

Microsoft Official Course



AZ-203T06

Connect to and consume
Azure, and third-party,
services

AZ-203T06

**Connect to and consume Azure,
and third-party, services**

MCT USE ONLY. STUDENT USE PROHIBITED



Contents

■	Module 0 Welcome to the course	1
	Start Here	1
■	Module 1 Develop an App Service Logic App	5
	Azure Logic Apps overview	5
	Create Logic Apps by using Visual Studio	9
	Create custom connectors for Logic Apps	21
	Create custom templates for Logic Apps	27
	Review questions	32
■	Module 2 Integrate Azure Search within solutions	35
	Create and query an Azure Search index	35
	Full text search in Azure Search	56
	Review questions	66
■	Module 3 API Management	69
	Introduction to the API Management service	69
	Securing your APIs	76
	Defining API policies	83
	Review questions	92
■	Module 4 Develop event-based solutions	95
	Implement solutions that use Azure Event Grid	95
	Implement solutions that use Azure Event Hubs	111
	Implement solutions that use Azure Notification Hubs	126
	Review questions	142
■	Module 5 Develop message-based solutions	143
	Implement solutions that use Azure Service Bus	143
	Implement solutions that use Azure Queue Storage queues	164
	Review questions	171



Module 0 Welcome to the course

Start Here

Welcome

Welcome to the **Connect to and consume Azure, and third-party, services** course. This course is part of a series of courses to help you prepare for the **AZ-203: Developing Solutions for Microsoft Azure**¹ certification exam.

Candidates for this exam are Azure Developers who design and build cloud solutions such as applications and services. They participate in all phases of development, from solution design, to development and deployment, to testing and maintenance. They partner with cloud solution architects, cloud DBAs, cloud administrators, and clients to implement the solution.

Candidates should be proficient in developing apps and services by using Azure tools and technologies, including storage, security, compute, and communications.

Candidates must have at least one year of experience developing scalable solutions through all phases of software development and be skilled in at least one cloud-supported programming language.

Exam study areas

AZ-203 includes six study areas, as shown in the table. The percentages indicate the relative weight of each area on the exam. The higher the percentage, the more questions you are likely to see in that area.

AZ-203 Study Areas	Weight
Develop Azure Infrastructure as a Service compute solutions	10-15%
Develop Azure Platform as a Service compute solutions	20-25%
Develop for Azure storage	15-20%
Implement Azure security	10-15%

¹ <https://www.microsoft.com/en-us/learning/exam-az-203.aspx>

AZ-203 Study Areas	Weight
Monitor, troubleshoot, and optimize Azure solutions	15-20%
Connect to and consume Azure, and third-party, services	20-25%

✓ This course will focus on preparing you for the **Connect to and consume Azure, and third-party, services** area of the AZ-203 certification exam.

Course description

This course is all about communication between apps and services. Students will learn how to create and manage their own APIs by using API Management, and how to use the different event- and message-based services in Azure within their development solutions.

Throughout the course students learn how to create and integrate these resources by using the Azure Portal, Azure CLI, REST, and application code.

Level: Intermediate

Audience:

- Students in this course are interested in Azure development or in passing the Microsoft Azure Developer Associate certification exam.
- Students should have 1-2 years experience as a developer. This course assumes students know how to code and have a fundamental knowledge of Azure.
- It is recommended that students have some experience with PowerShell or Azure CLI, working in the Azure portal, and with at least one Azure-supported programming language. Most of the examples in this course are presented in C# .NET.

Course Syllabus

Module 1: Develop an App Service Logic App

- Azure Logic Apps overview
- Create Logic Apps by using Visual Studio
- Create custom connectors for Logic Apps
- Create custom templates for Logic Apps

Module 2: Integrate Azure Search within solutions

- Create and query an Azure Search index
- Full text search in Azure Search

Module 3: API Management

- Introduction to the API Management service
- Securing your APIs
- Defining API policies

Module 4: Develop event-based solutions

- Implement solutions that use Azure Event Grid

- Implement solutions that use Azure Event Hubs
- Implement solutions that use Azure Notification Hubs

Module 5: Develop message-based solutions

- Implement solutions that use Azure Service Bus
- Implement solutions that use Azure Queue Storage queues



Module 1 Develop an App Service Logic App

Azure Logic Apps overview

Azure Logic Apps explained

Logic Apps helps you build solutions that integrate apps, data, systems, and services across enterprises or organizations by automating tasks and business processes as workflows. Logic Apps is cloud service in Azure that simplifies how you design and create scalable solutions for app integration, data integration, system integration, enterprise application integration (EAI), and business-to-business (B2B) communication, whether in the cloud, on premises, or both.

For example, here are just a few workloads that you can automate with logic apps:

- Process and route orders across on-premises systems and cloud services.
- Move uploaded files from an SFTP or FTP server to Azure Storage.
- Send email notifications with Office 365 when events happen in various systems, apps, and services.
- Monitor tweets for a specific subject, analyze the sentiment, and create alerts or tasks for items that need review.

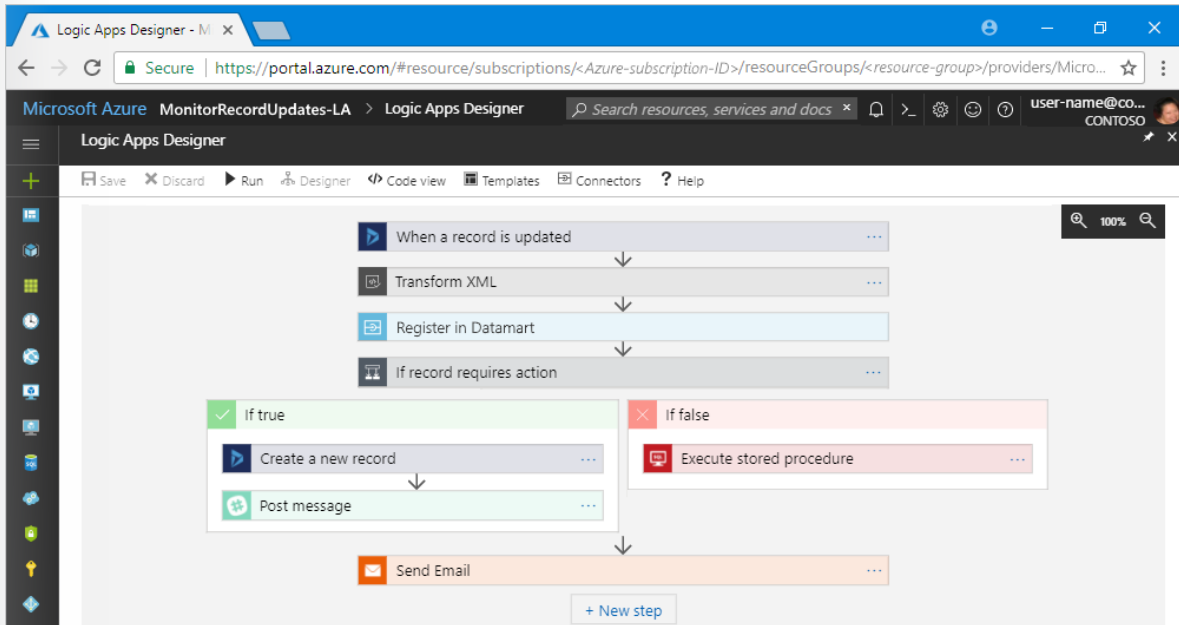
To build integration solutions with logic apps, choose from a growing gallery that has **200+ connectors**¹, including other Azure services such as Service Bus, Functions, and Storage; SQL, Office 365, Dynamics, BizTalk, Salesforce, SAP, Oracle DB, file shares, and many more. These connectors provide triggers, actions, or both for creating logic apps that securely access and process data in real time

How does Logic Apps work?

Every logic app workflow starts with a trigger, which fires when a specific event happens, or when new available data meets specific criteria. Many triggers include basic scheduling capabilities so that you can specify how regularly your workloads run. For more custom scheduling scenarios, start your workflows with the Schedule trigger. Learn more about how to build schedule-based workflows.

¹ <https://docs.microsoft.com/en-us/azure/connectors/apis-list>

Each time that the trigger fires, the Logic Apps engine creates a logic app instance that runs the workflow's actions. These actions can also include data conversions and flow controls, such as conditional statements, switch statements, loops, and branching. For example, this logic app starts with a Dynamics 365 trigger with the built-in criteria "When a record is updated". If the trigger detects an event that matches this criteria, the trigger fires and runs the workflow's actions. Here, these actions include XML transformation, data updates, decision branching, and email notifications.



You can build your logic apps visually with the Logic Apps Designer, available in the Azure portal through your browser and in Visual Studio. For more custom logic apps, you can create or edit logic app definitions in JavaScript Object Notation (JSON) by working in "code view" mode. You can also use Azure PowerShell commands and Azure Resource Manager templates for select tasks. Logic apps deploy and run in the cloud on Azure. For a more detailed introduction, watch this video: [Use Azure Enterprise Integration Services to run cloud apps at scale](#)

Connectors for Azure Logic Apps

Connectors play an integral part when you create automated workflows with Azure Logic Apps. By using connectors in your logic apps, you expand the capabilities for your on-premises and cloud apps to perform tasks with the data that you create and already have. Connectors are available as either built-ins or managed connectors.

- **Built-ins:** These built-in actions and triggers help you create logic apps that run on custom schedules, communicate with other endpoints, receive and respond to requests, and call Azure functions, Azure API Apps (Web Apps), your own APIs managed and published with Azure API Management, and nested logic apps that can receive requests. You can also use built-in actions that help you organize and control your logic app's workflow, and also work with data.
- **Managed connectors:** These connectors provide triggers and actions for accessing other services and systems. Some connectors require that you first create connections that are managed by Azure Logic Apps. Managed connectors are organized into these groups:

Managed API connectors	Create logic apps that use services such as Azure Blob Storage, Office 365, Dynamics, Power BI, OneDrive, Salesforce, SharePoint Online, and many more.
On-premises connectors	After you install and set up the on-premises data gateway, these connectors help your logic apps access on-premises systems such as SQL Server, SharePoint Server, Oracle DB, file shares, and others.
Integration account connectors	Available when you create and pay for an integration account, these connectors transform and validate XML, encode and decode flat files, and process business-to-business (B2B) messages with AS2, EDIFACT, and X12 protocols.
Enterprise connectors	Provide access to enterprise systems such as SAP and IBM MQ for an additional cost.

Components of a Connector

Each connector offers a set of operations classified as 'Actions' and 'Triggers'. Once you connect to the underlying service, these operations can be easily leveraged within your apps and workflows.

Actions

Actions are changes directed by a user. For example, you would use an action to look up, write, update, or delete data in a SQL database. All actions directly map to operations defined in the Swagger.

Triggers

Several connectors provide triggers that can notify your app when specific events occur. For example, the FTP connector has the `OnUpdatedFile` trigger. You can build either a Logic App or Flow that listens to this trigger and performs an action whenever the trigger fires.

There are two types of trigger.

- **Polling Triggers:** These triggers call your service at a specified frequency to check for new data. When new data is available, it causes a new run of your workflow instance with the data as input.
- **Push Triggers:** These triggers listen for data on an endpoint, that is, they wait for an event to occur. The occurrence of this event causes a new run of your workflow instance.

B2B scenarios and the Enterprise Integration Pack

For business-to-business (B2B) workflows and seamless communication with Azure Logic Apps, you can enable enterprise integration scenarios with Microsoft's cloud-based solution, the Enterprise Integration Pack. Organizations can exchange messages electronically, even if they use different protocols and formats. The pack transforms different formats into a format that organizations' systems can interpret

and process. Organizations can exchange messages through industry-standard protocols, including AS2, X12, and EDIFACT. You can also secure messages with both encryption and digital signatures.

If you are familiar with BizTalk Server or Microsoft Azure BizTalk Services, the Enterprise Integration features are easy to use because most concepts are similar. One major difference is that Enterprise Integration uses integration accounts to simplify the storage and management of artifacts used in B2B communications.

Architecturally, the Enterprise Integration Pack is based on “integration accounts”. These accounts are cloud-based containers that store all your artifacts, like schemas, partners, certificates, maps, and agreements. You can use these artifacts to design, deploy, and maintain your B2B apps and also to build B2B workflows for logic apps. But before you can use these artifacts, you must first link your integration account to your logic app. After that, your logic app can access your integration account's artifacts.

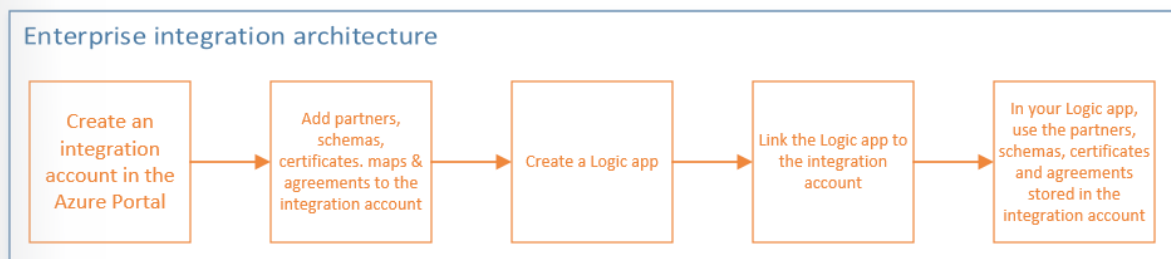
Why should you use enterprise integration?

- With enterprise integration, you can store all your artifacts in one place – your integration account.
- You can build B2B workflows and integrate with third-party software-as-a-service (SaaS) apps, on-premises apps, and custom apps by using the Azure Logic Apps engine and all its connectors.
- You can create custom code for your logic apps with Azure functions.

How to get started with enterprise integration

You can build and manage B2B apps with the Enterprise Integration Pack through the Logic App Designer in the **Azure portal**. You can also manage your logic apps with PowerShell.

Here are the high-level steps you must take before you can create apps in the Azure portal:

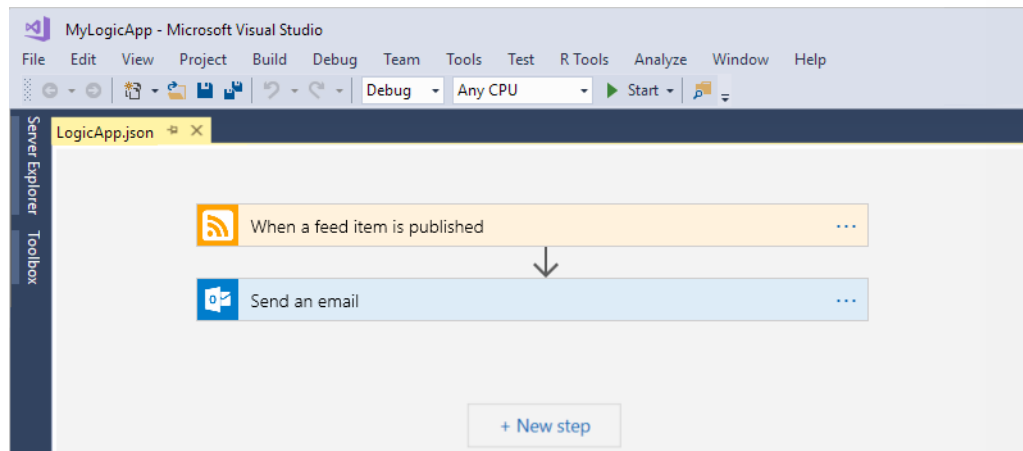


Create Logic Apps by using Visual Studio

Create Logic Apps by using Visual Studio

With Azure Logic Apps and Visual Studio, you can create workflows for automating tasks and processes that integrate apps, data, systems, and services across enterprises and organizations. This lesson shows how you can design and build these workflows by creating logic apps in Visual Studio and deploying those apps to Azure in the cloud. And although you can perform these tasks in the Azure portal, Visual Studio lets you add logic apps to source control, publish different versions, and create Azure Resource Manager templates for different deployment environments.

We'll be creating a logic app that monitors a website's RSS feed and sends email for each new item posted on the site. When you're done, your logic app looks like this high-level workflow:



Prerequisites

Before you start, make sure that you have these items:

- If you don't have an Azure subscription, **sign up for a free Azure account²**.
- Download and install these tools, if you don't have them already:
 - **Visual Studio 2017 or Visual Studio 2015 - Community edition or greater³**. This section uses Visual Studio Community 2017, which is free.
 - **Microsoft Azure SDK for .NET (2.9.1 or later)⁴** and **Azure PowerShell⁵**.
 - **Azure Logic Apps Tools for Visual Studio 2017⁶** or the **Visual Studio 2015⁷** version
- An email account that's supported by Logic Apps, such as Office 365 Outlook, Outlook.com, or Gmail. For other providers, **review the connectors list here⁸**. This logic app uses Office 365 Outlook. If you use a different provider, the overall steps are the same, but your UI might slightly differ.
- Access to the web while using the embedded Logic App Designer

² <https://azure.microsoft.com/free/>

³ <https://www.visualstudio.com/downloads>

⁴ <https://azure.microsoft.com/downloads/>

⁵ <https://github.com/Azure/azure-powershell#installation>

⁶ <https://marketplace.visualstudio.com/items?itemName=VinaySinghMSFT.AzureLogicAppsToolsforVisualStudio-18551>

⁷ <https://marketplace.visualstudio.com/items?itemName=VinaySinghMSFT.AzureLogicAppsToolsforVisualStudio>

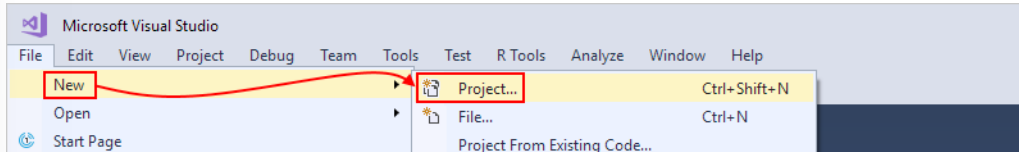
⁸ <https://docs.microsoft.com/connectors/>

- The designer requires an internet connection to create resources in Azure and to read the properties and data from connectors in your logic app. For example, if you use the Dynamics CRM Online connector, the designer checks your CRM instance for available default and custom properties.

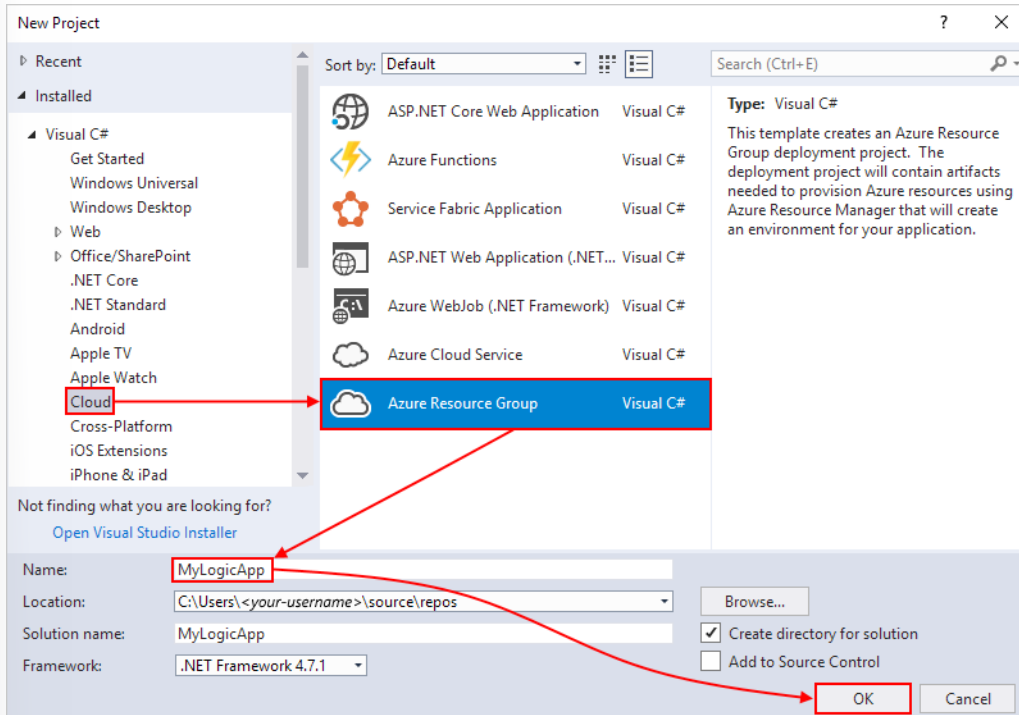
Create an Azure resource group project

To get started, create an Azure Resource Group project.

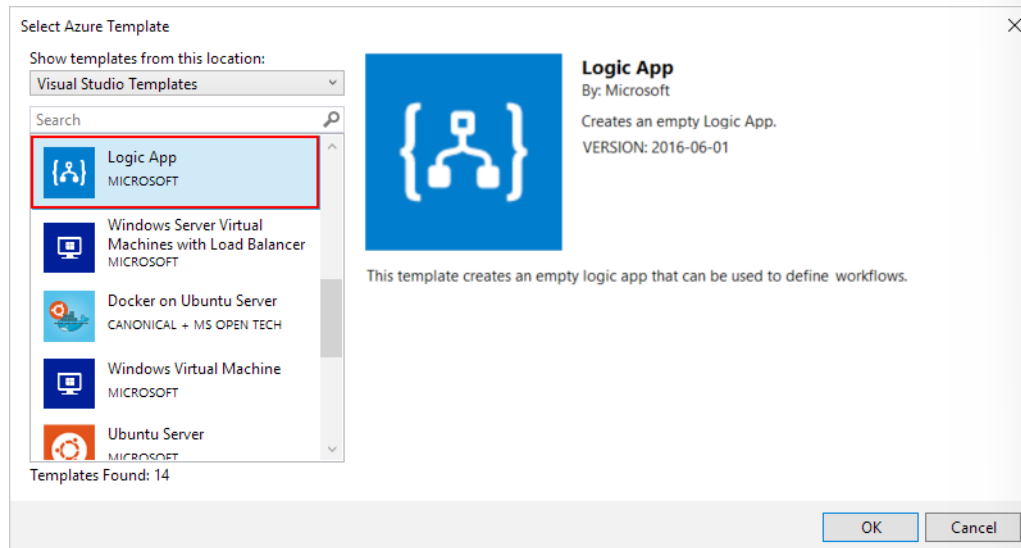
1. Start Visual Studio and sign in with your Azure account.
2. On the **File** menu, select **New > Project**.



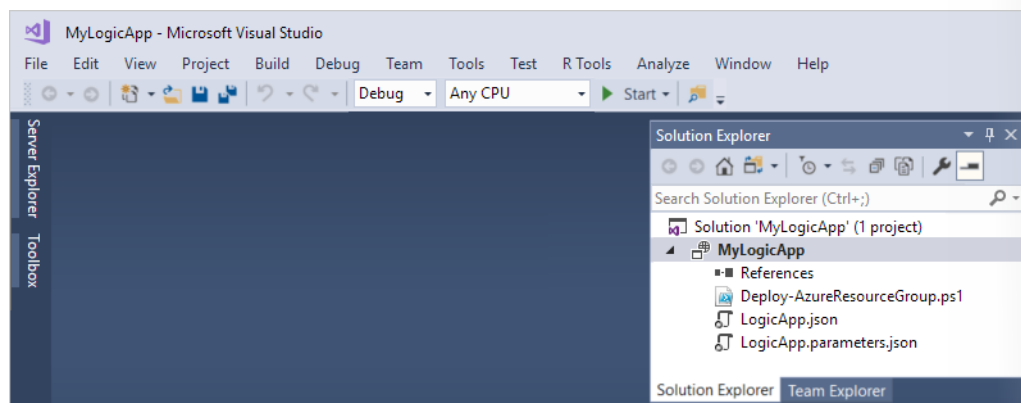
- 3.
4. Under **Installed**, select **Visual C#**. Select **Cloud > Azure Resource Group**. Name your project, for example:



- 5.
6. Select the **Logic App** template.



- 7.
8. After Visual Studio creates your project, Solution Explorer opens and shows your solution.

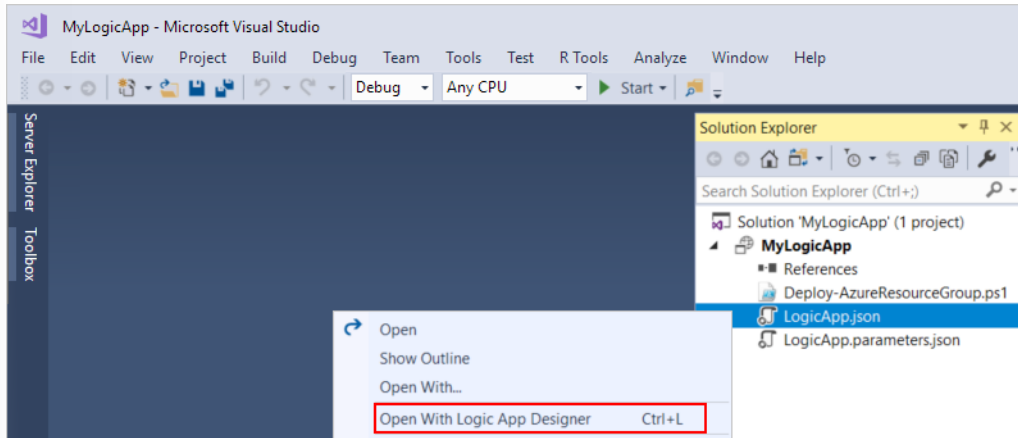


- 9.
10. In your solution, the **LogicApp.json** file not only stores the definition for your logic app but is also an Azure Resource Manager template that you can set up for deployment.

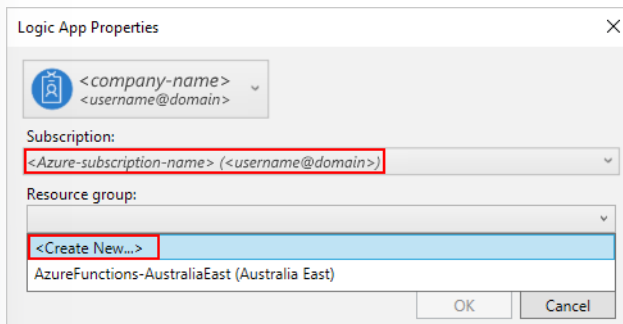
Create a blank Logic App

After you create your Azure Resource Group project, create and build your logic app starting from the Blank Logic App template.

1. In Solution Explorer, open the shortcut menu for the **LogicApp.json** file. Select **Open With Logic App Designer**.



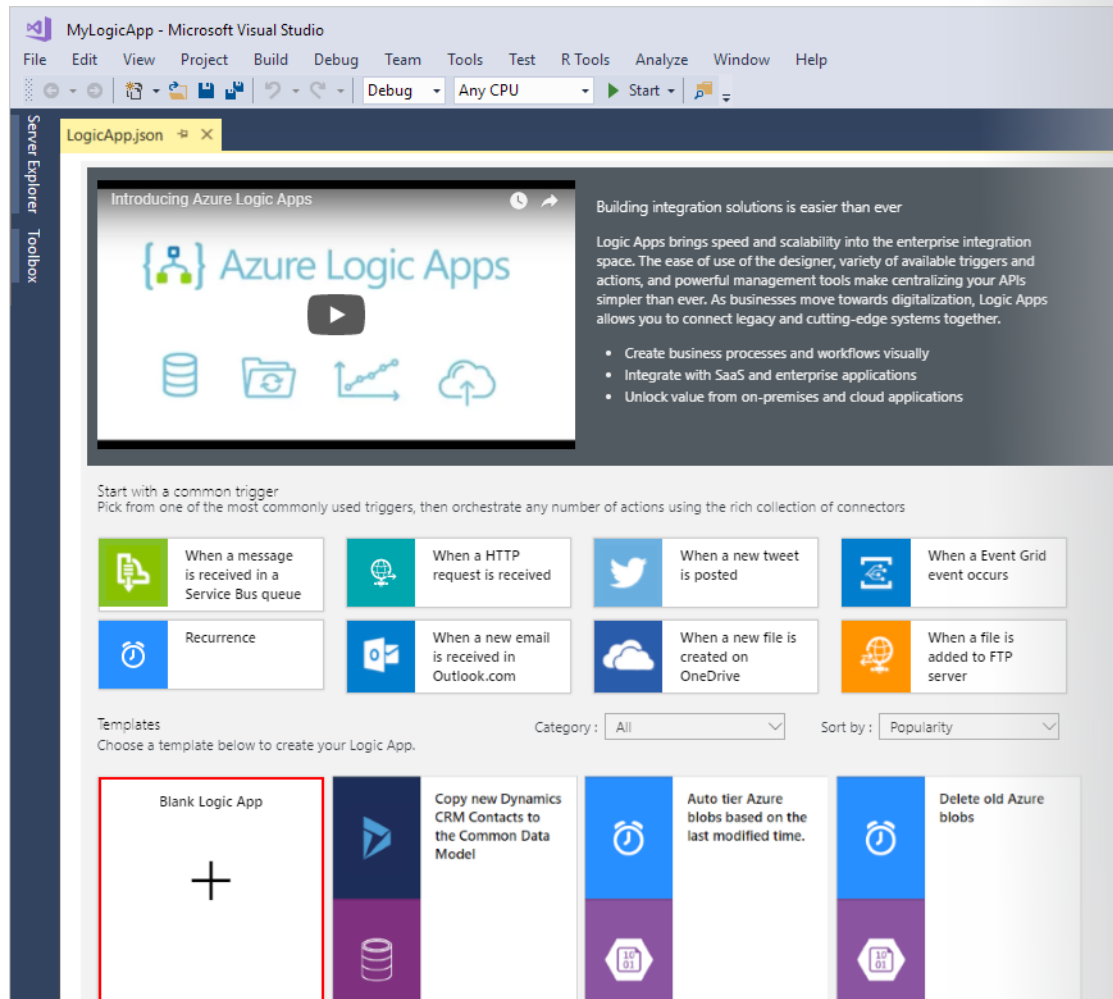
- 2.
3. For **Subscription**, select the Azure subscription that you to use. For **Resource Group**, select **Create New...**, which creates a new Azure resource group.



- 4.
5. Visual Studio needs your Azure subscription and a resource group for creating and deploying resources associated with your logic app and connections.

Setting	Sample Value	Description
User profile list	Contoso (jamalhartnett@contoso.com)	By default, the account that you used to sign in
Subscription	Pay-As-You-Go (jamalhartnett@contoso.com)	The name for your Azure subscription and associated account
Resource Group	MyLogicApp-RG (West US)	The Azure resource group and location for storing and deploying resources for your logic app
Location	MyLogicApp-RG (West US)	A different location if you don't want to use the resource group location

6. The Logic Apps Designer opens and shows a page with an introduction video and commonly used triggers. Scroll past the video and triggers. Under **Templates**, select **Blank Logic App**.



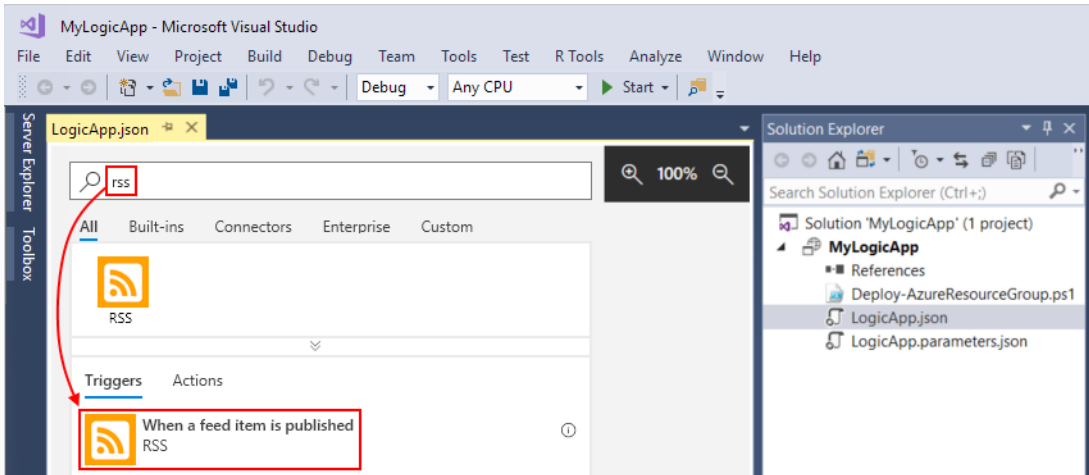
7.

Build the Logic App workflow

Creating the trigger

Next, add a trigger that fires when a new RSS feed item appears. Every logic app must start with a trigger, which fires when specific criteria is met. Each time the trigger fires, the Logic Apps engine creates a logic app instance that runs your workflow.

1. In Logic App Designer, enter "rss" in the search box. Select this trigger: **When a feed item is published**



- 2.
3. Provide this information for your trigger as shown and described:

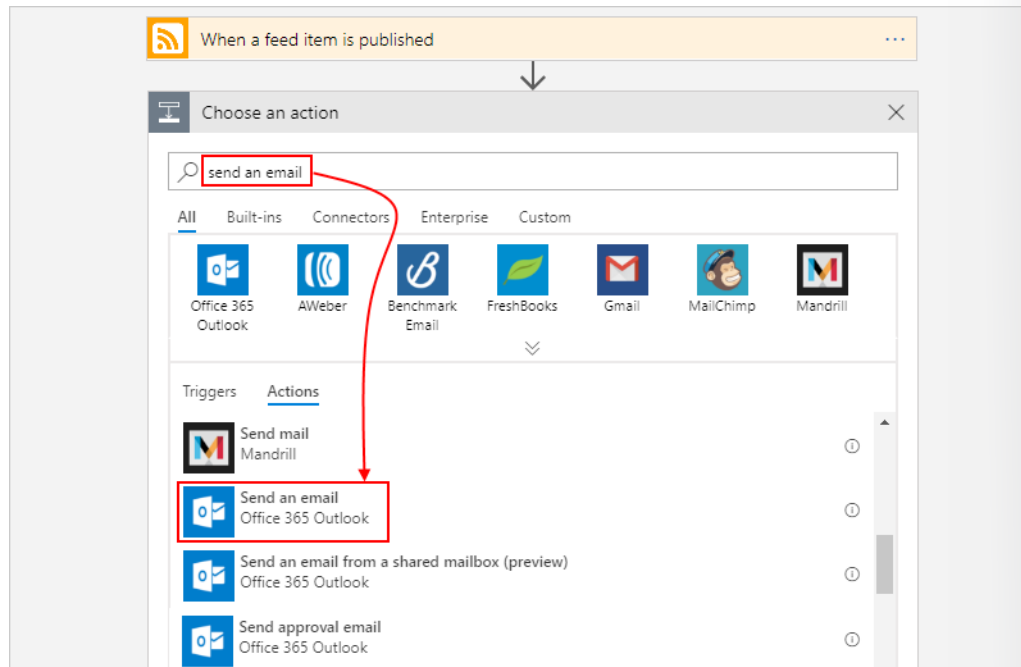
- 4.

Property	Value	Description
The RSS feed URL	http://feeds.reuters.com/reuters/topNews	The link for the RSS feed that you want to monitor
Interval	1	The number of intervals to wait between checks
Frequency	Minute	The unit of time for each interval between checks

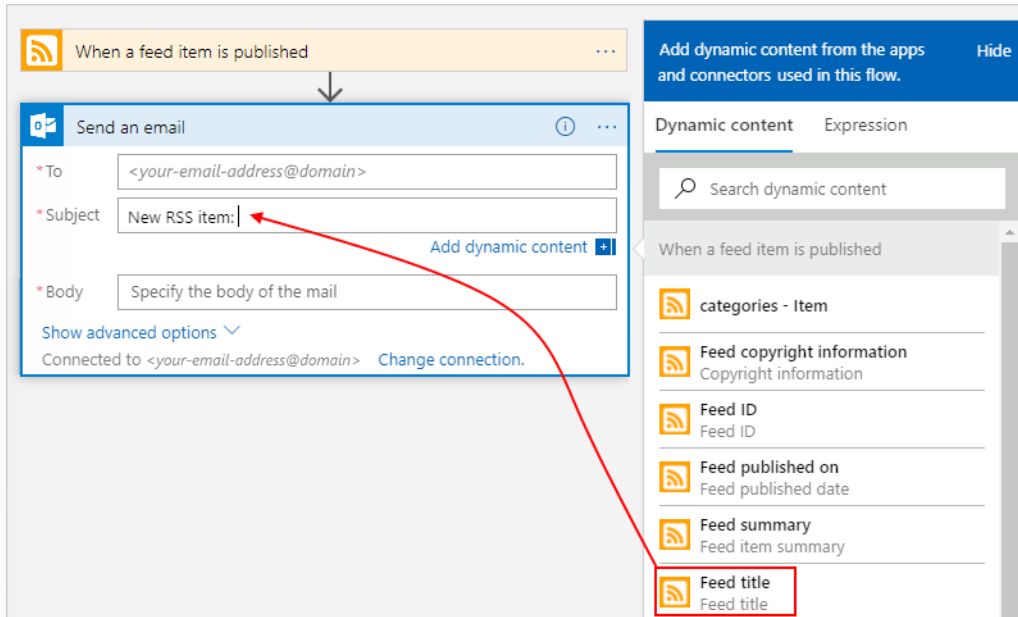
5. Together, the interval and frequency define the schedule for your logic app's trigger. This logic app checks the feed every minute.
 6. Save your project.
- Your logic app now has a trigger, but won't actually do anything until you add an action.

Adding an action

1. Under the **When a feed item is published** trigger, choose** + New step > Add an action**.
2. Use "send an email" as your filter. From the actions list, select the "send an email" action for the provider that you want.

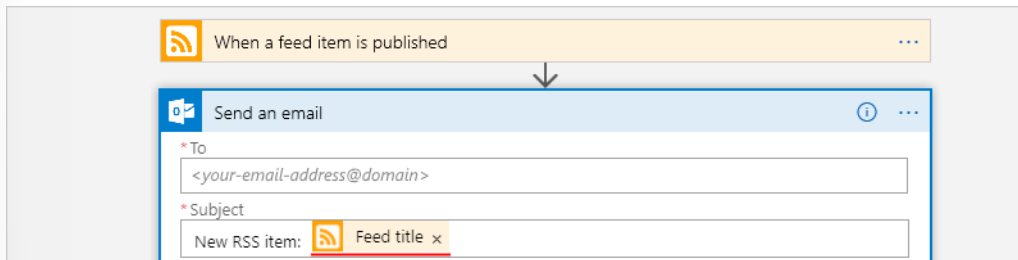


- 3.
4. To filter the actions list to a specific app or service, you can select that app or service first:
 - For Azure work or school accounts, select Office 365 Outlook.
 - For personal Microsoft accounts, select Outlook.com.
5. If asked for credentials, sign in to your email account so that Logic Apps creates a connection to your email account.
6. In the **Send an email** action, specify the data that you want the email to include.
7. a. In the **To** box, enter the recipient's email address. For testing purposes, you can use your own email address. For now, ignore the **Add dynamic content** list that appears. When you click inside some edit boxes, this list appears and shows any available parameters from the previous step that you can include as inputs in your workflow.
8. b. In the Subject box, enter this text with a trailing blank space: `New RSS item:`
9. c. From the **Add dynamic content** list, select **Feed title** to include the RSS item title.



10.

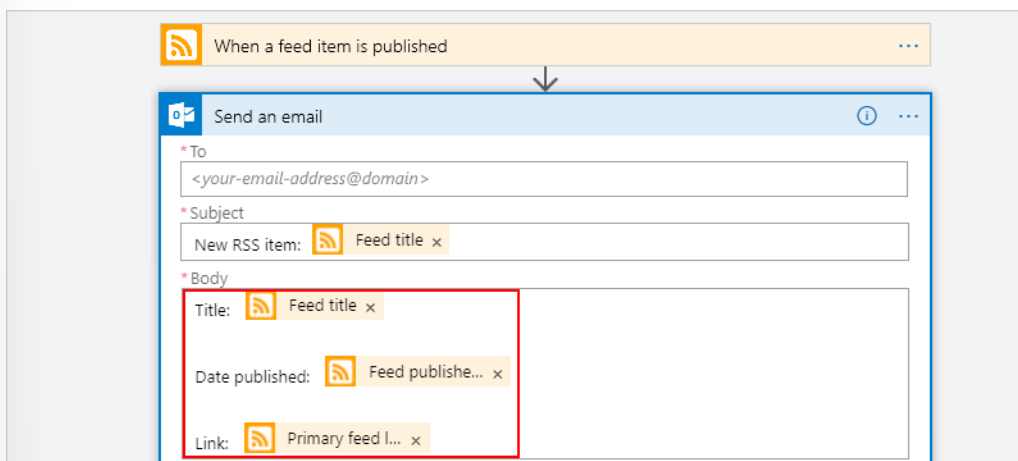
11. When you're done, the email subject looks like this example:



12.

13. If a "For each" loop appears on the designer, then you selected a token for an array, for example, the categories-Item token. For these kinds of tokens, the designer automatically adds this loop around the action that references that token. That way, your logic app performs the same action on each array item. To remove the loop, choose the ellipses (...) on the loop's title bar, then choose Delete.

14. d. In the **Body** box, enter this text, and select these tokens for the email body. To add blank lines in an edit box, press **Shift + Enter**.

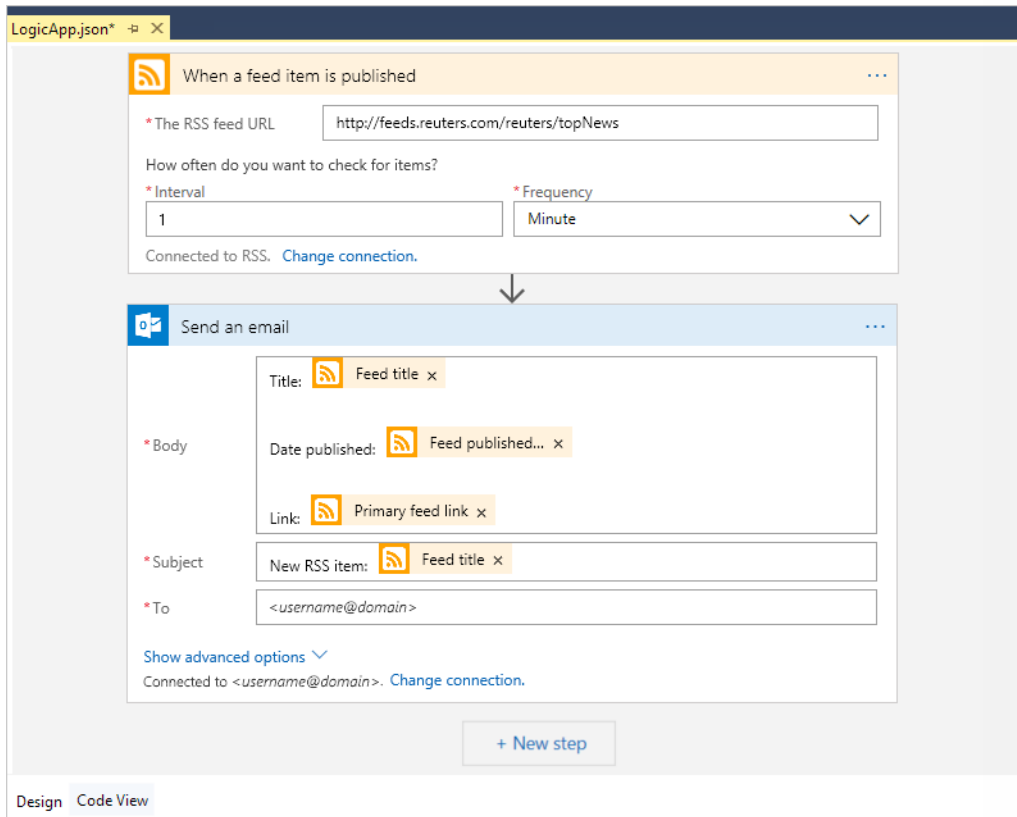


15.

Property	Description
Feed Title	The item's title
Feed published on	The item's publishing date and time
Primary feed link	Minute

16. Save your logic app.

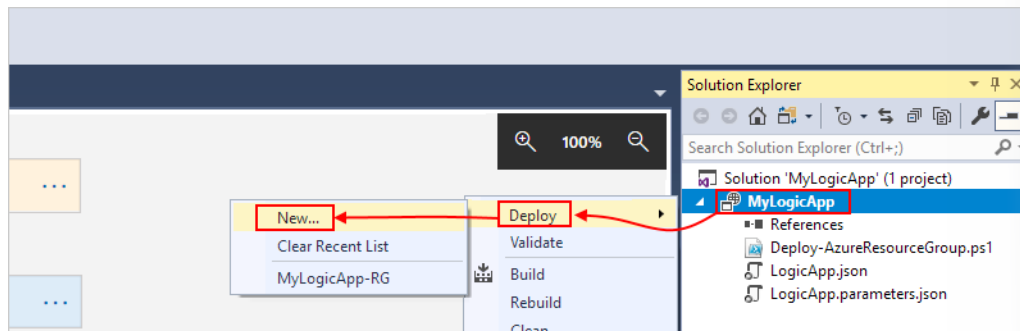
When you're done, your logic app looks like this example:



Deploy the Logic App to Azure

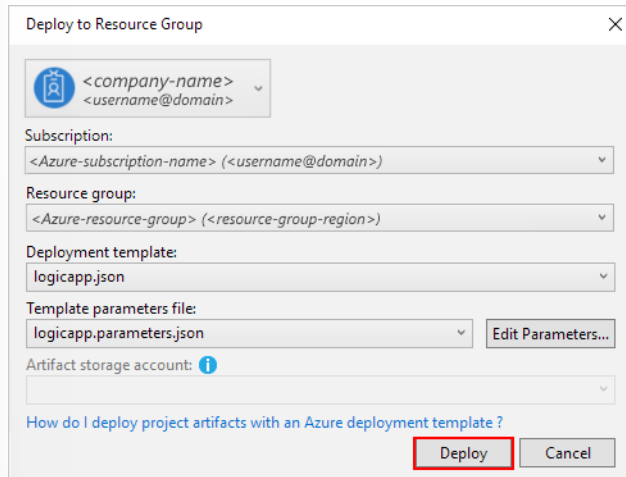
Before you can run your logic app, deploy the app from Visual Studio to Azure, which just takes a few steps.

1. In Solution Explorer, on your project's shortcut menu, select **Deploy > New**. If prompted, sign in with your Azure account.



2.

- For this deployment, keep the Azure subscription, resource group, and other default settings. When you're ready, choose **Deploy**.



Deploy to Resource Group

<company-name>
<username@domain>

Subscription:
<Azure-subscription-name> (<username@domain>)

Resource group:
<Azure-resource-group> (<resource-group-region>)

Deployment template:
logicapp.json

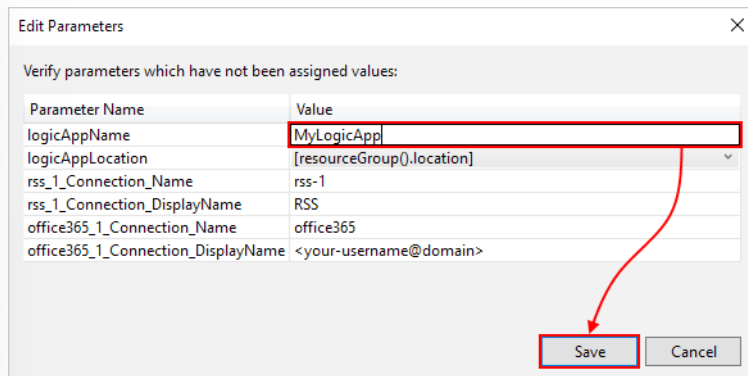
Template parameters file:
logicapp.parameters.json Edit Parameters...

Artifact storage account: i

[How do I deploy project artifacts with an Azure deployment template ?](#)

Deploy Cancel

-
-
-
-
- If the **Edit Parameters** box appears, provide the resource name for the logic app to use at deployment, then save your settings, for example:



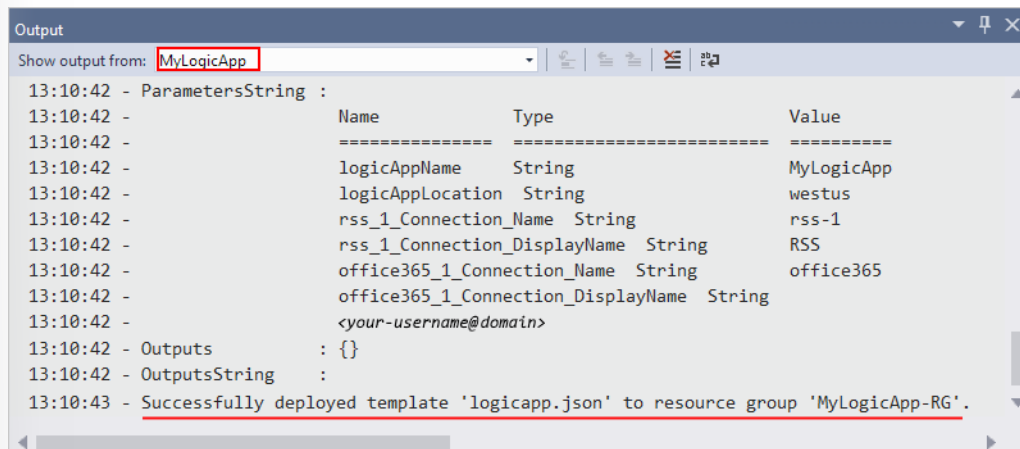
Edit Parameters

Verify parameters which have not been assigned values:

Parameter Name	Value
logicAppName	MyLogicApp
logicAppLocation	[resourceGroup().location]
rss_1_Connection_Name	rss-1
rss_1_Connection_DisplayName	RSS
office365_1_Connection_Name	office365
office365_1_Connection_DisplayName	<your-username@domain>

Save Cancel

-
-
-
-
-
-
- When deployment starts, your app's deployment status appears in the Visual Studio **Output** window. If the status doesn't appear, open the **Show output from** list, and select your Azure resource group.



Output

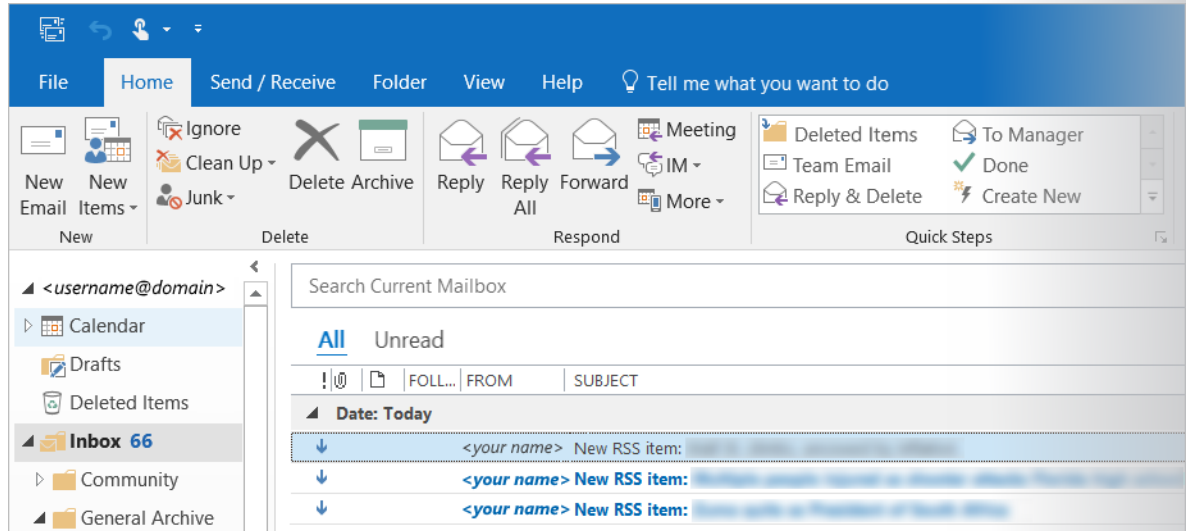
Show output from: MyLogicApp

```

13:10:42 - ParametersString :
13:10:42 -
13:10:42 -      Name      Type      Value
13:10:42 -      =====
13:10:42 -      logicAppName String      MyLogicApp
13:10:42 -      logicAppLocation String      westus
13:10:42 -      rss_1_Connection_Name String      rss-1
13:10:42 -      rss_1_Connection_DisplayName String      RSS
13:10:42 -      office365_1_Connection_Name String      office365
13:10:42 -      office365_1_Connection_DisplayName String
13:10:42 -      <your-username@domain>
13:10:42 - Outputs      : {}
13:10:42 - OutputsString :
13:10:43 - Successfully deployed template 'logicapp.json' to resource group 'MyLogicApp-RG'.
  
```

-
-
-
-
-
-
-
-

9. After deployment finishes, your logic app is live in the Azure portal and checks the RSS feed based on your specified schedule (every minute). If the RSS feed has new items, your logic app sends an email for each new item. Otherwise, your logic app waits until the next interval before checking again.
10. For example, here are sample emails that this logic app sends. If you don't get any emails, check your junk email folder.



- 11.
12. Technically, when the trigger checks the RSS feed and finds new items, the trigger fires, and the Logic Apps engine creates an instance of your logic app workflow that runs the actions in the workflow. If the trigger doesn't find new items, the trigger doesn't fire and "skips" instantiating the workflow.

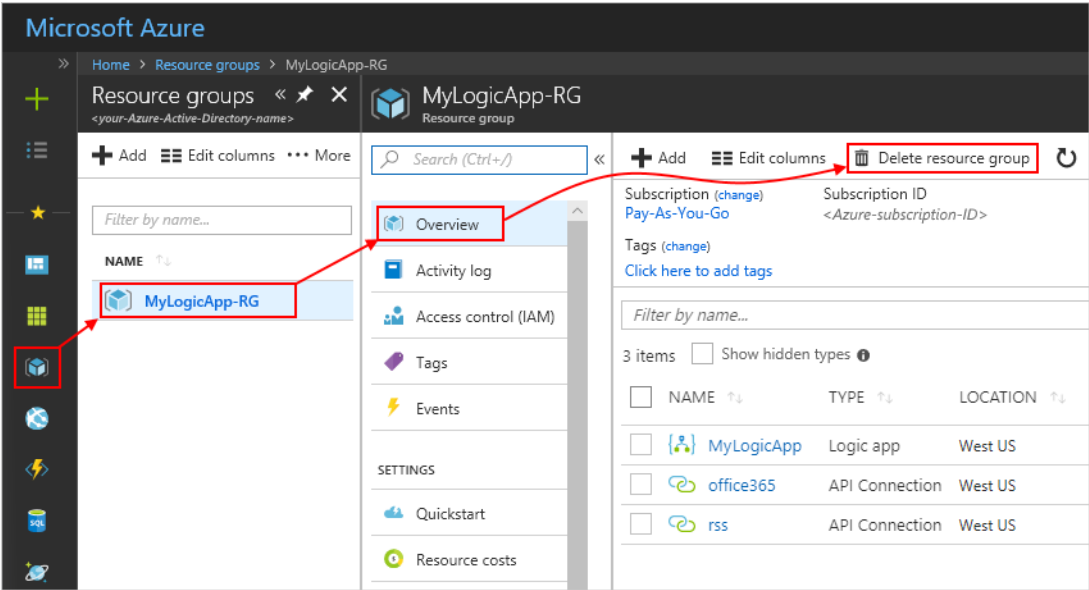
Congratulations, you've now successfully built and deployed your logic app with Visual Studio!

Clean up resources

When no longer needed, delete the resource group that contains your logic app and related resources.

1. Sign in to the **Azure portal**⁹ with the same account used to create your logic app.
2. On the main Azure menu, select **Resource groups**. Select the resource group for your logic app, and then select **Overview**.
3. On the **Overview** page, choose **Delete resource group**. Enter the resource group name as confirmation, and choose **Delete**.

⁹ <https://portal.azure.com/>



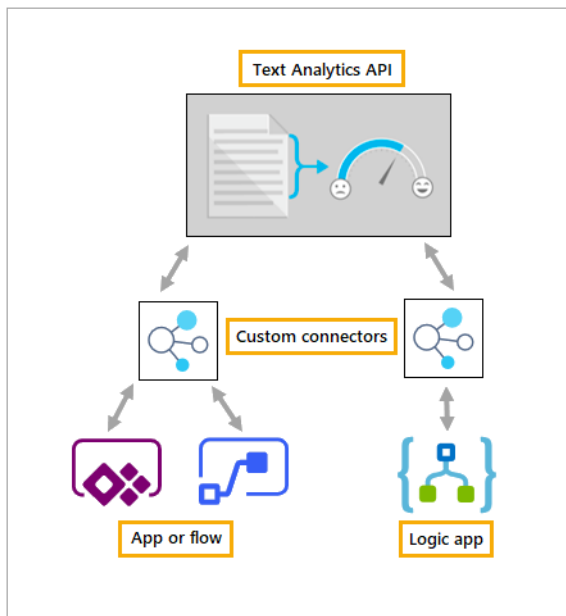
4.

Create custom connectors for Logic Apps

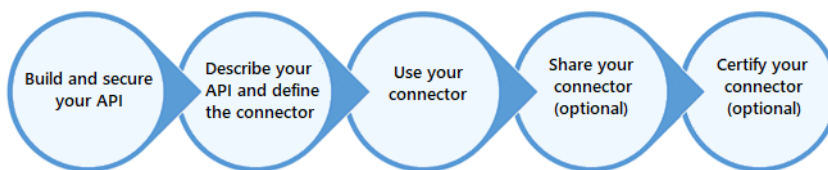
Custom connectors overview

Without writing any code, you can build workflows and apps with Azure Logic Apps, Microsoft Flow, and PowerApps. To help you integrate your data and business processes, these services offer 180+ connectors - for Microsoft services and products, as well as other services, like GitHub, Salesforce, Twitter, and more.

Sometimes though, you might want to call APIs, services, and systems that aren't available as prebuilt connectors. To support more tailored scenarios, you can build *custom connectors* with their own triggers and actions. These connectors are *function-based* - data is returned based on calling specific functions in the underlying service. The following diagram shows a custom connector for an API that detects sentiment in text.



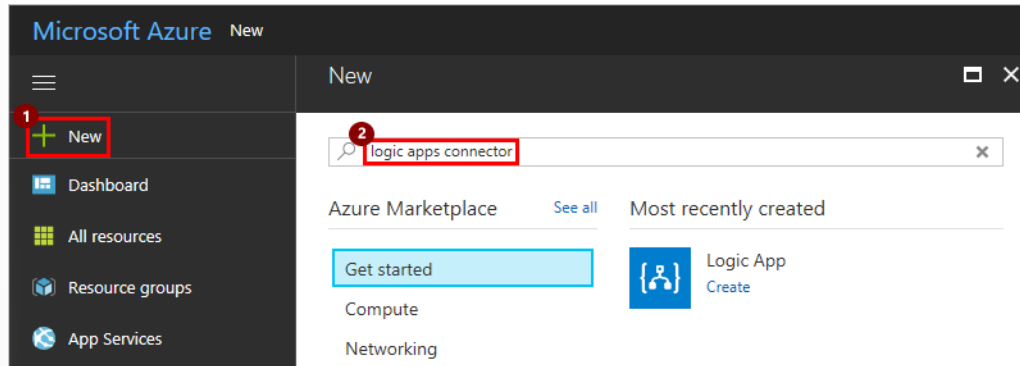
The following diagram shows the high-level tasks involved in creating and using custom connectors:



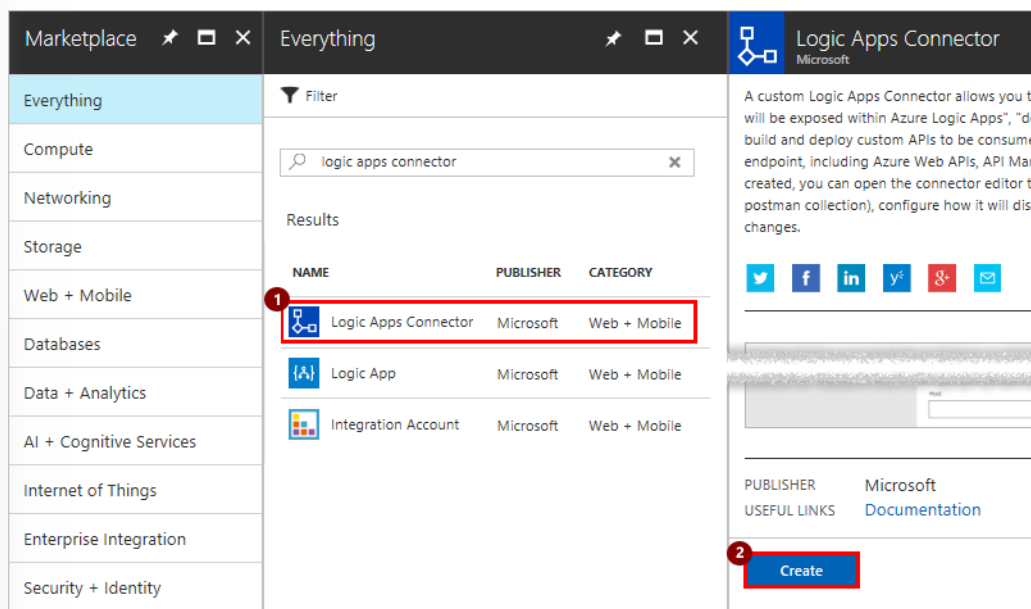
Create a custom connector in Logic Apps

In Logic Apps, you first create a custom connector (covered in this topic), and then you define the behavior of the connector using an OpenAPI definition or a Postman collection (covered in subsequent topics).

1. In the Azure portal, on the main Azure menu, choose** New**. In the search box, enter "logic apps connector" as your filter, and press Enter.



2.

3. From the results list, choose **Logic Apps Connector > Create**.

4.

5. Provide details for registering your connector as described in the table. When you're done, choose **Pin to dashboard > Create**.

Create Logic App custom connector

Logic App Custom Connector

* Name
SentimentDemo

* Subscription
Pay-As-You-Go

* Resource group
☐ Create new ☒ Use existing
Contoso Integration RG

Location
Brazil South

☒ Pin to dashboard

Create Automation options

6.

Property	Suggested Value	Description
Name	custom-connector-name	"SentimentDemo"
Subscription	Azure-subscription-name	Select your Azure subscription.
Resource Group	Azure-resource-group-name	Create or select an Azure group for organizing your Azure resources.
Location	deployment-region	A different location if you don't want to use the resource group location

- After Azure deploys your connector, the custom connector menu opens automatically. If not, choose your custom connector from the Azure dashboard.

Next Steps

Now that you've created a custom connector in Logic Apps, we'll show you how to define the behavior of the connector using an OpenAPI definition as an example.

Import the OpenAPI definition

To create a custom connector, you must describe the API you want to connect to so that the connector understands the API's operations and data structures.

Prerequisites

- An **OpenAPI definition**¹⁰ that describes the example API. When creating a custom connector, the OpenAPI definition must be less than 1 MB.
- An **API key**¹¹ for the Cognitive Services Text Analytics API
- One of the following subscriptions:
 - Azure**¹², if you're using Logic Apps
 - Microsoft Flow**¹³
 - PowerApps**¹⁴

You're now ready to work with the OpenAPI definition you downloaded. All the required information is contained in the definition, and you can review and update this information as you go through the custom connector wizard.

- Go to the Azure portal, and open the Logic Apps connector you created in the previous section.
- In your connector's menu, choose Logic Apps Connector, then choose Edit.

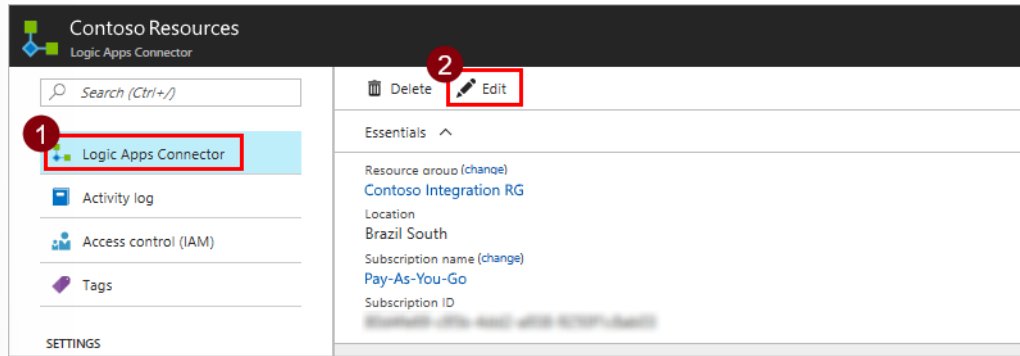
¹⁰ https://procsi.blob.core.windows.net/docs/SentimentDemo.openapi_definition.json

¹¹ <https://docs.microsoft.com/en-us/connectors/custom-connectors/index#get-an-api-key>

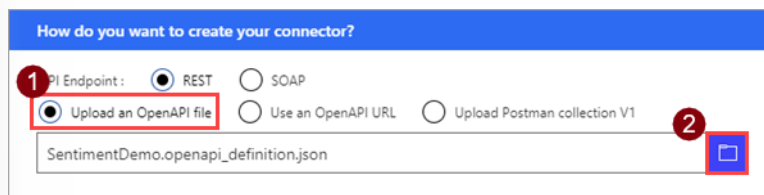
¹² <https://azure.microsoft.com/get-started/>

¹³ <https://docs.microsoft.com/flow/sign-up-sign-in>

¹⁴ <https://docs.microsoft.com/powerapps/signup-for-powerapps>



- 3.
4. Under **General**, choose **Upload an OpenAPI file**, then navigate to the OpenAPI definition that you created.




- 5.

Review general details

From this point, we'll show the Microsoft Flow UI, but the steps are largely the same across all three technologies. We'll point out any differences. In this part of the topic, we'll mostly review the UI and show you how the values correspond to sections of the OpenAPI file.

1. At the top of the wizard, make sure the name is set to "SentimentDemo", then choose **Create connector**.
2. On the **General** page, review the information that was imported from the OpenAPI definition, including the API host and the base URL for the API. The connector uses the API host and the base URL to determine how to call the API.

General information



Upload connector icon
Supported file formats are PNG and JPG. (< 1MB)

↑ Upload

Icon background color

A color to show behind the icon (e.g., '#007ee5')

Description

Uses the Cognitive Services Text Analytics Sentiment API to determine whether text is positive or negative

☐ Connect via on-premises data gateway [Learn more](#)

Scheme

☒ HTTPS ☐ HTTP

* Host

westus.api.cognitive.microsoft.com

Base URL

/

- 3.
4. The following section of the OpenAPI definition contains information for this page of the UI:

```
"info": {
  "version": "1.0.0",
  "title": "SentimentDemo",
  "description": "Uses the Cognitive Services Text Analytics Sentiment
API to determine whether text is positive or negative"
},
"host": "westus.api.cognitive.microsoft.com",
"basePath": "/",
"schemes": [
  "https"
]
```

Review authentication type

There are several options available for authentication in custom connectors. The Cognitive Services APIs use API key authentication, so that's what's specified in the OpenAPI definition.

On the **Security** page, review the authentication information for the API key.

The screenshot shows the 'Authentication type' configuration in the Azure Logic App portal. It has a blue header 'Authentication type' and a subtitle 'Choose what authentication is implemented by your API *'. Below this is a text input field containing 'API Key'. A horizontal separator line follows. Below the separator is another blue header 'API Key' and a subtitle 'Users will be required to provide the API Key when creating a connection'. There is a table with three columns: 'Parameter label', 'Parameter name', and 'Parameter location'. The table contains one row with the following values: 'API Key' in the 'Parameter label' column, 'OCP-APIM-Subscription-Ke' in the 'Parameter name' column, and 'Header' in the 'Parameter location' column.

Parameter label	Parameter name	Parameter location
API Key	OCP-APIM-Subscription-Ke	Header

The label is displayed when someone first makes a connection with the custom connector; you can choose **Edit** and change this value. The parameter name and location must match what the API expects, in this case "Ocp-Apim-Subscription-Key" and "Header".

The following section of the OpenAPI definition contains information for this page of the UI:

```
"securityDefinitions": {  
  "api_key": {  
    "type": "apiKey",  
    "in": "header",  
    "name": "Ocp-Apim-Subscription-Key"  
  }  
}
```

Create custom templates for Logic Apps

Logic App deployment template overview

After a logic app has been created, you might want to create it as an Azure Resource Manager template. This way, you can easily deploy the logic app to any environment or resource group where you might need it. For more about Resource Manager templates, see **authoring Azure Resource Manager templates**¹⁵ and **deploying resources by using Azure Resource Manager templates**¹⁶.

A logic app has three basic components:

- **Logic app resource:** Contains information about things like pricing plan, location, and the workflow definition.
- **Workflow definition:** Describes your logic app's workflow steps and how the Logic Apps engine should execute the workflow. You can view this definition in your logic app's **Code View** window. In the logic app resource, you can find this definition in the `definition` property.
- **Connections:** Refers to separate resources that securely store metadata about any connector connections, such as a connection string and an access token. In the logic app resource, your logic app references these resources in the `parameters` section.

You can view all these pieces of existing logic apps by using a tool like Azure Resource Explorer.

To make a template for a logic app to use with resource group deployments, you must define the resources and parameterize as needed. For example, if you're deploying to a development, test, and production environment, you likely want to use different connection strings to a SQL database in each environment. Or, you might want to deploy within different subscriptions or resource groups.

Create a Logic App deployment template

The easiest way to have a valid logic app deployment template is to use the Visual Studio Tools for Logic Apps. The Visual Studio tools generate a valid deployment template that can be used across any subscription or location.

A few other tools can assist you as you create a logic app deployment template. You can author by hand, that is, by using the resources already discussed here to create parameters as needed. Another option is to use a **logic app template creator**¹⁷ PowerShell module. This open-source module first evaluates the logic app and any connections that it is using, and then generates template resources with the necessary parameters for deployment. For example, if you have a logic app that receives a message from an Azure Service Bus queue and adds data to an Azure SQL database, the tool preserves all the orchestration logic and parameterizes the SQL and Service Bus connection strings so that they can be set at deployment.

Note: Connections must be within the same resource group as the logic app.

Install the logic app template PowerShell module

The easiest way to install the module is via the PowerShell Gallery, by using the command `Install-Module -Name LogicAppTemplate`.

¹⁵ <https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates>

¹⁶ <https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-template-deploy>

¹⁷ <https://github.com/jeffhollan/LogicAppTemplateCreator>

For the module to work with any tenant and subscription access token, we recommend that you use it with the **ARMClient**¹⁸ command-line tool. This **blog post**¹⁹ discusses ARMClient in more detail.

Generate a logic app template by using PowerShell

After PowerShell is installed, you can generate a template by using the following command:

```
armclient token $SubscriptionId | Get-LogicAppTemplate -LogicApp MyApp  
-ResourceGroup MyRG -SubscriptionId $SubscriptionId -Verbose | Out-File C:\  
template.json
```

\$SubscriptionId is the Azure subscription ID. This line first gets an access token via ARMClient, then pipes it through to the PowerShell script, and then creates the template in a JSON file.

Add parameters to a Logic App template

After you create your logic app template, you can continue to add or modify parameters that you might need. For example, if your definition includes a resource ID to an Azure function or nested workflow that you plan to deploy in a single deployment, you can add more resources to your template and parameterize IDs as needed. The same applies to any references to custom APIs or Swagger endpoints you expect to deploy with each resource group.

Add references for dependent resources to Visual Studio deployment templates

When you want your logic app to reference dependent resources, you can use **Azure Resource Manager template functions**²⁰ in your logic app deployment template. For example, you might want your logic app to reference an Azure Function or integration account that you want to deploy alongside your logic app. Follow these guidelines about how to use parameters in your deployment template so that the Logic App Designer renders correctly.

You can use logic app parameters in these kinds of triggers and actions:

- Child workflow
- Function app
- APIM call
- API connection runtime URL
- API connection path

And you can use template functions such as parameters, variables, resourceId, concat, etc. For example, here's how you can replace the Azure Function resource ID:

```
"parameters":{  
  "functionName": {  
    "type":"string",  
    "minLength":1,  
    "defaultValue":"<FunctionName>"  
  }  
}
```

¹⁸ <https://github.com/projectkudu/ARMClient>

¹⁹ <http://blog.davidebbo.com/2015/01/azure-resource-manager-client.html>

²⁰ <https://docs.microsoft.com/azure/azure-resource-manager/resource-group-template-functions>

```
},
```

And where you would use parameters:

```
"MyFunction": {
  "type": "Function",
  "inputs": {
    "body": {},
    "function": {
      "id": "[resourceId('Microsoft.Web/sites/functions', 'function-
App', parameters('functionName'))]"
    }
  },
  "runAfter": {}
}
```

As another example you can parameterize the Service Bus send message operation:

```
"Send_message": {
  "type": "ApiConnection",
  "inputs": {
    "host": {
      "connection": {
        "name": "@parameters('$connections')['servicebus']
['connectionId']"
      }
    },
    "method": "post",
    "path": "[concat('/@{encodeURIComponent('', parameters('queue-
uname'), '')}/messages')]",
    "body": {
      "ContentData": "@{base64(triggerBody())}"
    },
    "queries": {
      "systemProperties": "None"
    }
  },
  "runAfter": {}
}
```

Note: `host.runtimeUrl` is optional and can be removed from your template if present.

For the Logic App Designer to work when you use parameters, you must provide default values, for example:

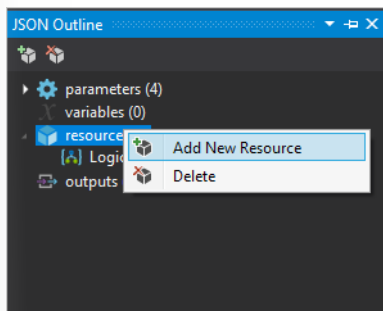
```
"parameters": {
  "IntegrationAccount": {
    "type": "string",
    "minLength": 1,
    "defaultValue": "/subscriptions/<subscriptionID>/resourceGroups/<re-
sourceGroupName>/providers/Microsoft.Logic/integrationAccounts/<integra-
tionAccountName>"
  }
}
```

```
}  
},
```

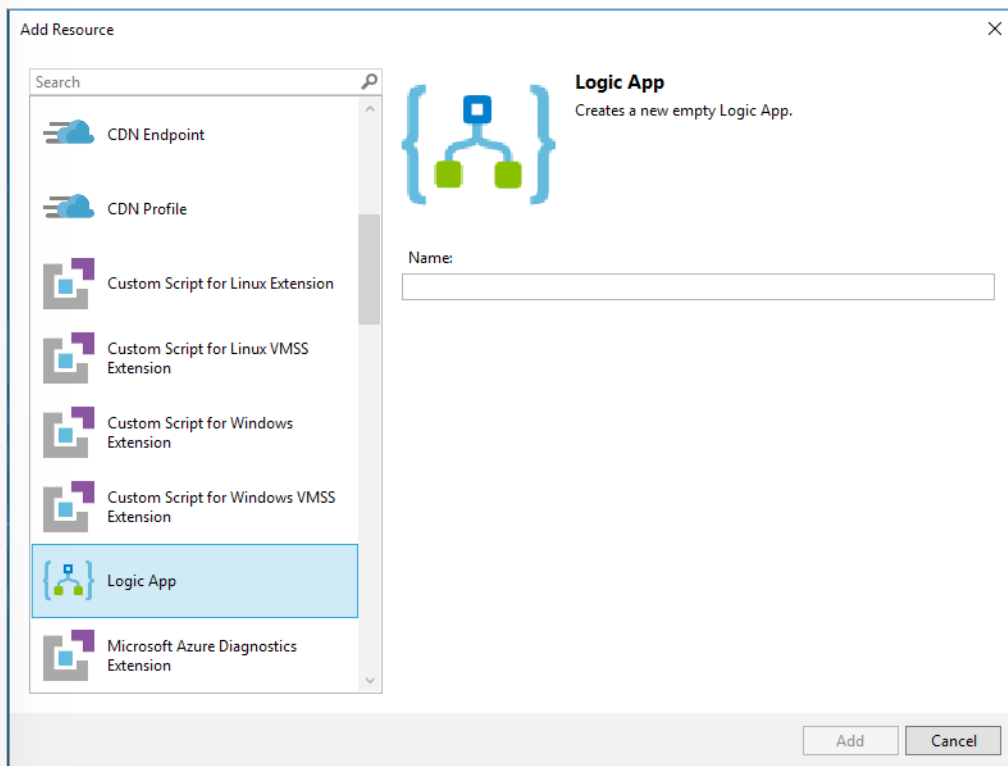
Adding your Logic App to an existing Resource Group

If you have an existing Resource Group project, you can add your logic app to that project in the JSON Outline window. You can also add another logic app alongside the app you previously created.

1. Open the `<template>.json` file.
2. To open the JSON Outline window, go to **View > Other Windows > JSON Outline**.
3. To add a resource to the template file, click **Add Resource** at the top of the JSON Outline window. Or in the JSON Outline window, right-click resources, and select **Add New Resource**.



- 4.
5. In the **Add Resource** dialog box, find and select **Logic App**. Name your logic app, and choose **Add**.



- 6.

Deploy a logic app template

You can deploy your template by using any tools like PowerShell, REST API, Visual Studio Team Services Release Management, and template deployment through the Azure portal. Also, to store the values for parameters, we recommend that you create a parameter file. Learn how to deploy resources with Azure Resource Manager templates and PowerShell or deploy resources with Azure Resource Manager templates and the Azure portal.

Authorize OAuth connections

After deployment, the logic app works end-to-end with valid parameters. However, you must still authorize OAuth connections to generate a valid access token. To authorize OAuth connections, open the logic app in the Logic Apps Designer, and authorize these connections. Or for automated deployment, you can use a script to consent to each OAuth connection. There's an example script on GitHub under the **LogicAppConnectionAuth**²¹ project.

²¹ <https://github.com/logicappsio/LogicAppConnectionAuth>

Review questions

Module 1 review questions

API Management

Azure API Management is an API gateway-as-a-service that streamlines many of the common tasks that are necessary when creating an API for external users, can you name some things that includes?

> Click to see suggested answer

- Creating a successful and useful developer portal.
- Helping to secure API endpoints from anonymous or unwanted access.
- Managing existing developer access through cache mechanisms, throttling, and other policies.
- Building a monitoring and analytics platform to diagnose issues and monitor adoption.
- Providing business users and developers with deep insights into how each API is specifically used.

Azure Application Gateway

Can Azure Application Gateway support app load balancing outside of the transport layer?

> Click to see suggested answer

Alerts can notify you of an errant scenario by either sending an email notification or invoking a webhook:

Azure Application Gateway supports application layer (layer 7 in the OSI model) load balancing. Since Azure Application Gateway is at the application layer, the service can handle custom routing, session affinity, Secure Sockets Layer (SSL) termination, firewall management, redirection, and other scenarios that require some knowledge of the application code. Application Gateway can perform URL-based routing and route traffic based on the incoming URL.

API management policies

Policies are a powerful capability of API Management that allows the Azure portal to change the behavior of the API through configuration. Policies are a quick way to change the behavior of an API for external developers without requiring any code changes in the actual back-end API application. How would you change the behavior of the API endpoint?

> Click to see suggested answer

To change the behavior of the API endpoint within API Management, you add XML elements to either the `inbound` or the `outbound` element. For example, you can convert XML data that comes from your back-end API to JSON for external users:

```
<policies>
  <inbound>
    <base />
  </inbound>
  <outbound>
```

```
        <base />
        <xml-to-json kind="direct" apply="always" consider-accept-header="-
false" />
    </outbound>
</policies>
```




Module 2 Integrate Azure Search within solutions

Create and query an Azure Search index

Azure Search overview

Azure Search is a search-as-a-service cloud solution that gives developers APIs and tools for adding a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications. Query execution is over a user-defined index.

- Build a search corpus containing only your data, sourced from multiple content types and platforms.
- Leverage AI-powered indexing to extract text and features from image files, or entities and key phrases from raw text.
- Create intuitive search experiences with facet navigation and filters, synonyms, auto-complete, and text analysis for “did you mean” auto-corrected search terms.
- Add geo-search for “find near me”, language analyzers for non-English full text search, and scoring logic for search rank.

Functionality is exposed through a simple REST API or .NET SDK that masks the inherent complexity of information retrieval. In addition to APIs, the Azure portal provides administration and content management support, with tools for prototyping and querying your indexes. Because the service runs in the cloud, infrastructure and availability are managed by Microsoft.

How to use Azure Search

Step 1: Provision service

You can provision an Azure Search service in the Azure portal or through the Azure Resource Management API. You can choose either the free service shared with other subscribers, or a paid tier that dedicates resources used only by your service. For paid tiers, you can scale a service in two dimensions:

- Add Replicas to grow your capacity to handle heavy query loads.

- Add Partitions to grow storage for more documents.

By handling document storage and query throughput separately, you can calibrate resourcing based on production requirements.

Step 2: Create index

Before you can upload searchable content, you must first define an Azure Search index. An index is like a database table that holds your data and can accept search queries. You define the index schema to map to reflect the structure of the documents you wish to search, similar to fields in a database.

A schema can be created in the Azure portal, or programmatically using the .NET SDK or REST API.

Step 3: Load data

After you define an index, you're ready to upload content. You can use either a push or pull model.

The pull model retrieves data from external data sources. It's supported through indexers that streamline and automate aspects of data ingestion, such as connecting to, reading, and serializing data. Indexers are available for Azure Cosmos DB, Azure SQL Database, Azure Blob Storage, and SQL Server hosted in an Azure VM. You can configure an indexer for on demand or scheduled data refresh.

The push model is provided through the SDK or REST APIs, used for sending updated documents to an index. You can push data from virtually any dataset using the JSON format.

Step 4: Search

After populating an index, you can issue search queries to your service endpoint using simple HTTP requests with REST API or the .NET SDK.

Azure Search feature summary

Category	Features
Full text search and text analysis	<p>Full text search is a primary use case for most search-based apps. Queries can be formulated using a supported syntax.</p> <p>Simple query syntax provides logical operators, phrase search operators, suffix operators, precedence operators.</p> <p>Lucene query syntax includes all operations in simple syntax, with extensions for fuzzy search, proximity search, term boosting, and regular expressions.</p>

Category	Features
Data integration	<p>Azure Search indexes accept data from any source, provided it is submitted as a JSON data structure.</p> <p>Optionally, for supported data sources in Azure, you can use indexers to automatically crawl Azure SQL Database, Azure Cosmos DB, or Azure Blob storage for searchable content in primary data stores. Azure Blob indexers can perform <i>document cracking</i> to extract text from major file formats, including Microsoft Office, PDF, and HTML documents.</p>
Linguistic analysis	<p>Analyzers are components used for text processing during indexing and search operations. There are two types.</p> <p>Custom lexical analyzers are used for complex search queries using phonetic matching and regular expressions.</p> <p>Language analyzers from Lucene or Microsoft are used to intelligently handle language-specific linguistics including verb tenses, gender, irregular plural nouns (for example, 'mouse' vs. 'mice'), word de-compounding, word-breaking (for languages with no spaces), and more.</p>
Geo-search	<p>Azure Search processes, filters, and displays geographic locations. It enables users to explore data based on the proximity of a search result to a physical location.</p>

Category	Features
User experience features	<p>Search suggestions also works off of partial text inputs in a search bar, but the results are actual documents in your index rather than query terms.</p> <p>Synonyms associates equivalent terms that implicitly expand the scope of a query, without the user having to provide the alternate terms.</p> <p>Faceted navigation is enabled through a single query parameter. Azure Search returns a faceted navigation structure you can use as the code behind a categories list, for self-directed filtering (for example, to filter catalog items by price-range or brand).</p> <p>Filters can be used to incorporate faceted navigation into your application's UI, enhance query formulation, and filter based on user- or developer-specified criteria. Create filters using the OData syntax.</p> <p>Hit highlighting applies text formatting to a matching keyword in search results. You can choose which fields return highlighted snippets.</p> <p>Sorting is offered for multiple fields via the index schema and then toggled at query-time with a single search parameter.</p> <p>Paging and throttling your search results is straightforward with the finely tuned control that Azure Search offers over your search results.</p>

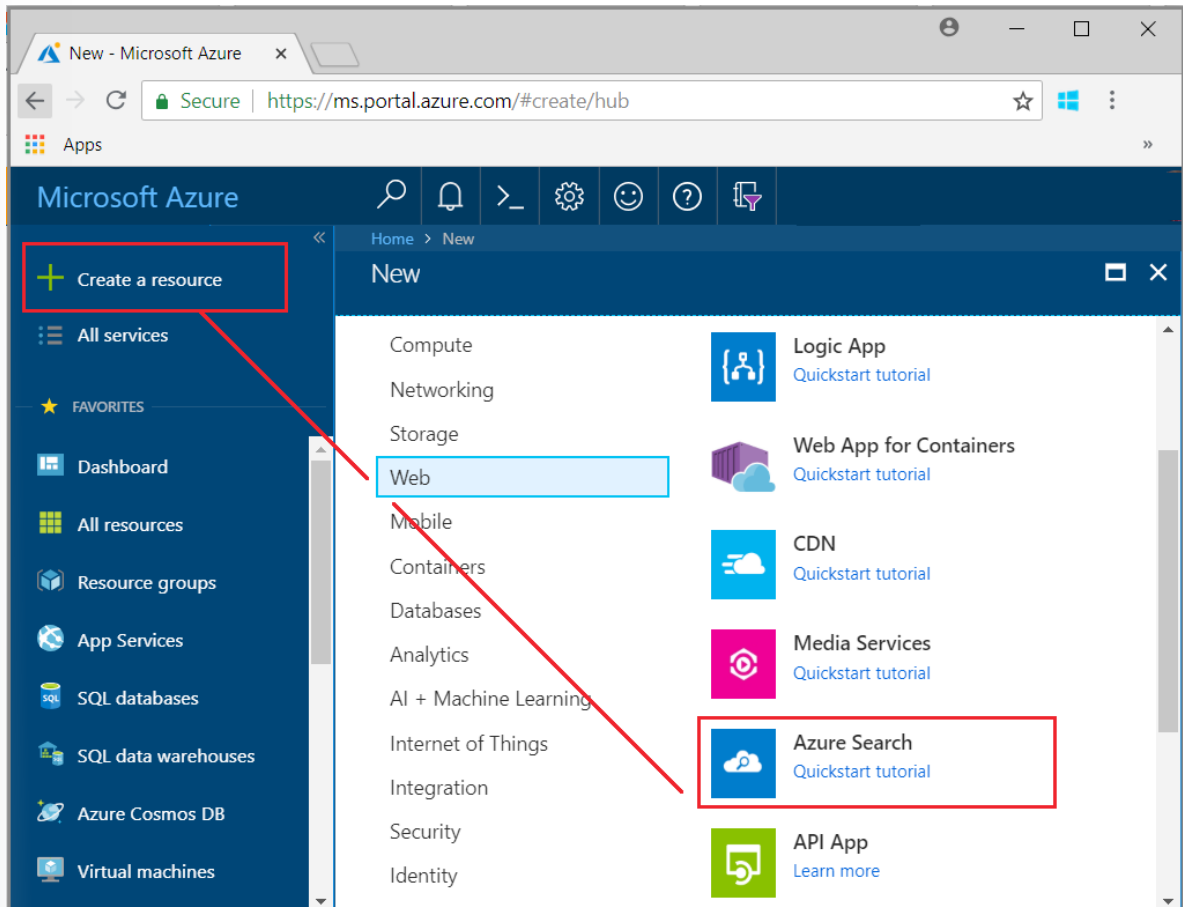
Category	Features
Relevance	<p>Simple scoring is a key benefit of Azure Search. Scoring profiles are used to model relevance as a function of values in the documents themselves. For example, you might want newer products or discounted products to appear higher in the search results. You can also build scoring profiles using tags for personalized scoring based on customer search preferences you've tracked and stored separately.</p>
Monitoring and reporting	<p>Search traffic analytics are collected and analyzed to unlock insights from what users are typing into the search box.</p> <p>Metrics on queries per second, latency, and throttling are captured and reported in portal pages with no additional configuration required. You can also easily monitor index and document counts so that you can adjust capacity as needed.</p>
Tools for prototyping and inspection	<p>In the portal, you can use the Import data wizard to configure indexers, index designer to stand up an index, and Search explorer to test queries and refine scoring profiles. You can also open any index to view its schema.</p>
Infrastructure	<p>The highly available platform ensures an extremely reliable search service experience. When scaled properly, Azure Search offers a 99.9% SLA.</p> <p>Fully managed and scalable as an end-to-end solution, Azure Search requires absolutely no infrastructure management. Your service can be tailored to your needs by scaling in two dimensions to handle more document storage, higher query loads, or both.</p>

Create an Azure Search service in the portal

Before we create the Search index we're going to make sure we have an Azure Search resource available for us to use. In this article you will learn how to create an Azure Search resource in the Azure portal.

Step 1: Find Azure Search

1. Sign in to the Azure portal.
2. Click the plus sign (+ **Create Resource**) in the top-left corner.
3. Use the search bar to find "Azure Search" or navigate to the resource through **Web > Azure Search**.



4.

Step 2: Enter the details and create the resource

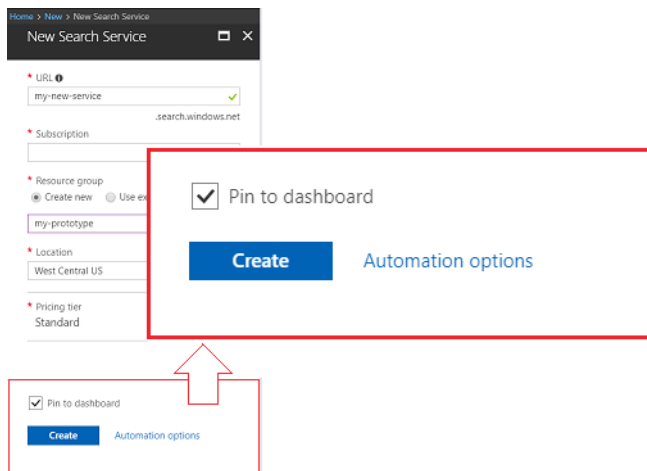
You are asked to provide the following information when creating a new Search Service.

- **Service name and URL endpoint:** A service name is part of the URL endpoint against which API calls are issued: `https://your-service-name.search.windows.net`. Enter your service name in the **URL** field. For example, if you want the endpoint to be `https://my-app-name-01.search.windows.net`, you would enter `my-app-name-01`.
- **Select a subscription:** If you have more than one subscription, choose one that also has data or file storage services. Azure Search can autodetect Azure Table and Blob storage, SQL Database, and Azure Cosmos DB for indexing via indexers, but only for services in the same subscription.
- **Select a resource group:** A resource group is a collection of Azure services and resources used together. For example, if you are using Azure Search to index a SQL database, then both services should be part of the same resource group. If you aren't combining resources into a single group, or if existing resource groups are filled with resources used in unrelated solutions, create a new resource group just for your Azure Search resource.

- **Tip:** Deleting a resource group also deletes the services within it. For prototype projects utilizing multiple services, putting all of them in the same resource group makes cleanup easier after the project is over.
- **Select a hosting location:** As an Azure service, Azure Search can be hosted in datacenters around the world. Prices can differ by geography. If you are planning to use cognitive search, choose a region with **feature availability**¹.
- **Select a pricing tier (SKU):** Azure Search is currently offered in multiple pricing tiers: Free, Basic, or Standard. Each tier has its own capacity and limits. See **Choose a pricing tier or SKU**² for guidance.
- Standard is usually chosen for production workloads, but most customers start with the Free service. A pricing tier **cannot be changed** once the service is created. If you need a higher or lower tier later, you have to re-create the service.

Create your service

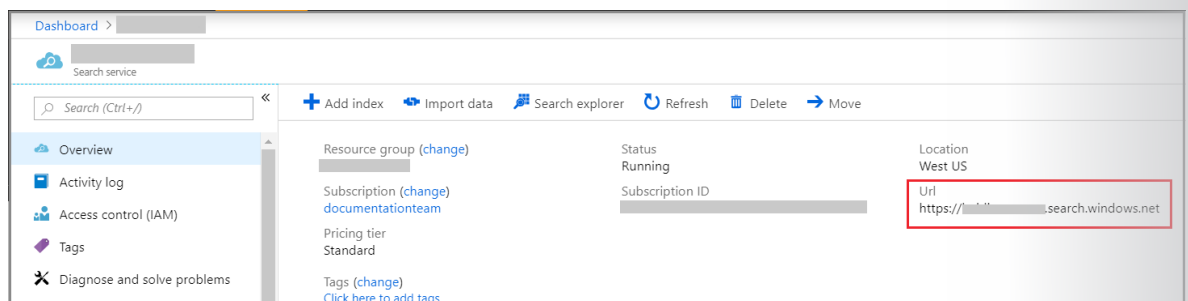
After you have entered all of the details, click **Create** to create the resource. Remember to pin your service to the dashboard for easy access whenever you sign in.



Step 3: Get a key and URL endpoint

With few exceptions, using your new service requires that you provide the URL endpoint and an authorization api-key.

1. In the service overview page, locate and copy the URL endpoint on the right side of the page.

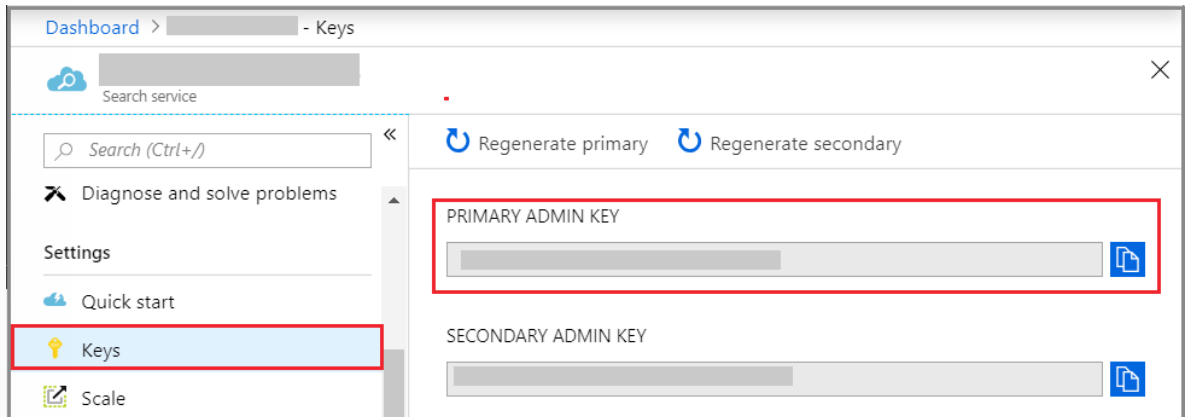


- 2.

¹ <https://docs.microsoft.com/en-us/azure/search/cognitive-search-quickstart-blob#supported-regions>

² <https://docs.microsoft.com/en-us/azure/search/search-sku-tier>

3. In the left navigation pane, select **Keys** and then copy either one of the admin keys (they are equivalent). Admin api-keys are required for creating, updating, and deleting objects on your service.



- 4.

An endpoint and key are not needed for portal-based tasks. The portal is already linked to your Azure Search resource with admin rights.

Next

We'll use the URL endpoint and authorization key to create a Search Index.

Create an Azure Search index using the .NET SDK

Now that the Azure Search resource is available, we will walk you through the process of creating an Azure Search index using the Azure Search .NET SDK.

Step 1: Identify your Azure Search service's admin api-key

Now that you have provisioned an Azure Search service, you are almost ready to issue requests against your service endpoint using the .NET SDK. First, you will need to obtain one of the admin api-keys that was generated for the search service you provisioned. The .NET SDK will send this api-key on every request to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

1. To find your service's api-keys, sign in to the Azure portal
2. Go to your Azure Search service's blade
3. Click on the "Keys" icon

Your service will have *admin* keys and *query* keys.

- Your primary and secondary *admin* keys grant full rights to all operations, including the ability to manage the service, create and delete indexes, indexers, and data sources. There are two keys so that you can continue to use the secondary key if you decide to regenerate the primary key, and vice-versa.
- Your *query* keys grant read-only access to indexes and documents, and are typically distributed to client applications that issue search requests.

For the purposes of creating an index, you can use either your primary or secondary admin key.

Step 2: Create an instance of the SearchServiceClient class

To start using the Azure Search .NET SDK, you will need to create an instance of the `SearchServiceClient` class. This class has several constructors. The one you want takes your search service name and a `SearchCredentials` object as parameters. `SearchCredentials` wraps your api-key.

The code below creates a new `SearchServiceClient` using values for the search service name and api-key that are stored in the application's config file (`appsettings.json` in the case of the sample application):

```
private static SearchServiceClient CreateSearchServiceClient(IConfiguration-
Root configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string adminApiKey = configuration["SearchServiceAdminApiKey"];

    SearchServiceClient serviceClient = new SearchServiceClient(searchSer-
viceName, new SearchCredentials(adminApiKey));
    return serviceClient;
}
```

`SearchServiceClient` has an `Indexes` property. This property provides all the methods you need to create, list, update, or delete Azure Search indexes.

Note: The `SearchServiceClient` class manages connections to your search service. In order to avoid opening too many connections, you should try to share a single instance of `SearchServiceClient` in your application if possible. Its methods are thread-safe to enable such sharing.

Step 3: Define your Azure Search index

A single call to the `Indexes.Create` method will create your index. This method takes as a parameter an `Index` object that defines your Azure Search index. You need to create an `Index` object and initialize it as follows:

1. Set the `Name` property of the `Index` object to the name of your index.
2. Set the `Fields` property of the `Index` object to an array of `Field` objects. The easiest way to create the `Field` objects is by calling the `FieldBuilder.BuildForType` method, passing a model class for the type parameter. A model class has properties that map to the fields of your index. This allows you to bind documents from your search index to instances of your model class.
3. If you don't plan to use a model class, you can still define your index by creating `Field` objects directly. You can provide the name of the field to the constructor, along with the data type (or analyzer for string fields). You can also set other properties like `IsSearchable`, `IsFilterable`, etc.

It is important that you keep your search user experience and business needs in mind when designing your index as each field must be assigned the appropriate properties. These properties control which search features (filtering, faceting, sorting full-text search, etc.) apply to which fields. For any property you do not explicitly set, the `Field` class defaults to disabling the corresponding search feature unless you specifically enable it.

For our example, we've named our index "hotels" and defined our fields using a model class. Each property of the model class has attributes which determine the search-related behaviors of the corresponding index field. The model class is defined as follows:


```
using System;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Microsoft.Spatial;
using Newtonsoft.Json;

// The SerializePropertyNamesAsCamelCase attribute is defined in the Azure
// Search .NET SDK.
// It ensures that Pascal-case property names in the model class are mapped
// to camel-case
// field names in the index.
[SerializePropertyNamesAsCamelCase]
public partial class Hotel
{
    [System.ComponentModel.DataAnnotations.Key]
    [IsFilterable]
    public string HotelId { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public double? BaseRate { get; set; }

    [IsSearchable]
    public string Description { get; set; }

    [IsSearchable]
    [Analyzer(AnalyzerName.AsString.FrLucene)]
    [JsonProperty("description_fr")]
    public string DescriptionFr { get; set; }

    [IsSearchable, IsFilterable, IsSortable]
    public string HotelName { get; set; }

    [IsSearchable, IsFilterable, IsSortable, IsFacetable]
    public string Category { get; set; }

    [IsSearchable, IsFilterable, IsFacetable]
    public string[] Tags { get; set; }

    [IsFilterable, IsFacetable]
    public bool? ParkingIncluded { get; set; }

    [IsFilterable, IsFacetable]
    public bool? SmokingAllowed { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public DateTimeOffset? LastRenovationDate { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public int? Rating { get; set; }

    [IsFilterable, IsSortable]
```

```
public GeographyPoint Location { get; set; }
}
```

We have carefully chosen the attributes for each property based on how we think they will be used in an application. For example, it is likely that people searching for hotels will be interested in keyword matches on the description field, so we enable full-text search for that field by adding the `IsSearchable` attribute to the `Description` property.

Please note that exactly *one* field in your index of type `string` must be the designated as the key field by adding the `Key` attribute (see `HotelId` in the above example).

The index definition above uses a language analyzer for the `description_fr` field because it is intended to store French text. See the **Language support topic**³ as well as the corresponding **blog post**⁴ for more information about language analyzers.

By default, the name of each property in your model class is used as the name of the corresponding field in the index. If you want to map all property names to camel-case field names, mark the class with the `SerializePropertyNameAsCamelCase` attribute. If you want to map to a different name, you can use the `JsonProperty` attribute like the `DescriptionFr` property above. The `JsonProperty` attribute takes precedence over the `SerializePropertyNameAsCamelCase` attribute.

Now that we've defined a model class, we can create an index definition very easily:

```
var definition = new Index()
{
    Name = "hotels",
    Fields = FieldBuilder.BuildForType<Hotel>()
};
```

Step 4: Create the index

Now that you have an initialized `Index` object, you can create the index simply by calling `Indexes.Create` on your `SearchServiceClient` object:

```
serviceClient.Indexes.Create(definition);
```

For a successful request, the method will return normally. If there is a problem with the request such as an invalid parameter, the method will throw `CloudException`.

When you're done with an index and want to delete it, just call the `Indexes.Delete` method on your `SearchServiceClient`. For example, this is how we would delete the "hotels" index:

```
serviceClient.Indexes.Delete("hotels");
```

Note: The example code shown above uses the synchronous methods of the Azure Search .NET SDK for simplicity. We recommend that you use the asynchronous methods in your own applications to keep them scalable and responsive. For example, in the examples above you could use `CreateAsync` and `DeleteAsync` instead of `Create` and `Delete`.

³ <https://docs.microsoft.com/rest/api/searchservice/Language-support>

⁴ <https://azure.microsoft.com/blog/language-support-in-azure-search/>

Next steps

After creating an Azure Search index, you will be ready to upload your content into the index so you can start searching your data.

Upload data to Azure Search using the .NET SDK

Now that the Azure Search resource and Search Index are in place it's time to import data.

In order to push documents into your index using the .NET SDK, you will need to:

1. Create a `SearchIndexClient` object to connect to your search index.
2. Create an `IndexBatch` containing the documents to be added, modified, or deleted.
3. Call the `Documents.Index` method of your `SearchIndexClient` to send the `IndexBatch` to your search index.

Step 1: Create an instance of the `SearchIndexClient` class

To import data into your index using the Azure Search .NET SDK, you will need to create an instance of the `SearchIndexClient` class. You can construct this instance yourself, but it's easier if you already have a `SearchServiceClient` instance to call its `Indexes.GetClient` method. For example, here is how you would obtain a `SearchIndexClient` for the index named "hotels" from a `SearchServiceClient` named `serviceClient`:

```
ISearchIndexClient indexClient = serviceClient.Indexes.GetClient("hotels");
```

`SearchIndexClient` has a `Documents` property. This property provides all the methods you need to add, modify, delete, or query documents in your index.

In a typical search application, index management and population is handled by a separate component from search queries. `Indexes.GetClient` is convenient for populating an index because it saves you the trouble of providing another `SearchCredentials`. It does this by passing the admin key that you used to create the `SearchServiceClient` to the new `SearchIndexClient`. However, in the part of your application that executes queries, it is better to create the `SearchIndexClient` directly so that you can pass in a query key instead of an admin key. This is consistent with the principle of least privilege and will help to make your application more secure.

Step 2: Create an `IndexBatch`

Decide which indexing action to use

To import data using the .NET SDK, you will need to package up your data into an `IndexBatch` object. An `IndexBatch` encapsulates a collection of `IndexAction` objects, each of which contains a document and a property that tells Azure Search what action to perform on that document (upload, merge, delete, etc). Depending on which of the below actions you choose, only certain fields must be included for each document:

Action	Description	Necessary fields for each document	Notes
Upload	An Upload action is similar to an "upsert" where the document will be inserted if it is new and updated/replaced if it exists.	key, plus any other fields you wish to define	When updating/replacing an existing document, any field that is not specified in the request will have its field set to null. This occurs even when the field was previously set to a non-null value.
Merge	Updates an existing document with the specified fields. If the document does not exist in the index, the merge will fail.	key, plus any other fields you wish to define	Any field you specify in a merge will replace the existing field in the document. This includes fields of type <code>DataType.Collection(DataType.String)</code> . For example, if the document contains a field tags with value <code>["budget"]</code> and you execute a merge with value <code>["economy", "pool"]</code> for tags, the final value of the tags field will be <code>["economy", "pool"]</code> . It will not be <code>["budget", "economy", "pool"]</code> .
MergeOrUpload	This action behaves like Merge if a document with the given key already exists in the index. If the document does not exist, it behaves like Upload with a new document.	key, plus any other fields you wish to define	-

Action	Description	Necessary fields for each document	Notes
Delete	Removes the specified document from the index.	key only	Any fields you specify other than the key field will be ignored. If you want to remove an individual field from a document, use <code>Merge</code> instead and simply set the field explicitly to null.

You can specify what action you want to use with the various static methods of the `IndexBatch` and `IndexAction` classes, as shown next.

Construct your `IndexBatch`

Now that you know which actions to perform on your documents, you are ready to construct the `IndexBatch`. The example below shows how to create a batch with a few different actions. Note that our example uses a custom class called `Hotel` that maps to a document in the "hotels" index.

```
var actions =
    new IndexAction<Hotel>[]
    {
        IndexAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                BaseRate = 199.0,
                Description = "Best hotel in town",
                DescriptionFr = "Meilleur hôtel en ville",
                HotelName = "Fancy Stay",
                Category = "Luxury",
                Tags = new[] { "pool", "view", "wifi", "concierge" },
                ParkingIncluded = false,
                SmokingAllowed = false,
                LastRenovationDate = new DateTimeOffset(2010, 6, 27, 0, 0,
0, TimeSpan.Zero),
                Rating = 5,
                Location = GeographyPoint.Create(47.678581, -122.131577)
            }),
        IndexAction.Upload(
            new Hotel()
            {
                HotelId = "2",
                BaseRate = 79.99,
                Description = "Cheapest hotel in town",
                DescriptionFr = "Hôtel le moins cher en ville",
                HotelName = "Roach Motel",
                Category = "Budget",
                Tags = new[] { "motel", "budget" },
```

```

        ParkingIncluded = true,
        SmokingAllowed = true,
        LastRenovationDate = new DateTimeOffset(1982, 4, 28, 0, 0,
0, TimeSpan.Zero),
        Rating = 1,
        Location = GeographyPoint.Create(49.678581, -122.131577)
    )),
    IndexAction.MergeOrUpload(
        new Hotel()
        {
            HotelId = "3",
            BaseRate = 129.99,
            Description = "Close to town hall and the river"
        }
    )),
    IndexAction.Delete(new Hotel() { HotelId = "6" })
};

var batch = IndexBatch.New(actions);

```

In this case, we are using `Upload`, `MergeOrUpload`, and `Delete` as our search actions, as specified by the methods called on the `IndexAction` class.

Assume that this example “hotels” index is already populated with a number of documents. Note how we did not have to specify all the possible document fields when using `MergeOrUpload` and how we only specified the document key (`HotelId`) when using `Delete`. Also, note that you can only include up to 1000 documents in a single indexing request.

Note: In this example, we are applying different actions to different documents. If you wanted to perform the same actions across all documents in the batch, instead of calling `IndexBatch.New`, you could use the other static methods of `IndexBatch`. For example, you could create batches by calling `IndexBatch.Merge`, `IndexBatch.MergeOrUpload`, or `IndexBatch.Delete`. These methods take a collection of documents (objects of type `Hotel` in this example) instead of `IndexAction` objects.

Step 3: Import data to the index

Now that you have an initialized `IndexBatch` object, you can send it to the index by calling `Documents.Index` on your `SearchIndexClient` object. The following example shows how to call `Index`, as well as some extra steps you will need to perform:

```

try
{
    indexClient.Documents.Index(batch);
}
catch (IndexBatchException e)
{
    // Sometimes when your Search service is under load, indexing will fail
    // for some of the documents in
    // the batch. Depending on your application, you can take compensating
    // actions like delaying and
    // retrying. For this simple demo, we just log the failed document keys
    // and continue.
    Console.WriteLine(

```

```

        "Failed to index some of the documents: {0}",
        String.Join(", ", e.IndexingResults.Where(r => !r.Succeeded).
Select(r => r.Key)));
    }

    Console.WriteLine("Waiting for documents to be indexed...\n");
    Thread.Sleep(2000);

```

Note the `try/catch` surrounding the call to the `Index` method. The catch block handles an important error case for indexing. If your Azure Search service fails to index some of the documents in the batch, an `IndexBatchException` is thrown by `Documents.Index`. This can happen if you are indexing documents while your service is under heavy load. **It is strongly recommended to explicitly handle this case in your code.** You can delay and then retry indexing the documents that failed, or you can log and continue like the sample does, or you can do something else depending on your application's data consistency requirements.

Finally, the code in the example above delays for two seconds. Indexing happens asynchronously in your Azure Search service, so the sample application needs to wait a short time to ensure that the documents are available for searching. Delays like this are typically only necessary in demos, tests, and sample applications.

How the .NET SDK handles documents

You may be wondering how the Azure Search .NET SDK is able to upload instances of a user-defined class like `Hotel` to the index. To help answer that question, let's look at the `Hotel` class we created previously:

```

[SerializePropertyNameAsCamelCase]
public partial class Hotel
{
    [Key]
    [IsFilterable]
    public string HotelId { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public double? BaseRate { get; set; }

    [IsSearchable]
    public string Description { get; set; }

    [IsSearchable]
    [Analyzer(AnalyzerName.AsString.FrLucene)]
    [JsonProperty("description_fr")]
    public string DescriptionFr { get; set; }

    [IsSearchable, IsFilterable, IsSortable]
    public string HotelName { get; set; }

    [IsSearchable, IsFilterable, IsSortable, IsFacetable]
    public string Category { get; set; }

    [IsSearchable, IsFilterable, IsFacetable]
    public string[] Tags { get; set; }

```

```

[IsFilterable, IsFacetable]
public bool? ParkingIncluded { get; set; }

[IsFilterable, IsFacetable]
public bool? SmokingAllowed { get; set; }

[IsFilterable, IsSortable, IsFacetable]
public DateTimeOffset? LastRenovationDate { get; set; }

[IsFilterable, IsSortable, IsFacetable]
public int? Rating { get; set; }

[IsFilterable, IsSortable]
public GeographyPoint Location { get; set; }

// ToString() method omitted for brevity...
}

```

The first thing to notice is that each public property of `Hotel` corresponds to a field in the index definition, but with one crucial difference: The name of each field starts with a lower-case letter ("camel case"), while the name of each public property of `Hotel` starts with an upper-case letter ("Pascal case"). This is a common scenario in .NET applications that perform data-binding where the target schema is outside the control of the application developer. Rather than having to violate the .NET naming guidelines by making property names camel-case, you can tell the SDK to map the property names to camel-case automatically with the `[SerializePropertyNamesAsCamelCase]` attribute.

The second important thing about the `Hotel` class are the data types of the public properties. The .NET types of these properties map to their equivalent field types in the index definition. For example, the `Category` string property maps to the `category` field, which is of type `DataType.String`. There are similar type mappings between `bool?` and `DataType.Boolean`, `DateTimeOffset?` and `DataType.DateTimeOffset`, and so forth. The specific rules for the type mapping are documented with the `Documents.Get` method in the **Azure Search .NET SDK reference**⁵.

Why you should use nullable data types

When designing your own model classes to map to an Azure Search index, we recommend declaring properties of value types such as `bool` and `int` to be nullable (for example, `bool?` instead of `bool`). If you use a non-nullable property, you have to **guarantee** that no documents in your index contain a null value for the corresponding field. Neither the SDK nor the Azure Search service will help you to enforce this.

This is not just a hypothetical concern: Imagine a scenario where you add a new field to an existing index that is of type `DataType.Int32`. After updating the index definition, all documents will have a null value for that new field (since all types are nullable in Azure Search). If you then use a model class with a non-nullable `int` property for that field, you will get a `JsonSerializationException` like this when trying to retrieve documents:

```
Error converting value {null} to type 'System.Int32'. Path 'IntValue'.
```

⁵ <https://docs.microsoft.com/dotnet/api/microsoft.azure.search.documents.operation.extensions.get>

For this reason, we recommend that you use nullable types in your model classes as a best practice.

Next steps

After populating your Azure Search index, you will be ready to start issuing queries to search for documents.

Query an Azure Search index using the .NET SDK

Now that you have created an Azure Search index, you are almost ready to issue queries using the .NET SDK. First, you will need to obtain one of the query api-keys that was generated for the search service you provisioned. The .NET SDK will send this api-key on every request to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Step 1: Identify your Azure Search service's query api-key

1. To find your service's api-keys you can sign in to the Azure portal
2. Go to your Azure Search service's blade
3. Click on the "Keys" icon

Your service will have *admin keys* and *query keys*.

- Your primary and secondary *admin keys* grant full rights to all operations, including the ability to manage the service, create and delete indexes, indexers, and data sources. There are two keys so that you can continue to use the secondary key if you decide to regenerate the primary key, and vice-versa.
- Your *query keys* grant read-only access to indexes and documents, and are typically distributed to client applications that issue search requests.

For the purposes of querying an index, you can use one of your query keys. Your admin keys can also be used for queries, but you should use a query key in your application code as this better follows the Principle of least privilege.

Step 2: Create an instance of the `SearchIndexClient` class

To issue queries with the Azure Search .NET SDK, you will need to create an instance of the `SearchIndexClient` class. This class has several constructors. The one you want takes your search service name, index name, and a `SearchCredentials` object as parameters. `SearchCredentials` wraps your api-key.

The code below creates a new `SearchIndexClient` for the "hotels" index we created earlier using values for the search service name and api-key that are stored in the application's config file (`appsettings.json` in the case of the sample application):

```
private static SearchIndexClient CreateSearchIndexClient(IConfigurationRoot configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string queryApiKey = configuration["SearchServiceQueryApiKey"];

    SearchIndexClient indexClient = new SearchIndexClient(searchService-
```

```
Name, "hotels", new SearchCredentials(queryApiKey));
    return indexClient;
}
```

`SearchIndexClient` has a `Documents` property. This property provides all the methods you need to query Azure Search indexes.

Step 3: Query your index

Searching with the .NET SDK is as simple as calling the `Documents.Search` method on your `SearchIndexClient`. This method takes a few parameters, including the search text, along with a `SearchParameters` object that can be used to further refine the query.

Types of Queries

The two main query types you will use are `search` and `filter`. A `search` query searches for one or more terms in all searchable fields in your index. A `filter` query evaluates a boolean expression over all filterable fields in an index. You can use searches and filters together or separately.

Both searches and filters are performed using the `Documents.Search` method. A search query can be passed in the `searchText` parameter, while a filter expression can be passed in the `Filter` property of the `SearchParameters` class. To filter without searching, just pass "*" for the `searchText` parameter. To search without filtering, just leave the `Filter` property unset, or do not pass in a `SearchParameters` instance at all.

Example Queries

The following sample code shows a few different ways to query the "hotels" index. Note that the documents returned with the search results are instances of the `Hotel` class. The sample code makes use of a `WriteDocuments` method to output the search results to the console.

```
SearchParameters parameters;
DocumentSearchResult<Hotel> results;

Console.WriteLine("Search the entire index for the term 'budget' and return
only the hotelName field:\n");

parameters =
    new SearchParameters()
    {
        Select = new[] { "hotelName" }
    };

results = indexClient.Documents.Search<Hotel>("budget", parameters);

WriteDocuments(results);

Console.Write("Apply a filter to the index to find hotels cheaper than $150
per night, ");
Console.WriteLine("and return the hotelId and description:\n");
```

```

parameters =
    new SearchParameters()
    {
        Filter = "baseRate lt 150",
        Select = new[] { "hotelId", "description" }
    };

results = indexClient.Documents.Search<Hotel>>("*", parameters);

WriteDocuments(results);

Console.WriteLine("Search the entire index, order by a specific field (lastRenovationDate) ");
Console.WriteLine("in descending order, take the top two results, and show only hotelName and ");
Console.WriteLine("lastRenovationDate:\n");

parameters =
    new SearchParameters()
    {
        OrderBy = new[] { "lastRenovationDate desc" },
        Select = new[] { "hotelName", "lastRenovationDate" },
        Top = 2
    };

results = indexClient.Documents.Search<Hotel>>("*", parameters);

WriteDocuments(results);

Console.WriteLine("Search the entire index for the term 'motel':\n");

parameters = new SearchParameters();
results = indexClient.Documents.Search<Hotel>("motel", parameters);

WriteDocuments(results);

```

Step 4: Handle search results

The `Documents.Search` method returns a `DocumentSearchResult` object that contains the results of the query. The example in the previous section used a method called `WriteDocuments` to output the search results to the console:

```

private static void WriteDocuments(DocumentSearchResult<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.Results)
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}

```

```
}
```

Here is what the results look like for the queries in the previous section, assuming the "hotels" index is populated with the sample data from earlier in this module:

Search the entire index for the term 'budget' and return only the hotelName field:

```
Name: Roach Motel
```

Apply a filter to the index to find hotels cheaper than \$150 per night, and return the hotelId and description:

```
ID: 2   Description: Cheapest hotel in town
ID: 3   Description: Close to town hall and the river
```

Search the entire index, order by a specific field (lastRenovationDate) in descending order, take the top two results, and show only hotelName and lastRenovationDate:

```
Name: Fancy Stay           Last renovated on: 6/27/2010 12:00:00 AM +00:00
Name: Roach Motel          Last renovated on: 4/28/1982 12:00:00 AM +00:00
```

Search the entire index for the term 'motel':

```
ID: 2   Base rate: 79.99           Description: Cheapest hotel in town   De-
description (French): Hôtel le moins cher en ville   Name: Roach Motel
Category: Budget           Tags: [motel, budget]   Parking included: yes
Smoking allowed: yes       Last renovated on: 4/28/1982 12:00:00 AM +00:00
Rating: 1/5               Location: Latitude 49.678581, longitude -122.131577
```

The sample code above uses the console to output search results. You will likewise need to display search results in your own application.

Full text search in Azure Search

How full text search works in Azure Search

This lesson is for developers who need a deeper understanding of how Lucene full text search works in Azure Search. For text queries, Azure Search will seamlessly deliver expected results in most scenarios, but occasionally you might get a result that seems “off” somehow. In these situations, having a background in the four stages of Lucene query execution (query parsing, lexical analysis, document matching, scoring) can help you identify specific changes to query parameters or index configuration that will deliver the desired outcome.

Azure Search uses Lucene for full text search, but Lucene integration is not exhaustive. We selectively expose and extend Lucene functionality to enable the scenarios important to Azure Search.

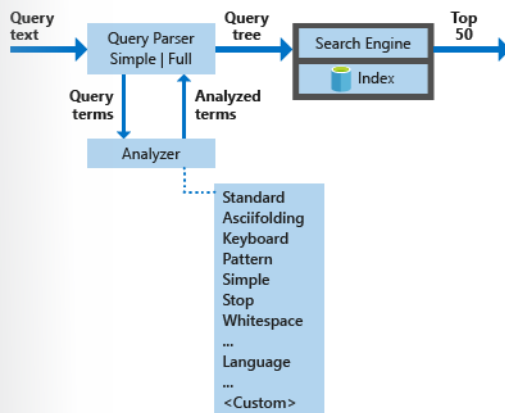
Architecture overview and diagram

Processing a full text search query starts with parsing the query text to extract search terms. The search engine uses an index to retrieve documents with matching terms. Individual query terms are sometimes broken down and reconstituted into new forms to cast a broader net over what could be considered as a potential match. A result set is then sorted by a relevance score assigned to each individual matching document. Those at the top of the ranked list are returned to the calling application.

Restated, query execution has four stages:

1. Query parsing
2. Lexical analysis
3. Document retrieval
4. Scoring

The diagram below illustrates the components used to process a search request.



Key components	Functional description
Query parsers	Separate query terms from query operators and create the query structure (a query tree) to be sent to the search engine.
Analyzers	Perform lexical analysis on query terms. This process can involve transforming, removing, or expanding of query terms.

Key components	Functional description
Index	An efficient data structure used to store and organize searchable terms extracted from indexed documents.
Search engine	Retrieves and scores matching documents based on the contents of the inverted index.

Anatomy of a search request

A search request is a complete specification of what should be returned in a result set. In simplest form, it is an empty query with no criteria of any kind. A more realistic example includes parameters, several query terms, perhaps scoped to certain fields, with possibly a filter expression and ordering rules.

The following example is a search request you might send to Azure Search using the REST API.

```
POST /indexes/hotels/docs/search?api-version=2017-11-11
{
  "search": "Spacious, air-condition* +\"Ocean view\"",
  "searchFields": "description, title",
  "searchMode": "any",
  "filter": "price ge 60 and price lt 300",
  "orderby": "geo.distance(location, geography'POINT(-159.476235
22.227659)')",
  "queryType": "full"
}
```

For this request, the search engine does the following:

1. Filters out documents where the price is at least \$60 and less than \$300.
2. Executes the query. In this example, the search query consists of phrases and terms: "Spacious, air-condition* +\"Ocean view\"" (users typically don't enter punctuation, but including it in the example allows us to explain how analyzers handle it). For this query, the search engine scans the description and title fields specified in `searchFields` for documents that contain "Ocean view", and additionally on the term "spacious", or on terms that start with the prefix "air-condition". The `searchMode` parameter is used to match on any term (default) or all of them, for cases where a term is not explicitly required (+).
3. Orders the resulting set of hotels by proximity to a given geography location, and then returned to the calling application.

The majority of this lesson is about processing of the search query: "Spacious, air-condition* +\"Ocean view\"". Filtering and ordering are out of scope.

Stage 1: Query parsing

As noted, the query string is the first line of the request:

```
"search": "Spacious, air-condition* +\"Ocean view\"",
```

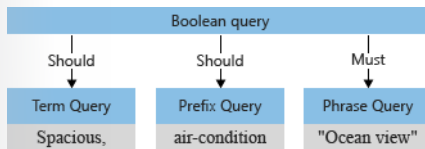
The query parser separates operators (such as * and + in the example) from search terms, and deconstructs the search query into subqueries of a supported type:

- *term query* for standalone terms (like spacious)

- *phrase query* for quoted terms (like ocean view)
- *prefix query* for terms followed by a prefix operator * (like air-condition)

Operators associated with a subquery determine whether the query “must be” or “should be” satisfied in order for a document to be considered a match. For example, `+“Ocean view”` is “must” due to the `+` operator.

The query parser restructures the subqueries into a *query tree* (an internal structure representing the query) it passes on to the search engine. In the first stage of query parsing, the query tree looks like this.



Supported parsers: Simple and Full Lucene

Azure Search exposes two different query languages, `simple` (default) and `full`. By setting the `queryType` parameter with your search request, you tell the query parser which query language you choose so that it knows how to interpret the operators and syntax. The Simple query language is intuitive and robust, often suitable to interpret user input as-is without client-side processing. It supports query operators familiar from web search engines. The Full Lucene query language, which you get by setting `queryType=full`, extends the default Simple query language by adding support for more operators and query types like wildcard, fuzzy, regex, and field-scoped queries. For example, a regular expression sent in Simple query syntax would be interpreted as a query string and not an expression. The example request in this article uses the Full Lucene query language.

Impact of searchMode on the parser

Another search request parameter that affects parsing is the `searchMode` parameter. It controls the default operator for Boolean queries: `any` (default) or `all`.

When `searchMode=any`, which is the default, the space delimiter between `spacious` and `air-condition` is `OR (||)`, making the sample query text equivalent to:

```
Spacious,||air-condition*+"Ocean view"
```

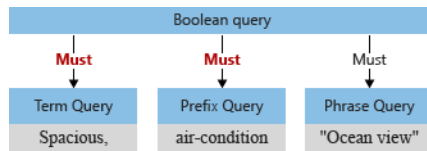
Explicit operators, such as `+` in `+“Ocean view”`, are unambiguous in boolean query construction (the term must match). Less obvious is how to interpret the remaining terms: `spacious` and `air-condition`. Should the search engine find matches on `ocean view` and `spacious` and `air-condition`? Or should it find `ocean view` plus either one of the remaining terms?

By default (`searchMode=any`), the search engine assumes the broader interpretation. Either field should be matched, reflecting “or” semantics. The initial query tree illustrated previously, with the two “should” operations, shows the default.

Suppose that we now set `searchMode=all`. In this case, the space is interpreted as an “and” operation. Each of the remaining terms must both be present in the document to qualify as a match. The resulting sample query would be interpreted as follows:

```
+Spacious,+air-condition*+"Ocean view"
```

A modified query tree for this query would be as follows, where a matching document is the intersection of all three subqueries:



Choosing `searchMode=any` over `searchMode=all` is a decision best arrived at by running representative queries. Users who are likely to include operators (common when searching document stores) might find results more intuitive if `searchMode=all` informs boolean query constructs.

Next

We'll cover lexical analysis and document retrieval in Azure Search.

Lexical analysis and document retrieval in Azure Search

Lexical analysis

Lexical analyzers process term queries and phrase queries after the query tree is structured. An analyzer accepts the text inputs given to it by the parser, processes the text, and then sends back tokenized terms to be incorporated into the query tree.

The most common form of lexical analysis is linguistic analysis which transforms query terms based on rules specific to a given language:

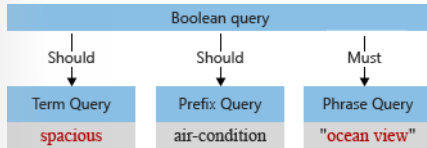
- Reducing a query term to the root form of a word
- Removing non-essential words (stopwords, such as "the" or "and" in English)
- Breaking a composite word into component parts
- Lower casing an upper case word

All of these operations tend to erase differences between the text input provided by the user and the terms stored in the index. Such operations go beyond text processing and require in-depth knowledge of the language itself. To add this layer of linguistic awareness, Azure Search supports a long list of language analyzers from both Lucene and Microsoft.

Note: Analysis requirements can range from minimal to elaborate depending on your scenario. You can control complexity of lexical analysis by selecting one of the predefined analyzers or by creating your own custom analyzer. Analyzers are scoped to searchable fields and are specified as part of a field definition. This allows you to vary lexical analysis on a per-field basis. Unspecified, the standard Lucene analyzer is used.

In our example, prior to analysis, the initial query tree has the term "Spacious," with an uppercase "S" and a comma that the query parser interprets as a part of the query term (a comma is not considered a query language operator).

When the default analyzer processes the term, it will lowercase "ocean view" and "spacious", and remove the comma character. The modified query tree will look as follows:



Testing analyzer behaviors

The behavior of an analyzer can be tested using the Analyze API. Provide the text you want to analyze to see what terms given analyzer will generate. For example, to see how the standard analyzer would process the text “air-condition”, you can issue the following request:

```
{
  "text": "air-condition",
  "analyzer": "standard"
}
```

The standard analyzer breaks the input text into the following two tokens, annotating them with attributes like start and end offsets (used for hit highlighting) as well as their position (used for phrase matching):

```
{
  "tokens": [
    {
      "token": "air",
      "startOffset": 0,
      "endOffset": 3,
      "position": 0
    },
    {
      "token": "condition",
      "startOffset": 4,
      "endOffset": 13,
      "position": 1
    }
  ]
}
```

Exceptions to lexical analysis

Lexical analysis applies only to query types that require complete terms – either a term query or a phrase query. It doesn’t apply to query types with incomplete terms – prefix query, wildcard query, regex query – or to a fuzzy query. Those query types, including the prefix query with term `air-condition*` in our example, are added directly to the query tree, bypassing the analysis stage. The only transformation performed on query terms of those types is lowercasing.

Document retrieval

Document retrieval refers to finding documents with matching terms in the index. This stage is understood best through an example. Let’s start with a hotels index having the following simple schema:

```
{
  "name": "hotels",
  "fields": [
    { "name": "id", "type": "Edm.String", "key": true, "searchable":
false },
    { "name": "title", "type": "Edm.String", "searchable": true },
    { "name": "description", "type": "Edm.String", "searchable": true }
  ]
}
```

Further assume that this index contains the following four documents:

```
{
  "value": [
    {
      "id": "1",
      "title": "Hotel Atman",
      "description": "Spacious rooms, ocean view, walking distance to
the beach."
    },
    {
      "id": "2",
      "title": "Beach Resort",
      "description": "Located on the north shore of the island of
Kaua'i. Ocean view."
    },
    {
      "id": "3",
      "title": "Playa Hotel",
      "description": "Comfortable, air-conditioned rooms with ocean
view."
    },
    {
      "id": "4",
      "title": "Ocean Retreat",
      "description": "Quiet and secluded"
    }
  ]
}
```

How terms are indexed

To understand retrieval, it helps to know a few basics about indexing. The unit of storage is an inverted index, one for each searchable field. Within an inverted index is a sorted list of all terms from all documents. Each term maps to the list of documents in which it occurs, as evident in the example below.

To produce the terms in an inverted index, the search engine performs lexical analysis over the content of documents, similar to what happens during query processing:

1. Text inputs are passed to an analyzer, lower-cased, stripped of punctuation, and so forth, depending on the analyzer configuration.

2. Tokens are the output of text analysis.
3. Terms are added to the index.

It's common, but not required, to use the same analyzers for search and indexing operations so that query terms look more like terms inside the index.

Inverted index for example documents

Returning to our example, for the **title** field, the inverted index looks like this:

Term	Document list
atman	1
beach	2
hotel	1, 3
ocean	4
playa	3
resort	3
retreat	4

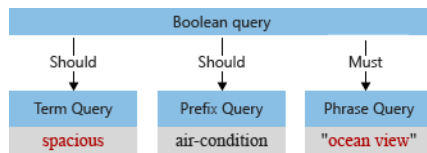
In the title field, only hotel shows up in two documents: 1, 3.

For the description field, the index is as follows:

Term	Document list
air	3
and	4
beach	1
conditioned	3
comfortable	3
distance	1
island	2
kaua'i	2
located	2
north	2
ocean	1, 2, 3
of	2
on	2
quiet	4
rooms	1, 3
secluded	4
shore	2
spacious	1
the	1, 2
to	1
view	1, 2, 3
walking	1
with	3

Matching query terms against indexed terms

Given the inverted indices above, let's return to the sample query and see how matching documents are found for our example query. Recall that the final query tree looks like this:



During query execution, individual queries are executed against the searchable fields independently.

- The TermQuery, "spacious", matches document 1 (Hotel Atman).
- The PrefixQuery, "air-condition*", doesn't match any documents.
- This is a behavior that sometimes confuses developers. Although the term air-conditioned exists in the document, it is split into two terms by the default analyzer. Recall that prefix queries, which contain partial terms, are not analyzed. Therefore terms with prefix "air-condition" are looked up in the inverted index and not found.
- The PhraseQuery, "ocean view", looks up the terms "ocean" and "view" and checks the proximity of terms in the original document. Documents 1, 2 and 3 match this query in the description field. Notice document 4 has the term ocean in the title but isn't considered a match, as we're looking for the "ocean view" phrase rather than individual words.

On the whole, for the query in question, the documents that match are 1, 2, 3.

Next

We'll cover document scoring and wrap up the topic.

Document scoring in Azure Search

Every document in a search result set is assigned a relevance score. The function of the relevance score is to rank higher those documents that best answer a user question as expressed by the search query. The score is computed based on statistical properties of terms that matched. At the core of the scoring formula is TF/IDF (term frequency-inverse document frequency). In queries containing rare and common terms, TF/IDF promotes results containing the rare term. For example, in a hypothetical index with all Wikipedia articles, from documents that matched the query the president, documents matching on president are considered more relevant than documents matching on the.

Scoring example

Recall the three documents that matched our example query:

```

search=Spacious, air-condition* +"Ocean view"

{
  "value": [
    {
      "@search.score": 0.25610128,
      "id": "1",
      "title": "Hotel Atman",
    }
  ]
}

```

```
      "description": "Spacious rooms, ocean view, walking distance to the beach."
    },
    {
      "@search.score": 0.08951007,
      "id": "3",
      "title": "Playa Hotel",
      "description": "Comfortable, air-conditioned rooms with ocean view."
    },
    {
      "@search.score": 0.05967338,
      "id": "2",
      "title": "Ocean Resort",
      "description": "Located on a cliff on the north shore of the island of Kauai. Ocean view."
    }
  ]
}
```

Document 1 matched the query best because both the term spacious and the required phrase ocean view occur in the description field. The next two documents match only the phrase ocean view. It might be surprising that the relevance score for document 2 and 3 is different even though they matched the query in the same way. It's because the scoring formula has more components than just TF/IDF. In this case, document 3 was assigned a slightly higher score because its description is shorter. Learn about Lucene's Practical Scoring Formula to understand how field length and other factors can influence the relevance score.

Some query types (wildcard, prefix, regex) always contribute a constant score to the overall document score. This allows matches found through query expansion to be included in the results, but without affecting the ranking.

An example illustrates why this matters. Wildcard searches, including prefix searches, are ambiguous by definition because the input is a partial string with potential matches on a very large number of disparate terms (consider an input of "tour*", with matches found on "tours", "tourettes", and "tourmaline"). Given the nature of these results, there is no way to reasonably infer which terms are more valuable than others. For this reason, we ignore term frequencies when scoring results in queries of types wildcard, prefix and regex. In a multi-part search request that includes partial and complete terms, results from the partial input are incorporated with a constant score to avoid bias towards potentially unexpected matches.

Score tuning

There are two ways to tune relevance scores in Azure Search:

1. **Scoring profiles** promote documents in the ranked list of results based on a set of rules. In our example, we could consider documents that matched in the title field more relevant than documents that matched in the description field. Additionally, if our index had a price field for each hotel, we could promote documents with lower price.
2. **Term boosting** (available only in the Full Lucene query syntax) provides a boosting operator `^` that can be applied to any part of the query tree. In our example, instead of searching on the prefix `air-condition*`, one could search for either the exact term `air-condition` or the prefix, but documents that match on the exact term are ranked higher by applying boost to the term query:
`air-condition^2||air-condition*`.

Scoring in a distributed index

All indexes in Azure Search are automatically split into multiple shards, allowing us to quickly distribute the index among multiple nodes during service scale up or scale down. When a search request is issued, it's issued against each shard independently. The results from each shard are then merged and ordered by score (if no other ordering is defined). It is important to know that the scoring function weights query term frequency against its inverse document frequency in all documents within the shard, not across all shards!

This means a relevance score could be different for identical documents if they reside on different shards. Fortunately, such differences tend to disappear as the number of documents in the index grows due to more even term distribution. It's not possible to assume on which shard any given document will be placed. However, assuming a document key doesn't change, it will always be assigned to the same shard.

In general, document score is not the best attribute for ordering documents if order stability is important. For example, given two documents with an identical score, there is no guarantee which one appears first in subsequent runs of the same query. Document score should only give a general sense of document relevance relative to other documents in the results set.

Wrap-up

The success of internet search engines has raised expectations for full text search over private data. For almost any kind of search experience, we now expect the engine to understand our intent, even when terms are misspelled or incomplete. We might even expect matches based on near equivalent terms or synonyms that we never actually specified.

From a technical standpoint, full text search is highly complex, requiring sophisticated linguistic analysis and a systematic approach to processing in ways that distill, expand, and transform query terms to deliver a relevant result. Given the inherent complexities, there are a lot of factors that can affect the outcome of a query. For this reason, investing the time to understand the mechanics of full text search offers tangible benefits when trying to work through unexpected results.

This article explored full text search in the context of Azure Search. We hope it gives you sufficient background to recognize potential causes and resolutions for addressing common query problems.

Review questions

Module 2 review questions

Azure Search index

Before you can perform your first search, you must first create an index in Azure Search. An index means several unique things within Azure Search, what are they?

> Click to see suggested answer

- An index is the scope used for queries over documents within Azure Search.
- Documents are uploaded, updated, and managed within the context of an index.
- Indexes inform the Azure Search engine about the properties (or fields) that are available in the indexes' documents and the capabilities that are appropriate for each document property.

Create an index using code

A single call to the `Indexes.Create` method will create your index. This method takes as a parameter an `Index` object that defines your Azure Search index. You need to create an `Index` object and initialize it as follows:

- Set the `Name` property of the `Index` object to the name of your index.
- Set the `Fields` property of the `Index` object to an array of `Field` objects.

What method makes it easy to create the `Field` objects?

> Click to see suggested answer

The easiest way to create the `Field` objects is by calling the `FieldBuilder.BuildForType` method, passing a model class for the type parameter. A model class has properties that map to the fields of your index. This allows you to bind documents from your search index to instances of your model class:

```
var definition = new Index()
{
    Name = "exampleindex",
    Fields = FieldBuilder.BuildForType<Document>()
};

serviceClient.Indexes.Create(definition);
```

Azure Search indexer

An indexer in Azure Search is a crawler that extracts searchable data and metadata from an external Azure data source and populates an index based on field-to-field mappings between the index and your data source. Indexers can offer features that are unique to the data source. In this respect, some aspects of indexer or data source configuration will vary by indexer type. However, all indexers share the same basic composition and requirements. What are the steps that are common to all indexers?

> Click to see suggested answer

1. **Create a data source:** An indexer pulls data from a data source that holds information, such as a connection string and possibly credentials. Data sources are configured and managed independently of the indexers that use them, which means that a data source can be used by multiple indexers to load more than one index at a time.
2. **Create an index:** An indexer will automate some tasks related to data ingestion, but creating an index is generally not one of them. As a prerequisite, you must have a predefined index with fields that match those in your external data source.
3. **Create and schedule the indexer:** The indexer definition is a construct specifying the index, data source, and schedule. An indexer can reference a data source from another service as long as that data source is from the same subscription.



Module 3 API Management

Introduction to the API Management service

API Management overview

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

Overview

To use API Management, administrators create APIs. Each API consists of one or more operations, and each API can be added to one or more products. To use an API, developers subscribe to a product that contains that API, and then they can call the API's operation, subject to any usage policies that may be in effect. 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 **Azure portal** is the administrative interface where you set up your API program. Use it to:
 - Define or import API schema.
 - Package APIs into products.
 - Set up policies like quotas or transformations on the APIs.
 - Get insights from analytics.
 - Manage users.
- 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.

APIs and operations

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

Products

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

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.

Groups

Groups are used to manage the visibility of products to developers. 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.

Developers

Developers represent the user accounts in an API Management service instance. Developers can be created or invited to join by administrators, or they can sign up from the Developer portal. Each developer is a member of one or more groups, and can subscribe to the products that grant visibility to those groups.

When developers subscribe to a product, they are granted the primary and secondary key for the product. This key is used when making calls into the product's APIs.

Policies

Policies are a powerful capability of API Management that allow the Azure portal 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 number of incoming calls from a developer, and many other policies are available.

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](#).

Developer portal

The developer portal is where developers can learn about your APIs, view and call operations, and subscribe to products. Prospective customers can visit the developer portal, view APIs and operations, and sign up. The URL for your developer portal is located on the dashboard in the Azure portal for your API Management service instance.

Next

We'll walk you through creating an API Management instance.

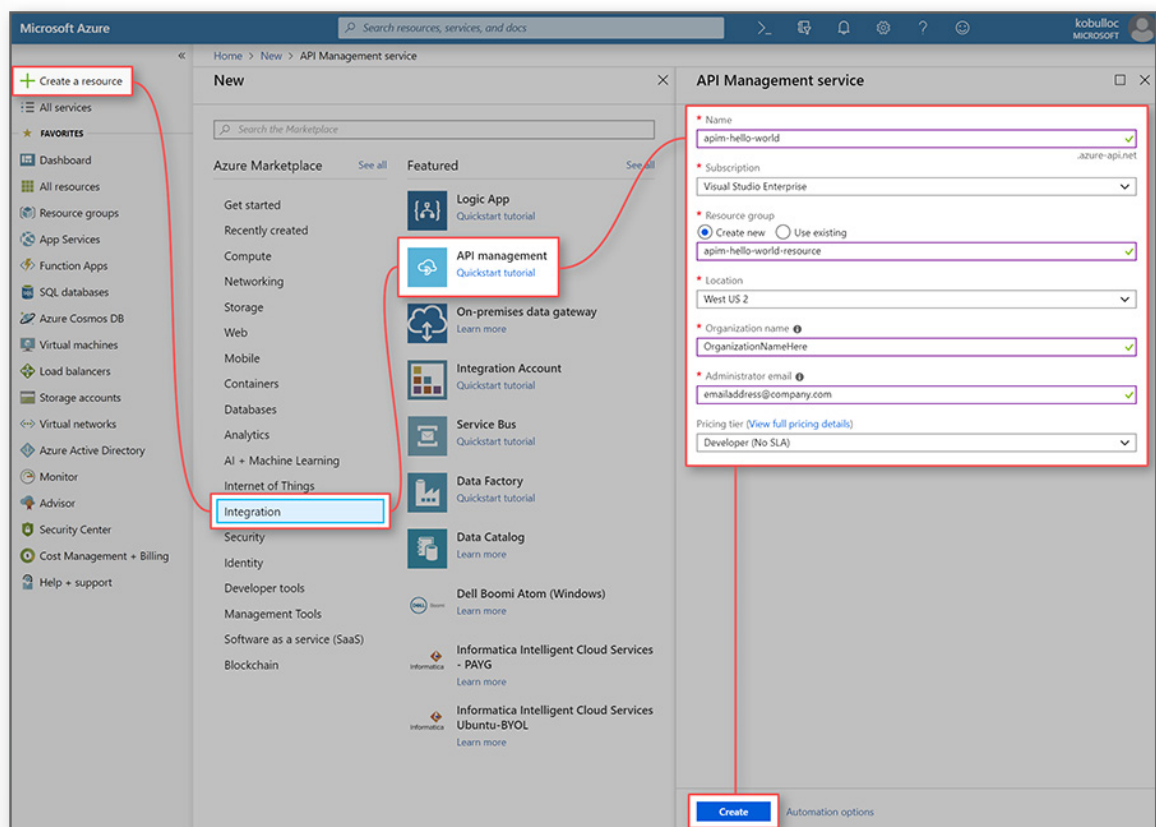
Create a new Azure API Management service instance by using the Azure Portal

Azure API Management (APIM) helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services. API Management provides the core competencies to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. APIM enables you to create and manage modern API gateways for existing backend services hosted anywhere.

Steps to create an APIM instance in the Azure Portal

It takes just a few steps to create a new APIM instance:

1. Log in to the Azure portal at <https://portal.azure.com>.
2. In the Azure portal, select **Create a resource** > **Integration** > **API management**.



- 3.
4. In the **API Management service** window, enter settings.

API Management service

*

Name

Enter the name

.azure-api.net

*

Subscription

Visual Studio Ultimate with MSDN

*

Resource group

Create new

Use existing

*

Location

West US

*

Organization name

Enter organization name

*

Administrator email

Enter administrator email

Pricing tier

[View full pricing details](#)

Developer (No SLA)

Create

Automation options

5.

Setting	Suggested value	Description
Name	A unique name for your API Management service	The name can't be changed later. Service name is used to generate a default domain name in the form of {name}.azure-api.net. Service name is used to refer to the service and the corresponding Azure resource.
Subscription	Your subscription	The subscription under which this new service instance will be created. You can select the subscription among the different Azure subscriptions that you have access to.
Resource Group	apimResourceGroup	You can select a new or existing resource. A resource group is a collection of resources that share lifecycle, permissions, and policies.

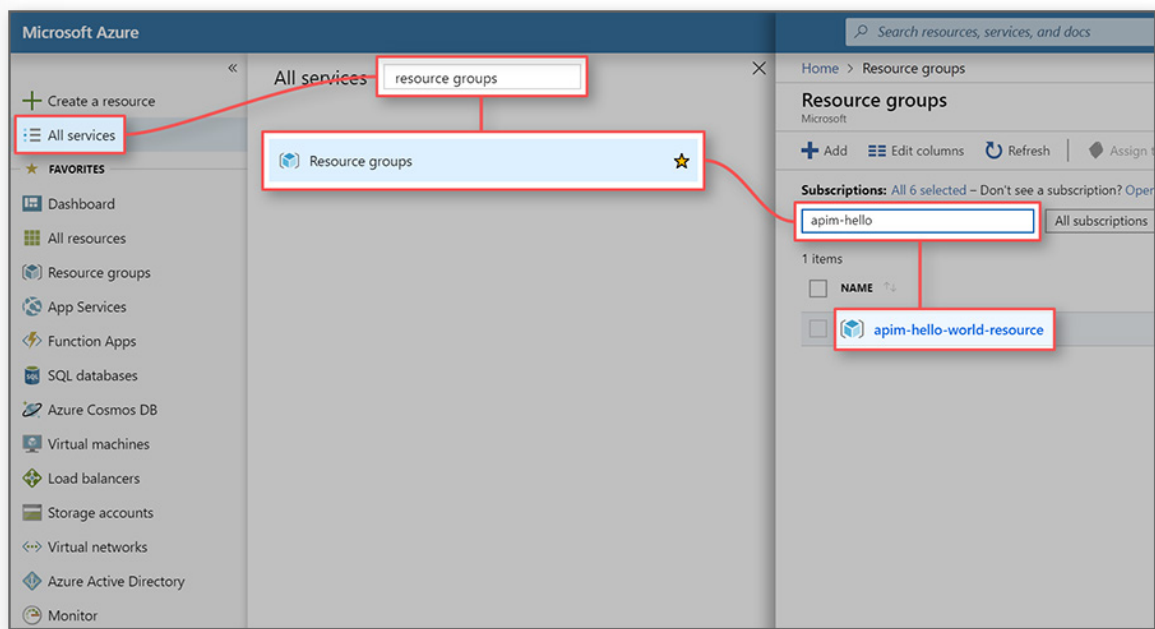
Setting	Suggested value	Description
Location	<i>West USA</i>	Select the geographic region near you. Only the available API Management service regions appear in the drop-down list box.
Organization name	The name of your organization	This name is used in a number of places, including the title of the developer portal and sender of notification emails.
Administrator email	<i>admin@org.com</i>	Set email address to which all the notifications from API Management will be sent.
Pricing tier	<i>Developer</i>	Set Developer tier to evaluate the service. Note: The Developer tier is not for production use.

- Choose **Create**. This is a long running operation and could take up to 15 minutes to complete. Selecting **Pin to dashboard** makes finding a newly created service easier.

Clean up resources

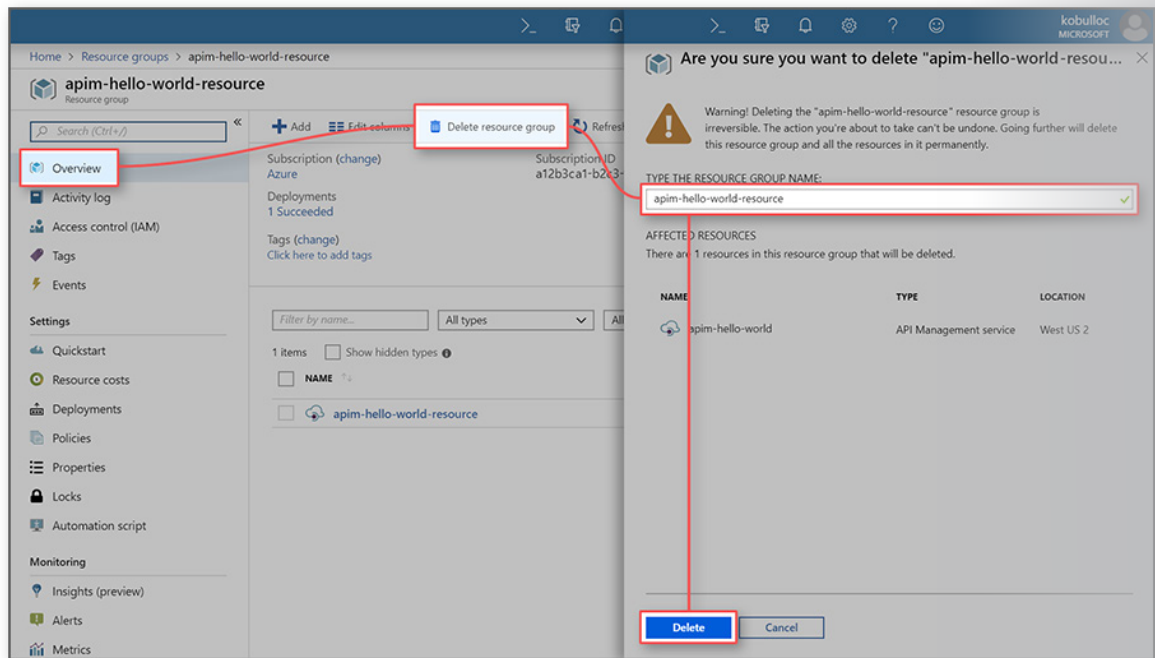
When no longer needed, you can remove the resource group and all related resources by following these steps:

- In the Azure portal, select **All services**.
- Input `resource groups` in the search box and click on the result.



-
-
-
- Find your resource group and click on it.

- Click **Delete resource group**.



-
- Confirm the deletion by inputting the name of your resource group.
- Click **Delete**.

Next

The next lesson will cover securing your API using the API Management service.

Securing your APIs

Understanding subscriptions in Azure API Management

Subscriptions are an important concept in Azure API Management. They're the most common way for API consumers to get access to APIs published through an API Management instance.

What are subscriptions?

When you publish APIs through API Management, it's easy and common to secure access to those APIs by using subscription keys. Developers who need to consume the published APIs must include a valid subscription key in HTTP requests when they make calls to those APIs. Otherwise, the calls are rejected immediately by the API Management gateway. They aren't forwarded to the back-end services.

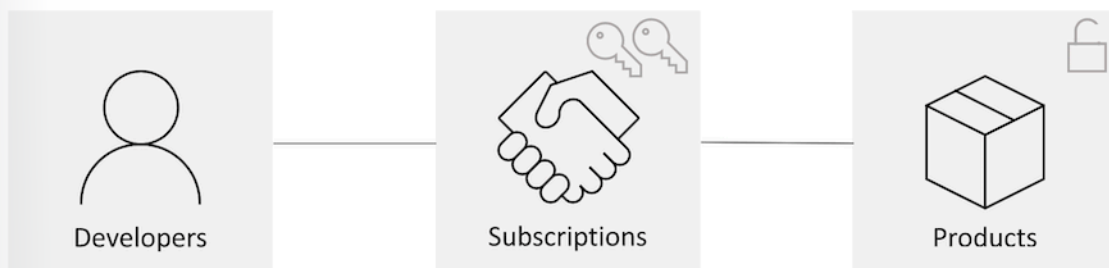
To get a subscription key for accessing APIs, a subscription is required. A subscription is essentially a named container for a pair of subscription keys. Developers who need to consume the published APIs can get subscriptions. And they don't need approval from API publishers. API publishers can also create subscriptions directly for API consumers.

Scope of subscriptions

Subscriptions can be associated with various scopes: product, all APIs, or an individual API.

Subscriptions for a product

Traditionally, subscriptions in API Management were always associated with a single API product scope. Developers found the list of products on the Developer Portal. Then they'd submit subscription requests for the products they wanted to use. After a subscription request is approved, either automatically or by API publishers, the developer can use the keys in it to access all APIs in the product.



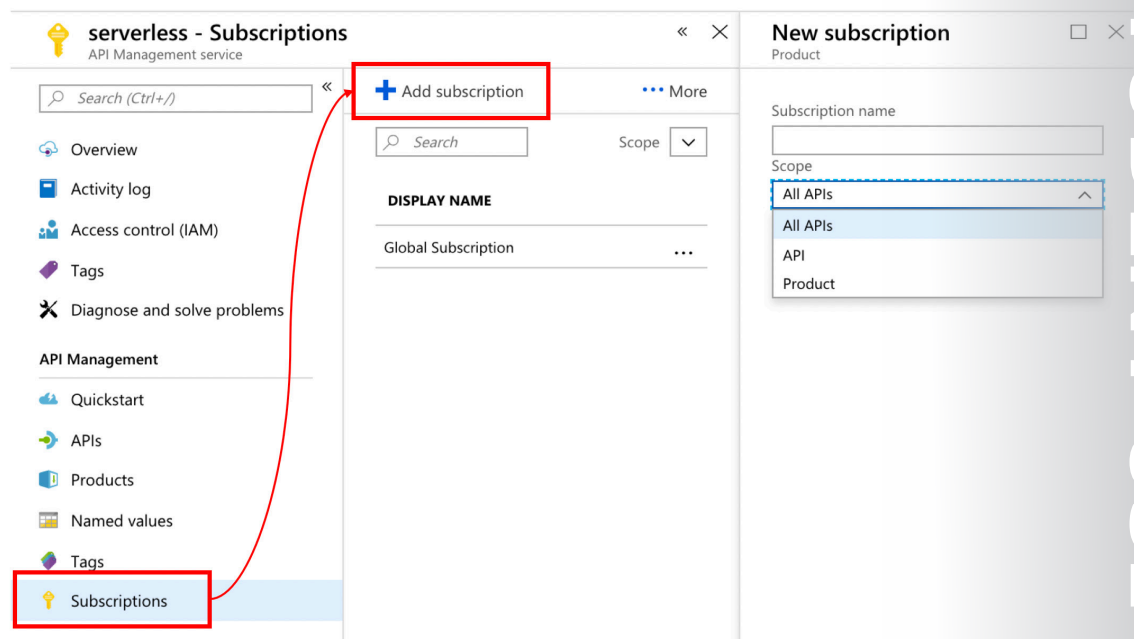
Tip: Under certain scenarios, API publishers might want to publish an API product to the public without the requirement of subscriptions. They can deselect the Require subscription option on the Settings page of the product in the Azure portal. As a result, all APIs under the product can be accessed without an API key.

Subscriptions for all APIs or an individual API

Note: Currently, this feature is available in the API Management Consumption tier only.

When we introduced the Consumption tier of API Management, we made a few changes to streamline key management:

- First, we added two more subscription scopes: all APIs and a single API. The scope of subscriptions is no longer limited to an API product. It's now possible to create keys that grant access to an API, or all APIs within an API Management instance, without needing to create a product and add the APIs to it first. Also, each API Management instance now comes with an immutable, all-APIs subscription. This subscription makes it easier and more straightforward to test and debug APIs within the test console.
- Second, API Management now allows standalone subscriptions. Subscriptions are no longer required to be associated with a developer account. This feature is useful in scenarios such as when several developers or teams share a subscription.
- Finally, API publishers can now create subscriptions directly in the Azure portal:
 - Select **Subscriptions** in the menu on the left.
 - Select **Add subscription**.
 - Provide a name of the subscription and select the scope.
 - Select **Save**.



Next

API Management also supports other mechanisms (client certificates and OAuth2.0) for securing access to APIs, and we'll cover those next in this lesson.

Securing APIs using client certificate authentication in API Management

API Management provides the capability to secure access to APIs (i.e., client to API Management) using client certificates. Currently, you can check the thumbprint of a client certificate against a desired value. You can also check the thumbprint against existing certificates uploaded to API Management.

This feature is available in the **Premium**, **Standard**, **Basic** and **Developer** tiers of API Management.

Checking the expiration date

Below policies can be configured to check if the certificate is expired:

```
<choose>
  <when condition="@ (context.Request.Certificate == null || context.
Request.Certificate.NotAfter < DateTime.Now) " >
    <return-response>
      <set-status code="403" reason="Invalid client certificate" />
    </return-response>
  </when>
</choose>
```

Checking the issuer and subject

Below policies can be configured to check the issuer and subject of a client certificate:

```
<choose>
  <when condition="@ (context.Request.Certificate == null || context.
Request.Certificate.Issuer != "trusted-issuer" || context.Request.Certifi-
cate.SubjectName != "expected-subject-name") " >
    <return-response>
      <set-status code="403" reason="Invalid client certificate" />
    </return-response>
  </when>
</choose>
```

Checking the thumbprint

Below policies can be configured to check the thumbprint of a client certificate:

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

Checking a thumbprint against certificates uploaded to API Management

The following example shows how to check the thumbprint of a client certificate against certificates uploaded to API Management:

```
<choose>
  <when condition="@ (context.Request.Certificate == null || !context.
Deployment.Certificates.Any(c => c.Value.Thumbprint == context.Request.
Certificate.Thumbprint)) " >
    <return-response>
      <set-status code="403" reason="Invalid client certificate" />
    </return-response>
  </when>
</choose>
```

Next

We'll cover protecting an API by using OAuth2.0.

Protect an API by using OAuth 2.0 with Azure Active Directory and API Management

Now we'll show you how to configure your Azure API Management instance to protect an API by using the OAuth 2.0 protocol with Azure Active Directory (Azure AD).

Overview

Here is a quick overview of the steps:

1. Register an application (backend-app) in Azure AD to represent the API.
2. Register another application (client-app) in Azure AD to represent a client application that needs to call the API.
3. In Azure AD, grant permissions to allow the client-app to call the backend-app.
4. Configure the Developer Console to use OAuth 2.0 user authorization.
5. Add the **validate-jwt** policy to validate the OAuth token for every incoming request.

Step 1: Register an application in Azure AD to represent the API

To protect an API with Azure AD, the first step is to register an application in Azure AD that represents the API.

1. Browse to your Azure AD tenant, and then browse to **App registrations**.
2. Select **New application registration**.
3. Provide a name of the application. (For this example, the name is `backend-app`.)
4. Choose **Web app / API** as the **Application type**.

5. For **Sign-on URL**, you can use `https://localhost` as a placeholder.
6. Select **Create**.

When the application is created, make a note of the **Application ID**, for use in a subsequent step.

Step 2: Register another application in Azure AD to represent a client application

Every client application that calls the API needs to be registered as an application in Azure AD as well. For this example, the sample client application is the Developer Console in the API Management developer portal. Here's how to register another application in Azure AD to represent the Developer Console.

1. Select **New application registration**.
2. Provide a name of the application. (For this example, the name is `client-app`.)
3. Choose **Web app / API** as the **Application type**.
4. For **Sign-on URL**, you can use `https://localhost` as a placeholder, or use the sign-in URL of your API Management instance. (For this example, the URL is `https://contoso5.portal.azure-api.net/signin`.)
5. Select **Create**.

When the application is created, make a note of the **Application ID**, for use in a subsequent step. Now, create a client secret for this application, for use in a subsequent step.

1. Select **Settings** again, and go to **Keys**.
2. Under **Passwords**, provide a **Key description**. Choose when the key should expire, and select **Save**.

Make a note of the key value.

Step 3: Grant permissions in Azure AD

Now that you have registered two applications to represent the API and the Developer Console, you need to grant permissions to allow the client-app to call the backend-app.

1. Browse to **Application registrations**.
2. Select `client-app`, and go to **Settings**.
3. Select **Required permissions > Add**.
4. Select **Select an API**, and search for `backend-app`.
5. Under **Delegated Permissions**, select `Access backend-app`.
6. Select **Select**, and then select **Done**.

Note: If Azure Active Directory is not listed under permissions to other applications, select **Add** to add it from the list.

Step 4: Enable OAuth 2.0 user authorization in the Developer Console

At this point, you have created your applications in Azure AD, and have granted proper permissions to allow the client-app to call the backend-app.

In this example, the Developer Console is the client-app. The following steps describe how to enable OAuth 2.0 user authorization in the Developer Console.

1. In Azure Portal, browse to your API Management instance.
2. Select **OAuth 2.0 > Add**.
3. Provide a **Display name** and **Description**.
4. For the Client registration page URL, enter a placeholder value, such as `http://localhost`. 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 this. In this example, users do not create and configure their own accounts, so you use a placeholder instead.
5. For **Authorization grant types**, select **Authorization code**.
6. Specify the **Authorization endpoint URL** and **Token endpoint URL**. Retrieve these values from the **Endpoints** page in your Azure AD tenant. Browse to the **App registrations** page again, and select **Endpoints**. Note:** Use the v1 endpoints here.**
7. Copy the **OAuth 2.0 Authorization Endpoint**, and paste it into the **Authorization endpoint URL** text box.
8. Copy the **OAuth 2.0 Token Endpoint**, and paste it into the **Token endpoint URL** text box. In addition to pasting in the token endpoint, add a body parameter named **resource**. For the value of this parameter, use the **Application ID** for the back-end app.
9. Next, specify the client credentials. These are the credentials for the client-app.
10. For **Client ID**, use the **Application ID** for the client-app.
11. For **Client secret**, use the key you created for the client-app earlier.
12. Immediately following the client secret is the **redirect_url** for the authorization code grant type. Make a note of this URL.
13. Select **Create**.
14. Go back to the **Settings** page of your client-app.
15. Select **Reply URLs**, and paste the **redirect_url** in the first row. In this example, you replaced `https://localhost` with the URL in the first row.

Now that you have configured an OAuth 2.0 authorization server, the Developer Console can obtain access tokens from Azure AD.

The next step is to enable OAuth 2.0 user authorization for your API. This enables the Developer Console to know that it needs to obtain an access token on behalf of the user, before making calls to your API.

1. Browse to your API Management instance, and go to **APIs**.
2. Select the API you want to protect. In this example, you use the `Echo API`.
3. Go to **Settings**.
4. Under **Security**, choose **OAuth 2.0**, and select the OAuth 2.0 server you configured earlier.
5. Select **Save**.

Step 5: Configure a JWT validation policy to pre-authorize requests

At this point, when a user tries to make a call from the Developer Console, the user is prompted to sign in. The Developer Console obtains an access token on behalf of the user.

But what if someone calls your API without a token or with an invalid token? For example, you can still call the API even if you delete the `Authorization` header. The reason is that API Management does not validate the access token at this point. It simply passes the `Authorization` header to the back-end API.

You can use the Validate JWT policy to pre-authorize requests in API Management, by validating the access tokens of each incoming request. If a request does not have a valid token, API Management blocks it. For example, you can add the following policy to the `<inbound>` policy section of the `Echo` API. It checks the audience claim in an access token, and returns an error message if the token is not valid. For information on how to configure policies, see [Set or edit policies](#).

```
<validate-jwt header-name="Authorization" failed-validation-httpcode="401"
failed-validation-error-message="Unauthorized. Access token is missing or
invalid.">
  <openid-config url="https://login.microsoftonline.com/{aad-tenant}/.
well-known/openid-configuration" />
  <required-claims>
    <claim name="aud">
      <value>{Application ID of backend-app}</value>
    </claim>
  </required-claims>
</validate-jwt>
```

Next

In the next lesson we're going to cover defining policies for APIs.

Defining API policies

Policies in Azure API Management overview

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

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.

Understanding policy configuration

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.

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.

The configuration is divided into `inbound`, `backend`, `outbound`, and `on-error`. The series of specified policy statements is executed in order for a request and a response.

```
<policies>
  <inbound>
    <!-- statements to be applied to the request go here -->
  </inbound>
  <backend>
    <!-- statements to be applied before the request is forwarded to
        the backend service go here -->
  </backend>
  <outbound>
    <!-- statements to be applied to the response go here -->
  </outbound>
  <on-error>
    <!-- statements to be applied if there is an error condition go here
-->
  </on-error>
</policies>
```

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.

Examples

Apply policies specified at different scopes

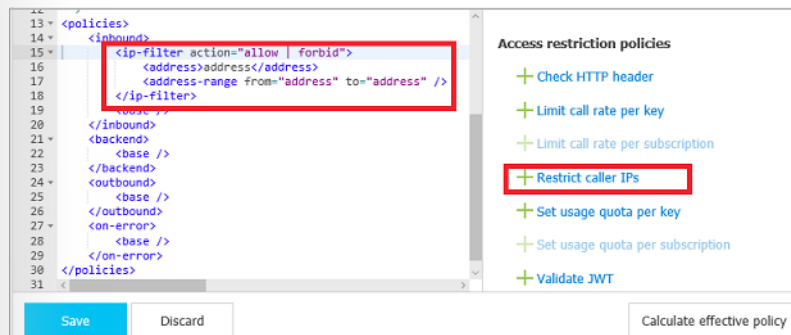
If you have a policy at the global level and a policy configured for an API, then whenever that particular API is used both policies will be applied. API Management allows for deterministic ordering of combined policy statements via the base element.

```
<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.

Restrict incoming requests

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>
```

Next

- Now that you have a basic understanding of policies we'll cover how to set, and edit, them.

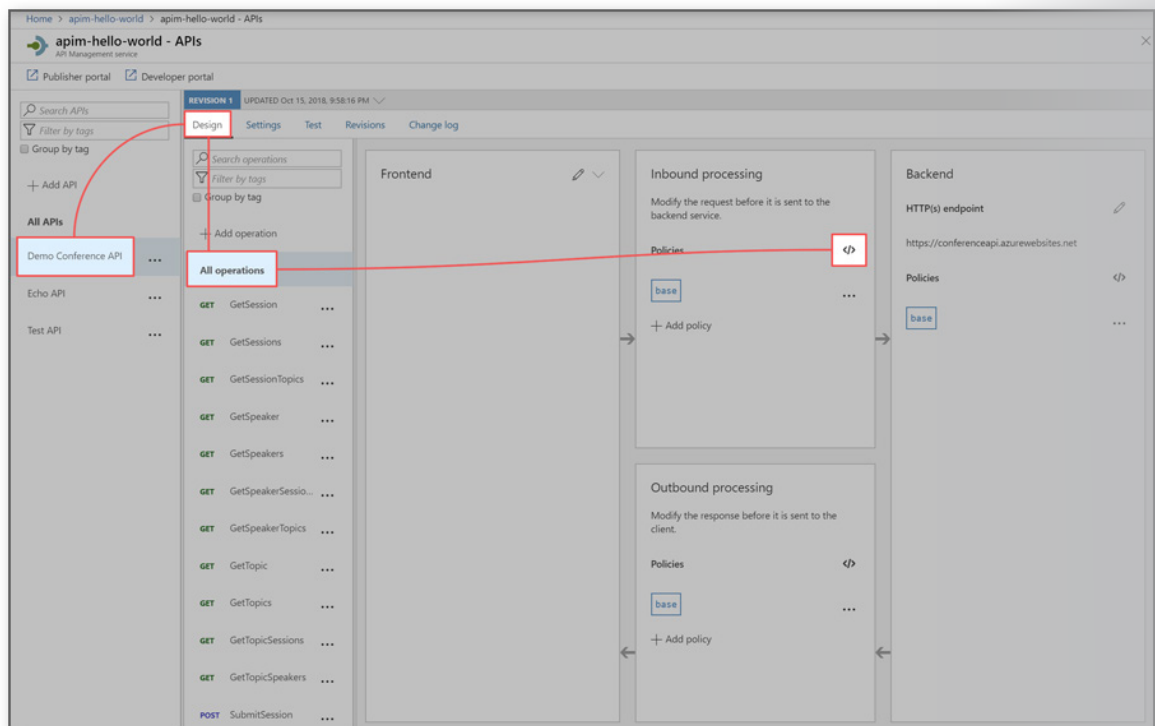
Setting and editing Azure API Management policies

The policy definition is an XML document that describes a sequence of inbound and outbound statements. The XML can be edited directly in the definition window. You can also select a predefined policy from the list that is provided to the right of the policy window. The statements applicable to the current scope are enabled and highlighted. Clicking an enabled statement adds the appropriate XML at the location of the cursor in the definition view.

Set or edit a policy

To set or edit a policy, follow the following steps:

1. Sign in to the Azure portal at <https://portal.azure.com>.
2. Browse to your APIM instance.
3. Click the **APIs** tab.



- 4.
5. Select one of the APIs that you previously imported.
6. Select the **Design** tab.

7. Select an operation to which you want to apply the policy. If you want to apply the policy to all operations, select **All operations**.
8. Select the `</>` (code editor) icon in the **Inbound processing** or **Outbound processing** section.
9. Paste the desired policy code into one of the appropriate blocks.

```
<policies>
  <inbound>
    <base />
  </inbound>
  <backend>
    <base />
  </backend>
  <outbound>
    <base />
  </outbound>
  <on-error>
    <base />
  </on-error>
</policies>
```

Configure scope

Policies can be configured globally or at the scope of a Product, API, or Operation. To begin configuring a policy, you must first select the scope at which the policy should apply.

Policy scopes are evaluated in the following order:

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

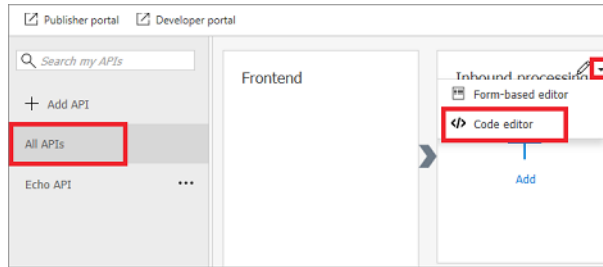
The statements within policies are evaluated according to the placement of the `<base>` element, if it is present. Global policy has no parent policy and using the `<base>` element in it has no effect.

To see the policies in the current scope in the policy editor, click **Recalculate effective policy for selected scope**.

Global scope

Global scope is configured for **All APIs** in your APIM instance.

1. Sign in to the Azure portal and navigate to your APIM instance.
2. Click **All APIs**. The image below shows how to start the code editor in the interface.



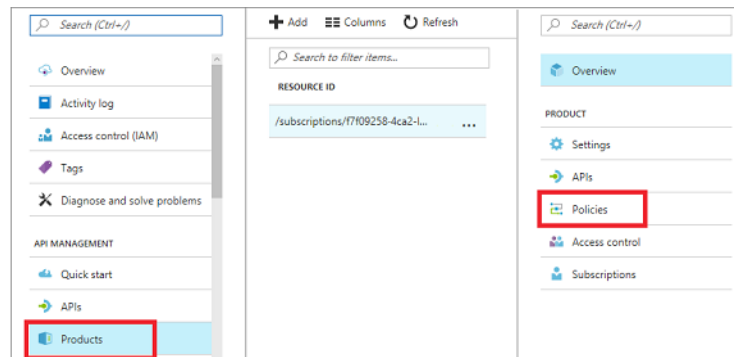
- 3.
4. Click the triangle icon.
5. Select **Code editor**.
6. Add or edit policies.
7. Press Save.

The changes are propagated to the API Management gateway immediately.

Product scope

Product scope is configured for the selected product.

1. Click **Products**.

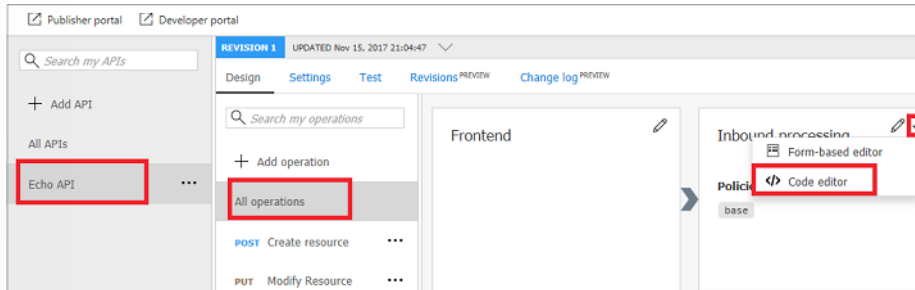


- 2.
3. Select the product to which you want to apply policies.
4. Click **Policies**.
5. Add or edit policies.
6. Press **Save**.

API scope

API scope is configured for All Operations of the selected API.

1. Select the **API** you want to apply policies to.

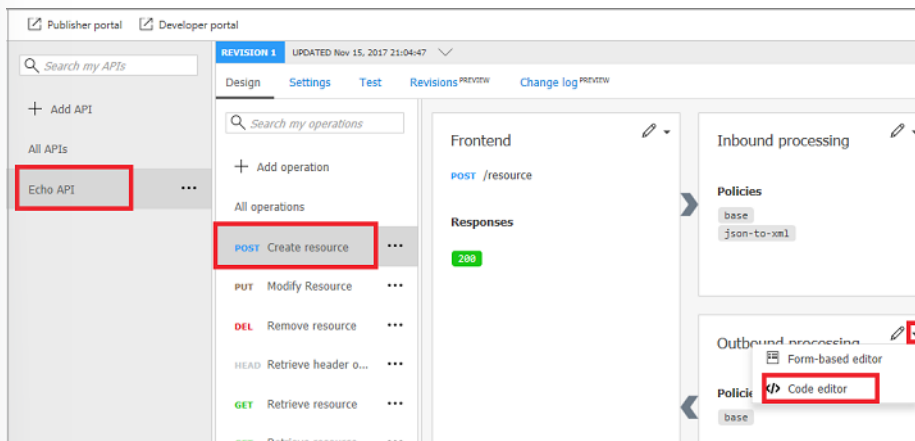


- 2.
3. Select **All** operations
4. Click the triangle icon.
5. Select **Code editor**.
6. Add or edit policies.
7. Press **Save**.

Operation scope

Operation scope is configured for the selected operation.

1. Select an **API**.
2. Select the operation you want to apply policies to.



- 3.
4. Click the triangle icon.
5. Select **Code editor**.
6. Add or edit policies.
7. Press **Save**.

Next

- Review the **Policy Reference**¹ for a full list of policy statements and their settings.
- We'll cover advanced policies in the next part of this lesson.

¹ <https://docs.microsoft.com/en-us/azure/api-management/api-management-policy-reference>

API Management advanced policies

This part of the lesson provides a reference for the following API Management 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.
- Limit concurrency - Prevents enclosed policies from executing by more than the specified number of requests at a time.
- Log to Event Hub - Sends messages in the specified format to an Event Hub defined by a Logger entity.
- Mock response - Aborts pipeline execution and returns a mocked response directly to the caller.
- 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.

Control flow

The `choose` policy applies enclosed policy statements based on the outcome of evaluation of Boolean expressions, similar to an if-then-else or a switch construct in a programming language.

Policy statement

```
<choose>
  <when condition="Boolean expression | Boolean constant">
    <!-- one or more policy statements to be applied if the above condi-
tion is true -->
  </when>
  <when condition="Boolean expression | Boolean constant">
    <!-- one or more policy statements to be applied if the above condi-
tion is true -->
  </when>
  <otherwise>
    <!-- one or more policy statements to be applied if none of the
above conditions are true -->
  </otherwise>
</choose>
```

The control flow policy must contain at least one `<when/>` element. The `<otherwise/>` element is optional. Conditions in `<when/>` elements are evaluated in order of their appearance within the policy. Policy statement(s) enclosed within the first `<when/>` element with condition attribute equals true will be applied. Policies enclosed within the `<otherwise/>` element, if present, will be applied if all of the `<when/>` element condition attributes are false.

Forward request

The `forward-request` policy forwards the incoming request to the backend service specified in the request context. The backend service URL is specified in the API settings and can be changed using the `set backend service` policy.

Removing this policy results in the request not being forwarded to the backend service and the policies in the outbound section are evaluated immediately upon the successful completion of the policies in the inbound section.

Policy statement

```
<forward-request timeout="time in seconds" follow-redirects="true | false"/>
```

Limit concurrency

The `limit-concurrency` policy prevents enclosed policies from executing by more than the specified number of requests at any time. Upon exceeding that number, new requests will fail immediately with a *429 Too Many Requests* status code.

Policy statement

```
<limit-concurrency key="expression" max-count="number">  
    <!-- nested policy statements -->  
</limit-concurrency>
```

Log to Event Hub

The `log-to-eventhub` policy sends messages in the specified format to an Event Hub defined by a Logger entity. As its name implies, the policy is used for saving selected request or response context information for online or offline analysis.

Policy statement

```
<log-to-eventhub logger-id="id of the logger entity" partition-id="index of  
the partition where messages are sent" partition-key="value used for parti-  
tion assignment">  
    Expression returning a string to be logged  
</log-to-eventhub>
```

Mock response

The `mock-response`, as the name implies, is used to mock APIs and operations. It aborts normal pipeline execution and returns a mocked response to the caller. The policy always tries to return responses of highest fidelity. It prefers response content examples, whenever available. It generates sample responses from schemas, when schemas are provided and examples are not. If neither examples or schemas are found, responses with no content are returned.

Policy statement

```
<mock-response status-code="code" content-type="media type"/>
```

Retry

The `retry` policy executes its child policies once and then retries their execution until the `retry condition` becomes false or `retry count` is exhausted.

Policy statement

```
<retry
  condition="boolean expression or literal"
  count="number of retry attempts"
  interval="retry interval in seconds"
  max-interval="maximum retry interval in seconds"
  delta="retry interval delta in seconds"
  first-fast-retry="boolean expression or literal">
  <!-- One or more child policies. No restrictions -->
</retry>
```

Return response

The `return-response` policy aborts pipeline execution and returns either a default or custom response to the caller. Default response is 200 OK with no body. Custom response can be specified via a context variable or policy statements. When both are provided, the response contained within the context variable is modified by the policy statements before being returned to the caller.

Policy statement

```
<return-response response-variable-name="existing context variable">
  <set-header/>
  <set-body/>
  <set-status/>
</return-response>
```

Next

- **Review samples²** of API Management policies

² <https://docs.microsoft.com/en-us/azure/api-management/policy-samples>

Review questions

Module 3 review questions

API Management

Azure API Management is an API gateway-as-a-service that streamlines many of the common tasks that are necessary when creating an API for external users, can you name some things that includes?

> Click to see suggested answer

- Creating a successful and useful developer portal.
- Helping to secure API endpoints from anonymous or unwanted access.
- Managing existing developer access through cache mechanisms, throttling, and other policies.
- Building a monitoring and analytics platform to diagnose issues and monitor adoption.
- Providing business users and developers with deep insights into how each API is specifically used.

Azure Application Gateway

Can Azure Application Gateway support app load balancing outside of the transport layer?

> Click to see suggested answer

Alerts can notify you of an errant scenario by either sending an email notification or invoking a webhook:

Azure Application Gateway supports application layer (layer 7 in the OSI model) load balancing. Since Azure Application Gateway is at the application layer, the service can handle custom routing, session affinity, Secure Sockets Layer (SSL) termination, firewall management, redirection, and other scenarios that require some knowledge of the application code. Application Gateway can perform URL-based routing and route traffic based on the incoming URL.

API management policies

Policies are a powerful capability of API Management that allows the Azure portal to change the behavior of the API through configuration. Policies are a quick way to change the behavior of an API for external developers without requiring any code changes in the actual back-end API application. How would you change the behavior of the API endpoint?

> Click to see suggested answer

To change the behavior of the API endpoint within API Management, you add XML elements to either the `inbound` or the `outbound` element. For example, you can convert XML data that comes from your back-end API to JSON for external users:

```
<policies>
  <inbound>
    <base />
  </inbound>
  <outbound>
```

```
        <base />
        <xml-to-json kind="direct" apply="always" consider-accept-header="-
false" />
    </outbound>
</policies>
```




Module 4 Develop event-based solutions

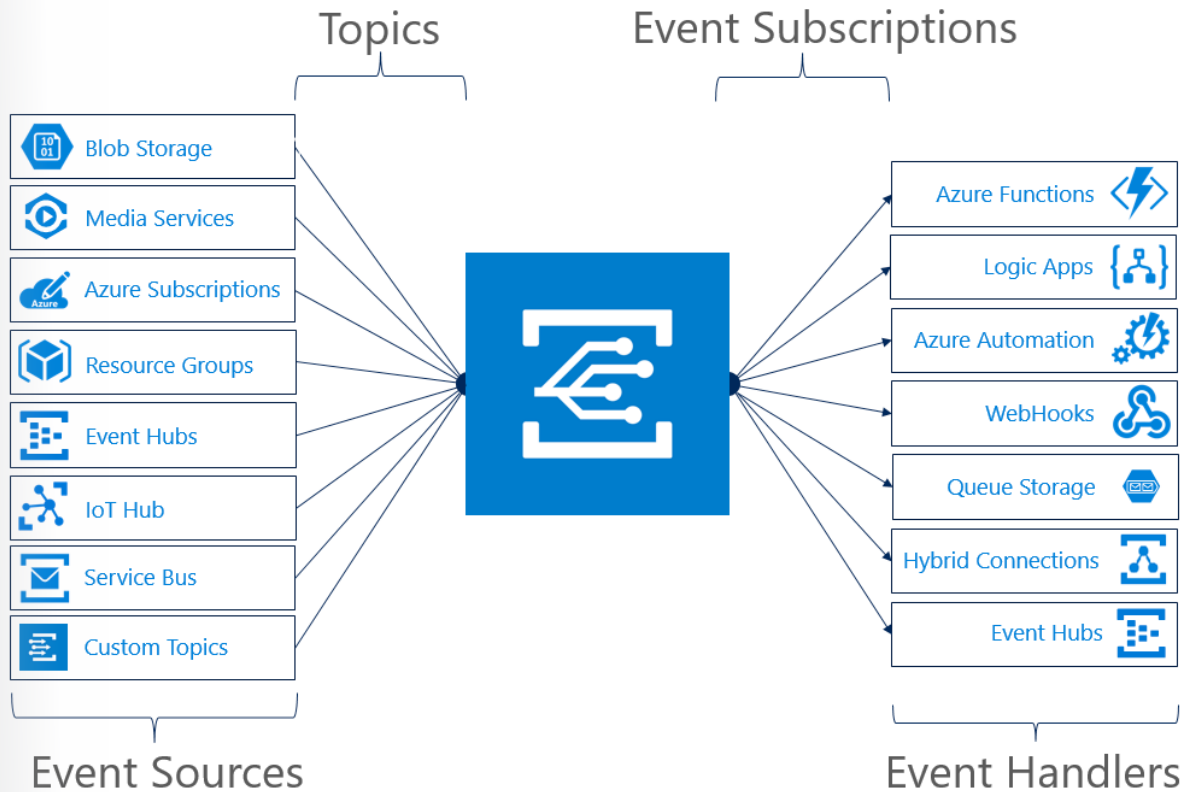
Implement solutions that use Azure Event Grid

Azure Event Grid overview

Azure Event Grid allows you to easily build applications with event-based architectures. First, select the Azure resource you would like to subscribe to, and then give the event handler or WebHook endpoint to send the event to. Event Grid has built-in support for events coming from Azure services, like storage blobs and resource groups. Event Grid also has support for your own events, using custom topics.

You can use filters to route specific events to different endpoints, multicast to multiple endpoints, and make sure your events are reliably delivered.

Currently, Azure Event Grid is available in all public regions.



Concepts in Azure Event Grid

There are five concepts in Azure Event Grid you need to understand to help you get started, and are described in more detail below:

- **Events** - What happened.
- **Event sources** - Where the event took place.
- **Topics** - The endpoint where publishers send events.
- **Event subscriptions** - The endpoint or built-in mechanism to route events, sometimes to more than one handler. Subscriptions are also used by handlers to intelligently filter incoming events.
- **Event handlers** - The app or service reacting to the event.

Events

An event is the smallest amount of information that fully describes something that happened in the system. Every event has common information like: source of the event, time the event took place, and unique identifier. Every event also has specific information that is only relevant to the specific type of event. For example, an event about a new file being created in Azure Storage has details about the file, such as the `lastTimeModified` value. Or, an Event Hubs event has the URL of the Capture file.

Each event is limited to 64 KB of data.

Event sources

An event source is where the event happens. Each event source is related to one or more event types. For example, Azure Storage is the event source for blob created events. IoT Hub is the event source for device created events. Your application is the event source for custom events that you define. Event sources are responsible for sending events to Event Grid.

Topics

The event grid topic provides an endpoint where the source sends events. The publisher creates the event grid topic, and decides whether an event source needs one topic or more than one topic. A topic is used for a collection of related events. To respond to certain types of events, subscribers decide which topics to subscribe to.

System topics are built-in topics provided by Azure services. You don't see system topics in your Azure subscription because the publisher owns the topics, but you can subscribe to them. To subscribe, you provide information about the resource you want to receive events from. As long as you have access the resource, you can subscribe to its events.

Custom topics are application and third-party topics. When you create or are assigned access to a custom topic, you see that custom topic in your subscription.

When designing your application, you have flexibility when deciding how many topics to create. For large solutions, create a custom topic for each category of related events. For example, consider an application that sends events related to modifying user accounts and processing orders. It's unlikely any event handler wants both categories of events. Create two custom topics and let event handlers subscribe to the one that interests them. For small solutions, you might prefer to send all events to a single topic. Event subscribers can filter for the event types they want.

Event subscriptions

A subscription tells Event Grid which events on a topic you're interested in receiving. When creating the subscription, you provide an endpoint for handling the event. You can filter the events that are sent to the endpoint. You can filter by event type, or subject pattern.

Event handlers

From an Event Grid perspective, an event handler is the place where the event is sent. The handler takes some further action to process the event. Event Grid supports several handler types. You can use a supported Azure service or your own webhook as the handler. Depending on the type of handler, Event Grid follows different mechanisms to guarantee the delivery of the event. For HTTP webhook event handlers, the event is retried until the handler returns a status code of 200 – OK. For Azure Storage Queue, the events are retried until the Queue service successfully processes the message push into the queue.

Next

- We'll take a look at the Azure Event Grid properties and schema that are present for all events.

Azure Event Grid event schema

Events consist of a set of five required string properties and a required data object. The properties are common to all events from any publisher. The data object has properties that are specific to each publisher. For system topics, these properties are specific to the resource provider, such as Azure Storage or Azure Event Hubs.

Event sources send events to Azure Event Grid in an array, which can have several event objects. When posting events to an event grid topic, the array can have a total size of up to 1 MB. Each event in the array is limited to 64 KB. If an event or the array is greater than the size limits, you receive the response 413 Payload Too Large.

Event Grid sends the events to subscribers in an array that has a single event. This behavior may change in the future.

Event schema

The following example shows the properties that are used by all event publishers:

```
[
  {
    "topic": string,
    "subject": string,
    "id": string,
    "eventType": string,
    "eventTime": string,
    "data": {
      object-unique-to-each-publisher
    },
    "dataVersion": string,
    "metadataVersion": string
  }
]
```

For example, the schema published for an Azure Blob storage event is:

```
[
  {
    "topic": "/subscriptions/{subscription-id}/resourceGroups/Storage/
providers/Microsoft.Storage/storageAccounts/xstoretestaccount",
    "subject": "/blobServices/default/containers/oc2d2817345i200097contain-
er/blobs/oc2d2817345i20002296blob",
    "eventType": "Microsoft.Storage.BlobCreated",
    "eventTime": "2017-06-26T18:41:00.9584103Z",
    "id": "831e1650-001e-001b-66ab-eeb76e069631",
    "data": {
      "api": "PutBlockList",
      "clientRequestId": "6d79dbfb-0e37-4fc4-981f-442c9ca65760",
      "requestId": "831e1650-001e-001b-66ab-eeb76e000000",
      "eTag": "0x8D4BCC2E4835CD0",
      "contentType": "application/octet-stream",
      "contentLength": 524288,
      "blobType": "BlockBlob",

```

```

    "url": "https://oc2d2817345i60006.blob.core.windows.net/oc2d2817345i-
200097container/oc2d2817345i20002296blob",
    "sequencer": "00000000000004420000000000028963",
    "storageDiagnostics": {
      "batchId": "b68529f3-68cd-4744-baa4-3c0498ec19f0"
    }
  },
  "dataVersion": "",
  "metadataVersion": "1"
}
]

```

Event properties

All events have the same following top-level data:

Property	Type	Description
topic	string	Full resource path to the event source. This field isn't writeable. Event Grid provides this value.
subject	string	Publisher-defined path to the event subject.
eventType	string	One of the registered event types for this event source.
eventTime	string	The time the event is generated based on the provider's UTC time.
id	string	Unique identifier for the event.
data	object	Event data specific to the resource provider.
dataVersion	string	The schema version of the data object. The publisher defines the schema version.
metadataVersion	string	The schema version of the event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value.

For custom topics, the event publisher determines the data object. The top-level data should have the same fields as standard resource-defined events.

When publishing events to custom topics, create subjects for your events that make it easy for subscribers to know whether they're interested in the event. Subscribers use the subject to filter and route events. Consider providing the path for where the event happened, so subscribers can filter by segments of that path. The path enables subscribers to narrowly or broadly filter events. For example, if you provide a three segment path like `/A/B/C` in the subject, subscribers can filter by the first segment `/A` to get a broad set of events. Those subscribers get events with subjects like `/A/B/C` or `/A/D/E`. Other subscribers can filter by `/A/B` to get a narrower set of events.

Sometimes your subject needs more detail about what happened. For example, the **Storage Accounts** publisher provides the subject `/blobServices/default/containers/<container-name>/blobs/<file>` when a file is added to a container. A subscriber could filter by the path `/blobServices/default/containers/testcontainer` to get all events for that container but not other containers in the storage account. A subscriber could also filter or route by the suffix `.txt` to only work with text files.

Next

- We'll cover Event Grid security and authentication.

Event Grid security and authentication

Azure Event Grid has three types of authentication:

- WebHook event delivery
- Event subscriptions
- Custom topic publishing

WebHook Event delivery

Webhooks are one of the many ways to receive events from Azure Event Grid. When a new event is ready, Event Grid service POSTs an HTTP request to the configured endpoint with the event in the request body.

Like many other services that support webhooks, Event Grid requires you to prove ownership of your Webhook endpoint before it starts delivering events to that endpoint. This requirement prevents a malicious user from flooding your endpoint with events. When you use any of the three Azure services listed below, the Azure infrastructure automatically handles this validation:

- Azure Logic Apps with Event Grid Connector
- Azure Automation via webhook
- Azure Functions with Event Grid Trigger

If you're using any other type of endpoint, such as an HTTP trigger based Azure function, your endpoint code needs to participate in a validation handshake with Event Grid. Event Grid supports two ways of validating the subscription.

1. **ValidationCode handshake (programmatic):** If you control the source code for your endpoint, this method is recommended. At the time of event subscription creation, Event Grid sends a subscription validation event to your endpoint. The schema of this event is similar to any other Event Grid event. The data portion of this event includes a `validationCode` property. Your application verifies that the validation request is for an expected event subscription, and echoes the validation code to Event Grid. This handshake mechanism is supported in all Event Grid versions.
2. **ValidationURL handshake (manual):** In certain cases, you can't access the source code of the endpoint to implement the ValidationCode handshake. For example, if you use a third-party service (like Zapier or IFTTT), you can't programmatically respond with the validation code.
3. Starting with version 2018-05-01-preview, Event Grid supports a manual validation handshake. If you're creating an event subscription with an SDK or tool that uses API version 2018-05-01-preview or later, Event Grid sends a `validationUrl` property in the data portion of the subscription validation event. To complete the handshake, find that URL in the event data and manually send a GET request to it. You can use either a REST client or your web browser.

4. The provided URL is valid for 10 minutes. During that time, the provisioning state of the event subscription is `AwaitingManualAction`. If you don't complete the manual validation within 10 minutes, the provisioning state is set to `Failed`. You'll have to create the event subscription again before starting the manual validation.

Validation details

- At the time of event subscription creation/update, Event Grid posts a subscription validation event to the target endpoint.
- The event contains a header value `"aeg-event-type: SubscriptionValidation"`.
- The event body has the same schema as other Event Grid events.
- The `eventType` property of the event is `Microsoft.EventGrid.SubscriptionValidationEvent`.
- The data property of the event includes a `validationCode` property with a randomly generated string. For example, `"validationCode: acb13..."`.
- If you're using API version 2018-05-01-preview, the event data also includes a `validationUrl` property with a URL for manually validating the subscription.
- The array contains only the validation event. Other events are sent in a separate request after you echo back the validation code.
- The EventGrid DataPlane SDKs have classes corresponding to the subscription validation event data and subscription validation response.

An example `SubscriptionValidationEvent` is shown in the following example:

```
[{
  "id": "2d1781af-3a4c-4d7c-bd0c-e34b19da4e66",
  "topic": "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "subject": "",
  "data": {
    "validationCode": "512d38b6-c7b8-40c8-89fe-f46f9e9622b6",
    "validationUrl": "https://rp-eastus2.eventgrid.azure.net:553/eventsubscriptions/etest/validate?id=B2E34264-7D71-453A-B5FB-B62D0FDC85EE&t=2018-04-26T20:30:54.4538837Z&apiVersion=2018-05-01-preview&token=1BN-qCxBSSE9OnNSfZM4%2b5H9zDegKMY6uJ%2fO2DFRkwQ%3d"
  },
  "eventType": "Microsoft.EventGrid.SubscriptionValidationEvent",
  "eventTime": "2018-01-25T22:12:19.4556811Z",
  "metadataVersion": "1",
  "dataVersion": "1"
}]
```

To prove endpoint ownership, echo back the validation code in the `validationResponse` property, as shown in the following example:

```
{
  "validationResponse": "512d38b6-c7b8-40c8-89fe-f46f9e9622b6"
}
```

You must return an `HTTP 200 OK` response status code. `HTTP 202 Accepted` is not recognized as a valid Event Grid subscription validation response.

Or, you can manually validate the subscription by sending a `GET` request to the validation URL. The event subscription stays in a pending state until validated.

Event delivery security

You can secure your webhook endpoint by adding query parameters to the webhook URL when creating an Event Subscription. Set one of these query parameters to be a secret such as an access token. The webhook can use the secret to recognize the event is coming from Event Grid with valid permissions. Event Grid will include these query parameters in every event delivery to the webhook.

When editing the Event Subscription, the query parameters aren't displayed or returned unless the `--include-full-endpoint-url` parameter is used in Azure CLI.

Finally, it's important to note that Azure Event Grid only supports HTTPS webhook endpoints.

Event subscription

To subscribe to an event, you must prove that you have access to the event source and handler. Proving that you own a WebHook was covered in the preceding section. If you're using an event handler that isn't a WebHook (such as an event hub or queue storage), you need write access to that resource. This permissions check prevents an unauthorized user from sending events to your resource.

You must have the **Microsoft.EventGrid/EventSubscriptions/Write** permission on the resource that is the event source. You need this permission because you're writing a new subscription at the scope of the resource. The required resource differs based on whether you're subscribing to a system topic or custom topic. Both types are described in this section.

System topics (Azure service publishers)

For system topics, you need permission to write a new event subscription at the scope of the resource publishing the event. The format of the resource is: `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/{resource-provider}/{resource-type}/{resource-name}`

For example, to subscribe to an event on a storage account named *myacct*, you need the **Microsoft.EventGrid/EventSubscriptions/Write** permission on: `/subscriptions/####/resourceGroups/testrg/providers/Microsoft.Storage/storageAccounts/myacct`

Custom topics

For custom topics, you need permission to write a new event subscription at the scope of the event grid topic. The format of the resource is: `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.EventGrid/topics/{topic-name}`

For example, to subscribe to a custom topic named *mytopic*, you need the **Microsoft.EventGrid/EventSubscriptions/Write** permission on: `/subscriptions/####/resourceGroups/testrg/providers/Microsoft.EventGrid/topics/mytopic`

Custom topic publishing

Custom topics use either Shared Access Signature (SAS) or key authentication. We recommend SAS, but key authentication provides simple programming, and is compatible with many existing webhook publishers.

You include the authentication value in the HTTP header. For SAS, use `aeg-sas-token` for the header value. For key authentication, use `aeg-sas-key` for the header value.

Key authentication

Key authentication is the simplest form of authentication. Use the format: `aeg-sas-key: <your key>`

For example, you pass a key with:

```
aeg-sas-key: VxbGWce53249Mt8wuotr0GPmyJ/nDT4hgdEj9DpBeRr38arnnm5OFg==
```

SAS tokens

SAS tokens for Event Grid include the resource, an expiration time, and a signature. The format of the SAS token is: `r={resource}&e={expiration}&s={signature}`.

The resource is the path for the event grid topic to which you're sending events. For example, a valid resource path is: `https://<youtopic>.<region>.eventgrid.azure.net/eventGrid/api/events`

You generate the signature from a key.

For example, a valid `aeg-sas-token` value is:

```
aeg-sas-token: r=https%3a%2f%2fmytopic.eventgrid.azure.net%2feventGrid%2fapi%2fevent&e=6%2f15%2f2017+6%3a20%3a15+PM&s=a4oNHpRZy-gINC%2fBPjdDLORc6THPy3tDcGHw1zP4OajQ%3d
```

The following example creates a SAS token for use with Event Grid:

```
static string BuildSharedAccessSignature(string resource, DateTime expirationUtc, string key)
{
    const char Resource = 'r';
    const char Expiration = 'e';
    const char Signature = 's';

    string encodedResource = HttpUtility.UrlEncode(resource);
    var culture = CultureInfo.CreateSpecificCulture("en-US");
    var encodedExpirationUtc = HttpUtility.UrlEncode(expirationUtc.ToString(culture));

    string unsignedSas = $"{{Resource}}={{encodedResource}}&{{Expiration}}={{encodedExpirationUtc}}";
    using (var hmac = new HMACSHA256(Convert.FromBase64String(key)))
    {
        string signature = Convert.ToBase64String(hmac.ComputeHash(Encoding.UTF8.GetBytes(unsignedSas)));
    }
}
```

```

        string encodedSignature = HttpUtility.UrlEncode(signature);
        string signedSas = $"{unsignedSas}&{Signature}={encodedSignature}";

        return signedSas;
    }
}

```

Management Access Control

Azure Event Grid allows you to control the level of access given to different users to do various management operations such as list event subscriptions, create new ones, and generate keys. Event Grid uses Azure's role-based access control (RBAC).

Operation types

Event Grid supports the following actions:

- Microsoft.EventGrid/*/read
- Microsoft.EventGrid/*/write
- Microsoft.EventGrid/*/delete
- Microsoft.EventGrid/eventSubscriptions/getFullUrl/action
- Microsoft.EventGrid/topics/listKeys/action
- Microsoft.EventGrid/topics/regenerateKey/action

The last three operations return potentially secret information, which gets filtered out of normal read operations. It's recommended that you restrict access to these operations.

Next

- We'll cover the different ways to filter which events are sent to your endpoint.

Event filtering for Event Grid subscriptions

When creating an event subscription, you have three options for filtering:

- Event types
- Subject begins with or ends with
- Advanced fields and operators

Event type filtering

By default, all event types for the event source are sent to the endpoint. You can decide to send only certain event types to your endpoint. For example, you can get notified of updates to your resources, but not notified for other operations like deletions. In that case, filter by the `Microsoft.Resources.ResourceWriteSuccess` event type. Provide an array with the event types, or specify `All` to get all event types for the event source.

The JSON syntax for filtering by event type is:

```

"filter": {
  "includedEventTypes": [

```

```

        "Microsoft.Resources.ResourceWriteFailure",
        "Microsoft.Resources.ResourceWriteSuccess"
    ]
}

```

Subject filtering

For simple filtering by subject, specify a starting or ending value for the subject. For example, you can specify the subject ends with `.txt` to only get events related to uploading a text file to storage account. Or, you can filter the subject begins with `/blobServices/default/containers/testcontainer` to get all events for that container but not other containers in the storage account.

When publishing events to custom topics, create subjects for your events that make it easy for subscribers to know whether they're interested in the event. Subscribers use the subject property to filter and route events. Consider adding the path for where the event happened, so subscribers can filter by segments of that path. The path enables subscribers to narrowly or broadly filter events. If you provide a three segment path like `/A/B/C` in the subject, subscribers can filter by the first segment `/A` to get a broad set of events. Those subscribers get events with subjects like `/A/B/C` or `/A/D/E`. Other subscribers can filter by `/A/B` to get a narrower set of events.

The JSON syntax for filtering by event type is:

```

"filter": {
  "subjectBeginsWith": "/blobServices/default/containers/mycontainer/log",
  "subjectEndsWith": ".jpg"
}

```

Advanced filtering

To filter by values in the data fields and specify the comparison operator, use the advanced filtering option. In advanced filtering, you specify the:

- operator type - The type of comparison.
- key - The field in the event data that you're using for filtering. It can be a number, boolean, or string.
- value or values - The value or values to compare to the key.

The JSON syntax for using advanced filters is:

```

"filter": {
  "advancedFilters": [
    {
      "operatorType": "NumberGreaterThanOrEquals",
      "key": "Data.Key1",
      "value": 5
    },
    {
      "operatorType": "StringContains",
      "key": "Subject",
      "values": ["container1", "container2"]
    }
  ]
}

```

```
}
```

Operator

The available operators for numbers and strings are:

Number Operators	String Operators
NumberGreaterThan	StringContains
NumberGreaterThanOrEquals	StringBeginsWith
NumberLessThan	StringEndsWith
NumberLessThanOrEquals	StringIn
NumberIn	StringNotIn
NumberNotIn	

The available operator for booleans is: `BoolEquals`

Note: All string comparisons are case-insensitive.

Key

The table below shows the available keys for events in the Event Grid, and Cloud Events, schema:

Event Grid schema	Cloud Events schema
Id	EventId
Topic	Source
Subject	EventType
EventType	EventTypeVersion
DataVersion	Event data (like <code>Data.key1</code>)
Event data (like <code>Data.key1</code>)	

For custom input schema, use the event data fields (like `Data.key1`).

Values

The values can be:

- number
- string
- boolean
- array

Limitations

Advanced filtering has the following limitations:

- Five advanced filters per event grid subscription
- 512 characters per string value
- Five values for in and not in operators
- The key can only have one level of nesting (like `data.key1`)

- Custom event schemas can be filtered only on top-level fields

The same key can be used in more than one filter.

Next

- We'll cover how to create custom events by walking through the process (through the Azure CLI) of creating a custom topic, subscribing to the custom topic, and triggering the event to view the result

Azure Event Grid custom events

In this part of the lesson, you use the Azure CLI to create a custom topic, subscribe to the custom topic, and trigger the event to view the result. Typically, you send events to an endpoint that processes the event data and takes actions. However, to simplify things, you send the events to a web app that collects and displays the messages.

The walkthrough below uses the Azure CLI but you can use the CLI locally if you're running version 2.0.24 or later.

Create a resource group

Event Grid topics are Azure resources, and must be placed in an Azure resource group. The resource group is a logical collection into which Azure resources are deployed and managed.

Create a resource group with the `az group create` command.

The following example creates a resource group named *gridResourceGroup* in the *westus2* location.

```
az group create --name gridResourceGroup --location westus2
```

Enable Event Grid resource provider

If you haven't previously used Event Grid in your Azure subscription, you may need to register the Event Grid resource provider. Run the following command to register the provider:

```
az provider register --namespace Microsoft.EventGrid
```

It may take a moment for the registration to finish. To check the status, run:

```
az provider show --namespace Microsoft.EventGrid --query "registration-State"
```

When `registrationState` is `Registered`, you're ready to continue.

Create a custom topic

An event grid topic provides a user-defined endpoint that you post your events to. The following example creates the custom topic in your resource group. Replace `<your-topic-name>` with a unique name for your topic. The custom topic name must be unique because it's part of the DNS entry. Additionally, it must be between 3-50 characters and contain only values a-z, A-Z, 0-9, and `-`

```
topicname=<your-topic-name>
```



```
az eventgrid topic create --name $topicname -l westus2 -g gridResourceGroup
```

Create a message endpoint

Before subscribing to the custom topic, let's create the endpoint for the event message. Typically, the endpoint takes actions based on the event data. To simplify this quickstart, you deploy a pre-built web app that displays the event messages. The deployed solution includes an App Service plan, an App Service web app, and source code from GitHub.

Replace `<your-site-name>` with a unique name for your web app. The web app name must be unique because it's part of the DNS entry.

```
sitename=<your-site-name>

az group deployment create \
  --resource-group gridResourceGroup \
  --template-uri "https://raw.githubusercontent.com/Azure-Samples/azure-
event-grid-viewer/master/azuredeploy.json" \
  --parameters siteName=$sitename hostingPlanName=viewerhost
```

The deployment may take a few minutes to complete. After the deployment has succeeded, view your web app to make sure it's running. In a web browser, navigate to: `https://<your-site-name>.azurewebsites.net`.

You should see the site with no messages currently displayed.

Subscribe to a custom topic

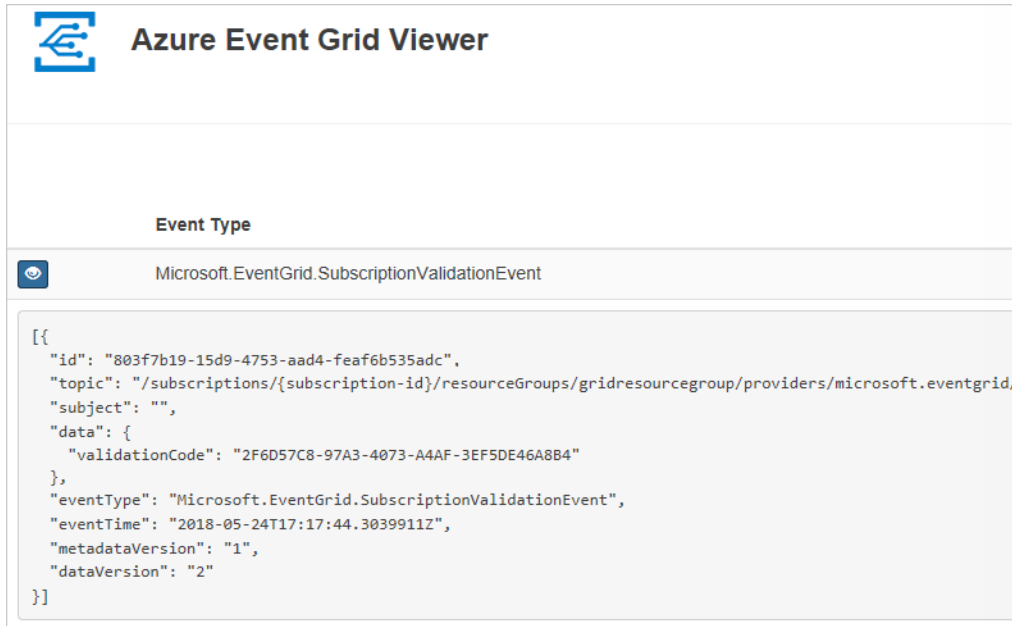
You subscribe to an event grid topic to tell Event Grid which events you want to track and where to send those events. The following example subscribes to the custom topic you created, and passes the URL from your web app as the endpoint for event notification.

The endpoint for your web app must include the suffix `/api/updates/`.

```
endpoint=https://$sitename.azurewebsites.net/api/updates

az eventgrid event-subscription create \
  -g gridResourceGroup \
  --topic-name $topicname \
  --name demoViewerSub \
  --endpoint $endpoint
```

View your web app again, and notice that a subscription validation event has been sent to it. Select the eye icon to expand the event data. Event Grid sends the validation event so the endpoint can verify that it wants to receive event data. The web app includes code to validate the subscription.



Send an event to your custom topic

Let's trigger an event to see how Event Grid distributes the message to your endpoint. First, let's get the URL and key for the custom topic.

```
endpoint=$(az eventgrid topic show --name $topicname -g gridResourceGroup
--query "endpoint" --output tsv)
key=$(az eventgrid topic key list --name $topicname -g gridResourceGroup
--query "key1" --output tsv)
```

To simplify this article, you use sample event data to send to the custom topic. Typically, an application or Azure service would send the event data. The following example creates sample event data:

```
event='[ {"id": "'"$RANDOM"'", "eventType": "recordInserted", "subject":
"myapp/vehicles/motorcycles", "eventTime": "'`date +%Y-%m-%dT%H:%M:%S%z`'",
"data":{ "make": "Ducati", "model": "Monster"},"dataVersion": "1.0"} ] '
```

The data element of the JSON is the payload of your event. Any well-formed JSON can go in this field. You can also use the subject field for advanced routing and filtering.

CURL is a utility that sends HTTP requests. In this article, use CURL to send the event to the topic.

```
curl -X POST -H "aeg-sas-key: $key" -d "$event" $endpoint
```

You've triggered the event, and Event Grid sent the message to the endpoint you configured when subscribing. View your web app to see the event you just sent.

```
[{
  "id": "1807",
  "eventType": "recordInserted",
  "subject": "myapp/vehicles/motorcycles",
  "eventTime": "2017-08-10T21:03:07+00:00",
```

```
"data": {
  "make": "Ducati",
  "model": "Monster"
},
"dataVersion": "1.0",
"metadataVersion": "1",
"topic": "/subscriptions/{subscription-id}/resourceGroups/{re-
source-group}/providers/Microsoft.EventGrid/topics/{topic}"
}]
```

Clean up resources

If you plan to continue working with this event or the event viewer app, don't clean up the resources created in this article. Otherwise, use the following command to delete the resources you created in this article.

```
az group delete --name gridResourceGroup
```

Next

- In the next lesson we'll cover Azure Event Hubs.

Implement solutions that use Azure Event Hubs

Azure Event Hubs overview

Azure Event Hubs is a scalable event processing service that ingests and processes large volumes of events and data, with low latency and high reliability.

This lesson starts off with an overview of the features and terminology, and provides technical and implementation details about Event Hubs components and features.

Namespace

An Event Hubs namespace provides a unique scoping container, referenced by its fully qualified domain name, in which you create one or more event hubs or Kafka topics.

Event Hubs for Apache Kafka

This feature provides an endpoint that enables customers to talk to Event Hubs using the Kafka protocol. This integration provides customers a Kafka endpoint. This enables customers to configure their existing Kafka applications to talk to Event Hubs, giving an alternative to running their own Kafka clusters. Event Hubs for Apache Kafka supports Kafka protocol 1.0 and later.

With this integration, you don't need to run Kafka clusters or manage them with Zookeeper. This also allows you to work with some of the most demanding features of Event Hubs like Capture, Auto-inflate, and Geo-disaster Recovery.

This integration also allows applications like Mirror Maker or framework like Kafka Connect to work clusterless with just configuration changes.

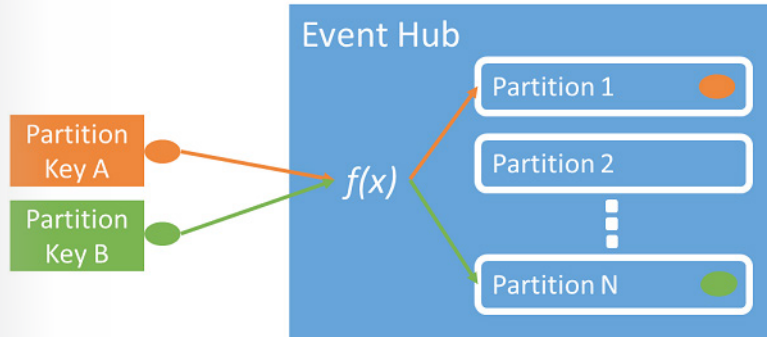
Event publishers

Any entity that sends data to an event hub is an event producer, or event publisher. Event publishers can publish events using HTTPS or AMQP 1.0 or Kafka 1.0 and later. Event publishers use a Shared Access Signature (SAS) token to identify themselves to an event hub, and can have a unique identity, or use a common SAS token.

Publishing an event

You can publish an event via AMQP 1.0, Kafka 1.0 (and later), or HTTPS. Event Hubs provides client libraries and classes for publishing events to an event hub from .NET clients. For other runtimes and platforms, you can use any AMQP 1.0 client, such as Apache Qpid. You can publish events individually, or batched. A single publication (event data instance) has a limit of 1 MB, regardless of whether it is a single event or a batch. Publishing events larger than this threshold results in an error. It is a best practice for publishers to be unaware of partitions within the event hub and to only specify a partition key (introduced in the next section), or their identity via their SAS token.

The choice to use AMQP or HTTPS is specific to the usage scenario. AMQP requires the establishment of a persistent bidirectional socket in addition to transport level security (TLS) or SSL/TLS. AMQP has higher network costs when initializing the session, however HTTPS requires additional SSL overhead for every request. AMQP has higher performance for frequent publishers.



Event Hubs ensures that all events sharing a partition key value are delivered in order, and to the same partition. If partition keys are used with publisher policies, then the identity of the publisher and the value of the partition key must match. Otherwise, an error occurs.

Publisher policy

Event Hubs enables granular control over event publishers through *publisher policies*. Publisher policies are run-time features designed to facilitate large numbers of independent event publishers. With publisher policies, each publisher uses its own unique identifier when publishing events to an event hub, using the following mechanism:

```
//[my namespace].servicebus.windows.net/[event hub name]/publishers/[my  
publisher name]
```

You don't have to create publisher names ahead of time, but they must match the SAS token used when publishing an event, in order to ensure independent publisher identities. When using publisher policies, the PartitionKey value is set to the publisher name. To work properly, these values must match.

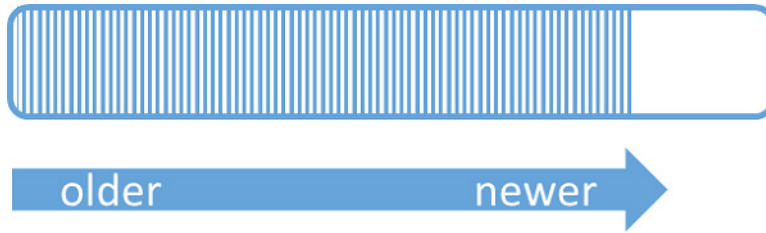
Capture

Event Hubs Capture enables you to automatically capture the streaming data in Event Hubs and save it to your choice of either a Blob storage account, or an Azure Data Lake Service account. You can enable Capture from the Azure portal, and specify a minimum size and time window to perform the capture. Using Event Hubs Capture, you specify your own Azure Blob Storage account and container, or Azure Data Lake Service account, one of which is used to store the captured data. Captured data is written in the Apache Avro format.

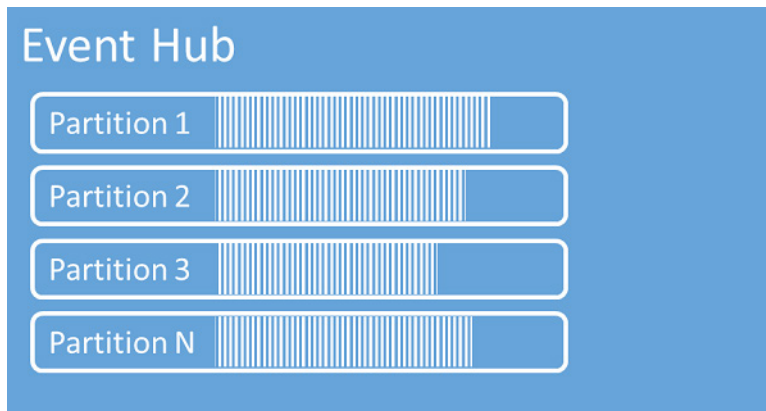
Partitions

Event Hubs provides message streaming through a partitioned consumer pattern in which each consumer only reads a specific subset, or partition, of the message stream. This pattern enables horizontal scale for event processing and provides other stream-focused features that are unavailable in queues and topics.

A partition is an ordered sequence of events that is held in an event hub. As newer events arrive, they are added to the end of this sequence. A partition can be thought of as a "commit log."



Event Hubs retains data for a configured retention time that applies across all partitions in the event hub. Events expire on a time basis; you cannot explicitly delete them. Because partitions are independent and contain their own sequence of data, they often grow at different rates.



The number of partitions is specified at creation and must be between 2 and 32. The partition count is not changeable, so you should consider long-term scale when setting partition count. Partitions are a data organization mechanism that relates to the downstream parallelism required in consuming applications. The number of partitions in an event hub directly relates to the number of concurrent readers you expect to have. You can increase the number of partitions beyond 32 by contacting the Event Hubs team.

While partitions are identifiable and can be sent to directly, sending directly to a partition is not recommended. Instead, you can use higher level constructs introduced in the Event publisher and Capacity sections.

Partitions are filled with a sequence of event data that contains the body of the event, a user-defined property bag, and metadata such as its offset in the partition and its number in the stream sequence.

Partition key

You can use a partition key to map incoming event data into specific partitions for the purpose of data organization. The partition key is a sender-supplied value passed into an event hub. It is processed through a static hashing function, which creates the partition assignment. If you don't specify a partition key when publishing an event, a round-robin assignment is used.

The event publisher is only aware of its partition key, not the partition to which the events are published. This decoupling of key and partition insulates the sender from needing to know too much about the downstream processing. A per-device or user unique identity makes a good partition key, but other attributes such as geography can also be used to group related events into a single partition.

SAS tokens

Event Hubs uses Shared Access Signatures, which are available at the namespace and event hub level. A SAS token is generated from a SAS key and is an SHA hash of a URL, encoded in a specific format. Using the name of the key (policy) and the token, Event Hubs can regenerate the hash and thus authenticate the sender. Normally, SAS tokens for event publishers are created with only send privileges on a specific event hub. This SAS token URL mechanism is the basis for publisher identification introduced in the publisher policy.

Event consumers

Any entity that reads event data from an event hub is an event consumer. All Event Hubs consumers connect via the AMQP 1.0 session and events are delivered through the session as they become available. The client does not need to poll for data availability.

Consumer groups

The publish/subscribe mechanism of Event Hubs is enabled through consumer groups. A consumer group is a view (state, position, or offset) of an entire event hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream, and to read the stream independently at their own pace and with their own offsets.

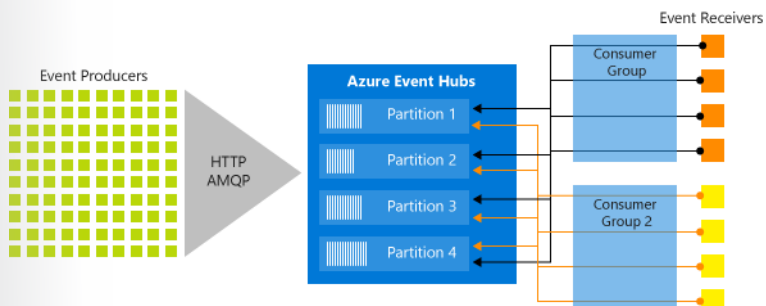
In a stream processing architecture, each downstream application equates to a consumer group. If you want to write event data to long-term storage, then that storage writer application is a consumer group. Complex event processing can then be performed by another, separate consumer group. You can only access partitions through a consumer group. There is always a default consumer group in an event hub, and you can create up to 20 consumer groups for a Standard tier event hub.

There can be at most 5 concurrent readers on a partition per consumer group; however it is recommended that there is only one active receiver on a partition per consumer group. Within a single partition, each reader receives all of the messages. If you have multiple readers on the same partition, then you process duplicate messages. You need to handle this in your code, which may not be trivial. However, it's a valid approach in some scenarios.

The following are examples of the consumer group URI convention:

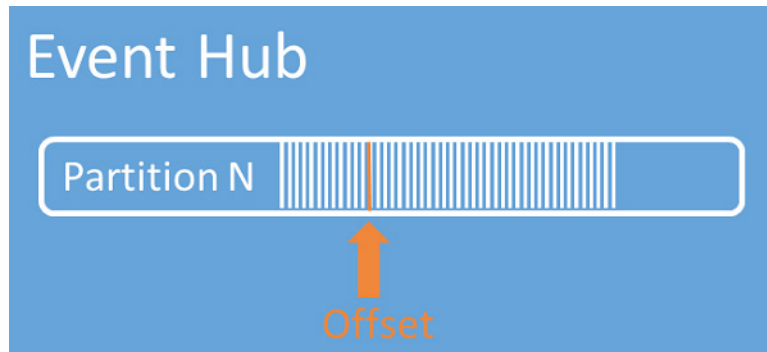
```
//[my namespace].servicebus.windows.net/[event hub name]/[Consumer Group #1]
//[my namespace].servicebus.windows.net/[event hub name]/[Consumer Group #2]
```

The following figure shows the Event Hubs stream processing architecture:



Stream offsets

An *offset* is the position of an event within a partition. You can think of an offset as a client-side cursor. The offset is a byte numbering of the event. This offset enables an event consumer (reader) to specify a point in the event stream from which they want to begin reading events. You can specify the offset as a timestamp or as an offset value. Consumers are responsible for storing their own offset values outside of the Event Hubs service. Within a partition, each event includes an offset.



Checkpointing

Checkpointing is a process by which readers mark or commit their position within a partition event sequence. Checkpointing is the responsibility of the consumer and occurs on a per-partition basis within a consumer group. This responsibility means that for each consumer group, each partition reader must keep track of its current position in the event stream, and can inform the service when it considers the data stream complete.

If a reader disconnects from a partition, when it reconnects it begins reading at the checkpoint that was previously submitted by the last reader of that partition in that consumer group. When the reader connects, it passes the offset to the event hub to specify the location at which to start reading. In this way, you can use checkpointing to both mark events as "complete" by downstream applications, and to provide resiliency if a failover between readers running on different machines occurs. It is possible to return to older data by specifying a lower offset from this checkpointing process. Through this mechanism, checkpointing enables both failover resiliency and event stream replay.

Common consumer tasks

All Event Hubs consumers connect via an AMQP 1.0 session, a state-aware bidirectional communication channel. Each partition has an AMQP 1.0 session that facilitates the transport of events segregated by partition.

Connect to a partition

When connecting to partitions, it is common practice to use a leasing mechanism to coordinate reader connections to specific partitions. This way, it is possible for every partition in a consumer group to have only one active reader. Checkpointing, leasing, and managing readers are simplified by using the `EventProcessorHost` class for .NET clients. The Event Processor Host is an intelligent consumer agent.

Read events

After an AMQP 1.0 session and link is opened for a specific partition, events are delivered to the AMQP 1.0 client by the Event Hubs service. This delivery mechanism enables higher throughput and lower latency than pull-based mechanisms such as HTTP GET. As events are sent to the client, each event data instance contains important metadata such as the offset and sequence number that are used to facilitate checkpointing on the event sequence.

Event data:

- Offset
- Sequence number
- Body
- User properties
- System properties

It is your responsibility to manage the offset.

Capacity

The throughput capacity of Event Hubs is controlled by throughput units. Throughput units are pre-purchased units of capacity. A single throughput unit includes the following capacity:

- **Ingress:** Up to 1 MB per second or 1000 events per second (whichever comes first).
- **Egress:** Up to 2 MB per second or 4096 events per second.

Beyond the capacity of the purchased throughput units, ingress is throttled and a `ServerBusyException` is returned. Egress does not produce throttling exceptions, but is still limited to the capacity of the purchased throughput units. If you receive publishing rate exceptions or are expecting to see higher egress, be sure to check how many throughput units you have purchased for the namespace. You can manage throughput units on the Scale blade of the namespaces in the Azure portal. You can also manage throughput units programmatically using the Event Hubs APIs.

Throughput units are pre-purchased and are billed per hour. Once purchased, throughput units are billed for a minimum of one hour. Up to 20 throughput units can be purchased for an Event Hubs namespace and are shared across all event hubs in that namespace.

You can purchase more throughput units in blocks of 20, up to 100 throughput units, by contacting Azure support. Beyond that limit, you can purchase blocks of 100 throughput units.

We recommend that you balance throughput units and partitions to achieve optimal scale. A single partition has a maximum scale of one throughput unit. The number of throughput units should be less than or equal to the number of partitions in an event hub.

Next

- We'll take a look at how Event Hubs Capture can automatically store all of the streaming data to Azure storage.

Capture events through Azure Event Hubs

Azure Event Hubs enables you to automatically capture the streaming data in Event Hubs in an Azure Blob storage or Azure Data Lake Storage account of your choice, with the added flexibility of specifying a

time or size interval. Setting up Capture is fast, there are no administrative costs to run it, and it scales automatically with Event Hubs throughput units. Event Hubs Capture is the easiest way to load streaming data into Azure, and enables you to focus on data processing rather than on data capture.

How Event Hubs Capture works

Event Hubs is a time-retention durable buffer for telemetry ingress, similar to a distributed log. The key to scaling in Event Hubs is the partitioned consumer model. Each partition is an independent segment of data and is consumed independently. Over time this data ages off, based on the configurable retention period. As a result, a given event hub never gets “too full.”

Event Hubs Capture enables you to specify your own Azure Blob storage account and container, or Azure Data Lake Store account, which are used to store the captured data. These accounts can be in the same region as your event hub or in another region, adding to the flexibility of the Event Hubs Capture feature.

Captured data is written in Apache Avro format: a compact, fast, binary format that provides rich data structures with inline schema. This format is widely used in the Hadoop ecosystem, Stream Analytics, and Azure Data Factory. More information about working with Avro is available later in this article.

Capture windowing

Event Hubs Capture enables you to set up a window to control capturing. This window is a minimum size and time configuration with a “first wins policy,” meaning that the first trigger encountered causes a capture operation. If you have a fifteen-minute, 100 MB capture window and send 1 MB per second, the size window triggers before the time window. Each partition captures independently and writes a completed block blob at the time of capture, named for the time at which the capture interval was encountered. The storage naming convention is as follows:

```
{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}
```

Note that the date values are padded with zeroes; an example filename might be:

```
https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/  
myeventhub/0/2017/12/08/03/03/17.avro
```

Scaling to throughput units

Event Hubs traffic is controlled by throughput units. A single throughput unit allows 1 MB per second or 1000 events per second of ingress and twice that amount of egress. Standard Event Hubs can be configured with 1-20 throughput units, and you can purchase more with a quota increase support request. Usage beyond your purchased throughput units is throttled. Event Hubs Capture copies data directly from the internal Event Hubs storage, bypassing throughput unit egress quotas and saving your egress for other processing readers, such as Stream Analytics or Spark.

Once configured, Event Hubs Capture runs automatically when you send your first event, and continues running. To make it easier for your downstream processing to know that the process is working, Event Hubs writes empty files when there is no data. This process provides a predictable cadence and marker that can feed your batch processors.

Next

- We'll cover the Azure Event Hubs security model.

Azure Event Hubs authentication and security model

Azure Event Hubs security model:

- Only clients that present valid credentials can send data to an event hub.
- A client cannot impersonate another client.
- A rogue client can be blocked from sending data to an event hub.

Client authentication

The Event Hubs security model is based on a combination of Shared Access Signature (SAS) tokens and *event publishers*. An event publisher defines a virtual endpoint for an event hub. The publisher can only be used to send messages to an event hub. It is not possible to receive messages from a publisher.

Typically, an event hub employs one publisher per client. All messages that are sent to any of the publishers of an event hub are enqueued within that event hub. Publishers enable fine-grained access control and throttling.

Each Event Hubs client is assigned a unique token, which is uploaded to the client. The tokens are produced such that each unique token grants access to a different unique publisher. A client that possesses a token can only send to one publisher, but no other publisher. If multiple clients share the same token, then each of them shares a publisher.

Although not recommended, it is possible to equip devices with tokens that grant direct access to an event hub. Any device that holds this token can send messages directly into that event hub. Such a device will not be subject to throttling. Furthermore, the device cannot be blacklisted from sending to that event hub.

All tokens are signed with a SAS key. Typically, all tokens are signed with the same key. Clients are not aware of the key; this prevents other clients from manufacturing tokens.

Create the SAS key

When creating an Event Hubs namespace, the service automatically generates a 256-bit SAS key named `RootManageSharedAccessKey`. This rule has an associated pair of primary and secondary keys that grant send, listen, and manage rights to the namespace. You can also create additional keys. It is recommended that you produce a key that grants send permissions to the specific event hub. For the remainder of this topic, it is assumed that you named this key *EventHubSendKey*.

The following example creates a send-only key when creating the event hub:

```
// Create namespace manager.
string serviceNamespace = "YOUR_NAMESPACE";
string namespaceManageKeyName = "RootManageSharedAccessKey";
string namespaceManageKey = "YOUR_ROOT_MANAGE_SHARED_ACCESS_KEY";
Uri uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
string.Empty);
TokenProvider td = TokenProvider.CreateSharedAccessSignatureTokenProvid-
```

```

er(namespaceManageKeyName, namespaceManageKey);
NamespaceManager nm = new NamespaceManager(namespaceUri, namespaceManageTokenProvider);

// Create event hub with a SAS rule that enables sending to that event hub
EventHubDescription ed = new EventHubDescription("MY_EVENT_HUB") { PartitionCount = 32 };
string eventHubSendKeyName = "EventHubSendKey";
string eventHubSendKey = SharedAccessAuthorizationRule.GenerateRandomKey();
SharedAccessAuthorizationRule eventHubSendRule = new SharedAccessAuthorizationRule(eventHubSendKeyName, eventHubSendKey, new[] { AccessRights.Send });
ed.Authorization.Add(eventHubSendRule);
nm.CreateEventHub(ed);

```

Generate tokens

You can generate tokens using the SAS key. You must produce only one token per client. Tokens can then be produced using the following method. All tokens are generated using the EventHubSendKey key. Each token is assigned a unique URI.

```

public static string SharedAccessSignatureTokenProvider.GetSharedAccessSignature(string keyName, string sharedAccessKey, string resource, TimeSpan tokenTimeToLive)

```

When calling this method, the URI should be specified as `//<NAMESPACE>.servicebus.windows.net/<EVENT_HUB_NAME>/publishers/<PUBLISHER_NAME>`. For all tokens, the URI is identical, with the exception of PUBLISHER_NAME, which should be different for each token. Ideally, PUBLISHER_NAME represents the ID of the client that receives that token.

This method generates a token with the following structure:

```

SharedAccessSignature sr={URI}&sig={HMAC_SHA256_SIGNATURE}&se={EXPIRATION_TIME}&skn={KEY_NAME}

```

The token expiration time is specified in seconds from Jan 1, 1970. The following is an example of a token:

```

SharedAccessSignature sr=contoso&sig=nPzdNN%2Gli0ifrfJwaK4mkK0RqAB%2by-JULt%2bGFmBHG77A%3d&se=1403130337&skn=RootManageSharedAccessKey

```

Typically, the tokens have a lifespan that resembles or exceeds the lifespan of the client. If the client has the capability to obtain a new token, tokens with a shorter lifespan can be used.

Sending data

Once the tokens have been created, each client is provisioned with its own unique token.

When the client sends data into an event hub, it tags its send request with the token. To prevent an attacker from eavesdropping and stealing the token, the communication between the client and the event hub must occur over an encrypted channel.

Blacklisting clients

If a token is stolen by an attacker, the attacker can impersonate the client whose token has been stolen. Blacklisting a client renders that client unusable until it receives a new token that uses a different publisher.

Authentication of back-end applications

To authenticate back-end applications that consume the data generated by Event Hubs clients, Event Hubs employs a security model that is similar to the model that is used for Service Bus topics. An Event Hubs consumer group is equivalent to a subscription to a Service Bus topic. A client can create a consumer group if the request to create the consumer group is accompanied by a token that grants manage privileges for the event hub, or for the namespace to which the event hub belongs. A client is allowed to consume data from a consumer group if the receive request is accompanied by a token that grants receive rights on that consumer group, the event hub, or the namespace to which the event hub belongs.

The current version of Service Bus does not support SAS rules for individual subscriptions. The same holds true for Event Hubs consumer groups. SAS support will be added for both features in the future.

In the absence of SAS authentication for individual consumer groups, you can use SAS keys to secure all consumer groups with a common key. This approach enables an application to consume data from any of the consumer groups of an event hub.

Next

- A quick tutorial showing how to create an event hub by using the Azure CLI.

Create an event hub using Azure CLI

In this quick tutorial, you create an event hub using Azure CLI in the Azure Cloud Shell. If you choose to install and use Azure CLI locally, this tutorial requires that you are running Azure CLI version 2.0.4 or later. Run `az --version` to check your version.

In this tutorial you will:

1. Sign in to Azure
2. Create a resource group for your event hub
3. Create an Event Hubs namespace
4. Create an event hub

Step 1: Sign in to Azure

The following steps are not required if you're running commands in Cloud Shell. If you're running the CLI locally, perform the following steps to sign in to Azure and set your current subscription:

Run the following command to sign in to Azure:

```
az login
```

Set the current subscription context. Replace `MyAzureSub` with the name of the Azure subscription you want to use:

```
az account set --subscription MyAzureSub
```

Step 2: Create a resource group

A resource group is a logical collection of Azure resources. All resources are deployed and managed in a resource group. Run the following command to create a resource group:

```
# Create a resource group. Specify a name for the resource group.
az group create --name <resource group name> --location eastus
```

Step 3: Create an Event Hubs namespace

An Event Hubs namespace provides a unique scoping container, referenced by its fully qualified domain name, in which you create one or more event hubs. To create a namespace in your resource group, run the following command:

```
# Create an Event Hubs namespace. Specify a name for the Event Hubs namespace.
az eventhubs namespace create --name <Event Hubs namespace> --resource-group <resource group name> -l <region, for example: East US>
```

Step 4: Create an event hub

Run the following command to create an event hub:

```
# Create an event hub. Specify a name for the event hub.
az eventhubs eventhub create --name <event hub name> --resource-group <resource group name> --namespace-name <Event Hubs namespace>
```

Congratulations! You have used Azure CLI to create an Event Hubs namespace, and an event hub within that namespace.

Next

- We'll cover some common scenarios in writing code using Azure Event Hubs. If you want to practice you can use the event hub you just created.

Programming guide for Azure Event Hubs

This section of the lesson discusses some common scenarios in writing code using Azure Event Hubs.

Event publishers

You send events to an event hub either using HTTP POST or via an AMQP 1.0 connection. The choice of which to use and when depends on the specific scenario being addressed. AMQP 1.0 connections are metered as brokered connections in Service Bus and are more appropriate in scenarios with frequent higher message volumes and lower latency requirements, as they provide a persistent messaging channel.

When using the .NET managed APIs, the primary constructs for publishing data to Event Hubs are the `EventHubClient` and `EventData` classes. `EventHubClient` provides the AMQP communication channel over which events are sent to the event hub. The `EventData` class represents an event, and is used to publish messages to an event hub. This class includes the body, some metadata, and header information about the event. Other properties are added to the `EventData` object as it passes through an event hub.

Get started

The .NET classes that support Event Hubs are provided in the `Microsoft.Azure.EventHubs` NuGet package. You can install using the Visual Studio Solution explorer, or the Package Manager Console in Visual Studio. To do so, issue the following command in the Package Manager Console window:

```
Install-Package Microsoft.Azure.EventHubs
```

Create an Event Hubs client

The primary class for interacting with Event Hubs is `Microsoft.Azure.EventHubs.EventHubClient`. You can instantiate this class using the `CreateFromConnectionString` method, as shown in the following example:

```
private const string EventHubConnectionString = "Event Hubs namespace
connection string";
private const string EventHubName = "event hub name";

var connectionStringBuilder = new EventHubsConnectionStringBuilder(Event
HubConnectionString)
{
    EntityPath = EventHubName
};
eventHubClient = EventHubClient.CreateFromConnectionString(connection-
StringBuilder.ToString());
```

Send events to an event hub

You send events to an event hub by creating an `EventHubClient` instance and sending it asynchronously via the `SendAsync` method. This method takes a single `EventData` instance parameter and asynchronously sends it to an event hub.

Event serialization

The `EventData` class has two overloaded constructors that take a variety of parameters, bytes or a byte array, that represent the event data payload. When using JSON with `EventData`, you can use `Encoding.UTF8.GetBytes()` to retrieve the byte array for a JSON-encoded string. For example:

```
for (var i = 0; i < numMessagesToSend; i++)
{
    var message = $"Message {i}";
    Console.WriteLine($"Sending message: {message}");
}
```



```
await eventHubClient.SendAsync(new EventData(Encoding.UTF8.GetBytes(message)));
}
```

Partition key

When sending event data, you can specify a value that is hashed to produce a partition assignment. You specify the partition using the `PartitionSender.PartitionID` property. However, the decision to use partitions implies a choice between availability and consistency.

Availability considerations

Using a partition key is optional, and you should consider carefully whether or not to use one. If you don't specify a partition key when publishing an event, a round-robin assignment is used. In many cases, using a partition key is a good choice if event ordering is important. When you use a partition key, these partitions require availability on a single node, and outages can occur over time; for example, when compute nodes reboot and patch. As such, if you set a partition ID and that partition becomes unavailable for some reason, an attempt to access the data in that partition will fail. If high availability is most important, do not specify a partition key; in that case events are sent to partitions using the round-robin model described previously. In this scenario, you are making an explicit choice between availability (no partition ID) and consistency (pinning events to a partition ID).

Another consideration is handling delays in processing events. In some cases, it might be better to drop data and retry than to try to keep up with processing, which can potentially cause further downstream processing delays. For example, with a stock ticker it's better to wait for complete up-to-date data, but in a live chat or VOIP scenario you'd rather have the data quickly, even if it isn't complete.

Given these availability considerations, in these scenarios you might choose one of the following error handling strategies:

- Stop (stop reading from Event Hubs until things are fixed)
- Drop (messages aren't important, drop them)
- Retry (retry the messages as you see fit)

Batch event send operations

Sending events in batches can help increase throughput. You can use the `CreateBatch` API to create a batch to which data objects can later be added for a `SendAsync` call.

A single batch must not exceed the 1 MB limit of an event. Additionally, each message in the batch uses the same publisher identity. It is the responsibility of the sender to ensure that the batch does not exceed the maximum event size. If it does, a client Send error is generated. You can use the helper method `EventHubClient.CreateBatch` to ensure that the batch does not exceed 1 MB. You get an empty `EventDataBatch` from the `CreateBatch` API and then use `TryAdd` to add events to construct the batch.

Send asynchronously and send at scale

You send events to an event hub asynchronously. Sending asynchronously increases the rate at which a client is able to send events. `SendAsync` returns a `Task` object. You can use the `RetryPolicy` class on the client to control client retry options.

Event consumers

The `EventProcessorHost` class processes data from Event Hubs. You should use this implementation when building event readers on the .NET platform. `EventProcessorHost` provides a thread-safe, multi-process, safe runtime environment for event processor implementations that also provides check-pointing and partition lease management.

To use the `EventProcessorHost` class, you can implement `IEventProcessor`. This interface contains four methods:

- `OpenAsync`
- `CloseAsync`
- `ProcessEventsAsync`
- `ProcessErrorAsync`

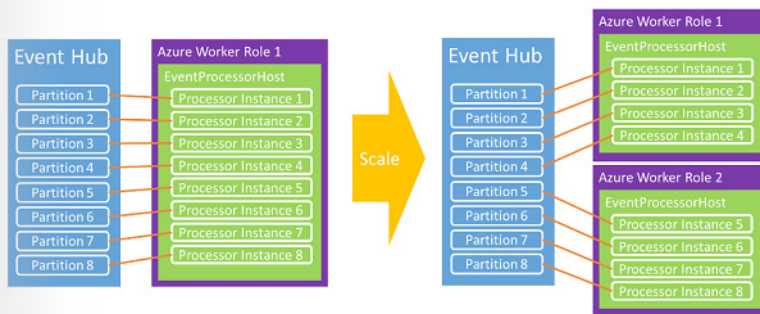
To start event processing, instantiate `EventProcessorHost`, providing the appropriate parameters for your event hub. For example:

```
var eventProcessorHost = new EventProcessorHost(
    EventHubName,
    PartitionReceiver.DefaultConsumerGroupName,
    EventHubConnectionString,
    StorageConnectionString,
    StorageContainerName);
```

Then, call `RegisterEventProcessorAsync` to register your `IEventProcessor` implementation with the runtime:

```
await eventProcessorHost.RegisterEventProcessorAsync<SimpleEventProcessor>();
```

At this point, the host attempts to acquire a lease on every partition in the event hub using a “greedy” algorithm. These leases last for a given timeframe and must then be renewed. As new nodes, worker instances in this case, come online, they place lease reservations and over time the load shifts between nodes as each attempts to acquire more leases.



Over time, an equilibrium is established. This dynamic capability enables CPU-based autoscaling to be applied to consumers for both scale-up and scale-down. Because Event Hubs does not have a direct concept of message counts, average CPU utilization is often the best mechanism to measure back end or consumer scale. If publishers begin to publish more events than consumers can process, the CPU increase on consumers can be used to cause an auto-scale on worker instance count.

The `EventProcessorHost` class also implements an Azure storage-based checkpointing mechanism. This mechanism stores the offset on a per partition basis, so that each consumer can determine what the last checkpoint from the previous consumer was. As partitions transition between nodes via leases, this is the synchronization mechanism that facilitates load shifting.

Publisher revocation

In addition to the advanced run-time features of `EventProcessorHost`, Event Hubs enables publisher revocation in order to block specific publishers from sending event to an event hub. These features are useful if a publisher token has been compromised, or a software update is causing them to behave inappropriately. In these situations, the publisher's identity, which is part of their SAS token, can be blocked from publishing events.

Next

- something

Implement solutions that use Azure Notification Hubs

Azure Notification Hubs overview

Azure Notification Hubs provide an easy-to-use and scaled-out push engine that allows you to send notifications to any platform (iOS, Android, Windows, Kindle, Baidu, etc.) from any backend (cloud or on-premises).

What are push notifications?

Push notifications is a form of app-to-user communication where users of mobile apps are notified of certain desired information, usually in a pop-up or dialog box. Users can generally choose to view or dismiss the message. Choosing the former opens the mobile application that communicated the notification.

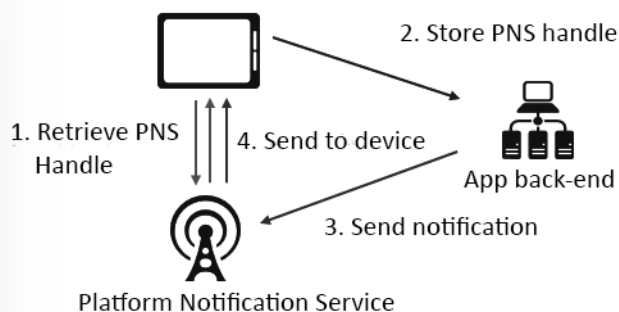
Push notifications are vital for consumer apps in increasing app engagement and usage, and for enterprise apps in communicating up-to-date business information. It's the best app-to-user communication because it is energy-efficient for mobile devices, flexible for the notifications senders, and available when corresponding applications are not active.

How push notifications work

Push notifications are delivered through platform-specific infrastructures called Platform Notification Systems (PNSes). They offer barebone push functionalities to deliver a message to a device with a provided handle, and have no common interface. To send a notification to all customers across the iOS, Android, and Windows versions of an app, the developer must work with Apple Push Notification Service (APNS), Firebase Cloud Messaging (FCM), and Windows Notification Service (WNS).

At a high level, here is how push works:

1. The client app decides it wants to receive notification. Hence, it contacts the corresponding PNS to retrieve its unique and temporary push handle. The handle type depends on the system (for example, WNS has URIs while APNS has tokens).
2. The client app stores this handle in the app back-end or provider.
3. To send a push notification, the app back-end contacts the PNS using the handle to target a specific client app.
4. The PNS forwards the notification to the device specified by the handle.



5.

The challenges of push notifications

PNSes are powerful. However, they leave much work to the app developer to implement even common push notification scenarios, such as broadcasting push notifications to segmented users.

Pushing notifications requires complex infrastructure that is unrelated to the application's main business logic. Some of the infrastructural challenges are:

- **Platform dependency**
 - The backend needs to have complex and hard-to-maintain platform-dependent logic to send notifications to devices on various platforms as PNSes are not unified.
- **Scale**
 - Per PNS guidelines, device tokens must be refreshed upon every app launch. The backend is dealing with a large amount of traffic and database access just to keep the tokens up-to-date. When the number of devices grows to hundreds and thousands of millions, the cost of creating and maintaining this infrastructure is massive.
 - Most PNSes do not support broadcast to multiple devices. A simple broadcast to a million devices results in a million calls to the PNSes. Scaling this amount of traffic with minimal latency is nontrivial.
- **Routing**
 - Though PNSes provide a way to send messages to devices, most apps notifications are targeted at users or interest groups. The backend must maintain a registry to associate devices with interest groups, users, properties, etc. This overhead adds to the time to market and maintenance costs of an app.

Next

- We'll examine the security model of Azure Notification Hubs.

Azure Notification Hubs security model

This topic describes the security model of Azure Notification Hubs. Because Notification Hubs are a Service Bus entity, they implement the same security model as Service Bus.

Shared Access Signature Security (SAS)

Notification Hubs implements an entity-level security scheme called SAS (Shared Access Signature). This scheme enables messaging entities to declare up to 12 authorization rules in their description that grant rights on that entity.

Each rule contains a name, a key value (shared secret), and a set of rights, as explained below. When creating a Notification Hub, two rules are automatically created: one with Listen rights (that the client app uses) and one with all rights (that the app backend uses).

When performing registration management from client apps, if the information sent via notifications is not sensitive (for example, weather updates), a common way to access a Notification Hub is to give the key value of the rule Listen-only access to the client app, and to give the key value of the rule full access to the app backend.

It is not recommended that you embed the key value in Windows Store client apps. A way to avoid embedding the key value is to have the client app retrieve it from the app backend at startup.

It is important to understand that the key with Listen access allows a client app to register for any tag. If your app must restrict registrations to specific tags to specific clients (for example, when tags represent user IDs), then your app backend must perform the registrations. Note that in this way, the client app will not have direct access to Notification Hubs.

Security claims

Similar to other entities, Notification Hub operations are allowed for three security claims: Listen, Send, and Manage.

Claim	Description	Operations allowed
Listen	Create/Update, Read, and Delete single registrations	<ul style="list-style-type: none">• Create/Update registration• Read registration• Read all registrations for a handle• Delete registration
Send	Send messages to the notification hub	<ul style="list-style-type: none">• Send message
Manage	CRUDs on Notification Hubs (including updating PNS credentials, and security keys), and read registrations based on tags	<ul style="list-style-type: none">• Create/Update/Read/Delete notification hubs• Read registrations by tag

Notification Hubs accept claims granted by Microsoft Azure Access Control tokens, and by signature tokens generated with shared keys configured directly on the Notification Hub.

Next

- In our next topic we'll cover device registration and management in Azure Notification Hubs.

Registration management

This topic explains how to register devices with notification hubs in order to receive push notifications. The topic describes registrations at a high level, then introduces the two main patterns for registering devices: registering from the device directly to the notification hub, and registering through an application backend.

What is device registration

Device registration with a Notification Hub is accomplished using a Registration or Installation.

Registrations

A registration associates the Platform Notification Service (PNS) handle for a device with tags and possibly a template. The PNS handle could be a ChannelURI, device token, or GCM registration id. Tags are used to route notifications to the correct set of device handles. (We'll be covering tags for registration in more detail in the next topic in this lesson.)

Note: Azure Notification Hubs supports a maximum of 60 tags per registration.

Installations

An Installation is an enhanced registration that includes a bag of push related properties. It is the latest and best approach to registering your devices. However, it is not supported by client-side .NET SDK (Notification Hub SDK for backend operations) as of yet. This means if you are registering from the client device itself, you would have to use the Notification Hubs REST API approach to support installations. If you are using a backend service, you should be able to use Notification Hub SDK for backend operations.

The following are some key advantages to using installations:

- Creating or updating an installation is fully idempotent. So you can retry it without any concerns about duplicate registrations.
- The installation model makes it easy to do individual pushes - targeting specific device. A system tag `$InstallationId:[installationId]` is automatically added with each installation based registration. So you can call a send to this tag to target a specific device without having to do any additional coding.
- Using installations also enables you to do partial registration updates. The partial update of an installation is requested with a `PATCH` method using the JSON-Patch standard. This is useful when you want to update tags on the registration. You don't have to pull down the entire registration and then resend all the previous tags again.

Below is a JavaScript example showing some of the available properties for an installation. For a complete listing of the installation properties, see **Create or Overwrite an Installation with REST API**¹ or **Installation Properties**².

```
// Example installation format to show some supported properties
{
  installationId: "",
  expirationTime: "",
  tags: [],
  platform: "",
  pushChannel: "",
  .....
  templates: {
    "templateName1" : {
      body: "",
      tags: [] },
    "templateName2" : {
      body: "",
      // Headers are for Windows Store only
      headers: {
        "X-WNS-Type": "wns/tile" }
      tags: [] }
  },
  secondaryTiles: {
    "tileId1": {
      pushChannel: "",
      tags: [],
```

¹ <https://msdn.microsoft.com/library/azure/mt621153.aspx>

² https://msdn.microsoft.com/library/azure/microsoft.azure.notificationhubs.installation_properties.aspx

```

        templates: {
            "otherTemplate": {
                bodyTemplate: "",
                headers: {
                    ... }
                tags: [] }
        }
    }
}

```

Note: By default, registrations and installations do not expire.

Registrations and installations must contain a valid PNS handle for each device/channel. Because PNS handles can only be obtained in a client app on the device, one pattern is to register directly on that device with the client app. On the other hand, security considerations and business logic related to tags might require you to manage device registration in the app back-end.

Templates

If you want to use Templates, the device installation also holds all templates associated with that device in a JSON format (see sample above). The template names help target different templates for the same device.

Each template name maps to a template body and an optional set of tags. Moreover, each platform can have additional template properties. For Windows Store (using WNS) and Windows Phone 8 (using MPNS), an additional set of headers can be part of the template. In the case of APNs, you can set an expiry property to either a constant or to a template expression.

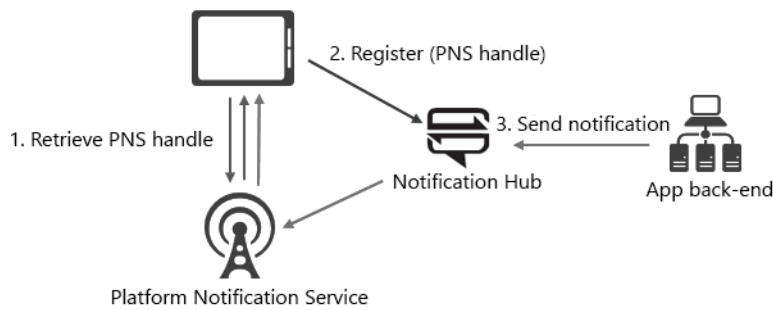
Secondary Tiles for Windows Store Apps

For Windows Store client applications, sending notifications to secondary tiles is the same as sending them to the primary one. This is also supported in installations. Secondary tiles have a different ChannelUri, which the SDK on your client app handles transparently.

The SecondaryTiles dictionary uses the same TileId that is used to create the SecondaryTiles object in your Windows Store app. As with the primary ChannelUri, ChannelUris of secondary tiles can change at any moment. In order to keep the installations in the notification hub updated, the device must refresh them with the current ChannelUris of the secondary tiles.

Registration management from the device

When managing device registration from client apps, the backend is only responsible for sending notifications. Client apps keep PNS handles up-to-date, and register tags. The following picture illustrates this pattern.



The device first retrieves the PNS handle from the PNS, then registers with the notification hub directly. After the registration is successful, the app backend can send a notification targeting that registration. We'll provide more information about how to send notifications in the next topic in the lesson.

In this case, you use only Listen rights to access your notification hubs from the device.

Registering from the device is the simplest method, but it has some drawbacks:

- A client app can only update its tags when the app is active. For example, if a user has two devices that register tags related to sport teams, when the first device registers for an additional tag (for example, Seahawks), the second device will not receive the notifications about the Seahawks until the app on the second device is executed a second time. More generally, when tags are affected by multiple devices, managing tags from the backend is a desirable option.
- Since apps can be hacked, securing the registration to specific tags requires extra care, as explained in the section "Tag-level security."

Example code to register with a notification hub from a device using an installation

At this time, this is only supported using the Notification Hubs REST API.

You can also use the `PATCH` method using the JSON-Patch standard for updating the installation.

```

class DeviceInstallation
{
    public string installationId { get; set; }
    public string platform { get; set; }
    public string pushChannel { get; set; }
    public string[] tags { get; set; }
}

private async Task<HttpStatusCode> CreateOrUpdateInstallationAsync(DeviceInstallation deviceInstallation,
    string hubName, string listenConnectionString)
{
    if (deviceInstallation.installationId == null)
        return HttpStatusCode.BadRequest;

    // Parse connection string (https://msdn.microsoft.com/library/azure/dn495627.aspx)
    ConnectionStringUtility connectionSaSUtil = new ConnectionStringUtili-
```



```

ty(listenConnectionString);
    string hubResource = "installations/" + deviceInstallation.installa-
tionId + "?";
    string apiVersion = "api-version=2015-04";

    // Determine the targetUri that we will sign
    string uri = connectionSaSUtil.Endpoint + hubName + "/" + hubResource +
apiVersion;

    ///== Generate SaS Security Token for Authorization header ==
    // See, https://msdn.microsoft.com/library/azure/dn495627.aspx
    string SasToken = connectionSaSUtil.getSasToken(uri, 60);

    using (var httpClient = new HttpClient())
    {
        string json = JsonConvert.SerializeObject(deviceInstallation);

        httpClient.DefaultRequestHeaders.Add("Authorization", SasToken);

        var response = await httpClient.PutAsync(uri, new StringContent(j-
son, System.Text.Encoding.UTF8, "application/json"));
        return response.StatusCode;
    }
}

var channel = await PushNotificationChannelManager.CreatePushNotification-
ChannelForApplicationAsync();

string installationId = null;
var settings = ApplicationData.Current.LocalSettings.Values;

// If we have not stored a installation id in application data, create and
store as application data.
if (!settings.ContainsKey("__NHInstallationId"))
{
    installationId = Guid.NewGuid().ToString();
    settings.Add("__NHInstallationId", installationId);
}

installationId = (string)settings["__NHInstallationId"];

var deviceInstallation = new DeviceInstallation
{
    installationId = installationId,
    platform = "wns",
    pushChannel = channel.Uri,
    //tags = tags.ToArray<string>()
};

var statusCode = await CreateOrUpdateInstallationAsync(deviceInstallation,
"<HUBNAME>", "<SHARED LISTEN CONNECTION STRING>");

```

```

if (statusCode != HttpStatusCode.Accepted)
{
    var dialog = new MessageDialog(statusCode.ToString(), "Registration
failed. Installation Id : " + installationId);
    dialog.Commands.Add(new UICommand("OK"));
    await dialog.ShowAsync();
}
else
{
    var dialog = new MessageDialog("Registration successful using installa-
tion Id : " + installationId);
    dialog.Commands.Add(new UICommand("OK"));
    await dialog.ShowAsync();
}

```

Example code to register with a notification hub from a device using a registration

These methods create or update a registration for the device on which they are called. This means that in order to update the handle or the tags, you must overwrite the entire registration. Remember that registrations are transient, so you should always have a reliable store with the current tags that a specific device needs.

```

// Initialize the Notification Hub
NotificationHubClient hub = NotificationHubClient.CreateClientFromConnection-
String(listenConnString, hubName);

// The Device id from the PNS
var pushChannel = await PushNotificationChannelManager.CreatePushNotifica-
tionChannelForApplicationAsync();

// If you are registering from the client itself, then store this registra-
tion id in device
// storage. Then when the app starts, you can check if a registration id
already exists or not before
// creating.
var settings = ApplicationData.Current.LocalSettings.Values;

// If we have not stored a registration id in application data, store in
application data.
if (!settings.ContainsKey("__NHRegistrationId"))
{
    // make sure there are no existing registrations for this push handle
    (used for iOS and Android)
    string newRegistrationId = null;
    var registrations = await hub.GetRegistrationsByChannelAsync(pushChan-
nel.Uri, 100);
    foreach (RegistrationDescription registration in registrations)
    {

```

```

        if (newRegistrationId == null)
        {
            newRegistrationId = registration.RegistrationId;
        }
        else
        {
            await hub.DeleteRegistrationAsync(registration);
        }
    }

    newRegistrationId = await hub.CreateRegistrationIdAsync();

    settings.Add("__NHRegistrationId", newRegistrationId);
}

string regId = (string)settings["__NHRegistrationId"];

RegistrationDescription registration = new WindowsRegistrationDescription(pushChannel.Uri);
registration.RegistrationId = regId;
registration.Tags = new HashSet<string>(YourTags);

try
{
    await hub.CreateOrUpdateRegistrationAsync(registration);
}
catch (Microsoft.WindowsAzure.Messaging.RegistrationGoneException e)
{
    settings.Remove("__NHRegistrationId");
}

```

Registration management from a backend

Managing registrations from the backend requires writing additional code. The app from the device must provide the updated PNS handle to the backend every time the app starts (along with tags and templates), and the backend must update this handle on the notification hub.

The advantages of managing registrations from the backend include the ability to modify tags to registrations even when the corresponding app on the device is inactive, and to authenticate the client app before adding a tag to its registration.

Example code to register with a notification hub from a backend using an installation

The client device still gets its PNS handle and relevant installation properties as before and calls a custom API on the backend that can perform the registration and authorize tags etc. The backend can leverage the Notification Hub SDK for backend operations.

You can also use the `PATCH` method using the JSON-Patch standard for updating the installation.

```

// Initialize the Notification Hub
NotificationHubClient hub = NotificationHubClient.CreateClientFromConnectionString(listenConnString, hubName);

// Custom API on the backend
public async Task<HttpResponseMessage> Put(DeviceInstallation deviceUpdate)
{
    Installation installation = new Installation();
    installation.InstallationId = deviceUpdate.InstallationId;
    installation.PushChannel = deviceUpdate.Handle;
    installation.Tags = deviceUpdate.Tags;

    switch (deviceUpdate.Platform)
    {
        case "mpns":
            installation.Platform = NotificationPlatform.Mpns;
            break;
        case "wns":
            installation.Platform = NotificationPlatform.Wns;
            break;
        case "apns":
            installation.Platform = NotificationPlatform.Apns;
            break;
        case "gcm":
            installation.Platform = NotificationPlatform.Gcm;
            break;
        default:
            throw new HttpResponseException(HttpStatusCode.BadRequest);
    }

    // In the backend we can control if a user is allowed to add tags
    //installation.Tags = new List<string>(deviceUpdate.Tags);
    //installation.Tags.Add("username:" + username);

    await hub.CreateOrUpdateInstallationAsync(installation);

    return Request.CreateResponse(HttpStatusCode.OK);
}

```

Example code to register with a notification hub from a device using a registration ID

From your app backend, you can perform basic CRUDS operations on registrations. For example:

```

var hub = NotificationHubClient.CreateClientFromConnectionString("{connectionString}", "hubName");

// create a registration description object of the correct type, e.g.

```

```
var reg = new WindowsRegistrationDescription(channelUri, tags);

// Create
await hub.CreateRegistrationAsync(reg);

// Get by id
var r = await hub.GetRegistrationAsync<RegistrationDescription>("id");

// update
r.Tags.Add("myTag");

// update on hub
await hub.UpdateRegistrationAsync(r);

// delete
await hub.DeleteRegistrationAsync(r);
```

The backend must handle concurrency between registration updates. Service Bus offers optimistic concurrency control for registration management. At the HTTP level, this is implemented with the use of ETag on registration management operations. This feature is transparently used by Microsoft SDKs, which throw an exception if an update is rejected for concurrency reasons. The app backend is responsible for handling these exceptions and retrying the update if necessary.

Next

- We'll cover using templates for registrations.

Using templates for registrations

Templates enable a client application to specify the exact format of the notifications it wants to receive. Using templates, an app can realize several different benefits, including the following:

- A platform-agnostic backend
- Personalized notifications
- Client-version independence
- Easy localization

Using templates cross-platform

The standard way to send push notifications is to send, for each notification that is to be sent, a specific payload to platform notification services (WNS, APNS). For example, to send an alert to APNS, the payload is a JSON object of the following form:

```
{"aps": {"alert" : "Hello!" }}
```

To send a similar toast message on a Windows Store application, the XML payload is as follows:

```
<toast>
  <visual>
    <binding template=\"ToastText01\">
```

```

        <text id=\"1\">Hello!</text>
    </binding>
</visual>
</toast>

```

You can create similar payloads for MPNS (Windows Phone) and FCM (Android) platforms.

This requirement forces the app backend to produce different payloads for each platform, and effectively makes the backend responsible for part of the presentation layer of the app. Some concerns include localization and graphical layouts (especially for Windows Store apps that include notifications for various types of tiles).

The Notification Hubs template feature enables a client app to create special registrations, called template registrations, which include, in addition to the set of tags, a template. The Notification Hubs template feature enables a client app to associate devices with templates whether you are working with Installations (preferred) or Registrations. Given the preceding payload examples, the only platform-independent information is the actual alert message (Hello!). A template is a set of instructions for the Notification Hub on how to format a platform-independent message for the registration of that specific client app. In the preceding example, the platform-independent message is a single property: `message = Hello!`.

The template for the iOS client app registration is as follows:

```

{"aps": {"alert": "${message}"}}

```

The corresponding template for the Windows Store client app is:

```

<toast>
    <visual>
        <binding template=\"ToastText01\">
            <text id=\"1\">${message}</text>
        </binding>
    </visual>
</toast>

```

Notice that the actual message is substituted for the expression `${message}`. This expression instructs the Notification Hub, whenever it sends a message to this particular registration, to build a message that follows it and switches in the common value.

If you are working with Installation model, the installation “templates” key holds a JSON of multiple templates. If you are working with Registration model, the client application can create multiple registrations in order to use multiple templates; for example, a template for alert messages and a template for tile updates. Client applications can also mix native registrations (registrations with no template) and template registrations.

The Notification Hub sends one notification for each template without considering whether they belong to the same client app. This behavior can be used to translate platform-independent notifications into more notifications. For example, the same platform-independent message to the Notification Hub can be seamlessly translated in a toast alert and a tile update, without requiring the backend to be aware of it. Some platforms (for example, iOS) might collapse multiple notifications to the same device if they are sent in a short period of time.

Template expression language

Templates are limited to XML or JSON document formats. Also, you can only place expressions in particular places; for example, node attributes or values for XML, string property values for JSON.

The following table shows the language allowed in templates:

Expression	Description
<code>\$(prop)</code>	Reference to an event property with the given name. Property names are not case-sensitive. This expression resolves into the property's text value or into an empty string if the property is not present.
<code>\$(prop, n)</code>	As above, but the text is explicitly clipped at <i>n</i> characters, for example <code>\$(title, 20)</code> clips the contents of the title property at 20 characters.
<code>.(prop, n)</code>	As above, but the text is suffixed with three dots as it is clipped. The total size of the clipped string and the suffix does not exceed <i>n</i> characters. <code>.(title, 20)</code> with an input property of "This is the title line" results in This is the title...
<code>%(prop)</code>	Similar to <code>\$(name)</code> except that the output is URI-encoded.
<code>#(prop)</code>	Used in JSON templates (for example, for iOS and Android templates). This function works exactly the same as <code>\$(prop)</code> previously specified, except when used in JSON templates (for example, Apple templates). In this case, if this function is not surrounded by <code>"{'','}'</code> (for example, <code>'myJson-Property' : '#(name)'</code>), and it evaluates to a number in Javascript format, for example, regexp: <code>(0 ([1-9][0-9]*))([0-9]+)?((e E)(+ -)?[0-9]+)?</code> , then the output JSON is a number. For example, <code>'badge: '#(name)'</code> becomes <code>'badge' : 40</code> (and not <code>'40'</code>).
<code>'text' or "text"</code>	A literal. Literals contain arbitrary text enclosed in single or double quotes.
<code>expr1 + expr2</code>	The concatenation operator joining two expressions into a single string.

The expressions can be any of the preceding forms.

When using concatenation, the entire expression must be surrounded with `{}`. For example, `{$(prop) + ' - ' + $(prop2)}`.

For example, the following template is **not** a valid XML template:

```
<tile>
  <visual>
    <binding $(property)>
      <text id="1">Seattle, WA</text>
    </binding>
  </visual>
</tile>
```

As explained earlier, when using concatenation, expressions must be wrapped in curly brackets. For example:

```
<tile>
  <visual>
    <binding template="ToastText01">
      <text id="1">{'Hi, ' + $(name)}</text>
    </binding>
  </visual>
</tile>
```

Next

- In the next topic we'll be covering routing and tag expressions.

Routing and tag expressions

Tag expressions enable you to target specific sets of devices, or more specifically registrations, when sending a push notification through Notification Hubs.

Targeting specific registrations

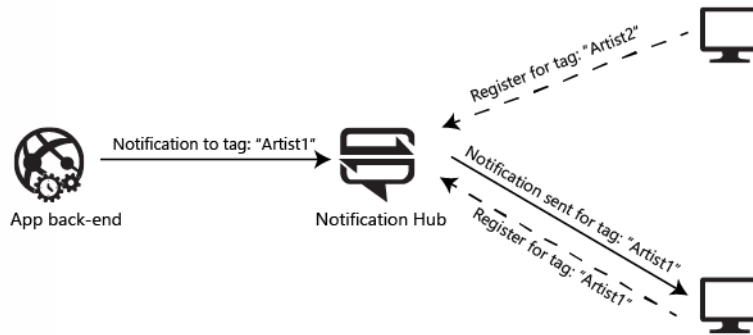
The only way to target specific notification registrations is to associate tags with them, then target those tags. As discussed in Registration Management, in order to receive push notifications an app has to register a device handle on a notification hub. Once a registration is created on a notification hub, the application backend can send push notifications to it. The application backend can choose the registrations to target with a specific notification in the following ways:

1. **Broadcast:** all registrations in the notification hub receive the notification.
2. **Tag:** all registrations that contain the specified tag receive the notification.
3. **Tag expression:** all registrations whose set of tags match the specified expression receive the notification.

Tags

A tag can be any string, up to 120 characters, containing alphanumeric and the following non-alphanumeric characters: `'_', '@', '#', '.', ':', '-'`. The following example shows an application from which you can

receive toast notifications about specific music groups. In this scenario, a simple way to route notifications is to label registrations with tags that represent the different artists, as in the following picture:



In this picture, the message tagged Artist1 reaches only the client that registered with the tag Artist1.

You can send notifications to tags using the send notifications methods of the `Microsoft.Azure.NotificationHubs.NotificationHubClient` class in the Microsoft Azure Notification Hubs SDK. You can also use Node.js, or the Push Notifications REST APIs. Here's an example using the SDK.

```

Microsoft.Azure.NotificationHubs.NotificationOutcome outcome = null;

// Windows 8.1 / Windows Phone 8.1
var toast = @"<toast><visual><binding template=""ToastText01""><text
id=""1"">" +
"You requested a Artist1 notification</text></binding></visual></toast>";
outcome = await Notifications.Instance.Hub.SendWindowsNativeNotificationAsyn-
c(toast, "Artist1");

// Windows 10
toast = @"<toast><visual><binding template=""ToastGeneric""><text
id=""1"">" +
"You requested an Artist1 notification</text></binding></visual></toast>";
outcome = await Notifications.Instance.Hub.SendWindowsNativeNotificationAsyn-
c(toast, "Artist1");
  
```

Tags do not have to be pre-provisioned and can refer to multiple app-specific concepts. For example, users of this example application can comment on bands and want to receive toasts, not only for the comments on their favorite bands, but also for all comments from their friends, regardless of the band on which they are commenting.

Tag expressions

There are cases in which a notification has to target a set of registrations that is identified not by a single tag, but by a Boolean expression on tags.

Consider a sports application that sends a reminder to everyone in Anytown about a game between the HomeTeam and VisitingTeam. If the client app registers tags about interest in teams and location, then the notification should be targeted to everyone in Anytown who is interested in either the HomeTeam or the VisitingTeam. This condition can be expressed with the following Boolean expression:

```
(follows_HomeTeam || follows_VisitingTeam) && location_Anytown )
```

Tag expressions can contain all Boolean operators, such as AND (&&), OR (||), and NOT (!). They can also contain parentheses. Tag expressions are limited to 20 tags if they contain only ORs; otherwise they are limited to 6 tags.

Here's an example for sending notifications with tag expressions using the SDK.

```
Microsoft.Azure.NotificationHubs.NotificationOutcome outcome = null;

String userTag = "(location_AnyTown && !follows_HomeTeam)";

// Windows 8.1 / Windows Phone 8.1
var toast = @"<toast><visual><binding template=""ToastText01""><text
id=""1"">" +
"You want info on the HomeTeam</text></binding></visual></toast>";
outcome = await Notifications.Instance.Hub.SendWindowsNativeNotificationAsyn-
c(toast, userTag);

// Windows 10
toast = @"<toast><visual><binding template=""ToastGeneric""><text
id=""1"">" +
"You want info on the HomeTeam</text></binding></visual></toast>";
outcome = await Notifications.Instance.Hub.SendWindowsNativeNotificationAsyn-
c(toast, userTag);
```

Review questions

Module 4 review questions

Event-driven architecture

Modern application requirements stipulate that applications we build should be able to handle a high volume and velocity of data, process that data in real time, and allow multiple systems to respond to the same data. To help handle these application scenarios, many modern systems are built using an architectural style referred to as event-driven architecture. What are some of the common implementations of an event-driven architecture that you will commonly see on the Microsoft Azure platform?

> Click to see suggested answer

- **Simple event processing.** An event immediately triggers an action in the consumer. For example, you could use Azure Functions with an Azure Service Bus trigger so that a function executes whenever a message is published to a Service Bus topic.
- **Complex event processing.** A consumer processes a series of events, looking for patterns in the event data, by using a technology such as Azure Stream Analytics or Apache Storm. For example, you could aggregate readings from an embedded device over a time window and generate a notification if the moving average crosses a certain threshold.
- **Event stream processing.** You use a data streaming platform, such as Azure IoT Hub or Apache Kafka, as a pipeline to ingest events and feed them to stream processors. The stream processors act to process or transform the stream. There may be multiple stream processors for different subsystems of the application. This approach is a good fit for Internet of Things (IoT) workloads.

Azure Event Hubs

Azure Event Hubs is a big data streaming platform and event ingestion service, capable of receiving and processing millions of events per second. What key components does Event Hubs contain?

> Click to see suggested answer

- **Event producers:** Any entities that send data to an event hub. Event publishers can publish events using HTTPS or Advanced Message Queuing Protocol (AMQP) 1.0 or Apache Kafka (1.0 and above).
- **Partitions:** Each consumer only reads a specific subset, or partition, of the message stream.
- **Consumer groups:** Views (state, position, or offset) of an entire event hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream and to each read the stream independently at its own pace and with its own offsets.
- **Throughput units:** Pre-purchased units of capacity that control the throughput capacity of Event Hubs.
- **Event receivers:** Any entities that read event data from event hubs. All Event Hubs consumers connect via the AMQP 1.0 session, and events are delivered through the session as they become available.



Module 5 Develop message-based solutions

Implement solutions that use Azure Service Bus

Azure Service Bus overview

Microsoft Azure Service Bus is a fully managed enterprise integration message broker. Service Bus is most commonly used to decouple applications and services from each other, and is a reliable and secure platform for asynchronous data and state transfer. Data is transferred between different applications and services using messages. A message is in binary format, which can contain JSON, XML, or just text.

Some common messaging scenarios are:

- Messaging: transfer business data, such as sales or purchase orders, journals, or inventory movements.
- Decouple applications: improve reliability and scalability of applications and services (client and service do not have to be online at the same time).
- Topics and subscriptions: enable 1:n relationships between publishers and subscribers.
- Message sessions: implement workflows that require message ordering or message deferral.

Namespaces

A namespace is a scoping container for all messaging components. Multiple queues and topics can reside within a single namespace, and namespaces often serve as application containers.

Queues

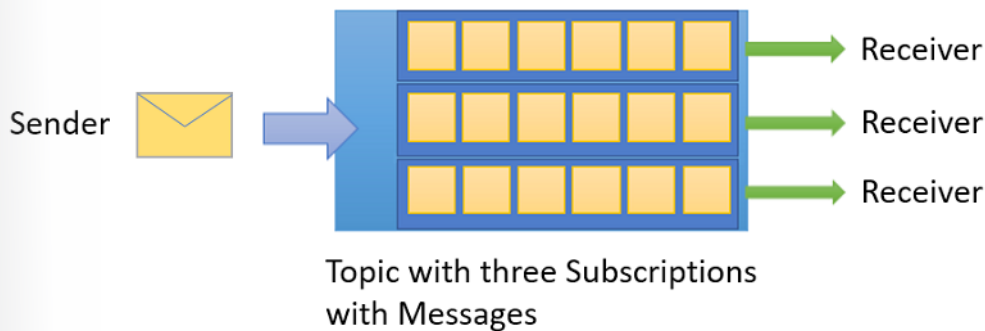
Messages are sent to and received from queues. Queues enable you to store messages until the receiving application is available to receive and process them.



Messages in queues are ordered and timestamped on arrival. Once accepted, the message is held safely in redundant storage. Messages are delivered in pull mode, which delivers messages on request.

Topics

You can also use topics to send and receive messages. While a queue is often used for point-to-point communication, topics are useful in publish/subscribe scenarios.



Topics can have multiple, independent subscriptions. A subscriber to a topic can receive a copy of each message sent to that topic. Subscriptions are named entities, which are durably created but can optionally expire or auto-delete.

In some scenarios, you may not want individual subscriptions to receive all messages sent to a topic. If so, you can use rules and filters to define conditions that trigger optional actions, filter specified messages, and set or modify message properties.

Advanced features

Service Bus also has advanced features that enable you to solve more complex messaging problems. The following table describes these key features:

Advanced Feature	Description
Message sessions	To realize a first-in, first-out (FIFO) guarantee in Service Bus, use sessions. Message sessions enable joint and ordered handling of unbounded sequences of related messages.

Advanced Feature	Description
Auto-forwarding	The auto-forwarding feature enables you to chain a queue or subscription to another queue or topic that is part of the same namespace. When auto-forwarding is enabled, Service Bus automatically removes messages that are placed in the first queue or subscription (source) and puts them in the second queue or topic (destination).
Dead-lettering	Service Bus supports a dead-letter queue (DLQ) to hold messages that cannot be delivered to any receiver, or messages that cannot be processed. You can then remove messages from the DLQ and inspect them.
Scheduled delivery	You can submit messages to a queue or topic for delayed processing; for example, to schedule a job to become available for processing by a system at a certain time.
Message deferral	When a queue or subscription client receives a message that it is willing to process, but for which processing is not currently possible due to special circumstances within the application, the entity has the option to defer retrieval of the message to a later point. The message remains in the queue or subscription, but it is set aside.
Batching	Client-side batching enables a queue or topic client to delay sending a message for a certain period of time. If the client sends additional messages during this time period, it transmits the messages in a single batch.
Transactions	A transaction groups two or more operations together into an execution scope. Service Bus supports grouping operations against a single messaging entity (queue, topic, subscription) within the scope of a transaction.

Advanced Feature	Description
Filtering and actions	Subscribers can define which messages they want to receive from a topic. These messages are specified in the form of one or more named subscription rules. For each matching rule condition, the subscription produces a copy of the message, which may be differently annotated for each matching rule.
Auto-delete on idle	Auto-delete on idle enables you to specify an idle interval after which the queue is automatically deleted. The minimum duration is 5 minutes.
Duplicate detection	If an error occurs that causes the client to have any doubt about the outcome of a send operation, duplicate detection takes the doubt out of these situations by enabling the sender to re-send the same message, and the queue or topic discards any duplicate copies.
SAS, RBAC, and Managed identities	Service Bus supports security protocols such as Shared Access Signatures (SAS), Role Based Access Control (RBAC) and Managed identities for Azure resources.
Geo-disaster recovery	When Azure regions or datacenters experience downtime, Geo-disaster recovery enables data processing to continue operating in a different region or datacenter.
Security	Service Bus supports standard AMQP 1.0 and HTTP/REST protocols.

Next

- In the next topic we'll compare event vs. messaging services.

Event vs. message services

Azure offers three services that assist with delivering event messages throughout a solution. These services are:

- Event Grid
- Event Hubs
- Service Bus

Although they have some similarities, each service is designed for particular scenarios, and there's an important distinction to note between services that deliver an event and services that deliver a message.

Event

An event is a lightweight notification of a condition or a state change. The publisher of the event has no expectation about how the event is handled. The consumer of the event decides what to do with the notification. Events can be discrete units or part of a series.

Discrete events report state change and are actionable. To take the next step, the consumer only needs to know that something happened. The event data has information about what happened but doesn't have the data that triggered the event. For example, an event notifies consumers that a file was created. It may have general information about the file, but it doesn't have the file itself. Discrete events are ideal for serverless solutions that need to scale.

Series events report a condition and are analyzable. The events are time-ordered and interrelated. The consumer needs the sequenced series of events to analyze what happened.

Message

A message is raw data produced by a service to be consumed or stored elsewhere. The message contains the data that triggered the message pipeline. The publisher of the message has an expectation about how the consumer handles the message. A contract exists between the two sides. For example, the publisher sends a message with the raw data, and expects the consumer to create a file from that data and send a response when the work is done.

Comparison of services

Service	Purpose	Type	When to use
Event Grid	Reactive programming	Event distribution (discrete)	React to status changes
Event Hubs	Big data pipeline	Event streaming (series)	Telemetry and distributed data streaming
Service Bus	High-value enterprise messaging	Message	Order processing and financial transactions

Next

- We'll take a look at Service Bus queues, topics, and subscriptions.

Service Bus queues, topics, and subscriptions

Microsoft Azure Service Bus supports a set of cloud-based, message-oriented middleware technologies including reliable message queuing and durable publish/subscribe messaging. These “brokered” messaging capabilities can be thought of as decoupled messaging features that support publish-subscribe, temporal decoupling, and load balancing scenarios using the Service Bus messaging workload. Decoupled communication has many advantages; for example, clients and servers can connect as needed and perform their operations in an asynchronous fashion.

The messaging entities that form the core of the messaging capabilities in Service Bus are queues, topics and subscriptions, and rules/actions.

Queues

Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue, and only one message consumer receives and processes each message. A key benefit of using queues is to achieve “temporal decoupling” of application components. In other words, the producers (senders) and consumers (receivers) do not have to be sending and receiving messages at the same time, because messages are stored durably in the queue. Furthermore, the producer does not have to wait for a reply from the consumer in order to continue to process and send messages.

A related benefit is “load leveling,” which enables producers and consumers to send and receive messages at different rates. In many applications, the system load varies over time; however, the processing time required for each unit of work is typically constant. Intermediating message producers and consumers with a queue means that the consuming application only has to be provisioned to be able to handle average load instead of peak load. The depth of the queue grows and contracts as the incoming load varies. This capability directly saves money with regard to the amount of infrastructure required to service the application load. As the load increases, more worker processes can be added to read from the queue. Each message is processed by only one of the worker processes. Furthermore, this pull-based load balancing allows for optimum use of the worker computers even if the worker computers differ with regard to processing power, as they pull messages at their own maximum rate. This pattern is often termed the “competing consumer” pattern.

Using queues to intermediate between message producers and consumers provides an inherent loose coupling between the components. Because producers and consumers are not aware of each other, a consumer can be upgraded without having any effect on the producer.

Create queues

You create queues using the Azure portal, PowerShell, CLI, or Resource Manager templates. You then send and receive messages using a `QueueClient` object.

Receive modes

You can specify two different modes in which Service Bus receives messages: *ReceiveAndDelete* or *PeekLock*.

In the **ReceiveAndDelete** mode, the receive operation is single-shot; that is, when Service Bus receives the request, it marks the message as being consumed and returns it to the application. *ReceiveAndDelete* mode is the simplest model and works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, consider a scenario in which the consumer issues the receive request and then crashes before processing it. Because Service Bus marks the message as being consumed, when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **PeekLock** mode, the receive operation becomes two-stage, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives the request, it finds the next message to be consumed, locks it to prevent other consumers from receiving it, and then returns it to the application. After the application finishes processing the message (or stores it reliably for future processing), it completes the second stage of the receive process by calling `CompleteAsync` on the received message. When Service Bus sees the `CompleteAsync` call, it marks the message as being consumed.

If the application is unable to process the message for some reason, it can call the `AbandonAsync` method on the received message (instead of `CompleteAsync`). This method enables Service Bus to

unlock the message and make it available to be received again, either by the same consumer or by another competing consumer. Secondly, there is a timeout associated with the lock and if the application fails to process the message before the lock timeout expires (for example, if the application crashes), then Service Bus unlocks the message and makes it available to be received again (essentially performing an `AbandonAsync` operation by default).

In the event that the application crashes after processing the message, but before the `CompleteAsync` request is issued, the message is redelivered to the application when it restarts. This process is often called *At Least Once* processing; that is, each message is processed at least once. However, in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then additional logic is required in the application to detect duplicates, which can be achieved based upon the `MessageId` property of the message, which remains constant across delivery attempts. This feature is known as *Exactly Once* processing.

Topics and subscriptions

In contrast to queues, in which each message is processed by a single consumer, topics and subscriptions provide a one-to-many form of communication, in a publish/subscribe pattern. Useful for scaling to large numbers of recipients, each published message is made available to each subscription registered with the topic. Messages are sent to a topic and delivered to one or more associated subscriptions, depending on filter rules that can be set on a per-subscription basis. The subscriptions can use additional filters to restrict the messages that they want to receive. Messages are sent to a topic in the same way they are sent to a queue, but messages are not received from the topic directly. Instead, they are received from subscriptions. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the topic. Messages are received from a subscription identically to the way they are received from a queue.

By way of comparison, the message-sending functionality of a queue maps directly to a topic and its message-receiving functionality maps to a subscription. Among other things, this feature means that subscriptions support the same patterns described earlier in this section with regard to queues: competing consumer, temporal decoupling, load leveling, and load balancing.

Create topics and subscriptions

Creating a topic is similar to creating a queue, as described in the previous section. You then send messages using the `TopicClient` class. To receive messages, you create one or more subscriptions to the topic. Similar to queues, messages are received from a subscription using a `SubscriptionClient` object instead of a `QueueClient` object. Create the subscription client, passing the name of the topic, the name of the subscription, and (optionally) the receive mode as parameters.

Rules and actions

In many scenarios, messages that have specific characteristics must be processed in different ways. To enable this processing, you can configure subscriptions to find messages that have desired properties and then perform certain modifications to those properties. While Service Bus subscriptions see all messages sent to the topic, you can only copy a subset of those messages to the virtual subscription queue. This filtering is accomplished using subscription filters. Such modifications are called filter actions. When a subscription is created, you can supply a filter expression that operates on the properties of the message, both the system properties (for example, `Label`) and custom application properties (for example, `StoreName`.) The SQL filter expression is optional in this case; without a SQL filter expression, any filter action defined on a subscription will be performed on all the messages for that subscription.

Next

- In the next topic we'll cover messages, payloads, and serialization.

Messages, payloads, and serialization in Azure Service Bus

Messages carry a payload as well as metadata, in the form of key-value pair properties, describing the payload and giving handling instructions to Service Bus and applications. Occasionally, that metadata alone is sufficient to carry the information that the sender wants to communicate to receivers, and the payload remains empty.

The object model of the official Service Bus clients for .NET and Java reflect the abstract Service Bus message structure, which is mapped to and from the wire protocols Service Bus supports.

A Service Bus message consists of a binary payload section that Service Bus never handles in any form on the service-side, and two sets of properties. The broker properties are predefined by the system. These predefined properties either control message-level functionality inside the broker, or they map to common and standardized metadata items. The user properties are a collection of key-value pairs that can be defined and set by the application.

The predefined broker properties are listed in the following table. The names are used with all official client APIs and also in the `BrokerProperties` JSON object of the HTTP protocol mapping.

The equivalent names used at the AMQP protocol level are listed in parentheses.

Property Name	Description
ContentType (content-type)	Optionally describes the payload of the message, with a descriptor following the format of RFC2045, Section 5; for example, application/json.
CorrelationId (correlation-id)	Enables an application to specify a context for the message for the purposes of correlation; for example, reflecting the MessageId of a message that is being replied to.
DeadLetterSource	Only set in messages that have been dead-lettered and subsequently auto-forwarded from the dead-letter queue to another entity. Indicates the entity in which the message was dead-lettered. This property is read-only.
DeliveryCount	Number of deliveries that have been attempted for this message. The count is incremented when a message lock expires, or the message is explicitly abandoned by the receiver. This property is read-only.

Property Name	Description
EnqueuedSequenceNumber	For messages that have been auto-forwarded, this property reflects the sequence number that had first been assigned to the message at its original point of submission. This property is read-only.
EnqueuedTimeUtc	The UTC instant at which the message has been accepted and stored in the entity. This value can be used as an authoritative and neutral arrival time indicator when the receiver does not want to trust the sender's clock. This property is read-only.
ExpiresAtUtc (absolute-expiry-time)	The UTC instant at which the message is marked for removal and no longer available for retrieval from the entity due to its expiration. Expiry is controlled by the TimeToLive property and this property is computed from EnqueuedTimeUtc+TimeToLive. This property is read-only.
ForcePersistence	For queues or topics that have the EnableExpress flag set, this property can be set to indicate that the message must be persisted to disk before it is acknowledged. This is the standard behavior for all non-express entities.
Label (subject)	This property enables the application to indicate the purpose of the message to the receiver in a standardized fashion, similar to an email subject line.
LockedUntilUtc	For messages retrieved under a lock (peek-lock receive mode, not pre-settled) this property reflects the UTC instant until which the message is held locked in the queue/subscription. When the lock expires, the DeliveryCount is incremented and the message is again available for retrieval. This property is read-only.

Property Name	Description
LockToken	<p>The lock token is a reference to the lock that is being held by the broker in peek-lock receive mode.</p> <p>The token can be used to pin the lock permanently through the Deferral API and, with that, take the message out of the regular delivery state flow. This property is read-only.</p>
MessageId (message-id)	<p>The message identifier is an application-defined value that uniquely identifies the message and its payload. The identifier is a free-form string and can reflect a GUID or an identifier derived from the application context. If enabled, the duplicate detection feature identifies and removes second and further submissions of messages with the same MessageId.</p>
PartitionKey	<p>For partitioned entities, setting this value enables assigning related messages to the same internal partition, so that submission sequence order is correctly recorded. The partition is chosen by a hash function over this value and cannot be chosen directly. For session-aware entities, the SessionId property overrides this value.</p>
ReplyTo (reply-to)	<p>This optional and application-defined value is a standard way to express a reply path to the receiver of the message. When a sender expects a reply, it sets the value to the absolute or relative path of the queue or topic it expects the reply to be sent to.</p>
ReplyToSessionId (reply-to-group-id)	<p>This value augments the ReplyTo information and specifies which SessionId should be set for the reply when sent to the reply entity.</p>
ScheduledEnqueueTimeUtc	<p>For messages that are only made available for retrieval after a delay, this property defines the UTC instant at which the message will be logically enqueued, sequenced, and therefore made available for retrieval.</p>

Property Name	Description
SequenceNumber	<p>The sequence number is a unique 64-bit integer assigned to a message as it is accepted and stored by the broker and functions as its true identifier. For partitioned entities, the topmost 16 bits reflect the partition identifier. Sequence numbers monotonically increase and are gapless. They roll over to 0 when the 48-64 bit range is exhausted. This property is read-only.</p>
SessionId (group-id)	<p>For session-aware entities, this application-defined value specifies the session affiliation of the message. Messages with the same session identifier are subject to summary locking and enable exact in-order processing and demultiplexing. For entities that are not session-aware, this value is ignored.</p>
Size	<p>Reflects the stored size of the message in the broker log as a count of bytes, as it counts towards the storage quota. This property is read-only.</p>
State	<p>Indicates the state of the message in the log. This property is only relevant during message browsing ("peek"), to determine whether a message is "active" and available for retrieval as it reaches the top of the queue, whether it is deferred, or is waiting to be scheduled. This property is read-only.</p>
TimeToLive	<p>This value is the relative duration after which the message expires, starting from the instant the message has been accepted and stored by the broker, as captured in EnqueueTimeUtc. When not set explicitly, the assumed value is the DefaultTimeToLive for the respective queue or topic. A message-level TimeToLive value cannot be longer than the entity's DefaultTimeToLive setting. If it is longer, it is silently adjusted.</p>

Property Name	Description
To (to)	This property is reserved for future use in routing scenarios and currently ignored by the broker itself. Applications can use this value in rule-driven auto-forward chaining scenarios to indicate the intended logical destination of the message.
ViaPartitionKey	If a message is sent via a transfer queue in the scope of a transaction, this value selects the transfer queue partition.

The abstract message model enables a message to be posted to a queue via HTTP (actually always HTTPS) and can be retrieved via AMQP. In either case, the message looks normal in the context of the respective protocol. The broker properties are translated as needed, and the user properties are mapped to the most appropriate location on the respective protocol message model. In HTTP, user properties map directly to and from HTTP headers; in AMQP they map to and from the application-properties map.

Message routing and correlation

A subset of the broker properties described previously, specifically `To`, `ReplyTo`, `ReplyToSessionId`, `MessageId`, `CorrelationId`, and `SessionId`, are used to help applications route messages to particular destinations. To illustrate this, consider a few patterns:

- **Simple request/reply:** A publisher sends a message into a queue and expects a reply from the message consumer. To receive the reply, the publisher owns a queue into which it expects replies to be delivered. The address of that queue is expressed in the `ReplyTo` property of the outbound message. When the consumer responds, it copies the `MessageId` of the handled message into the `CorrelationId` property of the reply message and delivers the message to the destination indicated by the `ReplyTo` property. One message can yield multiple replies, depending on the application context.
- **Multicast request/reply:** As a variation of the prior pattern, a publisher sends the message into a topic and multiple subscribers become eligible to consume the message. Each of the subscribers might respond in the fashion described previously. This pattern is used in discovery or roll-call scenarios and the respondent typically identifies itself with a user property or inside the payload. If `ReplyTo` points to a topic, such a set of discovery responses can be distributed to an audience.
- **Multiplexing:** This session feature enables multiplexing of streams of related messages through a single queue or subscription such that each session (or group) of related messages, identified by matching `SessionId` values, are routed to a specific receiver while the receiver holds the session under lock. Read more about the details of sessions [here](#).
- **Multiplexed request/reply:** This session feature enables multiplexed replies, allowing several publishers to share a reply queue. By setting `ReplyToSessionId`, the publisher can instruct the consumer(s) to copy that value into the `SessionId` property of the reply message. The publishing queue or topic does not need to be session-aware. As the message is sent, the publisher can then specifically wait for a session with the given `SessionId` to materialize on the queue by conditionally accepting a session receiver.

Routing inside of a Service Bus namespace can be realized using auto-forward chaining and topic subscription rules. Routing across namespaces can be realized using Azure LogicApps. As indicated in the previous list, the `To` property is reserved for future use and may eventually be interpreted by the broker

with a specially enabled feature. Applications that wish to implement routing should do so based on user properties and not lean on the `To` property; however, doing so now will not cause compatibility issues.

Payload serialization

When in transit or stored inside of Service Bus, the payload is always an opaque, binary block. The `ContentType` property enables applications to describe the payload, with the suggested format for the property values being a MIME content-type description according to IETF RFC2045; for example, `application/json; charset=utf-8`.

Unlike the Java or .NET Standard variants, the .NET Framework version of the Service Bus API supports creating `BrokeredMessage` instances by passing arbitrary .NET objects into the constructor.

When using the legacy SBMP protocol, those objects are then serialized with the default binary serializer, or with a serializer that is externally supplied. When using the AMQP protocol, the object is serialized into an AMQP object. The receiver can retrieve those objects with the `GetBody()` method, supplying the expected type. With AMQP, the objects are serialized into an AMQP graph of `ArrayList` and `IDictionary<string, object>` objects, and any AMQP client can decode them.

While this hidden serialization magic is convenient, applications should take explicit control of object serialization and turn their object graphs into streams before including them into a message, and do the reverse on the receiver side. This yields interoperable results. It should also be noted that while AMQP has a powerful binary encoding model, it is tied to the AMQP messaging ecosystem and HTTP clients will have trouble decoding such payloads.

Next

- A quick tutorial showing you how to create a Service Bus namespace and queue, and how to send and receive messages.

Send and receive messages from a Service Bus queue

This tutorial covers the following steps:

1. Create a Service Bus namespace, and queue, using the Azure CLI.
2. Write a .NET Core console application to send a set of messages to the queue.
3. Write a .NET Core console application to receive those messages from the queue.

The code examples below rely on the **Microsoft.Azure.ServiceBus** NuGet package.

Step 1: Create the Service Bus namespace and queue

Login in to the Azure portal then click the Cloud Shell button on the menu in the upper-right corner of the Azure portal, and from the **Select environment** dropdown, select **Bash**.

In Cloud Shell, from the Bash prompt issue the following commands to provision Service Bus resources. Be sure to replace all placeholders with the appropriate values:

```
# Create a resource group
az group create --name myResourceGroup --location eastus

# Create a Service Bus messaging namespace with a unique name
```



```

namespaceName=myNameSpace$RANDOM
az servicebus namespace create \
  --resource-group myResourceGroup \
  --name $namespaceName \
  --location eastus

# Create a Service Bus queue
az servicebus queue create --resource-group myResourceGroup \
  --namespace-name $namespaceName \
  --name myQueue

# Get the connection string for the namespace
connectionString=$(az servicebus namespace authorization-rule keys list \
  --resource-group myResourceGroup \
  --namespace-name $namespaceName \
  --name RootManageSharedAccessKey \
  --query primaryConnectionString --output tsv)

```

After the last command runs, copy and paste the connection string, and the queue name you selected, to a temporary location such as Notepad. You will need it in the next step.

Step 2: Write code to send messages to the queue

1. In Program.cs, add the following `using` statements at the top of the namespace definition, before the class declaration:

```

using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;

```

2. Within the Program class, declare the following variables. Set the `ServiceBusConnectionString` variable to the connection string that you obtained when creating the namespace, and set `QueueName` to the name that you used when creating the queue:

```

const string ServiceBusConnectionString = "<your_connection_string>";
const string QueueName = "<your_queue_name>";
static IQueueClient queueClient;

```

3. Replace the default contents of `Main()` with the following line of code:

```

MainAsync().GetAwaiter().GetResult();

```

4. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the `send messages` method:

```

static async Task MainAsync()
{
    const int numberOfMessages = 10;
    queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

    Console.Write-

```

```

Line("=====");
    Console.WriteLine("Press ENTER key to exit after sending all the mes-
sages.");
    Console.Write-
Line("=====");

    // Send messages.
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await queueClient.CloseAsync();
}

```

5. Directly after the `MainAsync()` method, add the following `SendMessagesAsync()` method that performs the work of sending the number of messages specified by `numberOfMessagesToSend` (currently set to 10):

```

static async Task SendMessagesAsync(int numberOfMessagesToSend)
{
    try
    {
        for (var i = 0; i < numberOfMessagesToSend; i++)
        {
            // Create a new message to send to the queue.
            string messageBody = $"Message {i}";
            var message = new Message(Encoding.UTF8.GetBytes(messageBody));

            // Write the body of the message to the console.
            Console.WriteLine($"Sending message: {messageBody}");

            // Send the message to the queue.
            await queueClient.SendAsync(message);
        }
    }
    catch (Exception exception)
    {
        Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Mes-
sage}");
    }
}

```

6. Here is what your `Program.cs` file should look like.

```

namespace CoreSenderApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

```

```
class Program
{
    // Connection String for the namespace can be obtained from the
    Azure portal under the
    // 'Shared Access policies' section.
    const string ServiceBusConnectionString = "<your_connection_
string>";
    const string QueueName = "<your_queue_name>";
    static IQueueClient queueClient;

    static void Main(string[] args)
    {
        MainAsync().GetAwaiter().GetResult();
    }

    static async Task MainAsync()
    {
        const int numberOfMessages = 10;
        queueClient = new QueueClient(ServiceBusConnectionString,
QueueName);

        Console.Write-
Line("=====");
        Console.WriteLine("Press ENTER key to exit after sending all
the messages.");
        Console.Write-
Line("=====");

        // Send Messages
        await SendMessagesAsync(numberOfMessages);

        Console.ReadKey();

        await queueClient.CloseAsync();
    }

    static async Task SendMessagesAsync(int numberOfMessagesToSend)
    {
        try
        {
            for (var i = 0; i < numberOfMessagesToSend; i++)
            {
                // Create a new message to send to the queue
                string messageBody = $"Message {i}";
                var message = new Message(Encoding.UTF8.GetBytes(mes-
sageBody));

                // Write the body of the message to the console
                Console.WriteLine($"Sending message: {messageBody}");
            }
        }
        catch { }
    }
}
```

```

        // Send the message to the queue
        await queueClient.SendAsync(message);
    }
}
catch (Exception exception)
{
    Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
}
}
}
}

```

7. Run the program, and check the Azure portal: click the name of your queue in the namespace Overview window. The queue Essentials screen is displayed. Notice that the Active Message Count value for the queue is now 10. Each time you run the sender application without retrieving the messages (as described in the next section), this value increases by 10. Also note that the current size of the queue increments the Current value in the Essentials window each time the app adds messages to the queue.

Step 3: Write code to receive messages to the queue

To receive the messages you just sent, create another .NET Core console application and install the Microsoft.Azure.ServiceBus NuGet package, similar to the previous sender application.

Repeat the first two steps from "Step 2: Write code to send messages to the queue" above. Then:

1. Replace the default contents of `Main()` with the following line of code:

```
MainAsync().GetAwaiter().GetResult();
```

2. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the `RegisterOnMessageHandlerAndReceiveMessages()` method:

```

static async Task MainAsync()
{
    queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

    Console.WriteLine(
        Line("=====");
        Console.WriteLine("Press ENTER key to exit after receiving all the
        messages.");
        Console.WriteLine(
        Line("=====");

    // Register the queue message handler and receive messages in a loop
    RegisterOnMessageHandlerAndReceiveMessages();

    Console.ReadKey();

    await queueClient.CloseAsync();
}

```

3. Directly after the `MainAsync()` method, add the following method that registers the message handler and receives the messages sent by the sender application:

```
static void RegisterOnMessageHandlerAndReceiveMessages()
{
    // Configure the message handler options in terms of exception handling,
    // number of concurrent messages to deliver, etc.
    var messageHandlerOptions = new MessageHandlerOptions(ExceptionRe-
    ceivedHandler)
    {
        // Maximum number of concurrent calls to the callback ProcessMes-
        // sagesAsync(), set to 1 for simplicity.
        // Set it according to how many messages the application wants to
        // process in parallel.
        MaxConcurrentCalls = 1,

        // Indicates whether the message pump should automatically complete
        // the messages after returning from user callback.
        // False below indicates the complete operation is handled by the
        // user callback as in ProcessMessagesAsync().
        AutoComplete = false
    };

    // Register the function that processes messages.
    queueClient.RegisterMessageHandler(ProcessMessagesAsync, message-
    HandlerOptions);
}
```

4. Directly after the previous method, add the following `ProcessMessagesAsync()` method to process the received messages:

```
static async Task ProcessMessagesAsync(Message message, CancellationToken
token)
{
    // Process the message.
    Console.WriteLine($"Received message: SequenceNumber:{message.System-
    Properties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

    // Complete the message so that it is not received again.
    // This can be done only if the queue Client is created in ReceiveMode.
    // PeekLock mode (which is the default).
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellation token passed as necessary to determine if
    // the queueClient has already been closed.
    // If queueClient has already been closed, you can choose to not call
    // CompleteAsync() or AbandonAsync() etc.
    // to avoid unnecessary exceptions.
}
```

5. Finally, add the following method to handle any exceptions that might occur:

```
// Use this handler to examine the exceptions received on the message pump.
static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exception-
ReceivedEventArgs)
{
    Console.WriteLine($"Message handler encountered an exception {excep-
tionReceivedEventArgs.Exception}.");
    var context = exceptionReceivedEventArgs.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
```

6. Here is what your Program.cs file should look like:

```
namespace CoreReceiverApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

    class Program
    {
        // Connection String for the namespace can be obtained from the
        Azure portal under the
        // 'Shared Access policies' section.
        const string ServiceBusConnectionString = "<your_connection_
string>";
        const string QueueName = "<your_queue_name>";
        static IQueueClient queueClient;

        static void Main(string[] args)
        {
            MainAsync().GetAwaiter().GetResult();
        }

        static async Task MainAsync()
        {
            queueClient = new QueueClient(ServiceBusConnectionString,
QueueName);

            Console.Write-
Line("=====");
            Console.WriteLine("Press ENTER key to exit after receiving all
the messages.");
            Console.Write-
Line("=====");
        }
    }
}
```

```

        // Register QueueClient's MessageHandler and receive messages
in a loop
        RegisterOnMessageHandlerAndReceiveMessages();

        Console.ReadKey();

        await queueClient.CloseAsync();
    }

    static void RegisterOnMessageHandlerAndReceiveMessages()
    {
        // Configure the MessageHandler Options in terms of exception
        handling, number of concurrent messages to deliver etc.
        var messageHandlerOptions = new MessageHandlerOptions(Excep-
        tionReceivedHandler)
        {
            // Maximum number of Concurrent calls to the callback
            `ProcessMessagesAsync`, set to 1 for simplicity.
            // Set it according to how many messages the application
            wants to process in parallel.
            MaxConcurrentCalls = 1,

            // Indicates whether MessagePump should automatically
            complete the messages after returning from User Callback.
            // False below indicates the Complete will be handled by
            the User Callback as in `ProcessMessagesAsync` below.
            AutoComplete = false
        };

        // Register the function that will process messages
        queueClient.RegisterMessageHandler(ProcessMessagesAsync, mes-
        sageHandlerOptions);
    }

    static async Task ProcessMessagesAsync(Message message, Cancell-
    ationToken token)
    {
        // Process the message
        Console.WriteLine($"Received message: SequenceNumber:{message.
        SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.
        Body)}");

        // Complete the message so that it is not received again.
        // This can be done only if the queueClient is created in
        ReceiveMode.PeekLock mode (which is default).
        await queueClient.CompleteAsync(message.SystemProperties.
        LockToken);

        // Note: Use the cancellation token passed as necessary to
        determine if the queueClient has already been closed.
        // If queueClient has already been Closed, you may chose to not

```

```

    call CompleteAsync() or AbandonAsync() etc. calls
        // to avoid unnecessary exceptions.
    }

    static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs
exceptionReceivedEventArgs)
    {
        Console.WriteLine($"Message handler encountered an exception
{exceptionReceivedEventArgs.Exception}.");
        var context = exceptionReceivedEventArgs.ExceptionReceivedCon-
text;

        Console.WriteLine("Exception context for troubleshooting:");
        Console.WriteLine($"- Endpoint: {context.Endpoint}");
        Console.WriteLine($"- Entity Path: {context.EntityPath}");
        Console.WriteLine($"- Executing Action: {context.Action}");
        return Task.CompletedTask;
    }
}

```

7. Run the program, and check the portal again. Notice that the **Active Message Count** and **Current** values are now 0.

Next

- We've wrapped up our coverage of Azure Service Bus, in the next lesson we'll cover Azure Queue Storage.

Implement solutions that use Azure Queue Storage queues

Azure Queue storage overview

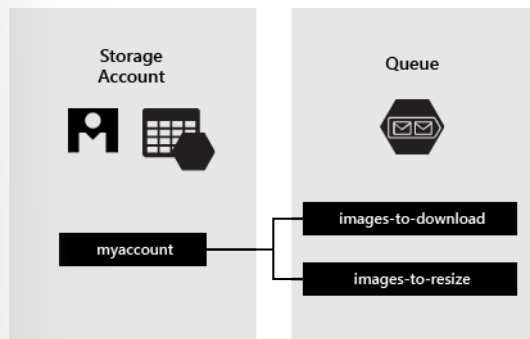
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue service components

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:
`https://<storage account>.queue.core.windows.net/<queue>`
- The following URL addresses a queue in the diagram:
`https://myaccount.queue.core.windows.net/images-to-download`
- **Storage account:** All access to Azure Storage is done through a storage account.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase.
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is seven days.

Next

- We'll show you some code examples to manage messages in Azure Queue storage.

Creating and managing messages in Azure Queue storage

In this topic we'll be covering how to create queues and manage messages in Azure Queue storage by showing code snippets from a Visual Studio project.

Prerequisites

The code examples rely on two NuGet packages:

- **Azure SDK for .NET¹**
- **Microsoft Azure Storage Library for .NET²**

Note: The Storage Client Library package is also included in the Azure SDK for .NET. However, it is recommend that you also install the Storage Client Library from NuGet to ensure that you always have the latest version of the client library.

The ODataLib dependencies in the Storage Client Library for .NET are resolved by the ODataLib packages available on NuGet, not from WCF Data Services. The ODataLib libraries can be downloaded directly, or referenced by your code project through NuGet. The specific ODataLib packages used by the Storage Client Library are OData, Edm, and Spatial. While these libraries are used by the Azure Table storage classes, they are required dependencies for programming with the Storage Client Library.

You'll also need an Azure Storage resource set up and the connection string if you want to try some of the code snippets below on your own.

Storage connection string

The Azure Storage Client Library for .NET supports using a storage connection string to configure endpoints and credentials for accessing storage services. The best way to maintain your storage connection string is in a configuration file.

To configure your connection string, open the app.config file from Solution Explorer in Visual Studio. Add the contents of the <appSettings> element shown below. Replace account-name with the name of your storage account, and account-key with your account access key:

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <appSettings>
    <add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key" />
  </appSettings>
</configuration>
```

For example, your configuration setting appears similar to:

¹ <https://azure.microsoft.com/downloads/>

² <https://www.nuget.org/packages/WindowsAzure.Storage/>

```
<add key="StorageConnectionString" value="DefaultEndpointsProto-
col=https;AccountName=storagesample;AccountKey=GMuzNHj1B3S9itqZJHHCnRk-
rokLkcSyW7yK9BRbGp0ENePunLPwBgpxV1Z/pVo9zpem/2xSHXkMqTHHLcx8XRA==" />
```

Create the Queue service client

The `CloudQueueClient` class enables you to retrieve queues stored in Queue storage. Here's one way to create the service client:

```
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();
```

Now you are ready to write code that reads data from and writes data to Queue storage.

Create a queue

This example shows how to create a queue if it does not already exist:

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a container.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Create the queue if it doesn't already exist
queue.CreateIfNotExists();
```

Insert a message into a queue

To insert a message into an existing queue, first create a new `CloudQueueMessage`. Next, call the `AddMessage` method. A `CloudQueueMessage` can be created from either a string (in UTF-8 format) or a byte array. Here is code which creates a queue (if it doesn't exist) and inserts the message 'Hello, World':

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Create the queue if it doesn't already exist.
queue.CreateIfNotExists();
```

```
// Create a message and add it to the queue.
CloudQueueMessage message = new CloudQueueMessage("Hello, World");
queue.AddMessage(message);
```

Peek at the next message

You can peek at the message in the front of a queue without removing it from the queue by calling the `PeekMessage` method.

```
// Retrieve storage account from connection string
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Peek at the next message
CloudQueueMessage peekedMessage = queue.PeekMessage();

// Display message.
Console.WriteLine(peekedMessage.AsString);
```

Change the contents of a queued message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than *n* times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Get the message from the queue and update the message contents.
CloudQueueMessage message = queue.GetMessage();
message.SetMessageContent("Updated contents.");
```

```
queue.UpdateMessage(message,
    TimeSpan.FromSeconds(60.0), // Make it invisible for another 60 seconds.
    MessageUpdateFields.Content | MessageUpdateFields.Visibility);
```

De-queue the next message

Your code de-queues a message from a queue in two steps. When you call `GetMessage`, you get the next message in a queue. A message returned from `GetMessage` becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call `DeleteMessage`. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls `DeleteMessage` right after the message has been processed.

```
// Retrieve storage account from connection string
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Get the next message
CloudQueueMessage retrievedMessage = queue.GetMessage();

//Process the message in less than 30 seconds, and then delete the message
queue.DeleteMessage(retrievedMessage);
```

Use Async-Await pattern with common Queue storage APIs

This example shows how to use the Async-Await pattern with common Queue storage APIs. The sample calls the asynchronous version of each of the given methods, as indicated by the `Async` suffix of each method. When an async method is used, the async-await pattern suspends local execution until the call completes. This behavior allows the current thread to do other work, which helps avoid performance bottlenecks and improves the overall responsiveness of your application.

```
// Create the queue if it doesn't already exist
if(await queue.CreateIfNotExistsAsync())
{
    Console.WriteLine("Queue '{0}' Created", queue.Name);
}
else
{
    Console.WriteLine("Queue '{0}' Exists", queue.Name);
}
```

```
// Create a message to put in the queue
CloudQueueMessage cloudQueueMessage = new CloudQueueMessage("My message");

// Async enqueue the message
await queue.AddMessageAsync(cloudQueueMessage);
Console.WriteLine("Message added");

// Async dequeue the message
CloudQueueMessage retrievedMessage = await queue.GetMessageAsync();
Console.WriteLine("Retrieved message with content '{0}'", retrievedMessage.
AsString);

// Async delete the message
await queue.DeleteMessageAsync(retrievedMessage);
Console.WriteLine("Deleted message");
```

Leverage additional options for de-queuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses the `GetMessages` method to get 20 messages in one call. Then it processes each message using a `foreach` loop. It also sets the invisibility timeout to five minutes for each message. Note that the 5 minutes starts for all messages at the same time, so after 5 minutes have passed since the call to `GetMessages`, any messages which have not been deleted will become visible again.

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

foreach (CloudQueueMessage message in queue.GetMessages(20, TimeSpan.
FromMinutes(5)))
{
    // Process all messages in less than 5 minutes, deleting each message
    after processing.
    queue.DeleteMessage(message);
}
```

Get the queue length

You can get an estimate of the number of messages in a queue. The `FetchAttributes` method asks the Queue service to retrieve the queue attributes, including the message count. The `ApproximateMes-`

sageCount property returns the last value retrieved by the `FetchAttributes` method, without calling the Queue service.

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Fetch the queue attributes.
queue.FetchAttributes();

// Retrieve the cached approximate message count.
int? cachedMessageCount = queue.ApproximateMessageCount;

// Display number of messages.
Console.WriteLine("Number of messages in queue: " + cachedMessageCount);
```

Delete a queue

To delete a queue and all the messages contained in it, call the `Delete` method on the queue object.

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Delete the queue.
queue.Delete();
```

Next

- This wraps up this module and course.

Review questions

Module 5 review questions

Event vs. message services

There's an important distinction between services that deliver an event and services that deliver a message. Can you describe some key differences?

> Click to see suggested answer

Event

An event is a lightweight notification of a condition or a state change. The publisher of the event has no expectation about how the event is handled. The consumer of the event decides what to do with the notification. Events can be discrete units or part of a series.

Discrete events report state change and are actionable. To take the next step, the consumer only needs to know that something happened. The event data has information about what happened but doesn't have the data that triggered the event. For example, an event notifies consumers that a file was created. It may have general information about the file, but it doesn't have the file itself. Discrete events are ideal for serverless solutions that need to scale.

Series events report a condition and are analyzable. The events are time-ordered and interrelated. The consumer needs the sequenced series of events to analyze what happened.

Message

A message is raw data produced by a service to be consumed or stored elsewhere. The message contains the data that triggered the message pipeline. The publisher of the message has an expectation about how the consumer handles the message. A contract exists between the two sides. For example, the publisher sends a message with the raw data, and expects the consumer to create a file from that data and send a response when the work is done.

Azure Service Bus queues

Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue, and only one message consumer receives and processes each message. What are the two different modes Service Bus receives messages?

> Click to see suggested answer

You can specify two different modes in which Service Bus receives messages: *ReceiveAndDelete* or *PeekLock*.

In the **ReceiveAndDelete** mode, the receive operation is single-shot; that is, when Service Bus receives the request, it marks the message as being consumed and returns it to the application. *ReceiveAndDelete* mode is the simplest model and works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, consider a scenario in which the

consumer issues the receive request and then crashes before processing it. Because Service Bus marks the message as being consumed, when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **PeekLock** mode, the receive operation becomes two-stage, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives the request, it finds the next message to be consumed, locks it to prevent other consumers from receiving it, and then returns it to the application. After the application finishes processing the message (or stores it reliably for future processing), it completes the second stage of the receive process by calling `CompleteAsync` on the received message. When Service Bus sees the `CompleteAsync` call, it marks the message as being consumed.