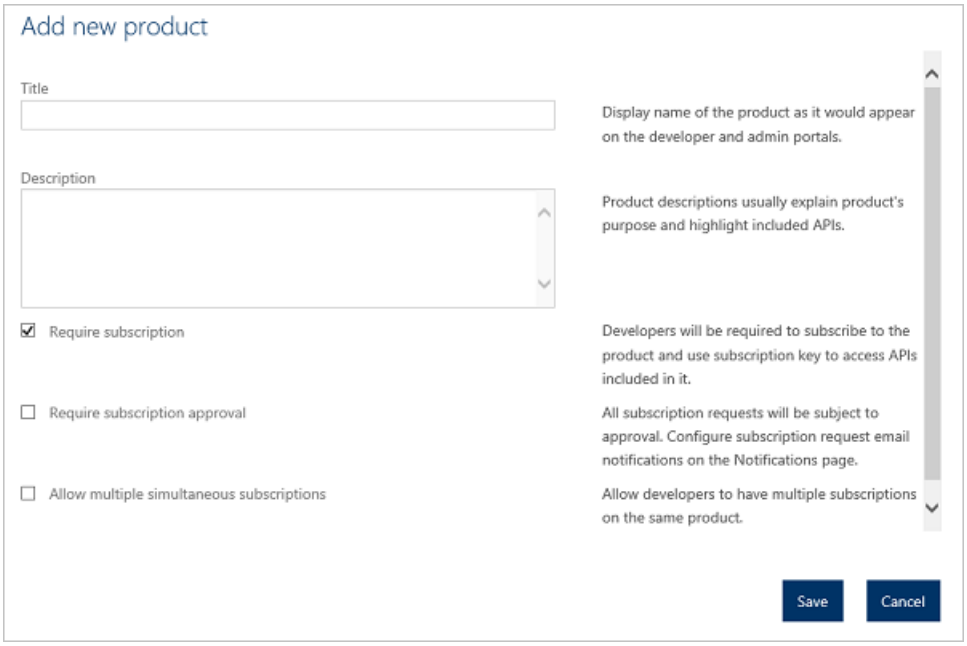
Operations are added and configured to an API in the publisher portal. To access the publisher portal, click **Publisher portal** in the Azure Portal for your API Management service.



> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

Click on **Products** in the menu on the left to display the **Products** page, and click **Add Product**.



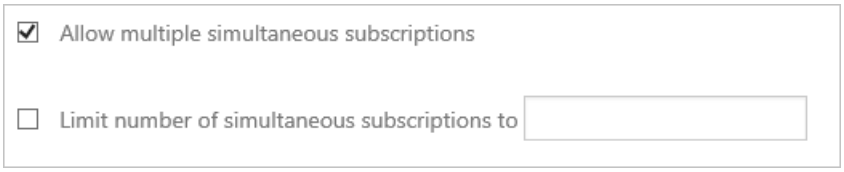


Enter a descriptive name for the product in the **Name** field and a description of the product in the **Description** field.

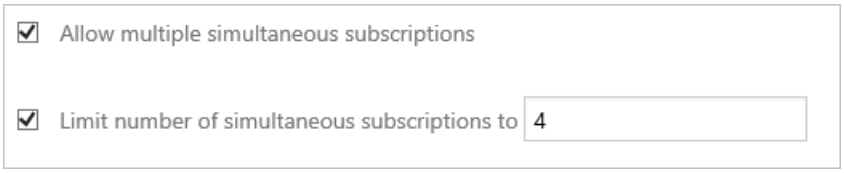
Products in API Management can be **Open** or **Protected**. Protected products must be subscribed to before they can be used, while open products can be used without a subscription. Check **Require subscription** to create a protected product that requires a subscription. This is the default setting.

Check **Require subscription approval** if you want an administrator to review and accept or reject subscription attempts to this product. If the box is unchecked, subscription attempts will be auto-approved. For more information on subscriptions, see View subscribers to a product.

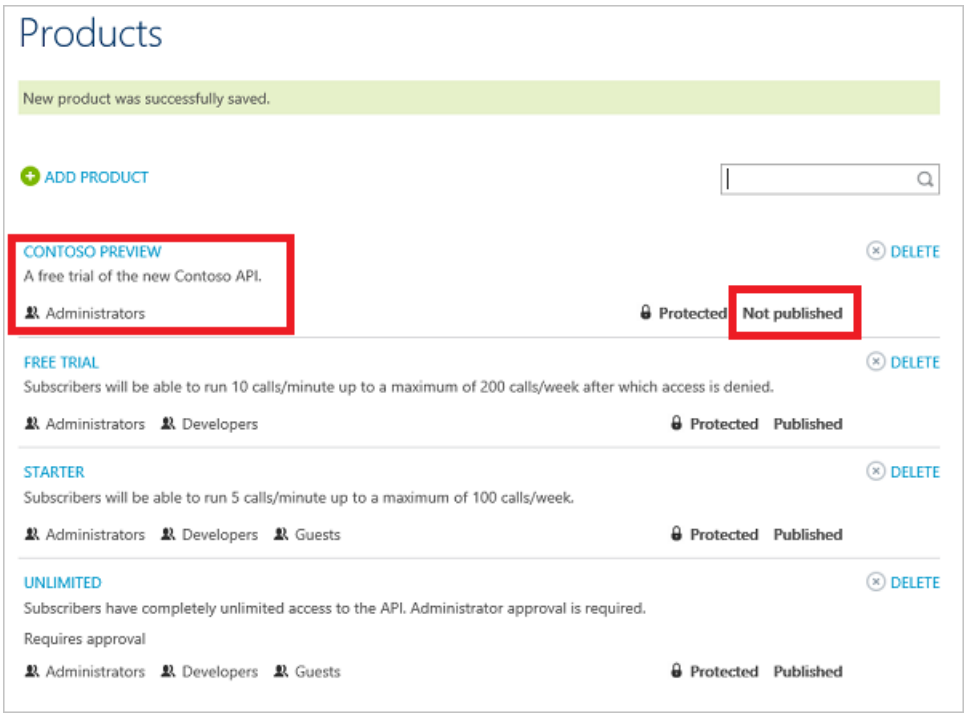
To allow developer accounts to subscribe multiple times to the product, check the **Allow multiple subscriptions** check box. If this box is not checked, each developer account can subscribe only a single time to the product.



To limit the count of multiple simultaneous subscriptions, check the **Limit number of simultaneous subscriptions to** check box and enter the subscription limit. In the following example, simultaneous subscriptions are limited to four per developer account.



Once all new product options are configured, click **Save** to create the new product.

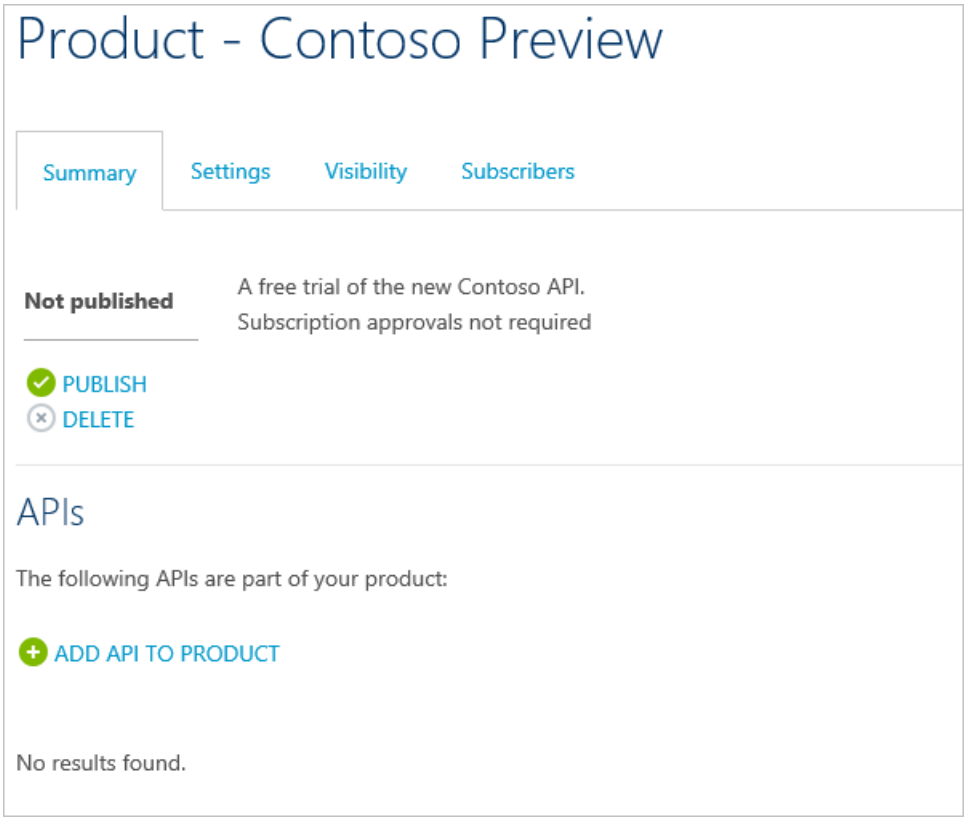


> By default new products are unpublished, and are visible only to the **Administrators** group.

To configure a product, click on the product name in the **Products** tab.

Add APIs to a product

The **Products** page contains four links for configuration: **Summary**, **Settings**, **Visibility**, and **Subscribers**. The **Summary** tab is where you can add APIs and publish or unpublish a product.

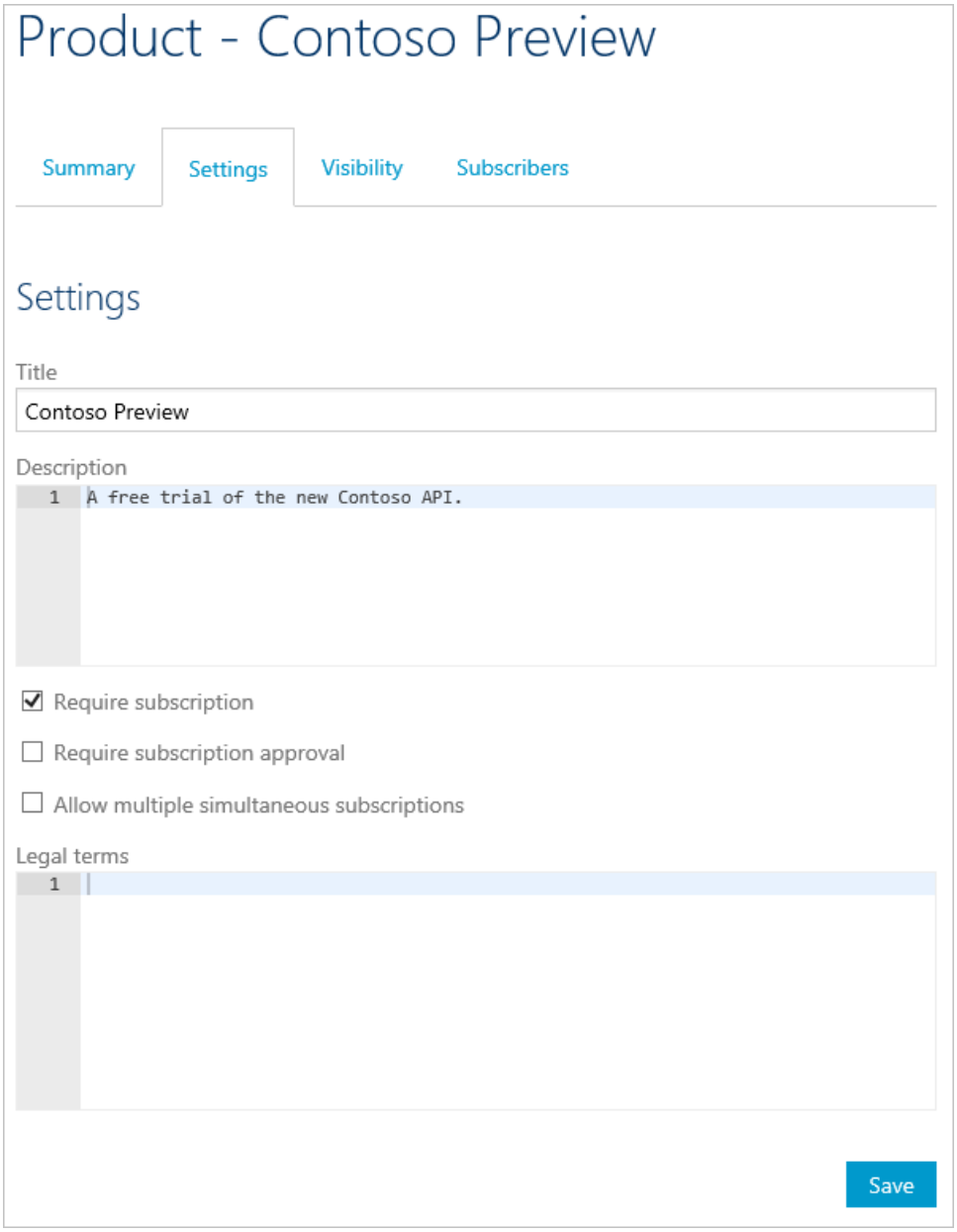


Before publishing your product you need to add one or more APIs. To do this, click **Add API to product**.

Select the desired APIs and click **Save**.

Add descriptive information to a product

The **Settings** tab allows you to provide detailed information about the product such as its purpose, the APIs it provides access to, and other useful information. The content is targeted at the developers that will be calling the API and can be written in plain text or HTML markup.



Check **Require subscription** to create a protected product that requires a subscription to be used, or clear the checkbox to create an open product that can be called without a subscription.

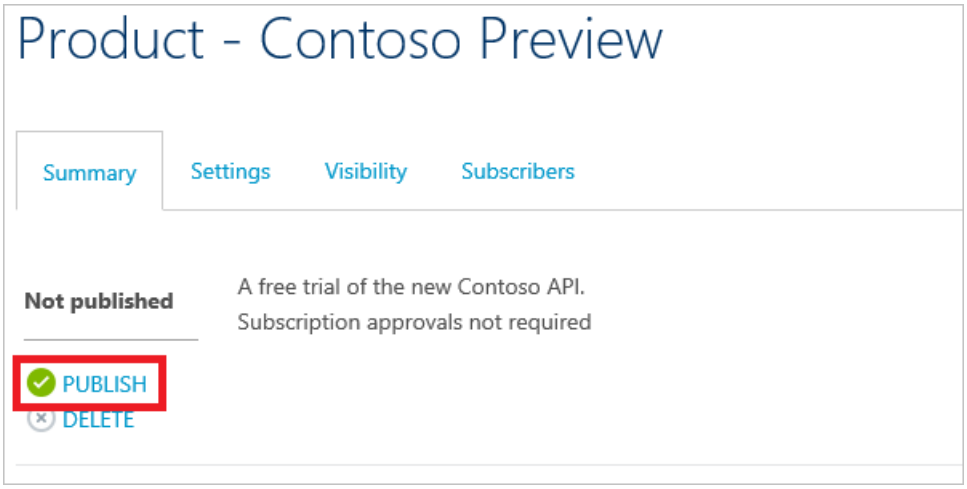
Select **Require subscription approval** if you want to manually approve all product subscription requests. By default all product subscriptions are granted automatically.

To allow developer accounts to subscribe multiple times to the product, check the **Allow multiple subscriptions** check box and optionally specify a limit. If this box is not checked, each developer account can subscribe only a single time to the product.

Optionally fill in the **Terms of use** field describing the terms of use for the product which subscribers must accept in order to use the product.

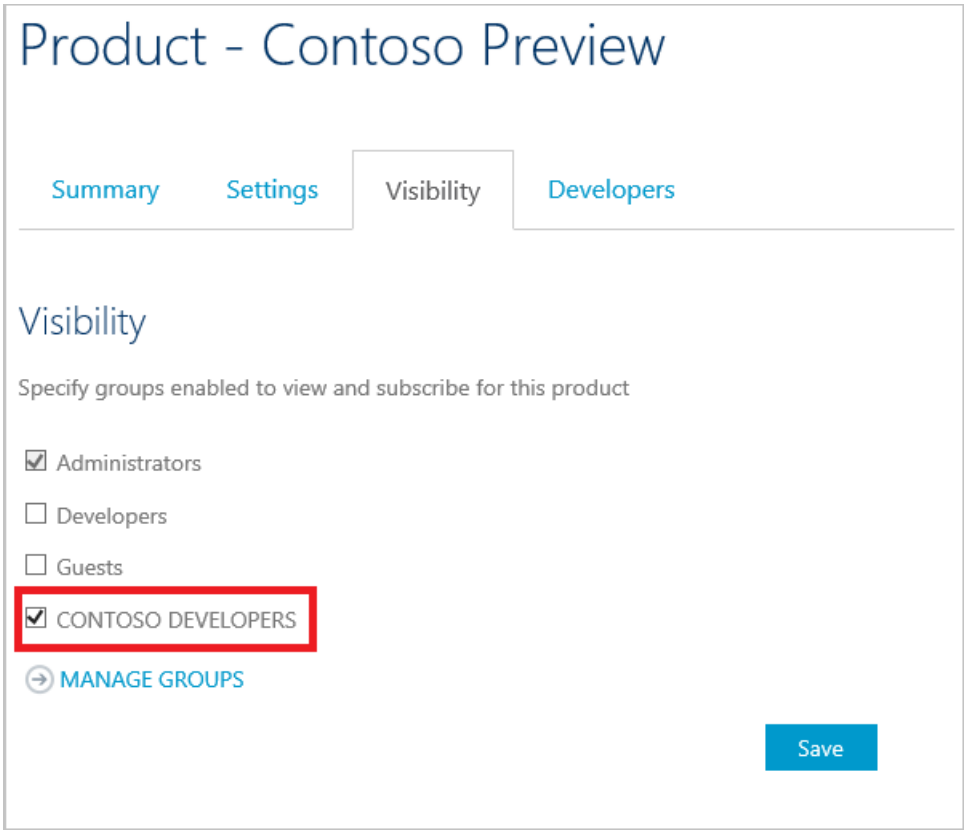
Publish a product

Before the APIs in a product can be called, the product must be published. On the **Summary** tab for the product, click **Publish**, and then click **Yes, publish it** to confirm. To make a previously published product private, click **Unpublish**.



Make a product visible to developers

The **Visibility** tab allows you to choose which roles are able to see the product on the developer portal and subscribe to the product.

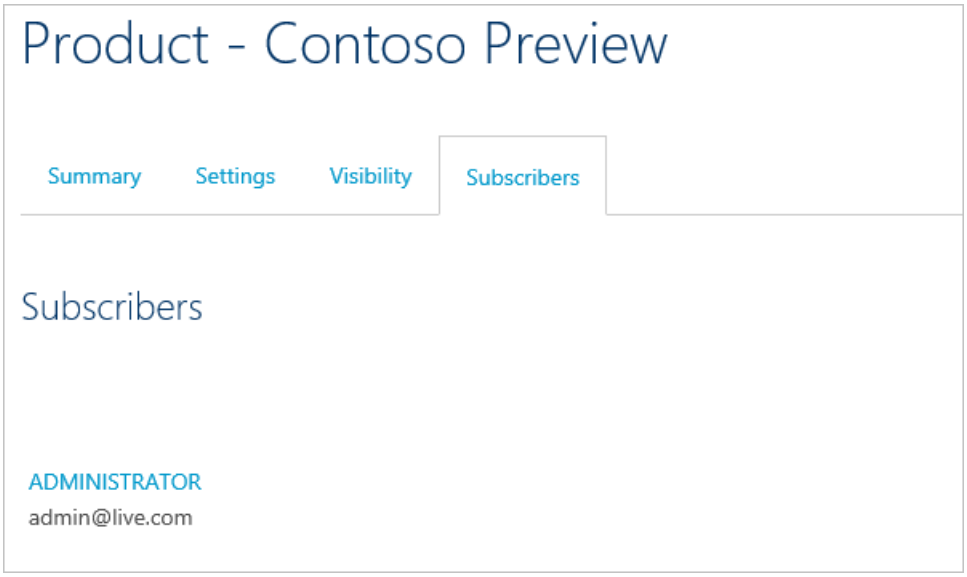


To enable or disable visibility of a product for the developers in a group, check or uncheck the check box beside the group and then click **Save**.

> For more information, see How to create and use groups to manage developer accounts in Azure API Management.

View subscribers to a product

The **Subscribers** tab lists the developers who have subscribed to the product. The details and settings for each developer can be viewed by clicking on the developer's name. In this example no developers have yet subscribed to the product.



Next steps

Once the desired APIs are added and the product published, developers can subscribe to the product and begin to call the APIs. For a tutorial that demonstrates these items as well as advanced product configuration see How create and configure advanced product settings in Azure API Management.

For more information about working with products, see the following video.

How to add operations to an API in Azure API Management

11/15/2016 • 4 min to read • Edit on GitHub

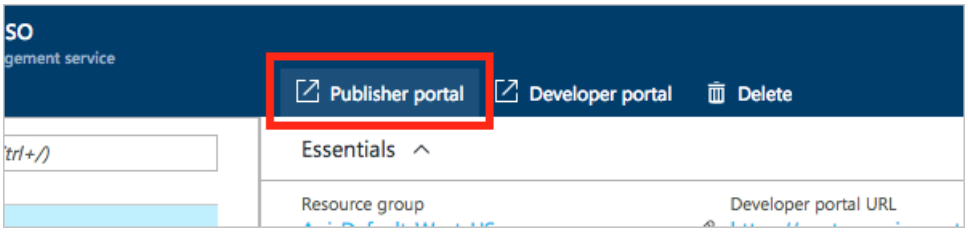
Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

Before an API in API Management can be used, operations must be added. This guide shows how to add and configure different types of operations to an API in API Management.

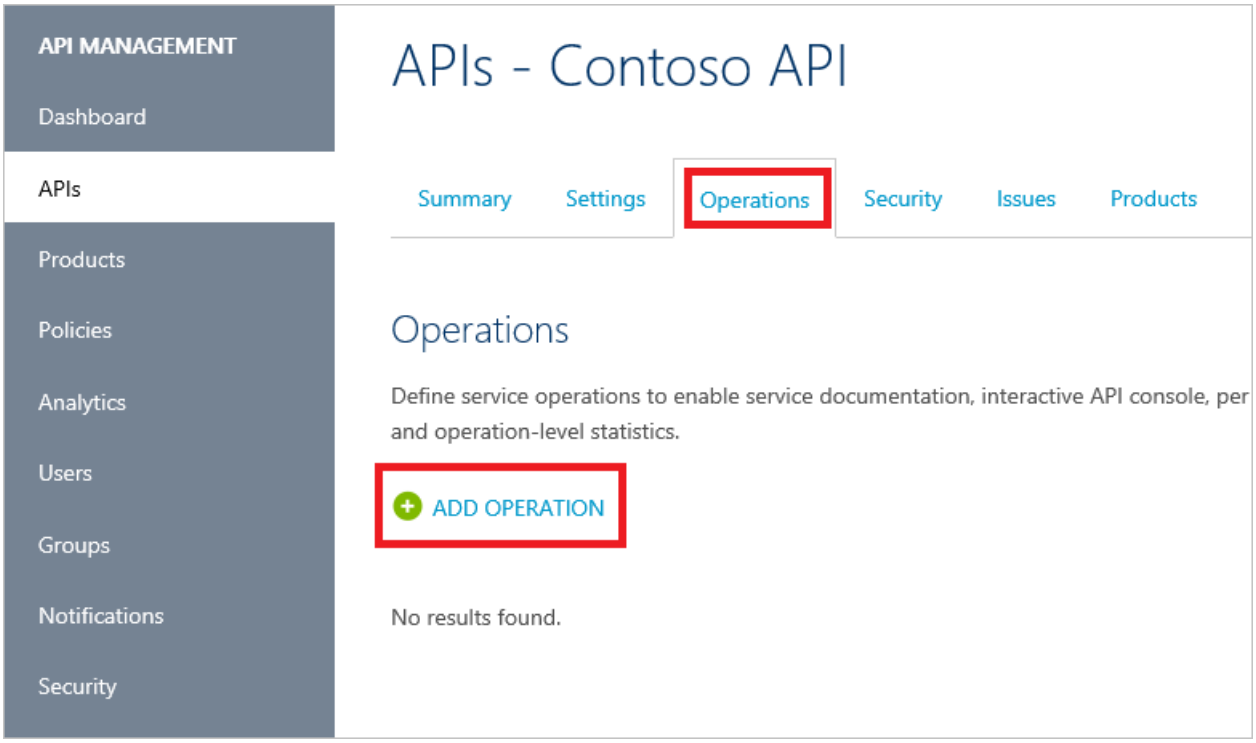
Add an operation

Operations are added and configured to an API in the publisher portal. To access the publisher portal, click **Publisher portal** in the Azure Portal for your API Management service.

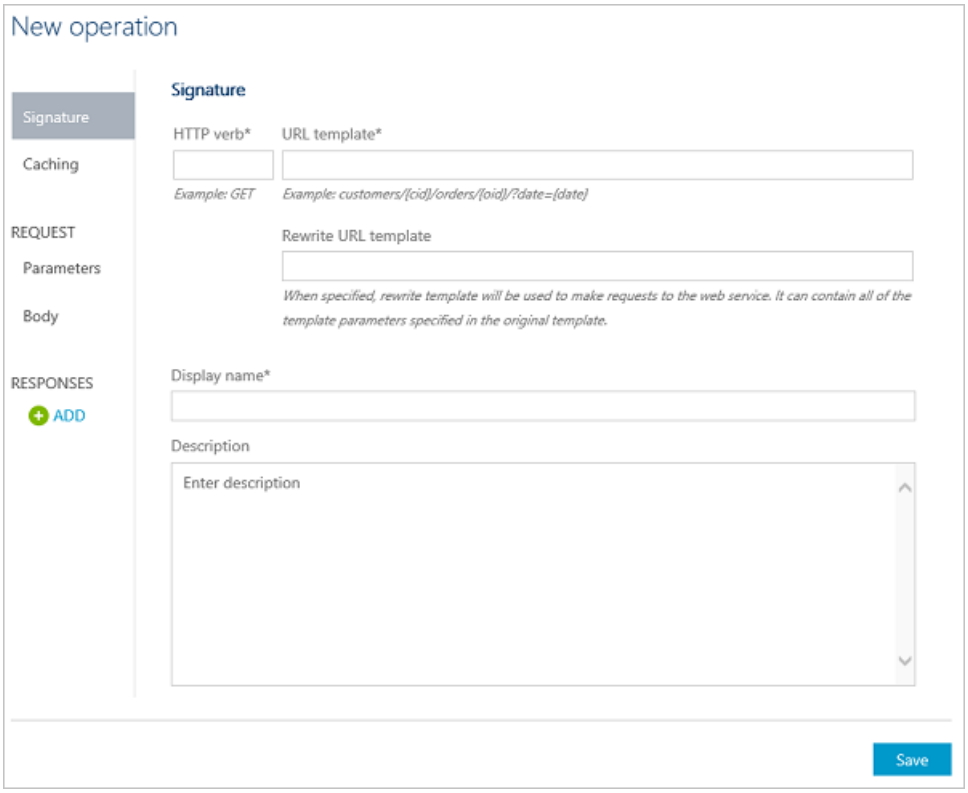


> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

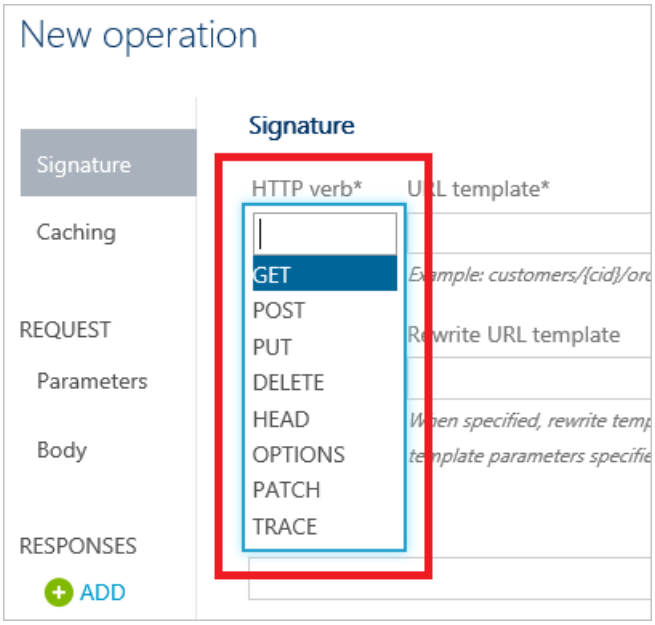
Select the desired API in the publisher portal and then select the **Operations** tab.



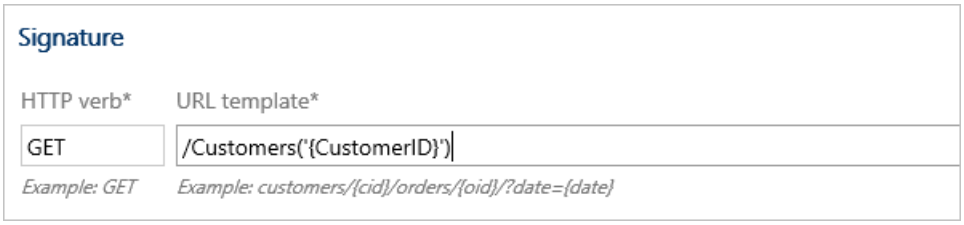
Click **Add Operation** to add a new operation. The **New operation** will be displayed and the **Signature** tab will be selected by default.



Specify the **HTTP verb** by choosing from the drop-down list.

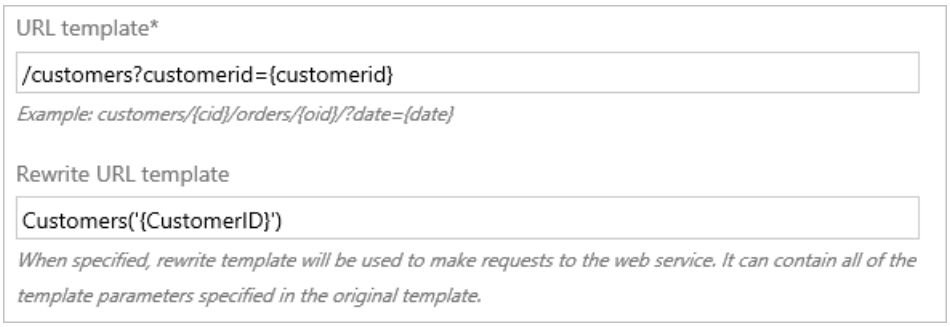


Define the URL template by typing in a URL fragment consisting of one or more URL path segments and zero or more query string parameters. The URL template, appended to the base URL of the API, identifies a single HTTP operation. It may contain one or more named variable parts that are identified by curly braces. These variable parts are called template parameters and are dynamically assigned values extracted from the request's URL when the request is being processed by the API Management platform.



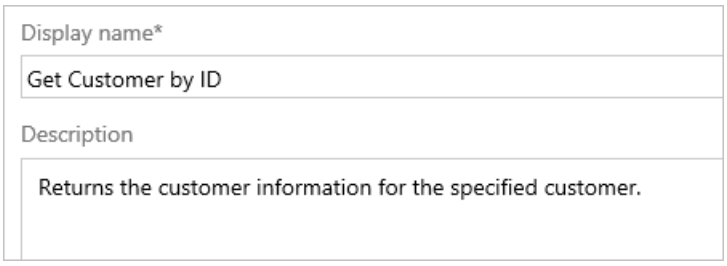
If desired, specify the **Rewrite URL template**. This allows you to use the standard URL template for processing incoming requests on the front-end, while calling the back-end via a converted URL according to the rewrite

template. Template parameters from the URL template should be used in the rewrite template. The following example shows how content type encoded as path segment in the web service from the previous example can be provided as a query parameter in the API published via the API Management platform using the URL templates.



Callers to the operation will use the format `/customers?customerid=ALFKI` and this will be mapped to `/Customers('ALFKI')` when the back-end service is invoked.

Display name and **Description** provide a description of the operation and are used to provide documentation to the developers using this API in the developer portal.

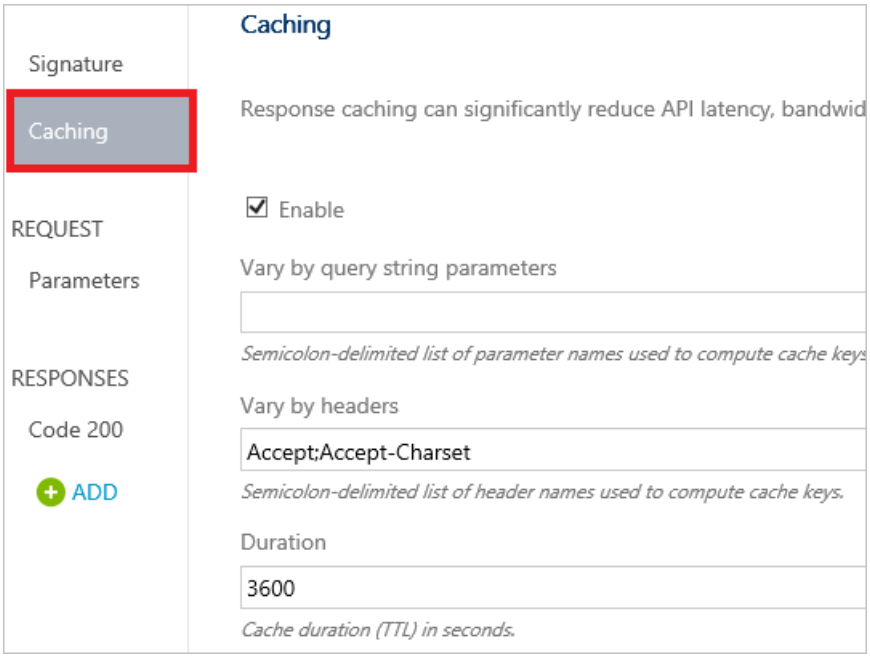


The operation description can be specified as plain text or HTML in the **Description** text box.

Operation caching

Response caching reduces latency perceived by the API consumers, lowers bandwidth consumption and decreases the load on the HTTP web service implementing the API.

To easily and quickly enable caching for the operation, select the **Caching** tab and check the **Enable** checkbox.



Duration specifies the time period during which the operation response remains in the cache. The default value is 3600 seconds or 1 hour.

Cache keys are used to differentiate between responses so that the response corresponding to each different cache key will get its own separate cached value. Optionally, enter specific query string parameters and/or HTTP headers to be used in computing cache key values in the **Vary by query string parameters** and **Vary by headers** text boxes respectively. When none are specified, full request URL and the following HTTP header values are used in cache key generation: **Accept** and **Accept-Charset**.

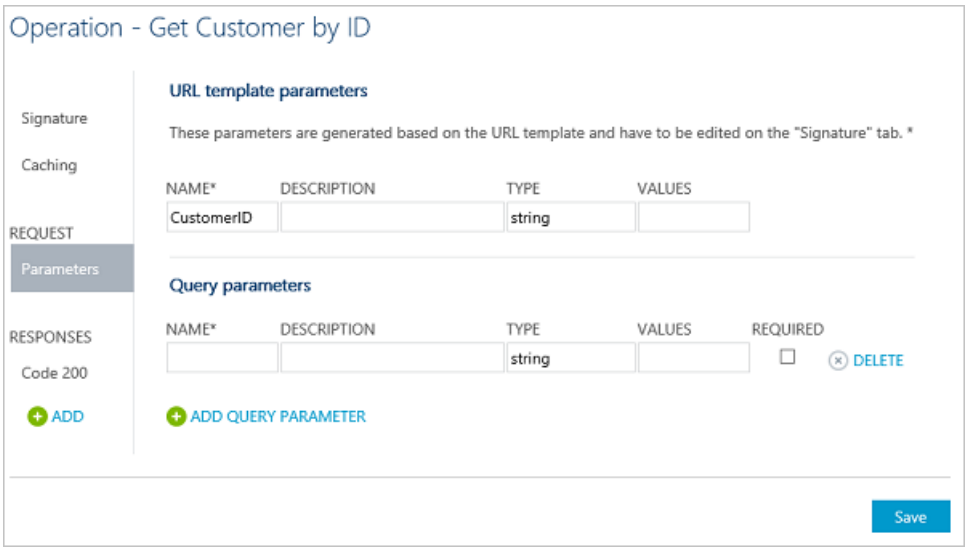
> For more information on caching and caching policies, see How to cache operation results in Azure API Management.

Request parameters

Operation parameters are managed on the Parameters tab. Parameters specified in the **URL Template** on the **Signature** tab are added automatically and can be changed only by editing the URL template. Additional parameters can be entered manually.

To add a new query parameter, click **Add Query Parameter** and enter the following information:

- **Name** - parameter name.
- **Description** - a brief description of the parameter (optional).
- **Type** - parameter type, selected in the drop down.
- **Values** - values that can be assigned to this parameter. One of the values can be marked as default (optional).
- **Required** - make the parameter mandatory by checking the checkbox.



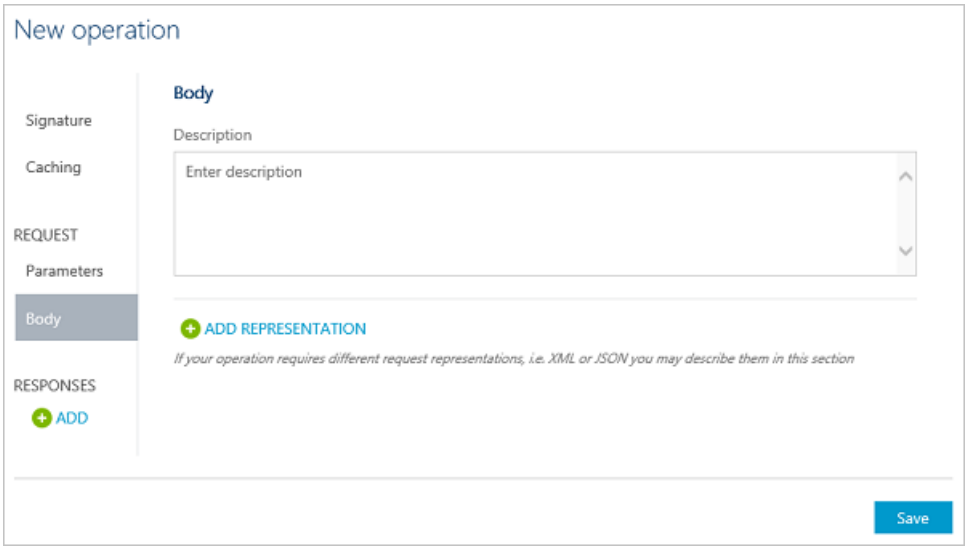
Request body

If the operation allows (e.g. PUT, POST) and requires a body you may provide an example of it in all of the supported representation formats (e.g. json, XML).

> The request body is used for documentation purposes only and is not validated.

To enter a request body, switch to the **Body** tab.

Click **Add Representation**, start typing desired content type name (e.g. application/json), select it in the drop-down, and paste the desired request body example in the selected format into the text box.

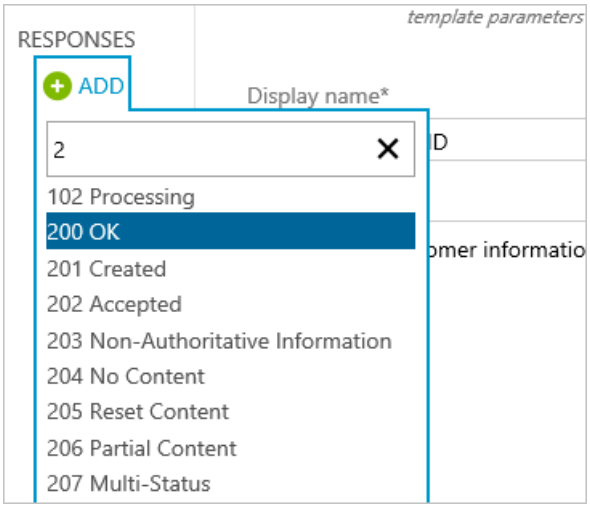


In additional to representations, you can also specify an optional text description in the **Description** text box.

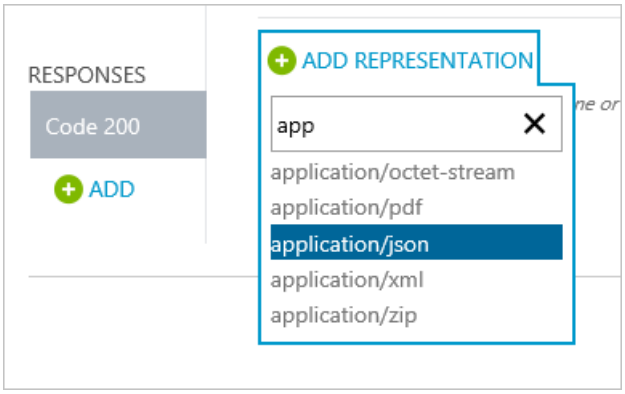
Responses

It is a good practice to provide examples of responses for all status codes that the operation may produce. Each status code may have more than one response body example, one for each of the supported content types.

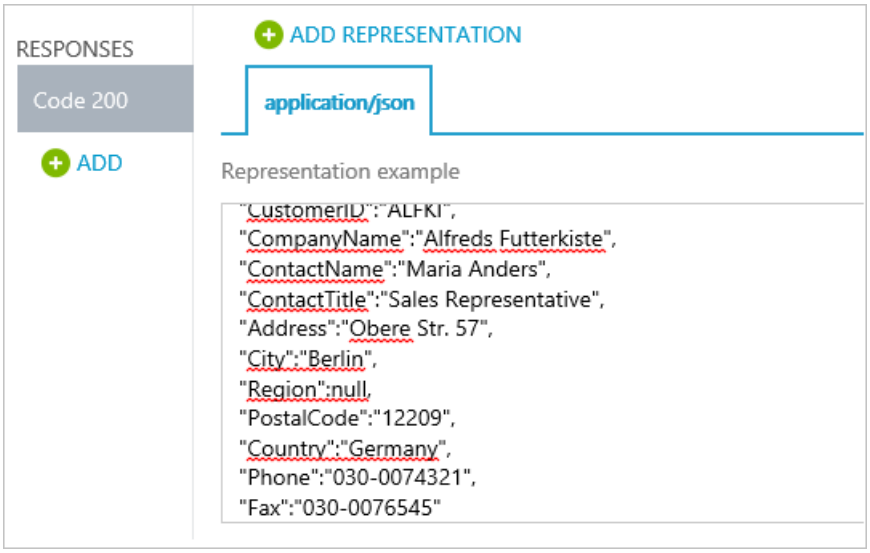
To add a response, click **Add** and start typing the desired status code. In this example the status code is **200 OK**. Once the code is displayed in the drop-down, select it, and the response code is created and added to your operation.



Click **Add Representation**, start typing the desired content type name (e.g. application/json) and then select it in the drop down.



Paste the response body example in the selected format into the text box.



If desired, add an optional description into the **Description** text box.

Once the operation is configured, click **Save**.

Next steps

Once the operations are added to an API, the next step is to associate the API with a product and publish it so that developers can call its operations.

- How to create and publish a product

How to create APIs in Azure API Management

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • v-aljenk • Jamie Strusz

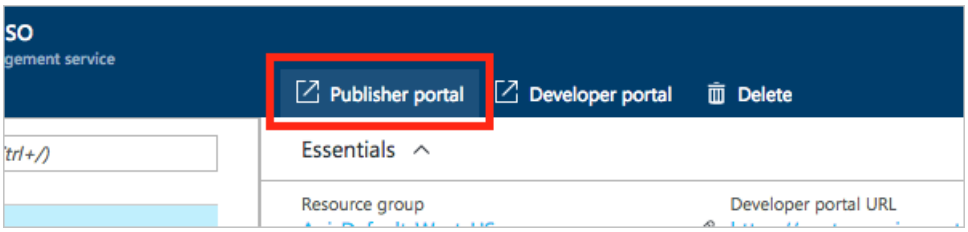
---

An API in API Management represents a set of operations that can be invoked by client applications. New APIs are created in the publisher portal, and then the desired operations are added. Once the operations are added, the API is added to a product and can be published. Once an API is published, it can be subscribed to and used by developers.

This guide shows the first step in the process: how to create and configure a new API in API Management. For more information on adding operations and publishing a product, see How to add operations to an API and How to create and publish a product.

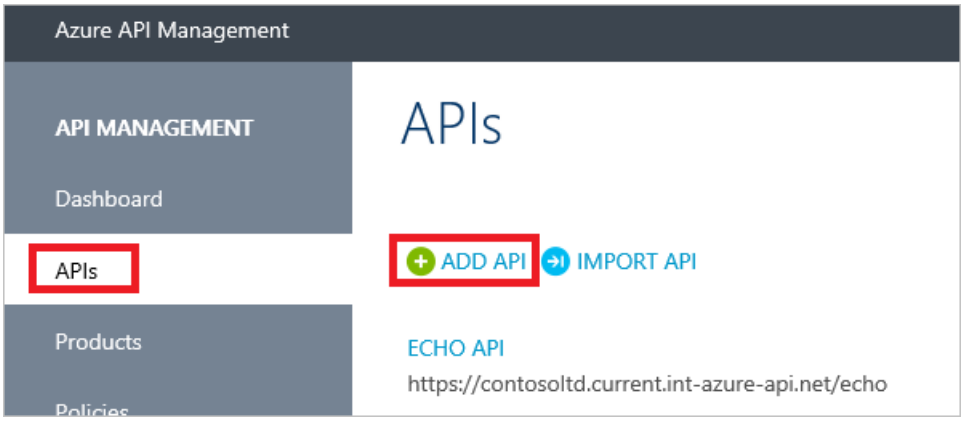
Create a new API

APIs are created and configured in the publisher portal. To access the publisher portal, click **Publisher portal** in the Azure Portal for your API Management service.

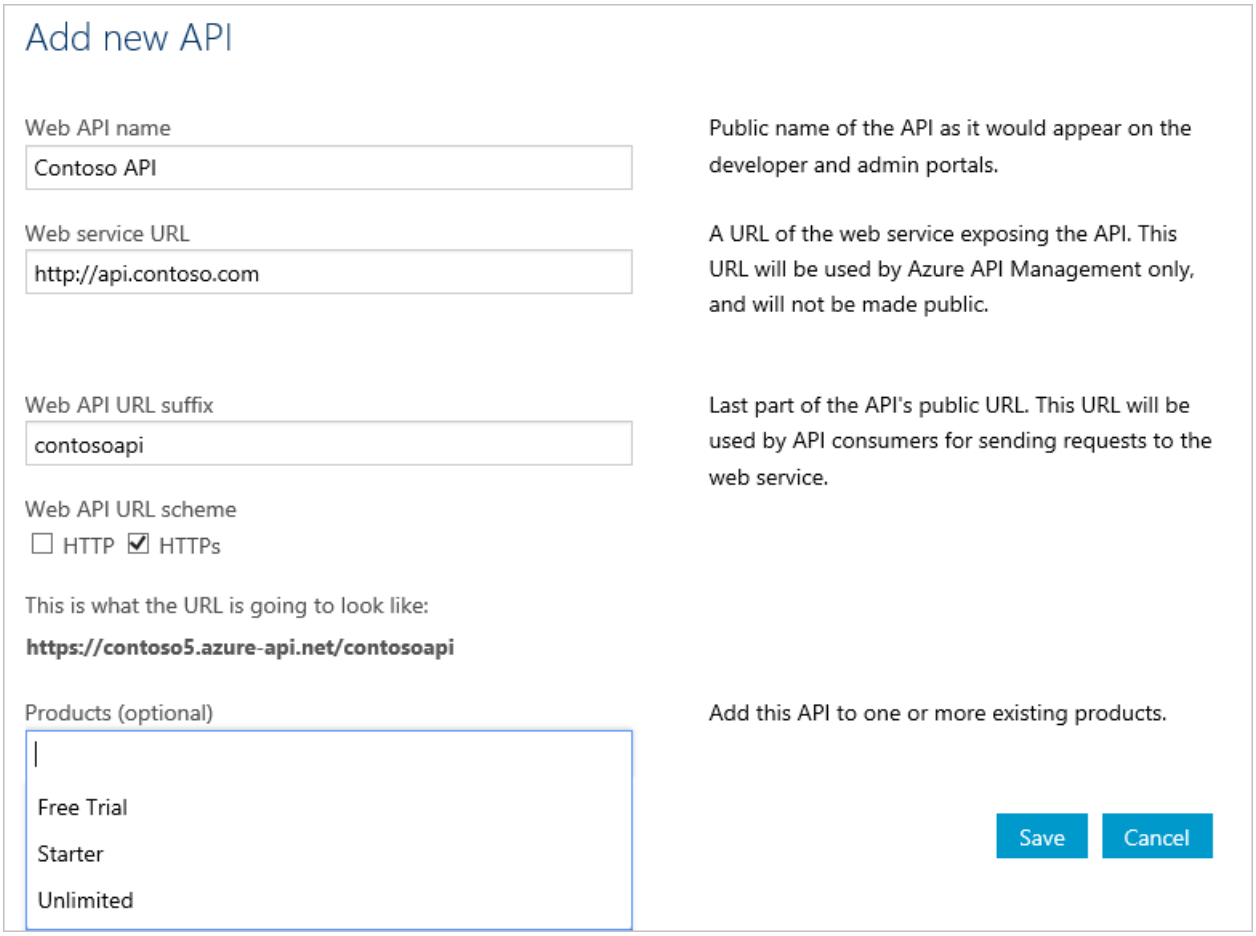


> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

Click **APIs** from the **API Management** menu on the left, and then click **add API**.



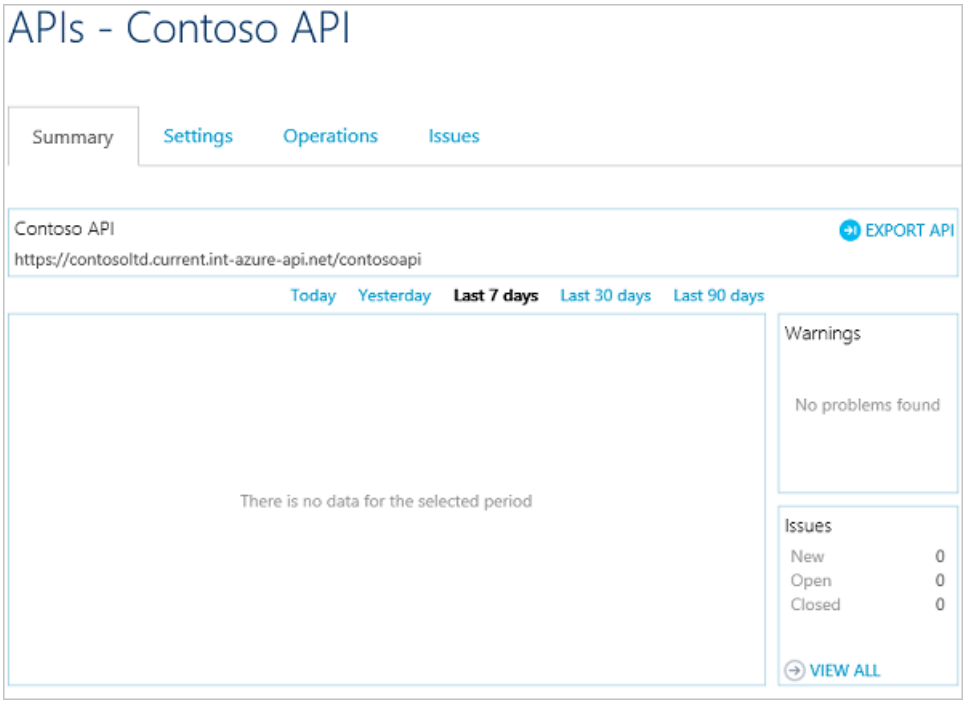
Use the **Add new API** window to configure the new API.



The following fields are used to configure the new API.

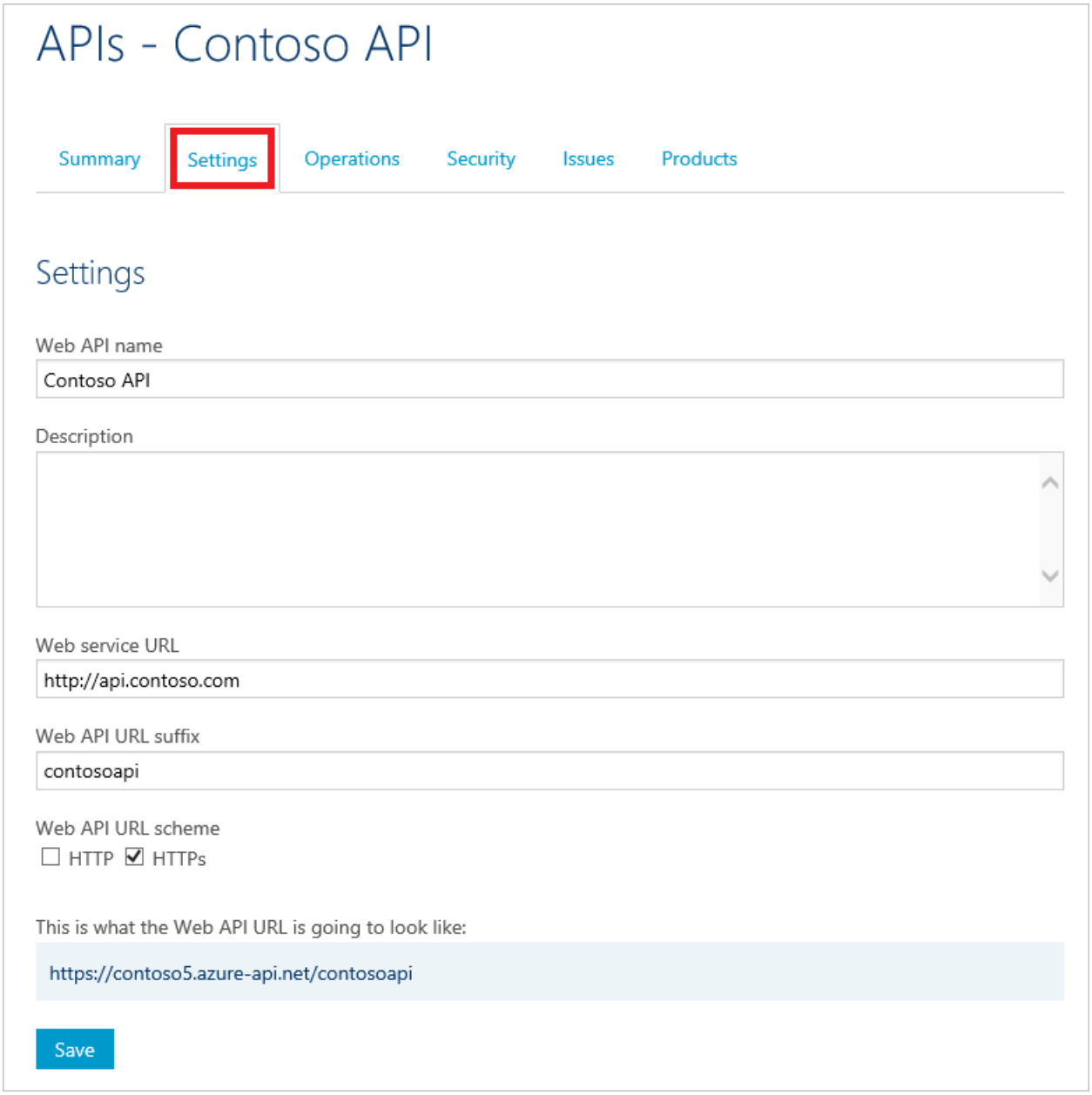
- **Web API name** provides a unique and descriptive name for the API. It is displayed in the developer and publisher portals.
- **Web service URL** references the HTTP service implementing the API. API management forwards requests to this address.
- **Web API URL suffix** is appended to the base URL for the API management service. The base URL is common for all APIs hosted by an API Management service instance. API Management distinguishes APIs by their suffix and therefore the suffix must be unique for every API for a given publisher.
- **Web API URL scheme** determines which protocols can be used to access the API. HTTPs is specified by default.
- To optionally add this new API to a product, click the **Products (optional)** drop-down and choose a product. This step can be repeated multiple times to add the API to multiple products.

Once the desired values are configured, click **Save**. Once the new API is created, the summary page for the API is displayed in the publisher portal.



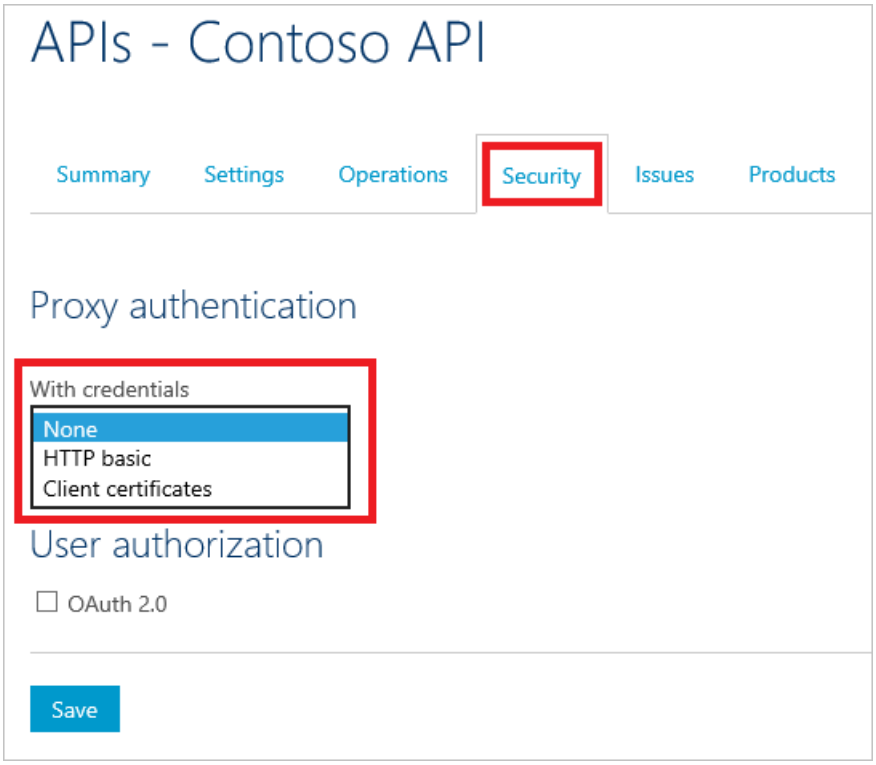
Configure API settings

You can use the **Settings** tab to verify and edit the configuration for an API. **Web API name**, **Web service URL**, and **Web API URL suffix** are initially set when the API is created and can be modified here. **Description** provides an optional description, and **Web API URL scheme** determines which protocols can be used to access the API.



To configure gateway authentication for the backend service implementing the API, select the **Security** tab. The **With credentials** drop-down can be used to configure **HTTP basic** or **Client certificates** authentication. To use HTTP basic authentication, simply enter the desired credentials. For information on using client certificate authentication, see How to secure back-end services using client certificate authentication in Azure API Management.

The **Security** tab can also be used to configure **User authorization** using OAuth 2.0. For more information, see How to authorize developer accounts using OAuth 2.0 in Azure API Management.



Click **Save** to save any changes you make to the API settings.

Next steps

Once an API is created and the settings configured, the next steps are to add the operations to the API, add the API to a product, and publish it so that it is available for developers. For more information, see the following articles.

- How to add operations to an API
- How to create and publish a product

How to import the definition of an API with operations in Azure API Management

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

In API Management, new APIs can be created and the operations added manually, or the API can be imported along with the operations in one step.

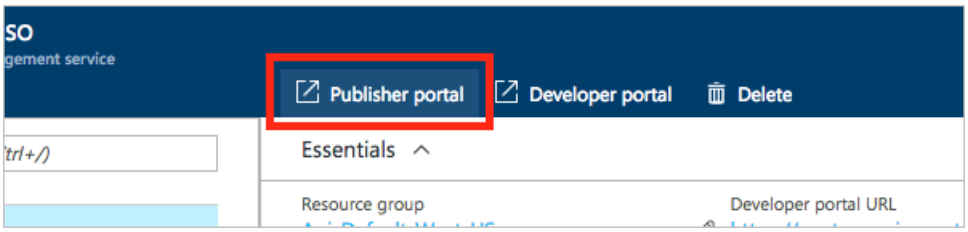
APIs and their operations can be imported using the following formats.

- WADL
- Swagger

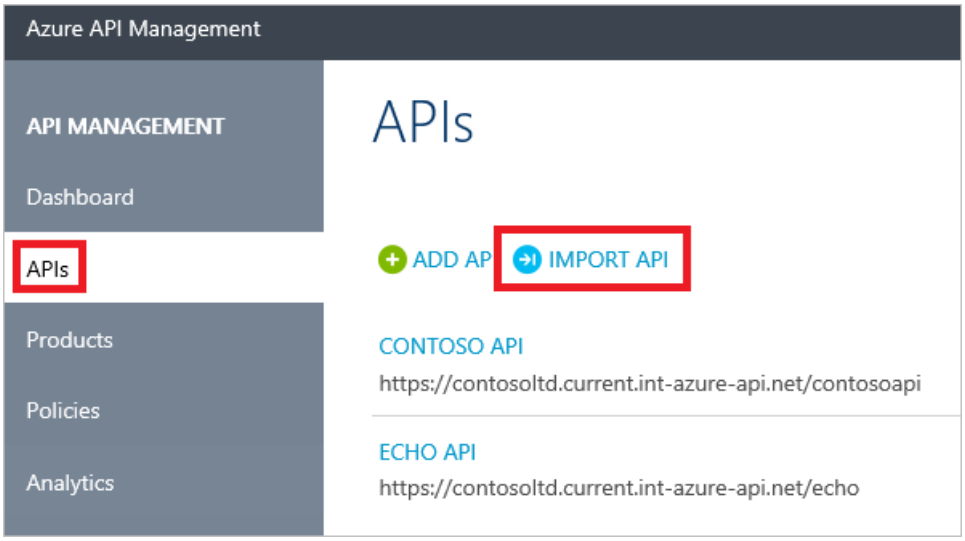
This guide shows how create a new API and import its operations in one step. For information on manually creating an API and adding operations, see How to create APIs and How to add operations to an API.

Import an API

APIs are created and configured in the publisher portal. To access the publisher portal, click **Publisher portal** in the Azure Portal for your API Management service. If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.



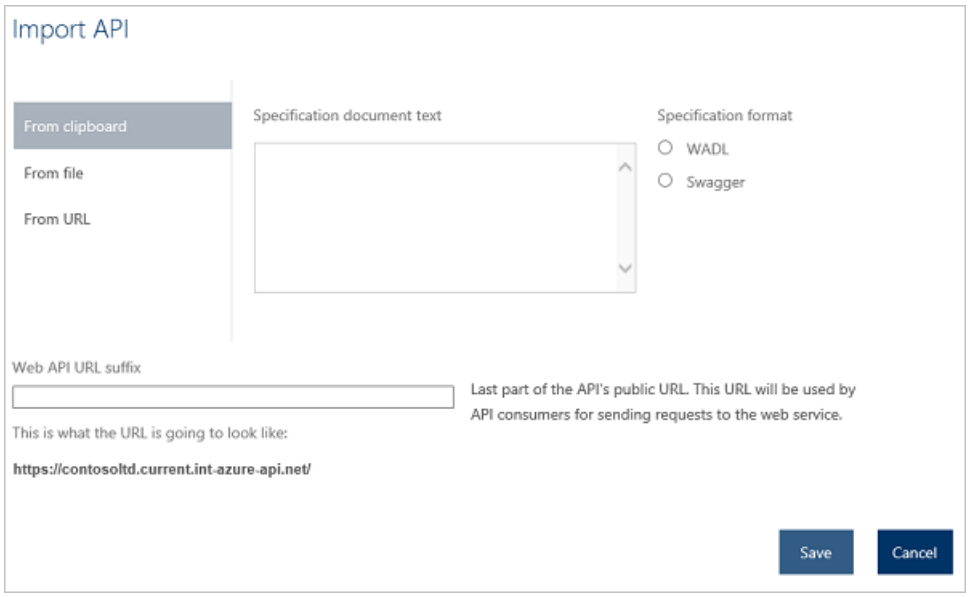
Click **APIs** from the **API Management** menu on the left, and then click **import API**.



The **Import API** window has three tabs that correspond to the three ways to provide the API specification.

- **From clipboard** allows you to paste the API specification into the designated text box.

- **From file** allows you to browse to and select the file that contains the API specification.
- **From URL** allows you to supply the URL to the specification for the API.



After providing the API specification, use the radio buttons on the right to indicate the specification format. The following formats are supported.

- WADL
- Swagger

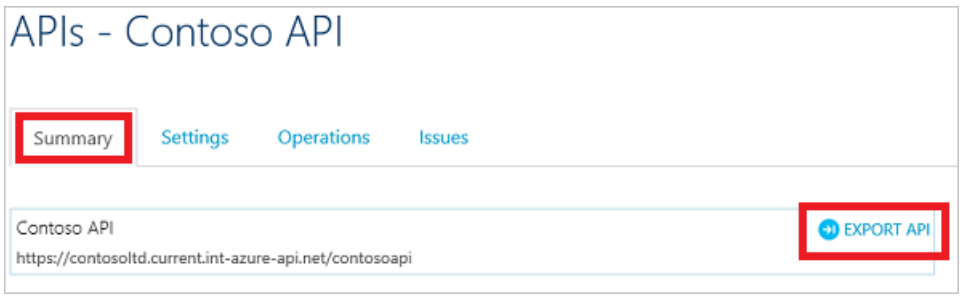
Next, enter a **Web API URL suffix**. This is appended to the base URL for your API management service. The base URL is common for all APIs hosted on each instance of an API Management service. API Management distinguishes APIs by their suffix and therefore the suffix must be unique for every API in a specific API management service instance.

Once all values are entered, click **Save** to create the API and the associated operations.

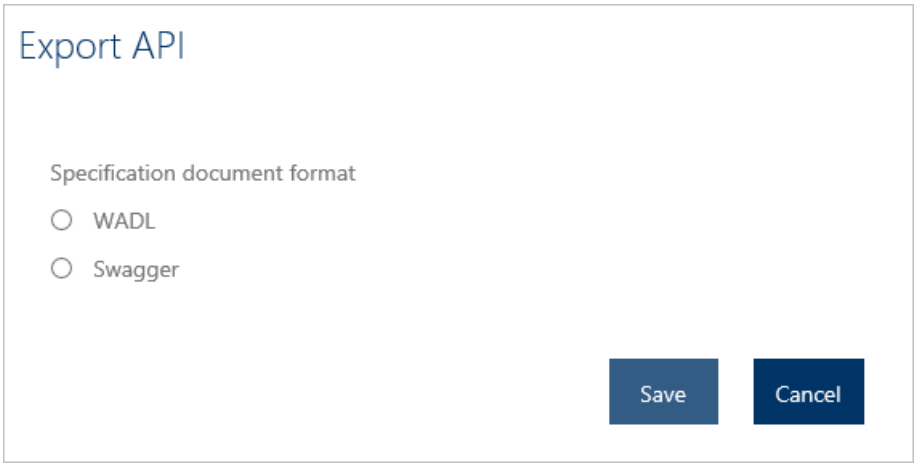
> **NOTE**
> For a tutorial of importing a basic calculator API in Swagger format, see Manage your first API in Azure API Management.

Export an API

In addition to importing new APIs, you can export the definitions of your APIs from the publisher portal. To do so, click **Export API** from the **Summary tab** of your **API**.



APIs can be exported using WADL or Swagger. Select the desired format, click **Save**, and choose the location in which to save the file.



Next steps

Once an API is created and the operations imported, you can review and configure any additional settings, add the API to a Product, and publish it so that it is available for developers. For more information, see the following guides.

- How to configure API settings
- How to create and publish a product

How to setup VPN connections in Azure API Management

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Anton Babadjanov (Microsoft) • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson • Vlad Vinogradsky • miaojiang • v-aljenk • Jamie Strusz

API Management's VPN support allows you to connect your API Management gateway to an Azure Virtual Network (classic). This allows API Management customers to securely connect to their backend web services that are on-premises or are otherwise inaccessible to the public internet.

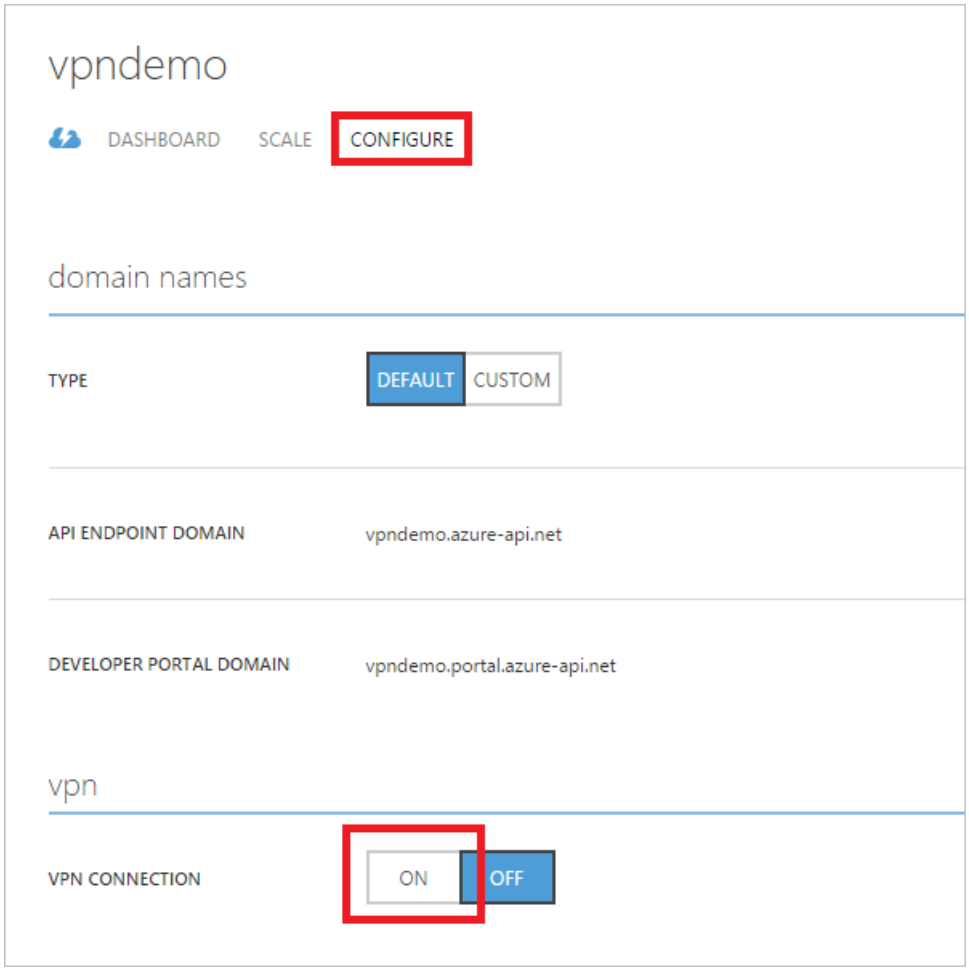
> **NOTE**
> Azure API Management works with classic VNETs. For information on creating a classic VNET, see Create a virtual network (classic) by using the Azure Portal. For information on connecting classic VNETs to ARM VNETS, see Connecting classic VNets to new VNets.

Enable VPN connections

> VPN connectivity is only available in the **Premium** and **Developer** tiers. To switch to it, open your API Management service in the Azure Classic Portal and then open the **Scale** tab. Under the **General** section select the Premium tier and click Save.

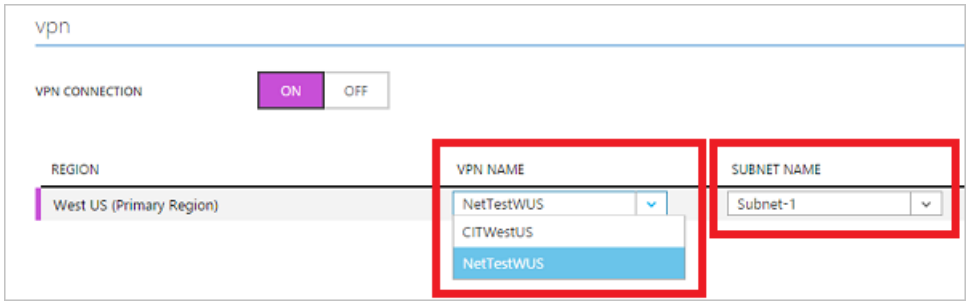
To enable VPN connectivity, open your API Management service in the Azure Classic Portal and switch to the **Configure** tab.

Under the VPN section, switch **VPN connection** to **On**.



You will now see a list of all regions where your API Management service is provisioned.

Select a VPN and subnet for every region. The list of VPNs is populated based on the virtual networks available in your Azure subscription that are setup in the region you are configuring.

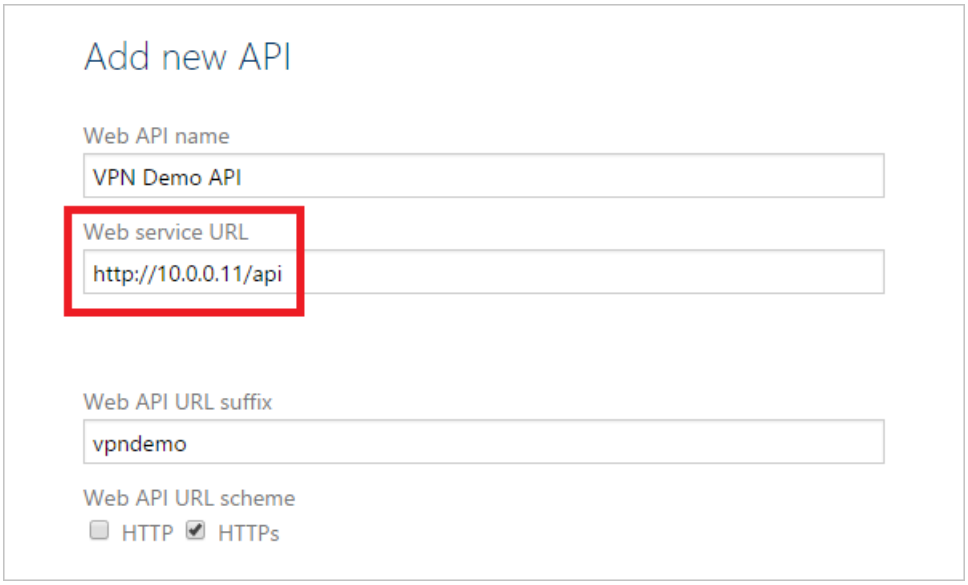


Click **Save** at the bottom of the screen. You will not be able to perform other operations on the API Management service from the Azure Classic Portal while it is updating. The service gateway will remain available and runtime calls should not be affected.

Note that the VIP address of the gateway will change each time VPN is enabled or disabled.

Connect to a web service behind VPN

After your API Management service is connected to the VPN, accessing web services within the virtual network is no different than accessing public services. Just type in the local address or the host name (if a DNS server was configured for the Azure Virtual Network) of your web service into the **Web service URL** field when creating a new API or editing an existing one.



Required ports for API Management VPN support

When an API Management service instance is hosted in a VNET, the ports in the following table are used. If these ports are blocked, the service may not function correctly. Having one or more of these ports blocked is the most common misconfiguration issue when using API Management with a VNET.

PORT(S)	DIRECTION	TRANSPORT PROTOCOL	PURPOSE	SOURCE / DESTINATION
80, 443	Inbound	TCP	Client communication to API Management	INTERNET / VIRTUAL_NETWORK
80,443	Outbound	TCP	API Management Dependency on Azure Storage and Azure Service Bus	VIRTUAL_NETWORK / INTERNET
1433	Outbound	TCP	API Management dependencies on SQL	VIRTUAL_NETWORK / INTERNET
9350, 9351, 9352, 9353, 9354	Outbound	TCP	API Management dependencies on Service Bus	VIRTUAL_NETWORK / INTERNET
5671	Outbound	AMQP	API Management dependency for Log to event Hub policy	VIRTUAL_NETWORK / INTERNET
6381, 6382, 6383	Inbound/Outbound	UDP	API Management dependencies on Redis Cache	VIRTUAL_NETWORK / VIRTUAL_NETWORK
445	Outbound	TCP	API Management Dependency on Azure File Share for GIT	VIRTUAL_NETWORK / INTERNET

Custom DNS server setup

API Management depends on a number of Azure services. When an API Management service instance is hosted in a VNET where a custom DNS server is used, it needs to be able to resolve hostnames of those Azure services.

Please follow this guidance on custom DNS setup.

Related content

- Create a virtual network with a site-to-site VPN connection using the Azure Classic Portal
- How to use the API Inspector to trace calls in Azure API Management

Custom caching in Azure API Management

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Darrel • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson

Azure API Management service has built-in support for HTTP response caching using the resource URL as the key. The key can be modified by request headers using the `vary-by` properties. This is useful for caching entire HTTP responses (aka representations), but sometimes it is useful to just cache a portion of a representation. The new cache-lookup-value and cache-store-value policies provide the ability to store and retrieve arbitrary pieces of data from within policy definitions. This ability also adds value to the previously introduced send-request policy because you can now cache responses from external services.

Architecture

API Management service uses a shared per-tenant data cache so that, as you scale up to multiple units you will still get access to the same cached data. However, when working with a multi-region deployment there are independent caches within each of the regions. Due to this, it is important to not treat the cache as a data store, where it is the only source of some piece of information. If you did, and later decided to take advantage of the multi-region deployment, then customers with users that travel may lose access to that cached data.

Fragment caching

There are certain cases where responses being returned contain some portion of data that is expensive to determine and yet remains fresh for a reasonable amount of time. As an example, consider a service built by an airline that provides information relating flight reservations, flight status, etc. If the user is a member of the airlines points program, they would also have information relating to their current status and mileage accumulated. This user-related information might be stored in a different system, but it may be desirable to include it in responses returned about flight status and reservations. This can be done using a process called fragment caching. The primary representation can be returned from the origin server using some kind of token to indicate where the user-related information is to be inserted.

Consider the following JSON response from a backend API.

```
{
  "airline" : "Air Canada",
  "flightno" : "871",
  "status" : "ontime",
  "gate" : "B40",
  "terminal" : "2A",
  "userprofile" : "$userprofile$"
}
```

And secondary resource at `/userprofile/{userid}` that looks like,

```
 { "username" : "Bob Smith", "Status" : "Gold" }
```

In order to determine the appropriate user information to include, we need to identify who the end user is. This mechanism is implementation dependent. As an example, I am using the `Subject` claim of a `JWT` token.

```
<set-variable
  name="enduserid"
  value="@(context.Request.Headers.GetValueOrDefault("Authorization","").Split(' ')[1].AsJwt()?.Subject)" />
```

We store this `enduserid` value in a context variable for later use. The next step is to determine if a previous request has already retrieved the user information and stored it in the cache. For this we use the `cache-lookup-value` policy.

```
  <cache-lookup-value
  key="@("userprofile-" + context.Variables["enduserid"])"
  variable-name="userprofile" />
```

If there is no entry in the cache that corresponds to the key value, then no `userprofile` context variable will be created. We check the success of the lookup using the `choose` control flow policy.

```
<choose>
    <when condition="@(!context.Variables.ContainsKey("userprofile"))">
        <!– If the userprofile context variable doesn't exist, make an HTTP request to retrieve it.  -->
    </when>
</choose>
```

If the `userprofile` context variable doesn't exist, then we are going to have to make an HTTP request to retrieve it.

```
<send-request
  mode="new"
  response-variable-name="userprofileresponse"
  timeout="10"
  ignore-error="true">

  <!-- Build a URL that points to the profile for the current end-user -->
  <set-url>@(new Uri(new Uri("https://apimairlineapi.azurewebsites.net/UserProfile/"),
      (string)context.Variables["enduserid"]).AbsoluteUri)
  </set-url>
  <set-method>GET</set-method>
</send-request>
```

We use the `enduserid` to construct the URL to the user profile resource. Once we have the response, we can pull the body text out of the response and store it back into a context variable.

```
<set-variable
    name="userprofile"
    value="@(((IResponse)context.Variables["userprofileresponse"]).Body.As<string>())" />
```

To avoid us having to make this HTTP request again, when the same user makes another request, we can store the user profile in the cache.

```
<cache-store-value
    key="@("userprofile-" + context.Variables["enduserid"])"
    value="@((string)context.Variables["userprofile"])" duration="100000" />
```

We store the value in the cache using the exact same key that we originally attempted to retrieve it with. The duration that we choose to store the value should be based on how often the information changes and how tolerant users are to out of date information.

It is important to realize that retrieving from the cache is still an out-of-process, network request and potentially can still add tens of milliseconds to the request. The benefits come when determining the user profile information takes significantly longer than that due to needing to do database queries or aggregate information from multiple

back-ends.

The final step in the process is to update the returned response with our user profile information.

```
<!—Update response body with user profile-->
<find-and-replace
    from='"$userprofile$"'
    to="@((string)context.Variables["userprofile"])" />
```

I chose to include the quotation marks as part of the token so that even when the replace doesn't occur, the response was still valid JSON. This was primarily to make debugging easier.

Once you combine all these steps together, the end result is a policy that looks like the following one.

```
 <policies>
    <inbound>
        <!-- How you determine user identity is application dependent -->
        <set-variable
          name="enduserid"
          value="@(context.Request.Headers.GetValueOrDefault("Authorization","").Split(' ')
[1].AsJwt()?.Subject)" />

        <!--Look for userprofile for this user in the cache -->
        <cache-lookup-value
          key="@("userprofile-" + context.Variables["enduserid"])"
          variable-name="userprofile" />

        <!-- If we don't find it in the cache, make a request for it and store it -->
        <choose>
            <when condition="@(!context.Variables.ContainsKey("userprofile"))">
                <!—Make HTTP request to get user profile -->
                <send-request
                  mode="new"
                  response-variable-name="userprofileresponse"
                  timeout="10"
                  ignore-error="true">

                   <!-- Build a URL that points to the profile for the current end-user -->
                    <set-url>@(new Uri(new Uri("https://apimairlineapi.azurewebsites.net/UserProfile/"),
(string)context.Variables["enduserid"]).AbsoluteUri)</set-url>
                     <set-method>GET</set-method>
                </send-request>

                <!—Store response body in context variable -->
                <set-variable
                  name="userprofile"
                  value="@(((IResponse)context.Variables["userprofileresponse"]).Body.As<string>())" />

                <!—Store result in cache -->
                <cache-store-value
                  key="@("userprofile-" + context.Variables["enduserid"])"
                  value="@((string)context.Variables["userprofile"])"
                  duration="100000" />
            </when>
        </choose>
        <base />
    </inbound>
    <outbound>
        <!—Update response body with user profile-->
        <find-and-replace
              from='"$userprofile$"'
              to="@((string)context.Variables["userprofile"])" />
        <base />
    </outbound>
 </policies>
```

This caching approach is primarily used in web sites where HTML is composed on the server side so that it can be rendered as a single page. However, it can also be useful in APIs where clients cannot do client side HTTP caching or it is desirable not to put that responsibility on the client.

This same kind of fragment caching can also be done on the backend web servers using a Redis caching server, however, using the API Management service to perform this work is useful when the cached fragments are coming from different back-ends than the primary responses.

Transparent versioning

It is common practice for multiple different implementation versions of an API to be supported at any one time. This is perhaps to support different environments, like dev, test, production, etc, or it may be to support older versions of the API to give time for API consumers to migrate to newer versions.

One approach to handling this instead of requiring client developers to change the URLs from `/v1/customers` to `/v2/customers` is to store in the consumer's profile data which version of the API they currently wish to use and call the appropriate backend URL. In order to determine the correct backend URL to call for a particular client, it is necessary to query some configuration data. By caching this configuration data, we can minimize the performance penalty of doing this lookup.

The first step is to determine the identifier used to configure the desired version. In this example, I chose to associate the version to the product subscription key.

```
    <set-variable name="clientid" value="@(context.Subscription.Key)" />
```

We then do a cache lookup to see if we already have retrieved the desired client version.

```
    <cache-lookup-value
    key="@("clientversion-" + context.Variables["clientid"])"
    variable-name="clientversion" />
```

Then we check to see if we did not find it in the cache.

```
<choose>
    <when condition="@(!context.Variables.ContainsKey("clientversion"))">
```

If we didn't then we go and retrieve it.

```
<send-request
    mode="new"
    response-variable-name="clientconfigureresponse"
    timeout="10"
    ignore-error="true">
            <set-url>@(new Uri(new Uri(context.Api.ServiceUrl.ToString() + "api/ClientConfig/"),
(string)context.Variables["clientid"]).AbsoluteUri)</set-url>
            <set-method>GET</set-method>
</send-request>
```

Extract the response body text from the response.

```
<set-variable
      name="clientversion"
      value="@(((IResponse)context.Variables["clientconfigureresponse"]).Body.As<string>())" />
```

Store it back in the cache for future use.

```
<cache-store-value
      key="@("clientversion-" + context.Variables["clientid"])"
      value="@((string)context.Variables["clientversion"])"
      duration="100000" />
```

And finally update the back-end URL to select the version of the service desired by the client.

```
<set-backend-service
      base-url="@(context.Api.ServiceUrl.ToString() + "api/" + (string)context.Variables["clientversion"] +
"/")" />
```

The completely policy is as follows.

```
 <inbound>
    <base />
    <set-variable name="clientid" value="@(context.Subscription.Key)" />
    <cache-lookup-value key="@("clientversion-" + context.Variables["clientid"])" variable-name="clientversion"
/>

    <!-- If we don't find it in the cache, make a request for it and store it -->
    <choose>
        <when condition="@(!context.Variables.ContainsKey("clientversion"))">
            <send-request mode="new" response-variable-name="clientconfigureresponse" timeout="10" ignore-
error="true">
                <set-url>@(new Uri(new Uri(context.Api.ServiceUrl.ToString() + "api/ClientConfig/"),
(string)context.Variables["clientid"]).AbsoluteUri)</set-url>
                <set-method>GET</set-method>
            </send-request>
            <!-- Store response body in context variable -->
            <set-variable name="clientversion"
value="@(((IResponse)context.Variables["clientconfigureresponse"]).Body.As<string>())" />
            <!-- Store result in cache -->
            <cache-store-value key="@("clientversion-" + context.Variables["clientid"])"
value="@((string)context.Variables["clientversion"])" duration="100000" />
        </when>
    </choose>
    <set-backend-service base-url="@(context.Api.ServiceUrl.ToString() + "api/" +
(string)context.Variables["clientversion"] + "/")" />
</inbound>
```

Enabling API consumers to transparently control which backend version is being accessed by clients without having to update and redeploy clients is a elegant solution that addresses many API versioning concerns.

Tenant Isolation

In larger, multi-tenant deployments some companies create separate groups of tenants on distinct deployments of backend hardware. This minimizes the number of customers who are impacted by a hardware issue on the backend. It also enables new software versions to be rolled out in stages. Ideally this backend architecture should be transparent to API consumers. This can be achieved in a similar way to transparent versioning because it is based on the same technique of manipulating the backend URL using configuration state per API key.

Instead of returning a preferred version of the API for each subscription key, you would return an identifier that relates a tenant to the assigned hardware group. That identifier can be used to construct the appropriate backend URL.

Summary

The freedom to use the Azure API management cache for storing any kind of data enables efficient access to configuration data that can affect the way an inbound request is processed. It can also be used to store data fragments that can augment responses, returned from a backend API.

Next steps

Please give us your feedback in the Disqus thread for this topic if there are other scenarios that these policies have enabled for you, or if there are scenarios you would like to achieve but do not feel are currently possible.

Monitor your APIs with Azure API Management, Event Hubs and Runscope

11/15/2016 • 13 min to read • Edit on GitHub

Contributors

Darrel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson

---

The API Management service provides many capabilities to enhance the processing of HTTP requests sent to your HTTP API. However, the existence of the requests and responses are transient. The request is made and it flows through the API Management service to your backend API. Your API processes the request and a response flows back through to the API consumer. The API Management service keeps some important statistics about the APIs for display in the Publisher portal dashboard, but beyond that, the details are gone.

By using the log-to-eventhub policy in the API Management service you can send any details from the request and response to an Azure Event Hub. There are a variety of reasons why you may want to generate events from HTTP messages being sent to your APIs. Some examples include audit trail of updates, usage analytics, exception alerting and 3rd party integrations.

This article demonstrates how to capture the entire HTTP request and response message, send it to an Event Hub and then relay that message to a third party service that provides HTTP logging and monitoring services.

Why Send From API Management Service?

It is possible to write HTTP middleware that can plug into HTTP API frameworks to capture HTTP requests and responses and feed them into logging and monitoring systems. The downside to this approach is the HTTP middleware needs to be integrated into the backend API and must match the platform of the API. If there are multiple APIs then each one must deploy the middleware. Often there are reasons why backend APIs cannot be updated.

Using the Azure API Management service to integrate with logging infrastructure provides a centralized and platform-independent solution. It is also scalable, in part due to the geo-replication capabilities of Azure API Management.

Why send to an Azure Event Hub?

It is a reasonable to ask, why create a policy that is specific to Azure Event Hubs? There are many different places where I might want to log my requests. Why not just send the requests directly to the final destination? That is an option. However, when making logging requests from an API management service, it is necessary to consider how logging messages will impact the performance of the API. Gradual increases in load can be handled by increasing available instances of system components or by taking advantage of geo-replication. However, short spikes in traffic can cause requests to be significantly delayed if requests to logging infrastructure start to slow under load.

The Azure Event Hubs is designed to ingress huge volumes of data, with capacity for dealing with a far higher number of events than the number of HTTP requests most APIs process. The Event Hub acts as a kind of sophisticated buffer between your API management service and the infrastructure that will store and process the messages. This ensures that your API performance will not suffer due to the logging infrastructure.

Once the data has been passed to an Event Hub it is persisted and will wait for Event Hub consumers to process it. The Event Hub does not care how it will be processed, it just cares about making sure the message will be successfully delivered.

Event Hubs have the ability to stream events to multiple consumer groups. This allows events to be processed by completely different systems. This enables supporting many integration scenarios without putting addition delays on the processing of the API request within the API Management service as only one event needs to be generated.

A policy to send application/http messages

An Event Hub accepts event data as a simple string. The contents of that string are completely up to you. To be able to package up an HTTP request and send it off to Event Hubs we need to format the string with the request or response information. In situations like this, if there is an existing format we can reuse, then we may not have to write our own parsing code. Initially I considered using the HAR for sending HTTP requests and responses. However, this format is optimized for storing a sequence of HTTP requests in a JSON based format. It contained a number of mandatory elements that added unnecessary complexity for the scenario of passing the HTTP message over the wire.

An alternative option was to use the `application/http` media type as described in the HTTP specification RFC 7230. This media type uses the exact same format that is used to actually send HTTP messages over the wire, but the entire message can be put in the body of another HTTP request. In our case we are just going to use the body as our message to send to Event Hubs. Conveniently, there is a parser that exists in Microsoft ASP.NET Web API 2.2 Client libraries that can parse this format and convert it into the native `HttpRequestMessage` and `HttpResponseMessage` objects.

To be able to create this message we need to take advantage of C# based Policy expressions in Azure API Management. Here is the policy which sends a HTTP request message to Azure Event Hubs.

```
<log-to-eventhub logger-id="conferencelogger" partition-id="0">
@{
    var requestLine = string.Format("{0} {1} HTTP/1.1\r\n",
                                            context.Request.Method,
                                            context.Request.Url.Path + context.Request.Url.QueryString);

    var body = context.Request.Body?.As<string>(true);
    if (body != null && body.Length > 1024)
    {
        body = body.Substring(0, 1024);
    }

    var headers = context.Request.Headers
                        .Where(h => h.Key != "Authorization" && h.Key != "Ocp-Apim-Subscription-Key")
                        .Select(h => string.Format("{0}: {1}", h.Key, String.Join(", ", h.Value)))
                        .ToArray<string>();

    var headerString = (headers.Any()) ? string.Join("\r\n", headers) + "\r\n" : string.Empty;

    return "request:"   + context.Variables["message-id"] + "\n"
                        + requestLine + headerString + "\r\n" + body;
}
</log-to-eventhub>
```

Policy declaration

There a few particular things worth mentioning about this policy expression. The log-to-eventhub policy has an attribute called logger-id which refers to the name of logger that has been created within the API Management service. The details of how to setup an Event Hub logger in the API Management service can be found in the document How to log events to Azure Event Hubs in Azure API Management. The second attribute is an optional parameter that instructs Event Hubs which partition to store the message in. Event Hubs use partitions to enable scalabilty and require a minimum of two. The ordered delivery of messages is only guaranteed within a partition. If we do not instruct Event Hub in which partition to place the message, it will use a round-robin algorithm to distribute the load. However, that may cause some of our messages to be processed out of order.

Partitions

To ensure our messages are delivered to consumers in order and take advantage of the load distribution capability of partitions, I chose to send HTTP request messages to one partition and HTTP response messages to a second partition. This will ensure an even load distribution and we can guarantee that all requests will be consumed in order and all responses will be consumed in order. It is possible for a response to be consumed before the corresponding request, but as that is not a problem as we have a different mechanism for correlating requests to responses and we know that requests always come before responses.

HTTP payloads

After building the `requestLine` we check to see if the request body should be truncated. The request body is truncated to only 1024. This could be increased, however individual Event Hub messages are limited to 256KB, so it is likely that some HTTP message bodies will not fit in a single message. When doing logging and analytics a significant amount of information can be derived from just the HTTP request line and headers. Also, many API requests only return small bodies and so the loss of information value by truncating large bodies is fairly minimal in comparison to the reduction in transfer, processing and storage costs to keep all body contents. One final note about processing the body is that we need to pass `true` to the As() method because we are reading the body contents, but was also want the backend API to be able to read the body. By passing true to this method we cause the body to be buffered so that it can be read a second time. This is important to be aware of if you have an API that does uploading of very large files or uses long polling. In these cases it would be best to avoid reading the body at all.

HTTP headers

HTTP Headers can be simply transferred over into the message format in a simple key/value pair format. We have chosen to strip out certain security sensitive fields, to avoid unnecessarily leaking credential information. It is unlikely that API keys and other credentials would be used for analytics purposes. If we wish to do analysis on the user and the particular product they are using then we could get that from the `context` object and add that to the message.

Message Metadata

When building the complete message to send to the event hub, the first line is not actually part of the `application/http` message. The first line is additional metadata consisting of whether the message is a request or response message and a message id which is used to correlate requests to responses. The message id is created by using another policy that looks like this:

```
<set-variable name="message-id" value="@(Guid.NewGuid())" />
```

We could have created the request message, stored that in a variable until the response was returned and then simply sent the request and response as a single message. However, by sending the request and response independently and using a message id to correlate the two, we get a bit more flexibility in the message size, the ability to take advantage of multiple partitions whilst maintaining message order and the request will appear in our logging dashboard sooner. There also may be some scenarios where a valid response is never sent to the event hub, possibly due to a fatal request error in the API Management service, but we still will have a record of the request.

The policy to send the response HTTP message looks very similar to the request and so the complete policy configuration looks like this:

```
  <policies>
      <inbound>
          <set-variable name="message-id" value="@(Guid.NewGuid())" />
          <log-to-eventhub logger-id="conferencelogger" partition-id="0">
          @{
              var requestLine = string.Format("{0} {1} HTTP/1.1\r\n",
                                                      context.Request.Method,
                                                      context.Request.Url.Path +
context.Request.Url.QueryString);

              var body = context.Request.Body?.As<string>(true);
              if (body != null && body.Length > 1024)
              {
                  body = body.Substring(0, 1024);
              }

              var headers = context.Request.Headers
                                   .Where(h => h.Key != "Authorization" && h.Key != "Ocp-Apim-Subscription-Key")
                                   .Select(h => string.Format("{0}: {1}", h.Key, String.Join(", ", h.Value)))
                                   .ToArray<string>();

              var headerString = (headers.Any()) ? string.Join("\r\n", headers) + "\r\n" : string.Empty;

              return "request:"   + context.Variables["message-id"] + "\n"
                                  + requestLine + headerString + "\r\n" + body;
          }
      </log-to-eventhub>
      </inbound>
      <backend>
          <forward-request follow-redirects="true" />
      </backend>
      <outbound>
          <log-to-eventhub logger-id="conferencelogger" partition-id="1">
          @{
              var statusLine = string.Format("HTTP/1.1 {0} {1}\r\n",
                                                context.Response.StatusCode,
                                                context.Response.StatusReason);

              var body = context.Response.Body?.As<string>(true);
              if (body != null && body.Length > 1024)
              {
                  body = body.Substring(0, 1024);
              }

              var headers = context.Response.Headers
                                           .Select(h => string.Format("{0}: {1}", h.Key, String.Join(", ",
h.Value)))
                                           .ToArray<string>();

              var headerString = (headers.Any()) ? string.Join("\r\n", headers) + "\r\n" : string.Empty;

              return "response:"  + context.Variables["message-id"] + "\n"
                                  + statusLine + headerString + "\r\n" + body;
          }
      </log-to-eventhub>
      </outbound>
  </policies>
```

The `set-variable` policy creates a value that is accessible by both the `log-to-eventhub` policy in the `<inbound>` section and the `<outbound>` section.

Receiving events from Event Hubs

Events from Azure Event Hub are received using the AMQP protocol. The Microsoft Service Bus team have made client libraries available to make the consuming events easier. There are two different approaches supported, one is

being a *Direct Consumer* and the other is using the `EventProcessorHost` class. Examples of these two approaches can be found in the Event Hubs Programming Guide. The short version of the differences is, `Direct Consumer` gives you complete control and the `EventProcessorHost` does some of the plumbing work for you but makes certain assumptions about how you will process those events.

EventProcessorHost

In this sample, we will use the `EventProcessorHost` for simplicity, however it may not the best choice for this particular scenario. `EventProcessorHost` does the hard work of making sure you don't have to worry about threading issues within a particular event processor class. However, in our scenario, we are simply converting the message to another format and passing it along to another service using an async method. There is no need for updating shared state and therefore no risk of threading issues. For most scenarios, `EventProcessorHost` is probably the best choice and it is certainly the easier option.

IEventProcessor

The central concept when using `EventProcessorHost` is to create a an implementation of the `IEventProcessor` interface which contains the method `ProcessEventAsync`. The essence of that method is shown here:

async Task IEventProcessor.ProcessEventsAsync(PartitionContext context, IEnumerable messages) {

```
       foreach (EventData eventData in messages)
       {
           _Logger.LogInfo(string.Format("Event received from partition: {0} - {1}",
context.Lease.PartitionId,eventData.PartitionKey));

           try
           {
               var httpMessage = HttpMessage.Parse(eventData.GetBodyStream());
               await _MessageContentProcessor.ProcessHttpMessage(httpMessage);
           }
           catch (Exception ex)
           {
               _Logger.LogError(ex.Message);
           }
       }
        ... checkpointing code snipped ...
    }
```

A list of EventData objects are passed into the method and we iterate over that list. The bytes of each method are parsed into a HttpMessage object and that object is passed to an instance of IHttpMessageProcessor.

HttpMessage

The `HttpMessage` instance contains three pieces of data:

```
public class HttpMessage
 {
     public Guid MessageId { get; set; }
     public bool IsRequest { get; set; }
     public HttpRequestMessage HttpRequestMessage { get; set; }
     public HttpResponseMessage HttpResponseMessage { get; set; }

   ... parsing code snipped ...

}
```

The `HttpMessage` instance contains a `MessageId` GUID that allows us to connect the HTTP request to the corresponding HTTP response and a boolean value that identifies if the object contains an instance of a HttpRequestMessage and HttpResponseMessage. By using the built in HTTP classes from `System.Net.Http`, I was able to take advantage of the `application/http` parsing code that is included in `System.Net.Http.Formatting`.

IHttpMessageProcessor

The `HttpMessage` instance is then forwarded to implementation of `IHttpMessageProcessor` which is an interface I created to decouple the receiving and interpretation of the event from Azure Event Hub and the actual processing of it.

Forwarding the HTTP message

For this sample, I decided it would be interesting to push the HTTP Request over to Runscope. Runscope is a cloud based service that specializes in HTTP debugging, logging and monitoring. They have a free tier, so it is easy to try and it allows us to see the HTTP requests in real-time flowing through our API Management service.

The `IHttpMessageProcessor` implementation looks like this,

```
  public class RunscopeHttpMessageProcessor : IHttpMessageProcessor
   {
       private HttpClient _HttpClient;
       private ILogger _Logger;
       private string _BucketKey;
       public RunscopeHttpMessageProcessor(HttpClient httpClient, ILogger logger)
       {
           _HttpClient = httpClient;
           var key = Environment.GetEnvironmentVariable("APIMEVENTS-RUNSCOPE-KEY",
EnvironmentVariableTarget.User);
           _HttpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("bearer", key);
           _HttpClient.BaseAddress = new Uri("https://api.runscope.com");
           _BucketKey = Environment.GetEnvironmentVariable("APIMEVENTS-RUNSCOPE-BUCKET",
EnvironmentVariableTarget.User);
           _Logger = logger;
       }

       public async Task ProcessHttpMessage(HttpMessage message)
       {
           var runscopeMessage = new RunscopeMessage()
           {
               UniqueIdentifier = message.MessageId
           };

           if (message.IsRequest)
           {
               _Logger.LogInfo("Sending HTTP request " + message.MessageId.ToString());
               runscopeMessage.Request = await RunscopeRequest.CreateFromAsync(message.HttpRequestMessage);
           }
           else
           {
               _Logger.LogInfo("Sending HTTP response " + message.MessageId.ToString());
               runscopeMessage.Response = await RunscopeResponse.CreateFromAsync(message.HttpResponseMessage);
           }

           var messagesLink = new MessagesLink() { Method = HttpMethod.Post };
           messagesLink.BucketKey = _BucketKey;
           messagesLink.RunscopeMessage = runscopeMessage;
           var runscopeResponse = await _HttpClient.SendAsync(messagesLink.CreateRequest());
           _Logger.LogDebug("Request sent to Runscope");
       }
   }
```

I was able to take advantage of an existing client library for Runscope that makes it easy to push `HttpRequestMessage` and `HttpResponseMessage` instances up into their service. In order to access the Runscope API you will need an account and an API Key. Instructions for getting an API key can be found in the Creating Applications to Access Runscope API screencast.

Complete sample

The source code and tests for the sample are on Github. You will need an API Management Service, a connected

Event Hub, and a Storage Account to run the sample for yourself.

The sample is just a simple Console application that listens for events coming from Event Hub, converts them into a `HttpRequestMessage` and `HttpResponseMessage` objects and then forwards them on to the Runscope API.

In the following animated image, you can see a request being made to an API in the Developer Portal, the Console application showing the message being received, processed and forwarded and then the request and response showing up in the Runscope Traffic inspector.

Summary

Azure API Management service provides an ideal place to capture the HTTP traffic travelling to and from your APIs. Azure Event Hubs is a highly scalable, low cost solution for capturing that traffic and feeding it into secondary processing systems for logging, monitoring and other sophisticated analytics. Connecting to 3rd party traffic monitoring systems like Runscope is a simple as a few dozen lines of code.

Next steps

- Learn more about Azure Event Hubs
  - Get started with Azure Event Hubs
  - Receive messages with EventProcessorHost
  - Event Hubs programming guide
- Learn more about API Management and Event Hubs integration
  - How to log events to Azure Event Hubs in Azure API Management
  - Logger entity reference
  - log-to-eventhub policy reference

Using external services from the Azure API Management service

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Darrel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson • Andy Pasic

---

The policies available in Azure API Management service can do a wide range of useful work based purely on the incoming request, the outgoing response and basic configuration information. However, being able to interact with external services from API Management policies opens up many more opportunities.

We have previously seen how we can interact with the Azure Event Hub service for logging, monitoring and analytics. In this article we will demonstrate policies that allow you to interact with any external HTTP based service. These policies can be used for triggering remote events or for retrieving information that will be used to manipulate the original request and response in some way.

Send-One-Way-Request

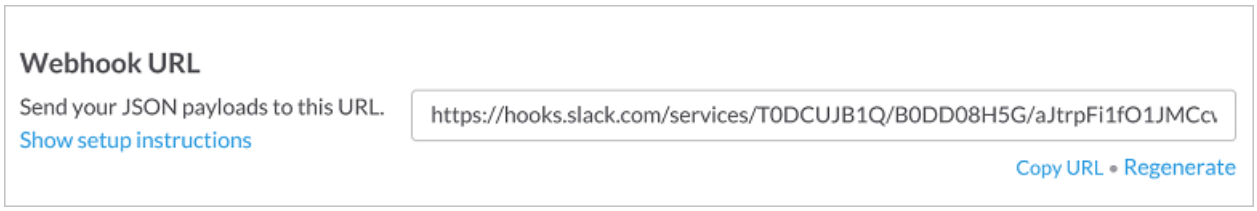
Possibly the simplest external interaction is the fire-and-forget style of request that allows an external service to be notified of some kind of important event. We can use the control flow policy `choose` to detect any kind of condition that we are interested in and then, if the condition is satisfied, we can make an external HTTP request using the send-one-way-request policy. This could be a request to a messaging system like Hipchat or Slack, or a mail API like SendGrid or MailChimp, or for critical support incidents something like PagerDuty. All of these messaging systems have simple HTTP APIs that we can easily invoke.

Alerting with Slack

The following example demonstrates how to send a message to a Slack chat room if the HTTP response status code is greater than or equal to 500. A 500 range error indicates a problem with our backend API that the client of our API cannot resolve themselves. It usually requires some kind of intervention on our part.

```
<choose>
    <when condition="@(context.Response.StatusCode >= 500)">
      <send-one-way-request mode="new">
        <set-url>https://hooks.slack.com/services/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</set-url>
        <set-method>POST</set-method>
        <set-body>@{
                return new JObject(
                        new JProperty("username","APIM Alert"),
                        new JProperty("icon_emoji", ":ghost:"),
                        new JProperty("text", String.Format("{0} {1}\nHost: {2}\n{3} {4}\n User: {5}",
                                                context.Request.Method,
                                                context.Request.Url.Path + context.Request.Url.QueryString,
                                                context.Request.Url.Host,
                                                context.Response.StatusCode,
                                                context.Response.StatusReason,
                                                context.User.Email
                                                ))
                        ).ToString();
            }</set-body>
      </send-one-way-request>
    </when>
</choose>
```

Slack has the notion of inbound web hooks. When configuring an inbound web hook, Slack generates a special URL which allows you to do a simple POST request and to pass a message into the Slack channel. The JSON body that we create is based on a format defined by Slack.



Is fire and forget good enough?

There are certain tradeoffs when using a fire-and-forget style of request. If for some reason, the request fails, then the failure will not be reported. In this particular situation, the complexity of having a secondary failure reporting system and the additional performance cost of waiting for the response is not warranted. For scenarios where it is essential to check the response, then the send-request policy is a better option.

Send-Request

The `send-request` policy enables using an external service to perform complex processing functions and return data to the API management service that can be used for further policy processing.

Authorizing reference tokens

A major function of API Management is protecting backend resources. If the authorization server used by your API creates JWT tokens as part of its OAuth2 flow, as Azure Active Directory does, then you can use the `validate-jwt` policy to verify the validity of the token. However, some authorization servers create what are called reference tokens that cannot be verified without making a call back to the authorization server.

Standardized introspection

In the past there has been no standardized way of verifying a reference token with an authorization server. However a recently proposed standard RFC 7662 was published by the IETF that defines how a resource server can verify the validity of a token.

Extracting the token

The first step is to extract the token from the Authorization header. The header value should be formatted with the `Bearer` authorization scheme, a single space and then the authorization token as per RFC 6750. Unfortunately there are cases where the authorization scheme is omitted. To account for this when parsing, we split the header value on a space and select the last string from the returned array of strings. This provides a workaround for badly formatted authorization headers.

```
<set-variable name="token" value="@(context.Request.Headers.GetValueOrDefault("Authorization","scheme
param").Split(' ').Last())" />
```

Making the validation request

Once we have the authorization token, we can make the request to validate the token. RFC 7662 calls this process introspection and requires that you `POST` a HTML form to the introspection resource. The HTML form must at least contain a key/value pair with the key `token`. This request to the authorization server must also be authenticated to ensure that malicious clients cannot go trawling for valid tokens.

```
<send-request mode="new" response-variable-name="tokenstate" timeout="20" ignore-error="true">
  <set-url>https://microsoft-apiappec990ad4c76641c6aea22f566efc5a4e.azurewebsites.net/introspection</set-url>
  <set-method>POST</set-method>
  <set-header name="Authorization" exists-action="override">
    <value>basic dXNlcm5hbWU6cGFzc3dvcmQ=</value>
  </set-header>
  <set-header name="Content-Type" exists-action="override">
    <value>application/x-www-form-urlencoded</value>
  </set-header>
  <set-body>@($"token={(string)context.Variables["token"]}")</set-body>
</send-request>
```

Checking the response

The `response-variable-name` attribute is used to give access the returned response. The name defined in this property can be used as a key into the `context.Variables` dictionary to access the `IResponse` object.

From the response object we can retrieve the body and RFC 7622 tells us that the response must be a JSON object and must contain at least a property called `active` that is a boolean value. When `active` is true then the token is considered valid.

Reporting failure

We use a `<choose>` policy to detect if the token is invalid and if so, return a 401 response.

```
<choose>
  <when condition="@((bool)((IResponse)context.Variables["tokenstate"]).Body.As<JObject>()["active"] == false)">
    <return-response response-variable-name="existing response variable">
      <set-status code="401" reason="Unauthorized" />
      <set-header name="WWW-Authenticate" exists-action="override">
        <value>Bearer error="invalid_token"</value>
      </set-header>
    </return-response>
  </when>
</choose>
```

As per RFC 6750 which describes how `bearer` tokens should be used, we also return a `WWW-Authenticate` header with the 401 response. The WWW-Authenticate is intended to instruct a client on how to construct a properly authorized request. Due to the wide variety of approaches possible with the OAuth2 framework, it is difficult to communicate all the needed information. Fortunately there are efforts underway to help clients discover how to properly authorize requests to a resource server.

Final solution

Putting all the pieces together, we get the following policy:

```
<inbound>
  <!-- Extract Token from Authorization header parameter -->
  <set-variable name="token" value="@(context.Request.Headers.GetValueOrDefault("Authorization","scheme
param").Split(' ').Last())" />

  <!-- Send request to Token Server to validate token (see RFC 7662) -->
  <send-request mode="new" response-variable-name="tokenstate" timeout="20" ignore-error="true">
    <set-url>https://microsoft-apiappec990ad4c76641c6aea22f566efc5a4e.azurewebsites.net/introspection</set-url>
    <set-method>POST</set-method>
    <set-header name="Authorization" exists-action="override">
      <value>basic dXNlcm5hbWU6cGFzc3dvcmQ=</value>
    </set-header>
    <set-header name="Content-Type" exists-action="override">
      <value>application/x-www-form-urlencoded</value>
    </set-header>
    <set-body>@($"token={(string)context.Variables["token"]}")</set-body>
  </send-request>

  <choose>
          <!-- Check active property in response -->
          <when condition="@((bool)((IResponse)context.Variables["tokenstate"]).Body.As<JObject>()["active"] ==
false)">
              <!-- Return 401 Unauthorized with http-problem payload -->
              <return-response response-variable-name="existing response variable">
                  <set-status code="401" reason="Unauthorized" />
                  <set-header name="WWW-Authenticate" exists-action="override">
                      <value>Bearer error="invalid_token"</value>
                  </set-header>
              </return-response>
          </when>
      </choose>
  <base />
</inbound>
```

This is only one of many examples of how the `send-request` policy can be used to integrate useful external services into the process of requests and responses flowing through the API Management service.

Response Composition

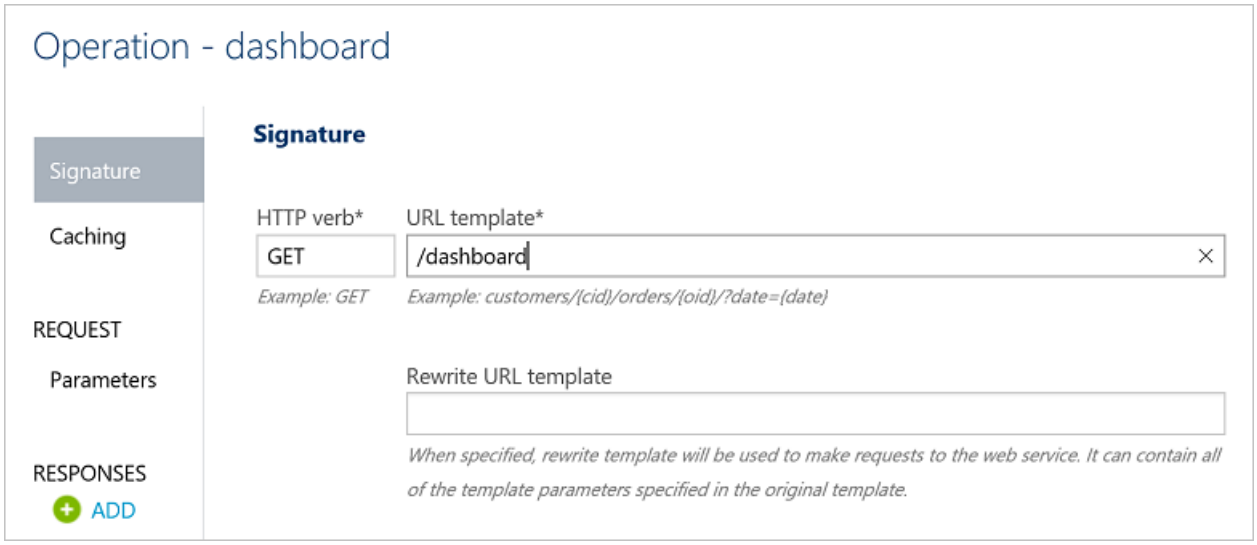
The `send-request` policy can be used for enhancing a primary request to a backend system, as we saw in the previous example, or it can be used as a complete replace for of the backend call. Using this technique we can easily create composite resources that are aggregated from multiple different systems.

Building a dashboard

Sometimes you want to be able to expose information that exists in multiple backend systems, for example, to drive a dashboard. The KPIs come from all different back-ends, but you would prefer not to provide direct access to them and it would be nice if all the information could be retrieved in a single request. Perhaps some of the backend information needs some slicing and dicing and a little sanitizing first! Being able to cache that composite resource would be a useful to reduce the backend load as you know users have a habit of hammering the F5 key in order to see if their underperforming metrics might change.

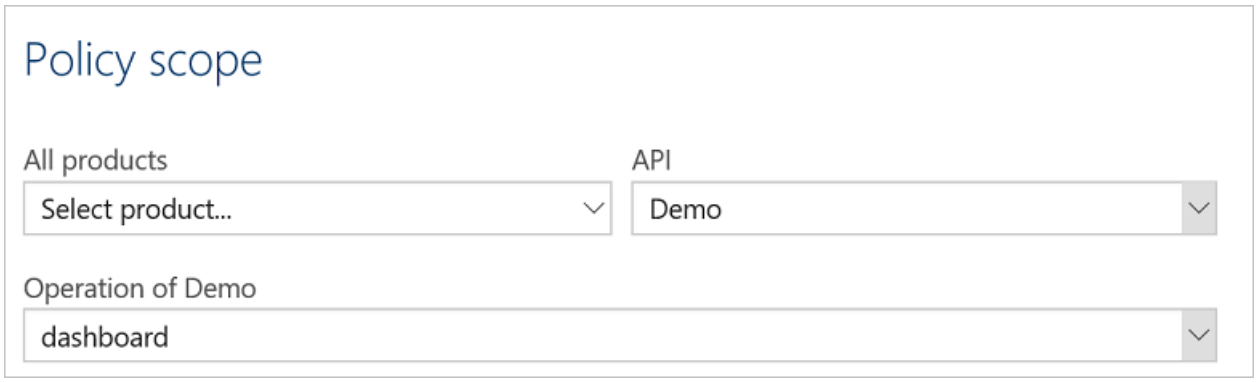
Faking the resource

The first step to building our dashboard resource is to configure a new operation in the API Management publisher portal. This will be a placeholder operation used to configure our composition policy to build our dynamic resource.



Making the requests

Once the `dashboard` operation has been created we can configure a policy specifically for that operation.



The first step is to extract any query parameters from the incoming request, so that we can forward them to our backend. In this example our dashboard is showing information based on a period of time an therefore has a `fromDate` and `toDate` parameter. We can use the `set-variable` policy to extract the information from the request URL.

```
<set-variable name="fromDate" value="@(context.Request.Url.Query["fromDate"].Last())">
<set-variable name="toDate" value="@(context.Request.Url.Query["toDate"].Last())">
```

Once we have this information we can make requests to all the backend systems. Each request constructs a new URL with the parameter information and calls its respective server and stores the response in a context variable.

```
<send-request mode="new" response-variable-name="revenuedata" timeout="20" ignore-error="true">
  <set-url>@($"https://accounting.acme.com/salesdata?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
  <set-method>GET</set-method>
</send-request>

<send-request mode="new" response-variable-name="materialdata" timeout="20" ignore-error="true">
  <set-url>@($"https://inventory.acme.com/materiallevels?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
  <set-method>GET</set-method>
</send-request>

<send-request mode="new" response-variable-name="throughputdata" timeout="20" ignore-error="true">
<set-url>@($"https://production.acme.com/throughput?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
  <set-method>GET</set-method>
</send-request>

<send-request mode="new" response-variable-name="accidentdata" timeout="20" ignore-error="true">
<set-url>@($"https://production.acme.com/throughput?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
  <set-method>GET</set-method>
</send-request>
```

These requests will execute in sequence, which is not ideal. In an upcoming release we will be introducing a new policy called `wait` that will enable all of these requests to execute in parallel.

Responding

To construct the composite response we can use the return-response policy. The `set-body` element can use an expression to construct a new `JObject` with all the component representations embedded as properties.

```
<return-response response-variable-name="existing response variable">
  <set-status code="200" reason="OK" />
  <set-header name="Content-Type" exists-action="override">
    <value>application/json</value>
  </set-header>
  <set-body>
    @(new JObject(new JProperty("revenuedata",((IResponse)context.Variables["revenuedata"]).Body.As<JObject>()),
                  new JProperty("materialdata",((IResponse)context.Variables["materialdata"]).Body.As<JObject>
()),
                  new JProperty("throughputdata",
((IResponse)context.Variables["throughputdata"]).Body.As<JObject>()),
                  new JProperty("accidentdata",((IResponse)context.Variables["accidentdata"]).Body.As<JObject>())
                  ).ToString())
  </set-body>
</return-response>
```

The complete policy looks as follows:

```
<policies>
    <inbound>

  <set-variable name="fromDate" value="@(context.Request.Url.Query["fromDate"].Last())">
  <set-variable name="toDate" value="@(context.Request.Url.Query["toDate"].Last())">

    <send-request mode="new" response-variable-name="revenuedata" timeout="20" ignore-error="true">
      <set-url>@($"https://accounting.acme.com/salesdata?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
      <set-method>GET</set-method>
    </send-request>

    <send-request mode="new" response-variable-name="materialdata" timeout="20" ignore-error="true">
      <set-url>@($"https://inventory.acme.com/materiallevels?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
      <set-method>GET</set-method>
    </send-request>

    <send-request mode="new" response-variable-name="throughputdata" timeout="20" ignore-error="true">
    <set-url>@($"https://production.acme.com/throughput?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
      <set-method>GET</set-method>
    </send-request>

    <send-request mode="new" response-variable-name="accidentdata" timeout="20" ignore-error="true">
    <set-url>@($"https://production.acme.com/throughput?from={(string)context.Variables["fromDate"]}&to=
{(string)context.Variables["fromDate"]}")"</set-url>
      <set-method>GET</set-method>
    </send-request>

    <return-response response-variable-name="existing response variable">
      <set-status code="200" reason="OK" />
      <set-header name="Content-Type" exists-action="override">
        <value>application/json</value>
      </set-header>
      <set-body>
        @(new JObject(new JProperty("revenuedata",((IResponse)context.Variables["revenuedata"]).Body.As<JObject>
()),
                      new JProperty("materialdata",
((IResponse)context.Variables["materialdata"]).Body.As<JObject>()),
                      new JProperty("throughputdata",
((IResponse)context.Variables["throughputdata"]).Body.As<JObject>()),
                      new JProperty("accidentdata",
((IResponse)context.Variables["accidentdata"]).Body.As<JObject>())
                      ).ToString())
      </set-body>
    </return-response>
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
    </outbound>
</policies>
```

In the configuration of the placeholder operation we can configure the dashboard resource to be cached for at least an hour because we understand the nature of the data means that even if it is an hour out of date, it will still be sufficiently effective to convey valuable information to the users.

Summary

Azure API Management service provides flexible policies that can be selectively applied to HTTP traffic and enables composition of backend services. Whether you want to enhance your API gateway with alerting functions, verification, validation capabilities or create new composite resources based on multiple backend services, the

`send-request` and related policies open a world of possibilities.

Watch a video overview of these policies

For more information on the send-one-way-request, send-request, and return-response policies covered in this article, please watch the following video.

Azure API Management Policy Reference

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Vlad Vinogradsky • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson • v-aljenk • Jamie Strusz

---

This section provides an index for the policies in the API Management policy reference. For information on adding and configuring policies, see Policies in API Management.

Policy expressions can be used as attribute values or text values in any of the API Management policies, unless the policy specifies otherwise. Some policies such as the Control flow and Set variable policies are based on policy expressions. For more information, see Advanced policies and Policy expressions

Policy reference index

- Access restriction policies
  - Check HTTP header - Enforces existence and/or value of a HTTP Header.
  - Limit call rate by subscription - Prevents API usage spikes by limiting call rate, on a per subscription basis.
  - Limit call rate by key - Prevents API usage spikes by limiting call rate, on a per key basis.
  - Restrict caller IPs - Filters (allows/denies) calls from specific IP addresses and/or address ranges.
  - Set usage quota by subscription - Allows you to enforce a renewable or lifetime call volume and/or bandwidth quota, on a per subscription basis.
  - Set usage quota by key - Allows you to enforce a renewable or lifetime call volume and/or bandwidth quota, on a per key basis.
  - Validate JWT - Enforces existence and validity of a JWT extracted from either a specified HTTP Header or a specified query parameter.
- Advanced policies
  - Control flow - Conditionally applies policy statements based on the results of the evaluation of Boolean expressions.
  - Forward request - Forwards the request to the backend service.
  - Log to Event Hub - Sends messages in the specified format to a message target defined by a Logger entity.
  - Retry - Retries execution of the enclosed policy statements, if and until the condition is met. Execution will repeat at the specified time intervals and up to the specified retry count.
  - Return response - Aborts pipeline execution and returns the specified response directly to the caller.
  - Send one way request - Sends a request to the specified URL without waiting for a response.
  - Send request - Sends a request to the specified URL.
  - Set request method - Allows you to change the HTTP method for a request.
  - Set status - Changes the HTTP status code to the specified value.
  - Set variable - Persist a value in a named context variable for later access.
  - Trace - Adds a string into the API Inspector output.
  - Wait - Waits for enclosed Send request, Get value from cache, or Control flow policies to complete before proceeding.
- Authentication policies
  - Authenticate with Basic - Authenticate with a backend service using Basic authentication.

    - Authenticate with client certificate - Authenticate with a backend service using client certificates.
- Caching policies
    - Get from cache - Perform cache look up and return a valid cached response when available.
    - Store to cache - Caches response according to the specified cache control configuration.
    - Get value from cache - Retrieve a cached item by key.
    - Store value in cache - Store an item in the cache by key.
    - Remove value from cache - Remove an item in the cache by key.
- Cross domain policies
    - Allow cross-domain calls - Makes the API accessible from Adobe Flash and Microsoft Silverlight browser-based clients.
    - CORS - Adds cross-origin resource sharing (CORS) support to an operation or an API to allow cross-domain calls from browser-based clients.
    - JSONP - Adds JSON with padding (JSONP) support to an operation or an API to allow cross-domain calls from JavaScript browser-based clients.
- Transformation policies
    - Convert JSON to XML - Converts request or response body from JSON to XML.
    - Convert XML to JSON - Converts request or response body from XML to JSON.
    - Find and replace string in body - Finds a request or response substring and replaces it with a different substring.
    - Mask URLs in content - Re-writes (masks) links in the response body so that they point to the equivalent link via the gateway.
    - Set backend service - Changes the backend service for an incoming request.
    - Set body - Sets the message body for incoming and outgoing requests.
    - Set HTTP header - Assigns a value to an existing response and/or request header or adds a new response and/or request header.
    - Set query string parameter - Adds, replaces value of, or deletes request query string parameter.
    - Rewrite URL - Converts a request URL from its public form to the form expected by the web service.

Next steps

For more information on policy expressions, see the following video.

Policies in Azure API Management

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • v-aljenk • Jamie Strusz

---

In Azure API Management, 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. Popular Statements include format conversion from XML to JSON and call rate limiting to restrict the amount of incoming calls from a developer. Many more policies are available out of the box.

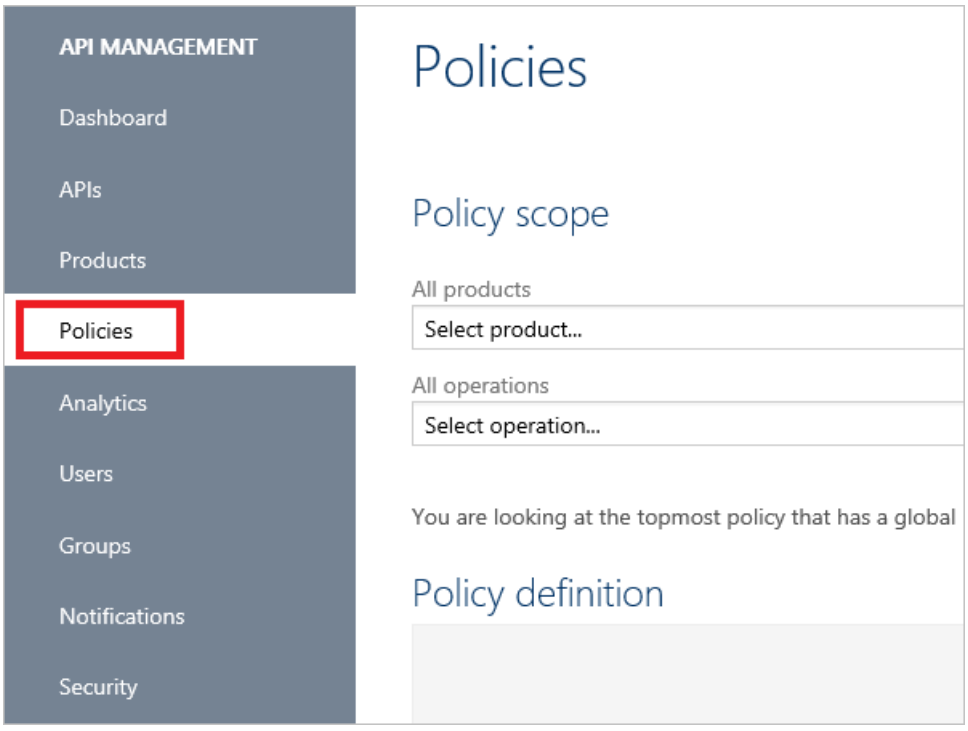
See the Policy Reference for a full list of policy statements and their settings.

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.

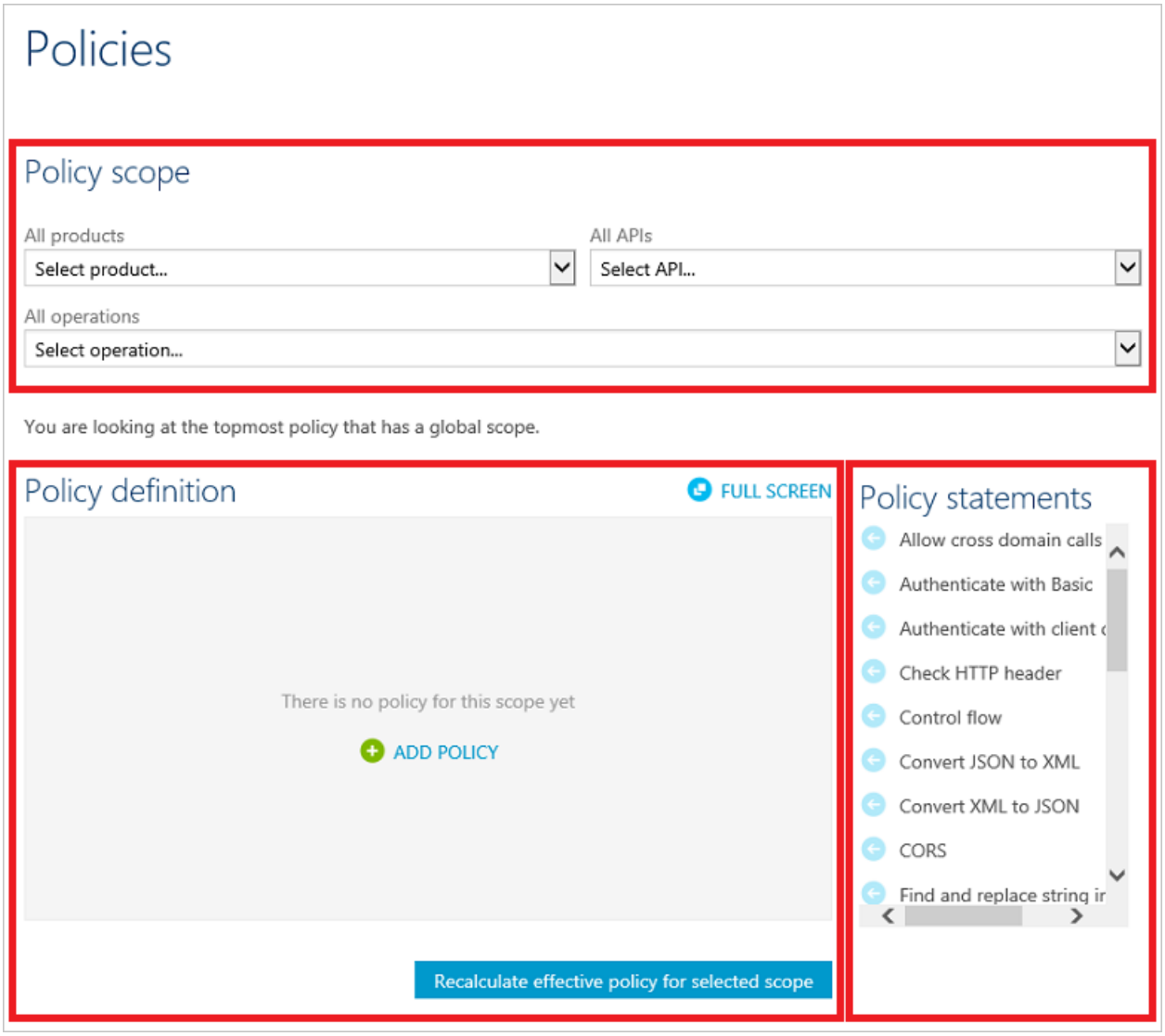
Policy expressions can be used as attribute values or text values in any of the API Management policies, unless the policy specifies otherwise. Some policies such as the Control flow and Set variable policies are based on policy expressions. For more information, see Advanced policies and Policy expressions.

How to configure policies

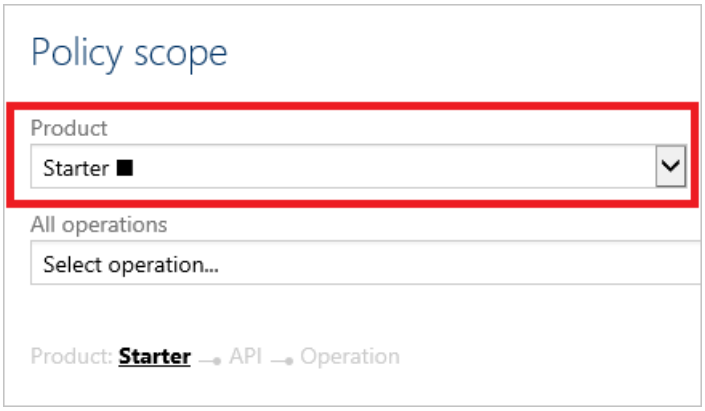
Policies can be configured globally or at the scope of a Product, API or Operation. To configure a policy, navigate to the Policies editor in the publisher portal.



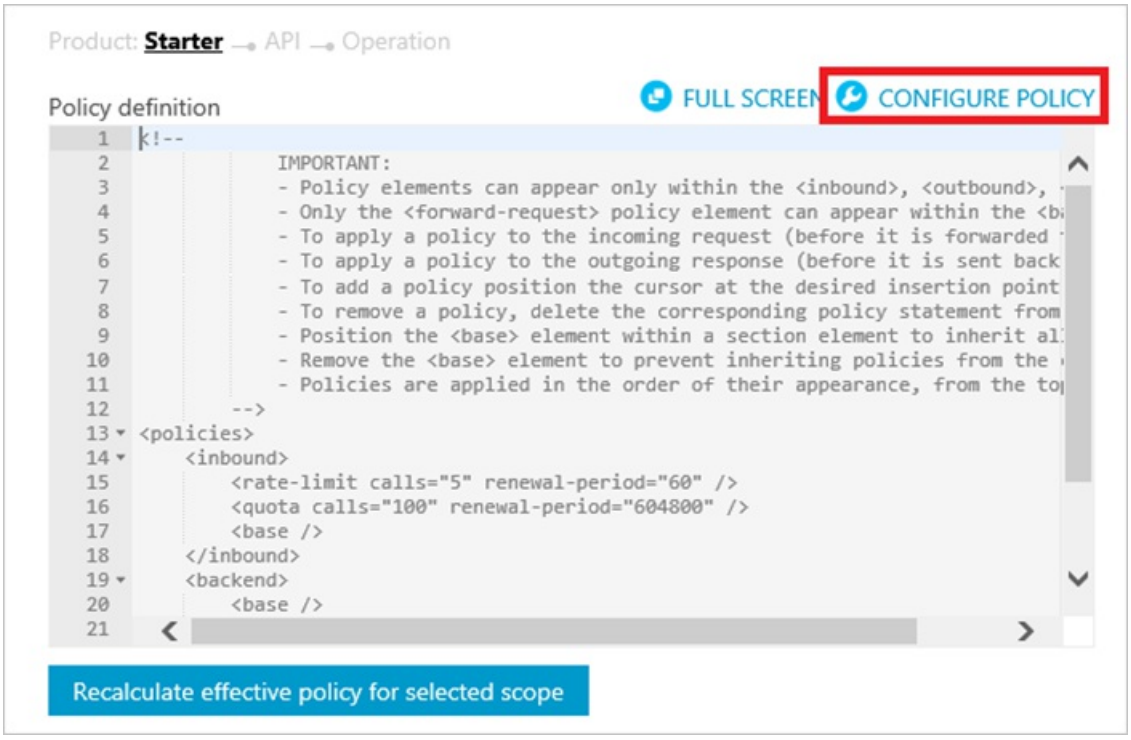
The policies editor consists of three main sections: the policy scope (top), the policy definition where policies are edited (left) and the statements list (right):



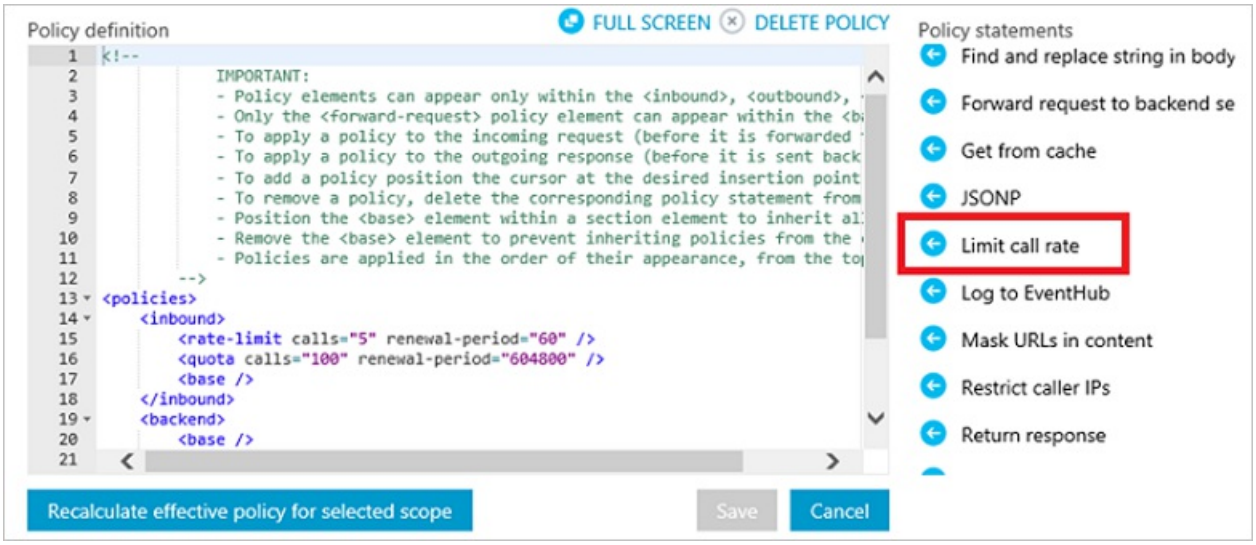
To begin configuring a policy you must first select the scope at which the policy should apply. In the screenshot below the **Starter** product is selected. Note that the square symbol next to the policy name indicates that a policy is already applied at this level.



Since a policy has already been applied, the configuration is shown in the definition view.



The policy is displayed read-only at first. In order to edit the definition click the **Configure Policy** action.



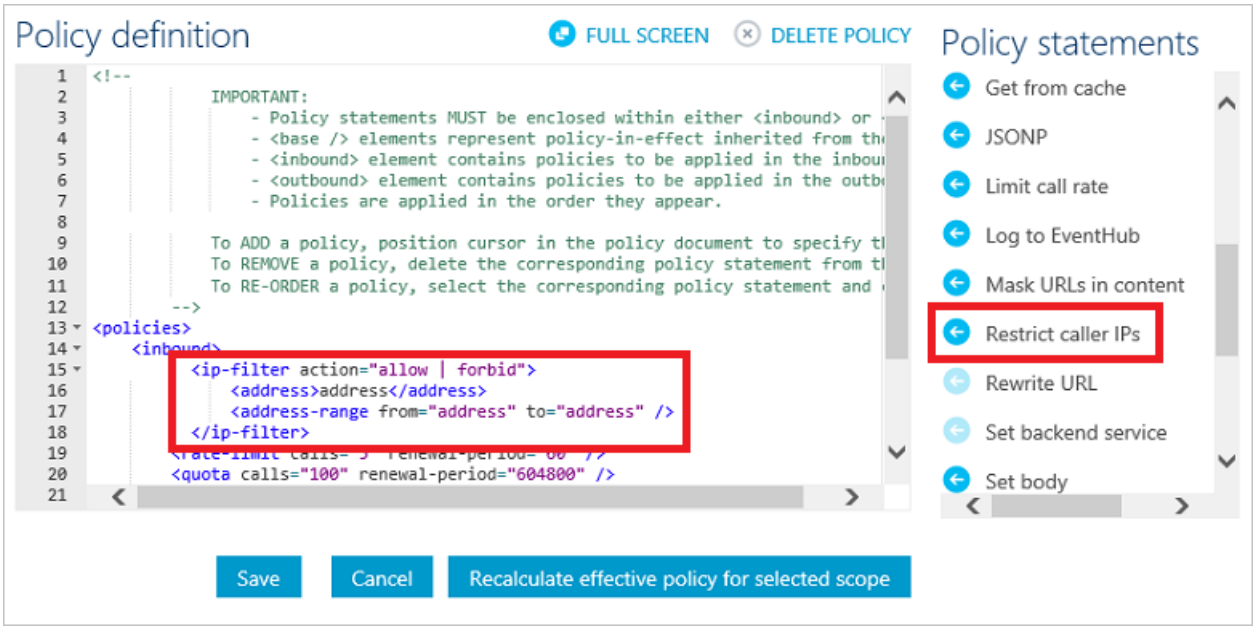
The policy definition is a simple XML document that describes a sequence of inbound and outbound statements. The XML can be edited directly in the definition window. A list of statements is provided to the right and statements applicable to the current scope are enabled and highlighted; as demonstrated by the **Limit Call Rate** statement in the screenshot above.

Clicking an enabled statement will add the appropriate XML at the location of the cursor in the definition view.

> **NOTE**
> If the policy that you want to add is not enabled, ensure that you are in the correct scope for that policy. Each policy statement is designed for use in certain scopes and policy sections. To review the policy sections and scopes for a policy, check the **Usage** section for that policy in the Policy Reference.

A full list of policy statements and their settings are available in the Policy Reference.

For example, to add a new statement to restrict incoming requests to specified IP addresses, place the cursor just inside the content of the `inbound` XML element and click the **Restrict caller IPs** statement.

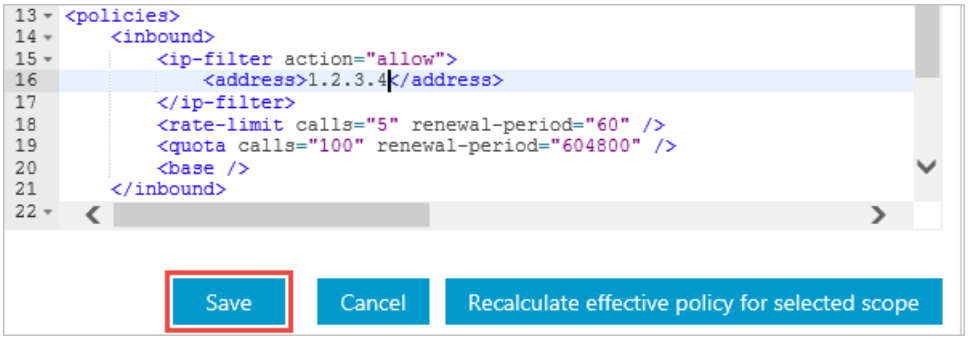


This will add an XML snippet to the `inbound` element that provides guidance on how to configure the statement.

```
<ip-filter action="allow | forbid">
    <address>address</address>
    <address-range from="address" to="address"/>
</ip-filter>
```

To limit inbound requests and accept only those from an IP address of 1.2.3.4 modify the XML as follows:

```
<ip-filter action="allow">
    <address>1.2.3.4</address>
</ip-filter>
```



When complete configuring the statements for the policy, click **Save** and the changes will be propagated to the API Management gateway immediately.

Understanding policy configuration

A policy is a series of statements that execute in order for a request and a response. The configuration is divided appropriately into `inbound`, `backend`, `outbound`, and `on-error` sections as shown in the following configuration.

```
<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>
```

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. By placing policy statements in the `on-error` section you can review the error by using the `context.LastError` property, inspect and customize the error response using the `set-body` policy, and configure what happens if an error occurs. There are error codes for built-in steps and for errors that may occur during the processing of policy statements. For more information, see Error handling in API Management policies.

Since policies can be specified at different levels (global, product, api and operation) the configuration provides a way for you to specify the order in which the policy definition's statements execute with respect to the parent policy.

Policy scopes are evaluated in the following order.

1. Global scope
2. Product scope
3. API scope
4. Operation scope

The statements within them are evaluated according to the placement of the `base` element, if it is present.

For example, if you have a policy at the global level and a policy configured for an API, then whenever that particular API is used both policies will be applied. API Management allows for deterministic ordering of combined policy statements via the base element.

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

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

If the same policy appears twice in the policy statement, the most recently evaluated policy is applied. You can use this to override policies that are defined at a higher scope. To see the policies in the current scope in the policy editor, click **Recalculate effective policy for selected scope**.

Note that global policy has no parent policy and using the `<base>` element in it has no effect.

Next steps

Check out following video on policy expressions.

How to use properties in Azure API Management policies

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

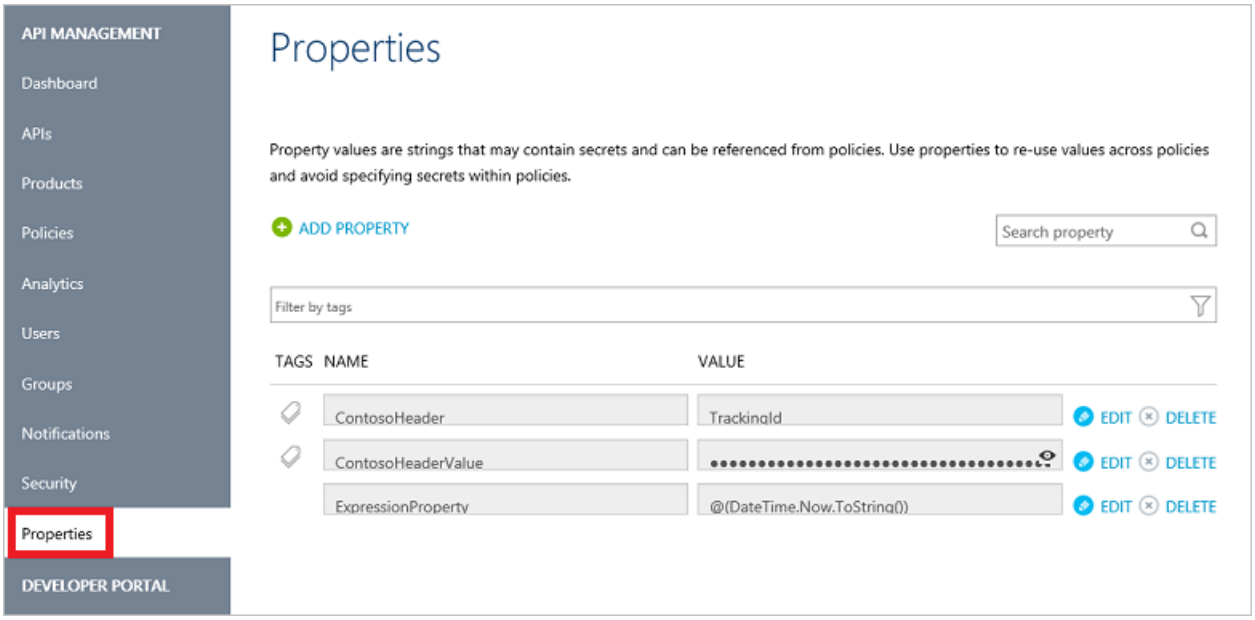
steved0x • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

API Management 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. Policy statements can be constructed using literal text values, policy expressions, and properties.

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. Each property has the following attributes.

ATTRIBUTE	TYPE	DESCRIPTION
Name	string	The name of the property. It may contain only letters, digits, period, dash, and underscore characters.
Value	string	The value of the property. It may not be empty or consist only of whitespace.
Secret	boolean	Determines whether the value is a secret and should be encrypted or not.
Tags	array of string	Optional tags that when provided can be used to filter the property list.

Properties are configured in the publisher portal on the **Properties** tab. In the following example, three properties are configured.



Property values can contain literal strings and policy expressions. The following table shows the previous three sample properties and their attributes. The value of `ExpressionProperty` is a policy expression that returns a string containing the current date and time. The property `ContosoHeaderValue` is marked as a secret, so its value is not displayed.

NAME	VALUE	SECRET	TAGS
ContosoHeader	TrackingId	False	Contoso
ContosoHeaderValue	••••••••••••••••••••	True	Contoso
ExpressionProperty	@(DateTime.Now.ToString())	False	

To use a property

To use a property in a policy, place the property name inside a double pair of braces like `{{ContosoHeader}}`, as shown in the following example.

```
<set-header name="{{ContosoHeader}}" exists-action="override">
  <value>{{ContosoHeaderValue}}</value>
</set-header>
```

In this example, `ContosoHeader` is used as the name of a header in a `set-header` policy, and `ContosoHeaderValue` is used as the value of that header. When this policy is evaluated during a request or response to the API Management gateway, `{{ContosoHeader}}` and `{{ContosoHeaderValue}}` are replaced with their respective property values.

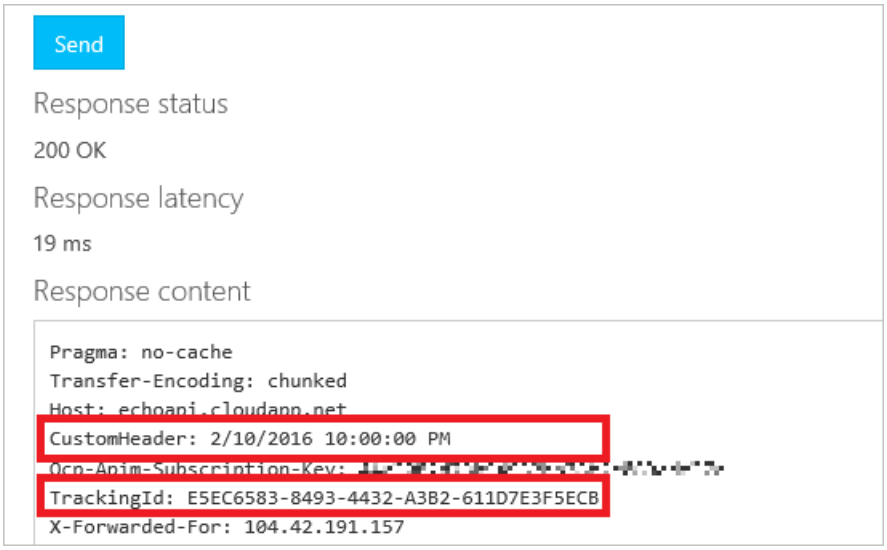
Properties can be used as complete attribute or element values as shown in the previous example, but they can also be inserted into or combined with part of a literal text expression as shown in the following example:
`<set-header name = "CustomHeader{{ContosoHeader}}" ...>`

Properties can also contain policy expressions. In the following example, the `ExpressionProperty` is used.

```
<set-header name="CustomHeader" exists-action="override">
    <value>{{ExpressionProperty}}</value>
</set-header>
```

When this policy is evaluated, `{{ExpressionProperty}}` is replaced with its value: `@(DateTime.Now.ToString())`. Since the value is a policy expression, the expression is evaluated and the policy proceeds with its execution.

You can test this out in the developer portal by calling an operation that has a policy with properties in scope. In the following example, an operation is called with the two previous example `set-header` policies with properties. Note that the response contains two custom headers that were configured using policies with properties.



If you look at the API Inspector trace for a call that includes the two previous sample policies with properties, you can see the two `set-header` policies with the property values inserted as well as the policy expression evaluation for the property that contained the policy expression.

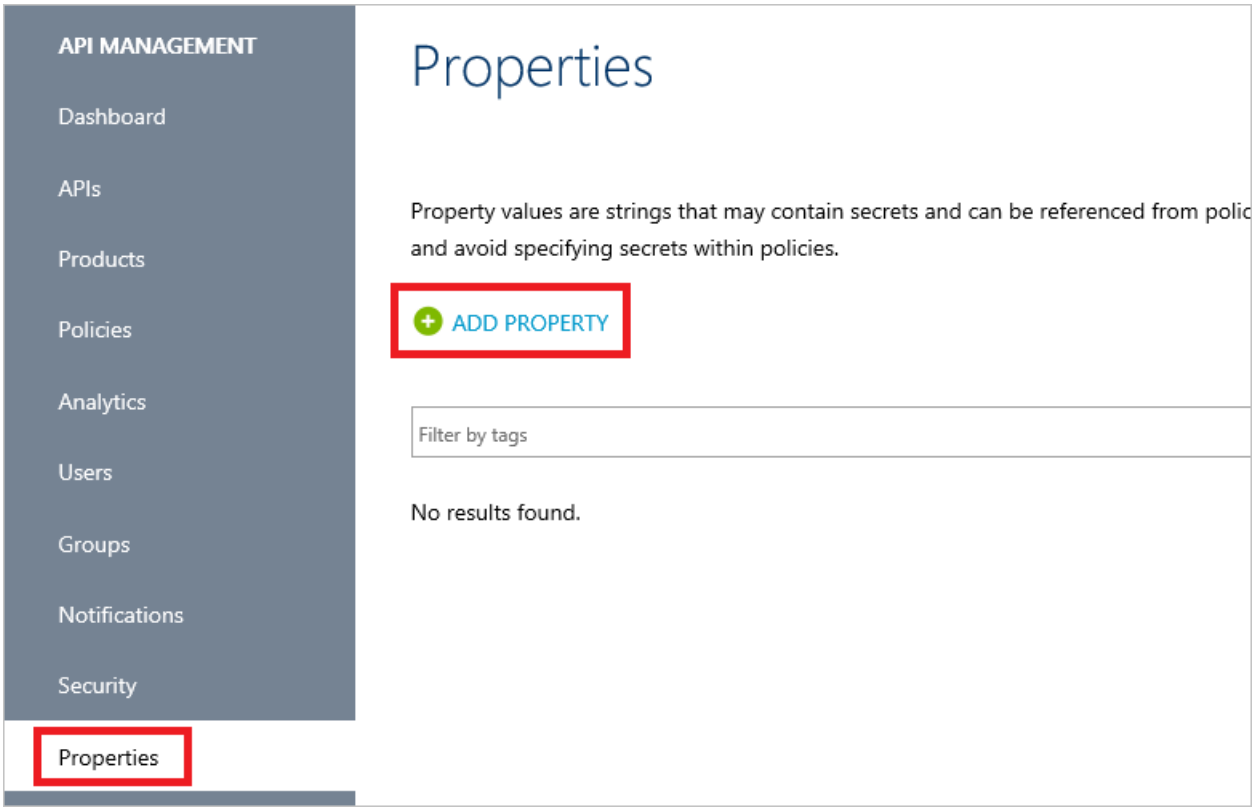

```
{
  "source": "set-header",
  "timestamp": "2016-02-12T18:32:36.2105880Z",
  "elapsed": "00:00:00.0156479",
  "data": {
    "message": "Specified value was assigned to the header (see below).",
    "header": {
      "name": "TrackingId",
      "value": "E5EC6583-8493-4432-A3B2-611D7E3F5ECB"
    }
  }
},
{
  "source": "policy-expressions",
  "timestamp": "2016-02-12T18:32:36.2105880Z",
  "elapsed": "00:00:00.0169241",
  "data": {
    "message": "Expression was successfully evaluated.",
    "expression": "DateTime.Now.ToString()",
    "value": "2/12/2016 6:32:36 PM"
  }
},
{
  "source": "set-header",
  "timestamp": "2016-02-12T18:32:36.2105880Z",
  "elapsed": "00:00:00.0169386",
  "data": {
    "message": "Specified value was assigned to the header (see below).",
    "header": {
      "name": "CustomHeader",
      "value": "2/12/2016 6:32:36 PM"
    }
  }
}
```


Note that while property values can contain policy expressions, property values can't contain other properties. If

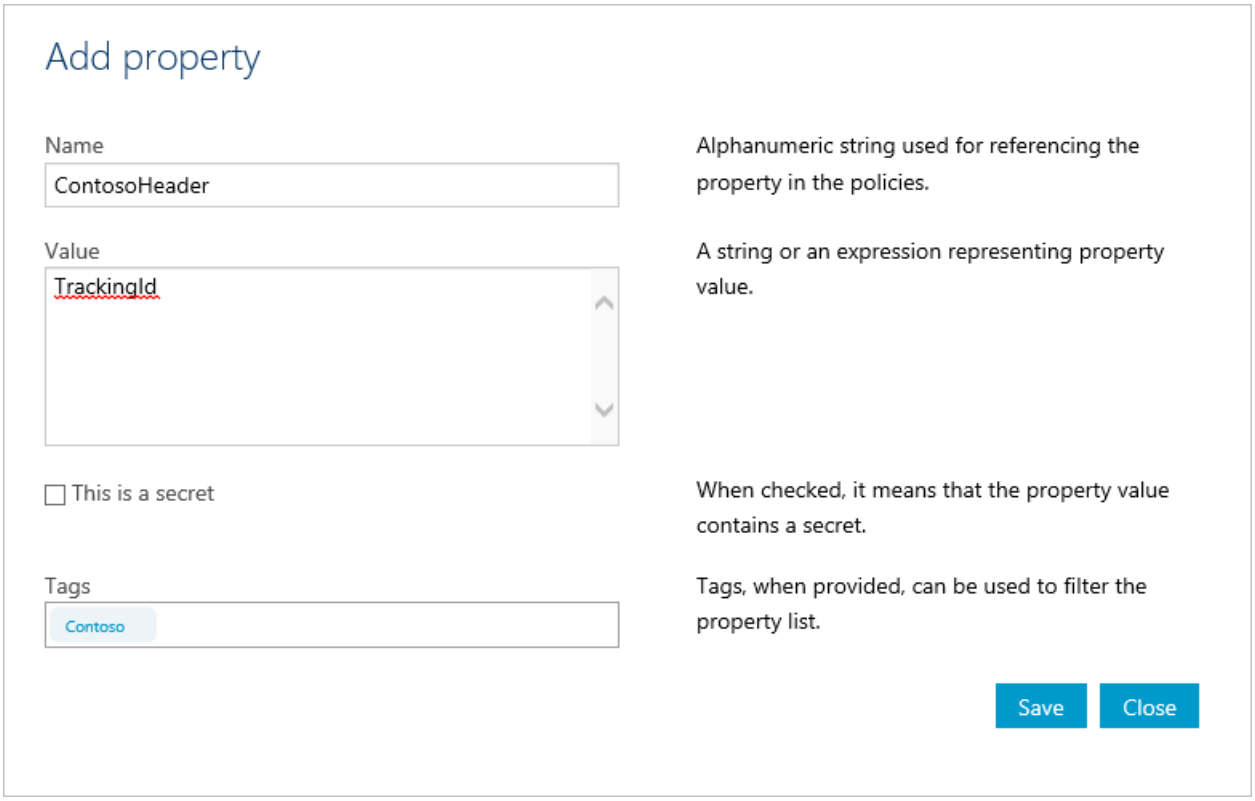
text containing a property reference is used for a property value, such as `Property value text {{MyProperty}}`, that property reference won't be replaced and will be included as part of the property value.

To create a property

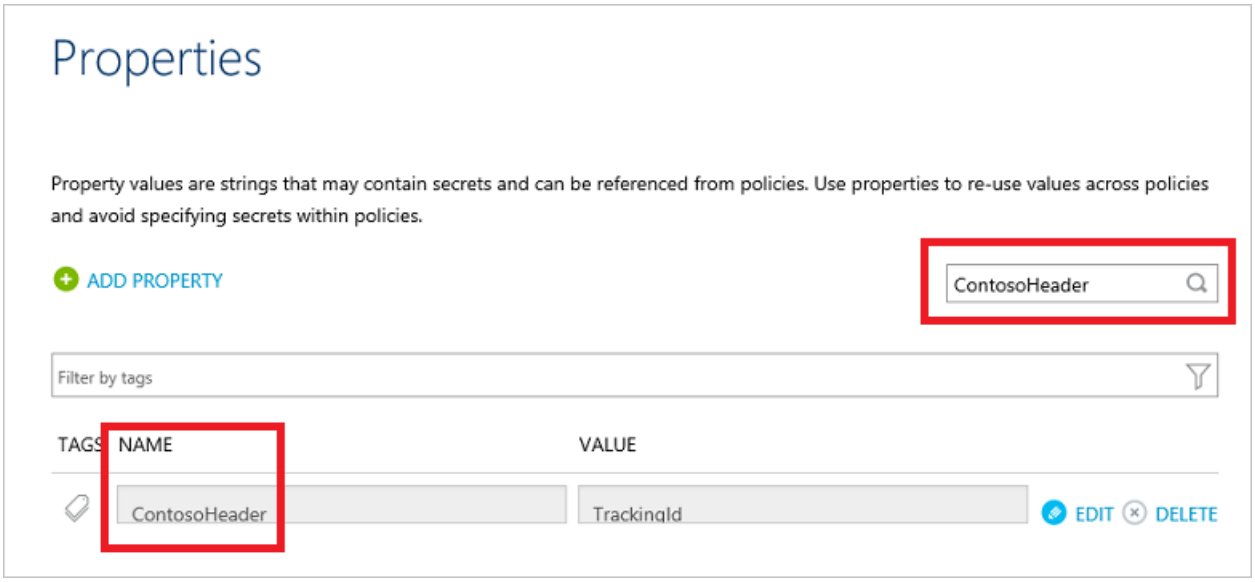
To create a property, click **Add property** on the **Properties** tab.



Name and **Value** are required values. If this property value is a secret, check the **This is a secret** checkbox. Enter one or more optional tags to help with organizing your properties, and click **Save**.



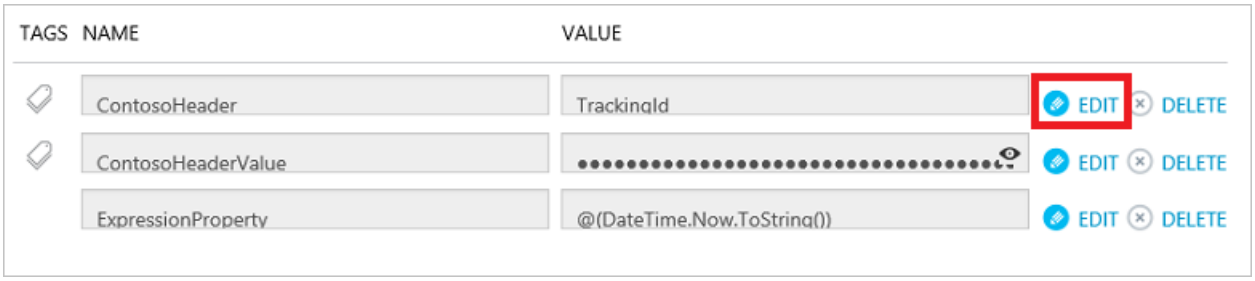
When a new property is saved, the **Search property** textbox is populated with the name of the new property and the new property is displayed. To display all properties, clear the **Search property** textbox and press enter.



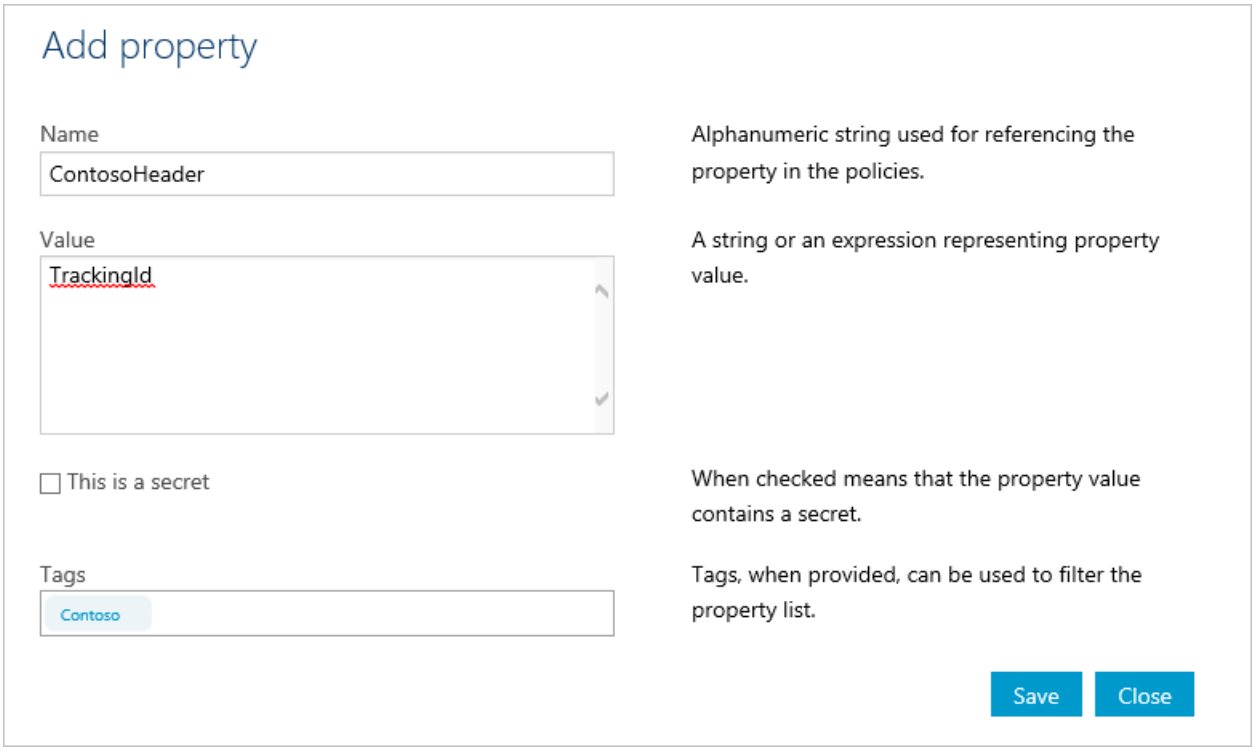
For information on creating a property using the REST API, see Create a property using the REST API.

To edit a property

To edit a property, click **Edit** beside the property to edit.



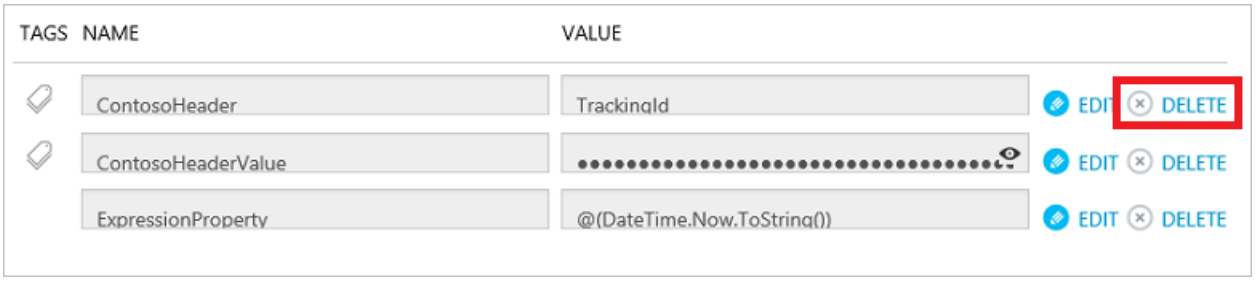
Make any desired changes, and click **Save**. If you change the property name, any policies that reference that property are automatically updated to use the new name.



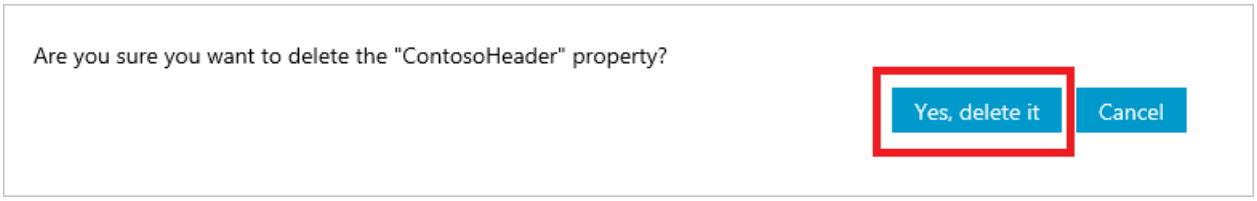
For information on editing a property using the REST API, see Edit a property using the REST API.

To delete a property

To delete a property, click **Delete** beside the property to delete.



Click **Yes, delete it** to confirm.

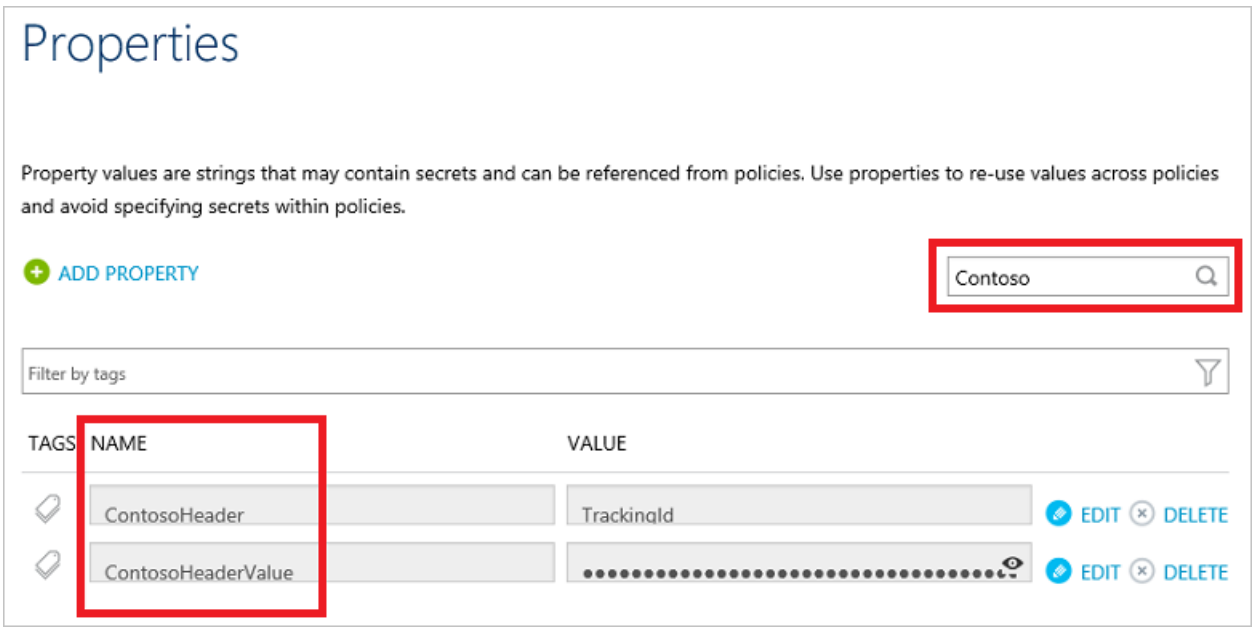


> **IMPORTANT**
> If the property is referenced by any policies, you will be unable to successfully delete it until you remove the property from all policies that use it.

For information on deleting a property using the REST API, see Delete a property using the REST API.

To search and filter properties

The **Properties** tab includes searching and filtering capabilities to help you manage your properties. To filter the property list by property name, enter a search term in the **Search property** textbox. To display all properties, clear the **Search property** textbox and press enter.



To filter the property list by tag values, enter one or more tags into the **Filter by tags** textbox. To display all properties, clear the **Filter by tags** textbox and press enter.

Next steps

- Learn more about working with policies
  - Policies in API Management
  - Policy reference
  - Policy expressions

Watch a video overview

Customize the developer portal in Azure API Management

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

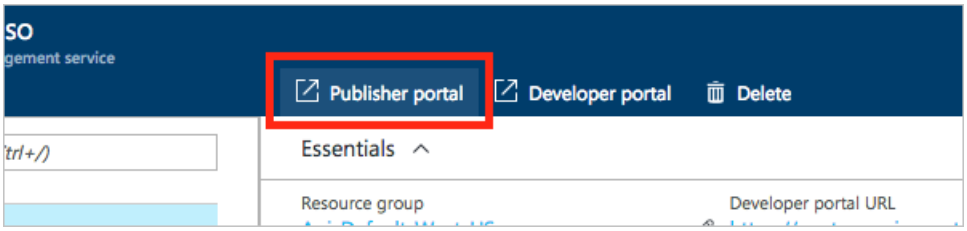
steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • Sigrid Elenga

This guide shows you how to modify the look and feel of the developer portal in Azure API Management for consistency with your brand.

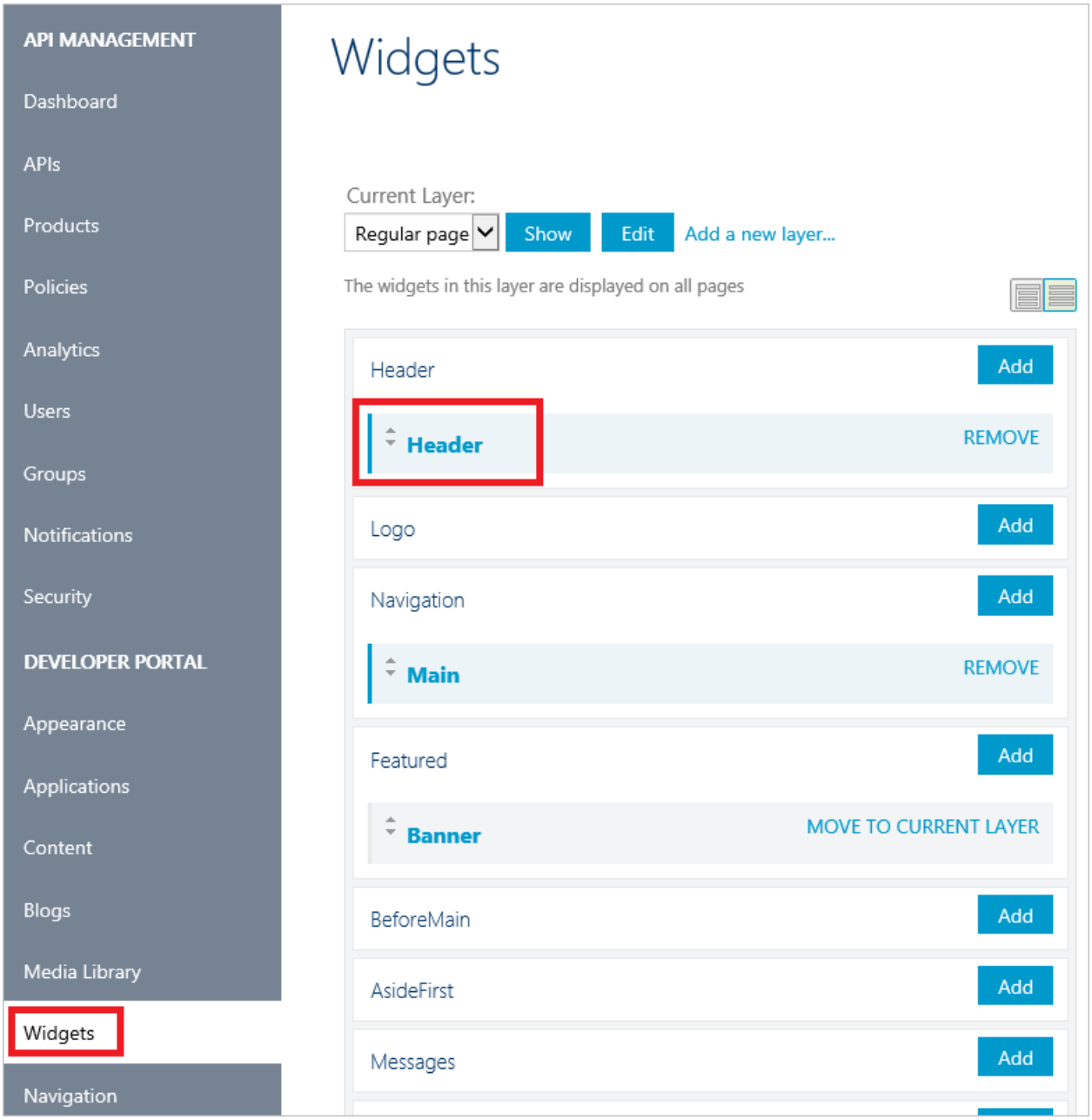
Change the text or logo in the page header

One of the key aspects of portal customization is replacing the text at the top of all pages with your company name or logo.

Content within the developer portal is modified via the publisher portal which is accessible from the Azure Portal. To reach it, click **Publisher portal** from the service toolbar of your API Management instance.



The developer portal is based on a content management system or CMS. The header that appears on every page is a special type of content known as a widget. To edit the contents of that widget, click **Widgets** from the **Developer Portal** menu on the left, and then select the **Header** widget from the list.



The contents of the header is editable from within the **Body** field. Change the text to "Fabrikam Developer Portal", and then click **Save** at the bottom of the page.

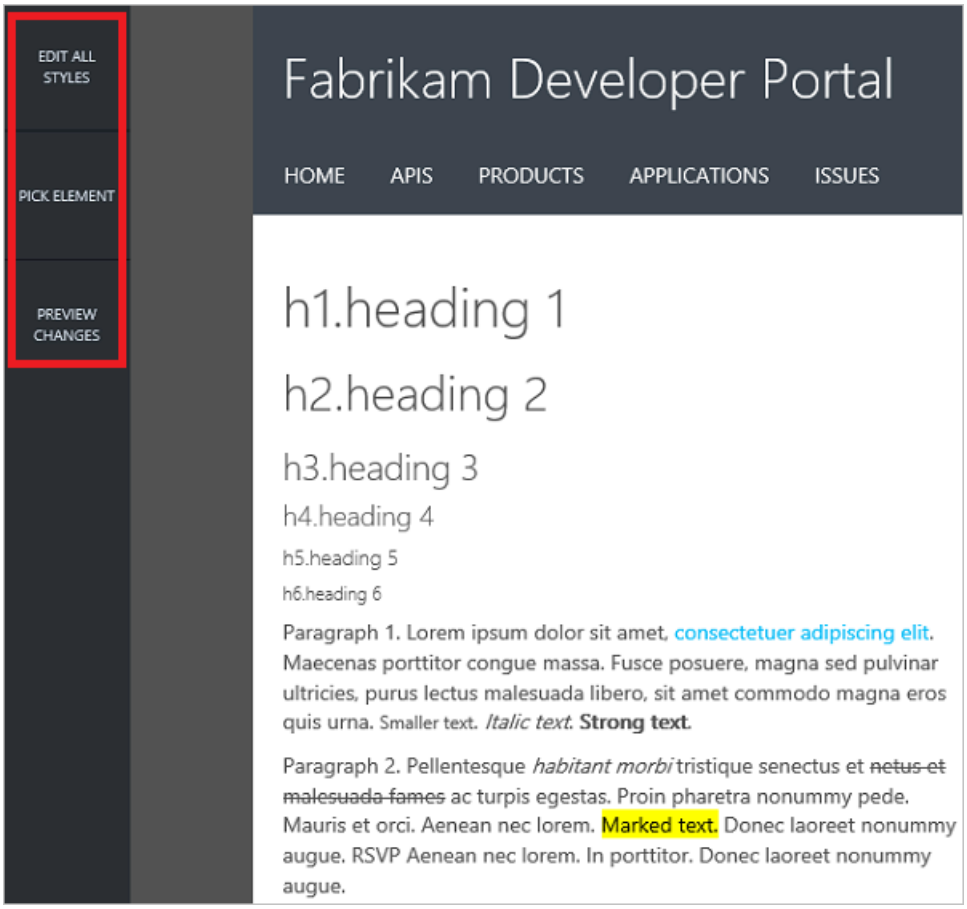
Now you should be able to see the new header on every page within the developer portal.

> To open the developer portal while in the publisher portal, click **Developer portal** in the top bar.

Change the styling of the headers

The colors, fonts, sizes, spacings, and other style-related elements of any page on the portal are defined by style rules. To edit the styles, click **Appearance** from the **Developer portal** menu in the publisher portal, and then click **Begin customization** to enable the styling editor.

Your browser switches to a hidden page within the developer portal that contains samples of content, with examples for all styling rules used anywhere on the site. To open the styling editor, move your cursor over the thin gray vertical line on the left-most part of the page. The editor toolbar should appear.



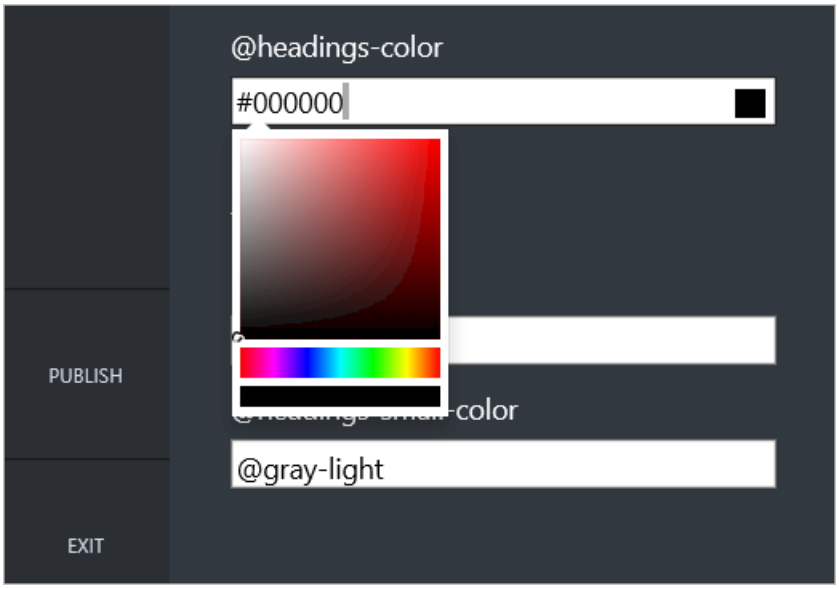
There are two main modes of editing styling rules - **Edit all rules** displays a list of all the style rules used anywhere, while **Pick element** allows you to select an element from the page you are on and displays styles only for that element.

In this section, we want to change the styling of only the headers. Click the **Pick element** option from the styling editor toolbar, and then click **Select an element to customize**. Elements now become highlighted as you hover over them with the mouse to signify what element's styles you would start editing if you clicked. Move the mouse over the text that represents the company name in the header ("Fabrikam Developer Portal" if you followed the instructions in the previous section), and then click it. A set of named and categorized styling rules appears within the styling editor.

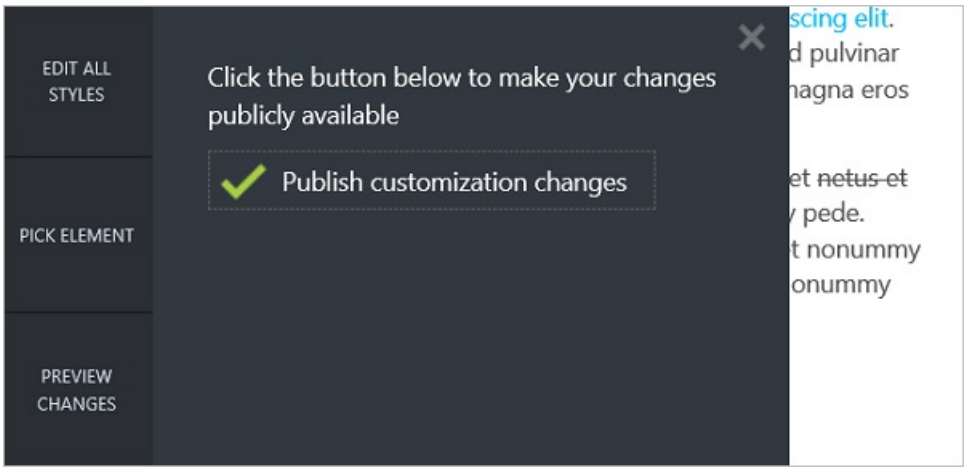
Each rule represents a styling property of the selected element. For example, for the header text selected above, the size of the text is in @font-size-h1 while the name of the font with alternatives is in @headings-font-family.

> If you're familiar with bootstrap, these rules are in fact LESS variables within the bootstrap theme used by the developer portal.

Let's change the color of the heading text. Select the entry in the **@headings-color** field and type **#000000**. This is the hex code for the color black. As you do this, you see that a square color indicator appears at the end of the text box. If you click this indicator, a color picker lets you to choose a color.



When you are done making changes to the styles of the selected element, click **Preview Changes** to see the results on the screen. At this time, they are visible only to administrators. To make these changes visible to everyone, click the **Publish** button in the styling editor and confirm the changes.

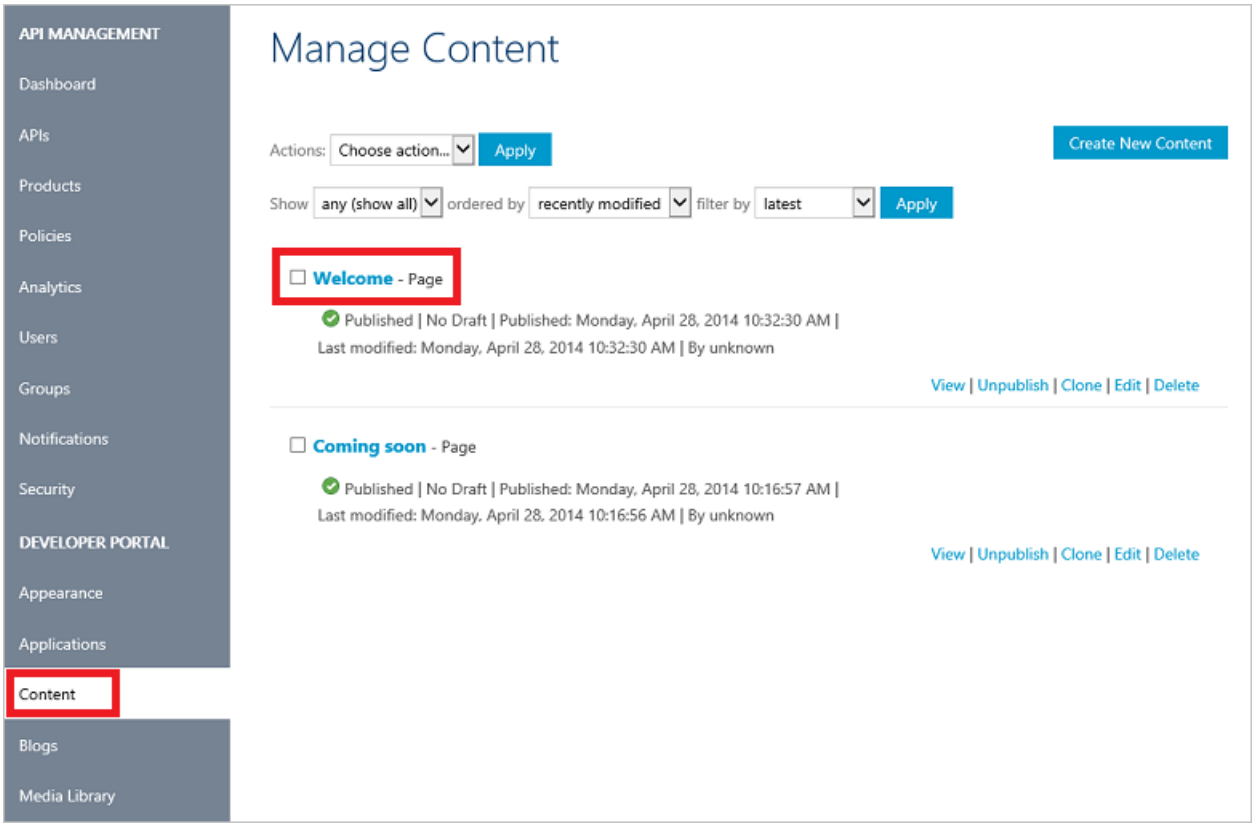


> To change the style rules that apply to any other element on the page, follow the same procedure as you did for the header. Click **Pick an element** from the styling editor, select the element you are interested in, and start modifying the values of the style rules displayed on the screen.

Edit the contents of a page

The developer portal consists of automatically generated pages like APIs, Products, Applications, Issues, and manually written content. Because it is based on a content management system, you can create such content as necessary.

To see the list of all existing content pages, click **Content** from the **Developer portal** menu in the publisher portal.



Click the **Welcome** page to edit what is displayed on the home page of the developer portal. Make the changes you want, preview them if necessary, and then click **Publish Now** to make them visible to everyone.

> The home page uses a special layout that allows it to display a banner at the top. This banner is not editable from the **Content** section. To edit this banner, click **Widgets** from the **Developer portal** menu, select **Home page** from the **Current Layer** drop-down list, and then open the **Banner** item under the **Featured section**. The contents of this widget are editable just like any other page.

Next steps

- Learn how to customize the content of developer portal pages using Developer portal templates.

How to authorize developer accounts using Azure Active Directory in Azure API Management

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • miaojiang

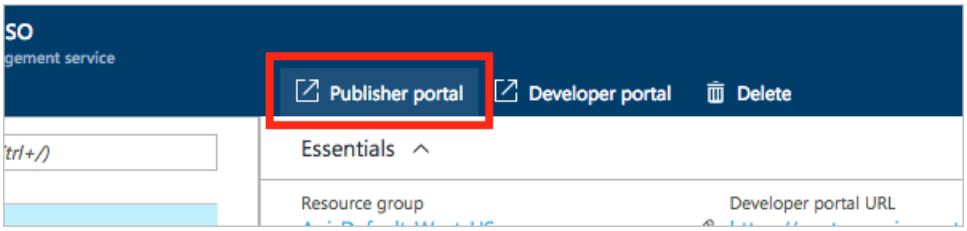
Overview

This guide shows you how to enable access to the developer portal for all users in one or more Azure Active Directories. This guide also shows you how to manage groups of Azure Active Directory users by adding external groups that contain the users of an Azure Active Directory.

> To complete the steps in this guide you must first have an Azure Active Directory in which to create an application.

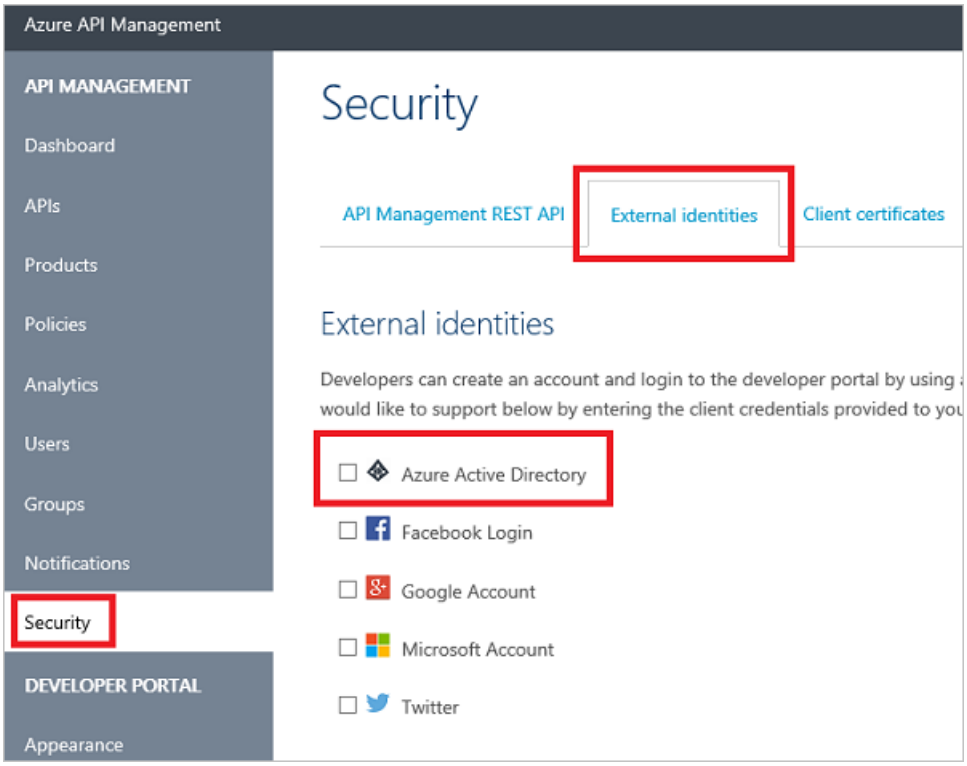
How to authorize developer accounts using Azure Active Directory

To get started, click **Publisher portal** in the Azure Portal for your API Management service. This takes you to the API Management publisher portal.

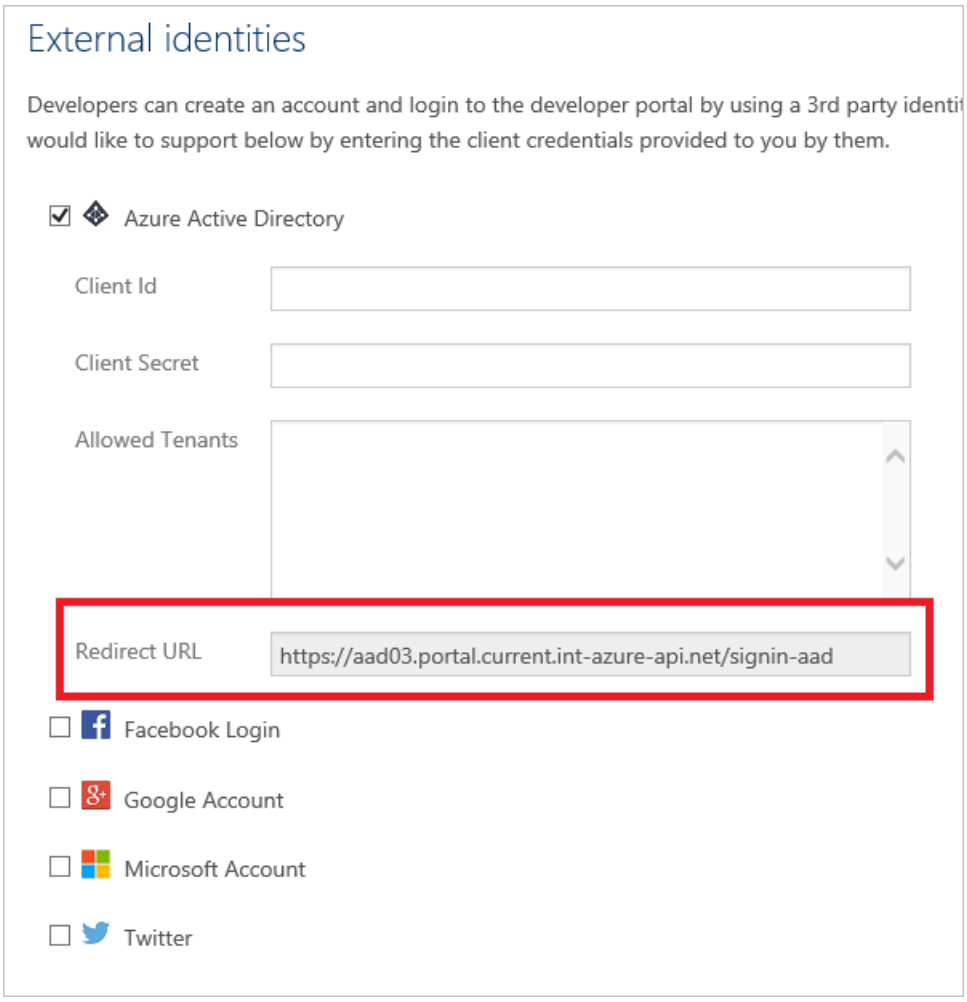


> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

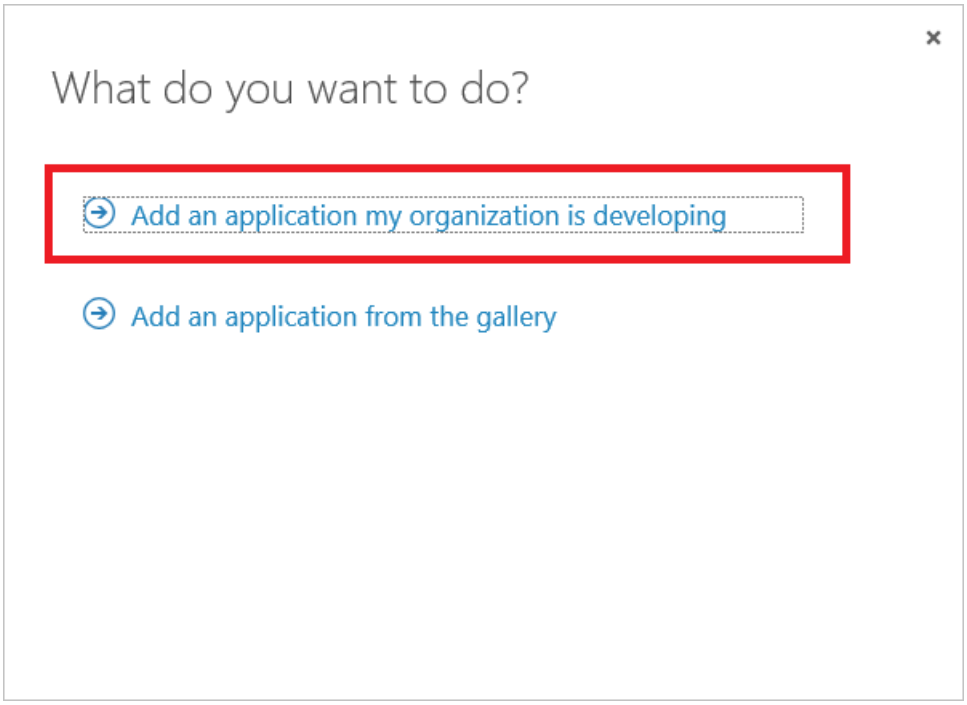
Click **Security** from the **API Management** menu on the left and click **External Identities**.



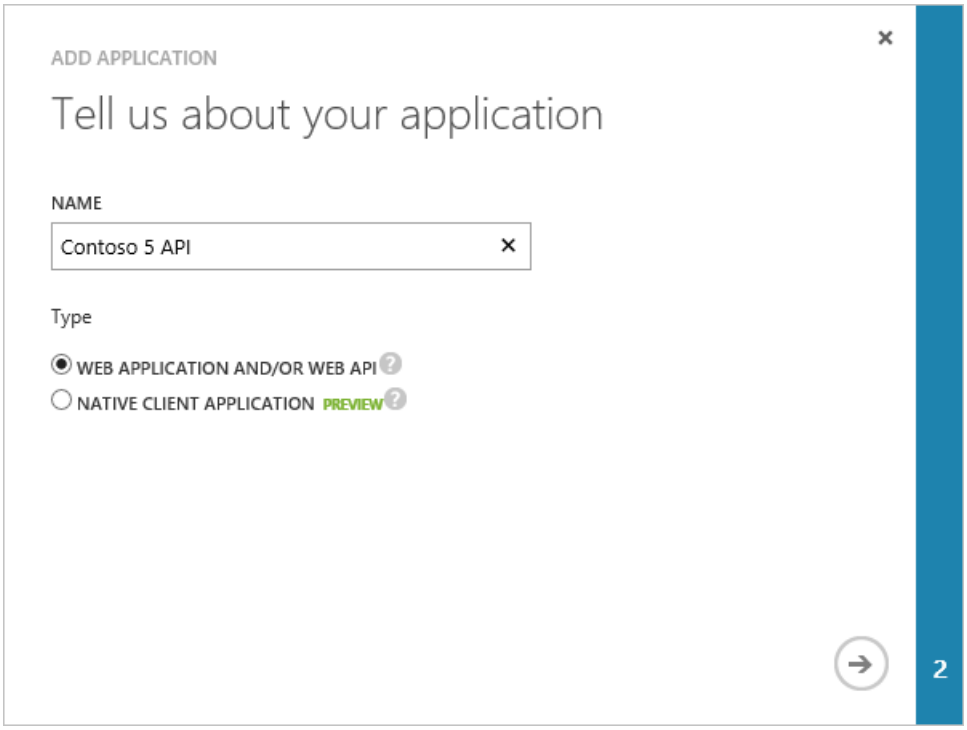
Click **Azure Active Directory**. Make a note of the **Redirect URL** and switch over to your Azure Active Directory in the Azure Classic Portal.



Click the **Add** button to create a new Azure Active Directory application, and choose **Add an application my organization is developing**.

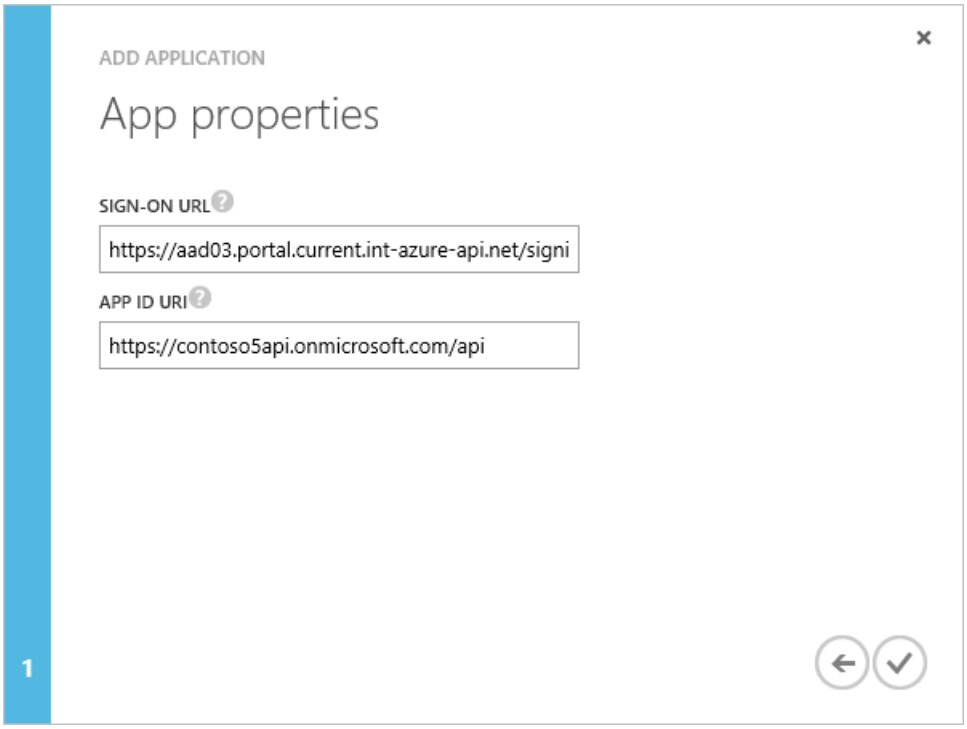


Enter a name for the application, select **Web application and/or Web API**, and click the next button.

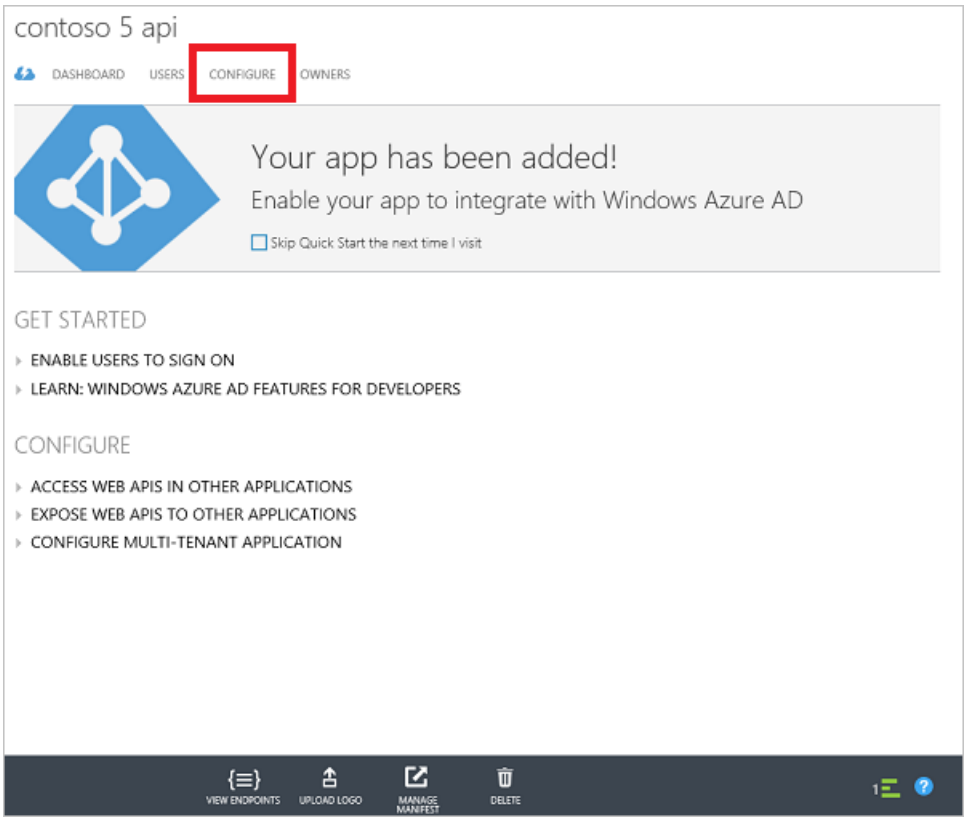


For **Sign-on URL**, enter the sign-on URL of your developer portal. In this example, the **Sign-on URL** is `https://aad03.portal.current.int-azure-api.net/signin`.

For the **App ID URL**, enter either the default domain or a custom domain for the Azure Active Directory, and append a unique string to it. In this example the default domain of **https://contoso5api.onmicrosoft.com** is used with the suffix of **/api** specified.

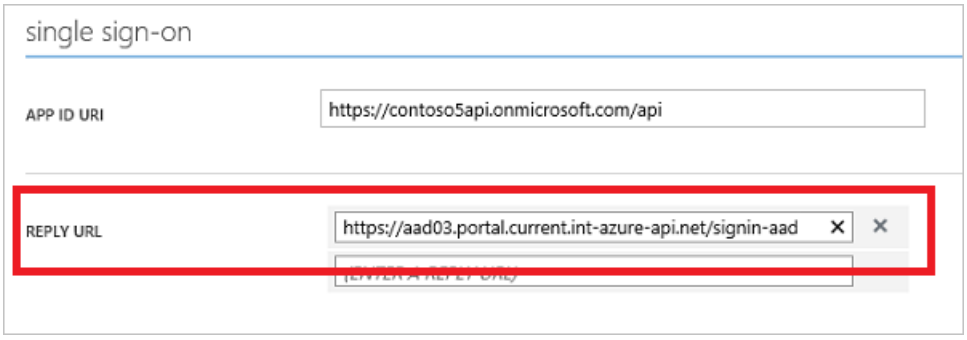


Click the check button to save and create the new application, and switch to the **Configure** tab to configure the new application.

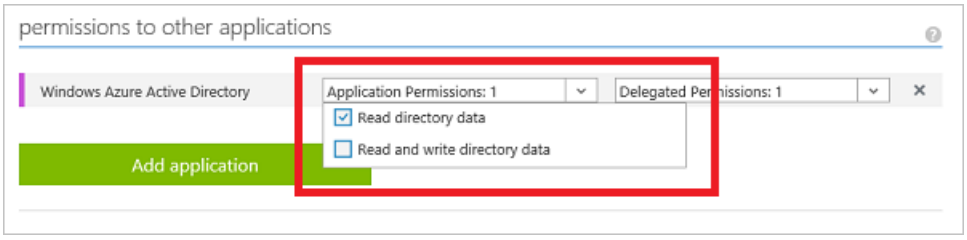


If multiple Azure Active Directories are going to be used for this application, click **Yes** for **Application is multi-tenant**. The default is **No**.

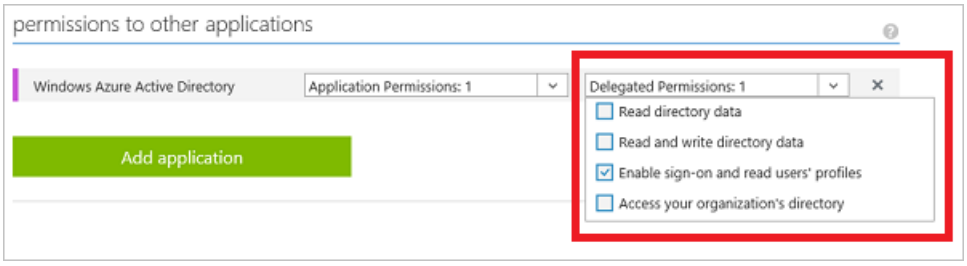
Copy the **Redirect URL** from the **Azure Active Directory** section of the **External Identities** tab in the publisher portal and paste it into the **Reply URL** text box.



Scroll to the bottom of the configure tab, select the **Application Permissions** drop-down, and check **Read directory data**.



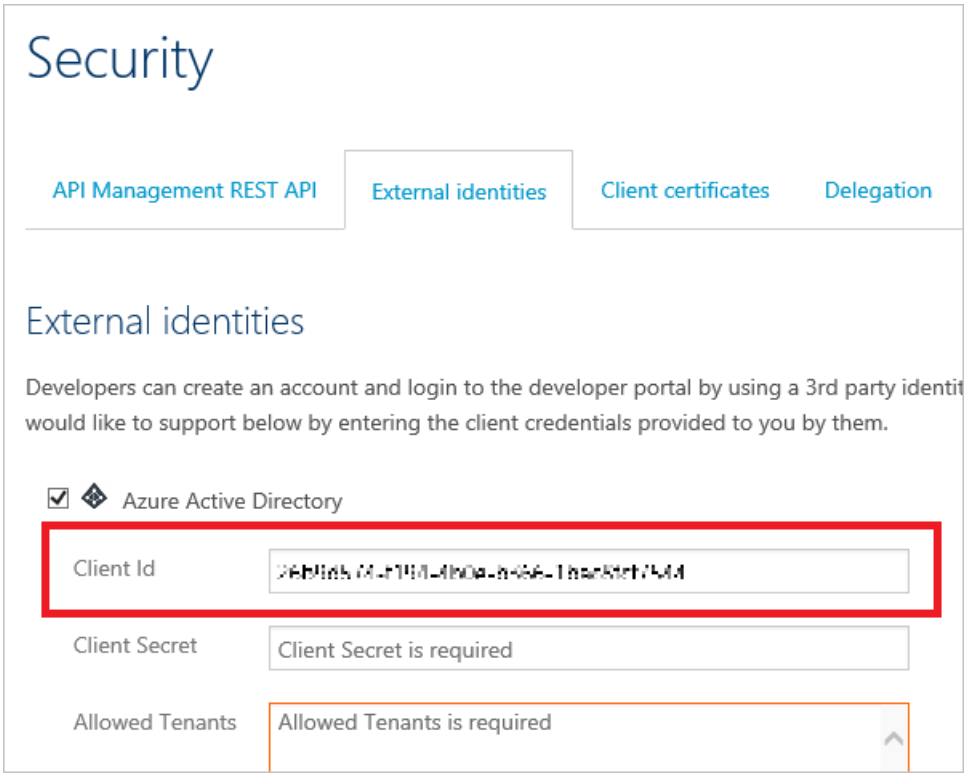
Select the **Delegate Permissions** drop-down, and check **Enable sign-on and read users' profiles**.



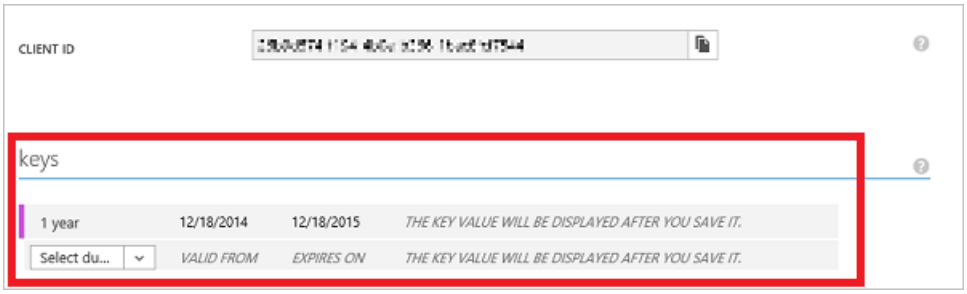
> For more information about application and delegated permissions, see Accessing the Graph API.

Copy the **Client Id** to the clipboard.

Switch back to the publisher portal and paste in the **Client Id** copied from the Azure Active Directory application configuration.

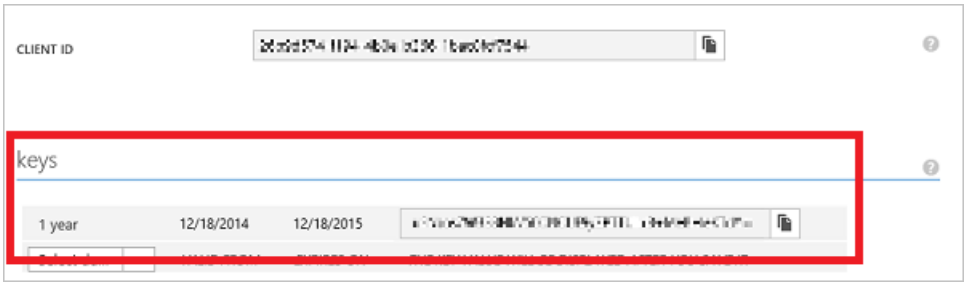


Switch back to the Azure Active Directory configuration, and click the **Select duration** drop-down in the **Keys** section and specify an interval. In this example **1 year** is used.

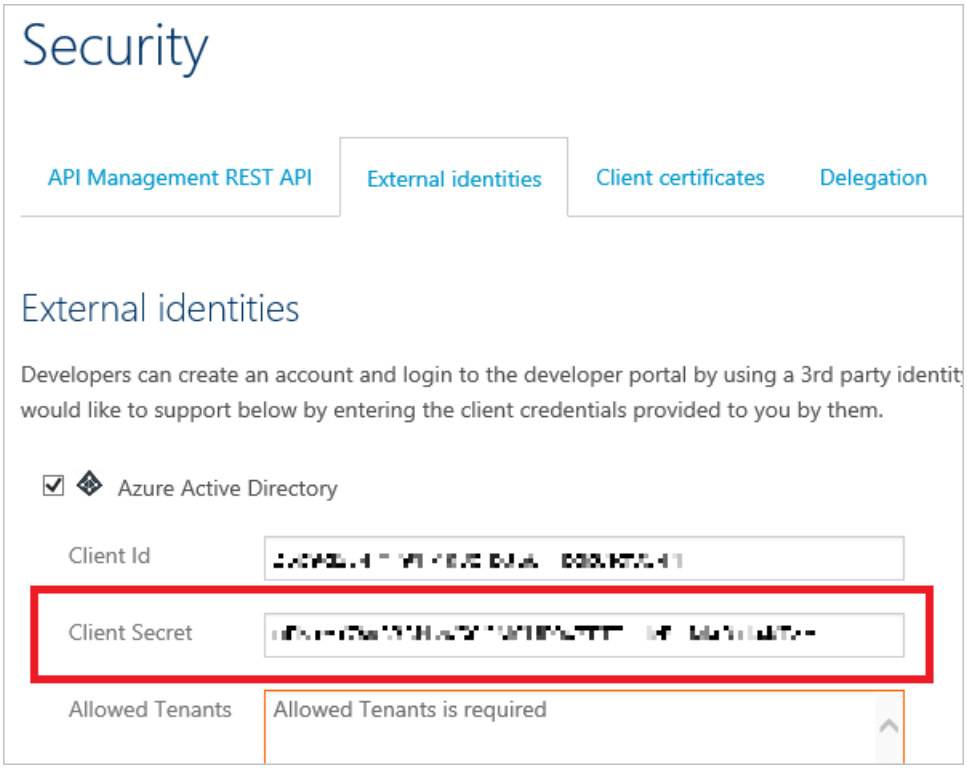


Click **Save** to save the configuration and display the key. Copy the key to the clipboard.

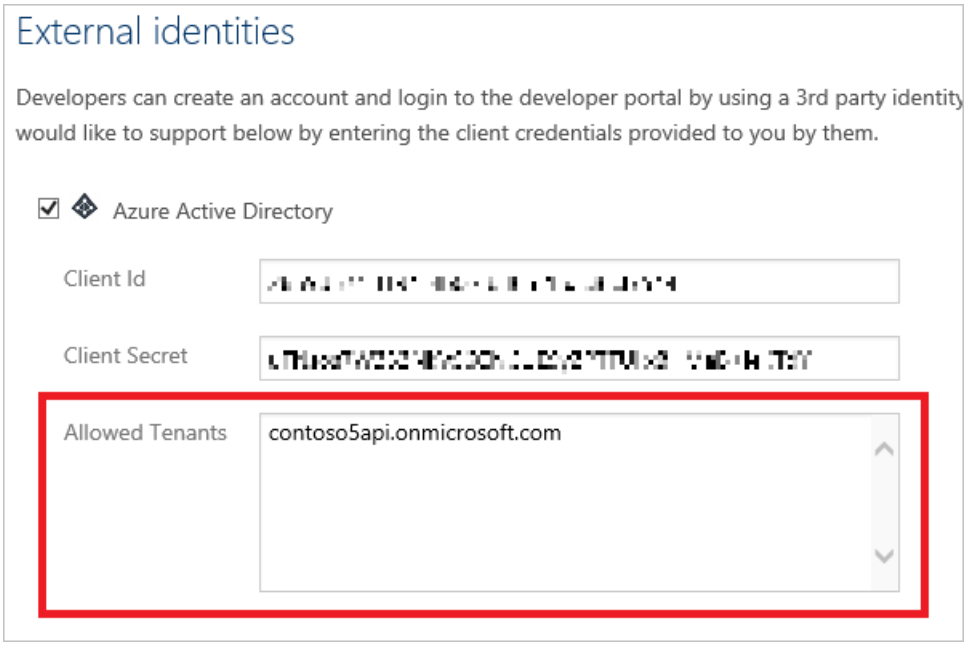
> Make a note of this key. Once you close the Azure Active Directory configuration window, the key cannot be displayed again.



Switch back to the publisher portal and paste the key into the **Client Secret** text box.

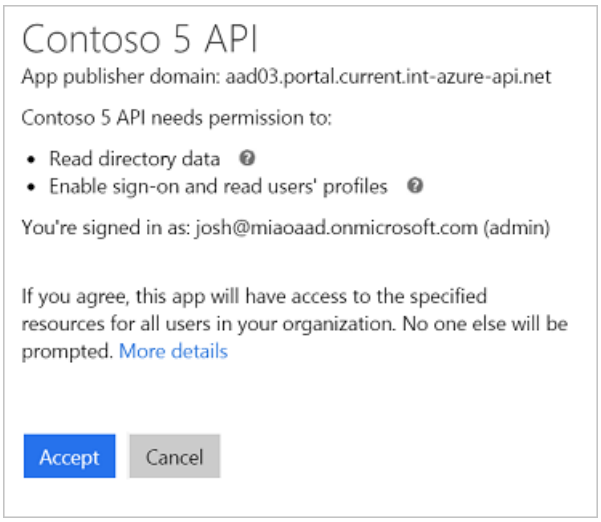


Allowed Tenants specifies which directories have access to the APIs of the API Management service instance. Specify the domains of the Azure Active Directory instances to which you want to grant access. You can separate multiple domains with newlines, spaces, or commas.



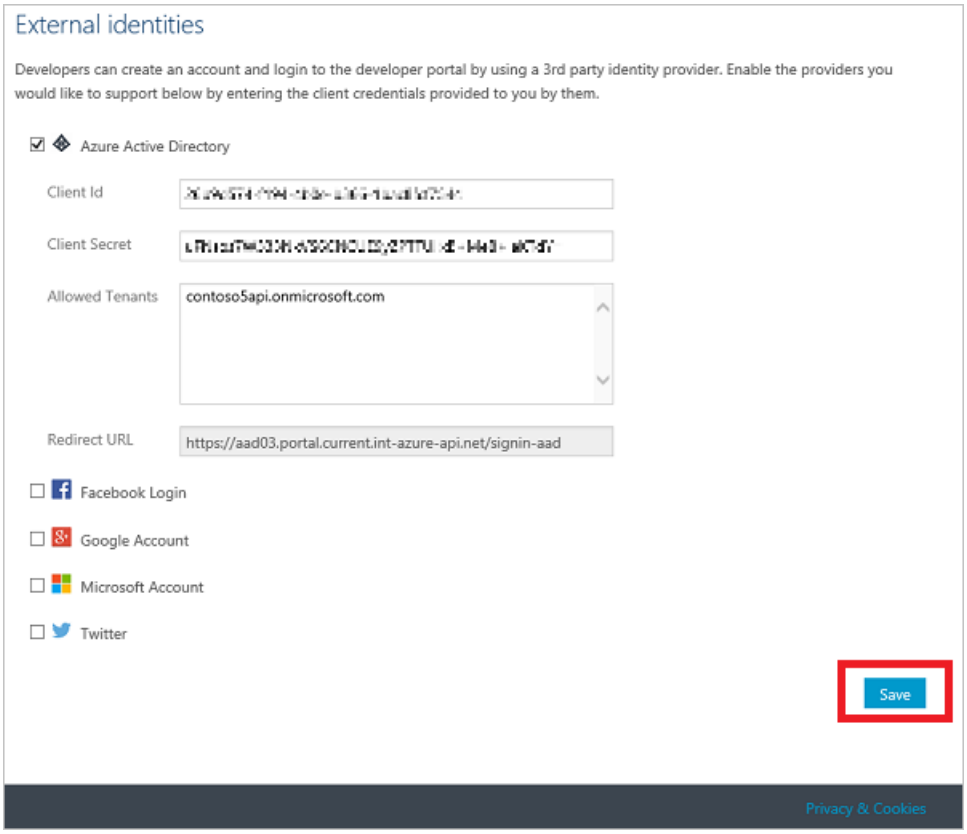
Multiple domains can be specified in the **Allowed Tenants** section. Before any user can log in from a different domain than the original domain where the application was registered, a global administrator of the different

domain must grant permission for the application to access directory data. To grant permission, a global administrator must log in to the application and click **Accept**. In the following example `miaoaad.onmicrosoft.com` has been added to **Allowed Tenants** and a global administrator from that domain is logging in for the first time.



> If a non-global administrator tries to log in before permissions are granted by a global administrator, the login attempt fails and an error screen is displayed.

Once the desired configuration is specified, click **Save**.



Once the changes are saved, the users in the specified Azure Active Directory can log into the Developer portal by following the steps in Log in to the Developer portal using an Azure Active Directory account.

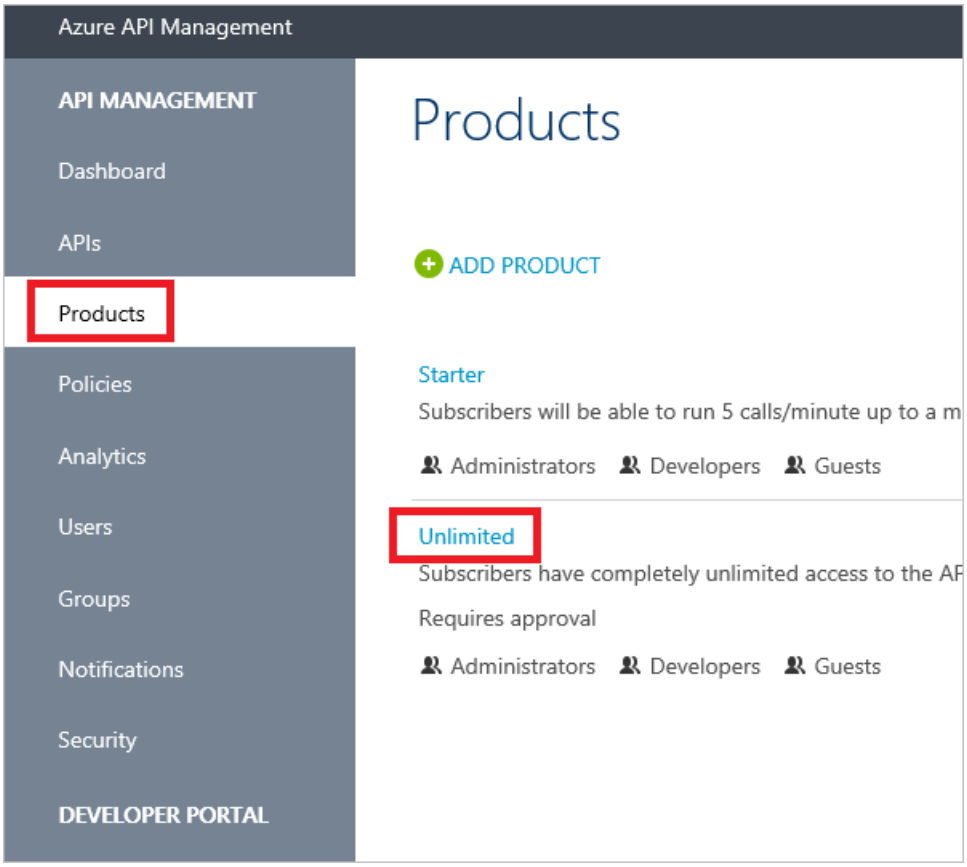
How to add an external Azure Active Directory Group

After enabling access for users in an Azure Active Directory, you can add Azure Active Directory groups into API Management to more easily manage the association of the developers in the group with the desired products.

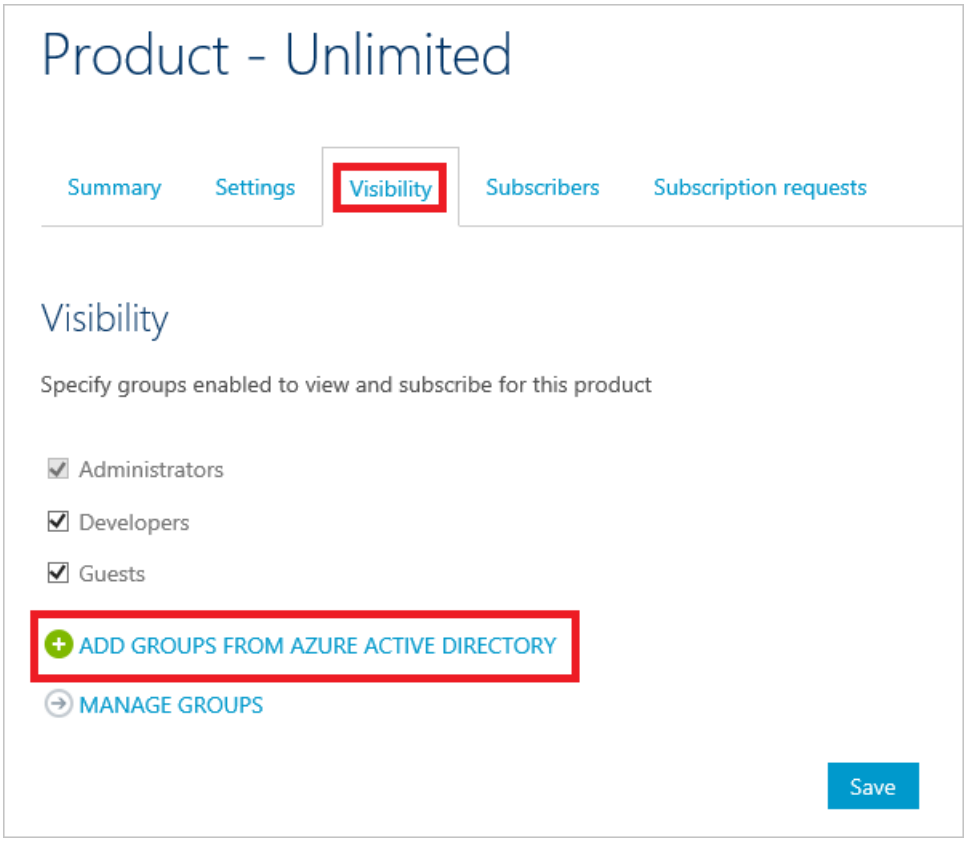
> In order to configure an external Azure Active Directory group, the Azure Active Directory must first be

configured in the Identities tab by following the procedure in the previous section.

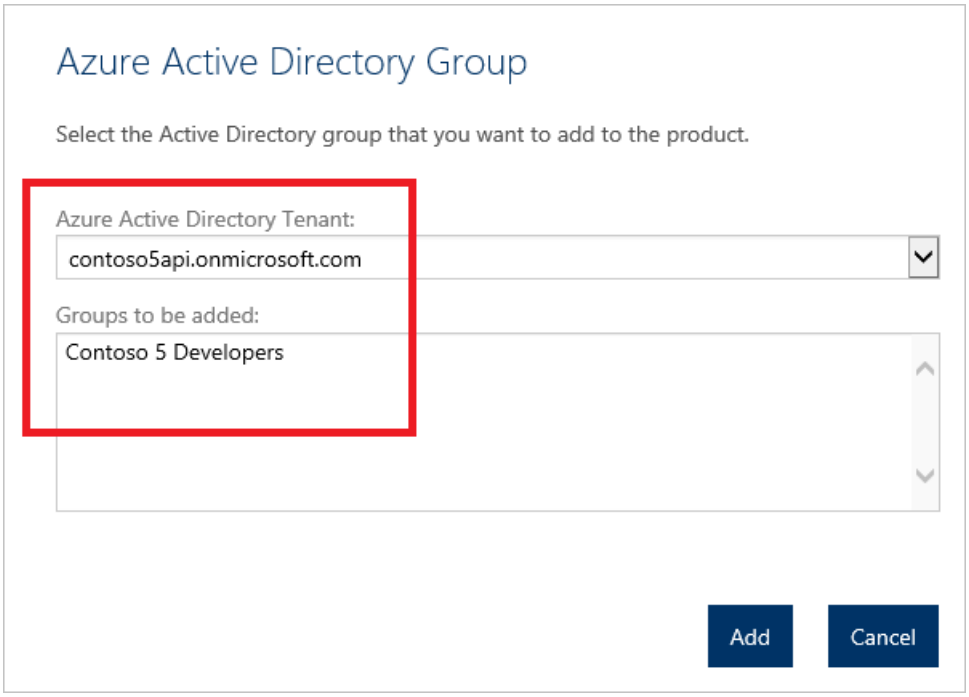
External Azure Active Directory groups are added from the **Visibility** tab of the product for which you wish to grant access to the group. Click **Products**, and then click the name of the desired product.



Switch to the **Visibility** tab, and click **Add Groups from Azure Active Directory**.

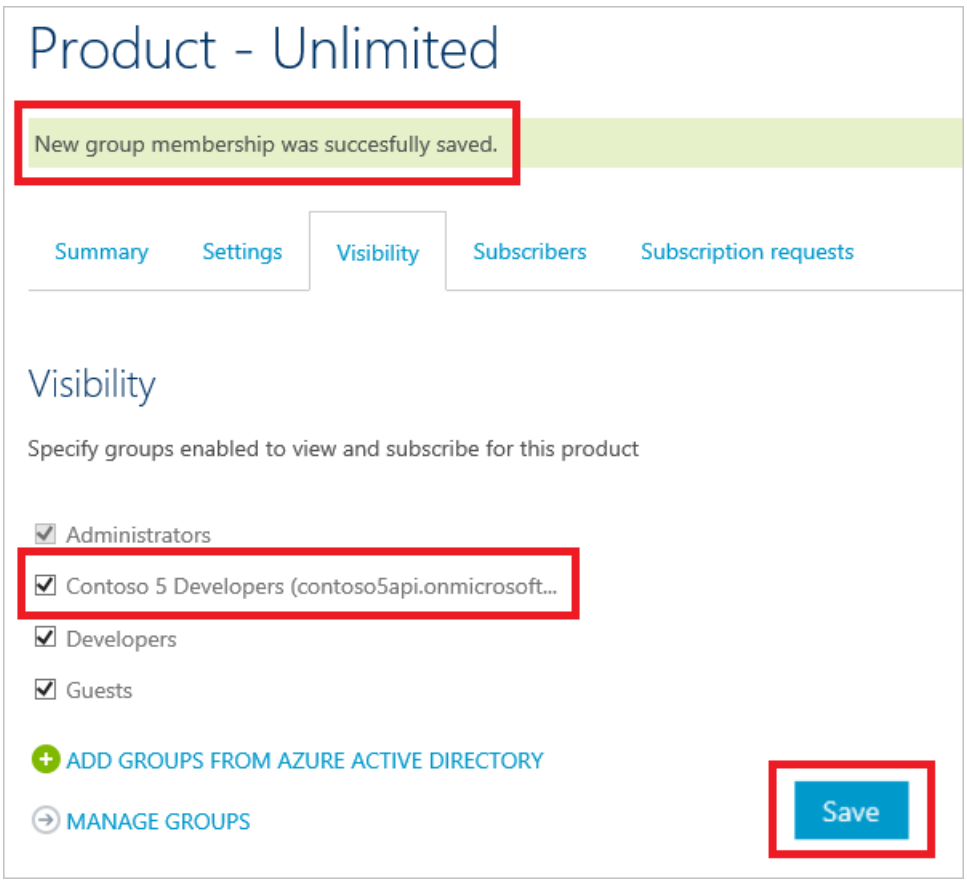


Select the **Azure Active Directory Tenant** from the drop-down list, and then type the name of the desired group in the **Groups** to be added text box.



This group name can be found in the **Groups** list for your Azure Active Directory, as shown in the following example.

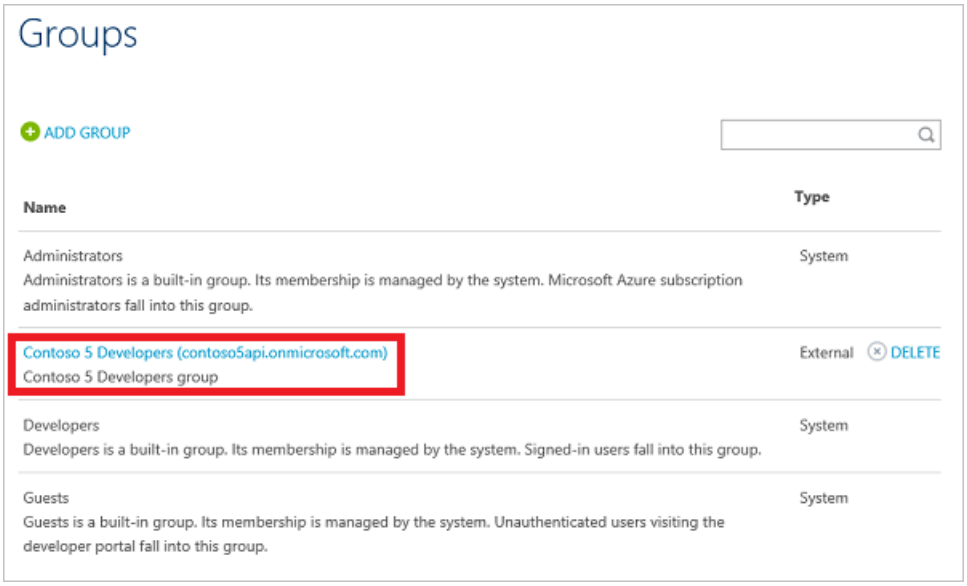
Click **Add** to validate the group name and add the group. In this example the **Contoso 5 Developers** external group is added.



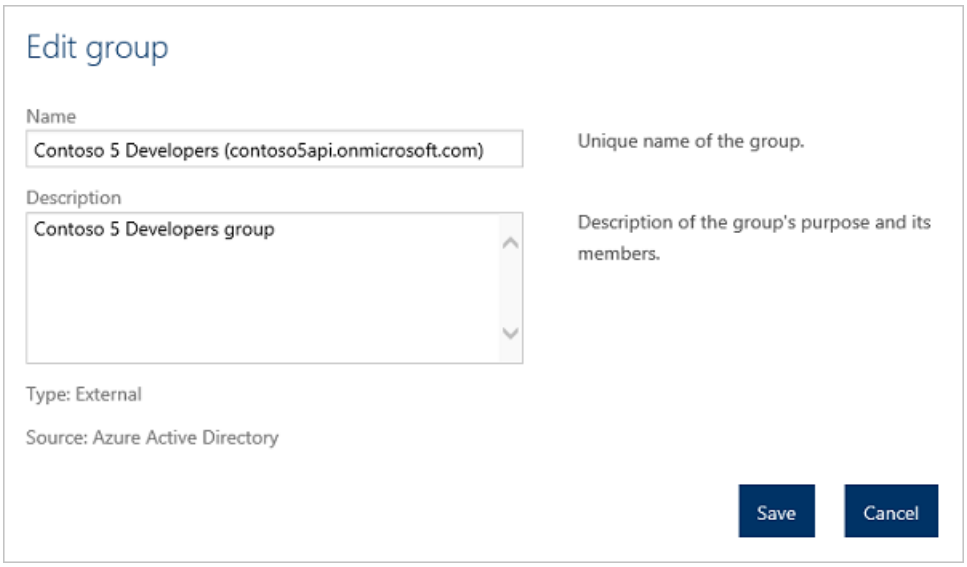
Click **Save** to save the new group selection.

Once an Azure Azure Active Directory group has been configured from one product, it is available to be checked on the **Visibility** tab for the other products in the API Management service instance.

To review and configure the properties for external groups once they have been added, click on the name of the group from the **Groups** tab.



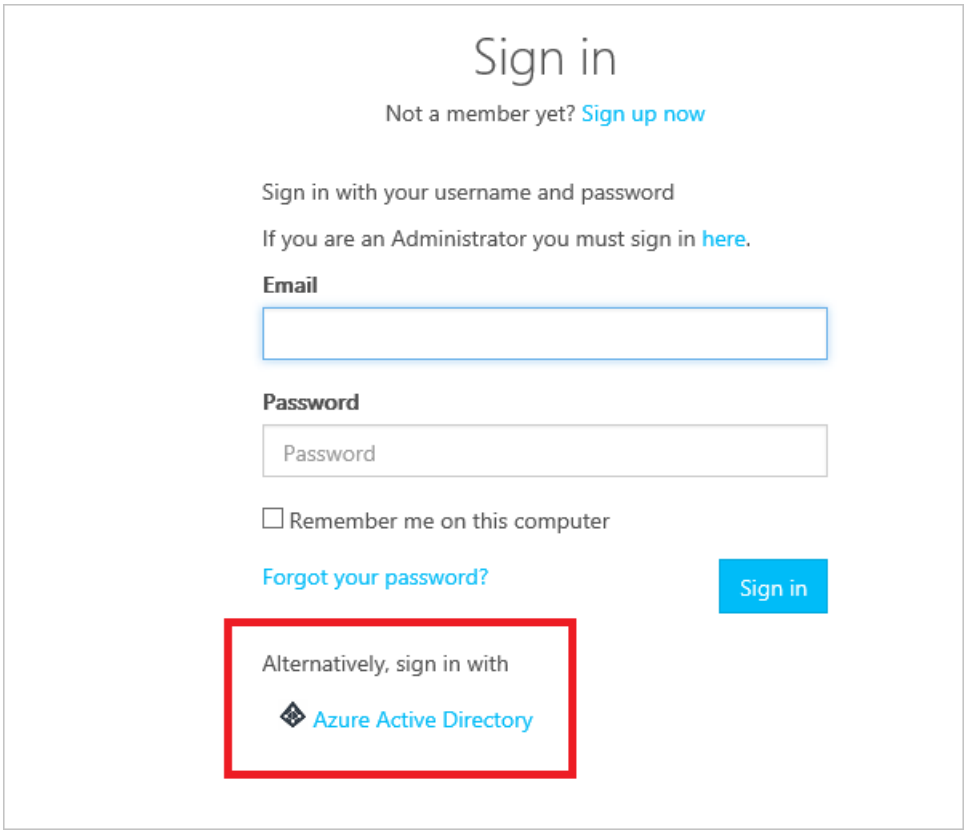
From here you can edit the **Name** and the **Description** of the group.



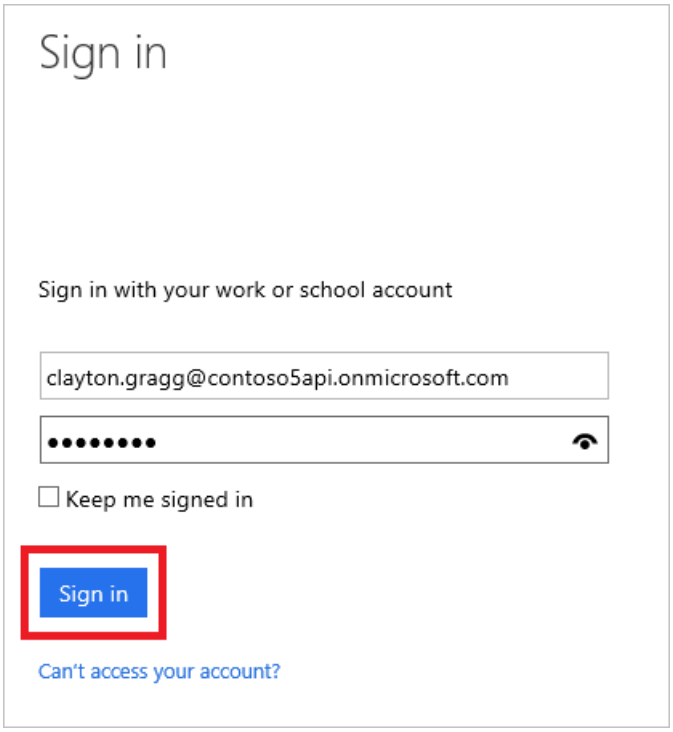
Users from the configured Azure Active Directory can log into the Developer portal and view and subscribe to any groups for which they have visibility by following the instructions in the following section.

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

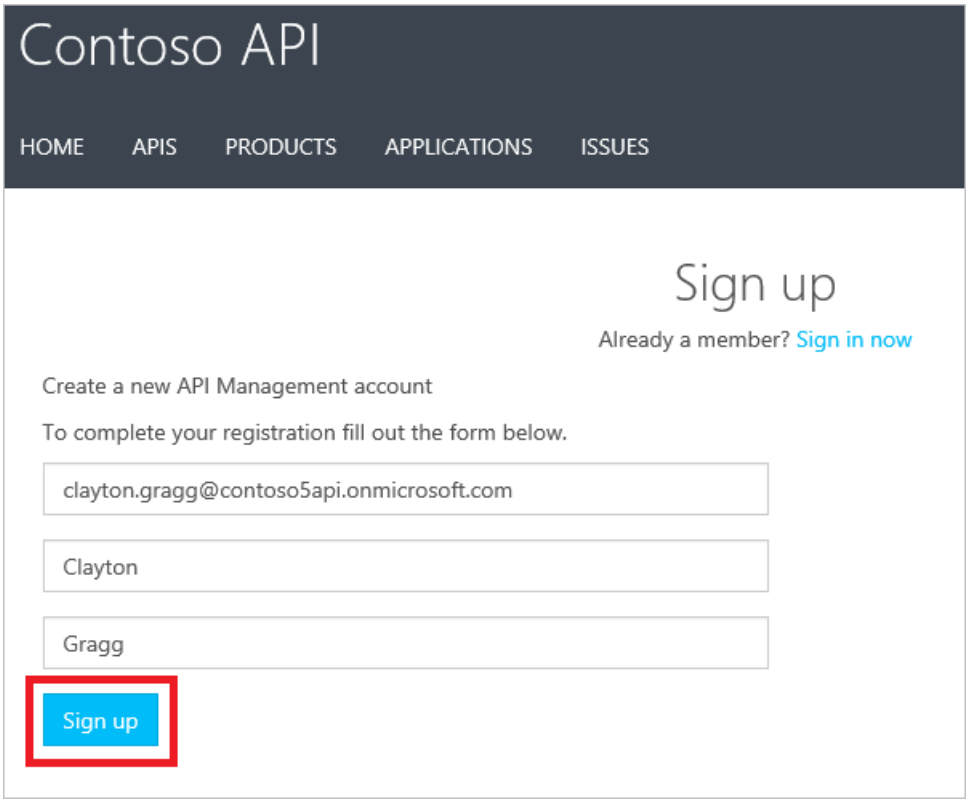
To log into the Developer portal using an Azure Active Directory account configured in the previous sections, open a new browser window using the **Sign-on URL** from the Active Directory application configuration, and click **Azure Active Directory**.



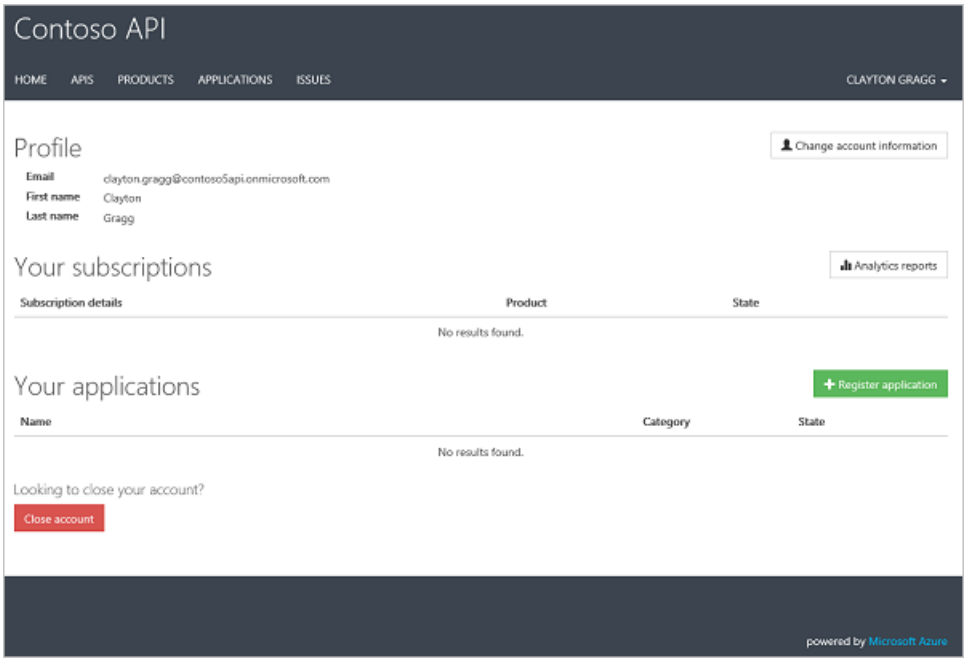
Enter the credentials of one of the users in your Azure Active Directory, and click **Sign in**.



You may be prompted with a registration form if any additional information is required. Complete the registration form and click **Sign up**.



Your user is now logged into the developer portal for your API Management service instance.



How to configure notifications and email templates in Azure API Management

11/15/2016 • 3 min to read • Edit on GitHub

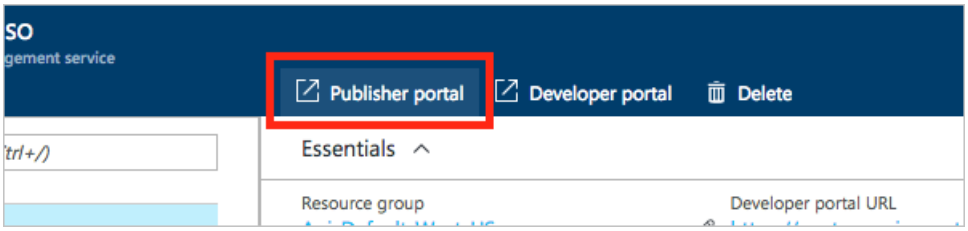
Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

API Management provides the ability to configure notifications for specific events, and to configure the email templates that are used to communicate with the administrators and developers of an API Management instance. This topic shows how to configure notifications for the available events, and provides an overview of configuring the email templates used for these events.

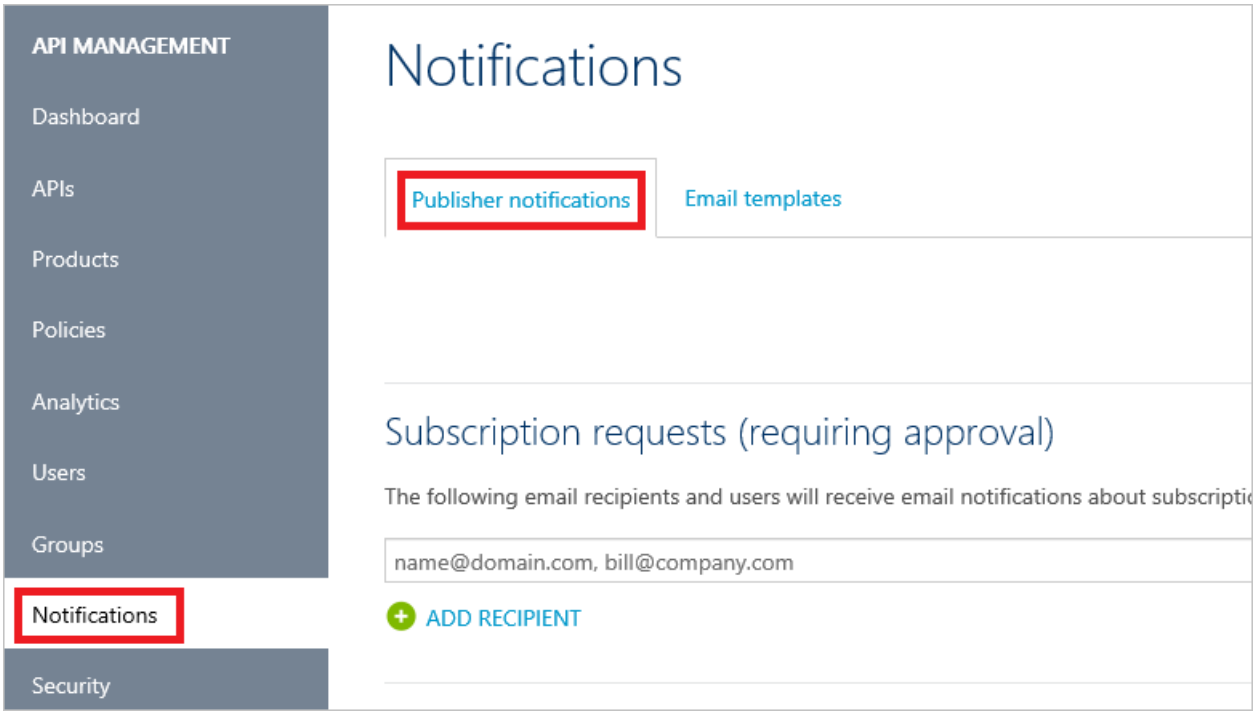
Configure publisher notifications

To configure notifications, click **Publisher portal** in the Azure Portal for your API Management service. This takes you to the API Management publisher portal.



> **NOTE**
> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

Click **Notifications** from the **API Management** menu on the left to view the available notifications.

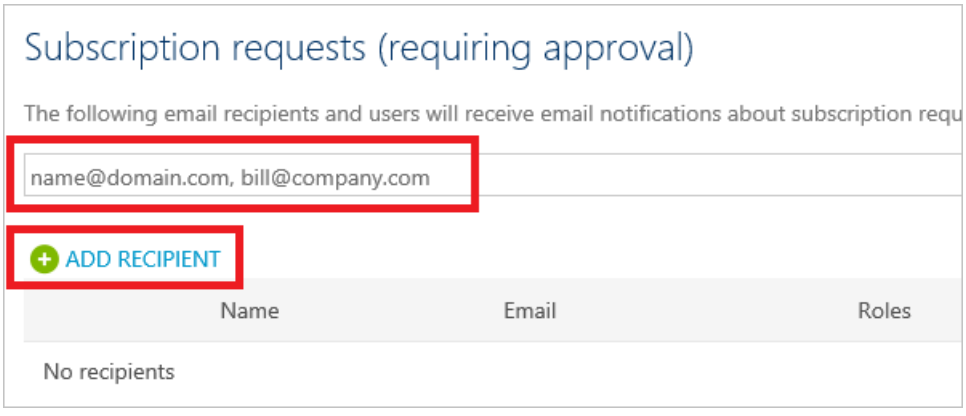


The following list of events can be configured for notifications.

- **Subscription requests (requiring approval)** - The specified email recipients and users will receive email notifications about subscription requests for API products requiring approval.
- **New subscriptions** - The specified email recipients and users will receive email notifications about new API product subscriptions.
- **Application gallery requests** - The specified email recipients and users will receive email notifications when new applications are submitted to the application gallery.
- **BCC** - The specified email recipients and users will receive email blind carbon copies of all emails sent to developers.
- **New issue or comment** - The specified email recipients and users will receive email notifications when a new issue or comment is submitted on the developer portal.
- **Close account message** - The specified email recipients and users will receive email notifications when an account is closed.
- **Approaching subscription quota limit** - The following email recipients and users will receive email notifications when subscription usage gets close to usage quota.

For each event, you can specify email recipients using the email address text box or you can select users from a list.

To specify the email addresses to be notified, enter them in the email address text box. If you have multiple email addresses, separate them using commas.



To specify the users to be notified, click **add recipient**, check the box beside the users to be notified, and click **OK**.

> **NOTE**
> Only administrators are displayed in the list.

After configuring the notification recipients, click **Save** to apply the updated notification recipients.

> **NOTE**
> If you navigate away from the **Publisher Notifications** tab the publisher portal alerts you if there are unsaved changes.

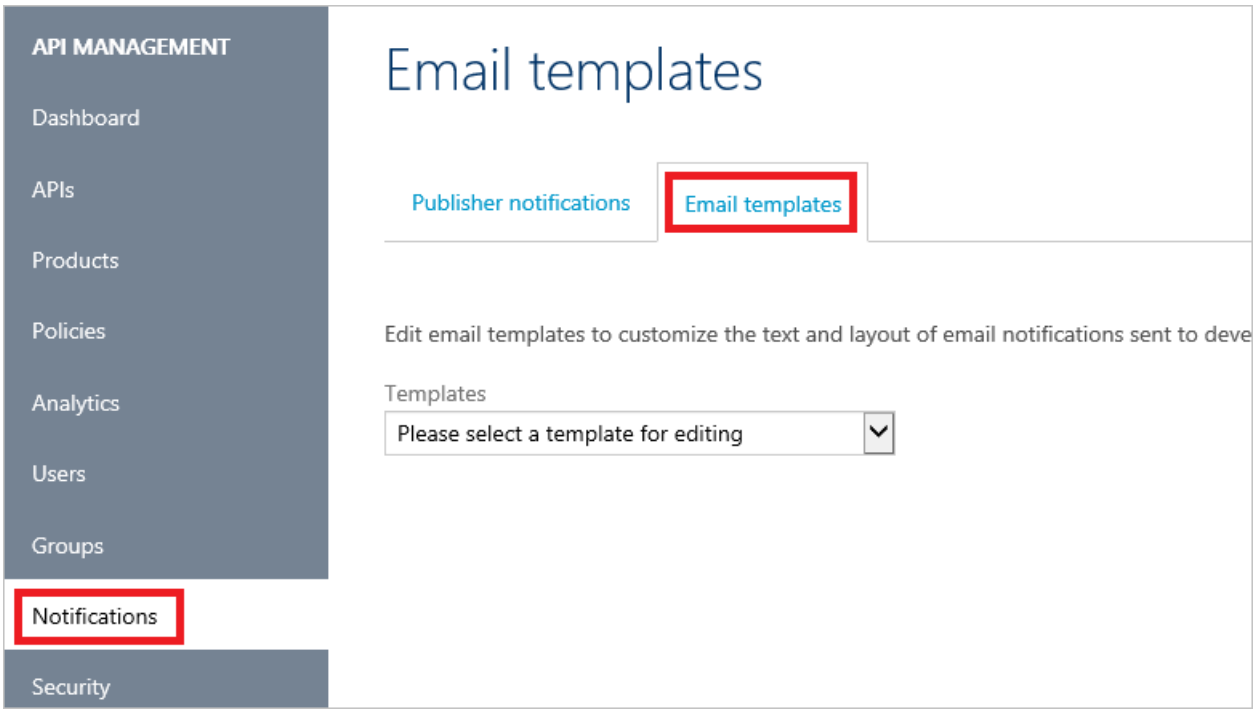
Configure email templates

API Management provides email templates for the email messages that are sent in the course of administering and using the service. The following email templates are provided.

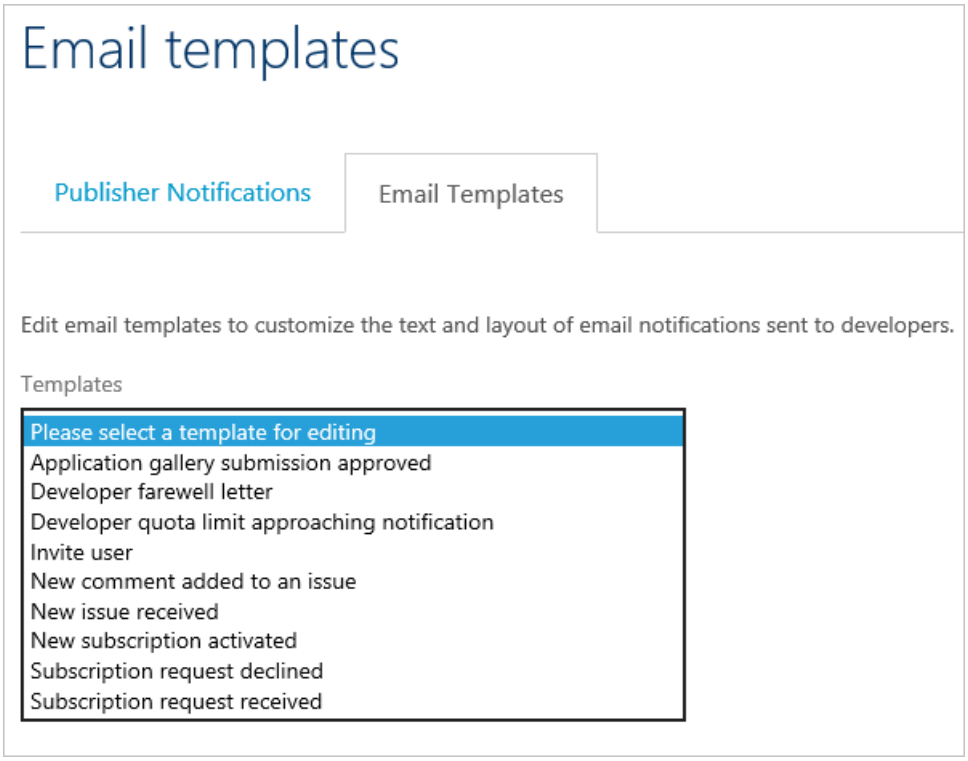
- Application gallery submission approved
- Developer farewell letter
- Developer quota limit approaching notification
- Invite user
- New comment added to an issue
- New issue received
- New subscription activated
- Subscription renewed confirmation
- Subscription request declines
- Subscription request received

These templates can be modified as desired.

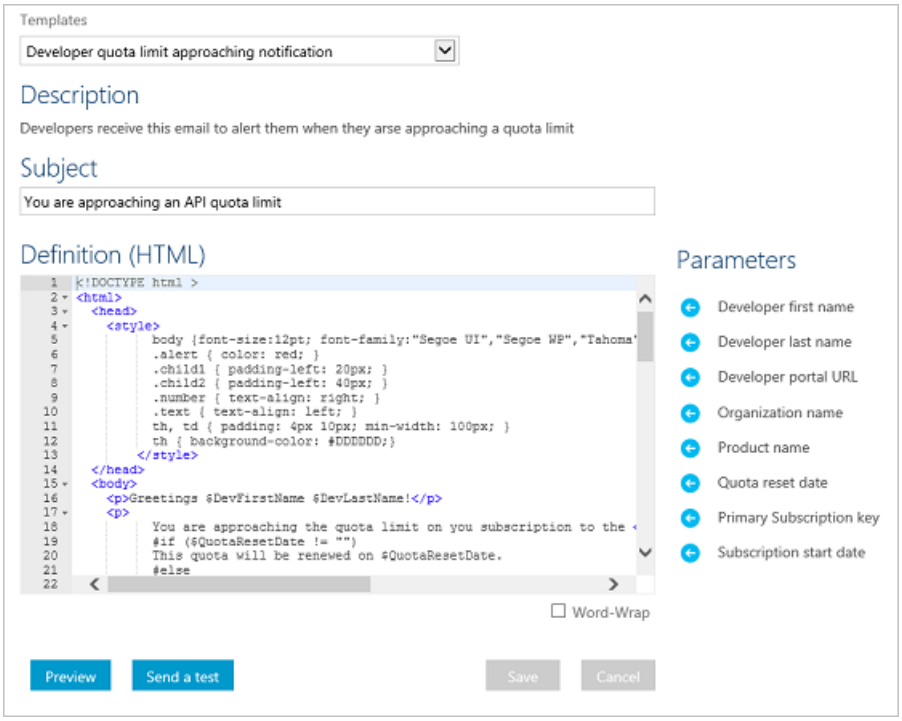
To view and configure the email templates for your API Management instance, click **Notifications** from the **API Management** menu on the left, and select the **Email Templates** tab.



To view or modify a specific template, select it from the **Templates** drop-down list.



Each email template has a subject in plain text, and a body definition in HTML format. Each item can be customized as desired.



The **Parameters** list contains a list of parameters, which when inserted into the subject or body, will be replaced the designated value when the email is sent. To insert a parameter, place the cursor where you wish the parameter to go, and click the arrow to the left of the parameter name.

Click **Preview** or **Send a test** to see how the email will look or send a test email.

> **NOTE**
> The parameters are not replaced with actual values when previewing or sending a test.

To save the changes to the email template, click **Save**, or to cancel the changes click **Cancel**.

How to authorize developer accounts using OAuth 2.0 in Azure API Management

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

Many APIs support OAuth 2.0 to secure the API and ensure that only valid users have access, and they can only access resources to which they're entitled. In order to use Azure API Management's interactive Developer Console with such APIs, the service allows you to configure your service instance to work with your OAuth 2.0 enabled API.

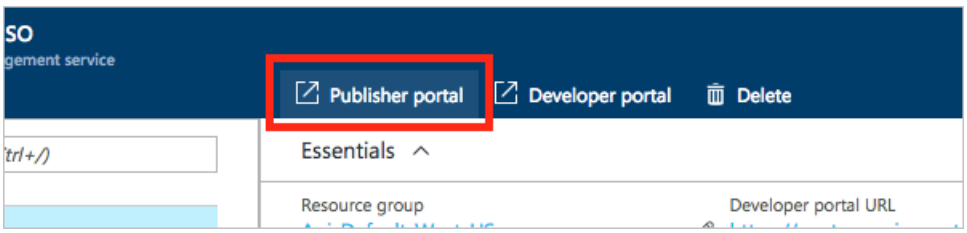
Prerequisites

This guide shows you how to configure your API Management service instance to use OAuth 2.0 authorization for developer accounts, but does not show you how to configure an OAuth 2.0 provider. The configuration for each OAuth 2.0 provider is different, although the steps are similar, and the required pieces of information used in configuring OAuth 2.0 in your API Management service instance are the same. This topic shows examples using Azure Active Directory as an OAuth 2.0 provider.

> **NOTE**
> For more information on configuring OAuth 2.0 using Azure Active Directory, see the WebApp-GraphAPI-DotNet sample.

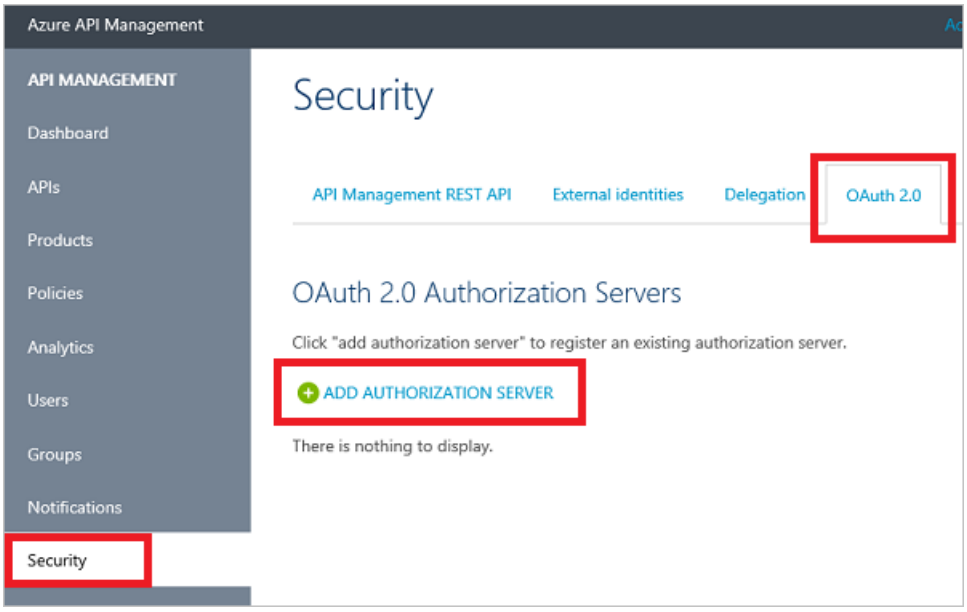
Configure an OAuth 2.0 authorization server in API Management

To get started, click **Publisher portal** in the Azure Portal for your API Management service.

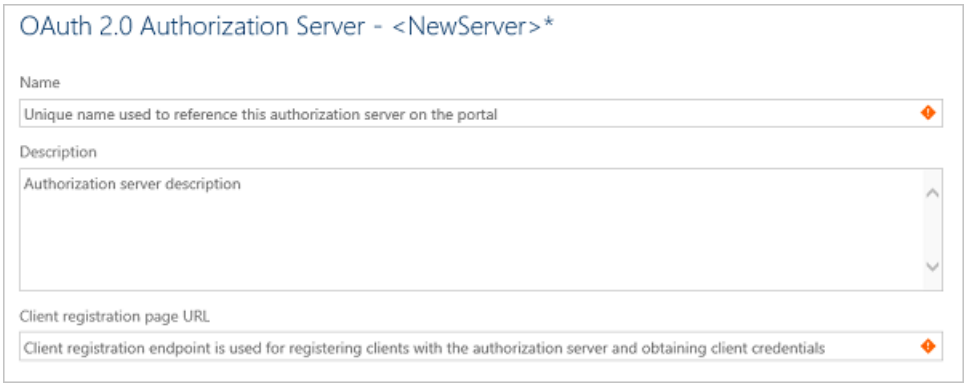


> **NOTE**
> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

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



After clicking **Add authorization server**, the new authorization server form is displayed.

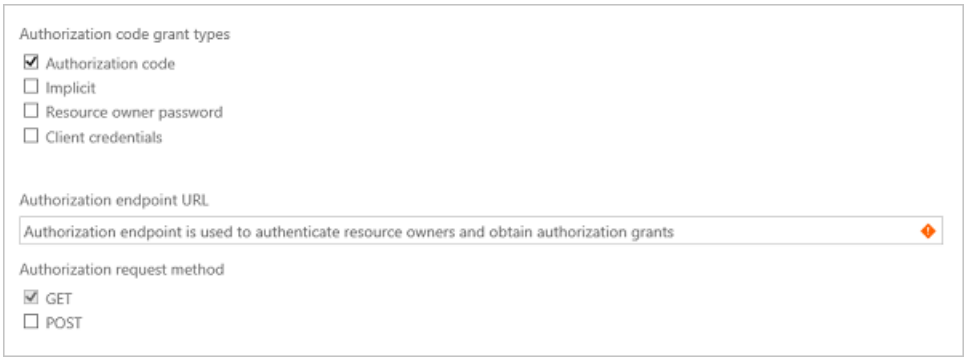


Enter a name and an optional description in the **Name** and **Description** fields.

> **NOTE**
> These fields are used to identify the OAuth 2.0 authorization server within the current API Management service instance and their values do not come from the OAuth 2.0 server.

Enter the **Client registration page URL**. This page is where users can create and manage their accounts, and varies depending on the OAuth 2.0 provider used. The **Client registration page URL** points to the page that users can use to create and configure their own accounts for OAuth 2.0 providers that support user management of accounts. Some organizations do not configure or use this functionality even if the OAuth 2.0 provider supports it. If your OAuth 2.0 provider does not have user management of accounts configured, enter a placeholder URL here such as the URL of your company, or a URL such as `https://placeholder.contoso.com`.

The next section of the form contains the **Authorization code grant types**, **Authorization endpoint URL**, and **Authorization request method** settings.



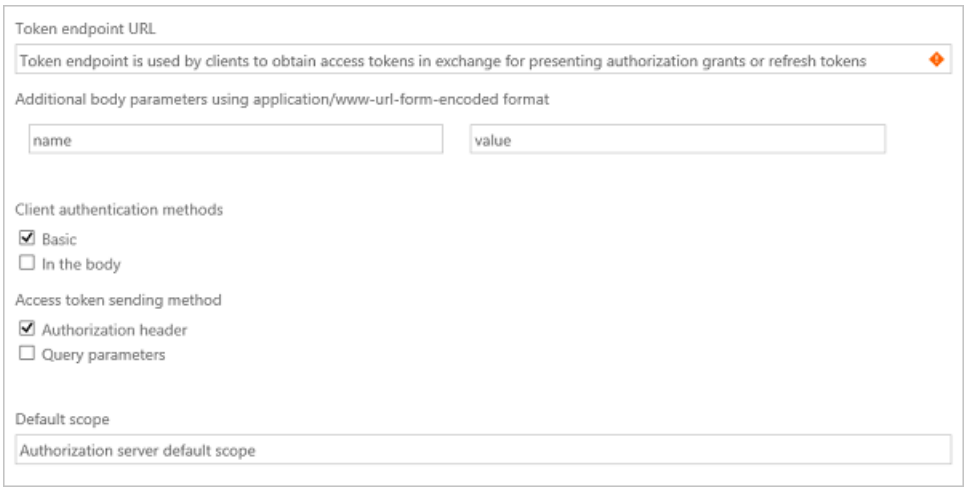
Specify the **Authorization code grant types** by checking the desired types. **Authorization code** is specified by default.

Enter the **Authorization endpoint URL**. For Azure Active Directory, this URL will be similar to the following URL, where `<client_id>` is replaced with the client id that identifies your application to the OAuth 2.0 server.

```
https://login.windows.net/<client_id>/oauth2/authorize
```

The **Authorization request method** specifies how the authorization request is sent to the OAuth 2.0 server. By default **GET** is selected.

The next section is where the **Token endpoint URL**, **Client authentication methods**, **Access token sending method**, and **Default scope** are specified.



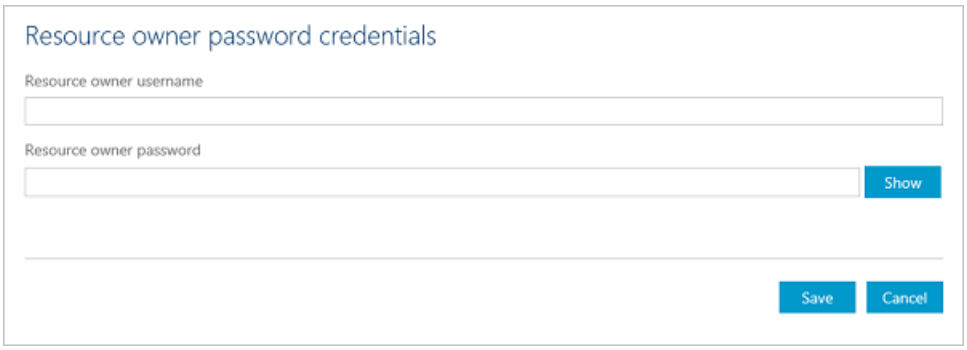
For an Azure Active Directory OAuth 2.0 server, the **Token endpoint URL** will have the following format, where `<APPID>` has the format of `yourapp.onmicrosoft.com`.

```
https://login.windows.net/<APPID>/oauth2/token
```

The default setting for **Client authentication methods** is **Basic**, and **Access token sending method** is **Authorization header**. These values are configured on this section of the form, along with the **Default scope**.

The **Client credentials** section contains the **Client ID** and **Client secret**, which are obtained during the creation and configuration process of your OAuth 2.0 server. Once the **Client ID** and **Client secret** are specified, the **redirect_uri** for the **authorization code** is generated. This URI is used to configure the reply URL in your OAuth 2.0 server configuration.

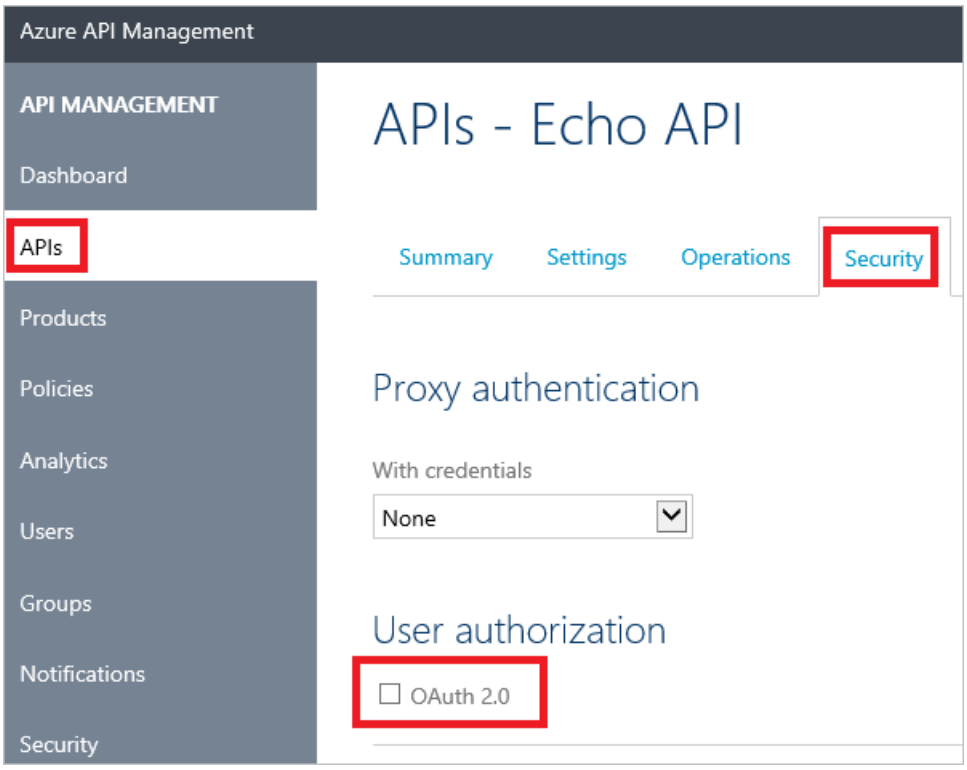
If **Authorization code grant types** is set to **Resource owner password**, the **Resource owner password credentials** section is used to specify those credentials; otherwise you can leave it blank.



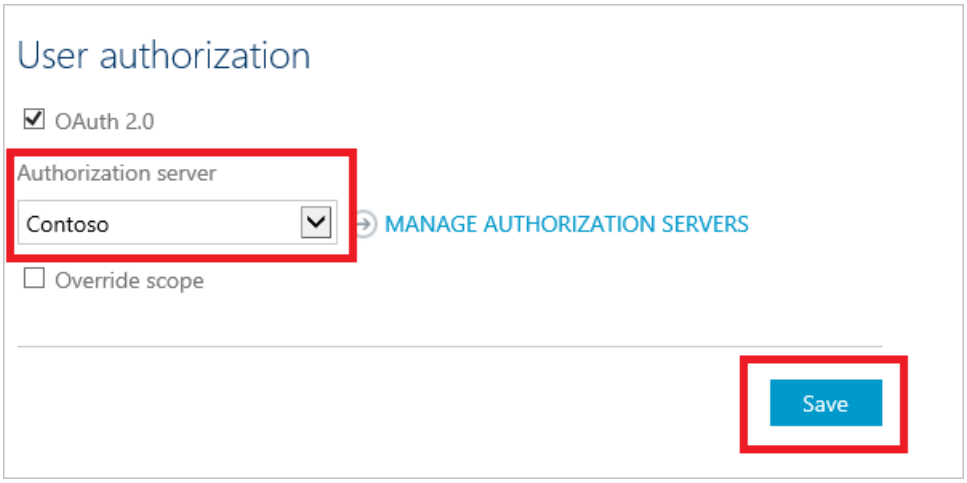
Once the form is complete, click **Save** to save the API Management OAuth 2.0 authorization server configuration. Once the server configuration is saved, you can configure APIs to use this configuration, as shown in the next section.

Configure an API to use OAuth 2.0 user authorization

Click **APIs** from the **API Management** menu on the left, click the name of the desired API, click **Security**, and then check the box for **OAuth 2.0**.



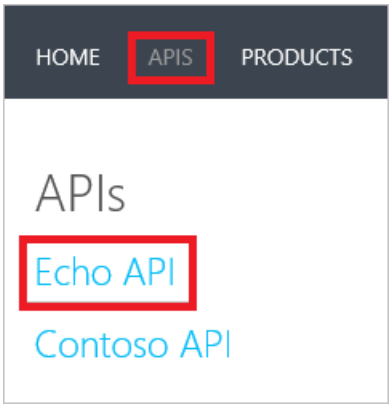
Select the desired **Authorization server** from the drop-down list, and click **Save**.



Test the OAuth 2.0 user authorization in the Developer Portal

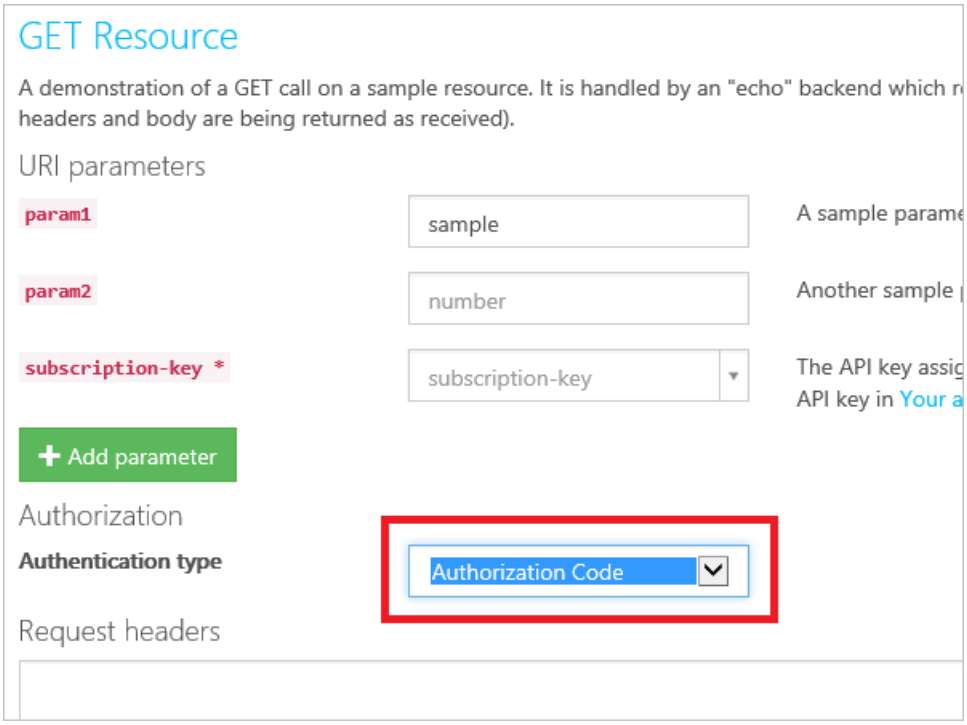
Once you have configured your OAuth 2.0 authorization server and configured your API to use that server, you can test it by going to the Developer Portal and calling an API. Click **Developer portal** in the top right menu.

Click **APIs** in the top menu and select **Echo API**.



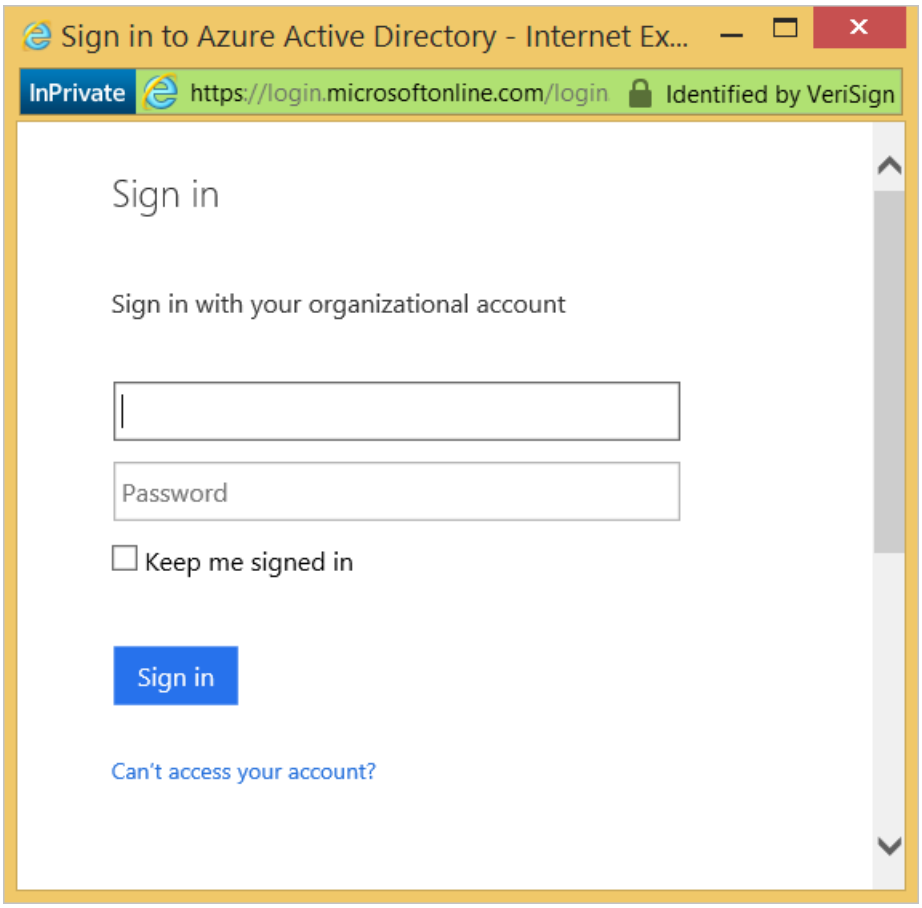
> **NOTE**
> If you have only one API configured or visible to your account, then clicking APIs takes you directly to the operations for that API.

Select the **GET Resource** operation, click **Open Console**, and then select **Authorization code** from the drop-down.

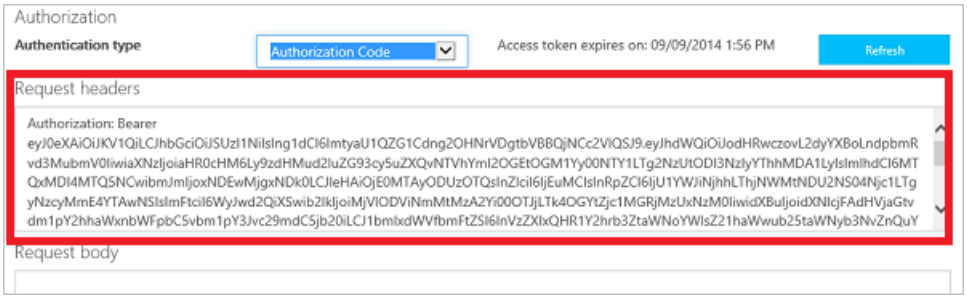


When **Authorization code** is selected, a pop-up window is displayed with the sign-in form of the OAuth 2.0 provider. In this example the sign-in form is provided by Azure Active Directory.

> **NOTE**
>
> If you have pop-ups disabled you will be prompted to enable them by the browser. After you enable them, select **Authorization code** again and the sign-in form will be displayed.



Once you have signed in, the **Request headers** are populated with an `Authorization : Bearer` header that authorizes the request.



At this point you can configure the desired values for the remaining parameters, and submit the request.

Next steps

For more information about using OAuth 2.0 and API Management, see the following video and accompanying article.

How to customize the Azure API Management developer portal using templates

11/15/2016 • 4 min to read • Edit on GitHub

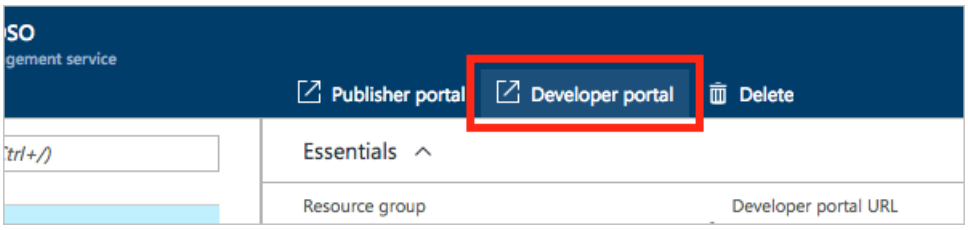
Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

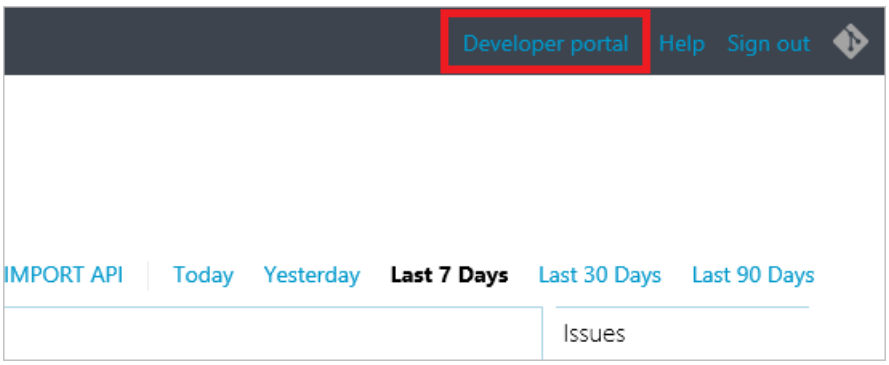
Azure API Management provides several customization features to allow administrators to customize the look and feel of the developer portal, as well as customize the content of the developer portal pages using a set of templates that configure the content of the pages themselves. Using DotLiquid syntax, and a provided set of localized string resources, icons, and page controls, you have great flexibility to configure the content of the pages as you see fit using these templates.

Developer portal templates overview

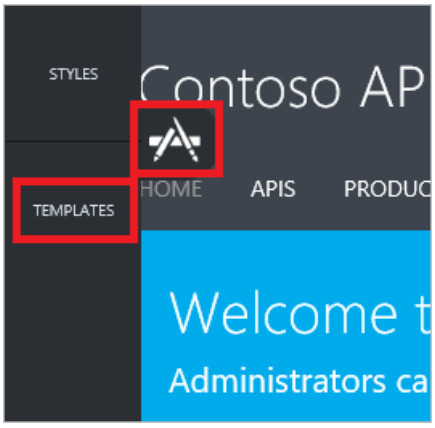
Developer portal templates are managed in the developer portal by administrators of the API Management service instance. To manage developer templates, navigate to your API Management service instance in the Azure Portal and click **Developer portal** from the toolbar.



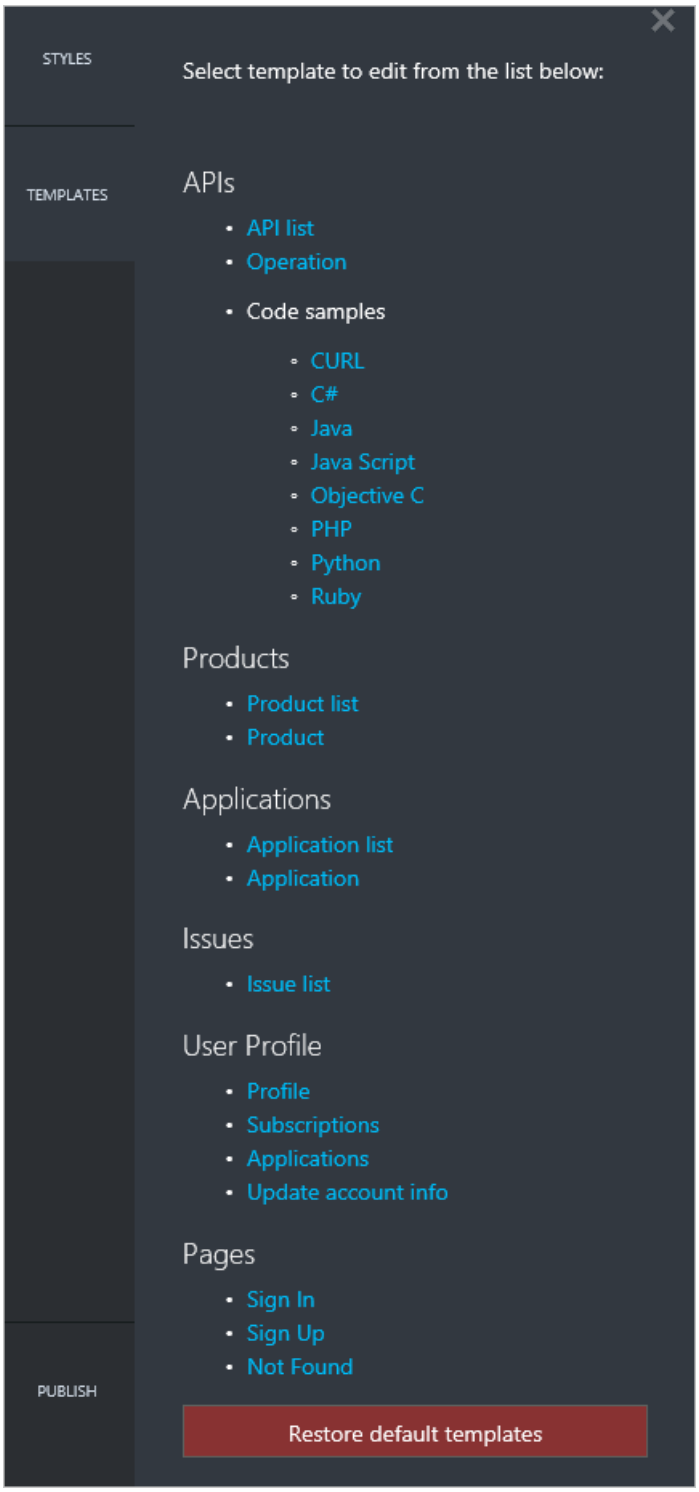
If you are already in the publisher portal, you can access the developer portal by clicking **Developer portal**.



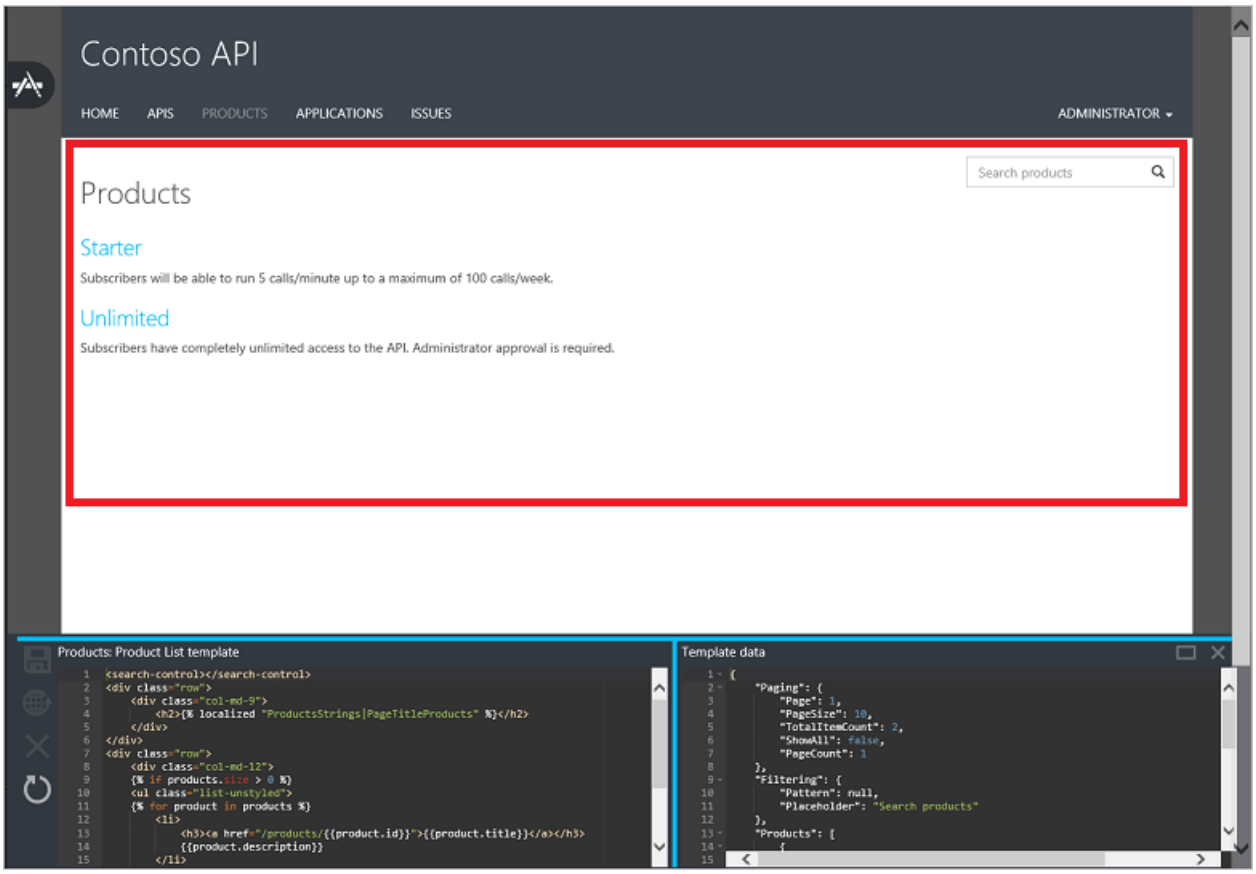
To access the developer portal templates, click the customize icon on the left to display the customization menu, and click **Templates**.



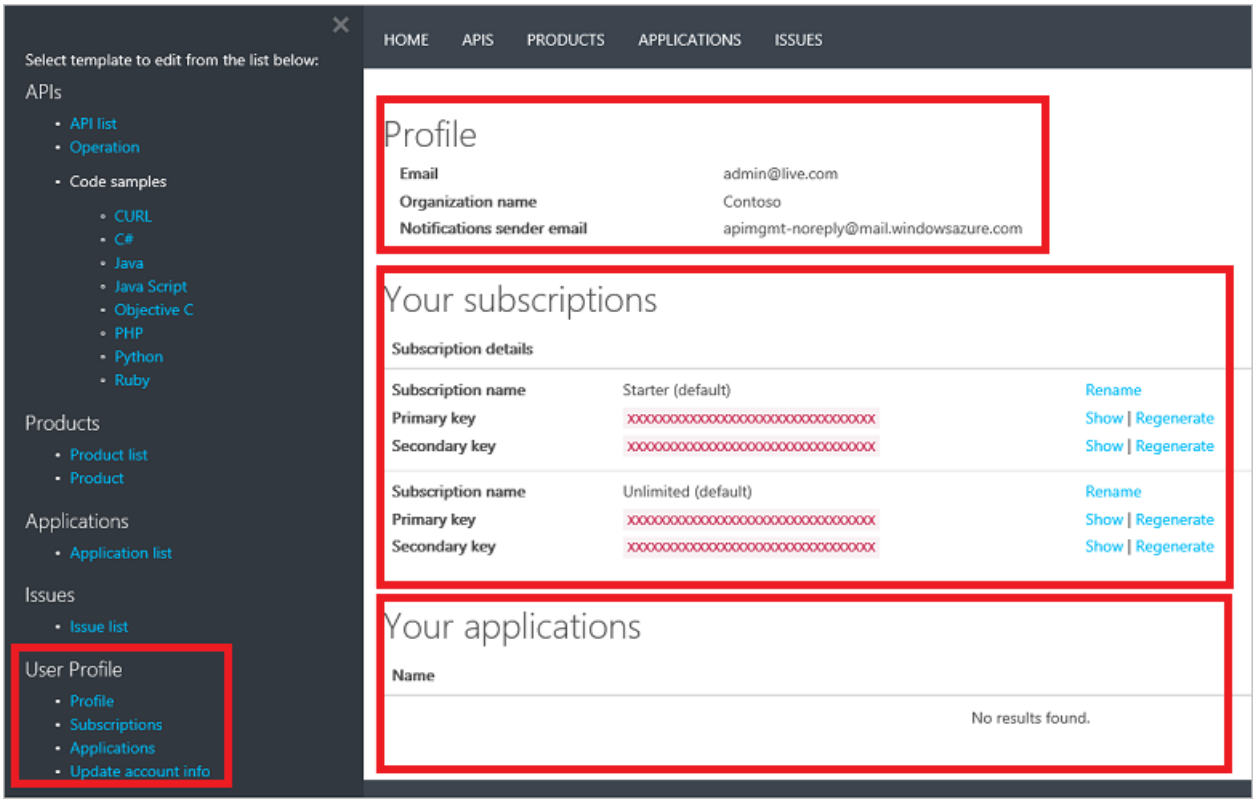
The templates list displays several categories of templates covering the different pages in the developer portal. Each template is different, but the steps to edit them and publish the changes are the same. To edit a template, click the name of the template.



Clicking a template takes you to the developer portal page that is customizable by that template. In this example the **Product list** template is displayed. The **Product list** template controls the area of the screen indicated by the red rectangle.



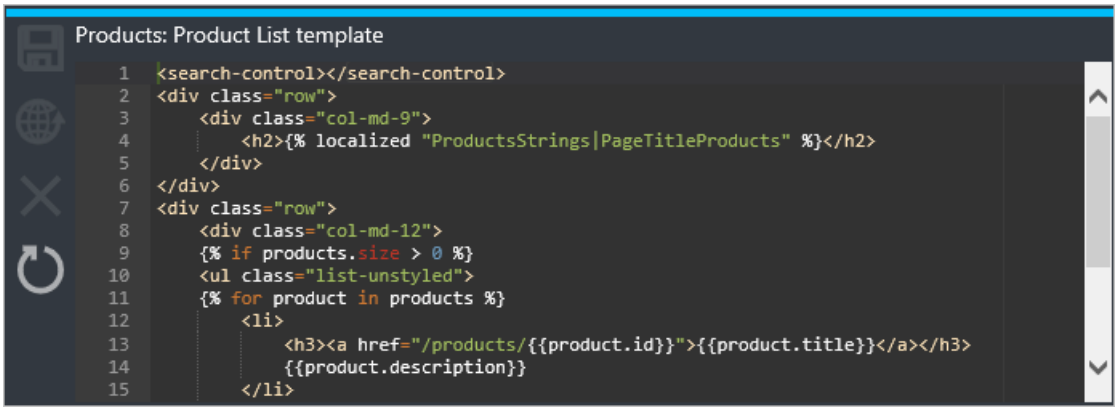
Some templates, like the **User Profile** templates, customize different parts of the same page.



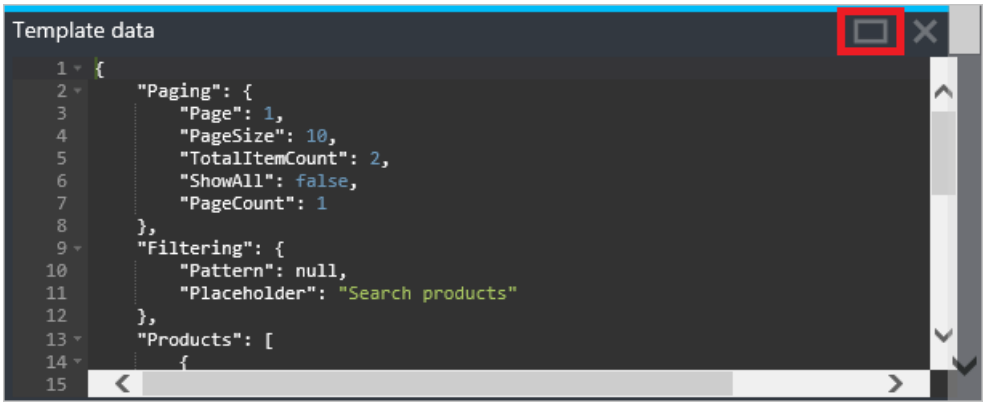
The editor for each developer portal template has two sections displayed at the bottom of the page. The left-hand side displays the editing pane for the template, and the right-hand side displays the data model for the template.

The template editing pane contains the markup that controls the appearance and behavior of the corresponding

page in the developer portal. The markup in the template uses the DotLiquid syntax. One popular editor for DotLiquid is DotLiquid for Designers. Any changes made to the template during editing are displayed in real-time in the browser, but are not visible to your customers until you save and publish the template.



The **Template data** pane provides a guide to the data model for the entities that are available for use in a particular template. It provides this guide by displaying the live data that are currently displayed in the developer portal. You can expand the template panes by clicking the rectangle in the upper-right corner of the **Template data** pane.



In the previous example there are two products displayed in the developer portal that were retrieved from the data displayed in the **Template data** pane, as shown in the following example.

```
{
    "Paging": {
        "Page": 1,
        "PageSize": 10,
        "TotalItemCount": 2,
        "ShowAll": false,
        "PageCount": 1
    },
    "Filtering": {
        "Pattern": null,
        "Placeholder": "Search products"
    },
    "Products": [
        {
            "Id": "56ec64c380ed850042060001",
            "Title": "Starter",
            "Description": "Subscribers will be able to run 5 calls/minute up to a maximum of 100 calls/week.",
            "Terms": "",
            "ProductState": 1,
            "AllowMultipleSubscriptions": false,
            "MultipleSubscriptionsCount": 1
        },
        {
            "Id": "56ec64c380ed850042060002",
            "Title": "Unlimited",
            "Description": "Subscribers have completely unlimited access to the API. Administrator approval is
required.",
            "Terms": null,
            "ProductState": 1,
            "AllowMultipleSubscriptions": false,
            "MultipleSubscriptionsCount": 1
        }
    ]
}
```


The markup in the **Product list** template processes the data to provide the desired output by iterating through the collection of products to display information and a link to each individual product. Note the `<search-control>` and `<page-control>` elements in the markup. These control the display of the searching and paging controls on the page. `ProductsStrings|PageTitleProducts` is a localized string reference that contains the `h2` header text for the page. For a list of string resources, page controls, and icons available for use in developer portal templates, see API Management developer portal templates reference.

```
<search-control></search-control>
<div class="row">
    <div class="col-md-9">
        <h2>{% localized "ProductsStrings|PageTitleProducts" %}</h2>
    </div>
</div>
<div class="row">
    <div class="col-md-12">
    {% if products.size > 0 %}
    <ul class="list-unstyled">
    {% for product in products %}
        <li>
            <h3><a href="/products/{{product.id}}">{{product.title}}</a></h3>
            {{product.description}}
        </li>
    {% endfor %}
    </ul>
    <paging-control></paging-control>
    {% else %}
    {% localized "CommonResources|NoItemsToDisplay" %}
    {% endif %}
    </div>
</div>
```

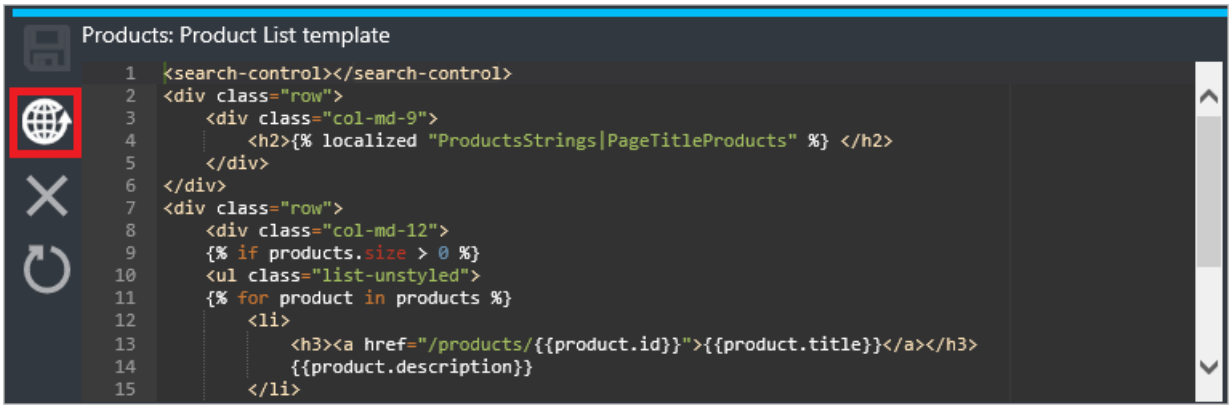
To save a template

To save a template, click save in the template editor.

Saved changes are not live in the developer portal until they are published.

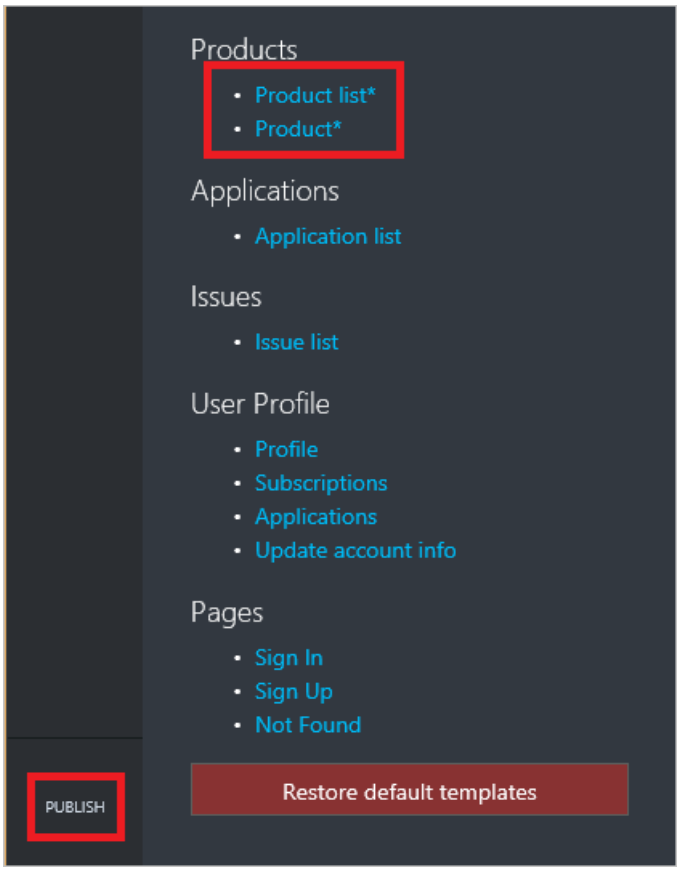
To publish a template

Saved templates can be published either individually, or all together. To publish an individual template, click publish in the template editor.

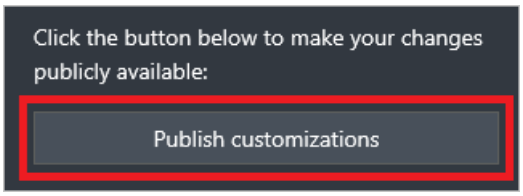


Click **Yes** to confirm and make the template live on the developer portal.

To publish all currently unpublished template versions, click **Publish** in the templates list. Unpublished templates are designated by an asterisk following the template name. In this example, the **Product list** and **Product** templates are being published.



Click **Publish customizations** to confirm.

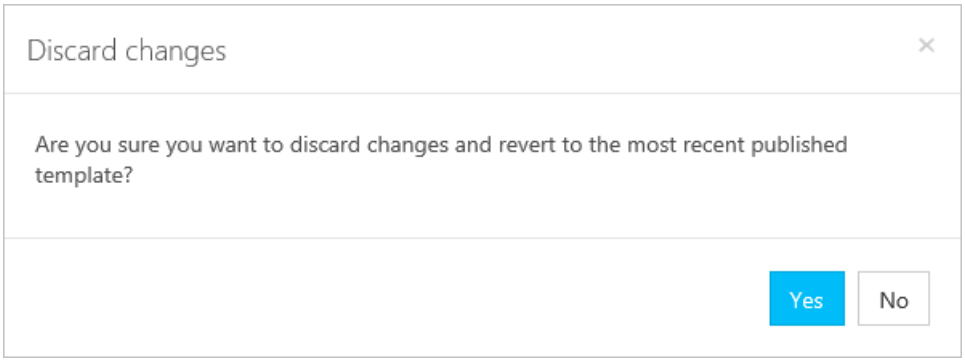


Newly published templates are effective immediately in the developer portal.

To revert a template to the previous version

To revert a template to the previous published version, click revert in the template editor.

Click **Yes** to confirm.



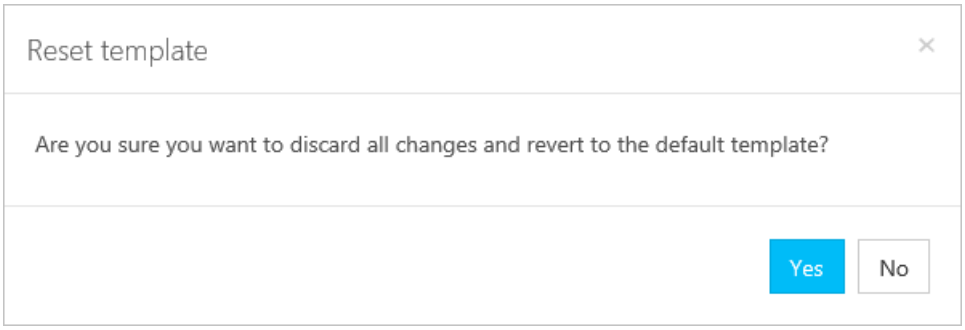
The previously published version of a template is live in the developer portal once the revert operation is complete.

To restore a template to the default version

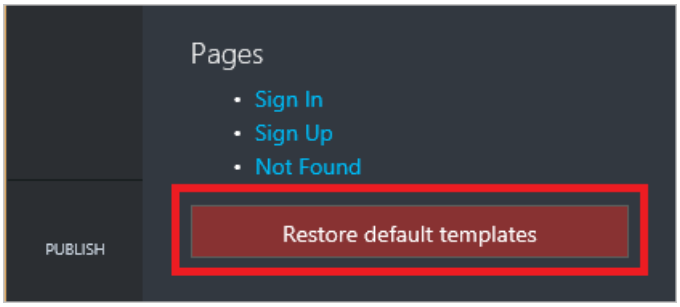
Restoring templates to their default version is a two-step process. First the templates must be restored, and then the restored versions must be published.

To restore a single template to the default version click restore in the template editor.

Click **Yes** to confirm.



To restore all templates to their default versions, click **Restore default templates** on the template list.



The restored templates must then be published individually or all at once by following the steps in To publish a template.

Developer portal templates reference

For reference information for developer portal templates, string resources, icons, and page controls, see API Management developer portal templates reference.

Watch a video overview

Watch the following video to see how to add a discussion board and ratings to the API and operation pages in the developer portal using templates.

How to create and use groups to manage developer accounts in Azure API Management

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

In API Management, groups are used to manage the visibility of products to developers. Products are first made visible to groups, and then developers in those groups can view and subscribe to the products that are associated with the groups.

API Management has the following immutable system groups.

- **Administrators** - Azure subscription administrators are members of this group. Administrators manage API Management service instances, creating the APIs, operations, and products that are used by developers.
- **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.

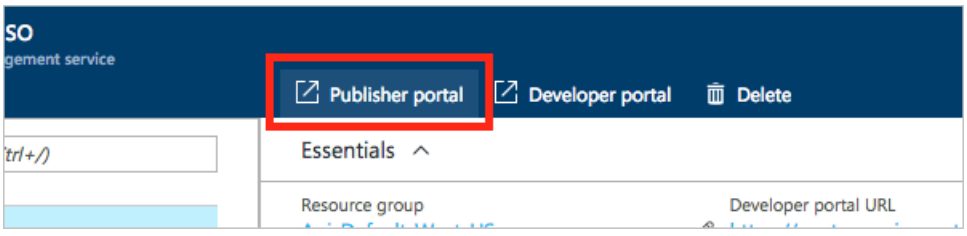
In addition to these system groups, administrators can create custom groups or leverage external groups in associated Azure Active Directory tenants. Custom and external groups can be used alongside system groups in giving developers visibility and access to API products. For example, you could create one custom group for developers affiliated with a specific partner organization and allow them access to the APIs from a product containing relevant APIs only. A user can be a member of more than one group.

This guide shows how administrators of an API Management instance can add new groups and associate them with products and developers.

> **NOTE**
> In addition to creating and managing groups in the publisher portal, you can create and manage your groups using the API Management REST API Group entity.

Create a group

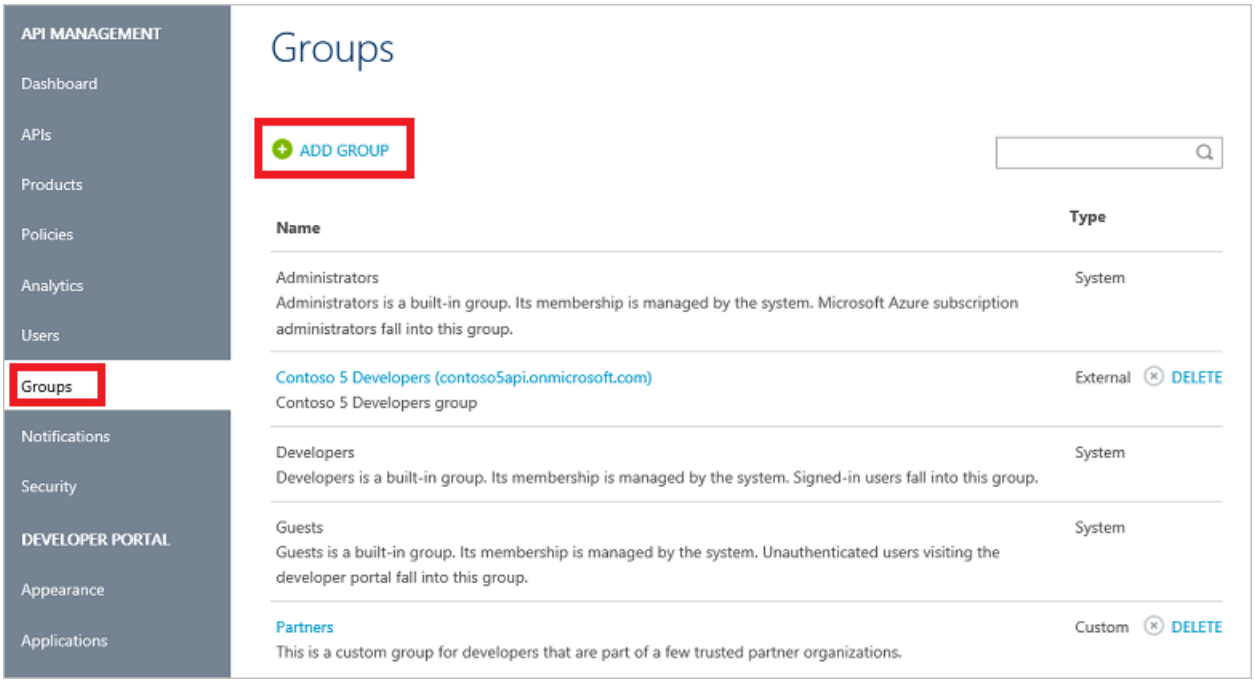
To create a new group, click **Publisher portal** in the Azure Portal for your API Management service. This takes you to the API Management publisher portal.



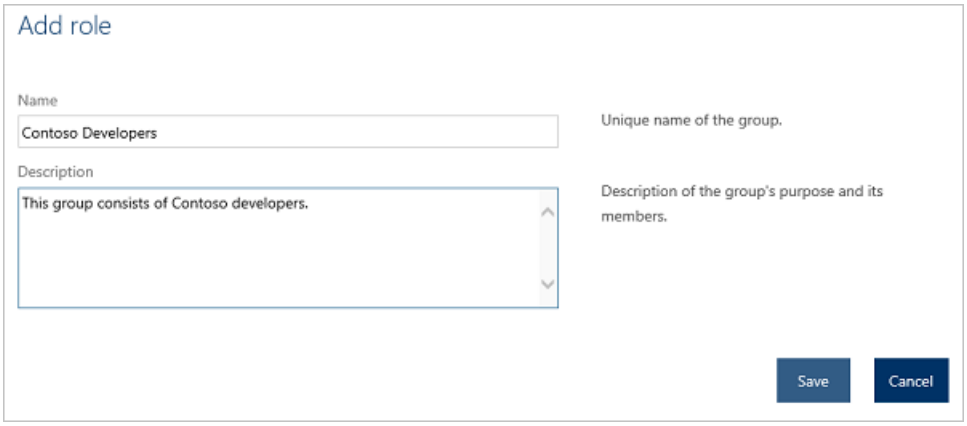
> If you have not yet created an API Management service instance, see Create an API Management service

> instance in the Get started with Azure API Management tutorial.

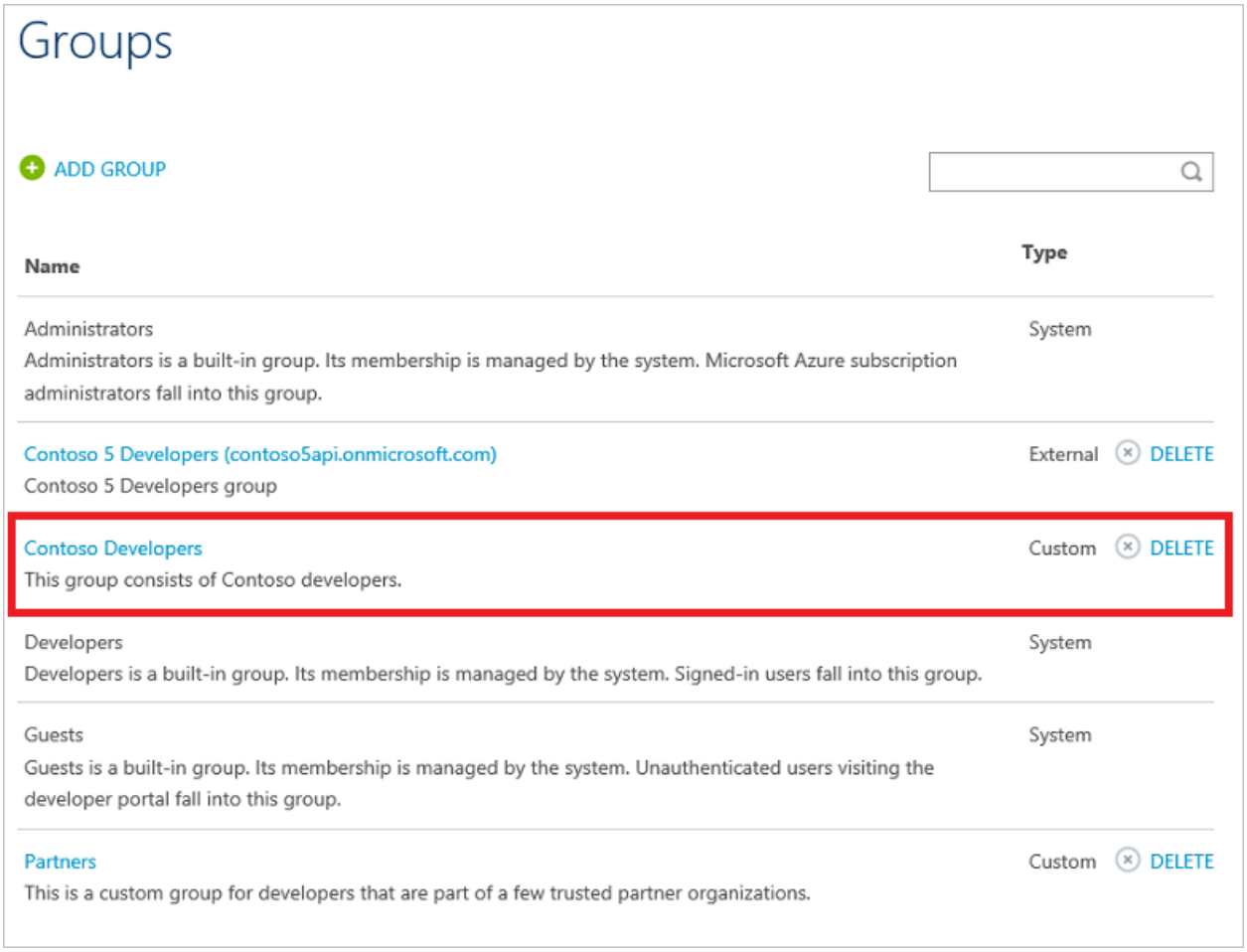
Click **Groups** from the **API Management** menu on the left, and then click **Add Group**.



Enter a unique name for the group and an optional description, and click **Save**.



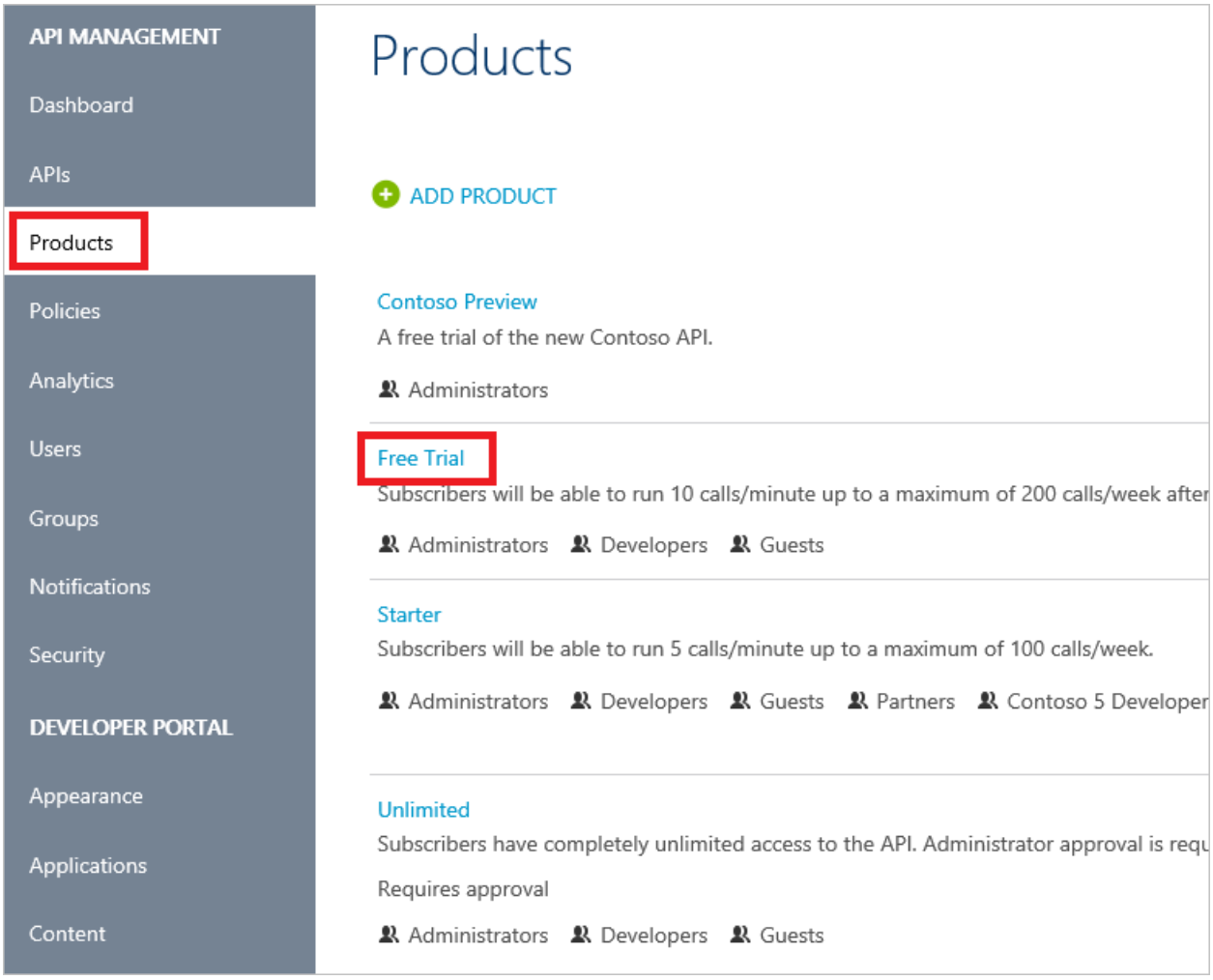
The new group is displayed in the groups tab. To edit the **Name** or **Description** of the group, click the name of the group in the list. To delete the group, click **Delete**.



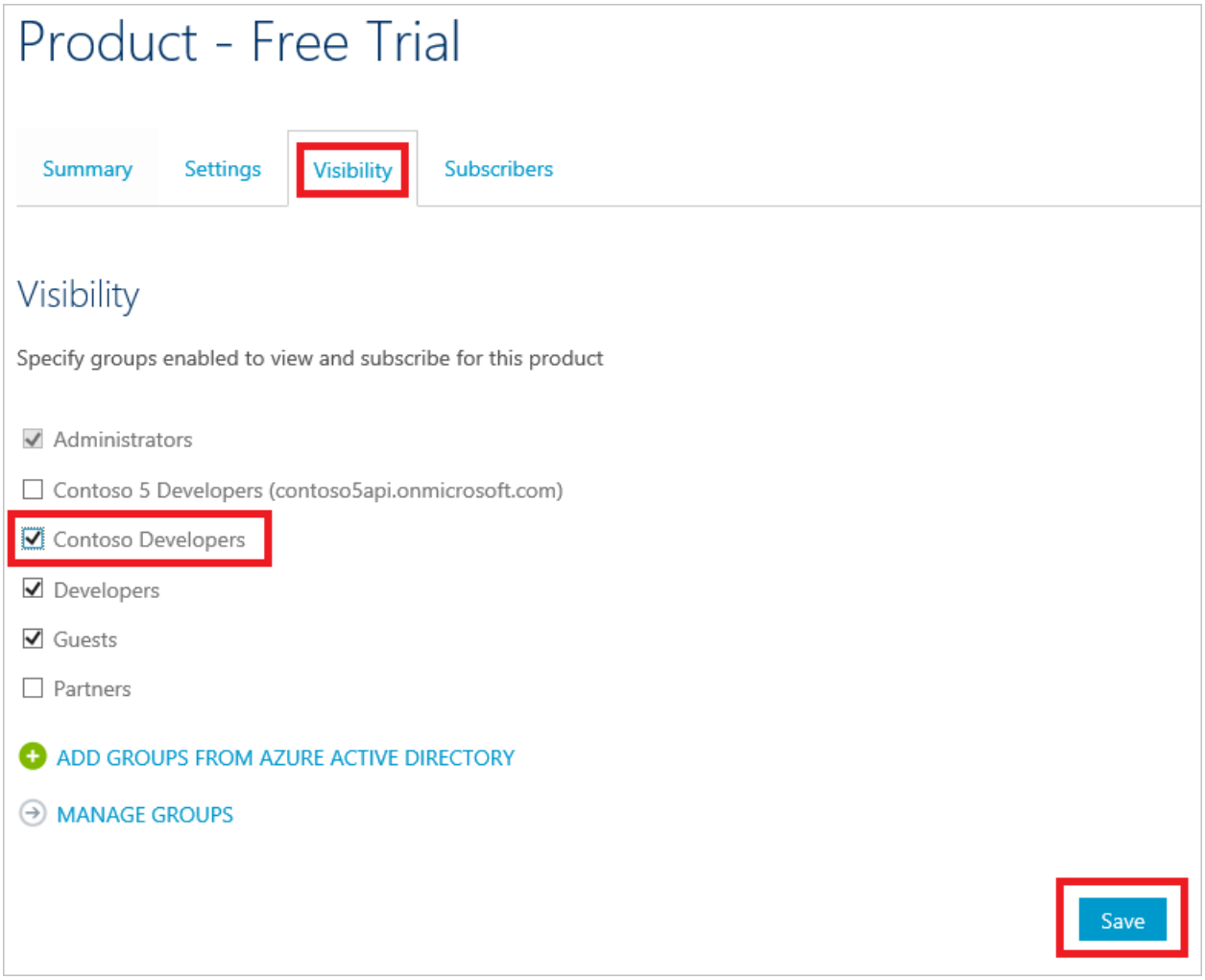
Now that the group is created, it can be associated with products and developers.

Associate a group with a product

To associate a group with a product, click **Products** from the **API Management** menu on the left, and then click the name of the desired product.



Select the **Visibility** tab to add and remove groups, and to view the current groups for the product. To add or remove groups, check or uncheck the checkboxes for the desired groups and click **Save**.



NOTE

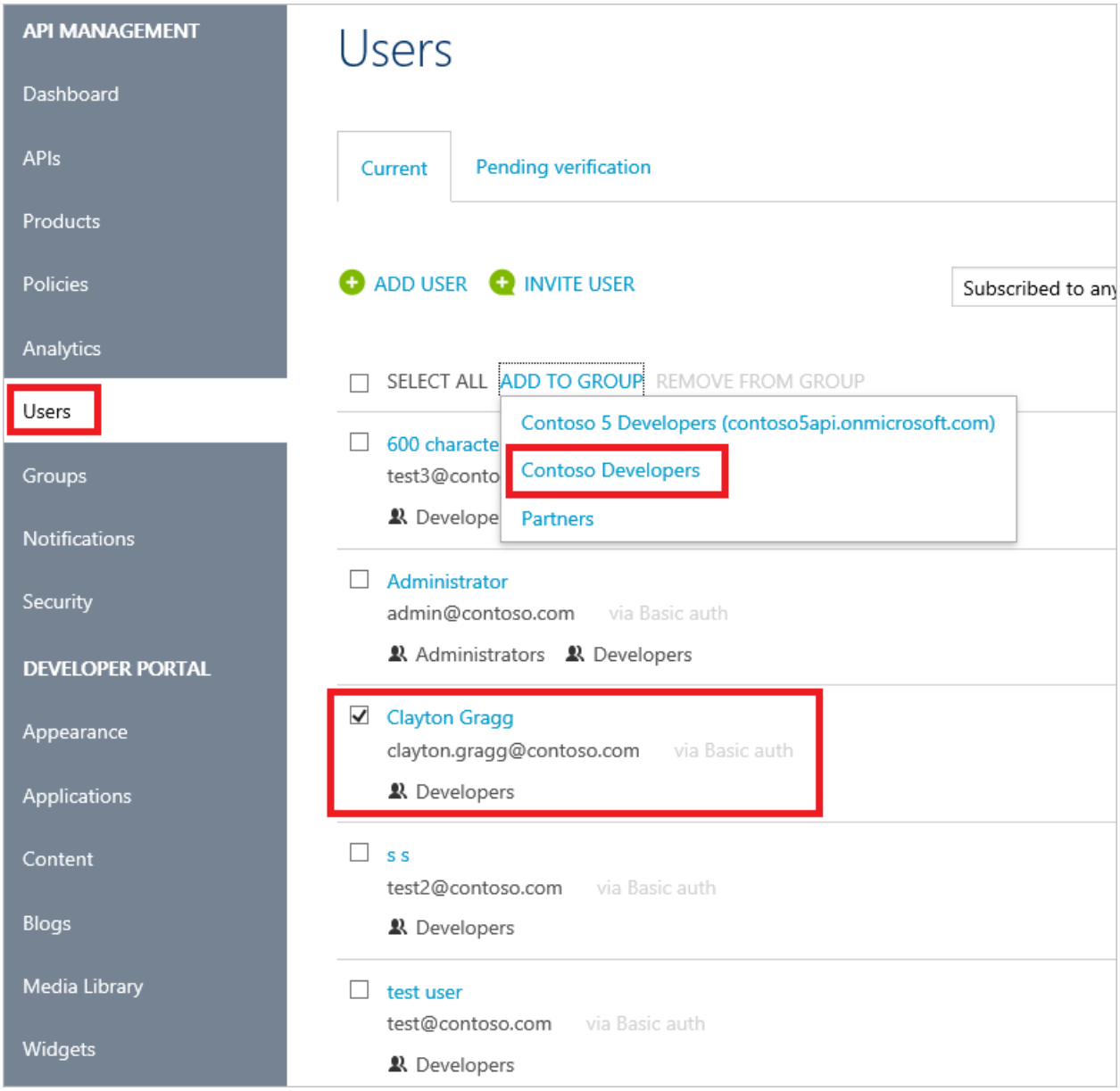
To add Azure Active Directory groups, see How to authorize developer accounts using Azure Active Directory in Azure API Management.

To configure groups from the **Visibility** tab for a product, click **Manage Groups**.

Once a product is associated with a group, developers in that group can view and subscribe to the product.

Associate groups with developers

To associate groups with developers, click **Users** from the **API Management** menu on the left, and then check the box beside the developers you wish to associate with a group.



Once the desired developers are checked, click the desired group in the **Add to Group** drop-down. Developers can be removed from groups by using the **Remove from Group** drop-down.

Once the association is added between the developer and the group, you can view it in the **Users** tab.

Next steps

- Once a developer is added to a group, they can view and subscribe to the products associated with that group. For more information, see How create and publish a product in Azure API Management,
- In addition to creating and managing groups in the publisher portal, you can create and manage your groups using the API Management REST API Group entity.

How to deploy an Azure API Management service instance to multiple Azure regions

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft) • v-aljenk • Jamie Strusz

API Management supports multi-region deployment which enables API publishers to distribute a single API management service across any number of desired Azure regions. This helps reduce request latency perceived by geographically distributed API consumers and also improves service availability if one region goes offline.

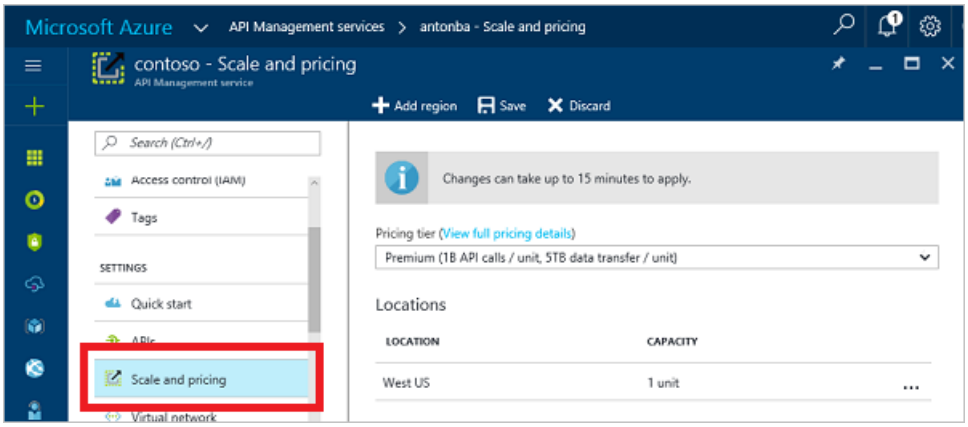
When an API Management service is created initially, it contains only one unit and resides in a single Azure region, which is designated as the Primary Region. Additional regions can be easily added through the Azure Portal. An API Management gateway server is deployed to each region and call traffic will be routed to the closest gateway. If a region goes offline, the traffic is automatically re-directed to the next closest gateway.

> **IMPORTANT**
> Multi-region deployment is only available in the **Premium** tier.

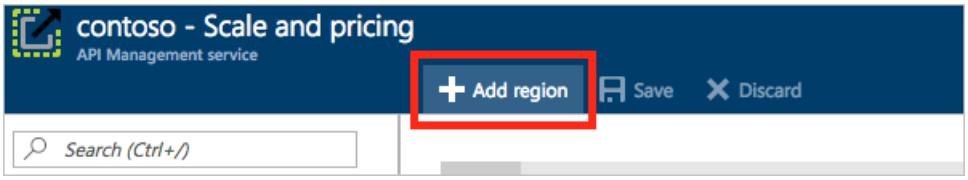
Deploy an API Management service instance to a new region

> **NOTE**
> If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.

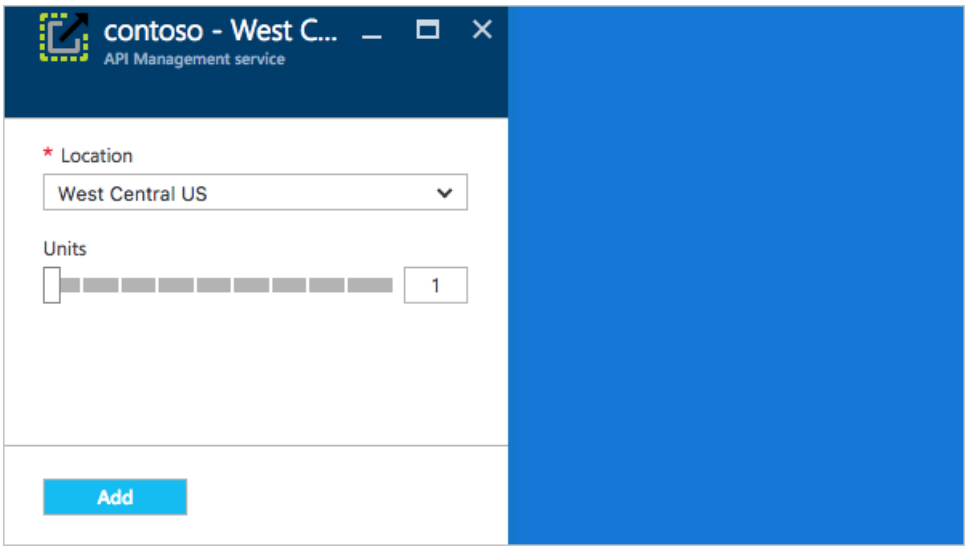
In the Azure Portal navigate to the **Scale and pricing** page for your API Management service instance.



To deploy to a new region, click on **+ Add region** from the toolbar.



Select the location from the drop-down list and set the number of units for with the slider.

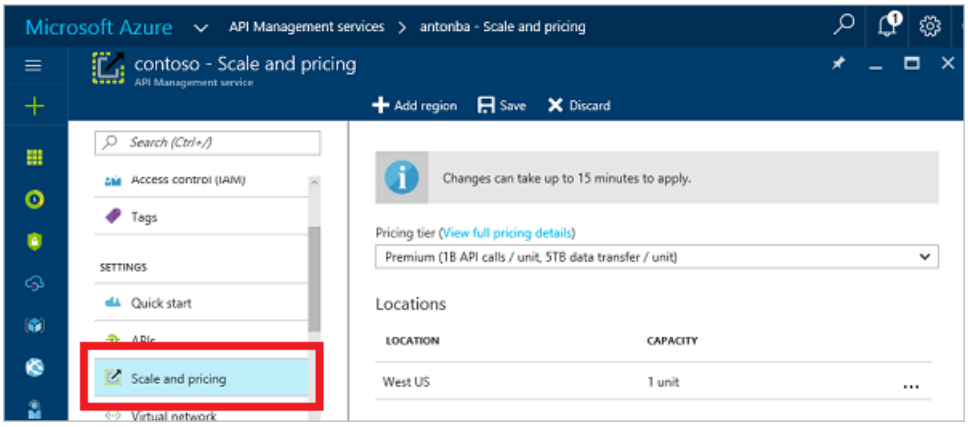


Click **Add** to place your selection in the Locations table.

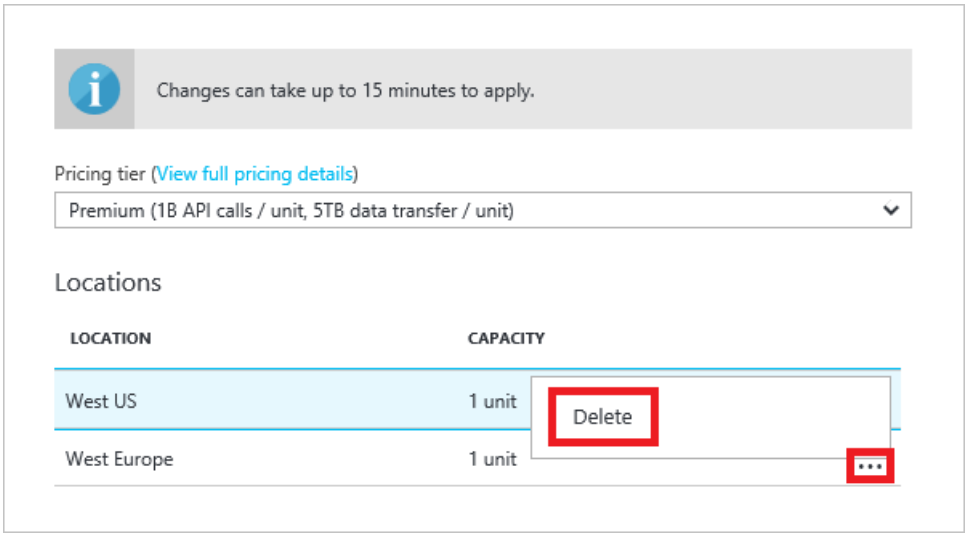
Repeat this process until you have all locations configured and click **Save** from the toolbar to start the deployment process.

Delete an API Management service instance from a location

In the Azure Portal navigate to the **Scale and pricing** page for your API Management service instance.



For the location you would like to remove open the context menu using the ... button at the right end of the table. Select the **Delete** option.



Confirm the deletion and click **Save** to apply the changes.

How to implement disaster recovery using service backup and restore in Azure API Management

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

steved0x • Ralph Squillace • Joseph Molnar • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Erik

---

By choosing to publish and manage your APIs via Azure API Management you are taking advantage of many fault tolerance and infrastructure capabilities that you would otherwise have to design, implement, and manage. The Azure platform mitigates a large fraction of potential failures at a fraction of the cost.

To recover from availability problems affecting the region where your API Management service is hosted you should be ready to reconstitute your service in a different region at any time. Depending on your availability goals and recovery time objective you might want to reserve a backup service in one or more regions and try to maintain their configuration and content in sync with the active service. The service backup and restore feature provides the necessary building block for implementing your disaster recovery strategy.

This guide shows how to authenticate Azure Resource Manager requests, and how to backup and restore your API Management service instances.

> **NOTE**
>
> The process for backing up and restoring an API Management service instance for disaster recovery can also be used for replicating API Management service instances for scenarios such as staging.
>
> Note that each backup expires after 7 days. If you attempt to restore a backup after the 7 day expiration period has expired, the restore will fail with a `Cannot restore: backup expired` message.

Authenticating Azure Resource Manager requests

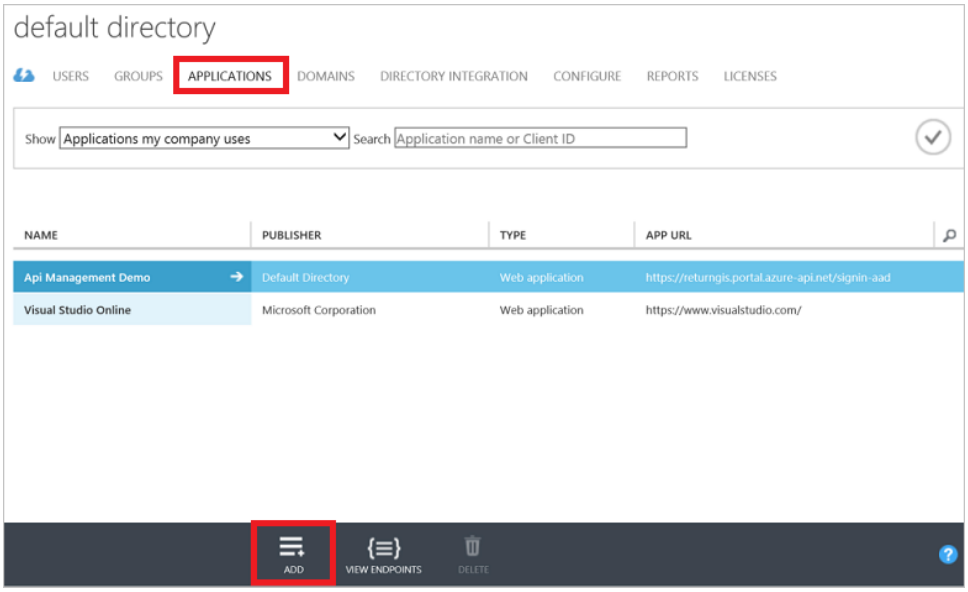
> **IMPORTANT**
>
> The REST API for backup and restore uses Azure Resource Manager and has a different authentication mechanism than the REST APIs for managing your API Management entities. The steps in this section describe how to authenticate Azure Resource Manager requests. For more information, see Authenticating Azure Resource Manager requests.

All of the tasks that you do on resources using the Azure Resource Manager must be authenticated with Azure Active Directory using the following steps.

- Add an application to the Azure Active Directory tenant.
- Set permissions for the application that you added.
- Get the token for authenticating requests to Azure Resource Manager.

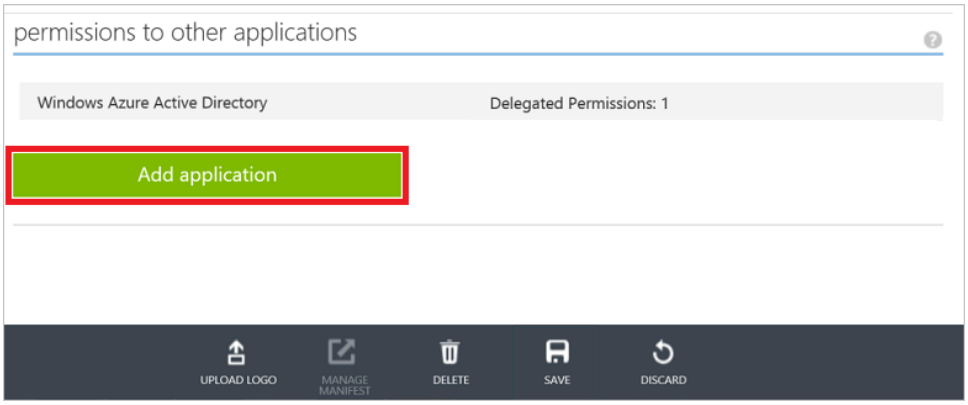
The first step is to create an Azure Active Directory application. Log into the Azure Classic Portal using the subscription that contains your API Management service instance and navigate to the **Applications** tab for your default Azure Active Directory.

> **NOTE**
>
> If the Azure Active Directory default directory is not visible to your account, contact the administrator of the Azure subscription to grant the required permissions to your account. For information on locating your default directory, see "Locate your default directory in the Azure classic portal" in Creating a Work or School identity in Azure Active Directory to use with Windows VMs.

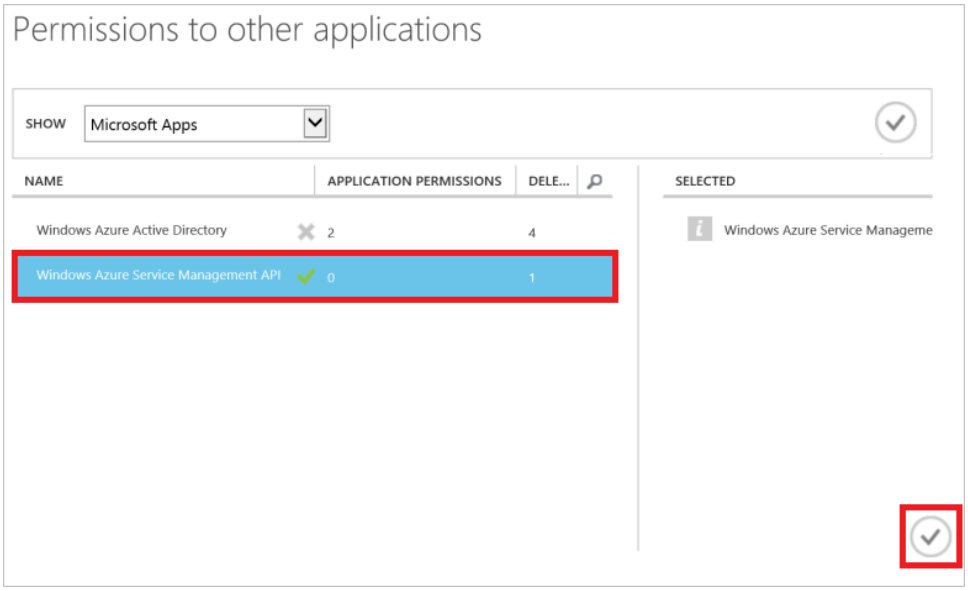


Click **Add**, **Add an application my organization is developing**, and choose **Native client application**. Enter a descriptive name, and click the next arrow. Enter a placeholder URL such as `http://resources` for the **Redirect URI**, as it is a required field, but the value is not used later. Click the check box to save the application.

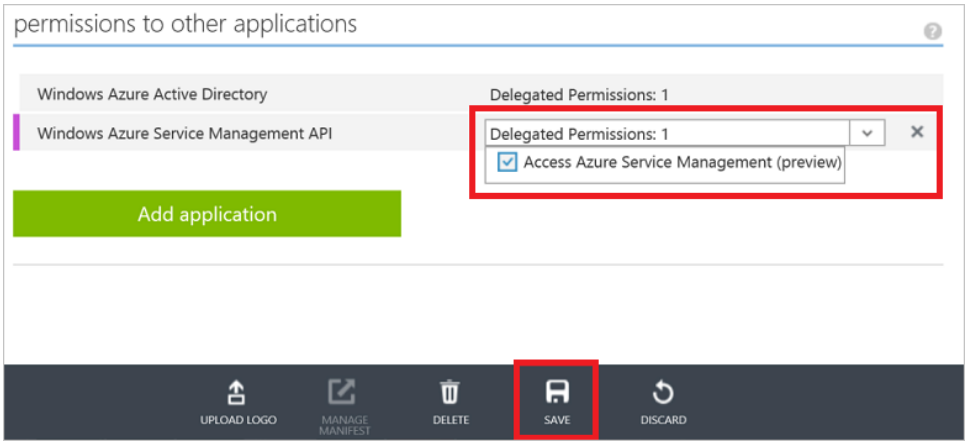
Once the application is saved, click **Configure**, scroll down to the **permissions to other applications** section, and click **Add application**.



Select **Windows Azure Service Management API** and click the checkbox to add the application.



Click **Delegated Permissions** beside the newly added **Windows Azure Service Management API** application, check the box for **Access Azure Service Management (preview)**, and click **Save**.

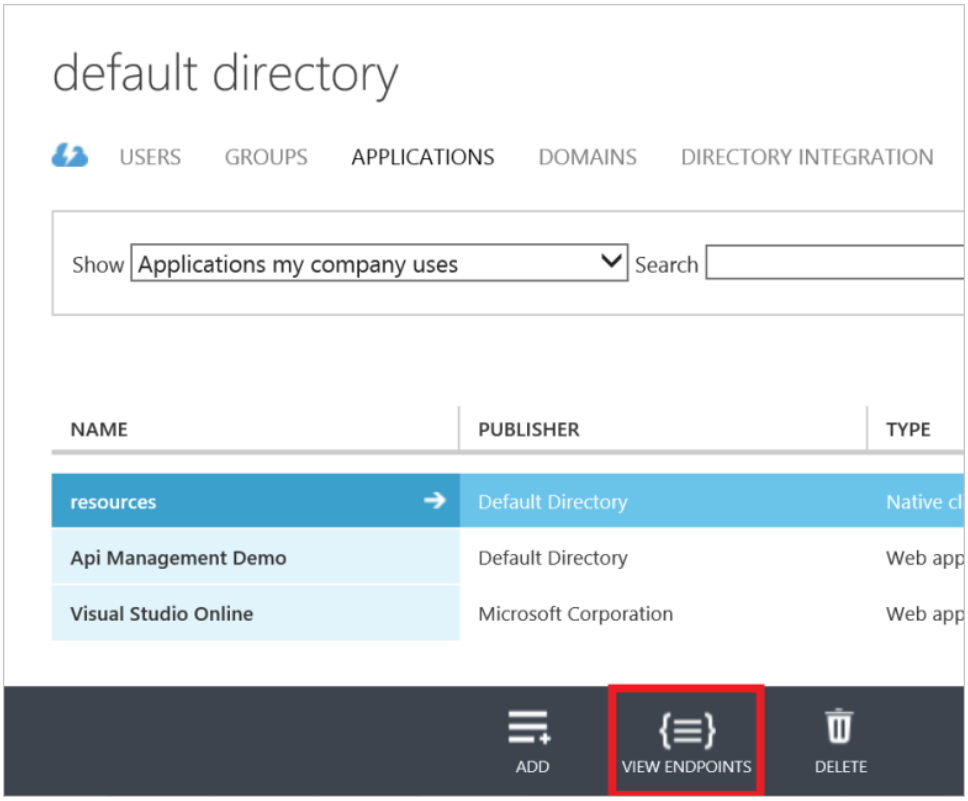


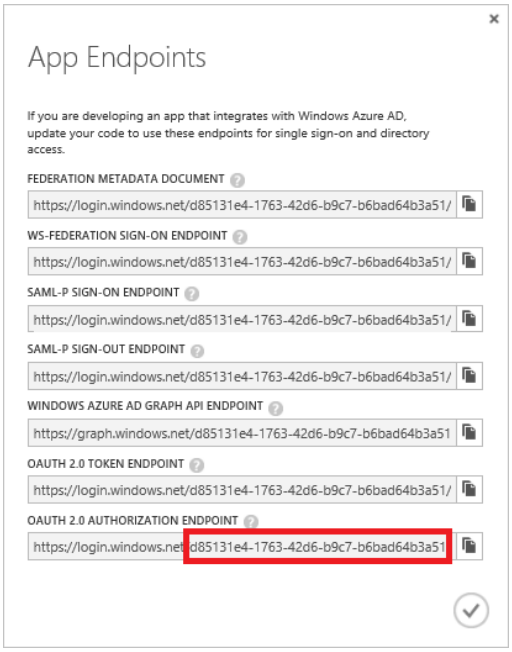
Prior to invoking the APIs that generate the backup and restore it, it is necessary to get a token. The following example uses the Microsoft.IdentityModel.Clients.ActiveDirectory nuget package to retrieve the token.

```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System;

namespace GetTokenResourceManagerRequests
{
    class Program
    {
        static void Main(string[] args)
        {
            var authenticationContext = new AuthenticationContext("https://login.windows.net/{tenant id}");
            var result = authenticationContext.AcquireToken("https://management.azure.com/", {application id}, new Uri({redirect uri});

            if (result == null) {
                throw new InvalidOperationException("Failed to obtain the JWT token");
            }

            Console.WriteLine(result.AccessToken);

            Console.ReadLine();
        }
    }
}
```

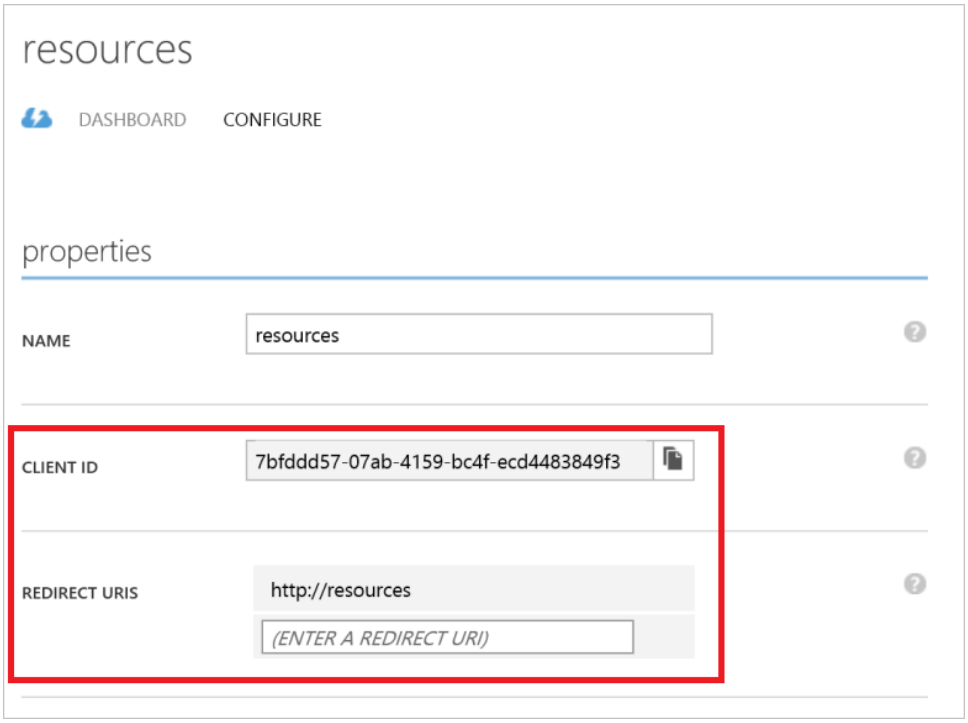
Replace `{tentand id}`, `{application id}`, and `{redirect uri}` using the following instructions.

Replace `{tenant id}` with the tenant id of the Azure Active Directory application you just created. You can access the id by clicking **View endpoints**.

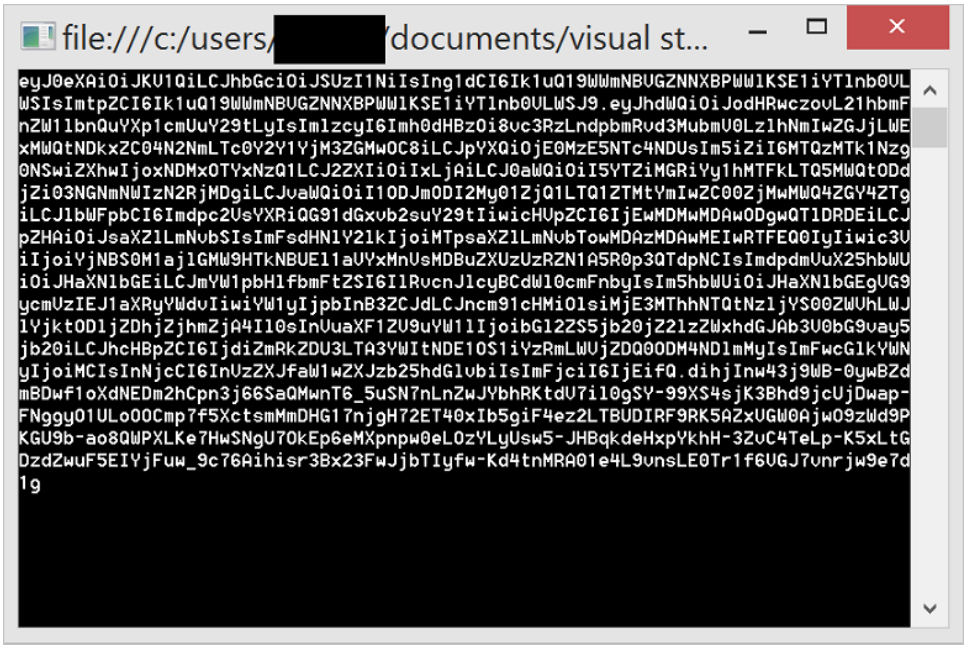




Replace `{application id}` and `{redirect uri}` using the **Client Id** and the URL from the **Redirect Uris** section from your Azure Active Directory application's **Configure** tab.



Once the values are specified, the code example should return a token similar to the following example.



Before calling the backup and restore operations described in the following sections, set the authorization request header for your REST call.

```
request.Headers.Add(HttpRequestHeader.Authorization, "Bearer " + token);
```

Backup an API Management service

To backup an API Management service issue the following HTTP request:

```
POST
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ApiManagement/service/{serviceName}/backup?
api-version={api-version}
```

where:

- `subscriptionId` - id of the subscription containing the API Management service you are attempting to backup
- `resourceGroupName` - a string in the form of 'Api-Default-{service-region}' where `service-region` identifies the Azure region where the API Management service you are trying to backup is hosted, e.g. `North-Central-US`
- `serviceName` - the name of the API Management service you are making a backup of specified at the time of its creation
- `api-version` - replace with `2014-02-14`

In the body of the request, specify the target Azure storage account name, access key, blob container name, and backup name:

```
'{
   storageAccount : {storage account name for the backup},
   accessKey : {access key for the account},
   containerName : {backup container name},
   backupName : {backup blob name}
}'
```

Set the value of the `Content-Type` request header to `application/json`.

Backup is a long running operation that may take multiple minutes to complete. If the request was successful and the backup process was initiated you'll receive a `202 Accepted` response status code with a `Location` header. Make 'GET' requests to the URL in the `Location` header to find out the status of the operation. While the backup is in progress you will continue to receive a '202 Accepted' status code. A Response code of `200 OK` will indicate successful completion of the backup operation.

Note:

- **Container** specified in the request body **must exist**.
- While backup is in progress you **should not attempt any service management operations** such as SKU upgrade or downgrade, domain name change, etc.
- Restore of a **backup is guaranteed only for 7 days** since the moment of its creation.
- **Usage data** used for creating analytics reports **is not included** in the backup. Use Azure API Management REST API to periodically retrieve analytics reports for safekeeping.
- The frequency with which you perform service backups will affect your recovery point objective. To minimize it we advise implementing regular backups as well as performing on-demand backups after making important changes to your API Management service.
- **Changes** made to the service configuration (e.g. APIs, policies, developer portal appearance) while backup operation is in process **might not be included in the backup and therefore will be lost**.

Restore an API Management service

To restore an API Management service from a previously created backup make the following HTTP request:

```
POST
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ApiManagement/service/{serviceName}/restore
api-version={api-version}
```

where:

- `subscriptionId` - id of the subscription containing the API Management service you are restoring a backup into
- `resourceGroupName` - a string in the form of 'Api-Default-{service-region}' where `service-region` identifies the Azure region where the API Management service you are restoring a backup into is hosted, e.g. `North-Central-US`
- `serviceName` - the name of the API Management service being restored into specified at the time of its creation
- `api-version` - replace with `2014-02-14`

In the body of the request, specify the backup file location, i.e. Azure storage account name, access key, blob container name, and backup name:

```
'{
    storageAccount : {storage account name for the backup},
    accessKey : {access key for the account},
    containerName : {backup container name},
    backupName : {backup blob name}
}'
```

Set the value of the `Content-Type` request header to `application/json`.

Restore is a long running operation that may take up to 30 or more minutes to complete. If the request was successful and the restore process was initiated you'll receive a `202 Accepted` response status code with a `Location` header. Make 'GET' requests to the URL in the `Location` header to find out the status of the operation. While the restore is in progress you will continue to receive '202 Accepted' status code. A response code of `200 OK` will indicate successful completion of the restore operation.

> **IMPORTANT**
>
> **The SKU** of the service being restored into **must match** the SKU of the backed up service being restored.
>
> **Changes** made to the service configuration (e.g. APIs, policies, developer portal appearance) while restore operation is in progress **could be overwritten**.

Next steps

Check out the following Microsoft blogs for two different walkthroughs of the backup/restore process.

- Replicate Azure API Management Accounts
  - Thank you to Gisela for her contribution to this article.
- Azure API Management: Backing Up and Restoring Configuration
  - The approach detailed by Stuart does not match the official guidance but it is very interesting.

How to use the API Inspector to trace calls in Azure API Management

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

API Management provides an API Inspector tool to help you with debugging and troubleshooting your APIs. The API Inspector can be used programmatically and can also be used directly from the developer portal.

In addition to tracing operations, API Inspector also traces policy expression evaluations. For a demonstration, see Cloud Cover Episode 177: More API Management Features and fast-forward to 21:00.

This guide provides a walk-through of using API Inspector.

> **NOTE**
> API Inspector traces are only generated and made available for requests containing subscription keys that belong to the administrator account.

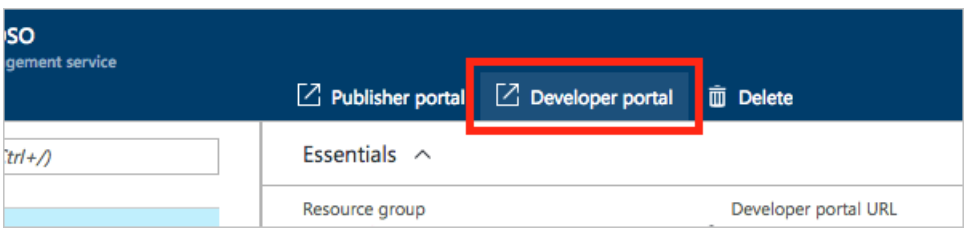
Use API Inspector to trace a call

To use API Inspector, add an **ocp-apim-trace: true** request header to your operation call, and then download and inspect the trace using the URL indicated by the **ocp-apim-trace-location** response header. This can be done programatically, and can also be done directly from the developer portal.

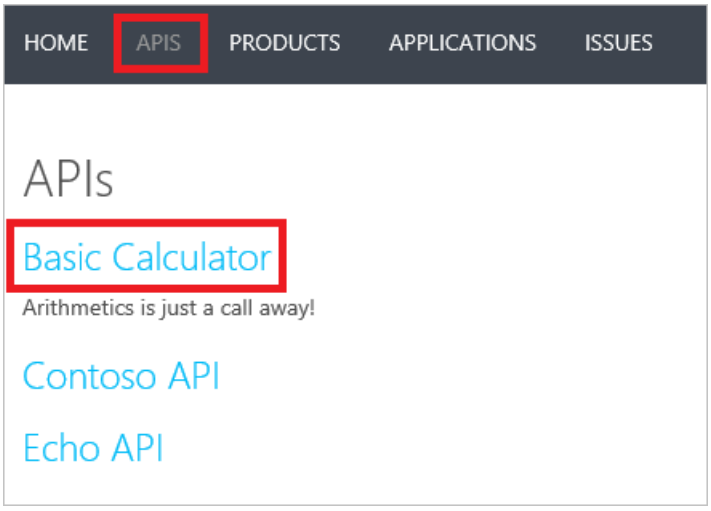
This tutorial shows how to use the API Inspector to trace operations using the Basic Calculator API that is configured in the Manage your first API getting started tutorial. If you haven't completed that tutorial it only takes a few moments to import the Basic Calculator API, or you can use another API of your choosing such as the Echo API. Each API Management service instance comes pre-configured with an Echo API that can be used to experiment with and learn about API Management. The Echo API returns back whatever input is sent to it. To use it, you can invoke any HTTP verb, and the return value will simply be what you sent.

To get started, click **Developer portal** in the Azure Portal for your API Management service. Operations can be called directly from the developer portal which provides a convenient way to view and test the operations of an API.

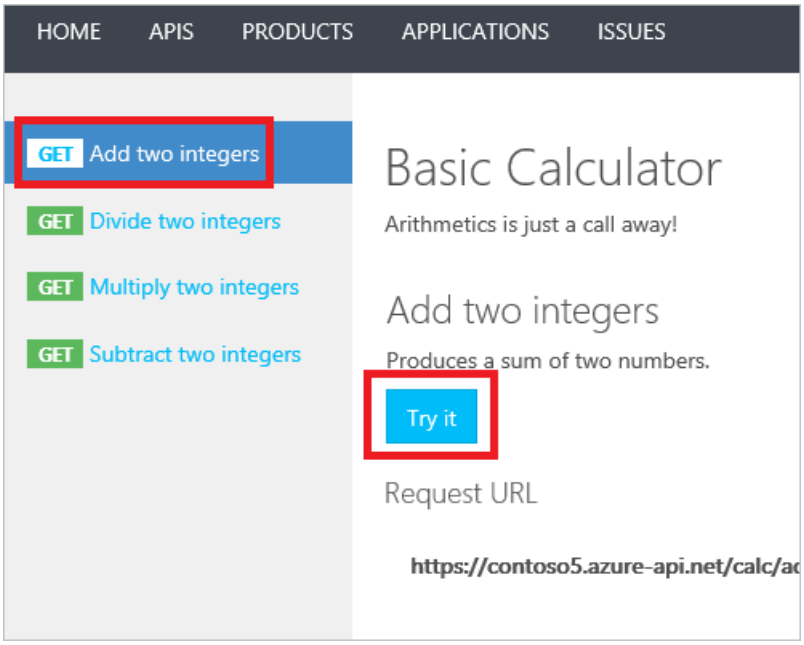
> If you haven't yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.



Click **APIs** from the top menu, and then click **Basic Calculator**.

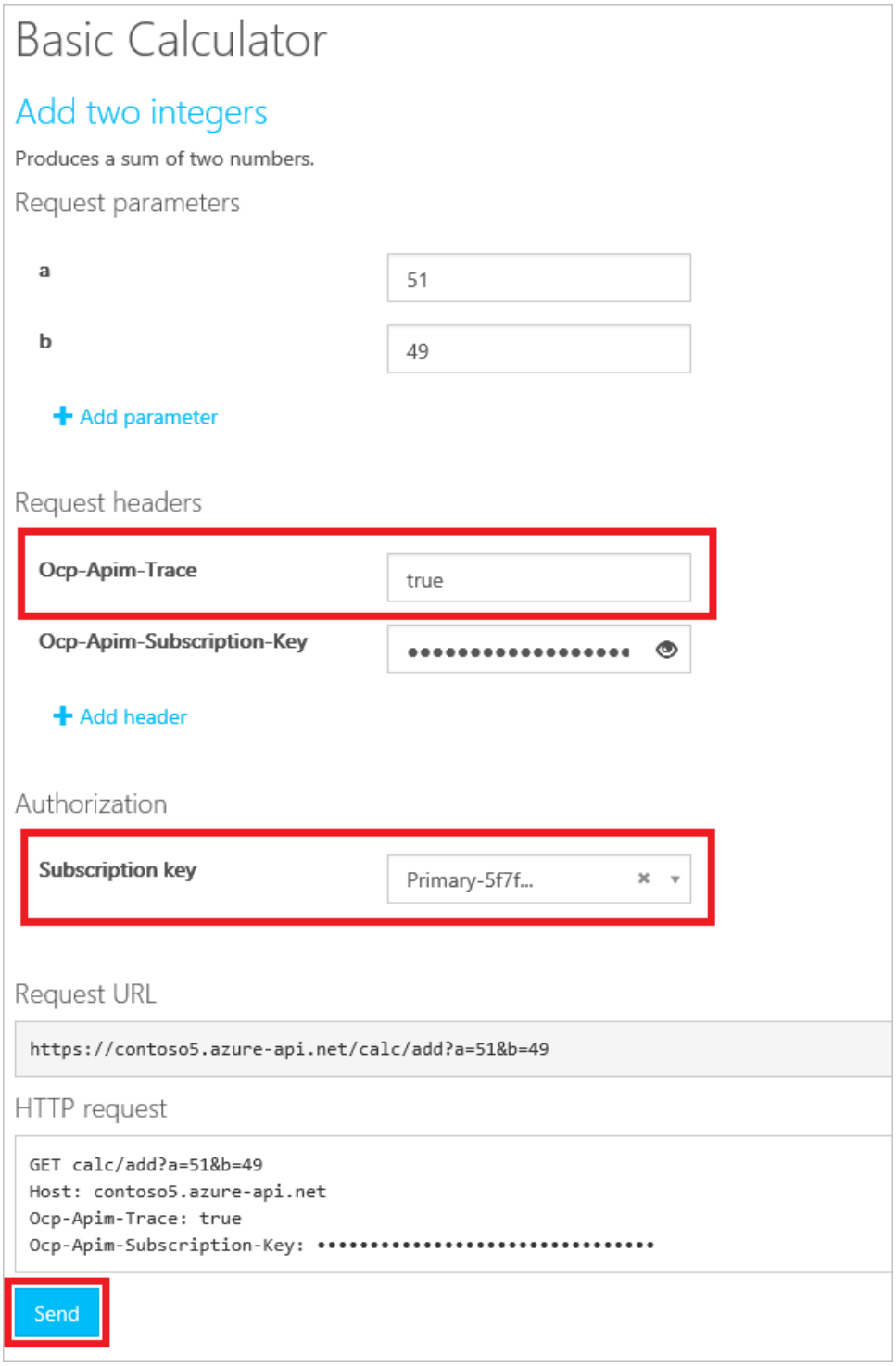


Click **Try it** to try the **Add two integers** operation.

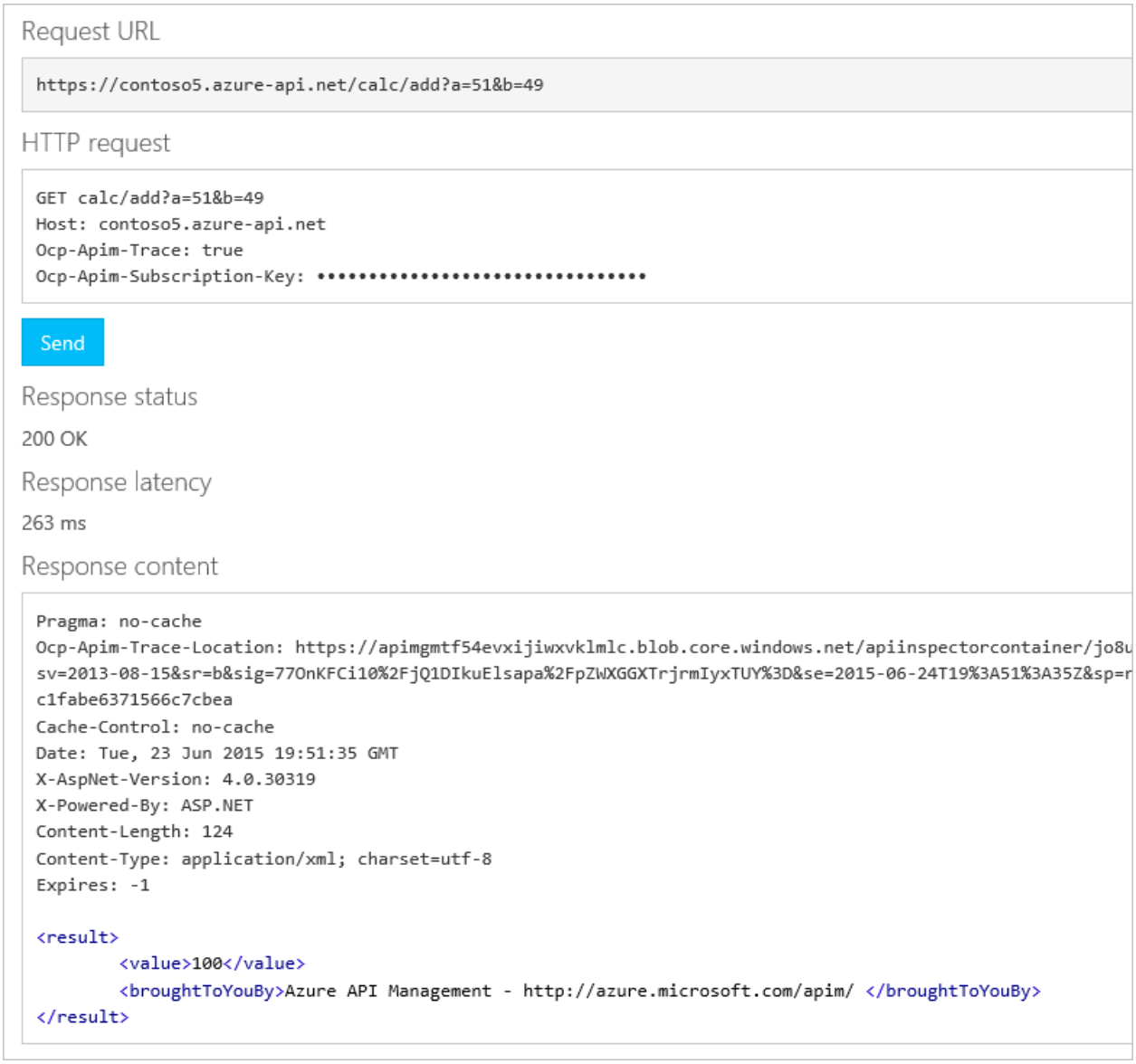


Keep the default parameter values, and select the subscription key for the product you want to use from the **subscription-key** drop-down.

By default in the developer portal the **Ocp-Apim-Trace** header is already set to **true**. This header configures whether or not a trace is generated.



Click **Send** to invoke the operation.



In the response headers will be an **ocp-apim-trace-location** with a value similar to the following example.

```
ocp-apim-trace-location :
https://contosoltdxw7zagdfsprykd.blob.core.windows.net/apiinspectorcontainer/ZW3e23NsW4wQyS-SHjS0Og2-2?sv=2013-
08-15&sr=b&sig=Mgx7cMHsLmVDv%2B%2BSzvg3JR8qGTHoOyIAV7xDsZbF7%2Bk%3D&se=2014-05-
04T21%3A00%3A13Z&sp=r&verify_guid=a56a17d83de04fcb8b9766df38514742
```

The trace can be downloaded from the specified location and reviewed as demonstrated in the next step.

Inspect the trace

To review the values in the trace, download the trace file from the **ocp-apim-trace-location** URL. It is a text file in JSON format, and contains entries similar to the following example.

```
{
    "traceId": "abcd8ea63d134c1fabe6371566c7cbea",
    "traceEntries": {
        "inbound": [
            {
                "source": "handler",
                "timestamp": "2015-06-23T19:51:35.2998610Z",
                "elapsed": "00:00:00.0725926",
                "data": {
                    "request": {
                        "method": "GET",
                        "url": "https://contoso5.azure-api.net/calc/add?a=51&b=49",
```

```
                "headers": [
                    {
                        "name": "Ocp-Apim-Subscription-Key",
                        "value": "5d7c41af64a44a68a2ea46580d271a59"
                    },
                    {
                        "name": "Connection",
                        "value": "Keep-Alive"
                    },
                    {
                        "name": "Host",
                        "value": "contoso5.azure-api.net"
                    }
                ]
            }
        }
    },
    {
        "source": "mapper",
        "timestamp": "2015-06-23T19:51:35.2998610Z",
        "elapsed": "00:00:00.0726213",
        "data": {
            "configuration": {
                "api": {
                    "from": "/calc",
                    "to": {
                        "scheme": "http",
                        "host": "calcapi.cloudapp.net",
                        "port": 80,
                        "path": "/api",
                        "queryString": "",
                        "query": {},
                        "isDefaultPort": true
                    }
                },
                "operation": {
                    "method": "GET",
                    "uriTemplate": "/add?a={a}&b={b}"
                },
                "user": {
                    "id": 1,
                    "groups": [
                        "Administrators",
                        "Developers"
                    ]
                },
                "product": {
                    "id": 1
                }
            }
        }
    },
    {
        "source": "handler",
        "timestamp": "2015-06-23T19:51:35.2998610Z",
        "elapsed": "00:00:00.0727522",
        "data": {
            "message": "Request is being forwarded to the backend service.",
            "request": {
                "method": "GET",
                "url": "http://calcapi.cloudapp.net/api/add?a=51&b=49",
                "headers": [
                    {
                        "name": "Ocp-Apim-Subscription-Key",
                        "value": "5d7c41af64a44a68a2ea46580d271a59"
                    },
                    {
                        "name": "X-Forwarded-For",
                        "value": "33.52.215.35"
```

```
                        }
                    ]
                }
            }
        }
    ],
    "outbound": [
        {
            "source": "handler",
            "timestamp": "2015-06-23T19:51:35.4256650Z",
            "elapsed": "00:00:00.1960601",
            "data": {
                "response": {
                    "status": {
                        "code": 200,
                        "reason": "OK"
                    },
                    "headers": [
                        {
                            "name": "Pragma",
                            "value": "no-cache"
                        },
                        {
                            "name": "Content-Length",
                            "value": "124"
                        },
                        {
                            "name": "Cache-Control",
                            "value": "no-cache"
                        },
                        {
                            "name": "Content-Type",
                            "value": "application/xml; charset=utf-8"
                        },
                        {
                            "name": "Date",
                            "value": "Tue, 23 Jun 2015 19:51:35 GMT"
                        },
                        {
                            "name": "Expires",
                            "value": "-1"
                        },
                        {
                            "name": "Server",
                            "value": "Microsoft-IIS/8.5"
                        },
                        {
                            "name": "X-AspNet-Version",
                            "value": "4.0.30319"
                        },
                        {
                            "name": "X-Powered-By",
                            "value": "ASP.NET"
                        }
                    ]
                }
            }
        },
        {
            "source": "handler",
            "timestamp": "2015-06-23T19:51:35.4256650Z",
            "elapsed": "00:00:00.1961112",
            "data": {
                "message": "Response headers have been sent to the caller. Starting to stream the response
body."
            }
        },
        {
            "source": "handler",
```

```
                "timestamp": "2015-06-23T19:51:35.4256650Z",
                "elapsed": "00:00:00.1963155",
                "data": {
                    "message": "Response body streaming to the caller is complete."
                }
            }
        ]
    }
}
```

Next steps

- Watch a demo of tracing policy expressions in Cloud Cover Episode 177: More API Management Features. Fast-forward to 21:00 to see the demo.

How to manage user accounts in Azure API Management

11/15/2016 • 2 min to read • Edit on GitHub

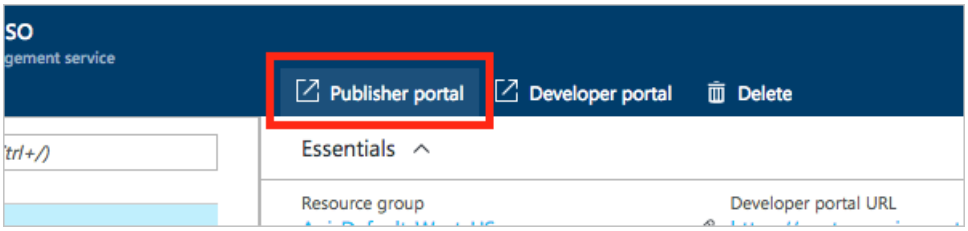
Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Anton Babadjanov (Microsoft)

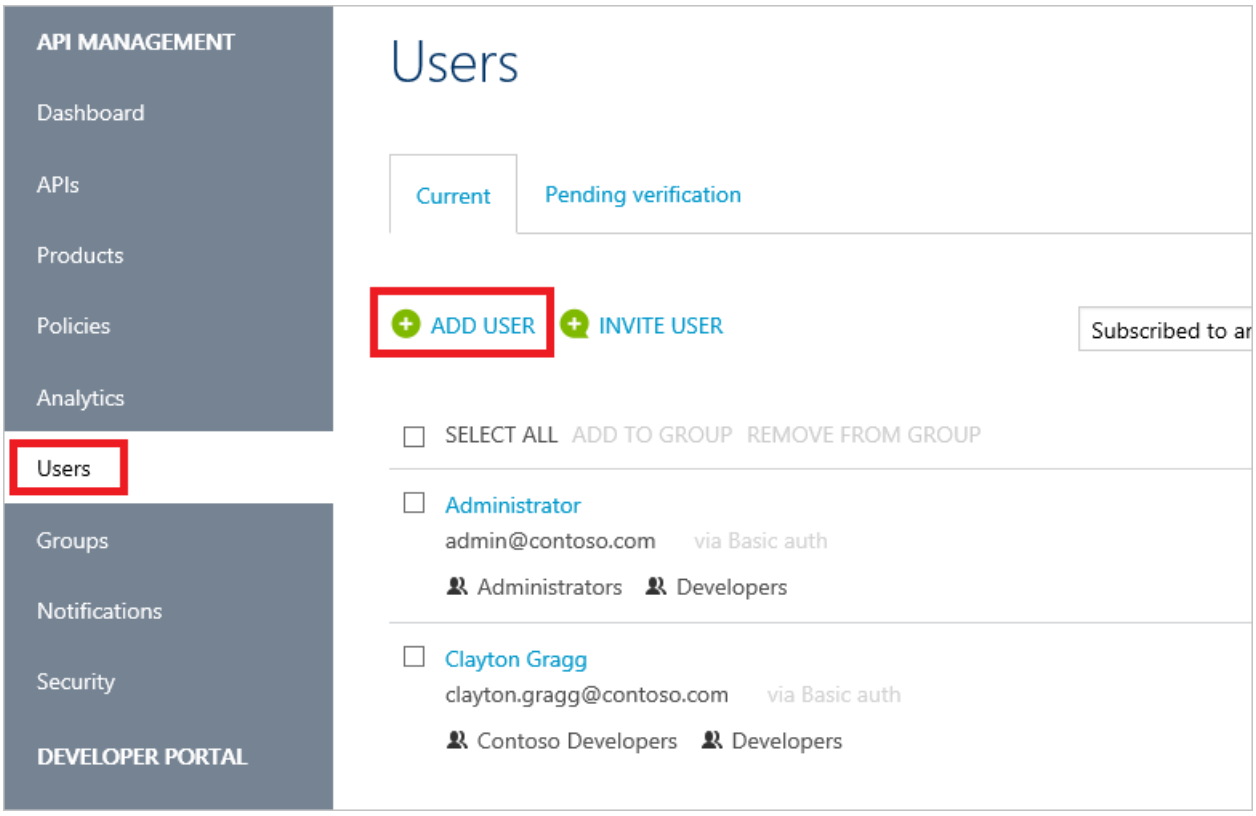
In API Management, developers are the users of the APIs that you expose using API Management. This guide shows to how to create and invite developers to use the APIs and products that you make available to them with your API Management instance. For information on managing user accounts programmatically, see the User entity documentation in the API Management REST reference.

Create a new developer

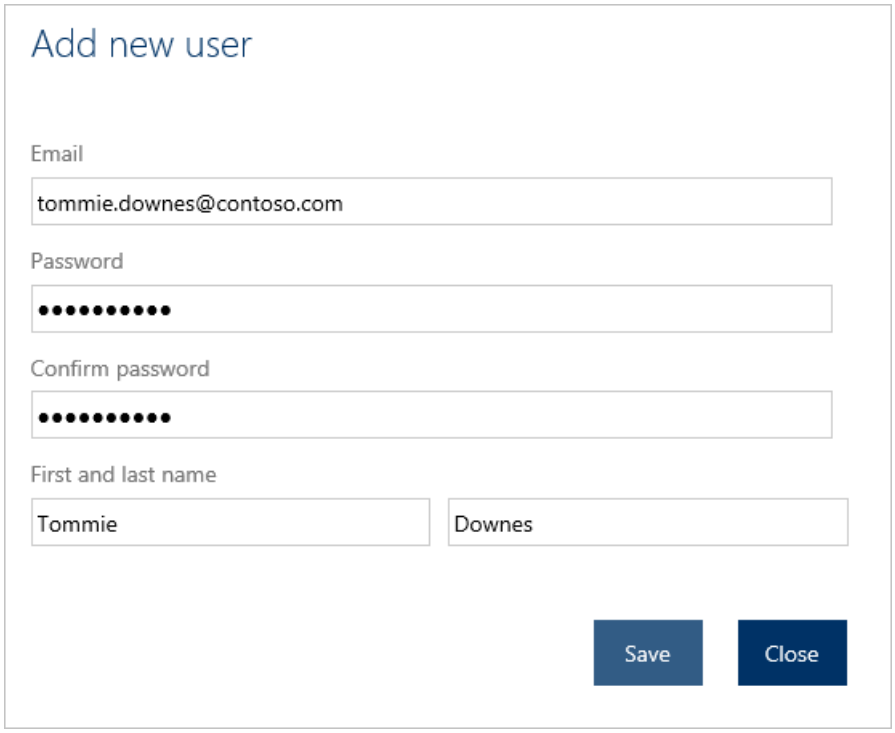
To create a new developer, click **Publisher portal** in the Azure Portal for your API Management service. This takes you to the API Management publisher portal. If you have not yet created an API Management service instance, see Create an API Management service instance in the Get started with Azure API Management tutorial.



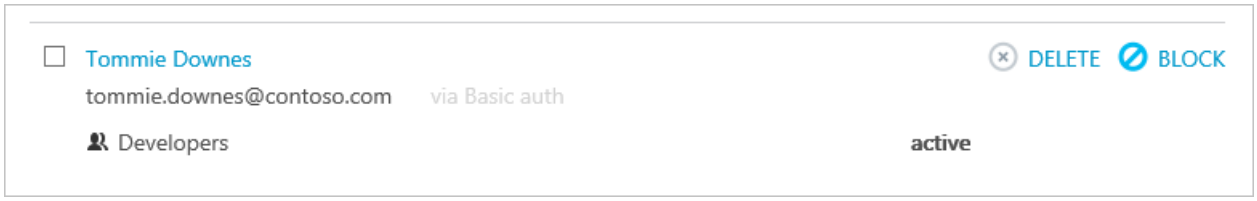
Click **Users** from the **API Management** menu on the left, and then click **add user**.



Enter the **Email**, **Password**, and **Name** for the new developer and click **Save**.



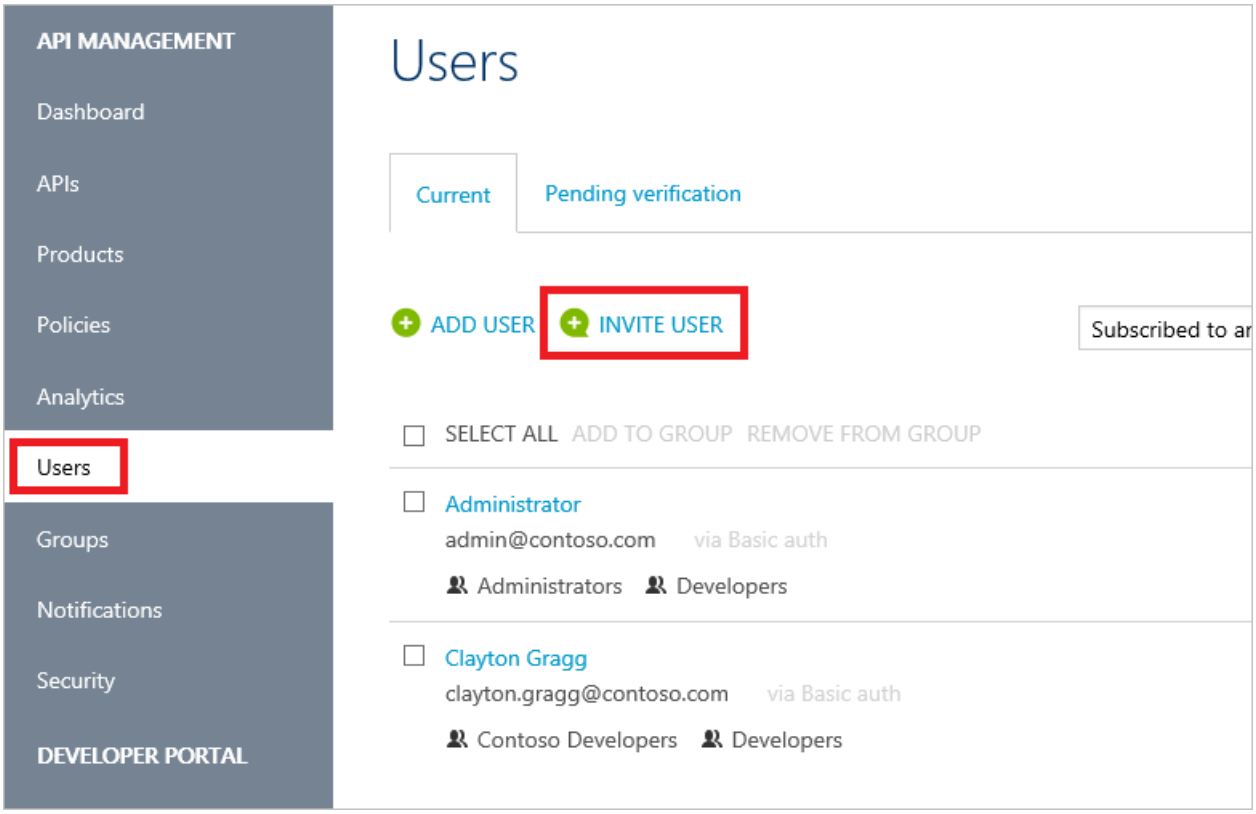
By default, newly created developer accounts are **Active**, and associated with the **Developers** group.



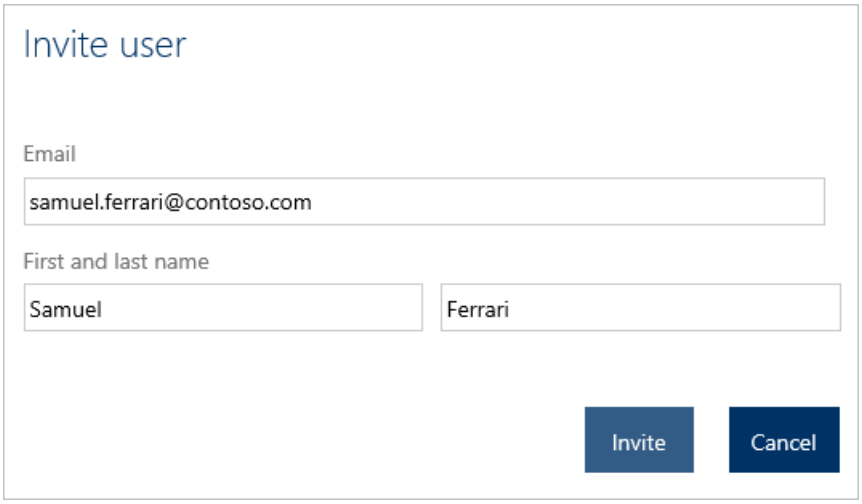
Developer accounts that are in an **active** state can be used to access all of the APIs for which they have subscriptions. To associate the newly created developer with additional groups, see How to associate groups with developers.

Invite a developer

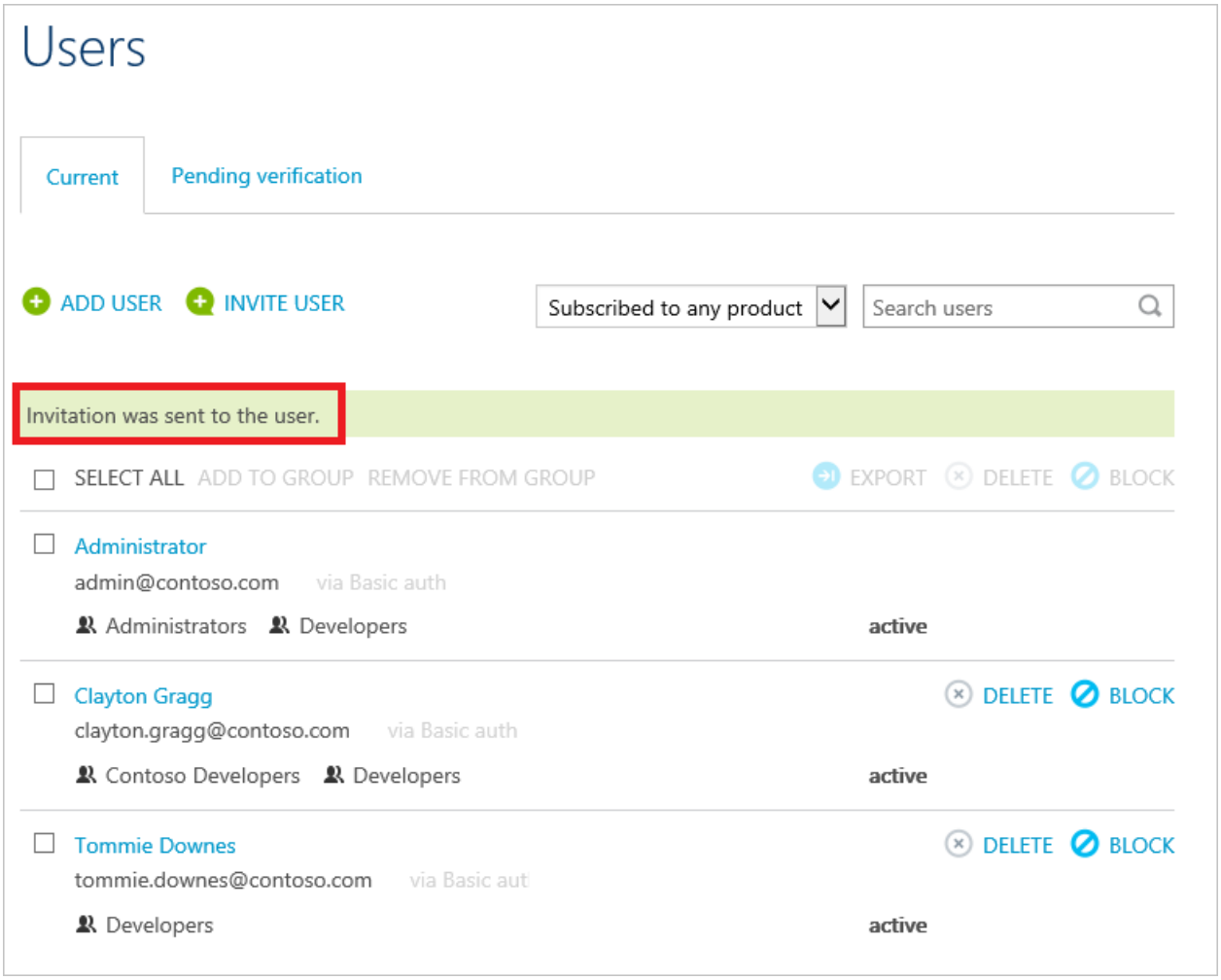
To invite a developer, click **Users** from the **API Management** menu on the left, and then click **Invite User**.



Enter the name and email address of the developer, and click **Invite**.



A confirmation message is displayed, but the newly invited developer does not appear in the list until after they accept the invitation.

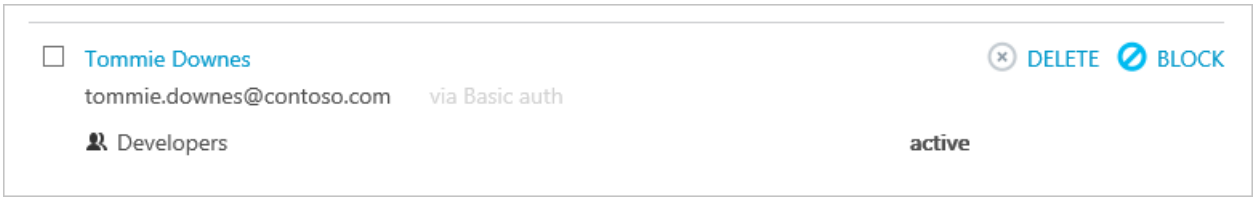


When a developer is invited, an email is sent to the developer. This email is generated using a template and is customizable. For more information, see Configure email templates.

Once the invitation is accepted, the account becomes active.

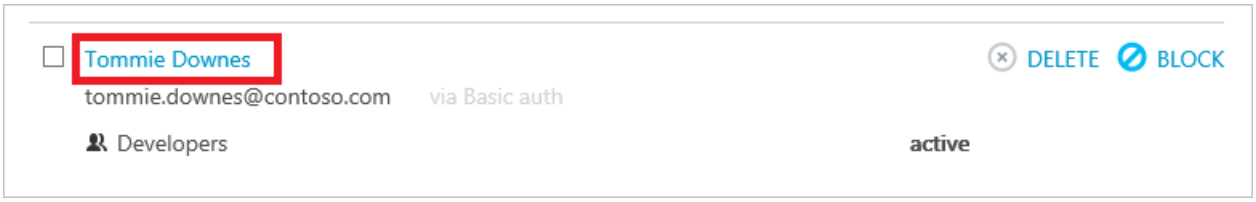
Deactivate or reactivate a developer account

By default, newly created or invited developer accounts are **Active**. To deactivate a developer account, click **Block**. To reactivate a blocked developer account, click **Activate**. A blocked developer account can not access the developer portal or call any APIs. To delete a user account, click **Delete**.

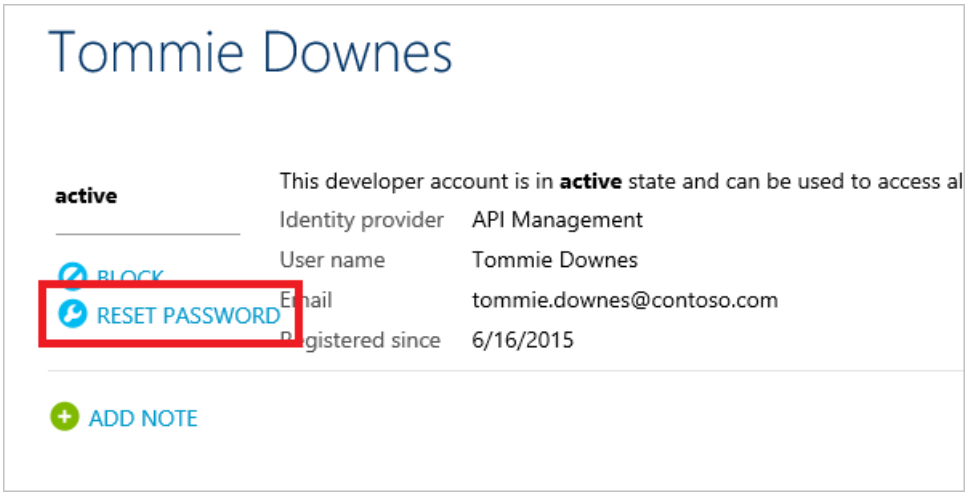


Reset a user password

To reset the password for a user account, click the name of the account.



Click **Reset password** to send a link to the user to reset their password.



To programmatically work with user accounts, see the User entity documentation in the API Management REST reference. To reset a user account password to a specific value, you can use the Update a user operation and specify the desired password.

Pending verification

Next steps

Once a developer account is created, you can associate it with roles and subscribe it to products and APIs. For more information, see How to create and use groups.

[]: ./media/api-management-howto-create-or-invite-developers/.png

Managing Azure API Management using Azure Automation

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Chris Sanders • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Steve Danielson

---

This guide will introduce you to the Azure Automation service, and how it can be used to simplify management of Azure API Management.

What is Azure Automation?

Azure Automation is an Azure service for simplifying cloud management through process automation. Using Azure Automation, manual, repeated, long-running, and error-prone tasks can be automated to increase reliability, efficiency, and time to value for your organization.

Azure Automation provides a highly-reliable, highly-available workflow execution engine that scales to meet your needs. In Azure Automation, processes can be kicked off manually, by 3rd-party systems, or at scheduled intervals so that tasks happen exactly when needed.

Reduce operational overhead and free up IT and DevOps staff to focus on work that adds business value by moving your cloud management tasks to be run automatically by Azure Automation.

How can Azure Automation help manage Azure API Management?

API Management can be managed in Azure Automation by using the Windows PowerShell cmdlets for Azure API Management API. Within Azure Automation you can write PowerShell workflow scripts to perform many of your API Management tasks using the cmdlets. You can also pair these cmdlets in Azure Automation with the cmdlets for other Azure services, to automate complex tasks across Azure services and 3rd party systems.

Here are some examples of using API Management with Automation:

- Azure API Management – Using PowerShell for backup and restore

Next Steps

Now that you've learned the basics of Azure Automation and how it can be used to manage Azure API Management, follow these links to learn more.

- See the Azure Automation getting started tutorial.

How to save and configure your API Management service configuration using Git

11/15/2016 • 10 min to read • Edit on GitHub

Contributors

steved0x • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

---

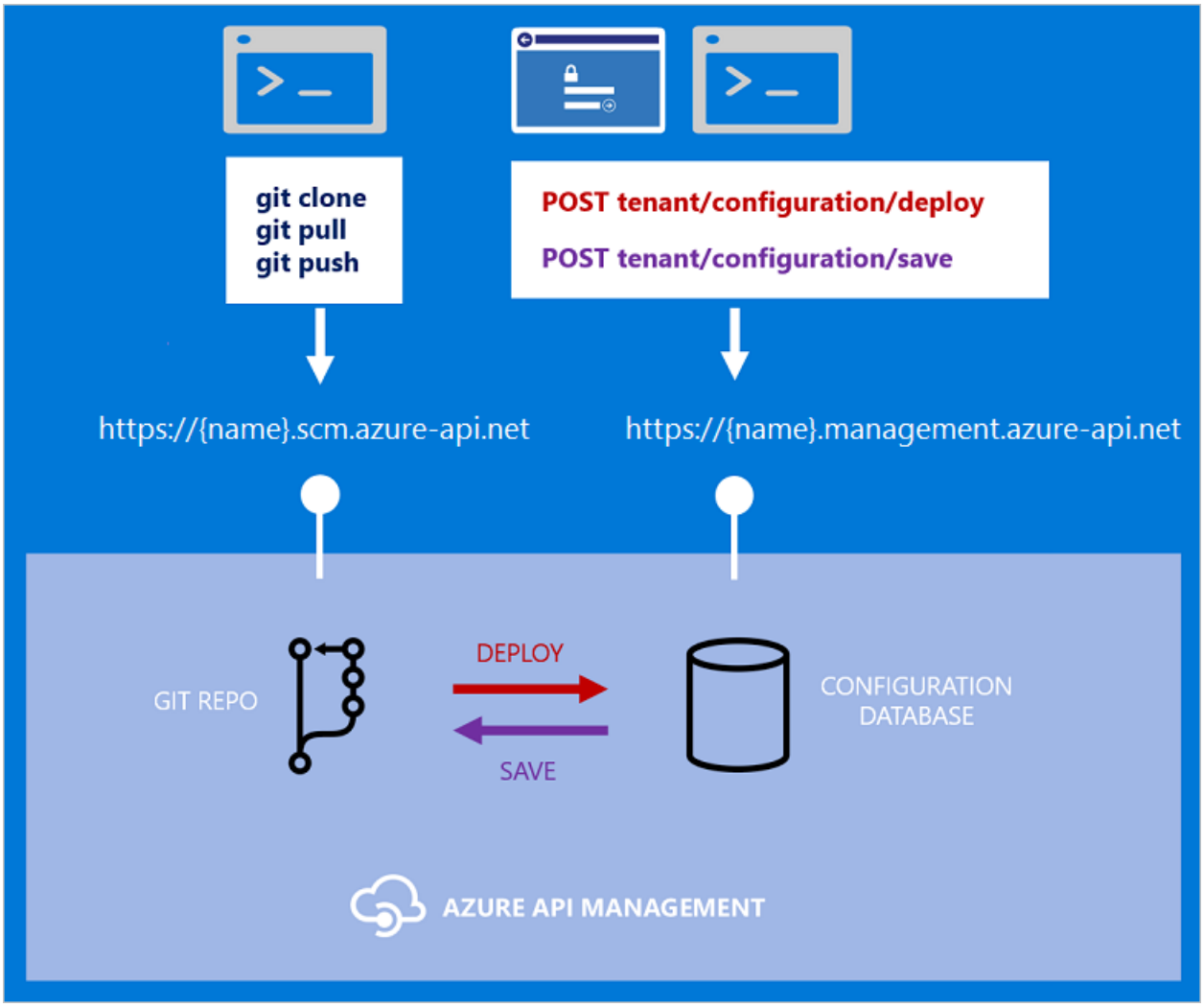
IMPORTANT

Git configuration for API Management is currently in preview. It is functionally complete, but is in preview because we are actively seeking feedback on this feature. It is possible that we may make a breaking change in response to customer feedback, so we recommend not depending on the feature for use in production environments. If you have any feedback or questions, please let us know at `apimgmt@microsoft.com`.

Each API Management service instance maintains a configuration database that contains information about the configuration and metadata for the service instance. Changes can be made to the service instance by changing a setting in the publisher portal, using a PowerShell cmdlet, or making a REST API call. In addition to these methods, you can also manage your service instance configuration using Git, enabling service management scenarios such as:

- Configuration versioning - download and store different versions of your service configuration
- Bulk configuration changes - make changes to multiple parts of your service configuration in your local repository and integrate the changes back to the server with a single operation
- Familiar Git toolchain and workflow - use the Git tooling and workflows that you are already familiar with

The following diagram shows an overview of the different ways to configure your API Management service instance.



When you make changes to your service using the publisher portal, PowerShell cmdlets, or the REST API, you are managing your service configuration database using the `https://{name}.management.azure-api.net` endpoint, as shown on the right side of the diagram. The left side of the diagram illustrates how you can manage your service configuration using Git and Git repository for your service located at `https://{name}.scm.azure-api.net`.

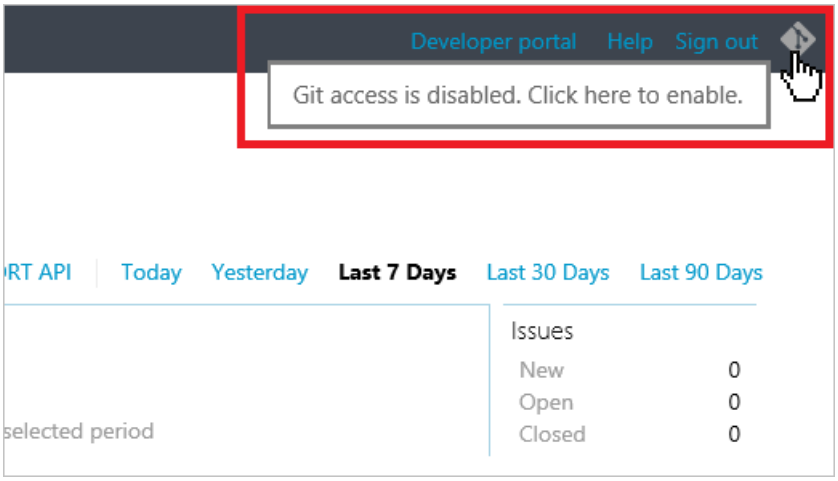
The following steps provide an overview of managing your API Management service instance using Git.

1. Enable Git access in your service
2. Save your service configuration database to your Git repository
3. Clone the Git repo to your local machine
4. Pull the latest repo down to your local machine, and commit and push changes back to your repo
5. Deploy the changes from your repo into your service configuration database

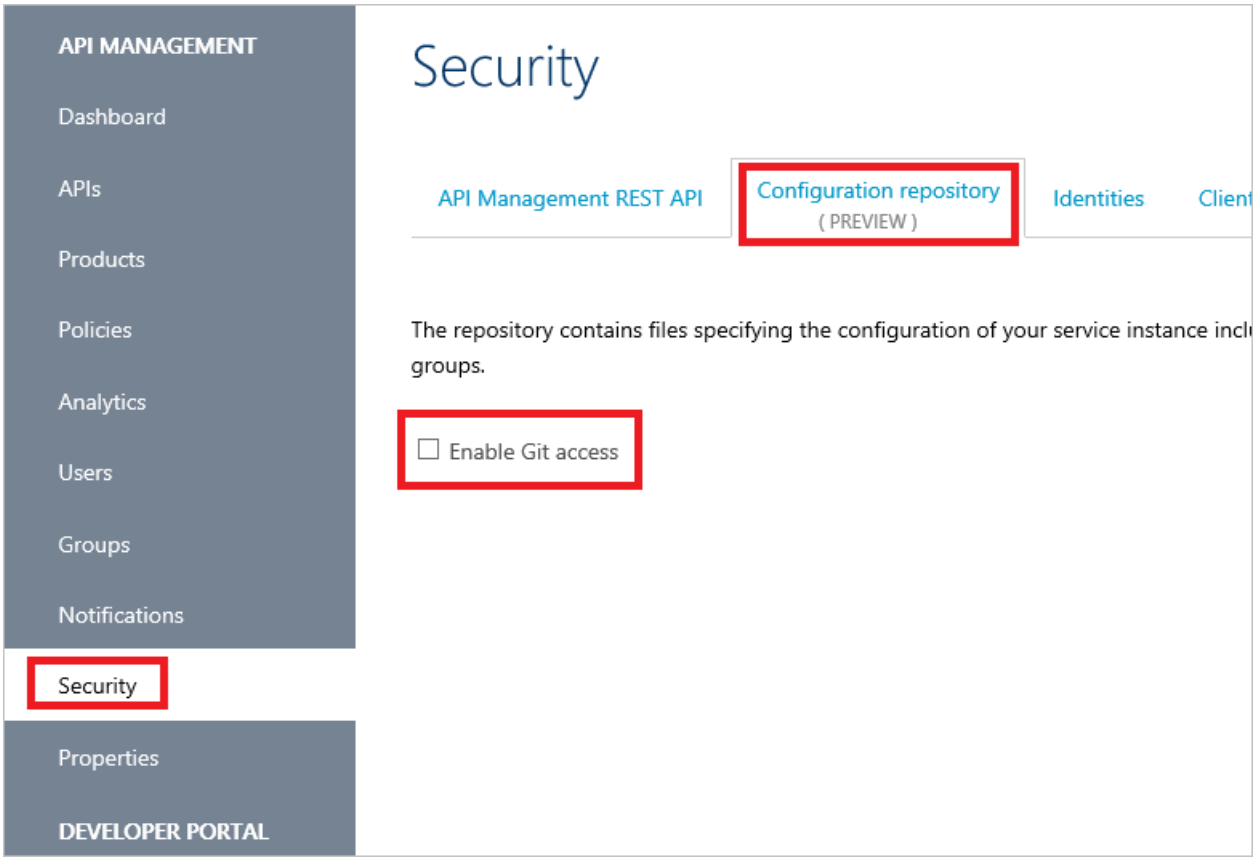
This article describes how to enable and use Git to manage your service configuration and provides a reference for the files and folders in the Git repository.

To enable Git access

You can quickly view the status of your Git configuration by viewing the Git icon in the upper-right corner of the publisher portal. In this example, Git access has not yet been enabled.

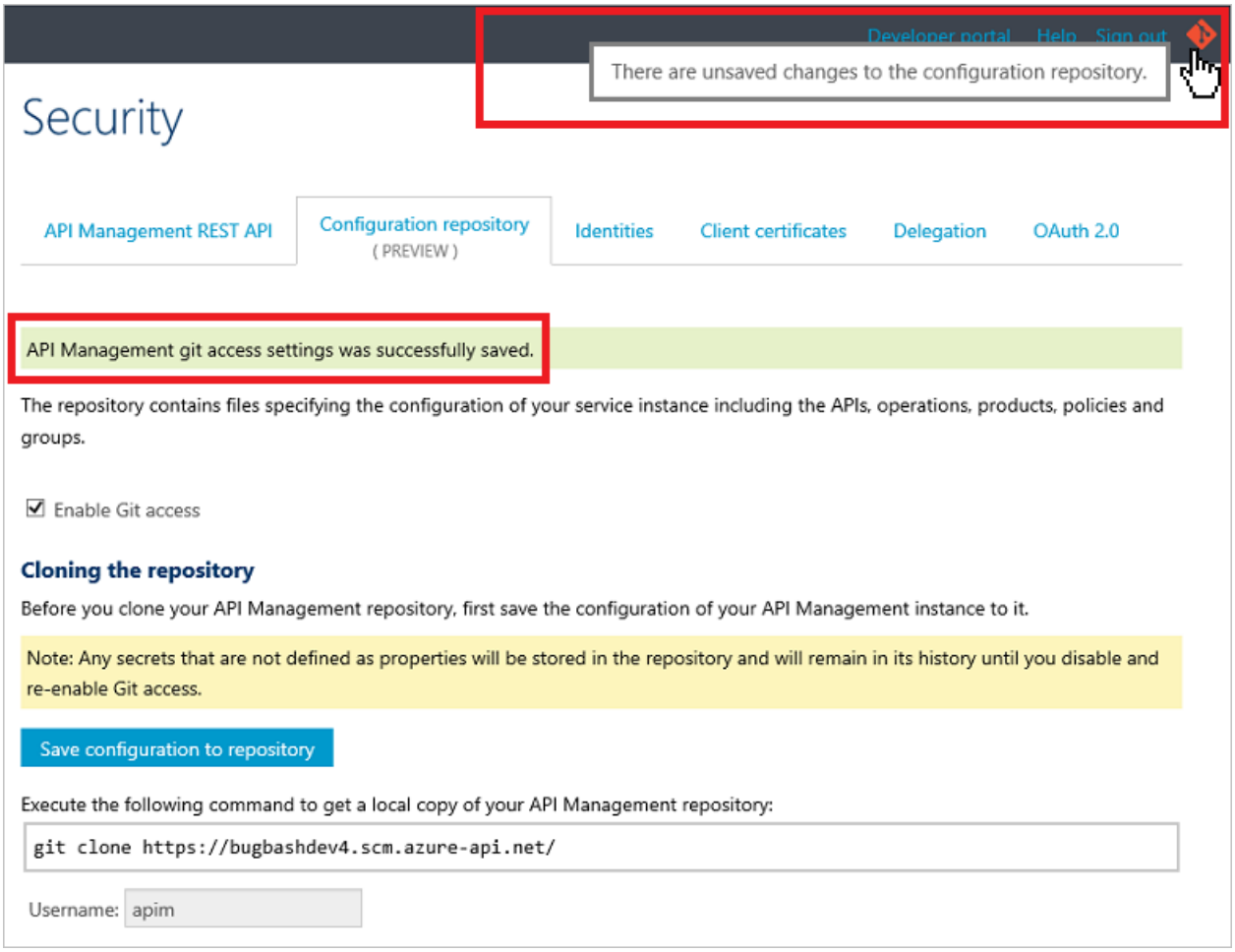


To view and configure your Git configuration settings, you can either click the Git icon, or click the **Security** menu and navigate to the **Configuration repository** tab.



To enable Git access, check the **Enable Git access** checkbox.

After a moment the change is saved and a confirmation message is displayed. Note that Git icon has changed to color to indicate that Git access is enabled and the status message now indicates that there are unsaved changes to the repository. This is because the API Management service configuration database has not yet been saved to the repository.

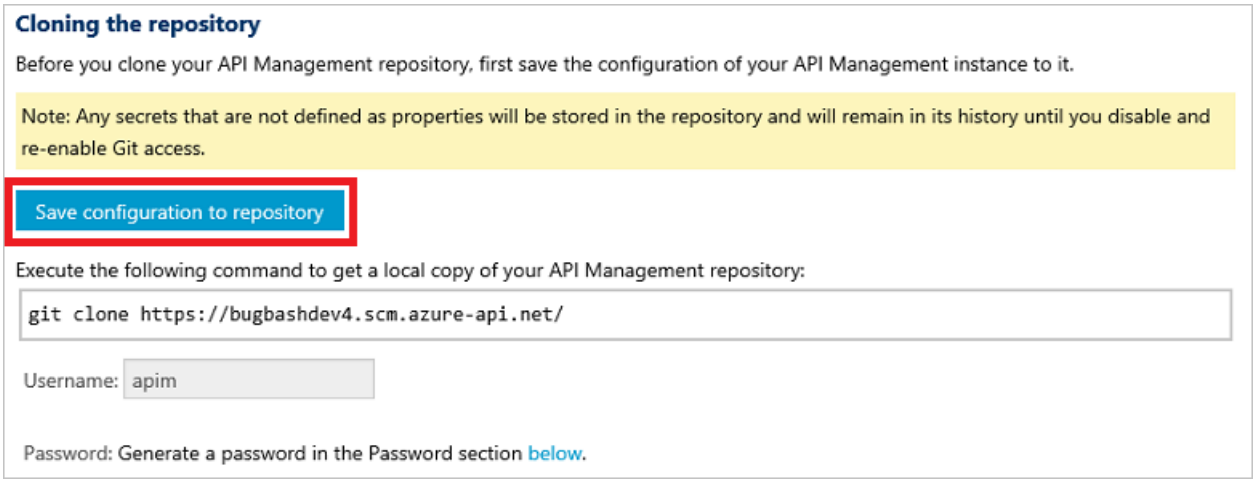


> **IMPORTANT**
>
> Any secrets that are not defined as properties will be stored in the repository and will remain in its history until you disable and re-enable Git access. Properties provide a secure place to manage constant string values, including secrets, across all API configuration and policies, so you don't have to store them directly in your policy statements. For more information, see How to use properties in Azure API Management policies.

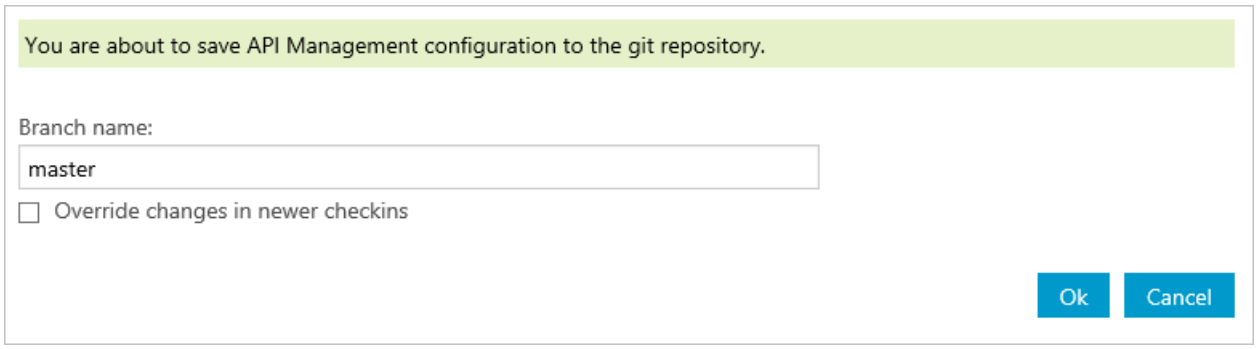
For information on enabling or disabling Git access using the REST API, see Enable or disable Git access using the REST API.

To save the service configuration to the Git repository

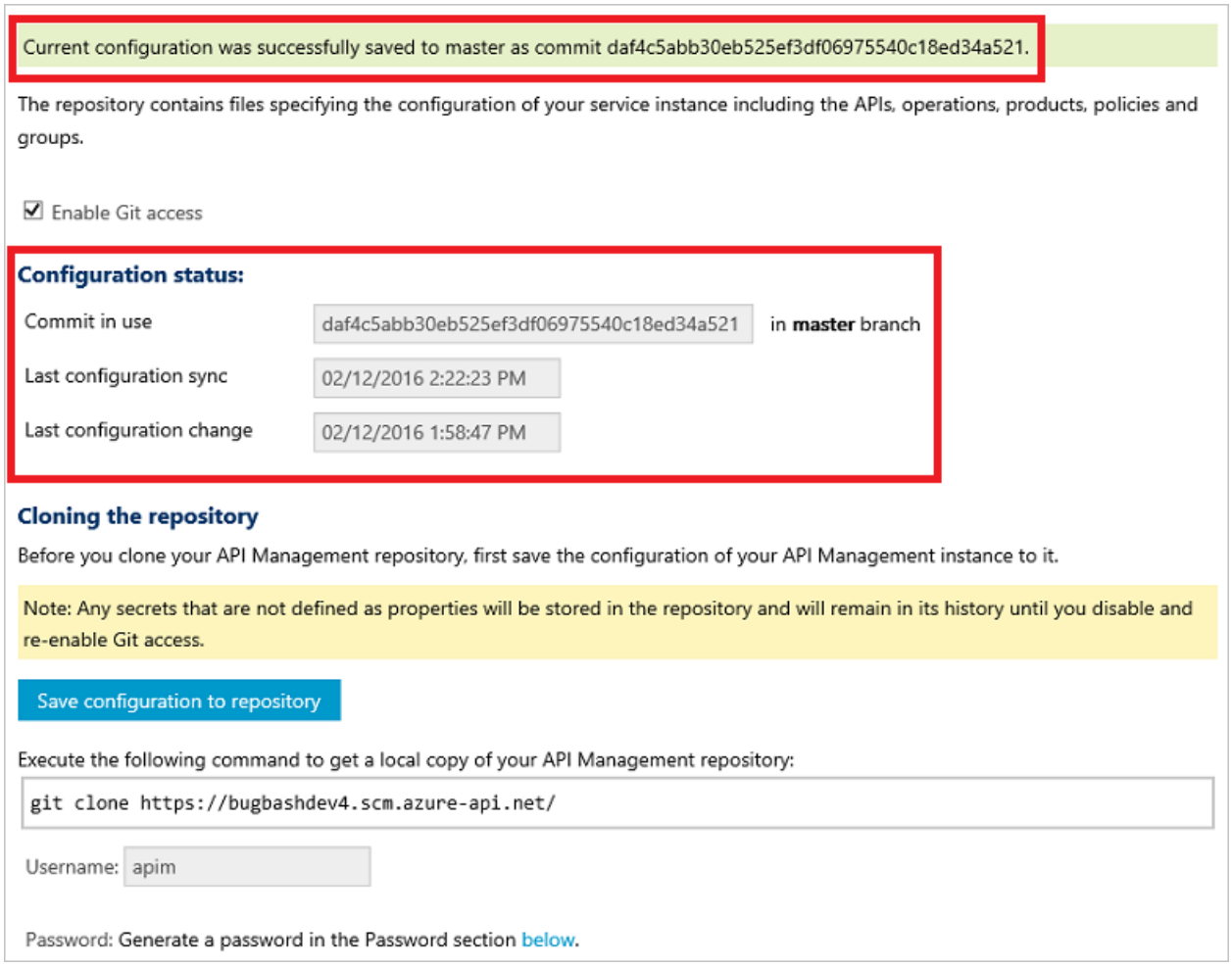
The first step before cloning the repository is to save the current state of the service configuration to the repository. Click **Save configuration to repository**.



Make any desired changes on the confirmation screen and click **Ok** to save.



After a few moments the configuration is saved, and the configuration status of the repository is displayed, including the date and time of the last configuration change and the last synchronization between the service configuration and the repository.

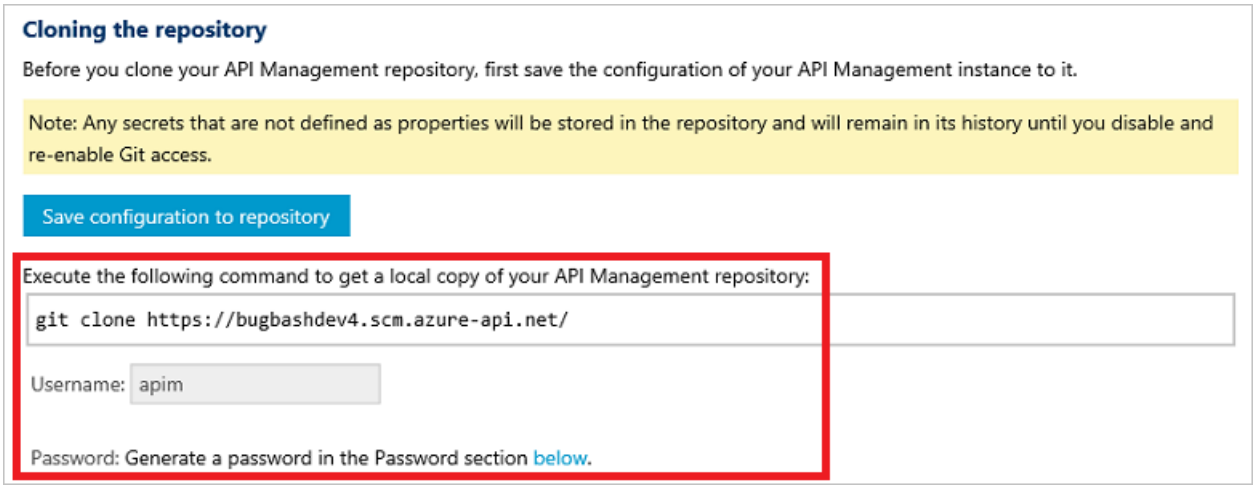


Once the configuration is saved to the repository, it can be cloned.

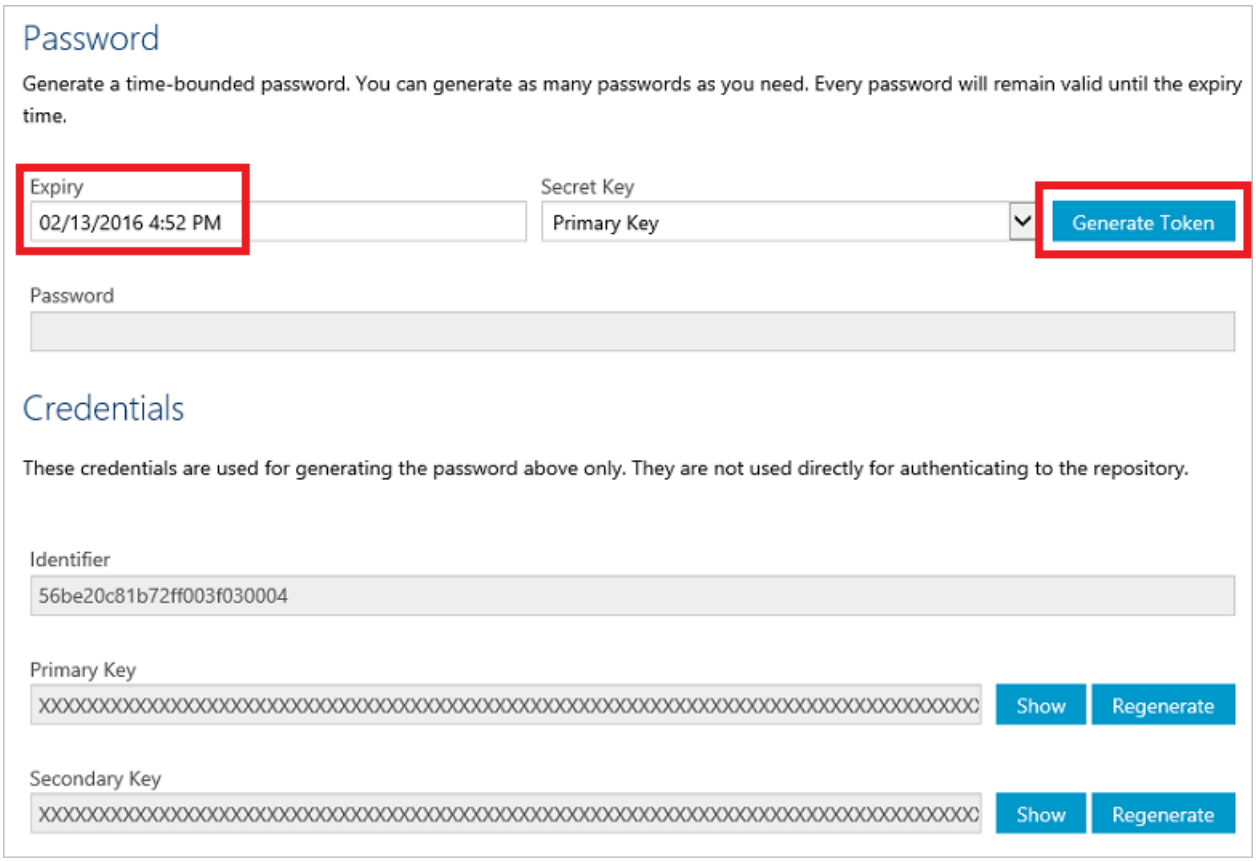
For information on performing this operation using the REST API, see Commit configuration snapshot using the REST API.

To clone the repository to your local machine

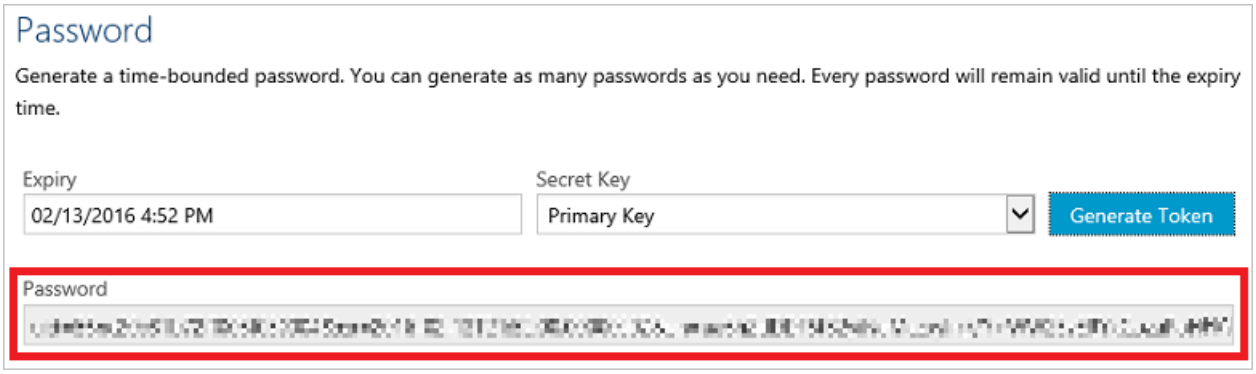
To clone a repository, you need the URL to your repository, a user name, and a password. The user name and URL are displayed near the top of the **Configuration repository** tab.



The password is generated at the bottom of the **Configuration repository** tab.



To generate a password, first ensure that the **Expiry** is set to the desired expiration date and time, and then click **Generate Token**.



> **IMPORTANT**
> Make a note of this password. Once you leave this page the password will not be displayed again.

The following examples use the Git Bash tool from Git for Windows but you can use any Git tool that you are familiar with.

Open your Git tool in the desired folder and run the following command to clone the git repository to your local machine, using the command provided by the publisher portal.

```
git clone https://bugbashdev4.scm.azure-api.net/
```

Provide the user name and password when prompted.

If you receive any errors, try modifying your `git clone` command to include the user name and password, as shown in the following example.

```
git clone https://username:password@bugbashdev4.scm.azure-api.net/
```

If this provides an error, try URL encoding the password portion of the command. One quick way to do this is to open Visual Studio, and issue the following command in the **Immediate Window**. To open the **Immediate Window**, open any solution or project in Visual Studio (or create a new empty console application), and choose **Windows**, **Immediate** from the **Debug** menu.

```
?System.NetWebUtility.UrlEncode("password from publisher portal")
```

Use the encoded password along with your user name and repository location to construct the git command.

```
git clone https://username:url encoded password@bugbashdev4.scm.azure-api.net/
```

Once the repository is cloned you can view and work with it in your local file system. For more information, see File and folder structure reference of local Git repository.

To update your local repository with the most current service instance configuration

If you make changes to your API Management service instance in the publisher portal or using the REST API, you must save these changes to the repository before you can update your local repository with the latest changes. To do this, click **Save configuration to repository** on the **Configuration repository** tab in the publisher portal, and then issue the following command in your local repository.

```
git pull
```

Before running `git pull` ensure that you are in the folder for your local repository. If you have just completed the `git clone` command, then you must change the directory to your repo by running a command like the following.

```
cd bugbashdev4.scm.azure-api.net/
```

To push changes from your local repo to the server repo

To push changes from your local repository to the server repository, you must commit your changes and then push them to the server repository. To commit your changes, open your Git command tool, switch to the directory

of your local repository, and issue the following commands.

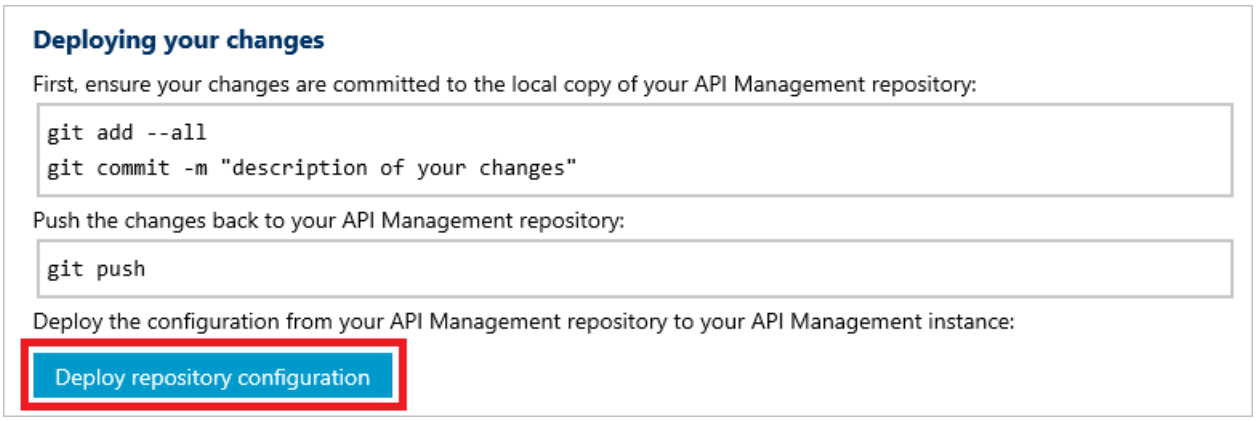
```
git add --all
git commit -m "Description of your changes"
```

To push all of the commits to the server, run the following command.

```
git push
```

To deploy any service configuration changes to the API Management service instance

Once your local changes are committed and pushed to the server repository, you can deploy them to your API Management service instance.



For information on performing this operation using the REST API, see Deploy Git changes to configuration database using the REST API.

File and folder structure reference of local Git repository

The files and folders in the local git repository contain the configuration information about the service instance.

ITEM	DESCRIPTION
root api-management folder	Contains top-level configuration for the service instance
apis folder	Contains the configuration for the apis in the service instance
groups folder	Contains the configuration for the groups in the service instance
policies folder	Contains the policies in the service instance
portalStyles folder	Contains the configuration for the developer portal customizations in the service instance
products folder	Contains the configuration for the products in the service instance
templates folder	Contains the configuration for the email templates in the service instance

Each folder can contain one or more files, and in some cases one or more folders, for example a folder for each API, product, or group. The files within each folder are specific for the entity type described by the folder name.

FILE TYPE	PURPOSE
json	Configuration information about the respective entity
html	Descriptions about the entity, often displayed in the developer portal
xml	Policy statements
css	Style sheets for developer portal customization

These files can be created, deleted, edited, and managed on your local file system, and the changes deployed back to the your API Management service instance.

> **NOTE**
>
> The following entities are not contained in the Git repository and cannot be configured using Git.
>
> - Users
> - Subscriptions
> - Properties
> - Developer portal entities other than styles

Root api-management folder

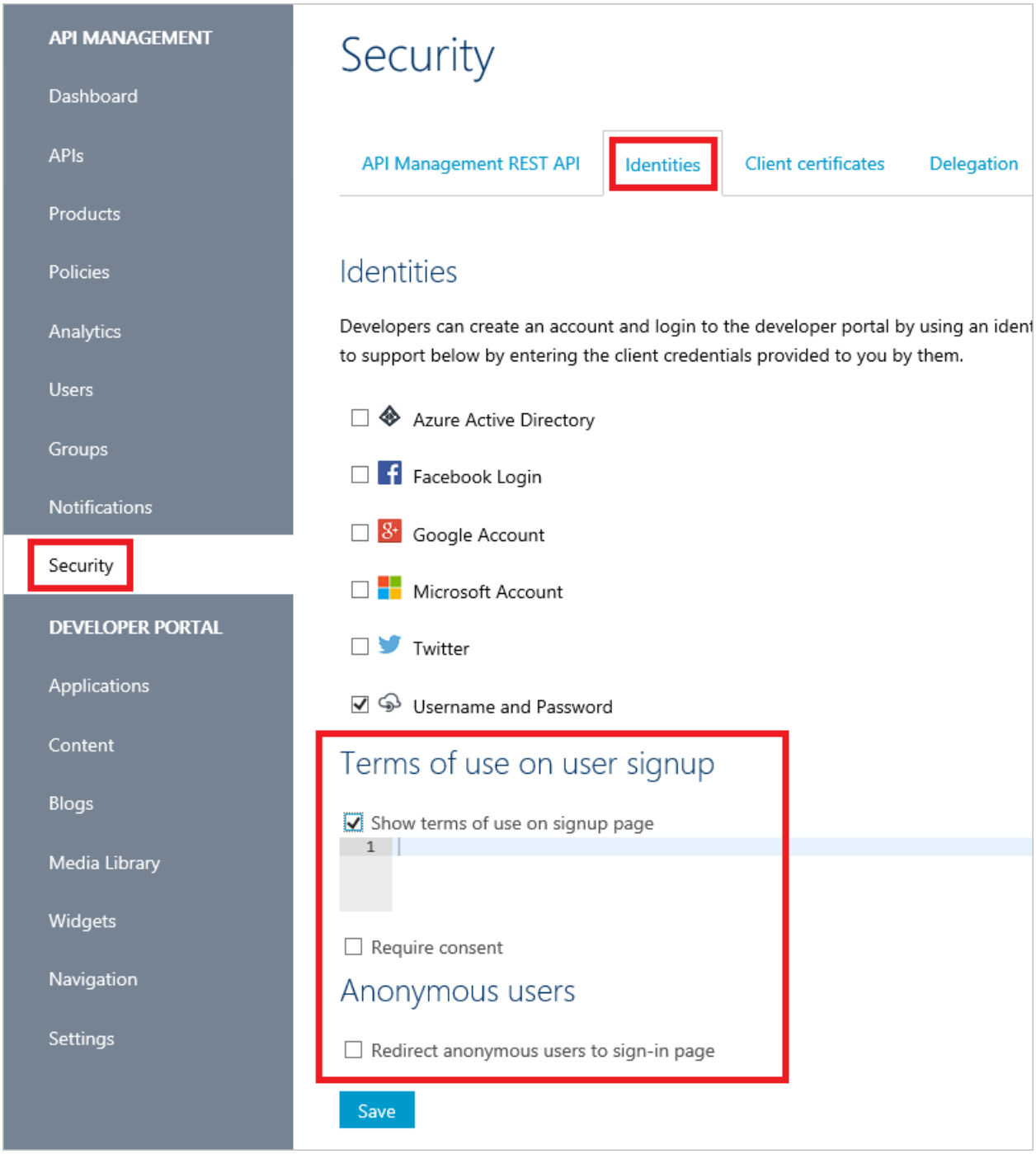
The root `api-management` folder contains a `configuration.json` file that contains top-level information about the service instance in the following format.

```
{
  "settings": {
    "RegistrationEnabled": "True",
    "UserRegistrationTerms": null,
    "UserRegistrationTermsEnabled": "False",
    "UserRegistrationTermsConsentRequired": "False",
    "DelegationEnabled": "False",
    "DelegationUrl": "",
    "DelegatedSubscriptionEnabled": "False",
    "DelegationValidationKey": ""
  },
  "$ref-policy": "api-management/policies/global.xml"
}
```

The first four settings (`RegistrationEnabled` , `UserRegistrationTerms` , `UserRegistrationTermsEnabled` , and `UserRegistrationTermsConsentRequired`) map to the following settings on the **Identities** tab in the **Security** section.

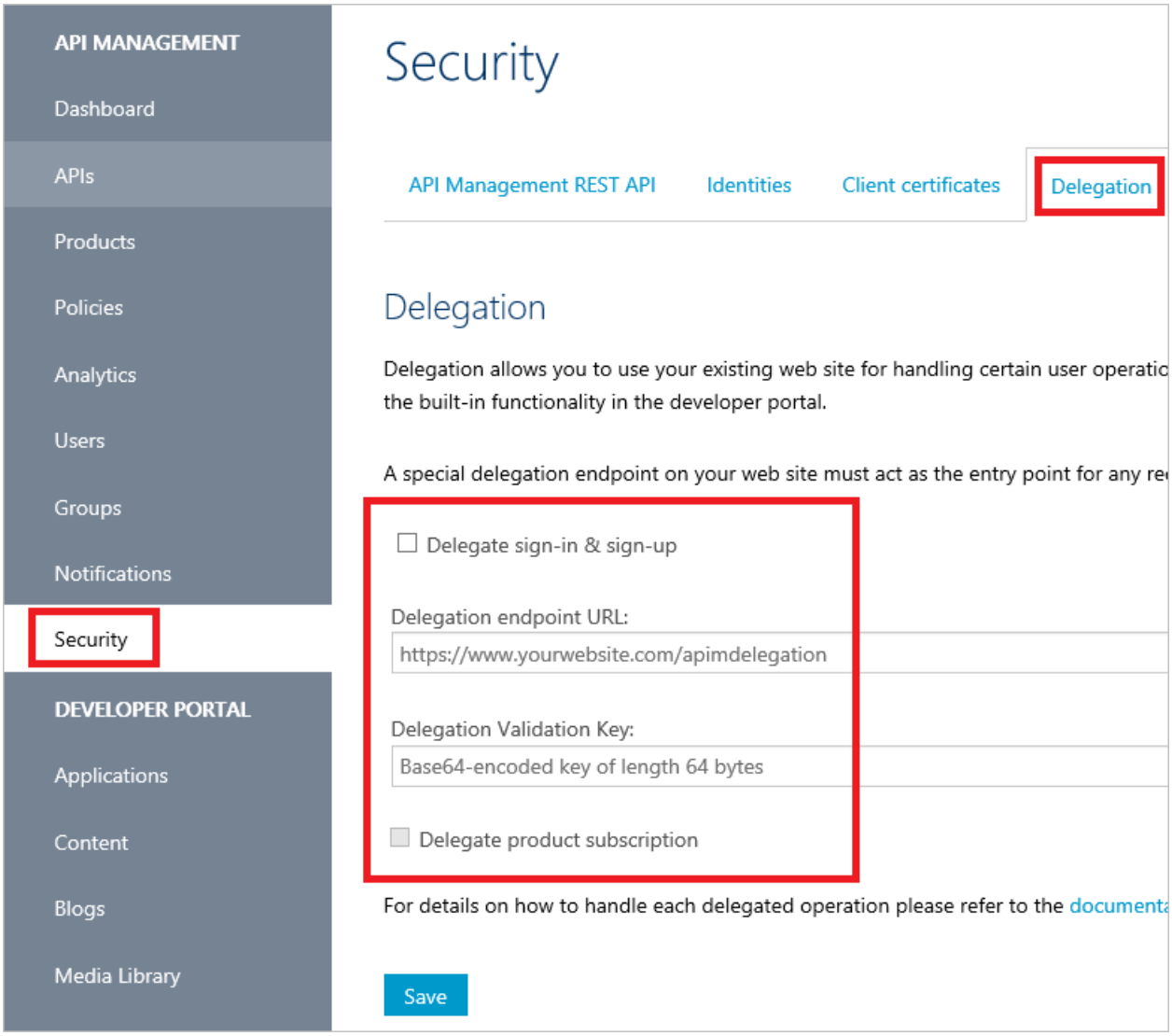
IDENTITY SETTING	MAPS TO
RegistrationEnabled	**Redirect anonymous users to sign-in page** checkbox
UserRegistrationTerms	**Terms of use on user signup** textbox
UserRegistrationTermsEnabled	**Show terms of use on signup page** checkbox

IDENTITY SETTING	MAPS TO
UserRegistrationTermsConsentRequired	**Require consent** checkbox



The next four settings (`DelegationEnabled` , `DelegationUrl` , `DelegatedSubscriptionEnabled` , and `DelegationValidationKey`) map to the following settings on the **Delegation** tab in the **Security** section.

DELEGATION SETTING	MAPS TO
DelegationEnabled	**Delegate sign-in & sign-up** checkbox
DelegationUrl	**Delegation endpoint URL** textbox
DelegatedSubscriptionEnabled	**Delegate product subscription** checkbox
DelegationValidationKey	**Delegate Validation Key** textbox



The final setting, `$ref-policy`, maps to the global policy statements file for the service instance.

apis folder

The `apis` folder contains a folder for each API in the service instance which contains the following items.

- `apis\<api name>\configuration.json` - this is the configuration for the API and contains information about the backend service URL and the operations. This is the same information that would be returned if you were to call Get a specific API with `export=true` in `application/json` format.
- `apis\<api name>\api.description.html` - this is the description of the API and corresponds to the `description` property of the API entity.
- `apis\<api name>\operations\` - this folder contains `<operation name>.description.html` files that map to the operations in the API. Each file contains the description of a single operation in the API which maps to the `description` property of the operation entity in the REST API.

groups folder

The `groups` folder contains a folder for each group defined in the service instance.

- `groups\<group name>\configuration.json` - this is the configuration for the group. This is the same information that would be returned if you were to call the Get a specific group operation.
- `groups\<group name>\description.html` - this is the description of the group and corresponds to the `description` property of the group entity.

policies folder

The `policies` folder contains the policy statements for your service instance.

- `policies\global.xml` - contains policies defined at global scope for your service instance.
- `policies\apis\<api name>\` - if you have any policies defined at API scope, they are contained in this folder.
- `policies\apis\<api name>\<operation name>\` folder - if you have any policies defined at operation scope, they are contained in this folder in `<operation name>.xml` files that map to the policy statements for each operation.
- `policies\products\` - if you have any policies defined at product scope, they are contained in this folder, which contains `<product name>.xml` files that map to the policy statements for each product.

portalStyles folder

The `portalStyles` folder contains configuration and style sheets for developer portal customizations for the service instance.

- `portalStyles\configuration.json` - contains the names of the style sheets used by the developer portal
- `portalStyles\<style name>.css` - each `<style name>.css` file contains styles for the developer portal (`Preview.css` and `Production.css` by default).

products folder

The `products` folder contains a folder for each product defined in the service instance.

- `products\<product name>\configuration.json` - this is the configuration for the product. This is the same information that would be returned if you were to call the Get a specific product operation.
- `products\<product name>\product.description.html` - this is the description of the product and corresponds to the `description` property of the product entity in the REST API.

templates

The `templates` folder contains configuration for the email templates of the service instance.

- `<template name>\configuration.json` - this is the configuration for the email template.
- `<template name>\body.html` - this is the body of the email template.

Next steps

For information on other ways to manage your service instance, see:

- Manage your service instance using the following PowerShell cmdlets
  - Service deployment PowerShell cmdlet reference
  - Service management PowerShell cmdlet reference
- Manage your service instance in the publisher portal
  - Manage your first API
- Manage your service instance using the REST API
  - API Management REST API reference

Watch a video overview

API design guidance

11/15/2016 • 33 min to read • Edit on GitHub

Contributors

Masashi Narumoto • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Stuart Leeks • Christopher Bennage • Trent Swanson • Dene Hager

patterns & practices
proven practices for predictable results

Overview

Many modern web-based solutions make the use of web services, hosted by web servers, to provide functionality for remote client applications. The operations that a web service exposes constitute a web API. A well-designed web API should aim to support:

- **Platform independence**. Client applications should be able to utilize the API that the web service provides without requiring how the data or operations that API exposes are physically implemented. This requires that the API abides by common standards that enable a client application and web service to agree on which data formats to use, and the structure of the data that is exchanged between client applications and the web service.
- **Service evolution**. The web service should be able to evolve and add (or remove) functionality independently from client applications. Existing client applications should be able to continue to operate unmodified as the features provided by the web service change. All functionality should also be discoverable, so that client applications can fully utilize it.

The purpose of this guidance is to describe the issues that you should consider when designing a web API.

Introduction to Representational State Transfer (REST)

In his dissertation in 2000, Roy Fielding proposed an alternative architectural approach to structuring the operations exposed by web services; REST. REST is an architectural style for building distributed systems based on hypermedia. A primary advantage of the REST model is that it is based on open standards and does not bind the implementation of the model or the client applications that access it to any specific implementation. For example, a REST web service could be implemented by using the Microsoft ASP.NET Web API, and client applications could be developed by using any language and toolset that can generate HTTP requests and parse HTTP responses.

> **NOTE**
> REST is actually independent of any underlying protocol and is not necessarily tied to HTTP. However, most common implementations of systems that are based on REST utilize HTTP as the application protocol for sending and receiving requests. This document focuses on mapping REST principles to systems designed to operate using HTTP.

The REST model uses a navigational scheme to represent objects and services over a network (referred to as *resources*). Many systems that implement REST typically use the HTTP protocol to transmit requests to access these resources. In these systems, a client application submits a request in the form of a URI that identifies a resource, and an HTTP method (the most common being GET, POST, PUT, or DELETE) that indicates the operation to be performed on that resource. The body of the HTTP request contains the data required to perform the operation. The important point to understand is that REST defines a stateless request model. HTTP requests should be independent

and may occur in any order, so attempting to retain transient state information between requests is not feasible. The only place where information is stored is in the resources themselves, and each request should be an atomic operation. Effectively, a REST model implements a finite state machine where a request transitions a resource from one well-defined non-transient state to another.

> **NOTE**
>
> The stateless nature of individual requests in the REST model enables a system constructed by following these principles to be highly scalable. There is no need to retain any affinity between a client application making a series of requests and the specific web servers handling those requests.

Another crucial point in implementing an effective REST model is to understand the relationships between the various resources to which the model provides access. These resources are typically organized as collections and relationships. For example, suppose that a quick analysis of an ecommerce system shows that there are two collections in which client applications are likely to be interested: orders and customers. Each order and customer should have its own unique key for identification purposes. The URI to access the collection of orders could be something as simple as */orders*, and similarly the URI for retrieving all customers could be */customers*. Issuing an HTTP GET request to the */orders* URI should return a list representing all orders in the collection encoded as an HTTP response:

```
GET http://adventure-works.com/orders HTTP/1.1
...
```

The response shown below encodes the orders as a JSON list structure:

```
HTTP/1.1 200 OK
...
Date: Fri, 22 Aug 2014 08:49:02 GMT
Content-Length: ...
[{"orderId":1,"orderValue":99.90,"productId":1,"quantity":1},
{"orderId":2,"orderValue":10.00,"productId":4,"quantity":2},
{"orderId":3,"orderValue":16.60,"productId":2,"quantity":4},
{"orderId":4,"orderValue":25.90,"productId":3,"quantity":1},
{"orderId":5,"orderValue":99.90,"productId":1,"quantity":1}]
```

To fetch an individual order requires specifying the identifier for the order from the *orders* resource, such as */orders/2*:

```
GET http://adventure-works.com/orders/2 HTTP/1.1
...
```

```
HTTP/1.1 200 OK
...
Date: Fri, 22 Aug 2014 08:49:02 GMT
Content-Length: ...
{"orderId":2,"orderValue":10.00,"productId":4,"quantity":2}
```

NOTE

For simplicity, these examples show the information in responses being returned as JSON text data. However, there is no reason why resources should not contain any other type of data supported by HTTP, such as binary or encrypted information; the content-type in the HTTP response should specify the type. Also, a REST model may be able to return the same data in different formats, such as XML or JSON. In this case, the web service should be able to perform content negotiation with the client making the request. The request can include an *Accept* header which specifies the preferred format that the client would like to receive and the web service should attempt to honor this format if at all possible.

Notice that the response from a REST request makes use of the standard HTTP status codes. For example, a request that returns valid data should include the HTTP response code 200 (OK), while a request that fails to find or delete a specified resource should return a response that includes the HTTP status code 404 (Not Found).

Design and structure of a RESTful web API

The keys to designing a successful web API are simplicity and consistency. A Web API that exhibits these two factors makes it easier to build client applications that need to consume the API.

A RESTful web API is focused on exposing a set of connected resources, and providing the core operations that enable an application to manipulate these resources and easily navigate between them. For this reason, the URIs that constitute a typical RESTful web API should be oriented towards the data that it exposes, and use the facilities provided by HTTP to operate on this data. This approach requires a different mindset from that typically employed when designing a set of classes in an object-oriented API which tends to be more motivated by the behavior of objects and classes. Additionally, a RESTful web API should be stateless and not depend on operations being invoked in a particular sequence. The following sections summarize the points you should consider when designing a RESTful web API.

Organizing the web API around resources

TIP

The URIs exposed by a REST web service should be based on nouns (the data to which the web API provides access) and not verbs (what an application can do with the data).

Focus on the business entities that the web API exposes. For example, in a web API designed to support the ecommerce system described earlier, the primary entities are customers and orders. Processes such as the act of placing an order can be achieved by providing an HTTP POST operation that takes the order information and adds it to the list of orders for the customer. Internally, this POST operation can perform tasks such as checking stock levels, and billing the customer. The HTTP response can indicate whether the order was placed successfully or not. Also note that a resource does not have to be based on a single physical data item. As an example, an order resource might be implemented internally by using information aggregated from many rows spread across several tables in a relational database but presented to the client as a single entity.

TIP

Avoid designing a REST interface that mirrors or depends on the internal structure of the data that it exposes. REST is about more than implementing simple CRUD (Create, Retrieve, Update, Delete) operations over separate tables in a relational database. The purpose of REST is to map business entities and the operations that an application can perform on these entities to the physical implementation of these entities, but a client should not be exposed to these physical details.

Individual business entities rarely exist in isolation (although some singleton objects may exist), but instead tend to be grouped together into collections. In REST terms, each entity and each collection are resources. In a RESTful web API, each collection has its own URI within the web service, and performing an HTTP GET request over a URI for a

collection retrieves a list of items in that collection. Each individual item also has its own URI, and an application can submit another HTTP GET request using that URI to retrieve the details of that item. You should organize the URIs for collections and items in a hierarchical manner. In the ecommerce system, the URI */customers* denotes the customer's collection, and */customers/5* retrieves the details for the single customer with the ID 5 from this collection. This approach helps to keep the web API intuitive.

> **TIP**
> Adopt a consistent naming convention in URIs; in general it helps to use plural nouns for URIs that reference collections.

You also need to consider the relationships between different types of resources and how you might expose these associations. For example, customers may place zero or more orders. A natural way to represent this relationship would be through a URI such as */customers/5/orders* to find all the orders for customer 5. You might also consider representing the association from an order back to a specific customer through a URI such as */orders/99/customer* to find the customer for order 99, but extending this model too far can become cumbersome to implement. A better solution is to provide navigable links to associated resources, such as the customer, in the body of the HTTP response message returned when the order is queried. This mechanism is described in more detail in the section Using the HATEOAS Approach to Enable Navigation To Related Resources later in this guidance.

In more complex systems there may be many more types of entity, and it can be tempting to provide URIs that enable a client application to navigate through several levels of relationships, such as */customers/1/orders/99/products* to obtain the list of products in order 99 placed by customer 1. However, this level of complexity can be difficult to maintain and is inflexible if the relationships between resources change in the future. Rather, you should seek to keep URIs relatively simple. Bear in mind that once an application has a reference to a resource, it should be possible to use this reference to find items related to that resource. The preceding query can be replaced with the URI */customers/1/orders* to find all the orders for customer 1, and then query the URI */orders/99/products* to find the products in this order (assuming order 99 was placed by customer 1).

> **TIP**
> Avoid requiring resource URIs more complex than *collection/item/collection*.

Another point to consider is that all web requests impose a load on the web server, and the greater the number of requests the bigger the load. You should attempt to define your resources to avoid "chatty" web APIs that expose a large number of small resources. Such an API may require a client application to submit multiple requests to find all the data that it requires. It may be beneficial to denormalize data and combine related information together into bigger resources that can be retrieved by issuing a single request. However, you need to balance this approach against the overhead of fetching data that might not be frequently required by the client. Retrieving large objects can increase the latency of a request and incur additional bandwidth costs for little advantage if the additional data is not often used.

Avoid introducing dependencies between the web API to the structure, type, or location of the underlying data sources. For example, if your data is located in a relational database, the web API does not need to expose each table as a collection of resources. Think of the web API as an abstraction of the database, and if necessary introduce a mapping layer between the database and the web API. In this way, if the design or implementation of the database changes (for example, you move from a relational database containing a collection of normalized tables to a denormalized NoSQL storage system such as a document database) client applications are insulated from these changes.

> **TIP**
> The source of the data that underpins a web API does not have to be a data store; it could be another service or line-of-business application or even a legacy application running on-premises within an organization.

Finally, it might not be possible to map every operation implemented by a web API to a specific resource. You can handle such *non-resource* scenarios through HTTP GET requests that invoke a piece of functionality and return the results as an HTTP response message. A web API that implements simple calculator-style operations such as add and subtract could provide URIs that expose these operations as pseudo resources and utilize the query string to specify the parameters required. For example a GET request to the URI */add?operand1=99&operand2=1* could return a response message with the body containing the value 100, and GET request to the URI */subtract?operand1=50&operand2=20* could return a response message with the body containing the value 30. However, only use these forms of URIs sparingly.

Defining operations in terms of HTTP methods

The HTTP protocol defines a number of methods that assign semantic meaning to a request. The common HTTP methods used by most RESTful web APIs are:

- **GET**, to retrieve a copy of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST**, to create a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.
- **PUT**, to replace or update the resource at the specified URI. The body of the request message specifies the resource to be modified and the values to be applied.
- **DELETE**, to remove the resource at the specified URI.

> **NOTE**
> The HTTP protocol also defines other less commonly-used methods, such as PATCH which is used to request selective updates to a resource, HEAD which is used to request a description of a resource, OPTIONS which enables a client information to obtain information about the communication options supported by the server, and TRACE which allows a client to request information that it can use for testing and diagnostics purposes.

The effect of a specific request should depend on whether the resource to which it is applied is a collection or an individual item. The following table summarizes the common conventions adopted by most RESTful implementations using the ecommerce example. Note that not all of these requests might be implemented; it depends on the specific scenario.

RESOURCE	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers (*if implemented*)	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists, otherwise return an error	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1 (*if implemented*)	Remove all orders for customer 1(*if implemented*)

The purpose of GET and DELETE requests are relatively straightforward, but there is scope for confusion concerning the purpose and effects of POST and PUT requests.

A POST request should create a new resource with data provided in the body of the request. In the REST model, you frequently apply POST requests to resources that are collections; the new resource is added to the collection.

> **NOTE**
> You can also define POST requests that trigger some functionality (and that don't necessarily return data), and these types of request can be applied to collections. For example you could use a POST request to pass a timesheet to a payroll processing service and get the calculated taxes back as a response.

A PUT request is intended to modify an existing resource. If the specified resource does not exist, the PUT request could return an error (in some cases, it might actually create the resource). PUT requests are most frequently applied to resources that are individual items (such as a specific customer or order), although they can be applied to collections, although this is less-commonly implemented. Note that PUT requests are idempotent whereas POST requests are not; if an application submits the same PUT request multiple times the results should always be the same (the same resource will be modified with the same values), but if an application repeats the same POST request the result will be the creation of multiple resources.

> **NOTE**
> Strictly speaking, an HTTP PUT request replaces an existing resource with the resource specified in the body of the request. If the intention is to modify a selection of properties in a resource but leave other properties unchanged, then this should be implemented by using an HTTP PATCH request. However, many RESTful implementations relax this rule and use PUT for both situations.

Processing HTTP requests

The data included by a client application in many HTTP requests, and the corresponding response messages from the web server, could be presented in a variety of formats (or media types). For example, the data that specifies the details for a customer or order could be provided as XML, JSON, or some other encoded and compressed format. A RESTful web API should support different media types as requested by the client application that submits a request.

When a client application sends a request that returns data in the body of a message, it can specify the media types it can handle in the Accept header of the request. The following code illustrates an HTTP GET request that retrieves the details of customer 1 and requests the result to be returned as JSON (the client should still examine the media type of the data in the response to verify the format of the data returned):

```
GET http://adventure-works.com/orders/2 HTTP/1.1
...
Accept: application/json
...
```

If the web server supports this media type, it can reply with a response that includes Content-Type header that specifies the format of the data in the body of the message:

> **NOTE**
> For maximum interoperability, the media types referenced in the Accept and Content-Type headers should be recognized MIME types rather than some custom media type.

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

If the web server does not support the requested media type, it can send the data in a different format. IN all cases it must specify the media type (such as *application/json*) in the Content-Type header. It is the responsibility of the client application to parse the response message and interpret the results in the message body appropriately.

Note that in this example, the web server successfully retrieves the requested data and indicates success by passing back a status code of 200 in the response header. If no matching data is found, it should instead return a status code of 404 (not found) and the body of the response message can contain additional information. The format of this information is specified by the Content-Type header, as shown in the following example:

```
GET http://adventure-works.com/orders/222 HTTP/1.1
...
Accept: application/json
...
```

Order 222 does not exist, so the response message looks like this:

```
HTTP/1.1 404 Not Found
...
Content-Type: application/json; charset=utf-8
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
{"message":"No such order"}
```

When an application sends an HTTP PUT request to update a resource, it specifies the URI of the resource and provides the data to be modified in the body of the request message. It should also specify the format of this data by using the Content-Type header. A common format used for text-based information is *application/x-www-form-urlencoded*, which comprises a set of name/value pairs separated by the & character. The next example shows an HTTP PUT request that modifies the information in order 1:

```
PUT http://adventure-works.com/orders/1 HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
ProductID=3&Quantity=5&OrderValue=250
```

If the modification is successful, it should ideally respond with an HTTP 204 status code, indicating that the process has been successfully handled, but that the response body contains no further information. The Location header in the response contains the URI of the newly updated resource:

```
HTTP/1.1 204 No Content
...
Location: http://adventure-works.com/orders/1
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
```

> **TIP**
> If the data in an HTTP PUT request message includes date and time information, make sure that your web service accepts dates and times formatted following the ISO 8601 standard.

If the resource to be updated does not exist, the web server can respond with a Not Found response as described earlier. Alternatively, if the server actually creates the object itself it could return the status codes HTTP 200 (OK) or HTTP 201 (Created) and the response body could contain the data for the new resource. If the Content-Type header of the request specifies a data format that the web server cannot handle, it should respond with HTTP status code 415 (Unsupported Media Type).

> **TIP**
> Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

The format of an HTTP POST requests that create new resources are similar to those of PUT requests; the message body contains the details of the new resource to be added. However, the URI typically specifies the collection to which the resource should be added. The following example creates a new order and adds it to the orders collection:

```
POST http://adventure-works.com/orders HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
productID=5&quantity=15&orderValue=400
```

If the request is successful, the web server should respond with a message code with HTTP status code 201 (Created). The Location header should contain the URI of the newly created resource, and the body of the response should contain a copy of the new resource; the Content-Type header specifies the format of this data:

```
HTTP/1.1 201 Created
...
Content-Type: application/json; charset=utf-8
Location: http://adventure-works.com/orders/99
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
{"orderID":99,"productID":5,"quantity":15,"orderValue":400}
```

> **TIP**
> If the data provided by a PUT or POST request is invalid, the web server should respond with a message with HTTP status code 400 (Bad Request). The body of this message can contain additional information about the problem with the request and the formats expected, or it can contain a link to a URL that provides more details.

To remove a resource, an HTTP DELETE request simply provides the URI of the resource to be deleted. The following example attempts to remove order 99:

```
DELETE http://adventure-works.com/orders/99 HTTP/1.1
...
```

If the delete operation is successful, the web server should respond with HTTP status code 204, indicating that the process has been successfully handled, but that the response body contains no further information (this is the same response returned by a successful PUT operation, but without a Location header as the resource no longer exists.) It is also possible for a DELETE request to return HTTP status code 200 (OK) or 202 (Accepted) if the deletion is performed asynchronously.

```
HTTP/1.1 204 No Content
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
```

If the resource is not found, the web server should return a 404 (Not Found) message instead.

> **TIP**
> If all the resources in a collection need to be deleted, enable an HTTP DELETE request to be specified for the URI of the collection rather than forcing an application to remove each resource in turn from the collection.

Filtering and paginating data

You should endeavour to keep the URIs simple and intuitive. Exposing a collection of resources through a single URI assists in this respect, but it can lead to applications fetching large amounts of data when only a subset of the information is required. Generating a large volume of traffic impacts not only the performance and scalability of the web server but also adversely affect the responsiveness of client applications requesting the data.

For example, if orders contain the price paid for the order, a client application that needs to retrieve all orders that have a cost over a specific value might need to retrieve all orders from the */orders* URI and then filter these orders locally. Clearly this process is highly inefficient; it wastes network bandwidth and processing power on the server hosting the web API.

One solution may be to provide a URI scheme such as */orders/ordervalue_greater_than_n* where *n* is the order price, but for all but a limited number of prices such an approach is impractical. Additionally, if you need to query orders based on other criteria, you can end up being faced with providing with a long list of URIs with possibly non-intuitive names.

A better strategy to filtering data is to provide the filter criteria in the query string that is passed to the web API, such as */orders?ordervaluethreshold=n*. In this example, the corresponding operation in the web API is responsible for parsing and handling the `ordervaluethreshold` parameter in the query string and returning the filtered results in the HTTP response.

Some simple HTTP GET requests over collection resources could potentially return a large number of items. To combat the possibility of this occurring you should design the web API to limit the amount of data returned by any single request. You can achieve this by supporting query strings that enable the user to specify the maximum number of items to be retrieved (which could itself be subject to an upperbound limit to help prevent Denial of Service attacks), and a starting offset into the collection. For example, the query string in the URI */orders?limit=25&offset=50* should retrieve 25 orders starting with the 50th order found in the orders collection. As with filtering data, the operation that implements the GET request in the web API is responsible for parsing and handling the `limit` and `offset` parameters in the query string. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection. You might also consider other intelligent paging strategies; for more information, see API Design Notes: Smart Paging

You can follow a similar strategy for sorting data as it is fetched; you could provide a sort parameter that takes a field name as the value, such as */orders?sort=ProductID*. However, note that this approach can have a deleterious effect on caching (query string parameters form part of the resource identifier used by many cache implementations as the key to cached data).

You can extend this approach to limit (project) the fields returned if a single resource item contains a large amount of data. For example, you could use a query string parameter that accepts a comma-delimited list of fields, such as */orders?fields=ProductID,Quantity*.

> **TIP**
> Give all optional parameters in query strings meaningful defaults. For example, set the `limit` parameter to 10 and the `offset` parameter to 0 if you implement pagination, set the sort parameter to the key of the resource if you implement ordering, and set the `fields` parameter to all fields in the resource if you support projections.

Handling large binary resources

A single resource may contain large binary fields, such as files or images. To overcome the transmission problems caused by unreliable and intermittent connections and to improve response times, consider providing operations that enable such resources to be retrieved in chunks by the client application. To do this, the web API should support the Accept-Ranges header for GET requests for large resources, and ideally implement HTTP HEAD requests for these resources. The Accept-Ranges header indicates that the GET operation supports partial results, and that a client application can submit GET requests that return a subset of a resource specified as a range of bytes. A HEAD request is similar to a GET request except that it only returns a header that describes the resource and an empty message body. A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests. The following example shows a HEAD request that obtains information about a product image:

```
HEAD http://adventure-works.com/products/10?fields=productImage HTTP/1.1
...
```

The response message contains a header that includes the size of the resource (4580 bytes), and the Accept-Ranges header that the corresponding GET operation supports partial results:

```
HTTP/1.1 200 OK
...
Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
...
```

The client application can use this information to construct a series of GET operations to retrieve the image in smaller chunks. The first request fetches the first 2500 bytes by using the Range header:

```
GET http://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
...
```

The response message indicates that this is a partial response by returning HTTP status code 206. The Content-Length header specifies the actual number of bytes returned in the message body (not the size of the resource), and the Content-Range header indicates which part of the resource this is (bytes 0-2499 out of 4580):

```
HTTP/1.1 206 Partial Content
...
Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580
...
_{binary data not shown}_
```

A subsequent request from the client application can retrieve the remainder of the resource by using an appropriate Range header:

```
GET http://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=2500-
...
```

The corresponding result message should look like this:

```
HTTP/1.1 206 Partial Content
...
Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2080
Content-Range: bytes 2500-4580/4580
...
```

Using the HATEOAS approach to enable navigation to related resources

One of the primary motivations behind REST is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

> **NOTE**
> Currently there are no standards or specifications that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible solution.

As an example, to handle the relationship between customers and orders, the data returned in the response for a specific order should contain URIs in the form of a hyperlink identifying the customer that placed the order, and the operations that can be performed on that customer.

```
GET http://adventure-works.com/orders/3 HTTP/1.1
Accept: application/json
...
```

The body of the response message contains a `links` array (highlighted in the code example) that specifies the nature of the relationship (*Customer*), the URI of the customer (*http://adventure-works.com/customers/3*), how to retrieve the details of this customer (*GET*), and the MIME types that the web server supports for retrieving this information (*text/xml* and *application/json*). This is all the information that a client application needs to be able to fetch the details of the customer. Additionally, the Links array also includes links for the other operations that can be performed, such as PUT (to modify the customer, together with the format that the web server expects the client to provide), and DELETE.

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"orderID":3,"productID":2,"quantity":4,"orderValue":16.60,"links":[(some links omitted)
{"rel":"customer","href":" http://adventure-works.com/customers/3", "action":"GET","types":
["text/xml","application/json"]},{"rel":"
customer","href":" http://adventure-works.com /customers/3", "action":"PUT","types":["application/x-www-form-
urlencoded"]},{"rel":"customer","href":" http://adventure-works.com /customers/3","action":"DELETE","types":[]}]}
```

For completeness, the Links array should also include self-referencing information pertaining to the resource that has been retrieved. These links have been omitted from the previous example, but are highlighted in the following code. Notice that in these links, the relationship *self* has been used to indicate that this is a reference to the resource being returned by the operation:

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"orderID":3,"productID":2,"quantity":4,"orderValue":16.60,"links":[{"rel":"self","href":" http://adventure-
works.com/orders/3", "action":"GET","types":["text/xml","application/json"]},{"rel":" self","href":"
http://adventure-works.com /orders/3", "action":"PUT","types":["application/x-www-form-urlencoded"]},
{"rel":"self","href":" http://adventure-works.com /orders/3", "action":"DELETE","types":[]},{"rel":"customer",
"href":" http://adventure-works.com /customers/3", "action":"GET","types":["text/xml","application/json"]},
{"rel":" customer" (customer links omitted)}]}
```

For this approach to be effective, client applications must be prepared to retrieve and parse this additional information.

Versioning a RESTful web API

It is highly unlikely that in all but the simplest of situations that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full control over that API, the developer does not have the same degree of control over client applications which may be built by third party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

No versioning

This is the simplest approach, and may be acceptable for some internal APIs. Big changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will simply ignore it.

For example, a request to the URI *http://adventure-works.com/customers/3* should return the details of a single customer containing `id`, `name`, and `address` fields expected by the client application:

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

> **NOTE**
>
> For the purposes of simplicity and clarity, the example responses shown in this section do not include HATEOAS links.

If the `DateCreated` field is added to the schema of the customer resource, then the response would look like this:

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA
98053"}
```

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from functioning correctly. In these situations you should consider one of the following approaches.

URI versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Extending the previous example, if the `address` field is restructured into sub-fields containing each constituent part of the address (such as `streetAddress`, `city`, `state`, and `zipCode`), this version of the resource could be exposed through a URI containing a version number, such as http://adventure-works.com/v2/customers/3:

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

This versioning mechanism is very simple but depends on the server routing the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as *http://adventure-works.com/customers/3?version=2*. The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also suffers from the same complications for implementing HATEOAS as the URI versioning mechanism.

> **NOTE**
>
> Some older web browsers and web proxies will not cache responses for requests that include a query string in the URL. This can have an adverse impact on performance for web applications that use a web API and that run from within such a web browser.

Header versioning

Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples utilize a custom header named *Custom-Header*. The value of this header indicates the version of web API.

Version 1:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
...
Custom-Header: api-version=1
...
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Version 2:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
...
Custom-Header: api-version=2
...
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

Note that as with the previous two approaches, implementing HATEOAS requires including the appropriate custom header in any links.

Media type versioning

When a client application sends an HTTP GET request to a web server it should stipulate the format of the content that it can handle by using an Accept header, as described earlier in this guidance. Frequently the purpose of the *Accept* header is to allow the client application to specify whether the body of the response should be XML, JSON, or some other common format that the client can parse. However, it is possible to define custom media types that include information enabling the client application to indicate which version of a resource it is expecting. The

following example shows a request that specifies an *Accept* header with the value *application/vnd.adventure-works.v1+json*. The *vnd.adventure-works.v1* element indicates to the web server that it should return version 1 of the resource, while the *json* element specifies that the format of the response body should be JSON:

```
GET http://adventure-works.com/customers/3 HTTP/1.1
...
Accept: application/vnd.adventure-works.v1+json
...
```

The code handling the request is responsible for processing the *Accept* header and honoring it as far as possible (the client application may specify multiple formats in the *Accept* header, in which case the web server can choose the most appropriate format for the response body). The web server confirms the format of the data in the response body by using the Content-Type header:

```
HTTP/1.1 200 OK
...
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8
...
Content-Length: ...
{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

If the Accept header does not specify any known media types, the web server could generate an HTTP 406 (Not Acceptable) response message or return a message with a default media type.

This approach is arguably the purest of the versioning mechanisms and lends itself naturally to HATEOAS, which can include the MIME type of related data in resource links.

> **NOTE**
>
> When you select a versioning strategy, you should also consider the implications on performance, especially caching on the web server. The URI versioning and Query String versioning schemes are cache-friendly inasmuch as the same URI/query string combination refers to the same data each time.
>
> The Header versioning and Media Type versioning mechanisms typically require additional logic to examine the values in the custom header or the Accept header. In a large-scale environment, many clients using different versions of a web API can result in a significant amount of duplicated data in a server-side cache. This issue can become acute if a client application communicates with a web server through a proxy that implements caching, and that only forwards a request to the web server if it does not currently hold a copy of the requested data in its cache.

More information

- The RESTful Cookbook contains an introduction to building RESTful APIs.
- The Web API Checklist contains a useful list of items to consider when designing and implementing a Web API.

API implementation guidance

11/15/2016 • 62 min to read • Edit on GitHub

Contributors

Masashi Narumoto • Joseph Molnar • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • alancameronwills • Den Delimarsky • Christopher Bennage • pitrov • Huachao Mao • Trent Swanson • Dene Hager

patterns & practices
proven practices for predictable results

Some topics in this guidance are under discussion and may change in the future. We welcome your feedback!

Overview

A carefully-designed RESTful web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data. This guidance focusses on best practices for implementing a web API and publishing it to make it available to client applications. Security concerns are described separately in the API Security Guidance document. You can find detailed information about web API design in the API Design Guidance document.

Considerations for implementing a RESTful web API

The following sections illustrate best practice for using the ASP.NET Web API template to build a RESTful web API. For detailed information on using the Web API template, visit the Learn About ASP.NET Web API page on the Microsoft website.

Considerations for implementing request routing

In a service implemented by using the ASP.NET Web API, each request is routed to a method in a *controller* class. The Web API framework provides two primary options for implementing routing; *convention-based* routing and *attribute-based* routing. Consider the following points when you determine the best way to route requests in your web API:

- **Understand the limitations and requirements of convention-based routing**.

  By default, the Web API framework uses convention-based routing. The Web API framework creates an initial routing table that contains the following entry:

  ```
  config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
  );
  ```

  Routes can be generic, comprising literals such as *api* and variables such as *{controller}* and *{id}*. Convention-based routing allows some elements of the route to be optional. The Web API framework determines which method to invoke in the controller by matching the HTTP method in the request to the initial part of the method name in the API, and then by matching any optional parameters. For example, if a controller named

orders contains the methods *GetAllOrders()* or *GetOrderByInt(int id)* then the GET request *http://www.adventure-works.com/api/orders/* will be directed to the method *GetAlllOrders()* and the GET request *http://www.adventure-works.com/api/orders/99* will be routed to the method *GetOrderByInt(int id)*. If there is no matching method available that begins with the prefix Get in the controller, the Web API framework replies with an HTTP 405 (Method Not Allowed) message. Additionally, name of the parameter (id) specified in the routing table must be the same as the name of the parameter for the *GetOrderById* method, otherwise the Web API framework will reply with an HTTP 404 (Not Found) response.

The same rules apply to POST, PUT, and DELETE HTTP requests; a PUT request that updates the details of order 101 would be directed to the URI *http://www.adventure-works.com/api/orders/101*, the body of the message will contain the new details of the order, and this information will be passed as a parameter to a method in the orders controller with a name that starts with the prefix *Put*, such as *PutOrder*.

The default routing table will not match a request that references child resources in a RESTful web API, such as *http://www.adventure-works.com/api/customers/1/orders* (find the details of all orders placed by customer 1). To handle these cases, you can add custom routes to the routing table:

```
config.Routes.MapHttpRoute(
    name: "CustomerOrdersRoute",
    routeTemplate: "api/customers/{custId}/orders",
    defaults: new { controller="Customers", action="GetOrdersForCustomer" })
);
```

This route directs requests that match the URI to the *GetOrdersForCustomer* method in the *Customers* controller. This method must take a single parameter named *custI*:

```
public class CustomersController : ApiController
{
    ...
    public IEnumerable<Order> GetOrdersForCustomer(int custId)
    {
        // Find orders for the specified customer
        var orders = ...
        return orders;
    }
    ...
}
```

TIP

Utilize the default routing wherever possible and avoid defining many complicated custom routes as this can result in brittleness (it is very easy to add methods to a controller that result in ambiguous routes) and reduced performance (the bigger the routing table, the more work the Web API framework has to do to work out which route matches a given URI). Keep the API and routes simple. For more information, see the section Organizing the Web API Around Resources in the API Design Guidance. If you must define custom routes, a preferable approach is to use attribute-based routing described later in this section.

For more information about convention-based routing, see the page Routing in ASP.NET Web API on the Microsoft website.

- **Avoid ambiguity in routing**.

  Convention-based routing can result in ambiguous pathways if multiple methods in a controller match the same route. In these situations, the Web API framework responds with an HTTP 500 (Internal Server Error) response message containing the text "Multiple actions were found that match the request".

- **Prefer attribute-based routing**.

Attribute-based routing provides an alternative means for connecting routes to methods in a controller. Rather than relying on the pattern-matching features of convention-based routing, you can explicitly annotate methods in a controller with the details of the route to which they should be associated. This approach help to remove possible ambiguities. Furthermore, as explicit routes are defined at design time this approach is more efficient than convention-based routing at runtime. The following code shows how to apply the *Route* attribute to methods in the Customers controller. These methods also use the HttpGet attribute to indicate that they should respond to *HTTP GET* requests. This attribute enables you to name your methods using any convenient naming scheme rather than that expected by convention-based routing. You can also annotate methods with the *HttpPost*, *HttpPut*, and *HttpDelete* attributes to define methods that respond to other types of HTTP requests.

```
public class CustomersController : ApiController
{
    ...
    [Route("api/customers/{id}")]
    [HttpGet]
    public Customer FindCustomerByID(int id)
    {
        // Find the matching customer
        var customer = ...
        return customer;
    }
    ...
    [Route("api/customers/{id}/orders")]
    [HttpGet]
    public IEnumerable<Order> FindOrdersForCustomer(int id)
    {
        // Find orders for the specified customer
        var orders = ...
        return orders;
    }
    ...
}
```

Attribute-based routing also has the useful side-effect of acting as documentation for developers needing to maintain the code in the future; it is immediately clear which method belongs to which route, and the *HttpGet* attribute clarifies the type of HTTP request to which the method responds.

Attribute-based routing enables you to define constraints which restrict how the parameters are matched. Constraints can specify the type of the parameter, and in some cases they can also indicate the acceptable range of parameter values. In the following example, the id parameter to the *FindCustomerByID* method must be a non-negative integer. If an application submits an HTTP GET request with a negative customer number, the Web API framework will respond with an HTTP 405 (Method Not Allowed) message:

```
public class CustomersController : ApiController
{
    ...
    [Route("api/customers/{id:int:min(0)}")]
    [HttpGet]
    public Customer FindCustomerByID(int id)
    {
        // Find the matching customer
        var customer = ...
        return customer;
    }
    ...
}
```

For more information on attribute-based routing, see the page Attribute Routing in Web API 2 on the Microsoft website.

- **Support Unicode characters in routes**.

  The keys used to identify resources in GET requests could be strings. Therefore, in a global application, you may need to support URIs that contain non-English characters.

- **Distinguish methods that should not be routed**.

  If you are using convention-based routing, indicate methods that do not correspond to HTTP actions by decorating them with the *NonAction* attribute. This typically applies to helper methods defined for use by other methods within a controller, and this attribute will prevent these methods from being matched and invoked by an errant HTTP request.

- **Consider the benefits and tradeoffs of placing the API in a subdomain**.

  By default, the ASP.NET web API organizes APIs into the */api* directory in a domain, such as *http://www.adventure-works.com/api/orders*. This directory resides in the same domain as any other services exposed by the same host. It may be beneficial to split the web API out into its own subdomain running on a separate host, with URIs such as *http://api.adventure-works.com/orders*. This separation enables you to partition and scale the web API more effectively without affecting any other web applications or services running in the *www.adventure-works.com* domain.

  However, placing a web API in a different subdomain can also lead to security concerns. Any web applications or services hosted at *www.adventure-works.com* that invoke a web API running elsewhere may violate the same-origin policy of many web browsers. In this situation, it will be necessary to enable cross-origin resource sharing (CORS) between the hosts. For more information, see the API Security Guidance document.

Considerations for processing requests

Once a request from a client application has been successfully routed to a method in a web API, the request must be processed in as efficient manner as possible. Consider the following points when you implement the code to handle requests:

- **GET, PUT, DELETE, HEAD, and PATCH actions should be idempotent**.

  The code that implements these requests should not impose any side-effects. The same request repeated over the same resource should result in the same state. For example, sending multiple DELETE requests to the same URI should have the same effect, although the HTTP status code in the response messages may be different (the first DELETE request might return status code 204 (No Content) while a subsequent DELETE request might return status code 404 (Not Found)).

> **NOTE**
> The article Idempotency Patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.

- **POST actions that create new resources should do so without unrelated side-effects**.

  If a POST request is intended to create a new resource, the effects of the request should be limited to the new resource (and possibly any directly related resources if there is some sort of linkage involved) For example, in an ecommerce system, a POST request that creates a new order for a customer might also amend inventory levels and generate billing information, but it should not modify information not directly related to the order or have any other side-effects on the overall state of the system.

- **Avoid implementing chatty POST, PUT, and DELETE operations**.

  Support POST, PUT and DELETE requests over resource collections. A POST request can contain the details

for multiple new resources and add them all to the same collection, a PUT request can replace the entire set of resources in a collection, and a DELETE request can remove an entire collection.

Note that the OData support included in ASP.NET Web API 2 provides the ability to batch requests. A client application can package up several web API requests and send them to the server in a single HTTP request, and receive a single HTTP response that contains the replies to each request. For more information, see the page Introducing Batch Support in Web API and Web API OData on the Microsoft website.

- **Abide by the HTTP protocol when sending a response back to a client application**.

  A web API must return messages that contain the correct HTTP status code to enable the client to determine how to handle the result, the appropriate HTTP headers so that the client understands the nature of the result, and a suitably formatted body to enable the client to parse the result. If you are using the ASP.NET Web API template, the default strategy for implementing methods that respond to HTTP POST requests is simply to return a copy of the newly created resource, as illustrated by the following example:

```
public class CustomersController : ApiController
{
    ...
    [Route("api/customers")]
    [HttpPost]
    public Customer CreateNewCustomer(Customer customerDetails)
    {
        // Add the new customer to the repository
        // This method returns a customer with a unique ID allocated
        // by the repository
        var newCust = repository.Add(customerDetails);
        // Return the newly added customer
        return newCust;
    }
    ...
}
```

  If the POST operation is successful, the Web API framework creates an HTTP response with status code 200 (OK) and the details of the customer as the message body. However, in this case, according to the HTTP protocol, a POST operation should return status code 201 (Created) and the response message should include the URI of the newly created resource in the Location header of the response message.

  To provide these features, return your own HTTP response message by using the `IHttpActionResult` interface. This approach gives you fine control over the HTTP status code, the headers in the response message, and even the format of the data in the response message body, as shown in the following code example. This version of the `CreateNewCustomer` method conforms more closely to the expectations of client following the HTTP protocol. The `Created` method of the `ApiController` class constructs the response message from the specified data, and adds the Location header to the results:

```
public class CustomersController : ApiController
{
    ...
    [Route("api/customers")]
    [HttpPost]
    public IHttpActionResult CreateNewCustomer(Customer customerDetails)
    {
        // Add the new customer to the repository
        var newCust = repository.Add(customerDetails);

        // Create a value for the Location header to be returned in the response
        // The URI should be the location of the customer including its ID,
        // such as http://adventure-works.com/api/customers/99
        var location = new Uri(...);

        // Return the HTTP 201 response,
        // including the location and the newly added customer
        return Created(location, newCust);
    }
    ...
}
```

- **Support content negotiation**.

  The body of a response message may contain data in a variety of formats. For example, an HTTP GET request could return data in JSON, or XML format. When the client submits a request, it can include an Accept header that specifies the data formats that it can handle. These formats are specified as media types. For example, a client that issues a GET request that retrieves an image can specify an Accept header that lists the media types that the client can handle, such as "image/jpeg, image/gif, image/png". When the web API returns the result, it should format the data by using one of these media types and specify the format in the Content-Type header of the response.

  If the client does not specify an Accept header, then use a sensible default format for the response body. As an example, the ASP.NET Web API framework defaults to JSON for text-based data.

  > **NOTE**
  > The ASP.NET Web API framework performs some automatic detection of Accept headers and handles them itself based on the type of the data in the body of the response message. For example, if the body of a response message contains a CLR (common language runtime) object, the ASP.NET Web API automatically formats the response as JSON with the Content-Type header of the response set to "application/json" unless the client indicates that it requires the results as XML, in which case the ASP.NET Web API framework formats the response as XML and sets the Content-Type header of the response to "text/xml". However, it may be necessary to handle Accept headers that specify different media types explicitly in the implementation code for an operation.

- **Provide links to support HATEOAS-style navigation and discovery of resources**.

  The API Design Guidance describes how following the HATEOAS approach enables a client to navigate and discover resources from an initial starting point. This is achieved by using links containing URIs; when a client issues an HTTP GET request to obtain a resource, the response should contain URIs that enable a client application to quickly locate any directly related resources. For example, in a web API that supports an e-commerce solution, a customer may have placed many orders. When a client application retrieves the details for a customer, the response should include links that enable the client application to send HTTP GET requests that can retrieve these orders. Additionally, HATEOAS-style links should describe the other operations (POST, PUT, DELETE, and so on) that each linked resource supports together with the corresponding URI to perform each request. This approach is described in more detail in the API Design Guidance document.

Currently there are no standards that govern the implementation of HATEOAS, but the following example illustrates one possible approach. In this example, an HTTP GET request that finds the details for a customer returns a response that include HATEOAS links that reference the orders for that customer:

```
GET http://adventure-works.com/customers/2 HTTP/1.1
Accept: text/json
...
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"CustomerID":2,"CustomerName":"Bert","Links":[
  {"rel":"self",
   "href":"http://adventure-works.com/customers/2",
   "action":"GET",
   "types":["text/xml","application/json"]},
  {"rel":"self",
   "href":"http://adventure-works.com/customers/2",
   "action":"PUT",
   "types":["application/x-www-form-urlencoded"]},
  {"rel":"self",
   "href":"http://adventure-works.com/customers/2",
   "action":"DELETE",
   "types":[]},
  {"rel":"orders",
   "href":"http://adventure-works.com/customers/2/orders",
   "action":"GET",
   "types":["text/xml","application/json"]},
  {"rel":"orders",
   "href":"http://adventure-works.com/customers/2/orders",
   "action":"POST",
   "types":["application/x-www-form-urlencoded"]}
]}
```

In this example, the customer data is represented by the `Customer` class shown in the following code snippet. The HATEOAS links are held in the `Links` collection property:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public List<Link> Links { get; set; }
    ...
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Action { get; set; }
    public string [] Types { get; set; }
}
```

The HTTP GET operation retrieves the customer data from storage and constructs a `Customer` object, and then populates the `Links` collection. The result is formatted as a JSON response message. Each link comprises the following fields:

- The relationship between the object being returned and the object described by the link. In this case "self" indicates that the link is a reference back to the object itself (similar to a `this` pointer in many object-oriented languages), and "orders" is the name of a collection containing the related order information.

- The hyperlink (`Href`) for the object being described by the link in the form of a URI.
- The type of HTTP request (`Action`) that can be sent to this URI.
- The format of any data (`Types`) that should be provided in the HTTP request or that can be returned in the response, depending on the type of the request.

  The HATEOAS links shown in the example HTTP response indicate that a client application can perform the following operations:

- An HTTP GET request to the URI *http://adventure-works.com/customers/2* to fetch the details of the customer (again). The data can be returned as XML or JSON.
- An HTTP PUT request to the URI *http://adventure-works.com/customers/2* to modify the details of the customer. The new data must be provided in the request message in x-www-form-urlencoded format.
- An HTTP DELETE request to the URI *http://adventure-works.com/customers/2* to delete the customer. The request does not expect any additional information or return data in the response message body.
- An HTTP GET request to the URI *http://adventure-works.com/customers/2/orders* to find all the orders for the customer. The data can be returned as XML or JSON.
- An HTTP PUT request to the URI *http://adventure-works.com/customers/2/orders* to create a new order for this customer. The data must be provided in the request message in x-www-form-urlencoded format.

Considerations for handling exceptions

By default, in the ASP.NET Web API framework, if an operation throws an uncaught exception the framework returns a response message with HTTP status code 500 (Internal Server Error). In many cases, this simplistic approach is not useful in isolation, and makes determining the cause of the exception difficult. Therefore you should adopt a more comprehensive approach to handling exceptions, considering the following points:

- **Capture exceptions and return a meaningful response to clients**.

  The code that implements an HTTP operation should provide comprehensive exception handling rather than letting uncaught exceptions propagate to the Web API framework. If an exception makes it impossible to complete the operation successfully, the exception can be passed back in the response message, but it should include a meaningful description of the error that caused the exception. The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. For example, if a user request causes a database update that violates a constraint (such as attempting to delete a customer that has outstanding orders), you should return status code 409 (Conflict) and a message body indicating the reason for the conflict. If some other condition renders the request unachievable, you can return status code 400 (Bad Request). You can find a full list of HTTP status codes on the Status Code Definitions page on the W3C website.

  The following code shows an example that traps different conditions and returns an appropriate response.

```
[HttpDelete]
[Route("customers/{id:int}")]
public IHttpActionResult DeleteCustomer(int id)
{
    try
    {
        // Find the customer to be deleted in the repository
        var customerToDelete = repository.GetCustomer(id);

        // If there is no such customer, return an error response
        // with status code 404 (Not Found)
        if (customerToDelete == null)
        {
                return NotFound();
        }

        // Remove the customer from the repository
        // The DeleteCustomer method returns true if the customer
        // was successfully deleted
        if (repository.DeleteCustomer(id))
        {
            // Return a response message with status code 204 (No Content)
            // To indicate that the operation was successful
            return StatusCode(HttpStatusCode.NoContent);
        }
        else
        {
            // Otherwise return a 400 (Bad Request) error response
            return BadRequest(Strings.CustomerNotDeleted);
        }
    }
    catch
    {
        // If an uncaught exception occurs, return an error response
        // with status code 500 (Internal Server Error)
        return InternalServerError();
    }
}
```

TIP

Do not include information that could be useful to an attacker attempting to penetrate your web API.For further information, visit the Exception Handling in ASP.NET Web API page on the Microsoft website.

NOTE

Many web servers trap error conditions themselves before they reach the web API. For example, if you configure authentication for a web site and the user fails to provide the correct authentication information, the web server should respond with status code 401 (Unauthorized). Once a client has been authenticated, your code can perform its own checks to verify that the client should be able access the requested resource. If this authorization fails, you should return status code 403 (Forbidden).

- **Handle exceptions in a consistent manner and log information about errors**.

  To handle exceptions in a consistent manner, consider implementing a global error handling strategy across the entire web API. You can achieve part of this by creating an exception filter that runs whenever a controller throws any unhandled exception that is not an `HttpResponseException` exception. This approach is described on the Exception Handling in ASP.NET Web API page on the Microsoft website.

  However, there are several situations where an exception filter will not catch an exception, including:

- Exceptions thrown from controller constructors.
- Exceptions thrown from message handlers.
- Exceptions thrown during routing.
- Exceptions thrown while serializing the content for a response message.

  To handle these cases, you may need to implement a more customized approach. You should also incorporate error logging which captures the full details of each exception; this error log can contain detailed information as long as it is not made accessible over the web to clients. The article Web API Global Error Handling on the Microsoft website shows one way of performing this task.

- **Distinguish between client-side errors and server-side errors**.

  The HTTP protocol distinguishes between errors that occur due to the client application (the HTTP 4xx status codes), and errors that are caused by a mishap on the server (the HTTP 5xx status codes). Make sure that you respect this convention in any error response messages.

Considerations for optimizing client-side data access

In a distributed environment such as that involving a web server and client applications, one of the primary sources of concern is the network. This can act as a considerable bottleneck, especially if a client application is frequently sending requests or receiving data. Therefore you should aim to minimize the amount of traffic that flows across the network. Consider the following points when you implement the code to retrieve and maintain data:

- **Support client-side caching**.

  The HTTP 1.1 protocol supports caching in clients and intermediate servers through which a request is routed by the use of the Cache-Control header. When a client application sends an HTTP GET request to the web API, the response can include a Cache-Control header that indicates whether the data in the body of the response can be safely cached by the client or an intermediate server through which the request has been routed, and for how long before it should expire and be considered out-of-date. The following example shows an HTTP GET request and the corresponding response that includes a Cache-Control header:

  ```
  GET http://adventure-works.com/orders/2 HTTP/1.1
  ...
  ```

  ```
  HTTP/1.1 200 OK
  ...
  Cache-Control: max-age=600, private
  Content-Type: text/json; charset=utf-8
  Content-Length: ...
  {"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
  ```

  In this example, the Cache-Control header specifies that the data returned should be expired after 600 seconds, and is only suitable for a single client and must not be stored in a shared cache used by other clients (it is *private*). The Cache-Control header could specify *public* rather than *private* in which case the data can be stored in a shared cache, or it could specify *no-store* in which case the data must **not** be cached by the client. The following code example shows how to construct a Cache-Control header in a response message:

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...
        // Create a Cache-Control header for the response
        var cacheControlHeader = new CacheControlHeaderValue();
        cacheControlHeader.Private = true;
        cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
        ...

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            CacheControlHeader = cacheControlHeader
        };
        return response;
    }
    ...
}
```

This code makes use of a custom `IHttpActionResult` class named `OkResultWithCaching`. This class enables the controller to set the cache header contents:

```
public class OkResultWithCaching<T> : OkNegotiatedContentResult<T>
{
    public OkResultWithCaching(T content, ApiController controller)
        : base(content, controller) { }

    public OkResultWithCaching(T content, IContentNegotiator contentNegotiator, HttpRequestMessage
request, IEnumerable<MediaTypeFormatter> formatters)
        : base(content, contentNegotiator, request, formatters) { }

    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }

    public override async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response = await base.ExecuteAsync(cancellationToken);

        response.Headers.CacheControl = this.CacheControlHeader;
        response.Headers.ETag = ETag;

        return response;
    }
}
```

NOTE

The HTTP protocol also defines the *no-cache* directive for the Cache-Control header. Rather confusingly, this directive does not mean "do not cache" but rather "revalidate the cached information with the server before returning it"; the data can still be cached, but it is checked each time it is used to ensure that it is still current.

Cache management is the responsibility of the client application or intermediate server, but if properly implemented it can save bandwidth and improve performance by removing the need to fetch data that has already been recently retrieved.

The *max-age* value in the Cache-Control header is only a guide and not a guarantee that the corresponding data won't change during the specified time. The web API should set the max-age to a suitable value depending on the expected volatility of the data. When this period expires, the client should discard the object from the cache.

> **NOTE**
>
> Most modern web browsers support client-side caching by adding the appropriate cache-control headers to requests and examining the headers of the results, as described. However, some older browsers will not cache the values returned from a URL that includes a query string. This is not usually an issue for custom client applications which implement their own cache management strategy based on the protocol discussed here.
>
> Some older proxies exhibit the same behavior and might not cache requests based on URLs with query strings. This could be an issue for custom client applications that connect to a web server through such a proxy.

- **Provide ETags to Optimize Query Processing**.

  When a client application retrieves an object, the response message can also include an *ETag* (Entity Tag). An ETag is an opaque string that indicates the version of a resource; each time a resource changes the Etag is also modified. This ETag should be cached as part of the data by the client application. The following code example shows how to add an ETag as part of the response to an HTTP GET request. This code uses the `GetHashCode` method of an object to generate a numeric value that identifies the object (you can override this method if necessary and generate your own hash using an algorithm such as MD5) :

```
public class OrdersController : ApiController
{
    ...
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...

        var hashedOrder = order.GetHashCode();
        string hashedOrderEtag = String.Format("\"{0}\"", hashedOrder);
        var eTag = new EntityTagHeaderValue(hashedOrderEtag);

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            ...,
            ETag = eTag
        };
        return response;
    }
    ...
}
```

  The response message posted by the web API looks like this:

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
ETag: "2147483648"
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

> **TIP**
> For security reasons, do not allow sensitive data or data returned over an authenticated (HTTPS) connection to be cached.

A client application can issue a subsequent GET request to retrieve the same resource at any time, and if the resource has changed (it has a different ETag) the cached version should be discarded and the new version added to the cache. If a resource is large and requires a significant amount of bandwidth to transmit back to the client, repeated requests to fetch the same data can become inefficient. To combat this, the HTTP protocol defines the following process for optimizing GET requests that you should support in a web API:

- The client constructs a GET request containing the ETag for the currently cached version of the resource referenced in an If-None-Match HTTP header:

```
GET http://adventure-works.com/orders/2 HTTP/1.1
If-None-Match: "2147483648"
...
```

- The GET operation in the web API obtains the current ETag for the requested data (order 2 in the above example), and compares it to the value in the If-None-Match header.
- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should return an HTTP response with an empty message body and a status code of 304 (Not Modified).
- If the current ETag for the requested data does not match the ETag provided by the request, then the data has changed and the web API should return an HTTP response with the new data in the message body and a status code of 200 (OK).
- If the requested data no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).
- The client uses the status code to maintain the cache. If the data has not changed (status code 304) then the object can remain cached and the client application should continue to use this version of the object. If the data has changed (status code 200) then the cached object should be discarded and the new one inserted. If the data is no longer available (status code 404) then the object should be removed from the cache.

  > **NOTE**
  > If the response header contains the Cache-Control header no-store then the object should always be removed from the cache regardless of the HTTP status code.

  The code below shows the `FindOrderByID` method extended to support the If-None-Match header. Notice that if the If-None-Match header is omitted, the specified order is always retrieved:

```
public class OrdersController : ApiController
{
     ...
  [Route("api/orders/{id:int:min(0)}")]
  [HttpGet]
  public IHttpActionResult FindOrderById(int id)
  {
      try
      {
          // Find the matching order
          Order order = ...;

          // If there is no such order then return NotFound
          if (order == null)
          {
              return NotFound();
          }

          // Generate the ETag for the order
          var hashedOrder = order.GetHashCode();
          string hashedOrderEtag = String.Format("\"{0}\"", hashedOrder);

          // Create the Cache-Control and ETag headers for the response
          IHttpActionResult response = null;
          var cacheControlHeader = new CacheControlHeaderValue();
          cacheControlHeader.Public = true;
          cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
          var eTag = new EntityTagHeaderValue(hashedOrderEtag);

          // Retrieve the If-None-Match header from the request (if it exists)
          var nonMatchEtags = Request.Headers.IfNoneMatch;

          // If there is an ETag in the If-None-Match header and
          // this ETag matches that of the order just retrieved,
          // then create a Not Modified response message
          if (nonMatchEtags.Count > 0 &&
              String.Compare(nonMatchEtags.First().Tag, hashedOrderEtag) == 0)
          {
              response = new EmptyResultWithCaching()
              {
                  StatusCode = HttpStatusCode.NotModified,
                  CacheControlHeader = cacheControlHeader,
                  ETag = eTag
              };
          }
          // Otherwise create a response message that contains the order details
          else
          {
              response = new OkResultWithCaching<Order>(order, this)
              {
                  CacheControlHeader = cacheControlHeader,
                  ETag = eTag
              };
          }

          return response;
      }
      catch
      {
          return InternalServerError();
      }
  }
...
}
```

This example incorporates an additional custom `IHttpActionResult` class named `EmptyResultWithCaching`. This class simply acts as a wrapper around an `HttpResponseMessage` object

that does not contain a response body:

```
public class EmptyResultWithCaching : IHttpActionResult
{
  public CacheControlHeaderValue CacheControlHeader { get; set; }
  public EntityTagHeaderValue ETag { get; set; }
  public HttpStatusCode StatusCode { get; set; }
  public Uri Location { get; set; }

  public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
  {
      HttpResponseMessage response = new HttpResponseMessage(StatusCode);
      response.Headers.CacheControl = this.CacheControlHeader;
      response.Headers.ETag = this.ETag;
      response.Headers.Location = this.Location;
      return response;
  }
}
```

> **TIP**
> In this example, the ETag for the data is generated by hashing the data retrieved from the underlying data source. If the ETag can be computed in some other way, then the process can be optimized further and the data only needs to be fetched from the data source if it has changed. This approach is especially useful if the data is large or accessing the data source can result in significant latency (for example, if the data source is a remote database).

- Use ETags to Support Optimistic Concurrency.

  To enable updates over previously cached data, the HTTP protocol supports an optimistic concurrency strategy. If, after fetching and caching a resource, the client application subsequently sends a PUT or DELETE request to change or remove the resource, it should include in If-Match header that references the ETag. The web API can then use this information to determine whether the resource has already been changed by another user since it was retrieved and send an appropriate response back to the client application as follows:

  - The client constructs a PUT request containing the new details for the resource and the ETag for the currently cached version of the resource referenced in an If-Match HTTP header. The following example shows a PUT request that updates an order:

    ```
    PUT http://adventure-works.com/orders/1 HTTP/1.1
    If-Match: "2282343857"
    Content-Type: application/x-www-form-urlencoded
    ...
    Date: Fri, 12 Sep 2014 09:18:37 GMT
    Content-Length: ...
    productID=3&quantity=5&orderValue=250
    ```

  - The PUT operation in the web API obtains the current ETag for the requested data (order 1 in the above example), and compares it to the value in the If-Match header.
  - If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should perform the update, returning a message with HTTP status code 204 (No Content) if it is successful. The response can include Cache-Control and ETag headers for the updated version of the resource. The response should always include the Location header that references the URI of the newly updated resource.
  - If the current ETag for the requested data does not match the ETag provided by the request, then the data has been changed by another user since it was fetched and the web API should return an HTTP response

with an empty message body and a status code of 412 (Precondition Failed).

- If the resource to be updated no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).
- The client uses the status code and response headers to maintain the cache. If the data has been updated (status code 204) then the object can remain cached (as long as the Cache-Control header does not specify no-store) but the ETag should be updated. If the data was changed by another user changed (status code 412) or not found (status code 404) then the cached object should be discarded.

The next code example shows an implementation of the PUT operation for the Orders controller:

```
public class OrdersController : ApiController
{
     ...
  [HttpPut]
  [Route("api/orders/{id:int}")]
          public IHttpActionResult UpdateExistingOrder(int id, DTOOrder order)
  {
      try
      {
          var baseUri = Constants.GetUriFromConfig();
          var orderToUpdate = this.ordersRepository.GetOrder(id);
          if (orderToUpdate == null)
          {
              return NotFound();
          }

          var hashedOrder = orderToUpdate.GetHashCode();
          string hashedOrderEtag = String.Format("\"{0}\"", hashedOrder);

          // Retrieve the If-Match header from the request (if it exists)
          var matchEtags = Request.Headers.IfMatch;

          // If there is an Etag in the If-Match header and
          // this etag matches that of the order just retrieved,
          // or if there is no etag, then update the Order
          if (((matchEtags.Count > 0 &&
               String.Compare(matchEtags.First().Tag, hashedOrderEtag) == 0)) ||
               matchEtags.Count == 0)
          {
              // Modify the order
              orderToUpdate.OrderValue = order.OrderValue;
              orderToUpdate.ProductID = order.ProductID;
              orderToUpdate.Quantity = order.Quantity;

              // Save the order back to the data store
              // ...

              // Create the No Content response with Cache-Control, ETag, and Location headers
              var cacheControlHeader = new CacheControlHeaderValue();
              cacheControlHeader.Private = true;
              cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);

              hashedOrder = order.GetHashCode();
              hashedOrderEtag = String.Format("\"{0}\"", hashedOrder);
              var eTag = new EntityTagHeaderValue(hashedOrderEtag);

              var location = new Uri(string.Format("{0}/{1}/{2}", baseUri, Constants.ORDERS, id));
              var response = new EmptyResultWithCaching()
              {
                  StatusCode = HttpStatusCode.NoContent,
                  CacheControlHeader = cacheControlHeader,
                  ETag = eTag,
                  Location = location
              };

              return response;
```

```
            }

            // Otherwise return a Precondition Failed response
            return StatusCode(HttpStatusCode.PreconditionFailed);
        }
        catch
        {
            return InternalServerError();
        }
    }
    ...
}
```

> **TIP**
> Use of the If-Match header is entirely optional, and if it is omitted the web API will always attempt to update the specified order, possibly blindly overwriting an update made by another user. To avoid problems due to lost updates, always provide an If-Match header.

Considerations for handling large requests and responses

There may be occasions when a client application needs to issue requests that send or receive data that may be several megabytes (or bigger) in size. Waiting while this amount of data is transmitted could cause the client application to become unresponsive. Consider the following points when you need to handle requests that include significant amounts of data:

- **Optimize requests and responses that involve large objects**.

  Some resources may be large objects or include large fields, such as graphics images or other types of binary data. A web API should support streaming to enable optimized uploading and downloading of these resources.

  The HTTP protocol provides the chunked transfer encoding mechanism to stream large data objects back to a client. When the client sends an HTTP GET request for a large object, the web API can send the reply back in piecemeal *chunks* over an HTTP connection. The length of the data in the reply may not be known initially (it might be generated), so the server hosting the web API should send a response message with each chunk that specifies the Transfer-Encoding: Chunked header rather than a Content-Length header. The client application can receive each chunk in turn to build up the complete response. The data transfer completes when the server sends back a final chunk with zero size. You can implement chunking in the ASP.NET Web API by using the `PushStreamContent` class.

  The following example shows an operation that responds to HTTP GET requests for product images:

```
public class ProductImagesController : ApiController
{
    ...
    [HttpGet]
    [Route("productimages/{id:int}")]
    public IHttpActionResult Get(int id)
    {
        try
        {
            var container = ConnectToBlobContainer(Constants.PRODUCTIMAGESCONTAINERNAME);

            if (!BlobExists(container, string.Format("image{0}.jpg", id)))
            {
                return NotFound();
            }
            else
            {
                return new FileDownloadResult()
                {
                    Container = container,
                    ImageId = id
                };
            }
        }
        catch
        {
            return InternalServerError();
        }
    }
    ...
}
```

In this example, `ConnectBlobToContainer` is a helper method that connects to a specified container (name not shown) in Azure Blob storage. `BlobExists` is another helper method that returns a Boolean value that indicates whether a blob with the specified name exists in the blob storage container.

Each product has its own image held in blob storage. The `FileDownloadResult` class is a custom `IHttpActionResult` class that uses a `PushStreamContent` object to read the image data from appropriate blob and transmit it asynchronously as the content of the response message:

```
  public class FileDownloadResult : IHttpActionResult
  {
      public CloudBlobContainer Container { get; set; }
      public int ImageId { get; set; }

      public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
      {
          var response = new HttpResponseMessage();
          response.Content = new PushStreamContent(async (outputStream, _, __) =>
          {
              try
              {
                  CloudBlockBlob blockBlob = Container.GetBlockBlobReference(String.Format("image{0}.jpg",
ImageId));
                  await blockBlob.DownloadToStreamAsync(outputStream);
              }
              finally
              {
                  outputStream.Close();
              }
          });

          response.StatusCode = HttpStatusCode.OK;
          response.Content.Headers.ContentType = new MediaTypeHeaderValue("image/jpeg");
          return response;
      }
  }
```

You can also apply streaming to upload operations if a client needs to POST a new resource that includes a large object. The next example shows the Post method for the `ProductImages` controller. This method enables the client to upload a new product image:

```
  public class ProductImagesController : ApiController
  {
      ...
      [HttpPost]
      [Route("productimages")]
      public async Task<IHttpActionResult> Post()
      {
          try
          {
              if (!Request.Content.Headers.ContentType.MediaType.Equals("image/jpeg"))
              {
                  return StatusCode(HttpStatusCode.UnsupportedMediaType);
              }
              else
              {
                  var id = new Random().Next(); // Use a random int as the key for the new resource.
Should probably check that this key has not already been used
                  var container = ConnectToBlobContainer(Constants.PRODUCTIMAGESCONTAINERNAME);
                  return new FileUploadResult()
                  {
                      Container = container,
                      ImageId = id,
                      Request = Request
                  };
              }
          }
          catch
          {
              return InternalServerError();
          }
      }
      ...
  }
```

This code uses another custom `IHttpActionResult` class called `FileUploadResult`. This class contains the logic for uploading the data asynchronously:

```
  public class FileUploadResult : IHttpActionResult
  {
      public CloudBlobContainer Container { get; set; }
      public int ImageId { get; set; }
      public HttpRequestMessage Request { get; set; }

      public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
      {
          var response = new HttpResponseMessage();
          CloudBlockBlob blockBlob = Container.GetBlockBlobReference(String.Format("image{0}.jpg",
ImageId));
          await blockBlob.UploadFromStreamAsync(await Request.Content.ReadAsStreamAsync());
          var baseUri = string.Format("{0}://{1}:{2}", Request.RequestUri.Scheme, Request.RequestUri.Host,
Request.RequestUri.Port);
          response.Headers.Location = new Uri(string.Format("{0}/productimages/{1}", baseUri, ImageId));
          response.StatusCode = HttpStatusCode.OK;
          return response;
      }
  }
```

> **TIP**
> The volume of data that you can upload to a web service is not constrained by streaming, and a single request could conceivably result in a massive object that consumes considerable resources. If, during the streaming process, the web API determines that the amount of data in a request has exceeded some acceptable bounds, it can abort the operation and return a response message with status code 413 (Request Entity Too Large).

You can minimize the size of large objects transmitted over the network by using HTTP compression. This approach helps to reduce the amount of network traffic and the associated network latency, but at the cost of requiring additional processing at the client and the server hosting the web API. For example, a client application that expects to receive compressed data can include an Accept-Encoding: gzip request header (other data compression algorithms can also be specified). If the server supports compression it should respond with the content held in gzip format in the message body and the Content-Encoding: gzip response header.

> **TIP**
> You can combine encoded compression with streaming; compress the data first before streaming it, and specify the gzip content encoding and chunked transfer encoding in the message headers. Also note that some web servers (such as Internet Information Server) can be configured to automatically compress HTTP responses regardless of whether the web API compresses the data or not.

- **Implement partial responses for clients that do not support asynchronous operations**.

  As an alternative to asynchronous streaming, a client application can explicitly request data for large objects in chunks, known as partial responses. The client application sends an HTTP HEAD request to obtain information about the object. If the web API supports partial responses if should respond to the HEAD request with a response message that contains an Accept-Ranges header and a Content-Length header that indicates the total size of the object, but the body of the message should be empty. The client application can use this information to construct a series of GET requests that specify a range of bytes to receive. The web API should return a response message with HTTP status 206 (Partial Content), a Content-Length header that specifies the actual amount of data included in the body of the response message, and a Content-Range header that indicates which part (such as bytes 4000 to 8000) of the object this data represents.

HTTP HEAD requests and partial responses are described in more detail in the API Design Guidance document.

- **Avoid sending unnecessary Continue status messages in client applications**.

  A client application that is about to send a large amount of data to a server may determine first whether the server is actually willing to accept the request. Prior to sending the data, the client application can submit an HTTP request with an Expect: 100-Continue header, a Content-Length header that indicates the size of the data, but an empty message body. If the server is willing to handle the request, it should respond with a message that specifies the HTTP status 100 (Continue). The client application can then proceed and send the complete request including the data in the message body.

  If you are hosting a service by using IIS, the HTTP.sys driver automatically detects and handles Expect: 100-Continue headers before passing requests to your web application. This means that you are unlikely to see these headers in your application code, and you can assume that IIS has already filtered any messages that it deems to be unfit or too large.

  If you are building client applications by using the .NET Framework, then all POST and PUT messages will first send messages with Expect: 100-Continue headers by default. As with the server-side, the process is handled transparently by the .NET Framework. However, this process results in each POST and PUT request causing 2 round-trips to the server, even for small requests. If your application is not sending requests with large amounts of data, you can disable this feature by using the `ServicePointManager` class to create `ServicePoint` objects in the client application. A `ServicePoint` object handles the connections that the client makes to a server based on the scheme and host fragments of URIs that identify resources on the server. You can then set the `Expect100Continue` property of the `ServicePoint` object to false. All subsequent POST and PUT requests made by the client through a URI that matches the scheme and host fragments of the `ServicePoint` object will be sent without Expect: 100-Continue headers. The following code shows how to configure a `ServicePoint` object that configures all requests sent to URIs with a scheme of `http` and a host of `www.contoso.com`.

  ```
  Uri uri = new Uri("http://www.contoso.com/");
  ServicePoint sp = ServicePointManager.FindServicePoint(uri);
  sp.Expect100Continue = false;
  ```

  You can also set the static `Expect100Continue` property of the `ServicePointManager` class to specify the default value of this property for all subsequently created `ServicePoint` objects. For more information, see the ServicePoint Class page on the Microsoft website.

- **Support pagination for requests that may return large numbers of objects**.

  If a collection contains a large number of resources, issuing a GET request to the corresponding URI could result in significant processing on the server hosting the web API affecting performance, and generate a significant amount of network traffic resulting in increased latency.

  To handle these cases, the web API should support query strings that enable the client application to refine requests or fetch data in more manageable, discrete blocks (or pages). The ASP.NET Web API framework parses query strings and splits them up into a series of parameter/value pairs which are passed to the appropriate method, following the routing rules described earlier. The method should be implemented to accept these parameters using the same names specified in the query string. Additionally, these parameters should be optional (in case the client omits the query string from a request) and have meaningful default values. The code below shows the `GetAllOrders` method in the `Orders` controller. This method retrieves the details of orders. If this method was unconstrained, it could conceivably return a large amount of data. The `limit` and `offset` parameters are intended to reduce the volume of data to a smaller subset, in this case only the first 10 orders by default:

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders")]
    [HttpGet]
    public IEnumerable<Order> GetAllOrders(int limit=10, int offset=0)
    {
        // Find the number of orders specified by the limit parameter
        // starting with the order specified by the offset parameter
        var orders = ...
        return orders;
    }
    ...
}
```

A client application can issue a request to retrieve 30 orders starting at offset 50 by using the URI *http://www.adventure-works.com/api/orders?limit=30&offset=50*.

> **TIP**
> Avoid enabling client applications to specify query strings that result in a URI that is more than 2000 characters long. Many web clients and servers cannot handle URIs that are this long.

Considerations for maintaining responsiveness, scalability, and availability

The same web API might be utilized by many client applications running anywhere in the world. It is important to ensure that the web API is implemented to maintain responsiveness under a heavy load, to be scalable to support a highly varying workload, and to guarantee availability for clients that perform business-critical operations. Consider the following points when determining how to meet these requirements:

- **Provide Asynchronous Support for Long-Running Requests**.

  A request that might take a long time to process should be performed without blocking the client that submitted the request. The web API can perform some initial checking to validate the request, initiate a separate task to perform the work, and then return a response message with HTTP code 202 (Accepted). The task could run asynchronously as part of the web API processing, or it could be offloaded to an Azure WebJob (if the web API is hosted by an Azure Website) or a worker role (if the web API is implemented as an Azure cloud service).

  > **NOTE**
  > For more information about using WebJobs with Azure Website, visit the page Use WebJobs to run background tasks in Microsoft Azure Websites on the Microsoft website.

  The web API should also provide a mechanism to return the results of the processing to the client application. You can achieve this by providing a polling mechanism for client applications to periodically query whether the processing has finished and obtain the result, or enabling the web API to send a notification when the operation has completed.

  You can implement a simple polling mechanism by providing a *polling* URI that acts as a virtual resource using the following approach:

  1. The client application sends the initial request to the web API.
  2. The web API stores information about the request in a table held in table storage or Microsoft Azure Cache, and generates a unique key for this entry, possibly in the form of a GUID.

3. The web API initiates the processing as a separate task. The web API records the state of the task in the table as *Running*.
4. The web API returns a response message with HTTP status code 202 (Accepted), and the GUID of the table entry in the body of the message.
5. When the task has completed, the web API stores the results in the table, and sets the state of the task to *Complete*. Note that if the task fails, the web API could also store information about the failure and set the status to *Failed*.
6. While the task is running, the client can continue performing its own processing. It can periodically send a request to the URI */polling/{guid}* where *{guid}* is the GUID returned in the 202 response message by the web API.
7. The web API at the */polling/{guid}* URI queries the state of the corresponding task in the table and returns a response message with HTTP status code 200 (OK) containing this state (*Running*, *Complete*, or *Failed*). If the task has completed or failed, the response message can also include the results of the processing or any information available about the reason for the failure.

   If you prefer to implement notifications, the options available include:

8. Using an Azure Notification Hub to push asynchronous responses to client applications. The page Azure Notification Hubs Notify Users on the Microsoft website provides further details.
9. Using the Comet model to retain a persistent network connection between the client and the server hosting the web API, and using this connection to push messages from the server back to the client. The MSDN magazine article Building a Simple Comet Application in the Microsoft .NET Framework describes an example solution.
10. Using SignalR to push data in real-time from the web server to the client over a persistent network connection. SignalR is available for ASP.NET web applications as a NuGet package. You can find more information on the ASP.NET SignalR website.

    > **NOTE**
    > Comet and SignalR both utilize persistent network connections between the web server and the client application. This can affect scalability as a large number of clients may require an equally large number of concurrent connections.

- **Ensure that each request is stateless**.

  Each request should be considered atomic. There should be no dependencies between one request made by a client application and any subsequent requests submitted by the same client. This approach assists in scalability; instances of the web service can be deployed on a number of servers. Client requests can be directed at any of these instances and the results should always be the same. It also improves availability for a similar reason; if a web server fails requests can be routed to another instance (by using Azure Traffic Manager) while the server is restarted with no ill effects on client applications.

- **Track clients and implement throttling to reduce the chances of DOS attacks**.

  If a specific client makes a large number of requests within a given period of time it might monopolize the service and affect the performance of other clients. To mitigate this issue, a web API can monitor calls from client applications either by tracking the IP address of all incoming requests or by logging each authenticated access. You can use this information to limit resource access. If a client exceeds a defined limit, the web API can return a response message with status 503 (Service Unavailable) and include a Retry-After header that specifies when the client can send the next request without it being declined. This strategy can help to reduce the chances of a Denial Of Service (DOS) attack from a set of clients stalling the system.

- **Manage persistent HTTP connections carefully**.

The HTTP protocol supports persistent HTTP connections where they are available. The HTTP 1.0 specificiation added the Connection:Keep-Alive header that enables a client application to indicate to the server that it can use the same connection to send subsequent requests rather than opening new ones. The connection closes automatically if the client does not reuse the connection within a period defined by the host. This behavior is the default in HTTP 1.1 as used by Azure services, so there is no need to include Keep-Alive headers in messages.

Keeping a connection open can can help to improve responsiveness by reducing latency and network congestion, but it can be detrimental to scalability by keeping unnecessary connections open for longer than required, limiting the ability of other concurrent clients to connect. It can also affect battery life if the client application is running on a mobile device; if the application only makes occassional requests to the server, maintaining an open connection can cause the battery to drain more quickly. To ensure that a connection is not made persistent with HTTP 1.1, the client can include a Connection:Close header with messages to override the default behavior. Similarly, if a server is handling a very large number of clients it can include a Connection:Close header in response messages which should close the connection and save server resources.

> **NOTE**
> Persistent HTTP connections are a purely optional feature to reduce the network overhead associated with repeatedly establishing a communications channel. Neither the web API nor the client application should depend on a persistent HTTP connection being available. Do not use persistent HTTP connections to implement Comet-style notification systems; instead you should utilize sockets (or websockets if available) at the TCP layer. Finally, note Keep-Alive headers are of limited use if a client application communicates with a server via a proxy; only the connection with the client and the proxy will be persistent.

Considerations for publishing and managing a web API

To make a web API available for client applications, the web API must be deployed to a host environment. This environment is typically a web server, although it may be some other type of host process. You should consider the following points when publishing a web API:

- All requests must be authenticated and authorized, and the appropriate level of access control must be enforced.
- A commercial web API might be subject to various quality guarantees concerning response times. It is important to ensure that host environment is scalable if the load can vary significantly over time.
- If may be necessary to meter requests for monetization purposes.
- It might be necessary to regulate the flow of traffic to the web API, and implement throttling for specific clients that have exhausted their quotas.
- Regulatory requirements might mandate logging and auditing of all requests and responses.
- To ensure availability, it may be necessary to monitor the health of the server hosting the web API and restart it if necessary.

It is useful to be able to decouple these issues from the technical issues concerning the implementation of the web API. For this reason, consider creating a façade, running as a separate process and that routes requests to the web API. The façade can provide the management operations and forward validated requests to the web API. Using a façade can also bring many functional advantages, including:

- Acting as an integration point for multiple web APIs.
- Transforming messages and translating communications protocols for clients built by using varying technologies.
- Caching requests and responses to reduce load on the server hosting the web API.

Considerations for testing a web API

A web API should be tested as thoroughly as any other piece of software. You should consider creating unit tests to validate the functionality of each operation, as you would with any other type of application. For more information, see the page Verifying Code by Using Unit Tests on the Microsoft website.

> **NOTE**
> The sample web API available with this guidance includes a test project that shows how to perform unit testing over selected operations.

The nature of a web API brings its own additional requirements to verify that it operates correctly. You should pay particular attention to the following aspects:

- Test all routes to verify that they invoke the correct operations. Be especially aware of HTTP status code 405 (Method Not Allowed) being returned unexpectedly as this can indicate a mismatch between a route and the HTTP methods (GET, POST, PUT, DELETE) that can be dispatched to that route.

  Send HTTP requests to routes that do not support them, such as submitting a POST request to a specific resource (POST requests should only be sent to resource collections). In these cases, the only valid response *should* be status code 405 (Not Allowed).

- Verify that all routes are protected properly and are subject to the appropriate authentication and authorization checks.

  > **NOTE**
  > Some aspects of security such as user authentication are most likely to be the responsibility of the host environment rather than the web API, but it is still necessary to include security tests as part of the deployment process.

- Test the exception handling performed by each operation and verify that an appropriate and meaningful HTTP response is passed back to the client application.
- Verify that request and response messages are well-formed. For example, if an HTTP POST request contains the data for a new resource in x-www-form-urlencoded format, confirm that the corresponding operation correctly parses the data, creates the resources, and returns a response containing the details of the new resource, including the correct Location header.
- Verify all links and URIs in response messages. For example, an HTTP POST message should return the URI of the newly-created resource. All HATEOAS links should be valid.

  > **IMPORTANT**
  > If you publish the web API through an API Management Service, then these URIs should reflect the URL of the management service and not that of the web server hosting the web API.

- Ensure that each operation returns the correct status codes for different combinations of input. For example:
  - If a query is successful, it should return status code 200 (OK)
  - If a resource is not found, the operation should returs HTTP status code 404 (Not Found).
  - If the client sends a request that successfully deletes a resource, the status code should be 204 (No Content).
  - If the client sends a request that creates a new resource, the status code should be 201 (Created)

Watch out for unexpected response status codes in the 5xx range. These messages are usually reported by the host server to indicate that it was unable to fulfill a valid request.

- Test the different request header combinations that a client application can specify and ensure that the web API returns the expected information in response messages.
- Test query strings. If an operation can take optional parameters (such as pagination requests), test the different combinations and order of parameters.
- Verify that asynchronous operations complete successfully. If the web API supports streaming for requests that return large binary objects (such as video or audio), ensure that client requests are not blocked while the data is streamed. If the web API implements polling for long-running data modification operations, verify that that the operations report their status correctly as they proceed.

You should also create and run performance tests to check that the web API operates satisfactorily under duress. You can build a web performance and load test project by using Visual Studio Ultimate. For more information, see the page Run performance tests on an application before a release on the Microsoft website.

Publishing and managing a web API by using the Azure API Management Service

Azure provides the API Management Service which you can use to publish and manage a web API. Using this facility, you can generate a service that acts a façade for one or more web APIs. The service is itself a scalable web service that you can create and configure by using the Azure Management portal. You can use this service to publish and manage a web API as follows:

1. Deploy the web API to a website, Azure cloud service, or Azure virtual machine.
2. Connect the API management service to the web API. Requests sent to the URL of the management API are mapped to URIs in the web API. The same API management service can route requests to more than one web API. This enables you to aggregate multiple web APIs into a single management service. Similarly, the same web API can be referenced from more than one API management service if you need to restrict or partition the functionality available to different applications.

   > **NOTE**
   > The URIs in HATEOAS links generated as part of the response for HTTP GET requests should reference the URL of the API management service and not the web server hosting the web API.

3. For each web API, specify the HTTP operations that the web API exposes together with any optional parameters that an operation can take as input. You can also configure whether the API management service should cache the response received from the web API to optimize repeated requests for the same data. Record the details of the HTTP responses that each operation can generate. This information is used to generate documentation for developers, so it is important that it is accurate and complete.

   You can either define operations manually using the wizards provided by the Azure Management portal, or you can import them from a file containing the definitions in WADL or Swagger format.

4. Configure the security settings for communications between the API management service and the web server hosting the web API. The API management service currently supports Basic authentication and mutual authentication using certificates, and OAuth 2.0 user authorization.
5. Create a product. A product is the unit of publication; you add the web APIs that you previously connected to the management service to the product. When the product is published, the web APIs become available to developers.

> **NOTE**
>
> Prior to publishing a product, you can also define user-groups that can access the product and add users to these groups. This gives you control over the developers and applications that can use the web API. If a web API is subject to approval, prior to being able to access it a developer must send a request to the product administrator. The administrator can grant or deny access to the developer. Existing developers can also be blocked if circumstances change.

6. Configure policies for each web API. Policies govern aspects such as whether cross-domain calls should be allowed, how to authenticate clients, whether to convert between XML and JSON data formats transparently, whether to restrict calls from a given IP range, usage quotas, and whether to limit the call rate. Policies can be applied globally across the entire product, for a single web API in a product, or for individual operations in a web API.

You can find full details describing how to perform these tasks on the API Management page on the Microsoft website. The Azure API Management Service also provides its own REST interface, enabling you to build a custom interface for simplifying the process of configuring a web API. For more information, visit the Azure API Management REST API Reference page on the Microsoft website.

> **TIP**
>
> Azure provides the Azure Traffic Manager which enables you to implement failover and load-balancing, and reduce latency across multiple instances of a web site hosted in different geographic locations. You can use Azure Traffic Manager in conjunction with the API Management Service; the API Management Service can route requests to instances of a web site through Azure Traffic Manager. For more information, visit the Traffic Manager routing Methods page on the Microsoft website.
>
> In this structure, if you are using custom DNS names for your web sites, you should configure the appropriate CNAME record for each web site to point to the DNS name of the Azure Traffic Manager web site.

Supporting developers building client applications

Developers constructing client applications typically require information on how to access the web API, and documentation concerning the parameters, data types, return types, and return codes that describe the different requests and responses between the web service and the client application.

Documenting the REST operations for a web API

The Azure API Management Service includes a developer portal that describes the REST operations exposed by a web API. When a product has been published it appears on this portal. Developers can use this portal to sign up for access; the administrator can then approve or deny the request. If the developer is approved, they are assigned a subscription key that is used to authenticate calls from the client applications that they develop. This key must be provided with each web API call otherwise it will be rejected.

This portal also provides:

- Documentation for the product, listing the operations that it exposes, the parameters required, and the different responses that can be returned. Note that this information is generated from the details provided in step 3 in the list in the Publishing a web API by using the Microsoft Azure API Management Service section.
- Code snippets that show how to invoke operations from several languages, including JavaScript, C#, Java, Ruby, Python, and PHP.
- A developers' console that enables a developer to send an HTTP request to test each operation in the product and view the results.
- A page where the developer can report any issues or problems found.

The Azure Management portal enables you to customize the developer portal to change the styling and layout to match the branding of your organization.

Implementing a client SDK

Building a client application that invokes REST requests to access a web API requires writing a significant amount of code to construct each request and format it appropriately, send the request to the server hosting the web service, and parse the response to work out whether the request succeeded or failed and extract any data returned. To insulate the client application from these concerns, you can provide an SDK that wraps the REST interface and abstracts these low-level details inside a more functional set of methods. A client application uses these methods, which transparently convert calls into REST requests and then convert the responses back into method return values. This is a common technique that is implemented by many services, including the Azure SDK.

Creating a client-side SDK is a considerable undertaking as it has to be implemented consistently and tested carefully. However, much of this process can be made mechanical, and many vendors supply tools that can automate many of these tasks.

Monitoring a web API

Depending on how you have published and deployed your web API you can monitor the web API directly, or you can gather usage and health information by analyzing the traffic that passes through the API Management service.

Monitoring a web API directly

If you have implemented your web API by using the ASP.NET Web API template (either as a Web API project or as a Web role in an Azure cloud service) and Visual Studio 2013, you can gather availability, performance, and usage data by using ASP.NET Application Insights. Application Insights is a package that transparently tracks and records information about requests and responses when the web API is deployed to the cloud; once the package is installed and configured, you don't need to amend any code in your web API to use it. When you deploy the web API to an Azure web site, all traffic is examined and the following statistics are gathered:

- Server response time.
- Number of server requests and the details of each request.
- The top slowest requests in terms of average response time.
- The details of any failed requests.
- The number of sessions initiated by different browsers and user agents.
- The most frequently viewed pages (primarily useful for web applications rather than web APIs).
- The different user roles accessing the web API.

You can view this data in real time from the Azure Management portal. You can also create webtests that monitor the health of the web API. A webtest sends a periodic request to a specified URI in the web API and captures the response. You can specify the definition of a successful response (such as HTTP status code 200), and if the request does not return this response you can arrange for an alert to be sent to an administrator. If necessary, the administrator can restart the server hosting the web API if it has failed.

The Application Insights - Get started with ASP.NET page on the Microsoft website provides more information.

Monitoring a web API through the API Management Service

If you have published your web API by using the API Management service, the API Management page on the Azure Management portal contains a dashboard that enables you to view the overall performance of the service. The Analytics page enables you to drill down into the details of how the product is being used. This page contains the following tabs:

- **Usage**. This tab provides information about the number of API calls made and the bandwidth used to handle these calls over time. You can filter usage details by product, API, and operation.
- **Health**. This tab enables you view the outcome of API requests (the HTTP status codes returned), the

effectiveness of the caching policy, the API response time, and the service response time. Again, you can filter health data by product, API, and operation.

- **Activity**. This tab provides a text summary of the numbers of successful calls, failed called, blocked calls, average response time, and response times for each product, web API, and operation. This page also lists the number of calls made by each developer.
- **At a glance**. This tab displays a summary of the performance data, including the developers responsible for making the most API calls, and the products, web APIs, and operations that received these calls.

You can use this information to determine whether a particular web API or operation is causing a bottleneck, and if necessary scale the host environment and add more servers. You can also ascertain whether one or more applications are using a disproportionate volume of resources and apply the appropriate policies to set quotas and limit call rates.

> **NOTE**
> You can change the details for a published product, and the changes are applied immediately. For example, you can add or remove an operation from a web API without requiring that you republish the product that contains the web API.

Related patterns

- The façade pattern describes how to provide an interface to a web API.

More information

- The page Learn About ASP.NET Web API on the Microsoft website provides a detailed introduction to building RESTful web services by using the Web API.
- The page Routing in ASP.NET Web API on the Microsoft website describes how convention-based routing works in the ASP.NET Web API framework.
- For more information on attribute-based routing, see the page Attribute Routing in Web API 2 on the Microsoft website.
- The Basic Tutorial page on the OData website provides an introduction to the features of the OData protocol.
- The ASP.NET Web API OData page on the Microsoft website contains examples and further information on implementing an OData web API by using ASP.NET.
- The page Introducing Batch Support in Web API and Web API OData on the Microsoft website describes how to implement batch operations in a web API by using OData.
- The article Idempotency Patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.
- The Status Code Definitions page on the W3C website contains a full list of HTTP status codes and their descriptions.
- For detailed information on handling HTTP exceptions with the ASP.NET Web API, visit the Exception Handling in ASP.NET Web API page on the Microsoft website.
- The article Web API Global Error Handling on the Microsoft website describes how to implement a global error handling and logging strategy for a web API.
- The page Run background tasks with WebJobs on the Microsoft website provides information and examples on using WebJobs to perform background operations on an Azure Website.
- The page Azure Notification Hubs Notify Users on the Microsoft website shows how you can use an Azure Notification Hub to push asynchronous responses to client applications.
- The API Management page on the Microsoft website describes how to publish a product that provides controlled and secure access to a web API.
- The Azure API Management REST API Reference page on the Microsoft website describes how to use the API Management REST API to build custom management applications.

- The Traffic Manager Routing Methods page on the Microsoft website summarizes how Azure Traffic Manager can be used to load-balance requests across multiple instances of a website hosting a web API.
- The Application Insights - Get started with ASP.NET page on the Microsoft website provides detailed information on installing and configuring Application Insights in an ASP.NET Web API project.
- The page Verifying Code by Using Unit Tests on the Microsoft website provides detailed information on creating and managing unit tests by using Visual Studio.
- The page Run performance tests on an application before a release on the Microsoft website describes how to use Visual Studio Ultimate to create a web performance and load test project.