**Chapter 5. Connect to and consume Azure services and third-party services**

Nowadays, companies use different systems for different tasks that are usually performed by different departments. Although these separate systems work for solving a specific need, they usually act as independent actors in a big scenario. These independent actors manage information about the company that can potentially be duplicated by other independent actors.

When a company realizes that independent actors are managing their data, they usually try to make all the independent actors or systems work together and share information between them. This situation is independent of using cloud services or on-premises services. To make the independent actors work together, you need to make connections between each actor or service that needs to communicate with the other.

You can use different services and techniques to achieve this interconnection. Azure provides some useful services that allow different services to work together without making big changes to the interconnected services.

**Skills covered in this chapter:**

* [Skill 5.1: Develop an App Service Logic App](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev1sec1)
* [Skill 5.2: Implement API Management](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev1sec2)
* [Skill 5.3: Develop event-based solutions](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev1sec3)
* [Skill 5.4: Develop message-based solutions](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev1sec4)

**Skill 5.1: Develop an App Service Logic App**

Exchanging information between different applications is a goal for most companies. Sharing the information enriches the internal process and creates more insight into the information itself. By using the App Service Logic App, you can create workflows that interconnect different systems based on conditions and rules and easing the process of sharing information between them. Also, you can take advantage of the Logic Apps features to implement business process workflows.

**This skill covers how to**

* [Create a Logic App](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec1)
* [Create a custom connector for Logic Apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec2)
* [Create a custom template for Logic Apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec3)

**Create a Logic App**

Before you can interconnect two separate services, you need to fully understand which information you need to share between the services. Sometimes the information needs to undergo some transformations before a service can consume it. You could write code for making this interconnection, but this is a time-consuming and error-prone task.

Azure provides the App Service Logic Apps that allows interconnecting two or more services sharing information between them. A business process defines this interconnection between different services. Azure Logic Apps allows you to build complex interconnection scenarios by using some elements that ease the work:

* **Workflows** Define the source and destination of the information. It connects to different services by using connectors. A workflow defines the steps or actions that the information needs to make to deliver the information from the source to the correct destination. You use a graphical language to visualize, design, build, automate, and deploy a business process.
* **Managed Connectors** A connector is an object that allows your workflow to access data, services, and systems. Microsoft provides some prebuilt connectors to Microsoft services. These connectors are managed by Microsoft and provide the needed triggers and action objects to work with those services.
* **Triggers** Triggers are events that fire when certain conditions are met. You use a trigger as the entry or starting point of a workflow. For example, when a new message arrives at your company’s purchases mailbox, it can start a workflow that can access information from the subject and body of the message and create a new entry in the ERP system.
* **Actions** Actions are each of the steps that you configure in your workflow. Actions happen only when the workflow is executed. The workflow starts executing when a new trigger fires.
* **Enterprise Integration Pack** If you need to perform more advanced integrations, the Enterprise Integration Pack provides you with BizTalk Server capabilities.

***NOTE* AZURE LOGIC APP, AZURE FUNCTIONS, AZURE APP SERVICE WEBJOBS, AND MICROSOFT FLOW**

If you need to implement workflows, Microsoft provides some products that you can use for that task. Although there is some overlap of the features provided by Logic Apps, Functions, App Service WebJobs, and Power Automate, they are designed for different scenarios. You can review more details about the appropriate scenarios for each product at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs)*.*

You can use Azure Logic Apps for different purposes. The most obvious application for Azure Logic Apps would be implementing business processes. Although there is no direct mapping between Azure Logic Apps actions and Business Process Model Notation (BPMN), you can use Logic Apps for automating some simple business processes or integrate it with Business Process Model (BPM) engines for implementing more complex business processes. You also can use Azure Logic Apps for sending notifications when certain events happen or creating the folder and permission structure in a SharePoint Online document library when a project manager of your company creates a new project.

When you are creating a new Azure Logic App workflow, you need to think about how this workflow is going to start. This is the trigger of your workflow. A trigger can be an event that happened in a service, such as a new file has been uploaded to an Azure Storage Account. A workflow can also start based on a schedule. The schedule that you configure for starting a workflow is the trigger for the workflow. When you set your schedule and the appropriate time arrives, the Azure Logic Apps engine creates a new instance of your workflow. You can configure two different types of schedules:

* **Recurrence** In this type of trigger, you configure a regular time interval. You can configure a start date and time, and you can also configure the time zone for your schedule. When you configure the time interval, you can choose from seconds to months as the frequency. For example, you can configure a recurrence of executing the workflow every 2 minutes or every 3 weeks. Depending on the interval that you choose, you can select additional details for that interval. For the *Week* interval, you can select on which days the workflow is going to be executed. For the *Day* or *Week* intervals, you can select the hours or minutes for the execution of your workflow. The Recurrence trigger doesn’t process the missing recurrences. That is, if a recurrence is missing for whatever reason, the Recurrence trigger doesn’t restart the missing recurrence.
* **Sliding Window** This type of trigger is similar to the Recurrence trigger, except you cannot configure advance scheduling settings, such as specific days in the week or hours or minutes in a day. Another essential difference is that with the Sliding Window trigger, if a recurrence is missing, the Sliding Window trigger goes back and processes the missing recurrence.

Scheduling the execution of your Azure Logic App is not the only way to start your workflow. You can use other triggers for starting the execution of the workflow. In addition to the recurrence triggers that we reviewed previously in this section, you can use two additional types for triggers:

* **Polling** The trigger queries the configured system or service periodically for new data or if a new event happened. Depending on the specific trigger, you can configure the polling schedule. Once the new data or event happens, the trigger creates a new instance of your workflow, collects the information from the system, and passes the information to the newly created workflow instance.
* **Push** The trigger listens for new events or data to arrive at the configured system or service. As soon as the new data or event happens, the trigger creates a new instance of your workflow, passing the data to the newly created instance.

Once the workflow starts, it needs to follow some steps for doing the work that it’s supposed to do. Each of these steps is an action. An action can be something like getting data from an OData service reading information from a text file stored in an SFTP service, or even transforming the format of a file. There is also another kind of action that is as important as the data gathering or data transformation actions. Setting the value of a variable, loops, conditional or switch statements, or decision branching are other kinds of actions that is critical for defining your workflow. As with any other programming language, these structural actions enable you to control the execution flow of your workflow.

Triggers and actions are packaged together into connectors. You use the connectors for accessing data, events, and actions available from other applications or services. Azure Logic Apps offers thousands of connectors, but all of them fit in one of the following categories:

* **Built-in** This type of connector contains the fundamental triggers and actions available in Azure Logic Apps. Some of the connectors that fit in this category allow you to schedule the execution of the workflow, call other Azure Logic Apps or App Services, make HTTP and HTTPS calls to endpoints, process messages in batches, or make your Logic App callable from other services.
* **Managed** These connectors are developed, deployed, and maintained by Microsoft. You use these connectors for accessing cloud services like Office 365, Azure Blob Storage, SharePoint, and many others.
* **On-Premises** You use this kind of connector when you need your Azure Logic App to work with systems deployed in your on-premises infrastructure. Using these connectors, you can access data from File Systems, Oracle, MySQL, PostgreSQL, Microsoft SQL Server, IBM DB2, IBM Informix, or Teradata databases. For these connectors to work properly, you need to deploy an on-premises data gateway.
* **Integration Account** Allows you to connect your Azure Logic App with third-party business partners for creating Business-to-Business (B2B) solutions. Integrations Accounts are available only through the Enterprise Integration Pack (EIP) in Azure. You use this kind of connector for transforming the messages between the different B2B systems. You can apply AS2, EDIFACT, X12, or Flat files decoding or encoding, Liquid and XML transformations, or XML validations.
* **ISE** These are the connectors that you need to use when your Azure Logic App needs to run in an Integration Service Environment (ISE). This is a dedicated environment where you execute your Azure Logic Apps. There are special connectors designed for working in an ISE. 245Those connectors are marked with the label CORE if they are built-in connectors that you can use in an ISE or if they are managed connectors that you can use in an ISE, you will see the ISE label below the name of the connector. When using an ISE you are not limited to ISE connectors; you can also use regular connectors in an Integration Service Environment.

Additionally, to the classification that we reviewed in the previous list, the connectors can be classified as Standard or Enterprise connectors. This classification is essential because it directly affects the costs associated with running your Azure Logic App workflow. When you run a workflow, you are charged when you use the workflow, except when your workflow is running inside an ISE.

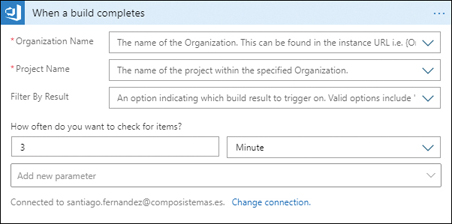
When you run your Azure Logic App workflow in the public, multitenant, global environment, you pay only for the actions that your workflow runs. An action is any of the steps that configure your workflow. Triggers, loops, conditional statements, or any of the control actions count for the calculation of the costs of your workflow execution. There are three pricing levels:

* **Basic** This level includes built-in connectors, triggers, and control workflow actions.
* **Standard** This includes the actions defined in managed connectors. Any custom connector that you create also fits into this category.
* **Enterprise** These are specialized connectors for integrating B2B applications with your Azure Logic App. Some examples of enterprise connectors are SAP, IBM 3270, or IBM MQ.

It is also important to note that n-premises connectors can be Standard or Enterprise connectors. In general, when you are calculating the costs of your Azure Logic App, you should review the list of Standard or Enterprise connectors for having an accurate idea of which type of connector you are using in your workflow. You can find a complete list of connectors by reviewing the article at [*https://docs.microsoft.com/en-us/connectors/connector-reference/*](https://docs.microsoft.com/en-us/connectors/connector-reference/).

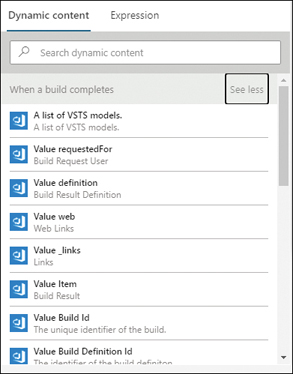
Now that you have a basic understanding of the different parts of an Azure Logic App workflow it is time to create your own workflow. The following procedure shows how to create an Azure Logic App workflow that writes a message in the Microsoft Teams app when a new build completes in Azure DevOps. For this procedure, you need an Azure DevOps account with a configured project that you can build. If you don’t have an Azure DevOps account yet, you can create one by following the quick start guide at [*https://docs.microsoft.com/en-us/azure/devops/user-guide/sign-up-invite-teammates?view=azure-devops*](https://docs.microsoft.com/en-us/azure/devops/user-guide/sign-up-invite-teammates?view=azure-devops). You also need a Microsoft Office 365 subscription with access to the Microsoft Teams application. Start by creating and configuring the Azure Logic App:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click the Create A Resource button at the top of the Azure portal.
3. On the New blade, on the Azure Marketplace list at the left side of the blade, click Integration.
4. In the Featured column on the right side of the blade, click Logic App.
5. On the Logic App blade, on the Resource Group drop-down menu, select the resource group where you want to create your Azure Logic App. Alternatively, you can create a new resource group by clicking the Create New link below the resource group drop-down menu.
6. Type a name in the Logic App Name text box.
7. In the Select The Location control, ensure that the Region option is selected.
8. Select a location from the Location drop-down menu.
9. Leave the Log Analytics option set to the Off value.
10. Click the Review + Create button at the bottom of the blade.
11. Click the Create button at the bottom of the blade.
12. In the Microsoft.EmptyWorkflow deployment window, click the Go To Resource button. This button appears once the deployment of your new Azure Logic App finishes successfully.
13. On the Logic Apps blade, in the Logic App Designer, click the Blank Logic App in the Templates section.
14. In the Logic Apps Designer, in the Search Connectors And Triggers text box, type **Azure DevOps**.
15. Click the Azure DevOps icon on the Results panel.
16. On the Triggers tab, select the trigger named When A Build Completes.
17. Click the Sign In button on the Azure DevOps element. At this point, you connect your Azure Subscription with your Azure DevOps account.
18. On the panel for the When A Build Completes trigger, shown in [Figure 5-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig01), select your Organization’s name from the Organization Name drop-down menu.



**Figure 5-1** Configuring an Azure Logic Apps trigger

1. On the Project Name drop-down menu, select the name of the project that you want.
2. Click the New Step button below the trigger panel.
3. On the Choose An Action panel, type **Teams** in the Search Connectors And Actions text box.
4. On the Actions tab, click the Post A Message (V3) action.
5. On the Post A Message (V3) action panel, select a team from the Team drop-down menu.
6. Select General from the Channel drop-down menu.
7. In the Message text area, type **The***.*
8. In the Dynamic Content dialog box, shown in [Figure 5-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig02), on the right side of the Message text area, click the See More link. This link shows the list of dynamic attributes that you can add to your message.



**Figure 5-2** Dynamic content from a connector trigger

1. Scroll down the list of dynamic attributes and click the Value Build Definition Name dynamic attribute.
2. Type **build finished with status** in the message area next to the dynamic attribute.
3. Click the Value Status dynamic attribute.
4. Click the Save button on the top-left corner of the Azure Logic Apps Designer blade.
5. Click the Run button at the top-left corner of the Azure Logic Apps Designer blade.

At this point, the Azure Logic Apps start listening for the configured trigger in Azure DevOps. When a new build finishes in Azure DevOps, Azure Logic Apps sends a message to the General channel in the configured team in your Microsoft Teams account. Now, you are going to create a pipeline in Azure DevOps for building an example project. Once the build finishes, you are going to receive a new message in the Microsoft Teams channel:

1. Navigate to the following GitHub example repo [*https://github.com/MicrosoftDocs/pipelines-dotnet-core*](https://github.com/MicrosoftDocs/pipelines-dotnet-core) and sign in to your account. If you don’t have a GitHub account, you can create a new one for free at [*https://github.com/join*](https://github.com/join)*.*
2. At the top-right corner of the project’s page, click the Fork button.
3. Open your Azure DevOps account ([*https://dev.azure.com*](https://dev.azure.com/)). If you see the generic Azure DevOps page describing the service instead of your Azure DevOps account, click the Sign In to Azure DevOps link below the Start Free button.
4. In the list of projects in your Azure DevOps account, click the project name that you configured in step 19 of the previous procedure.
5. Click the Pipelines element on the navigation bar on the left side of the project’s page.
6. Click the Create Pipeline button.
7. On the New Pipeline window, click GitHub.
8. Sign in to your GitHub account.
9. On the Authorize Azure Pipelines (OAuth) window, click the Authorize Azure Pipelines button at the bottom of the page.
10. In your Azure DevOps account, on the Select A Repository page, click the pipelines-dotnet-core project.
11. In your Azure DevOps account, on the Configure Your Pipeline page, click ASP.NET Core.
12. On the Review Your Pipeline YAML page, review the details for your pipeline.
13. Click the Run button at the top-right corner of the page.
14. Once the pipeline has been created, a page with the details of the execution of your pipeline appears. If everything works correctly, you should see a job with the success status in the list of Jobs in the Job panel.

At this point, the Azure DevOps agent is building the sample project. Once the build finishes, you should receive a new message in your Microsoft Teams channel, as shown in [Figure 5-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig03).

This is a screenshot of a message in a Microsoft Teams channel. The message shows the status of a finished build in Azure DevOps.

**Figure 5-3** A message in Microsoft Teams from Azure DevOps

***NEED MORE REVIEW?* AZURE LOGIC APP PRICING**

Calculating the costs associated with an Azure Logic App workflow can be complicated. Each of the iterations in a loop is one or more action executions. The different connector types are charged differently, depending on whether they are Basic, Standard, or Enterprise. A workflow executed in an Integration Service Environment has its own pricing. You can read an article with the details about how Azure calculates how much you have to pay for the execution of your workflow at [*https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-pricing*](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-pricing)*.*

**Images *EXAM TIP***

When you are working with Integration Service Environments (ISE), you have specific connectors that run inside the ISE. These special ISE connectors are marked with a label. Although ISE 249environments use dedicated resources for your environment, you still can use global, public, multitenant connectors, like Office 365 or Dropbox connectors.

**Create a custom connector for Logic Apps**

Microsoft provides more than 200 built-in connectors that you can use in your Azure Logic Apps workflow. Despite this number of connectors, there are times when you need some specific features that are not provided by the built-in or managed connectors, or you want to create a connector for your company’s application.

You can create custom connectors for Microsoft PowerAutomate (formerly known as Flow), PowerApps, and Azure Logic Apps. Although you cannot share Azure Logic Apps connectors with Microsoft PowerAutomate and PowerApps connectors, the principle for creating custom connectors are the same for the three platforms. A custom connector is basically a wrapper for a REST or SOAP API. This wrapper allows Azure Logic Apps to interact with the API of the application. The application that you want to include in the custom connector can be a public application, like Amazon Web Services, Google Calendar, or the API of your application published to the Internet. Using the on-premises data gateway, you can also connect the custom connector with an on-premises application deployed in your data center. Every custom connector has the following life cycle:

1. **Build your API** You can wrap any REST or SOAP API in a custom connector. If you are creating your API, you should consider using Azure Functions, Azure Web Apps, or Azure API Apps.
2. **Secure your API** You need to authenticate the access to your API. If you are implementing your application using Azure Functions, Azure Web Apps, or Azure API Apps, you can enable the Azure Active Directory authentication in the Azure portal for your application. You can also enforce authentication directly on your API’s code. You can use any of the following authentication mechanisms:
   * Generic OAuth 2.0
   * OAuth 2.0 for specific services, like Azure Active Directory, Dropbox, GitHub, or SalesForce
   * Basic Authentication
   * API Key
3. **Describe the API and define the custom connector** You need to provide a description of the different endpoints that your API has. Azure Logic Apps supports two different language-agnostic, machine-readable document formats that you can use for documenting this description: OpenAPI (formerly known as Swagger) or Postman collections. You can create a custom connector from the OpenAPI or Postman collection documentation.
4. **Use the connector in an Azure Logic Apps** Once you have created the custom connector, you can use it as a regular managed built-in connector in your workflow. You need to create a connection to your API using your custom connector. Then you can use the triggers and actions that you configured in your custom connector.
5. **Share your connector** Once you have created your custom connector, you can share it with other users in your organization. This step is optional.
6. **Certify your connector** If you want to share your custom connector with other users outside your organization, you need to send the custom connector to Microsoft. Then Microsoft can review your custom connector to ensure that it works correctly. Once the connector is reviewed and validated, Microsoft certifies it, and you can share it with users outside your organization.

Now that you have reviewed the life cycle of an Azure Logic App Custom Connector, you are going to create a custom connector for connecting with an API. For this example, you are going to create a simple Web API 2 application that simulates a book-managing system. Although this example is quite simple, it covers some key points that you need to consider when creating an Azure Logic App Custom connector. The API that you are going to implement in this example is not appropriate for production environments because it doesn’t take into consideration important aspects like performance or security. Use the following steps for creating the API that you are going to use for your custom connector:

1. Open Visual Studio 2019 on your computer.
2. In the start window, click Create A New Project in the Get Started column on the right side of the window.
3. In the Create A New Project window, from the All Languages drop-down menu, select C#.
4. On the Search For Templates text box type **asp.net**.
5. On the result list, click ASP.NET Web Application (.NET Framework).
6. Click the Next button at the bottom right corner of the window.
7. In the Configure Your New Project window, type a Project Name, a Location, and a Solution Name for your project.
8. Click the Create button at the bottom-right corner of the window.
9. In the Create A New ASP.NET Web Application window, select the Web API template on the template list in the middle of the left side of the window.
10. On the right side of the Create A New ASP.NET Web Application window, in the Authentication section, ensure the Authentication is set to No Authentication.
11. Click the Create button on the bottom-right corner of the window.
12. In the Visual Studio window, click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
13. On the NuGet Package Manager tab, click Browse.
14. Type **swashbuckle** and press Enter.
15. Click the Swashbuckle package.
16. On the right side of the NuGet Manager tab, click the check box next to your project.
17. Click the Install button.
18. In the Preview Changes window, click OK.
19. In the License Acceptance window, click the I Accept button.
20. Repeat steps 13 to 19 and install the TRex NuGet package.
21. On the Solution Explorer window, right-click the Models folder.
22. On the contextual menu, click Add > New Item.
23. In the Add New Item window, select Class from the list of new items.
24. In the name text box at the bottom of the window, type **Book.cs**.
25. Click the Add button at the bottom-right corner of the window.
26. Replace the content of the Book.cs file with the content in [Listing 5-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex01).

**Listing 5-1** *Book.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-1a)

**// C# .NET**

using System;

using TRex.Metadata;

namespace *<replace\_with\_your\_project\_name>*.Models

{

public class Book

{

public Book()

{

this.Id = Guid.NewGuid();

}

[Metadata("Callback ID", Visibility = VisibilityType.Internal)]

public Guid Id { get; }

[Metadata("Title", "The title of the book")]

public string Title { set; get; }

[Metadata("Author", "The author of the book")]

public string Author { set; get; }

}

}

1. Repeat steps 21 to 25 and create the class **Callback.cs**.
2. Replace the content of the Callback.cs file with the content in [Listing 5-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex02).

**Listing 5-2** *Callback.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-2a)

**// C# .NET**

using System;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Threading.Tasks;

using TRex.Metadata;

using Newtonsoft.Json.Linq;

namespace *<replace\_with\_your\_project\_name>*.Models

{

public class Callback

{

public Callback()

{

this.Id = Guid.NewGuid();

}

[Metadata("Callback ID", Visibility = VisibilityType.Internal)]

public Guid Id { get; }

[CallbackUrl]

[Metadata("Callback URL", Visibility = VisibilityType.Internal)]

public Uri Uri { set; get; }

public HttpResponseMessage InvokeAsync<TOutput>(TOutput triggerOutput)

{

HttpClient httpClient = new HttpClient();

httpClient.DefaultRequestHeaders.Accept.Clear();

httpClient.DefaultRequestHeaders.Accept.Add(

new MediaTypeWithQualityHeaderValue("application/json"));

return Task.Run(async () => await httpClient.PostAsJsonAsync(Uri,

JObject.FromObject(triggerOutput))).Result;

}

}

}

1. On the Solution Explorer window, right-click the name of your project. On the contextual menu, click Add > New Folder.
2. Type **Helpers** as the name of the new folder.
3. Repeat steps 21 to 25 and create two additional classes in the Helpers folder. The two classes should be **BooksSingleton.cs** and **CallbacksSingleton.cs**.
4. Replace the content of file BooksSingleton.cs with the content of [Listing 5-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex03).

**Listing 5-3** *BooksSingleton.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-3a)

**// C# .NET**

using *<replace\_with\_your\_project\_name>.*Models;

using System;

using System.Collections.Generic;

using System.Linq;

namespace *<replace\_with\_your\_project\_name>*.Helpers

{

public sealed class BooksSingleton

{

private static readonly BooksSingleton m\_instance = null;

private static readonly List<Book> \_books;

public static BooksSingleton Instance

{

get

{

return m\_instance;

}

}

static BooksSingleton()

{

m\_instance = new BooksSingleton();

\_books = new List<Book>();

}

private BooksSingleton()

{

}

public void AddBook(Book book)

{

if (book != null)

{

\_books.Add(book);

}

}

public void ModifyBook(Book book)

{

var bookToModify = \_books.SingleOrDefault(b => b.Id.Equals(book.Id));

if (bookToModify != null)

{

bookToModify.Author = book.Author;

bookToModify.Title = book.Title;

}

}

public IEnumerable<Book> GetBooks()

{

return \_books.ToArray();

}

public bool DeleteBookById(string id)

{

bool deleted = false;

Guid guidToRemove = Guid.Parse(id);

var booktToRemove = \_books.SingleOrDefault(b => b.Id.Equals(guidToRemove));

if (booktToRemove != null)

{

\_books.Remove(booktToRemove);

deleted = true;

}

return deleted;

}

public Book GetBookById(string id)

{

Guid guid = Guid.Parse(id);

return \_books.SingleOrDefault(b => b.Id.Equals(guid));

}

}

}

1. Replace the content of the file CallbacksSingleton.cs with the content in [Listing 5-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex04).

**Listing 5-4** *CallbacksSingleton.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-4a)

**// C# .NET**

using *<replace\_with\_your\_project\_name>.*Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace *<replace\_with\_your\_project\_name>.*Helpers

{

public class CallbacksSingleton

{

private static readonly CallbacksSingleton m\_instance = null;

private static readonly List<Callback> \_callbacks;

public static CallbacksSingleton Instance

{

get

{

return m\_instance;

}

}

static CallbacksSingleton()

{

m\_instance = new CallbacksSingleton();

\_callbacks = new List<Callback>();

}

private CallbacksSingleton()

{

}

public void AddCallback(Callback callback)

{

if (callback != null)

{

//avoid duplicates

Callback callbackToBeAdded = \_callbacks.FirstOrDefault(c => Uri.

Compare(c.Uri, callback.Uri, UriComponents.AbsoluteUri, UriFormat.

Unescaped, StringComparison.CurrentCultureIgnoreCase) == 0);

if (callbackToBeAdded == null)

\_callbacks.Add(callback);

}

}

public void ModifyCallback(Callback callback)

{

var callbackToModify = \_callbacks.SingleOrDefault(b =>

b.Id.Equals(callback.Id));

if (callbackToModify != null)

{

}

}

public IEnumerable<Callback> GetCallbacks()

{

return \_callbacks;

}

public bool DeleteCallbackById(string id)

{

bool deleted = false;

Guid guidToRemove = Guid.Parse(id);

var callbackToRemove = \_callbacks.SingleOrDefault(b =>

b.Id.Equals(guidToRemove));

if (callbackToRemove != null)

{

\_callbacks.Remove(callbackToRemove);

deleted = true;

}

return deleted;

}

public Callback GetCallbackById(string id)

{

Guid guid = Guid.Parse(id);

return \_callbacks.SingleOrDefault(b => b.Id.Equals(guid));

}

}

}

1. On the Solution Explorer window, expand the Controllers folders and remove the ValuesController.cs file.
2. Right-click the Controllers folder and then click Add > Controller.
3. In the Add New Scaffolded Item, select Web API 2 Controller – Empty.
4. Click the Add button on the bottom right corner of the window.
5. In the Add Controller window, type **BooksController** in the Controller Name text box.
6. Click the Add button.
7. Replace the content of the BooksController.cs file with the content of the [Listing 5-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex05).

**Listing 5-5** *BooksController.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-5a)

**// C# .NET**

using *<replace\_with\_your\_project\_name>.*Models;

*using <replace\_with\_your\_project\_name>.*Helpers;

using Swashbuckle.Swagger.Annotations;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Threading.Tasks;

using System.Web.Http;

using TRex.Metadata;

namespace *<replace\_with\_your\_project\_name>.*Controllers

{

public class BooksController : ApiController

{

private readonly BooksSingleton \_books = BooksSingleton.Instance;

private static readonly CallbacksSingleton \_callbacks = CallbacksSingleton.

Instance;

// Subscribe to newly created books

[HttpPost, Route("books/subscribe")]

[Metadata("New book created", "Fires whenever a new book is added to the

list.", VisibilityType.Important)]

[Trigger(TriggerType.Subscription, typeof(Book), "Book")]

[SwaggerResponseRemoveDefaults]

[SwaggerResponse(HttpStatusCode.Created, "Subscription created")]

[SwaggerResponse(HttpStatusCode.BadRequest, "Invalid subscription

configuration")]

public IHttpActionResult Subscribe(Callback callback)

{

\_callbacks.AddCallback(callback);

return CreatedAtRoute(nameof(Unsubscribe), new { subscriptionId = callback.

Id }, string.Empty);

}

[HttpDelete, Route("books/subscribe/{callbackId}", Name = nameof

(Unsubscribe))]

[Metadata("Unsubscribe", Visibility = VisibilityType.Internal)]

[SwaggerResponse(HttpStatusCode.OK)]

public IHttpActionResult Unsubscribe(string callbackID)

{

\_callbacks.DeleteCallbackById(callbackID);

return Ok();

}

[HttpGet, Route("books/subscriptions")]

[Metadata("Get subscriptions", "Get all the current subscriptions")]

[SwaggerResponse(HttpStatusCode.OK, "An array of subscriptions",

typeof(Array))]

public IEnumerable<Callback> GetCallbacks()

{

return \_callbacks.GetCallbacks();

}

// GET api/books

[HttpGet, Route("books")]

[Metadata("Get books", "Get all the books objects stored in the App")]

[SwaggerResponse(HttpStatusCode.OK, "An array of books", typeof(Array))]

public IEnumerable<Book> Get()

{

return \_books.GetBooks();

}

// GET api/books/5

[HttpGet, Route("books/{id}", Name = "GetBook")]

[Metadata("Get single book", "Get a single book object by its id. You can use

any GUID valid string")]

[SwaggerResponse(HttpStatusCode.OK, "An object represeting a book",

typeof(Book))]

public Book Get(string id)

{

return \_books.GetBookById(id);

}

// POST api/books

[HttpPost, Route("books")]

[Metadata("Add a new book", "Add a new book object to the system. A value

object is compound of a Title and an Author.")]

[SwaggerResponse(HttpStatusCode.Created)]

public IHttpActionResult Post([FromBody] Book book)

{

\_books.AddBook(book);

foreach(var callback in \_callbacks.GetCallbacks())

{

callback.InvokeAsync(book);

}

return CreatedAtRoute("GetBook", new { id = book.Id }, book);

}

// PUT api/books/5

[HttpPut, Route("books/{id}")]

[Metadata("Modify an existing book object", "Modify an existing book. You need

to provide the new values for the Title or Author of the book. You look for the

book object using its id")]

public void Put([FromBody] Book book)

{

\_books.ModifyBook(book);

}

// DELETE api/books/5

[Metadata("Delete a book object", "Delete a book object by its id.")]

[HttpDelete, Route("books/{id}")]

public void Delete(string id)

{

\_books.DeleteBookById(id);

}

}

}

1. Open the SwaggerConfig.cs file. You can find this file in the App\_Start folder.
2. In the SwaggerConfig.cs file, uncomment the line c.PrettyPrint();.
3. Add the line c.ReleaseTheTRex(); inside the EnableSwagger method, just after this line:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg259-1a)

//c.CustomProvider((defaultProvider) => new CachingSwaggerProvider

(defaultProvider));

1. Uncomment this line:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg259-2a)

c.DocExpansion(DocExpansion.List);

1. Add the line using TRex.Metadata at the beginning of the file.

At this point, you can test your API. Use the following steps for testing your API:

1. In your Visual Studio 2019 window, ensure that your API project is open. Then press F5.
2. Once your web application is loaded in your web browser, append the */swagger* URI to the URL in your web browser. The final address should look like similar to *https://localhost:44398/swagger*.
3. On your API list of methods, shown in [Figure 5-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig04), click the *POST /books* link. This expands the options for the *POST /books* endpoint.



**Figure 5-4** List of endpoints of a web API

1. In the *POST /books* endpoint green area, in the Parameters section, type a JSON object in the value text area. You can find an example of the needed structure in the Data Type column at the end of the same line.
2. Click the Try It Out! button in the bottom-left corner of the endpoint green area.
3. In the Response Body section, you should get the JSON representation of your newly created book with a valid ID assigned. You can also check that the book has been created successfully by clicking the Try It Out! button in the *GET /books* endpoint blue area.

Now that you have tested that your web API works correctly, you need to publish to an Azure App Service. An alternative to deploying the web API to an Azure App Service could be using an on-premises data gateway for connecting to the local deployment of our web API. Use the following steps for deploying the web API to an Azure App Service:

1. In your Visual Studio 2019 window, ensure that you have opened the web API.
2. On the right side of the Visual Studio window, in the Solution Explorer window, right-click the project’s name.
3. In the contextual menu, click Publish. This opens the Publish window.
4. In the Publish window, make sure that Azure is selected from the list of Targets on the right side of the window.
5. Click Next.
6. Select Azure App Service (Windows) in the Specific Target section.
7. Click Next.
8. On the App Service section, click Create A New Azure App Service at the bottom of the window. If you already have an Azure App Service plan that you want to use for hosting the App Service, you can select it in the tree control in this same window.
9. In the App Service (Windows) window, leave all options as is and click the Create button. Remember to delete the Resource Group and all its associated resources after you finish this example because these resources consume credit from your Azure subscription.
10. On the tree control, expand the newly created resource group and click the newly created Azure App Service.
11. Click the Finish button.
12. Click the Publish button.
13. Once the publishing process has finished, Visual Studio opens your default web browser with the URL of the newly deployed App Service. This URL will have the structure *https://<your\_app\_service\_name>.azurewebsites.net*.
14. Ensure that your API is working correctly by repeating the testing steps previously shown in this section. This time you need to use the URL *https://<your\_app\_service\_name>.azurewebsites.net/swagger*.

At this point, you are ready to create your Azure Logic App Custom Connector. Before proceeding to create your custom connector for your API, let’s dig a little bit on the code of the API to understand the Azure Logic App Custom Connector that you are going to create. The API allows you to create, delete, modify, and query a book in the list of books that the application can store. You can also get the complete list of books stored in the API. As previously reviewed in this section, a connector can have a trigger that represents an event that starts the sequence of actions in our workload. In our testing API, the trigger represents the event of adding a new book to the list of books. You can see how we fire the trigger in the following code snippet extracted from the Post method in the *BooksController*:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg261a)

\_books.AddBook(book);

foreach (var callback in \_callbacks.GetCallbacks())

{

callback.InvokeAsync(book);

}

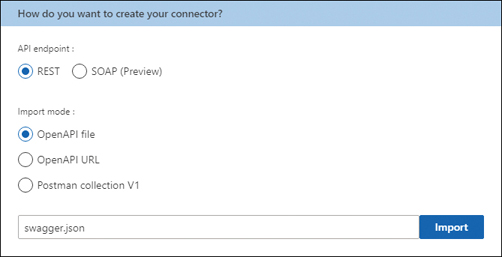
As you can see in the previous code, after we add a new book to the list of books, we call the InvokeAsync method of the callback object. The reason for doing this is because Azure Logic Apps works with two types of triggers: pull and push triggers. In this example, we decided to use a push trigger. This means that the API notifies the Azure Logic App when the event happens. The way the API can notify the Azure Logic App is by using the webhook pattern. For this reason, we need to provide two additional endpoints to the API, one for allowing the Azure Logic App to subscribe to the API and one for deleting the subscription. In the example, these endpoints are POST /books/subscribe and DELETE /books/subscribe/{callbackID} (you can view these endpoints in [Figure 5-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig03) earlier in this section).

The workflow subscribes to the API when the workflow changes in any way; for example, when you run it for the first time, change the input parameters to the trigger, or renew the credentials for connecting to your API. During the subscription process, your workflow provides the callback URL that your API needs to use for sending back the information needed for the workflow. The unsubscribe process happens automatically when the trigger, the workflow, or the subscription is deleted or disabled.

Swagger provides all this information about which endpoint Azure Logic App should use for subscribing to the events in your API, the structure of the data that uses your API, or the available endpoints for the connector. We could manually create the Swagger document that describes our API, but that would be a time-consuming and error-prone task. To ease the creation and management of the Swagger document that describes the API, we used the packages Swashbuckle ([*https://github.com/domaindrivendev/Swashbuckle*](https://github.com/domaindrivendev/Swashbuckle)) and TRex Metadata Library ([*https://github.com/nihaue/TRex*](https://github.com/nihaue/TRex)). The Swashbuckle package adds Swagger to the API, providing a Swagger generator and a UI for navigating and testing the API. TRex Metadata Library package extends the capabilities of Swashbuckle, so the Swagger document generated by Swashbuckle is ready to be consumed by the Logic App Designer.

Now that you have a better understanding of the key parts of the API that enable the interaction with Azure Logic Apps, it’s time to create a custom connector for your API:

1. In your web browser, download the Swagger definition of your API from *https://<your\_app\_service\_name>.azurewebsites.net/swagger/docs/v1*.
2. In your web browser, right-click the Swagger definition of the API, click Save As in the contextual menu, and save the page to your desktop as a file with the name **swagger.json**. You need this file for a later step.
3. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
4. Click Create A Resource button in the top area of the Azure portal.
5. Type *logic apps* in the Search The Marketplace text box.
6. Click Logic Apps Custom Connector in the result list.
7. Click the Create button.
8. On the Create Logic Apps Custom Connector panel, select an existing resource group in the Resource Group drop-down menu where you want to store your custom connector. Alternatively, you can create a new resource group by clicking the Create New link below the Resource Group drop-down menu.
9. Type a name for your custom connector in the Custom Connector Name text box.
10. In the Select The Location control, ensure that the option Region is selected.
11. Select a location in the Location drop-down menu.
12. Click the Review + Create button at the bottom of the panel.
13. Click the Create button.
14. Type the name of your newly created custom connector on the search text box at the top of the Azure portal.
15. Click the name of your Azure Logic App Custom Connector in the results.
16. Click the Edit button on the Custom Connector’s Overview blade.
17. On the Edit Logic Apps Custom Connector blade, click the Import button on the How Do You Want To Create Your Connector? panel, which is shown in [Figure 5-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig05).



**Figure 5-5** Importing the OpenAPI definition of a REST API

1. Choose the JSON file that you downloaded in step 2.
2. In the General Information section, review the information in this section. You don’t need to make any changes here.

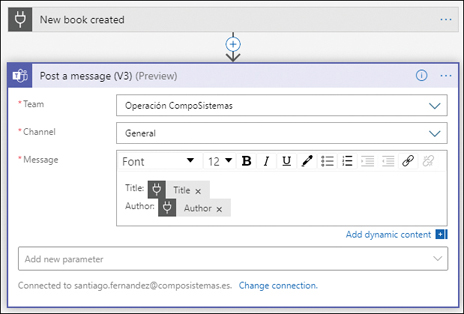
***NOTE* BASE URL**

In this example, the endpoint definition already contains the correct base URL in the endpoint definition. If you change the default Base URL property in the General Information section in your Azure Logic Apps Custom Connector, you will receive a 404 error every time you try to use your custom connector in an Azure Logic Apps workflow>.

1. Click the Security link on top of the page. You can also find a Security link at the bottom-left corner of the page. You can use either of these links for navigating to the Security section.
2. Review the options on the Security page. For this custom connector, you aren’t using any authentication. Ensure that the option No Authentication is selected.
3. Click the Definition link on top of the page. You can also find a Definition link at the bottom-left corner of the page. You can use either of these links for navigating to the Definition section.
4. Review the settings on the Definition page. On this page, you can find all the endpoints that have been defined in the Swagger JSON file that you imported in step 18. These endpoints translate into Actions or Triggers, depending on the definition in the JSON file. You can also manually add new Actions and Triggers as you need by using this page.
5. Scroll down the page until you see the Triggers section.
6. Click the NewBookCreated trigger. There should be a red circle with an exclamation mark beside the name of the trigger. This icon indicates that there is a problem with the definition of the trigger.
7. On the NewBookCreated trigger definition window, scroll to the end of the Request section. In the Body section inside the Request section, you should see the Callback object with a red circle with an exclamation mark.
8. Click the Callback object.
9. Click Edit on the contextual menu over the Callback object.
10. In the Body section of the callback object, click the Callback URL parameter with the red circle with an exclamation mark icon.
11. Click Edit on the contextual menu over the Callback URL parameter.
12. In the Callback URL parameter definition, in the Is Required? section, select Yes.
13. At the bottom of the Callback URL parameter definition page, in the Validation section, ensure that there is a green icon representing that the definition is okay.
14. Click the Back icon at the top-left corner of the Callback URL parameter definition page.
15. Click the Back icon at the top-left corner of the Callback object definition page.
16. Repeat steps 27 to 34 for the Callback objects with the exclamation mark inside a red circle icon in the References section on the left side of the Definition page.
17. Ensure that there is no exclamation mark inside a red circle icon on the Definition page.
18. Click the Update Connector link at the top-right corner of the Edit Logic Apps Custom Connector blade.

At this point, you have successfully created your Azure Logic Apps Custom Connector. In the following steps, you are going to create a workflow for testing your new custom connector. This workflow uses the trigger defined in your Azure Logic App Custom Connector. It gets the information from the newly created book in the API and puts the information in a Microsoft Teams channel:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click Create A Resource in the top area of the Azure portal.
3. Type **logic app** in the Search The Marketplace text box.
4. Click Logic App in the results list.
5. Click the Create button.
6. On the Logic App panel, select an existing resource group in the resource group drop-down menu, where you want to store your Logic App. Alternatively, you can create a new resource group by clicking the Create New link below the resource group drop-down menu.
7. Type a name for your Logic App in the Logic App Name text box.
8. In the Select The Location control, ensure that the option Region is selected.
9. Select a location in the Location drop-down menu.
10. Click the Review + Create button at the bottom of the panel.
11. Click the Create button.
12. Navigate to the newly created Azure Logic App.
13. On the Logic Apps Designer blade, choose the Blank Logic App template. If you don’t get the Logic Apps Designer blade as soon as you open your Azure Logic App, you can click Logic App Designer on the navigation menu on the left side of your Azure Logic App.
14. On the Logic App Designer, click the Custom tab.
15. Click your newly created custom connector.
16. In the Triggers section, click New Book Created. This is a simple trigger that requires no additional parameters.
17. Click the New Step button.
18. Type **Microsoft Teams** in the Search Connectors And Actions text box on the Choose An Action panel.
19. Click the Microsoft Teams icon.
20. Click the Post A Message (V3) (preview) action.
21. Grant access to your Microsoft Teams account.
22. On the Post A Message (V3) (preview) action panel, select an existing team from the Team drop-down menu.
23. Select the General channel in the Add Teams Channel ID drop-down menu.
24. Click inside the Message text box.
25. In the Message text area, type **Title:** followed by a space.
26. In the Dynamic Content dialog box, click the See More link in the New Book Created section.
27. In the New Book Created section, click Title.
28. Write a new line in the Message text area.
29. In the Message text area, type **Author:** followed by a space.
30. Repeat steps 26 and 27 for the Author property. Your Message area should look similar to [Figure 5-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig06).



**Figure 5-6** Send a Message to Microsoft Teams

1. Click the Save button in the top-left corner of the Logic Apps Designer blade.

***NEED MORE REVIEW?* VARIABLES**

You can declare and use variables in your workflow. These variables can contain data from your connectors, fixed values, or the result of some operations that you perform inside your workflow. You can learn more about how to use variables in your workflow by reviewing the article [*https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-variables-store-values*](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-variables-store-values)*.*

At this point, you should be ready to test your new Azure Logic Apps Custom Connector. For testing this workflow, you can use the same procedure that you used for testing the API earlier in this section. Create a new book in the API. The new book should appear in the Microsoft Teams channel that you configured in the Azure Logic Apps workflow, as shown in [Figure 5-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig07).



**Figure 5-7** Result of the workflow execution in Microsoft Teams

**Images *EXAM TIP***

You can create custom connectors for Azure Logic Apps, Microsoft Flow, and Microsoft PowerApps. You cannot reuse a connector created for Azure Logic Apps in Microsoft Flow or PowerApps (or vice versa). You can use the same OpenAPI definition to create a custom connector for these three services.

***NEED MORE REVIEW?* CUSTOM CONNECTOR**

You can learn more about custom connectors at [*https://docs.microsoft.com/en-us/connectors/custom-connectors/*](https://docs.microsoft.com/en-us/connectors/custom-connectors/)*.*

**Create a custom template for Logic Apps**

Once you have created an Azure Logic App, you can reuse it in other Azure subscriptions or share it with other colleagues. You can create a template from your working Azure Logic App to automate deployment processes. When you create a template, you are converting the definition of your Azure Logic App into an Azure Resource Manager (ARM) template. Using ARM templates enables you to take advantage of the flexibility of the ARM platform by separating the definition of the Azure Logic App from the values used in the logic app. When you deploy a new Azure Logic App from a template, you can provide a parameter file in the same way that you do with other ARM templates. Azure also provides some prebuilt Logic App templates. You can use these templates as a base for creating templates.

You can download an Azure Logic Apps template using several mechanisms:

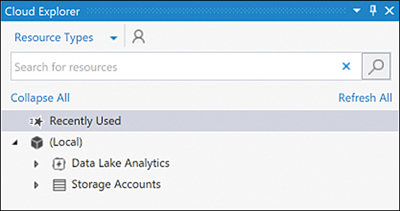
* **Azure portal** You can use the Export Template option in the Azure Logic App in the Azure portal for downloading the ARM template.
* **Visual Studio** You can use the Azure Logic Apps Tools extension for Visual Studio to connect your Azure subscription and download a template from your Azure Logic Apps.
* **PowerShell** You can use the LogicAppTemplate PowerShell module to download a template from your Azure Logic App.

A Logic App template is a JSON file comprised of three main areas:

* **Logic App resource** This section contains basic information about the Logic App itself. This information is the location of the resource, the pricing plans, and the workflow definition.
* **Workflow definition** This section contains the description of the workflow, including the triggers and actions in your workflow. This section also contains how the Logic App runs these triggers and actions.
* **Connections** This section stores the information about the connectors that you use in the workflow.

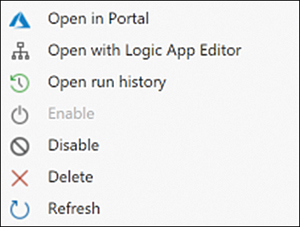
Use the following procedure to create a template from your Azure Logic App using Visual Studio:

1. Download the Azure Logic Apps Tool extension for Visual Studio 2019 at [*https://aka.ms/download-azure-logic-apps-tools-visual-studio-2019*](https://aka.ms/download-azure-logic-apps-tools-visual-studio-2019).
2. Install the Azure Logic Apps Tool extension.
3. Open Visual Studio 2019.
4. In the Visual Studio 2019 welcome window, click the Continue Without Code link below the Get Started section.
5. In the Visual Studio window, click View > Cloud Explorer.
6. In the Cloud Explorer window, shown in [Figure 5-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig08), click the user icon to open the Account Manager.



**Figure 5-8** Cloud Explorer window

1. Click the Manage Accounts link.
2. In the All Accounts section, click the Sign In link.
3. Sign in with an account that has privileges to access your Azure subscription.
4. Ensure that your Azure subscription appears in the list of subscriptions in the Cloud Explorer window.
5. Click the Apply button.
6. In the Cloud Explorer tree control, navigate to Your Subscription > Logic Apps.
7. Right-click the Logic App that you want to convert to a template.
8. In the contextual menu shown in [Figure 5-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig09), click Open With Logic App Editor.



**Figure 5-9** Logic App tool contextual menu

1. On the Logic App Editor tab, click the Download button.
2. Select a location to which you want to download the JSON file.

At this point, you can edit and customize your template. Once you are done with the modifications to your template, you can create a parameters file for deploying this template.

***NEED MORE REVIEW?* LOGIC APP TEMPLATES**

You can learn more by reading the following articles about Logic apps templates:

* **Create Logic App Templates** [*https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-azure-resource-manager-templates*](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-azure-resource-manager-templates)
* **Deploy Logic App Templates** [*https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-deploy-azure-resource-manager-templates*](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-deploy-azure-resource-manager-templates)

**Skill 5.2: Implement API Management**

Most of the applications and solutions that you can find or develop nowadays offer an API for accessing the features available in the solution. In business environments, it is quite usual that those solutions need to communicate with each other using their respective APIs. Sometimes, you need to expose your solutions to your clients to offer your services. In those situations, you need to ensure that you offer a consistent and secure API. Implementing the necessary mechanism for achieving an enterprise-grade level of security, consistency, and flexibility is not easy. If you also need to publish several of your services under a common API, this task is even harder.

Microsoft provides the Azure API Management (APIM) service. This service allows you to create an enterprise-grade API for your existing back-end services. Using APIM, you can securely publish your back-end applications, providing your customers with a platform protected against DOS attacks or JWT token validations.

**This skill covers how to**

* [Create an APIM instance](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec4)
* [Configure authentication for APIs](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec5)
* [Define policies for APIs](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec6)

**Create an APIM instance**

The API Management service allows you to expose a portion (or all) of the APIs offered by your back-end systems. By using the APIM service, you can unify all your back-end APIs in a common interface that you can offer to external users, such as clients or partners, and internal or external developers. In general, the APIM service is a façade of the APIs that you configure in your APIM instance. Thanks to this façade feature, you can customize the front-end API offered by the APIM instance without changing the back-end API.

When exposing your back-end systems, you are not limited to REST API back ends. You can use a back-end service that uses a SOAP API and then publish this SOAP API as a REST API. This means you can update your older back-end systems without needing to modify the code and take advantage of the greater level of integration of the REST APIs.

Use the following procedure to create a new APIM instance:

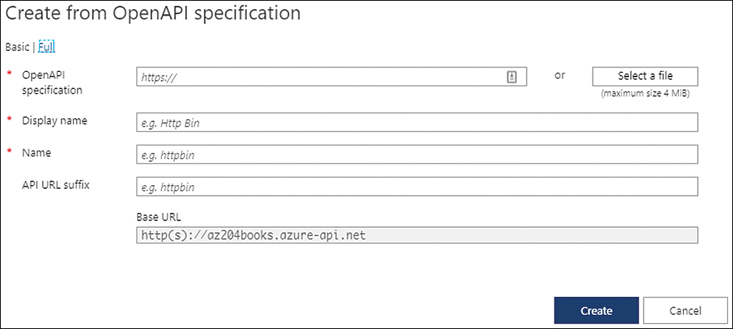
1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click Create A Resource at the top of the Azure portal.
3. On the New blade, click Integration in the Azure Marketplace column.
4. Click API Management in the Featured column. If the API Management service doesn’t appear in the Featured column, you can use the Search The Marketplace text box and look for the API Management service.
5. On the API Management Service blade, type a name for your new APIM instance.
6. Select a subscription from the Subscription drop-down menu.
7. Select a resource group from the Resource Group drop-down menu. Alternatively, you can create a new one by clicking the Create New link below the drop-down menu.
8. Select a location from the Location drop-down menu.
9. In the Organization Name text box, type the name of your organization. This name appears on the developer’s portal and email notifications.
10. In the Administrator Email, type the name of the email account that should receive all notifications from the APIM instance. By default, the value associated with this property is the email address of the logged-in user.
11. In the Pricing Tier, leave the Developer tier selected.
12. Click the Create button at the bottom of the blade. The process of creating the APIM instance takes several minutes. When your new APIM instance is ready, you receive a welcome email at the administrator email address that you configured in step 10.

***NOTE* PRICING TIERS**

The Developer pricing tier is appropriate for testing and development environments, but you should not use it for production because the Developer tier does not offer high-availability features and can be affected by disconnections during the updates of the node. You can review the full offer and the features available on each tier at [*https://azure.microsoft.com/en-us/pricing/details/api-management/*](https://azure.microsoft.com/en-us/pricing/details/api-management/)*.*

Once you have created your APIM instance, you can start adding APIs to your instance. In the following procedure, you are going to add two APIs. You are going to take advantage of the API that you created in the section “[Create A Custom Connector For Logic Apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec2)” previously in this chapter. For the second API, you are going to create a blank API definition and add only those methods that are appropriate for you.

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Type the name of your APIM instance in the Search text box at the top of the portal.
3. Click the name your APIM instance in the results list.
4. Click APIs on the navigation menu on your APIM instance blade.
5. On the Add A New API blade, click OpenAPI.
6. On the Create From OpenAPI Specification dialog box, shown in [Figure 5-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig10), in the OpenAPI Specification text box, type the URL of the Swagger definition of the API that you published in Azure in the section “[Create a custom connector for Logic Apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec2).” Remember that the URL should be similar to *https://<your\_app\_service\_name>.azurewebsites.net/swagger/docs/v1*.

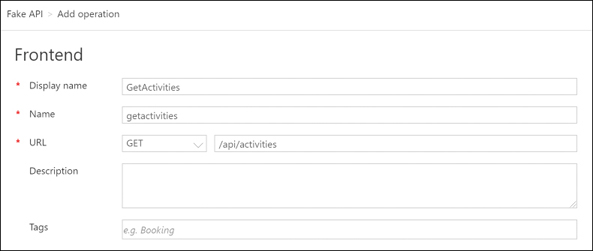


**Figure 5-10** Adding a back-end API to an APIM instance

1. Ensure that Azure automatically fills the Display Name and Name properties text boxes. This means that Azure was able to import the details of your API successfully.
2. Delete the content of the Display Name text box.
3. Type **Library** in the Display Name text box.
4. Ensure that the Name text box has the value **library**.
5. Type **library** in the API URL Suffix field. If you are going to connect more than one back-end API to the APIM instance, you need to provide a suffix for each API. The APIM instance uses this suffix for differentiating between the different APIs that you connected to the instance.
6. Click the Create button at the bottom of the dialog box.

At this point, you have added your first back-end API to the APIM instance by using the OpenAPI specification of your back-end API. In the following steps, you are going to add a back-end API without using any specification. Creating the front-end endpoints is useful if you need to connect only a few endpoints from your back-end API or if you don’t have the OpenAPI or SOAP specification of your API in any format:

1. Click APIs on the navigation menu in your APIM instance blade.
2. On the APIs blade, click Add API.
3. On the Add A New API page, click Blank API.
4. On the Create A Blank API dialog box, type **Fake API** in the Display Name text box.
5. Leave the Name property with the default value.
6. Type [**https://fakerestapi.azurewebsites.net**](https://fakerestapi.azurewebsites.net/) in the Web Service URL text box.
7. Type **fakeapi** in the API URL Suffix text box.
8. Click the Create button.
9. On the Design tab of the API blade with the newly added Fake API selected, click Add Operation.
10. On the Add Operation editor, shown in [Figure 5-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig11), type **GetActivities** in the Display Name text box.



**Figure 5-11** Adding an API operation to an API in an APIM instance

1. In the URL HTTP Method drop-down menu, ensure that the *GET* method is selected.
2. In the URL text box, type **/api/activities***.*
3. Click the Save button at the bottom of the editor.
4. On the API blade, ensure that Fake API is selected.
5. Click the Test tab.
6. Click the GetActivities operation.
7. Click the Send button at the bottom of the GetActivities operation panel. Using this panel, you can test each of the operations that are defined in your API.

At this point, you have two back-end APIs connected to your APIM instance. As you can see in the previous example, you don’t need to expose the entire back-end API. By adding the appropriate operations, you can publish only those parts of the back-end API that are useful for you. Once you have created the APIs in your APIM instance, you can grant access to these APIs to your developers by using the Developer portal. You can access the APIM Developer portal by using the URL *https://<your\_APIM\_name>.developer.azure-api.net/.*

***NEED MORE REVIEW?* AZURE API MANAGEMENT DEVELOPER PORTAL**

The Azure API Management developer portal allows you to provide your customers and third parties that want to integrate with your API with a single point of contact for requesting access to your application and providing documentation about your API. You can read more about how to customize the developer experience by reviewing the following article at [*https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-developer-portal*](https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-developer-portal).

Bear in mind that you need to associate a product to your API for publishing it. Because you didn’t associate your APIs to any product, your APIs won’t be available to the external world. You can associate an API to more than one product. By default, Azure provides two products: Starter and Unlimited. These products are associated with the Echo API demo that Azure automatically deploys when you create your API Management instance. Use the following procedure to create a product and associate it with your APIs:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Type the name of your APIM instance in the Search text box on the top-middle of the portal.
3. Click the name of your APIM instance in the results list.
4. Click Products on the navigation menu in your APIM instance blade.
5. Click the Add button on the top-left corner of the Products blade.
6. Type a Name in the Display Name text box on the Add Product panel.
7. Leave the value in the ID text box as is.
8. Type a description in the Description text area.
9. Select the Published value in the State switch control. If you don’t select this option at this time, you can publish later, or you can publish the Product using its panel.
10. Click the Select API button in the APIs section.
11. On the APIs blade, select Library and Fake APIs by clicking the check box beside the name of the API.
12. Click the Select button at the bottom of the panel.
13. Click the Create button at the bottom of the Add Product panel.

By default, when you create a new product, only members of the Administrators built-in group can access the product. You can configure this by using the Access Control section in the product.

***NEED MORE REVIEW?* REVISIONS AND VERSIONS**

During the lifetime of your API, you may need to make modifications by adding, updating, or removing operations to your API. You can make these modifications without disrupting the usage of your API by using revisions and versions. You can review how to work with revisions and versions in your API by reading the article at [*https://azure.microsoft.com/es-es/blog/versions-revisions/*](https://azure.microsoft.com/es-es/blog/versions-revisions/)*.*

**Configure authentication for APIs**

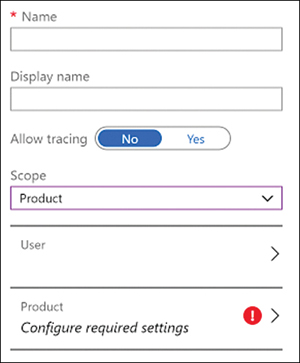
Once you have imported your back-end APIs, you need to configure the authentication for accessing these APIs. When you configure the security options in the APIM instance, the back-end API delegates the security to the APIM instance. This means that even your API has implemented its own authentication mechanism, it is never used when the API is accessed through the APIM instance.

This ability to hide the authentication of the back-end APIs is useful for unifying all the security using a consistent and unique authentication mechanism. You can manage the authentication options associated with a product or API by using subscriptions. A subscription manages the keys that a developer can use for accessing your API. If an HTTP request made to an API protected by a subscription does not provide a valid subscription key, the request is immediately rejected by the APIM gateway without reaching your back-end API. When you define a subscription, you can use three different scopes for applying it:

* **Product** The developer can access all the APIs configured in the product assigned to the subscription. Traditionally, the developer could request access to products by using the Developer portal. This is no longer a valid option. You need to provide access to the developer using the Azure portal and configure the appropriate APIM subscription.
* **All APIs** The developer can access all APIs in your APIM instance using the same subscription key.
* **API** The developer can access a single API in your APIM instance using a subscription key. There is no need for the API to be part of a product.

If you use the All APIs scope, you don’t need to associate the back-end API with an API. The Subscription using this scope allows access directly to all the APIs configured in your API Management instance. You can use the following procedure for creating a subscription and associating it with a program:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Type the name of your APIM instance in the Search text box at the top of the portal.
3. Click the name your APIM instance in the results list.
4. Click Subscriptions in the navigation menu in your APIM instance blade.
5. Click the Add Subscription button in the top-left corner of the Subscriptions blade.
6. On the New Subscription panel shown in [Figure 5-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig12), type a Name for the subscription. Beware that this name can only contain letters, numbers, and hyphens.



**Figure 5-12** Creating a new API Management Subscription

1. In the Scope drop-down menu, select the Product value.
2. Click the Product property.
3. In the Products panel, click the name of the product that you created in the previous section.
4. Click the Select button at the bottom of the panel.
5. Click the Save button at the bottom of the panel.
6. On the Subscription blade, click the ellipsis at the end of the row for your newly created subscription.
7. On the contextual menu, click Show/Hide keys. You can use either of these keys to access the APIs configured in the product associated with the Subscription. You need to use the Header Ocp-Apim-Subscription-Key for providing the subscription key in your HTTP requests.

When you are configuring a subscription, you can assign different users to the subscription by using the Users parameters in the New Subscription panel. This is a best practice way of providing different subscription keys to different groups of users.

***NEED MORE REVIEW?* OTHER AUTHENTICATION METHODS**

Using subscription and subscription keys is not the only mechanism for protecting access to your APIs. API Management allows you to use OAuth 2.0, client certificates, and IP whitelisting. You can use the following articles to review how to use other authentication mechanisms for protecting your APIs:

* **IP whitelisting** [*https://docs.microsoft.com/en-us/azure/api-management/api-management-access-restriction-policies#RestrictCallerIPs*](https://docs.microsoft.com/en-us/azure/api-management/api-management-access-restriction-policies#RestrictCallerIPs)
* **OAuth 2.0 authentication using Azure AD** [*https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-protect-backend-with-aad*](https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-protect-backend-with-aad)
* **Mutual authentication using client certificates** [*https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-mutual-certificates*](https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-mutual-certificates)

**Define policies for APIs**

When you publish a back-end API using the API Management service, all the requests made to your APIM instance are forwarded to the correct back-end API, and the response is sent back to the requestor. None of these requests or responses are altered or modified by default, but there could be some situations where you need to modify some requests and/or responses. An example of these modification needs is transforming the format of a response from XML to JSON. Another example could be throttling the number of incoming calls from a particular IP or user.

A policy is a mechanism that you can use to change the default behavior of the APIM gateway. Policies are XML documents that describe a sequence of inbound and outbound steps or statements. Each policy is made of four sections:

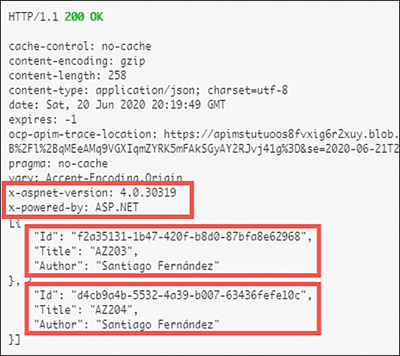
* **Inbound** In this section, you can find any statement that applies to requests from the managed API clients.
* **Back End** This section contains the steps that need to be applied to the request that should be sent from the API gateway to the back-end API.
* **Outbound** This section contains statements or modifications that you need to apply to the response before it’s sent to the requestor.
* **On-Error** In case there is an error on any of the other sections, the engine stops processing the remaining steps on the faulty section and jumps to this section.

When you are configuring or defining a policy, you need to bear in mind that you can apply it different scope levels:

* **Global** The policy applies to all APIs in your APIM instance. You can configure global policies by using the code editor on the All APIs policy editor in the APIs blade of your APIM instance.
* **Product** The policy applies to all APIs associated with a product. You can configure product policies on the Policies blade of the product in your API instance.
* **API** The policy applies to all operations configured in the API. You can configure API-scoped policies by using the code editor in the All Operations option on the Design Tab of the API in your APIM instance.
* **Operation** The policy applies only to a specific operation in your API. You can configure operation-scoped policies by using the code editor in the specific operation.

Policies are a powerful and very flexible mechanism that allows you to do a lot of useful work, such as applying caching to the HTTP requests, performing monitoring on the request and responses, authenticating with your back-end API using different authentication mechanisms, or even interacting with external services, among others. Use the following procedure to apply some transformations to the Library API that you configured in the “[Create an APIM instance](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec1)” section earlier in this chapter:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Type the name of your APIM instance in the Search text box at the top of the portal.
3. Click the name of your APIM instance in the results list.
4. Click APIs on the navigation menu in your APIM instance blade.
5. Click Library API in the APIs blade.
6. Click the *Get Books* operation.
7. Click the Test tab.
8. Click the Send button at the bottom of the tab. This should send a request to the Library API and get results similar to those shown in [Figure 5-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig13). In this procedure, you are going to transform the HTTP headers inside the red rectangles in [Figure 5-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig13). If you don’t see any books when you execute this test, create some books using the Swagger application of your API. You can review how to do this by consulting the testing process of the Library API in the “[Create a custom connector for Logic Apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec2)” section earlier in this chapter.



**Figure 5-13** Testing an API operation

1. Click the Design tab.
2. Click All Operations in the list of available operations for this API.
3. Click the icon next to Policies in the Outbound Processing section.
4. In the Policy Editor, move the cursor inside the Outbound section, before the base tag, and add a new line by pressing the Enter key.
5. Click the Show Snippets button in the top-right corner of the Policy Editor.
6. In the list of available policies on the right side of the Policy Editor, navigate to Transformation Policies.
7. Click the Set HTTP Header policy twice to insert the policies.
8. Modify the inserted policies with the following content:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg277-1a)

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

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

1. Add a new line below the inserted policies.
2. Add the following code snippet:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg277-2a)

<set-body>@{

var response = context.Response.Body.As<string>();

var arrayString = "{ \"Library\": " + response + "}";

JObject books = JObject.Parse(arrayString);

JArray modifiedBooks = new JArray();

foreach (JObject book in books["Library"].ToObject<JArray>())

{

book.Add("URL", "https://az204books.azure-api.net/library/books/" + book["Id"]);

modifiedBooks.Add(book);

}

return (string)modifiedBooks.ToString(Newtonsoft.Json.Formatting.None);

}</set-body>

1. Click the Save button at the bottom of the Policy Editor.
2. Repeat steps 6 to 8 to apply the transformation policies. You should notice that headers X-Powered-By and X-AspNet-Version are missing. Also, you should see that all books have an additional property URL pointing to the URL of the book.

As you can see in the previous example, the policies in the API Management service are compelling. You can even use C# code for making elaborate modifications to the requests and responses made to your API. Although this example shows part of the power of using policies with the APIM service, you should not use this example for a production environment, as some critical verifications are missing from this example policy. Because we created this policy for All Operations in the Library API, any call made to an operation different from Get Books is going to fail.

***NEED MORE REVIEW?* MORE ABOUT POLICIES**

There are a lot of useful things that you can do using policies—too many to cover in this section. If you want to learn more about APIM policies, you can review the following articles:

* **Error handling in API Management** [*https://docs.microsoft.com/en-us/azure/api-management/api-management-error-handling-policies*](https://docs.microsoft.com/en-us/azure/api-management/api-management-error-handling-policies)
* **How to Set or Edit Azure API Management Policies** [*https://docs.microsoft.com/en-us/azure/api-management/set-edit-policies*](https://docs.microsoft.com/en-us/azure/api-management/set-edit-policies)
* **Debug Your APIs Using Request Tracing** [*https://docs.microsoft.com/es-es/azure/api-management/api-management-howto-api-inspector*](https://docs.microsoft.com/es-es/azure/api-management/api-management-howto-api-inspector)

**Skill 5.3: Develop event-based solutions**

One of the main principles of code development is to reuse as much as possible. To make it possible to reuse the code, you need to ensure that the code is as loosely coupled as possible, which reduces the dependencies with other parts of the code or other systems to a minimum.

With this principle in mind, to make loosely coupled systems communicate, you need to use a kind of communication. Event-driven architectures allow communication between separate systems by sharing information through events.

In general, an event is a significant change in the system state that happens in the context of the system. An example of an event could be when a user adds an item to the shopping cart in an e-commerce application, or when an IoT device collects the information from its sensors.

Azure provides different services, like Event Grid, notification hubs, or event hubs, to cover the different needs when implementing Event-Driven architectures.

**This skill covers how to**

* [Implement solutions that use Azure Event Grid](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec7)
* [Implement solutions that use Azure Notification Hubs](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec8)
* [Implement solutions that use Azure Event Hub](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec9)

**Implement solutions that use Azure Event Grid**

Azure Event Grid allows you to create an application using serverless architecture by providing a confident platform for managing events. You can use Azure Event Grid for connecting to several types of data sources, like Azure Blob Storage, Azure Subscription, Event Hubs, IoT Hubs, and others; Azure Even Grid also allows you to use different event handlers to manage these events. You can also create your custom events for integrating your application with the Azure Event Grid. Before you can start using the Azure Event Grid in your solution, there are some basic concepts that we should review:

* **Event** This is a change of state in the source (for example, in an Azure Blob Storage or when an event happens when a new blob is added to the Azure Blob Storage).
* **Event source** This is the service or application when the event happens. There is an event source for every event type.
* **Event handler** This is the app or service that reacts to the event.
* **Topics** These are the endpoints where the event source can send the events. You can use topics for grouping several related events.
* **Event subscriptions** When a new event is added to a topic, that event can be processed by one or more event handlers. The event subscription is an endpoint or built-in mechanism to distribute the events between the different event handlers. Also, you can use subscriptions to filter incoming events.

An important consideration that you need to bear in mind is that an event does not contain the full information about the event itself. The event only contains information relevant to the event, such as the source of the event, a time when the event took place, and a unique identifier. For example, when a new blob is added to an Azure Blob Storage Account, the new blob event doesn’t contain the blob. Instead, the event contains a reference to the blob in the Azure Blob Storage Account.

When you need to work with events, you configure an event source to send events to a topic. Any system, or event handler, that needs to process those events subscribes to that topic. When new events arise, the event source pushes the event into the topic configured in the Azure Event Grids service. Any event handler subscribed to that topic reads the event and processes it according to its internal programming. There is no need for the event source to have event handlers 280subscribed to the topic; the event source pushes the event to the topic and forgets it. The following steps show how to create a custom topic. Then you are going to create console applications using C# to send events to the topic and process these events.

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the Search Resources, Services, And Docs text box on the top area of the Azure portal, type **event**.
3. Click Event Grid Topic in the results list.
4. On the Event Grid Topics blade, click the Add button in the top-left corner of the blade.
5. On the Create Topic panel, select a subscription in the Subscription drop-down menu.
6. Select a resource group in the Resource Group drop-down menu. Alternatively, you can create a new resource group by clicking the Create New link below the drop-down menu.
7. In the Name text box, type a name for the Event Grid Topic.
8. Select a location in the Location drop-down menu.
9. Click the Review + Create button at the bottom of the panel.
10. Click the Create button.

When the Azure Resource Manager finishes creating your new Event Grid Topic, you can subscribe to the topic for processing the events. Also, you can send your custom events to this topic. Use the following steps to publish custom events to your newly created Event Grid Topic:

1. Open Visual Studio 2019.
2. On the start window, click Create A New Project.
3. On the Create A New Project window, select the template Console App (.NET Core).
4. Click the Next button at the bottom-right corner of the window.
5. Type a Project Name.
6. Select a location for your solution.
7. Click the Create button.
8. Click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
9. On the NuGet – Solution tab, click Browse.
10. In the Search text box, type **Microsoft.Azure.EventGrid***.*
11. Click Microsoft.Azure.EventGrid in the results list.
12. On the right side of the NuGet – Solution tab, click the check box next to the name of your project.
13. Click the Install button.
14. On the Preview Changes window, click the OK button.
15. In the License Acceptance window, click the I Accept button.
16. Repeat steps 10 to 15 and install the Microsoft.Extensions.Configuration.Json NuGet Package.
17. In the Solution Explorer window, right-click your project’s name.
18. On the contextual menu, click Add > New Item.
19. On the Add New Item, type **json** in the Search text box.
20. Click the JSON File template.
21. Type **appsettings.json** in the Name text box.
22. Click the Add button at the bottom-right corner of the window.
23. On the Solution Explorer window, click the appsettings.json file.
24. On the properties window, set the Copy To Output Directory setting to Copy Always.
25. Open the appsettings.json file and replace the content of the file with the content of [Listing 5-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex06). You can get the access key from the Access Key blade in your Event Grid Topic.

**Listing 5-6** appsettings.json file

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-6a)

**{**

"EventGridAccessKey": "*<Your\_EventGridTopic\_Access\_Key>*",

"EventGridTopicEndpoint": "https://*<Your\_EventGrid\_Topic>.<region\_name>*-1.eventgrid.

azure.net/api/events"

}

1. On the Solution Explorer window, right-click your project’s name.
2. On the contextual menu, click Add > New Item.
3. In the Add New Item window, select Class from the list of new items.
4. In the name text box at the bottom of the window, type **NewItemCreatedEvent.cs.**
5. Click the Add button at the bottom-right corner of the window.
6. Replace the content of the NewItemCreatedEvent.cs file with the content of [Listing 5-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex07).

**Listing 5-7** NewItemCreatedEvent.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-7a)

**// C# .NET**

using Newtonsoft.Json;

namespace *<your\_project\_name>*

{

class NewItemCreatedEvent

{

[JsonProperty(PropertyName = "name")]

public string itemName;

}

}

1. Open the Program.cs file.
2. Add the following using statements:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#pg282a)

using Microsoft.Azure.EventGrid;

using Microsoft.Azure.EventGrid.Models;

using Microsoft.Extensions.Configuration;

using System.Collections.Generic;

1. Replace the content of the Main method with the content in [Listing 5-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex08).

**Listing 5-8** Program.cs Main method

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-8a)

**// C# .NET**

IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.

json");

IConfigurationRoot configuration = builder.Build();

string topicEndpoint = configuration["EventGridTopicEndpoint"];

string apiKey = configuration["EventGridAccessKey"];

string topicHostname = new Uri(topicEndpoint).Host;

TopicCredentials topicCredentials = new TopicCredentials(apiKey);

EventGridClient client = new EventGridClient(topicCredentials);

List<EventGridEvent> events = new List<EventGridEvent>();

events.Add(new EventGridEvent()

{

Id = Guid.NewGuid().ToString(),

EventType = "MyCompany.Items.NewItemCreated",

Data = new NewItemCreatedEvent()

{

itemName = "Item 1"

},

EventTime = DateTime.Now,

Subject = "Store A",

DataVersion = "3.7"

});

client.PublishEventsAsync(topicHostname, events).GetAwaiter().GetResult();

Console.WriteLine("Events published to the Event Grid Topic");

Console.ReadLine();

At this point, your console application publishes events to the Event Grid topic that you previously created. Press F5 to run your console application to ensure that everything compiles and works correctly; you will not be able to see the published message yet. Use the following steps to create a subscriber Azure Function that connects to the Event Grid Topic and processes these events:

1. Open Visual Studio 2019.
2. In the start window, click Create A New Project.
3. On the Create A New Project window, click the template Azure Functions.
4. Click Next.
5. Type a Project Name.
6. Select a location for your project.
7. Click Create.
8. On the Create A New Azure Functions Application window, click the Event Grid Trigger.
9. In the Storage Account drop-down menu on the right side of the window, click Browse.
10. In the Azure Storage window, select an Azure Storage Account from your subscription for using with the Azure Function. Alternatively, you can create a new Azure Storage Account by clicking the Create A Storage Account link at the bottom of the window.
11. Click the Add button.
12. Click Create.
13. Create a new empty C# class called NewItemCreatedEventData.
14. Replace the content of the NewItemCreatedEventData.cs file with the content of [Listing 5-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex09).

**Listing 5-9** NewItemCreatedEvent.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-9a)

**// C# .NET**

using Newtonsoft.Json;

namespace *<your\_project\_name>*

{

class NewItemCreatedEvent

{

[JsonProperty(PropertyName = "name")]

public string itemName;

}

}

1. Replace the content of Function1.cs with the content in [Listing 5-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex10).

**Listing 5-10** *Function1.cs*

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-10a)

**// C# .NET**

using Microsoft.Azure.WebJobs;

using Microsoft.Azure.EventGrid.Models;

using Microsoft.Azure.WebJobs.Extensions.EventGrid;

using Microsoft.Extensions.Logging;

using Newtonsoft.Json.Linq;

namespace *<your\_project\_name>*

{

public static class Function1

{

[FunctionName("EventGridTrigger")]

public static void Run([EventGridTrigger]EventGridEvent eventGridEvent,

ILogger log)

{

log.LogInformation("C# Event Grid trriger handling EventGrid Events.");

log.LogInformation($"New event received: {eventGridEvent.Data}");

if (eventGridEvent.Data is StorageBlobCreatedEventData)

{

var eventData = (StorageBlobCreatedEventData)eventGridEvent.Data;

log.LogInformation($"Got BlobCreated event data, blob URI {eventData.

Url}");

}

else if (eventGridEvent.EventType.Equals("MyCompany.Items.NewItemCreated"))

{

NewItemCreatedEventData eventData = ((JObject)eventGridEvent.Data).

ToObject< NewItemCreatedEventData >();

log.LogInformation($"New Item Custom Event, Name {eventData.

itemName}");

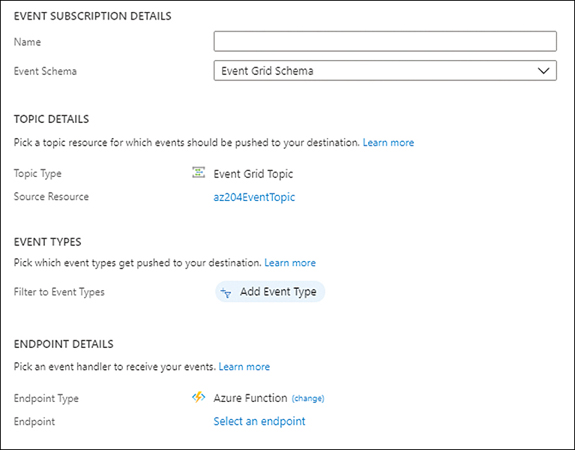
}

}

}

}

1. Publish the Azure Function to your Azure Subscription. Use the procedure at the following URL to publish an Azure Function to Azure: [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-vs#publish-to-azure*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-vs#publish-to-azure)*.*
2. Open your Event Grid Topic in the Azure portal.
3. On your Event Grid Topic Overview blade, click the Event Subscription button.
4. On the Create Event Subscription blade, shown in [Figure 5-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig14), type a Name for the subscription.

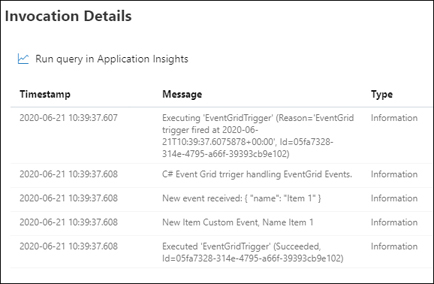


**Figure 5-14** Creating a subscription using a WebHook endpoint

1. In the Endpoint Type drop-down menu, select Azure Function.
2. Click the Select An Endpoint link below the Azure Function endpoint type.
3. On the Select Azure Function panel, in the Function App drop-down menu, select the Azure Function that you published previously in this section.
4. Leave the Slot drop-down menu with the Production value.
5. Ensure that your Azure Function’s name appears in the Function drop-down menu.
6. Click the Confirm Selection button.
7. Click the Create button.

At this point, you should be able to publish and process events using the Event Grid Topic that you created previously. Use the following steps to ensure that everything works correctly:

1. Open the publisher console application in Visual Studio 2019.
2. Run the console application to publish an event to the topic.
3. Open the Azure portal and navigate to your Azure Function.
4. In the Azure Functions blade, click Monitor in the tree control.
5. In the Monitor blade, click the Configure button for configuring the Application Insights integration. You need this integration for being able to capture invocation logs.
6. On the Application Insights blade, leave the Create New Resource option selected. Alternatively, you can use an existing Application Insights instance by using the option Select Existing Resource.
7. Click the OK button.
8. You should be able to see a list of invocations when the function has been called because a new event arrived at the Event Grid Topic.
9. Click one of the successful invocations; you will get a result similar to [Figure 5-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig15).



**Figure 5-15** Log messages from a successful event processing

***NOTE* AZURE FUNCTION MONITORING**

You need to have Application Insight integration enabled to be able to see the log messages generated from the Azure Function. Review the article about how to monitor Azure Functions using Application Insights at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-monitoring*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-monitoring)*.*

The Azure Function that we used in this example can manage not only custom events but also events from an Azure Storage Account. As an exercise, you can create a new subscription that listens only to Azure Storage Account events and uses the Azure Function that you published previously in this section to manage the events produced by the Azure Storage Account.

Another important consideration that you need to deal with when you add a handler to an Azure Event Grid subscription is the handler validation. Depending on the type of handler that you use, this validation process is performed automatically by the SDK, or you need to implement it manually. When you use an HTTP endpoint as an event handler, you need to deal with the subscription verification. This verification process consists of a verification code sent by the Event Grid service to the webhook endpoint. Your application needs to reply to the Event Grid service by using the same verification code. You can find a detailed example of how to perform this verification by reviewing the code available at [*https://github.com/Azure-Samples/azure-event-grid-viewer*](https://github.com/Azure-Samples/azure-event-grid-viewer).

***NEED MORE REVIEW?* DEAD LETTER AND RETRY POLICIES**

When you work with event-driven architectures, there can be situations when the event cannot deliver to the event handler. In those situations, it’s appropriate to set a retry strategy to try to recover the event before it expires. You can learn more about these retry policies and dead letter management at [*https://docs.microsoft.com/en-us/azure/event-grid/manage-event-delivery*](https://docs.microsoft.com/en-us/azure/event-grid/manage-event-delivery)*.*

**Images *EXAM TIP***

Event Grid is one of the services that Azure provides for exchanging information between different systems. These systems publish and consume events from the Event Grid, allowing you to decouple the different elements of your architecture. Ensure that you fully understand the role each element plays in the exchange of information using Event Grid.

**Implement solutions that use Azure Notification Hubs**

Developing applications that can be accessed using mobile devices can be challenging because you usually need to allow access to your application from different mobile platforms. The challenge becomes even bigger because different mobile platforms use different notification systems to send events. You need to deal with the Apple Push Notification Service (APNS), Google Firebase Cloud Messaging (FCM), or Windows Notification Service (WNS), and these are just the leading mobile platforms on the market. There are many other mobile platforms that you can use for pushing notifications to your mobile app.

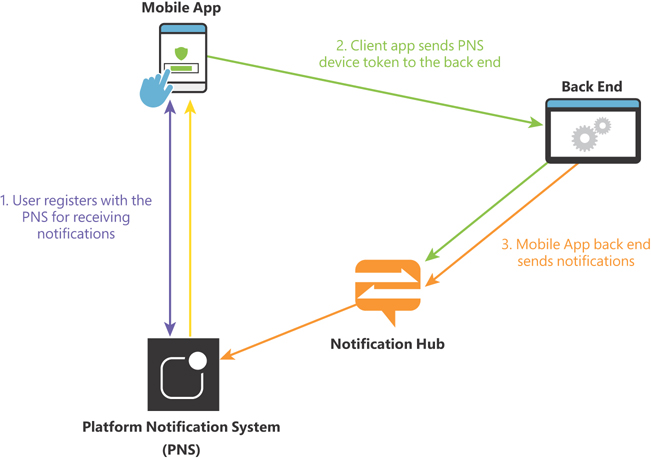
The Azure Notification Hubs provide an abstraction layer that you can use for connecting to different push notification mobile platforms. Thanks to this abstraction, you can send the notification message to the Notification Hub, which manages the message and delivers it to the appropriate platform. You can also define and use cross-platform templates. Using these templates, you ensure that your solution sends consistent messages independently of the mobile platform that you are using.

When you develop a mobile app, there is a high probability that you need to send information to your users when they are not using the app. In doing so, you use the well-known push notifications. This asynchronous communication mechanism allows you to interact with your users when they are offline. For making this interaction happens, some key players are part of this asynchronous communication:

* **The mobile app client** This is your actual mobile app, which runs on your user’s device. The user must register with the Platform Notification System (PNS) to receive notifications. This generates a PNS handler that is stored in the mobile app back end for sending notifications.
* **The mobile app back end** This is the back end for your app client, and it stores the PNS handler that the client received from the PNS. Using this handler, your back-end service can send push notifications to all registered users.
* **A Platform Notification System (PNS)** These platforms deliver the actual notification to the user’s device. PNSes are platform-dependent, and each vendor has its own PNS. Apple has the Apple Push Notification Service, Google uses the Firebase Cloud Messaging, and Microsoft uses the Windows Notification Service.

Even if your mobile app is targeted to a single platform, implementing push notifications requires a good amount of effort. This is because some Platform Notification Systems only focus on delivering the notification to the user’s device but don't deal with requirements like targeted notifications or broadcasting notifications. Another requirement for most PNSes is that device tokens need to be refreshed every time you release a new version of your app. This operation requires that your back end deals with a large amount of traffic and database updates simply to keep device tokes updated. If you need to support different mobile platforms, these tasks become even more complicated.

Microsoft provides you with the Azure Notification Hub. This service provides cross-platform push notification to your mobile app back end, allowing you to abstract from the details of managing each Platform Notification System to provide a consistent API for interacting with the Notification Hub. When you need to add push notifications to your mobile app, you integrate the Notification Hub service with your back-end service hosted on the Mobile App Service. [Figure 5-16](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig16) shows the workflow for sending push notifications to users using the Notification Hub.



**Figure 5-16** Push notification workflow using Notification Hub

***NOTE* NOTIFICATION HUB INTEGRATION**

Microsoft also provides an SDK for easing the direct integration between your native (iOS, Android, or Windows) or cross-platform (Xamarin or Cordoba) code and Azure Notification Hub, without using your back end. The drawback of this approach is that the Mobile Apps Client SDK removes all tags that you can associate with the device for security purposes. If you need these tags for performing segmented notifications, you should register your users’ devices using the back end .

The interaction between your back-end Mobile App and Notification Hub is performed using the Mobile App SDK for ASP.NET or Node.js web applications. Before your back-end application can 289send push notifications, you need to connect your App Service with your Notification Hub. Use the following procedure to make this connection:

1. Sign in to the Azure portal ([*http://portal.azure.com*](http://portal.azure.com/)).
2. At the top of the portal, click Create A Resource.
3. On the New blade, in the Search the Marketplace text box, type **notification**.
4. Click Notification Hub in the result list.
5. On the Notification Hub blade, click the Create button.
6. On the Create Notification Hub blade, select your subscription in the Subscription drop-down menu.
7. Select your resource group in the Resource Group drop-down menu. Alternatively, you can create a new resource group by clicking the Create New link.
8. Type a namespace in the Notification Hub Namespace text box. A namespace is a group of one or more hubs.
9. Type a name in the Notification Hub text box.
10. Select the location of the notification hub in the Location drop-down menu.
11. Leave the pricing tier as Free.
12. Click the Create button at the bottom of the blade.
13. Once the Notification Hub has been created, type the name of your new Notification Hub in the Search Resources, Services, And Docs text box at the top of the Azure portal.
14. At this point, you can configure the integration of the Notification Hub with each Platform Notification System that you want to use for sending notifications.

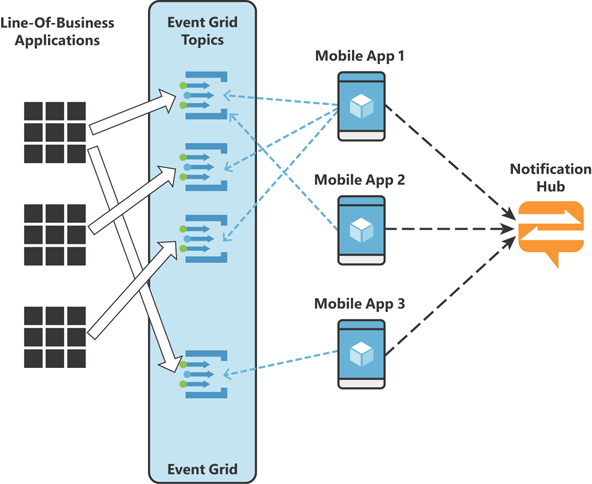
Now that you have your Notification Hub ready, you can register your PNS with your Notification Hub and send notifications to your mobile app. This step requires that you have a developer account associated with the PNS that you want to use. The following procedure shows how to create an application in the Firebase console:

1. Open the Firebase console ([*https://console.firebase.google.com*](https://console.firebase.google.com/)).
2. Click the Add Project button.
3. Type a name for your project.
4. Click the Continue button.
5. On the Google Analytics page, disable the Google Analytics For This Project.
6. Click the Create Project button.
7. Once your project is ready, click the Continue button. This forwards you to your project’s main page.
8. On your project’s main page, click the Android icon below the title Get Started By Adding Firebase To Your App.
9. On the Add Firebase To Your Android App, type a package name. You need this package name in the next example.
10. Click the Register App button.
11. Click the Download google-services.json button. You need this file later in this section.
12. Click the Next button.
13. Click the Next button again.
14. Click the Continue To Console button.
15. In the Console window, click the cog icon next to the Project Overview on the navigation panel on the left side of the console.
16. On the contextual menu, click Project Settings.
17. Click the Cloud Messaging tab.
18. In the Project Credentials section, copy the token associated with the Server Key. You need this in a later step.
19. Navigate to the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
20. Type the name of your Notification Hub in the Search Resources, Services, And Docs text box at the top of the Azure portal.
21. Click the Google (GCM/FCM) option in the Settings section on the navigation menu on the left side of your Notification Hub blade.
22. Paste the Server Key that you copied on step 18 in the API Key text box.
23. Click the Save button.

At this point, you have configured your Notification Hub for sending notifications to your Android application by using Firebase. For the sake of brevity, I omit how to program a mobile application for getting notifications from the Notification Hub. You can find examples of how to perform this notification management for each platform by reviewing the following articles:

* **iOS** [*https://docs.microsoft.com/en-us/azure/notification-hubs/ios-sdk-204*](https://docs.microsoft.com/en-us/azure/notification-hubs/ios-sdk-204)
* **Android** [*https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-android-push-notification-google-fcm-get-started*](https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-android-push-notification-google-fcm-get-started)
* **Windows Universal** [*https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-windows-store-dotnet-get-started-wns-push-notification*](https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-windows-store-dotnet-get-started-wns-push-notification)

When you need to add push notification support to your solution, you should think of the notification hub as a part of a bigger architecture. An example of this could be a solution that needs to connect your line-of-business applications with a mobile application. In such a scenario, a possible architecture could be to use Event Grid topics. The line-of-business applications would be the publishers of events to the appropriate topic, and then you can deploy one or more Azure Apps Services that are subscribed to these topics. When one of the line-of-business applications publishes an event in the Event Grid topic, your Azure App Service, which is acting as an event handler, can process the event and send a notification to your mobile users by using the Azure Notification Hub. [Figure 5-17](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig17) shows a schema of this architecture. As you can see in that figure, the key component of the architecture is the Event Grid service and the implementation of an event-driven architecture.



**Figure 5-17** Diagram of event-driven architecture, including notification hubs

***NEED MORE REVIEW?* SAMPLE ARCHITECTURE IMPLEMENTATION**

You can review a sample architecture implementation using Service Bus messages instead of Event Grid by reading the article at [*https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-enterprise-push-notification-architecture*](https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-enterprise-push-notification-architecture)*.*

**Implement solutions that use Azure Event Hub**

Azure Event Grid is an excellent service for implementing event-driven solutions, but it is only one piece of a more complex pipeline. Although Event Grid is appropriate for working with event-driven, reactive programming, It is not the best solution when you need to ingest millions of events per second with low latency.

Azure Event Hub is a more suitable solution when you require a service that can receive and process millions of events per second and provide low-latency event processing. Azure Event Hub is the front door of a big data pipeline that processes millions of events. Once the Azure Event Hub receives the data, it can deliver the event to Azure Event Grid, store the information in an Azure Blob Storage Account, or store the data in an Azure Data Lake Storage.

When you work with event hubs, you send events to the hub. The entity that sends events to the event hub is known as an event publisher. An event publisher can send events to the event hub by using any of these protocols: AMQP 1.0, Kafka 1.0 (or later), or HTTPS.

You can publish events to the event hub by sending a single event or grouping several events in a batch operation. Independently if you publish a single event or a batch of them, you are limited to a maximum size of 1 MB of data per publication. When Azure Event Hub stores an event, it distributes the different events in different partitions based on the partition key provided as one of the data of the event. Using this pattern, Azure Event Hub ensures that all events sharing the same partition key are delivered in order to the same partition.

A partition stores the events as they arrive at the partition. This way, the newer events are added to the end of the partition. You cannot delete events from a partition. Instead, you need to wait for the event to expire to be removed from the partition. As each partition is independent of other partitions in the event hub, the growth rates are different from partition to partition. You can define the number of partitions that your event hub contains during the creation of the event hub. You can create between 2 and 32 partitions, although you can extend the limit of 32 by contacting the Azure Event Hub team. Bear in mind that once you create the event hub and set the number of the partitions, you cannot change this number later. When planning the number of partitions to assign to the event hub, consider the maximum number of parallels downstream that need to connect to the event hub.

You can connect event receiver applications to an event hub by using consumer groups. A consumer group is equivalent to a downstream in a stream processing architecture. Using consumer groups, you can have different event receivers or consumers, accessing different views (state, position, or offset) of the partitions in the event hub. Event consumers connect to the event hub by using the AMQP protocol that sends the event to the client as soon as new data is available.

The following procedure shows how to create an Azure Event Hub:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Expand the navigation menu by clicking the icon with three parallels lines on the top-left corner of the Azure portal.
3. Click All Services on the navigation menu.
4. In the Search All text box, type **event**.
5. Click Event Hubs in the results list.
6. On the Event Hubs blade, click the Add button at the top-left corner of the blade.
7. On the Create Namespace panel, ensure that the correct subscription is selected in the Subscription drop-down menu.
8. Select a resource group from the Resource Group drop-down menu. Alternatively, you can create a new resource group by clicking the Create New link below the drop-down menu.
9. Type a name for the Event Hub namespace.
10. Select a location in the Location drop-down menu.
11. Select the Basic tier in the Pricing Tier drop-down menu.
12. Leave the Throughput Units as 1.
13. Click the Review + Create button at the bottom of the panel.
14. Click the Create button.
15. Navigate to your newly created Event Hub namespace.
16. On the Overview blade in the Event Hub namespace blade, click the Event Hub button.
17. On the Create Event Hub panel, type a Name for the Event Hub.
18. Leave the Partition Count at 2. Remember that you cannot change this value once the event hub is created.
19. Click the Create button.
20. Click Shared Access Policies in the navigation menu on the left side of the Event Hub namespace.
21. Click the RootManageSharedAccessKey.
22. Copy the Connection String-Primary Key value. You need this value for step 9 in the next procedure.

Once you have created your event hub’s namespace and your hub, you can start sending and consuming events from the hub. Use the following procedure to create two console applications, one for sending events and another for receiving events:

1. Open Visual Studio 2019.
2. On the Welcome screen, click Create A New Project.
3. Select the Console App (.NET Core) template.
4. Click the Next button.
5. Type a Project Name.
6. Select a location for the project.
7. Click the Create button.
8. Install the Microsoft.Azure.EventHubs NuGet package.
9. Replace the content of the Program.cs file with the content of [Listing 5-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex11). You received the Event Hub Namespace connection string in the last step of the previous procedure.

**Listing 5-11** Function1.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-11a)

**// C# .NET**

using System;

using System.Text;

using System.Threading.Tasks;

using Microsoft.Azure.EventHubs;

namespace *<your\_project\_name>*

{

class Program

{

private static EventHubClient eventHubClient;

private const string EventHubConnectionString = "*<Your\_event\_hub\_*

*namespace\_connection\_string>*";

private const string EventHubName = "*<your\_event\_hub\_name>*";

private const int numMessagesToSend = 100;

static void Main(string[] args)

{

var connectionStringBuilder = new EventHubsConnectionStringBuilder(

EventHubConnectionString)

{

EntityPath = EventHubName

};

eventHubClient = EventHubClient.CreateFromConnectionString(

connectionStringBuilder.ToString());

for (var i = 0; i < numMessagesToSend; i++)

{

try

{

var message = $"Message {i}";

Console.WriteLine($"Sending message: {message}");

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

}

catch (Exception exception)

{

Console.WriteLine($"{DateTime.Now} > Exception: {exception.Message}");

}

Task.Delay(10);

}

Console.WriteLine($"{numMessagesToSend} messages sent.");

eventHubClient.CloseAsync();

Console.WriteLine("Press ENTER to exit.");

Console.ReadLine();

}

}

}

At this point, you can press F5 and run the console application. This application console sends 100 messages to the event hub that you configured in the EventHubName constant. In the next procedure, you are going to create another application console for implementing an Event Processor Host. The Event Processor Host is an agent that helps you receive events from the event hub. The Event Processor automatically manages the persistent checkpoints and parallel event reception. The Event Processor Host requires an Azure Storage Account to process the persistent checkpoints.

***NOTE* EXAMPLE REQUIREMENTS**

You need to create an Azure Blob Storage container to run this example. You can review how to create a blob container and how to get the access key by reading the following articles:

* **Create a container** [*https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-portal#create-a-container*](https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-portal#create-a-container)
* **Get access keys** [*https://docs.microsoft.com/en-us/azure/storage/common/storage-account-manage#access-keys*](https://docs.microsoft.com/en-us/azure/storage/common/storage-account-manage#access-keys)

Follow these steps to create the console application that implements the Event Processor Host:

1. Open Visual Studio 2019.
2. On the Welcome screen, click Create A New Project.
3. Select the Console App (.NET Core) template.
4. Click the Next button.
5. Type a Project Name.
6. Select a location for the project.
7. Click the Create button.
8. Install the following NuGet packages:
   * Microsoft.Azure.EventHubs
   * Microsoft.Azure.EventHubs.Processor
9. Create a new empty C# class and name it **SimpleEventProcessor**. In later steps, this class implements the IEventProcessor interface that contains the signature of the methods needed for the Event Processor.
10. Replace the content of the SimpleEventProcessor.cs file with the content of [Listing 5-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex12).

**Listing 5-12** SimpleEventProcessor.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-12a)

**// C# .NET**

using Microsoft.Azure.EventHubs;

using Microsoft.Azure.EventHubs.Processor;

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading.Tasks;

namespace *<your\_project\_name>*

{

public class SimpleEventProcessor : IEventProcessor

{

public Task CloseAsync(PartitionContext context, CloseReason reason)

{

Console.WriteLine($"Processor Shutting Down. Partition

'{context.PartitionId}', Reason: '{reason}'.");

return Task.CompletedTask;

}

public Task OpenAsync(PartitionContext context)

{

Console.WriteLine($"SimpleEventProcessor initialized. Partition: '{context.

PartitionId}'");

return Task.CompletedTask;

}

public Task ProcessErrorAsync(PartitionContext context, Exception error)

{

Console.WriteLine($"Error on Partition: {context.PartitionId}, Error:

{error.Message}");

return Task.CompletedTask;

}

public Task ProcessEventsAsync(PartitionContext context, IEnumerable

<EventData> messages)

{

foreach (var eventData in messages)

{

var data = Encoding.UTF8.GetString(eventData.Body.Array, eventData.

Body.Offset, eventData.Body.Count);

Console.WriteLine($"Message received. Partition: '{context.

PartitionId}', Data: '{data}'");

}

return context.CheckpointAsync();

}

}

}

1. Replace the content of the Program.cs file with the content of the [Listing 5-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex13).

**Listing 5-13** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-13a)

// C# .NET

using Microsoft.Azure.EventHubs;

using Microsoft.Azure.EventHubs.Processor;

using System;

namespace *<your\_project\_name>*

{

class Program

{

private const string EventHubConnectionString =

"*<your\_event\_hub\_namespace\_connection\_string>*";

private const string EventHubName = "*<your\_event\_hub\_name>*";

private const string StorageContainerName = "*<your\_container\_name>*";

private const string StorageAccountName = "*<your\_storage\_account\_name>*";

private const string StorageAccountKey = "*<your\_storage\_account\_access\_key>*";

private static readonly string StorageConnectionString = string.Format(

$"DefaultEndpointsProtocol=https;

AccountName={StorageAccountName};

AccountKey={StorageAccountKey}");

static void Main(string[] args)

{

Console.WriteLine("Registering EventProcessor...");

var eventProcessorHost = new EventProcessorHost(

EventHubName,

PartitionReceiver.DefaultConsumerGroupName,

EventHubConnectionString,

StorageConnectionString,

StorageContainerName);

// Registers the Event Processor Host and starts receiving messages

eventProcessorHost.RegisterEventProcessorAsync<SimpleEventProcessor>();

Console.WriteLine("Receiving. Press ENTER to stop worker.");

Console.ReadLine();

// Disposes of the Event Processor Host

eventProcessorHost.UnregisterEventProcessorAsync();

}

}

}

Now you can press F5 and run your console application. The console application registers itself as an Event Processor and starts waiting for events not processed in the event hub. Because the default expiration time for the events in the event hub is one day, you should receive all the messages sent by your publishing console application in the previous example. If you run your event publisher console application without stopping the event processor console application, you should be able to see the messages in the event processor console almost in real time as they are sent to the event hub by the event publishing console. This simple example also shows how the event hub distributes the events across the different partitions.

**Images *EXAM TIP***

The Azure Event Hub is a service appropriate for processing huge amounts of events with low latency. You should consider the event hub as the starting point in an event processing pipeline. You can use the event hub as the event source of the Event Grid service.

***NEED MORE REVIEW?* EVENT HUBS CONCEPTS**

The Azure Event Hub service is designed to work with big data pipelines where you need to process millions of events per second. In those scenarios, making a bad decision when planning the deployment of an event hub can have a big effect on the performance. You can learn more about the event hub service by reading the article at [*https://docs.microsoft.com/en-in/azure/event-hubs/event-hubs-features*](https://docs.microsoft.com/en-in/azure/event-hubs/event-hubs-features)*.*

**Skill 5.4: Develop message-based solutions**

In the previous skill, we reviewed how to use event-driven services in which a publisher pushes a lightweight notification or event to the events management system and forgets about how the event is handled or if it is even processed.

In this section, we are going to review how to develop message-based solutions using Azure services. In general terms, a message is raw data produced by a service with the goal of being stored or processed elsewhere. This means that the publisher of the messages has an expectation of some other system or subscriber process the message. Because of this expectation, the subscriber needs to notify the publisher about the status of the message.

**This skill covers how to**

* [Implement solutions that use Azure Service Bus](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec10)
* [Implement solutions that use Azure Queue Storage queues](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05lev2sec11)

**Implement solutions that use Azure Service Bus**

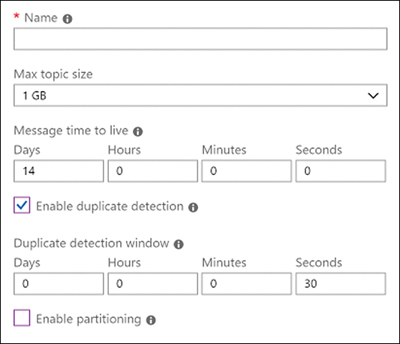
Azure Service Bus is an enterprise-level integration message broker that allows different applications to communicate with each other in a reliable way. A message is a raw data that an application sends asynchronously to the broker to be processed by another application connected to the broker. The message can contain JSON, XML, or text information.

There are some concepts that you need to review before starting to work with the Azure Service Bus:

* **Namespace** Is a container for all the components of the messaging. A single namespace can contain multiple queues and topics. You can use namespaces as application containers associating a single solution to a single namespace. The different components of your solution connect to the topics and queues in the namespace.
* **Queue** A queue is the container of messages. The queue stores the message until the receiving application retrieves and processes the message. The message queue works as a FIFO (First-In, First-Out) stack. As a new message arrives at the queue, the Service Bus service assigns a timestamp to the message. Once the message is processed, the message is held in redundant storage. Queues are appropriate for point-to-point communication scenarios in which a single application needs to communicate with another single application.
* **Topic** You use topics for sending and receiving messages. The difference between queues and topics is that topics can have several applications receiving messages used in publish/subscribe scenarios. A topic can have multiple subscriptions in which each subscription in a topic receives a copy of the message sent to the topic.

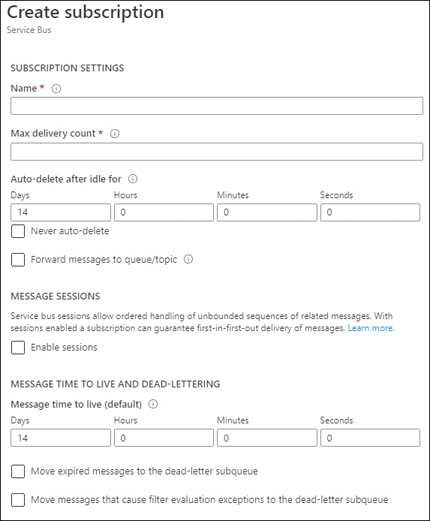
Use the following procedure to create an Azure Service Bus namespace; then, you can create a topic in the namespace. We are going to use that topic to create two console applications to send and receive the messages from the topic:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click Create A Resource at the top of the portal.
3. Click Integration in the Azure Marketplace column.
4. Click Service Bus in the Featured column.
5. On the Create Namespace panel, ensure that the correct subscription is selected in the Subscription drop-down menu.
6. Select a resource group in the Resource Group drop-down menu. Alternatively, you can create a new resource group by clicking the Create New link below the drop-down menu control.
7. Type a name for the Service Bus in the Namespace Name text box.
8. Select a location in the Location drop-down menu.
9. Select the Standard tier in the Pricing Tier drop-down menu. You cannot create topics in the Basic pricing tier; you need to use at least the Standard tier.
10. Click the Review + Create button at the bottom of the panel.
11. Click the Create button.
12. Go to the resource once the Azure Resource Manager finishes the deployment of your new Service Bus Namespace.
13. On the Overview blade in the Service Bus Namespace, click the Topic button.
14. On the Create Topic panel, shown in [Figure 5-18](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig18), type a Name for the topic.



**Figure 5-18** Creating a new topic

1. Leave the Max Topic Size and Message Time To Live parameters as they are.
2. Check Enable Duplicate Detection. This option ensures that the topic doesn’t store duplicated messages during the configured detection window.
3. Click the Create button.
4. Click Shared Access Policies on the navigation menu on the left side of the Service Bus Namespace.
5. Click the RootManageSharedAccessKey policy.
6. Copy the Primary Connection String. You are going to use the connection string later in this section.
7. Click Topics in the navigation menu on the left side of the Service Bus Namespace.
8. Click your topic.
9. On the Overview blade on the Service Bus Topic, click the Subscription button.
10. On the Create Subscription panel, shown in [Figure 5-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig19), type a name for the subscription.



**Figure 5-19** Creating a new subscription

1. Type **10** in the Max Delivery Count text box. This is the number of retries for delivering a message before moving the message to the Dead Letter Queue.
2. Leave the other properties as they are.
3. Click the Create button at the bottom of the panel.

Now you are going to create two console applications. One console application is going to publish messages to the Service Bus Topic; the other console application is going to subscribe to the Service Bus Topic, process the message, and update the processed message. Use the following procedure to create the console application that publishes messages to the Service Bus Topic:

1. Open Visual Studio 2019.
2. On the Welcome screen, click Create A New Project.
3. Select the Console App (.NET Core) template.
4. Click Next.
5. Type a Project Name.
6. Select a location for the project.
7. Click Create.
8. Install the Microsoft.Azure.ServiceBus NuGet package
9. Replace the content of the Program.cs file with the content of [Listing 5-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex14). Remember that you copied the connection string needed for this code in step 20 of the previous example.

**Listing 5-14** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-14a)

**// C# .NET**

using Microsoft.Azure.ServiceBus;

using System;

using System.Text;

namespace *<your\_project\_name>*

{

class Program

{

const string ServiceBusConnectionString =

"*<your\_service\_bus\_connection\_string>*";

const string TopicName = "*<your\_topic\_name>*";

const int numberOfMessagesToSend = 100;

static ITopicClient topicClient;

static void Main(string[] args)

{

topicClient = new TopicClient(ServiceBusConnectionString, TopicName);

Console.WriteLine("Press ENTER key to exit after sending all the

messages.");

Console.WriteLine();

// Send messages.

try

{

for (var i = 0; i < numberOfMessagesToSend; i++)

{

// Create a new message to send to the topic.

string messageBody = $"Message {i} {DateTime.Now}";

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 topic.

topicClient.SendAsync(message);

}

}

catch (Exception exception)

{

Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");

}

Console.ReadKey();

topicClient.CloseAsync();

}

}

}

You can now press F5 and publish messages to the topic. Once you publish the messages, you should be able to see an increase in the Message Count column in the Overview blade of your Service Bus Topic. The next steps show how to create the second console application that subscribes to the topic and processes the messages in the topic:

1. Open Visual Studio 2019.
2. On the Start window, click Create A New Project.
3. Select the Console App (.NET Core) template.
4. Click Next.
5. Type a Project Name.
6. Select a location for the project.
7. Click Create.
8. Install the Microsoft.Azure.ServiceBus NuGet package.
9. Replace the content of the Program.cs file with the content of [Listing 5-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex15).

**Listing 5-15** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-15a)

**// C# .NET**

using Microsoft.Azure.ServiceBus;

using System;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace *<your\_project\_name>*

{

class Program

{

const string ServiceBusConnectionString = "*<your\_service\_bus\_*

*connection\_string>*";

const string TopicName = "*<your\_topic\_name>*";

const string SubscriptionName = "*<your\_subscription\_name>*";

static ISubscriptionClient subscriptionClient;

static void Main(string[] args)

{

subscriptionClient = new SubscriptionClient(ServiceBusConnectionString,

TopicName, SubscriptionName);

Console.WriteLine("Press ENTER key to exit after receiving all the

messages.");

// Configure the message handler options in terms of exception handling,

number of concurrent messages to deliver, etc.

var messageHandlerOptions = new MessageHandlerOptions(

ExceptionReceivedHandler)

{

// 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 the message pump should automatically complete

// the messages after returning from user callback.

// False below indicates the user callback handles the complete

// operation as in ProcessMessagesAsync().

AutoComplete = false

};

// Register the function that processes messages.

subscriptionClient.RegisterMessageHandler(ProcessMessagesAsync,

messageHandlerOptions);

Console.ReadKey();

subscriptionClient.CloseAsync();

}

static async Task ProcessMessagesAsync(Message message, CancellationToken

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 subscriptionClient is created in

// ReceiveMode.PeekLock mode (which is the default).

await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

// Note: Use the cancellationToken passed as necessary to determine if the

// subscriptionClient has already been closed.

// If subscriptionClient has already been closed, you can choose to not

// call CompleteAsync() or AbandonAsync() etc.

// to avoid unnecessary exceptions.

}

// Use this handler to examine the exceptions received on the message pump.

static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs

exceptionReceivedEventArgs)

{

Console.WriteLine($"Message handler encountered an exception

{exceptionReceivedEventArgs.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;

}

}

}

You can now press F5 and run the console application. As the console application processes the messages in the topic, you can see that the count of the messages in the subscription is decreasing.

***NEED MORE REVIEW?* SERVICE BUS ADVANCED FEATURES**

You can learn more about Service Bus in the following articles:

* **Queues, Topics, and Subscriptions** [*https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions*](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions)
* **Service Bus Performance Improvements** [*https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements*](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-performance-improvements)
* **Topic Filters and Actions** [*https://docs.microsoft.com/en-us/azure/service-bus-messaging/topic-filters*](https://docs.microsoft.com/en-us/azure/service-bus-messaging/topic-filters)

**Implement solutions that use Azure Queue Storage queues**

Azure Queue Storage is the first service that Microsoft released for managing message queues. Although Azure Service Bus and Azure Queue Storage share some features, such as providing message queue services, Azure Queue Storage is more appropriate when your application needs to store more than 80 GB of messages in a queue. Another important feature of the Azure Queue Storage service that you need to consider is although the queues in the service work as a FIFO (First-In, First-Out) stack, the order of the message is not guaranteed.

***NOTE* AZURE QUEUE STORAGE VS. AZURE SERVICE BUS**

You can review a complete list of differences between these two queuing services at [*https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted*](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted)*.*

The maximum size of a single message that you can send to an Azure Queue is 64KB, although the total size of the queue can grow to over 80GB. You can only access an Azure Queue using the REST API or using the .NET Azure Storage SDK. Here are the steps for creating an Azure Queue Storage Account and a queue for sending and receiving messages:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click Create A Resource at the top of the portal.
3. Click Storage in the Azure Marketplace column.
4. Click Storage Account – Blob, File, Table, Queue in the Featured column.
5. On the Create Storage Account blade, select a subscription in the Subscription drop-down menu.
6. Select a resource group in the Resource Group drop-down menu.
7. Type a Storage Account Name.
8. Select a location in the Location drop-down menu.
9. Select Locally-Redundant Storage in the Replication drop-down menu.
10. Leave the other properties as is.
11. Click the Review + Create button.
12. Click the Create button.
13. Click the Go To Resource button once the deployment finishes.
14. Click Access Keys on the navigation menu in the Azure Storage account blade.
15. Copy the Connection String from the key1 section. You need this value later in this section.

At this point, you can create queues in your Azure Storage account by using the Azure portal. You can also add messages to the queue using the Azure portal. This approach is useful for development or testing purposes, but it is not suitable for applications. Use the following steps to create a console application that creates a new queue in your Azure Storage Account. The application also sends and reads messages from the queue:

1. On the Welcome screen, click Create A New Project.
2. Select the Console App (.NET Core) template.
3. Click Next.
4. Type a Project Name.
5. Select a location for the project.
6. Click Create.
7. Install the following NuGet packages:
   * Azure.Storage.Common
   * Azure.Storage.Queue
8. Replace the content of the Program.cs file with the content of [Listing 5-16](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05ex16).

**Listing 5-16** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05_images.xhtml#lis5-16a)

**// C# .NET**

using Azure.Storage.Queues;

using Azure.Storage.Queues.Models;

using System;

namespace *<your\_project\_name>*

{

class Program

{

private const string connectionString = "*<your\_storage\_account\_*

*connection\_string>*";

private const string queueName = "az204queue";

private const int maxNumOfMessages = 10;

static void Main(string[] args)

{

QueueClient queueClient = new QueueClient(connectionString, queueName);

//Create the queue

queueClient.CreateIfNotExists();

//Sending messages to the queue.

for (int i = 0; i < maxNumOfMessages; i++)

{

queueClient.SendMessageAsync($"Message {i} {DateTime.Now}");

}

//Getting the length of the queue

QueueProperties queueProperties = queueClient.GetProperties();

int? cachedMessageCount = queueProperties.ApproximateMessagesCount;

//Reading messages from the queue without removing the message

Console.WriteLine("Reading message from the queue without removing them

from the queue");

PeekedMessage[] peekedMessages = queueClient.PeekMessages((int)

cachedMessageCount);

foreach (PeekedMessage peekedMessage in peekedMessages)

{

Console.WriteLine($"Message read from the queue: {peekedMessage.

MessageText}");

//Getting the length of the queue

queueProperties = queueClient.GetProperties();

int? queueLenght = queueProperties.ApproximateMessagesCount;

Console.WriteLine($"Current lenght of the queue {queueLenght}");

}

//Reading messages removing it from the queue

Console.WriteLine("Reading message from the queue removing");

QueueMessage[] messages = queueClient.ReceiveMessages((int)

cachedMessageCount);

foreach (QueueMessage message in messages)

{

Console.WriteLine($"Message read from the queue: {message.

MessageText}");

//You need to process the message in less than 30 seconds.

queueClient.DeleteMessage(message.MessageId,message.PopReceipt);

//Getting the length of the queue

queueProperties = queueClient.GetProperties();

int? queueLenght = queueProperties.ApproximateMessagesCount;

Console.WriteLine($"Current lenght of the queue {queueLenght}");

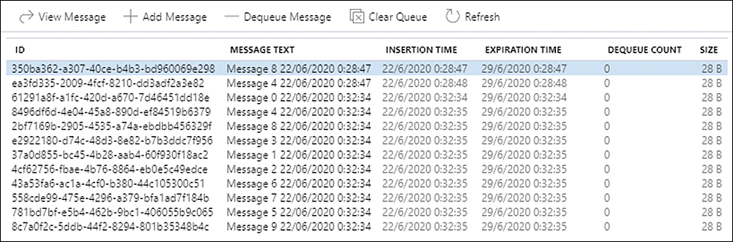
}

}

}

}

Press F5 to execute the console application that sends and reads messages from the queue. You can see how the messages are added to the queue by using the Azure portal and navigating to your Azure Storage account > Queues > az204queue. You should see a queue similar to one shown in [Figure 5-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05fig20).



**Figure 5-20** Creating a new subscription

***NEED MORE REVIEW?* PUBLISH-SUBSCRIBE PATTERN**

Although the Azure Queue Storage service doesn’t provide the ability to create subscriptions to the queues, you can easily implement the publish-subscribe pattern for communicating applications using Azure Queue Storage. You can learn how to implement this pattern by reviewing the article at [*https://docs.microsoft.com/en-us/learn/modules/communicate-between-apps-with-azure-queue-storage/*](https://docs.microsoft.com/en-us/learn/modules/communicate-between-apps-with-azure-queue-storage/)*.*

**Chapter summary**

* Azure App Service Logic Apps allows you to interconnect different services without needing to create specific code for the interconnection.
* Logic App Workflows define the steps needed to exchange information between applications.
* Microsoft provides connectors for getting and sending information to and from different services.
* Triggers are events fired on the source systems.
* Actions are each of the steps performed in a workflow.
* Azure Logic Apps provides a graphical editor that eases the process of creating workflows.
* You can create your custom connectors for connecting your application with Azure Logic Apps.
* A Custom Connector is a wrapper for a REST or SOAP API.
* You can create custom connectors for Azure Logic Apps, Microsoft Flow, and Microsoft PowerApps.
* You cannot reuse custom connectors created for Microsoft Flow or Microsoft PowerApps with Azure Logic Apps.
* You can export your Logic Apps as Azure Resource Manager templates.
* You can edit and modify the Logic Apps templates in Visual Studio.
* The API Management service allows you to publish your back-end REST or SOAP APIs using a common and secure front end.
* You need to create subscriptions in the APIM service for authenticating the access to the API.
* You need to create a product for publishing a back-end API.
* You can publish only some operations of your back-end APIs.
* APIM Policies allow you to modify the behavior of the APIM gateway.
* An event is a change in the state of an entity.
* In an event-driven architecture, the publisher doesn’t have the expectation that the event is processed or stored by a subscriber.
* Azure Event Grid is a service for implementing event-driven architectures.
* An Event Grid Topic is an endpoint where a publisher service can send events.
* Subscribers are services that read events from an Event Grid Topic.
* You can configure several types of services as event sources or event subscribers in Azure Event Grid.
* You can create custom events for sending them to the Event Grid.
* You can subscribe to your custom application with an Event Grid Topic by using WebHooks.
* The Azure Notification Hub is a service that unifies the push notifications on mobile platforms.
* You can connect the push notification services from the different manufacturers to the Azure Notification Hub.
* The Azure Event Hub is the entry point for Big Data event pipelines.
* Azure Event Hub is specialized in ingesting millions of events per second with low latency.
* You can use Azure Event Hub as an event source for the Event Grid service.
* You can use AMQP, Kafka, and HTTPS for connecting to Azure Event Hub.
* In a message-driven architecture, the publisher application has the expectation that the message is processed or stored by the subscriber.
* The subscriber needs to change the state once the message is processed.
* A message is raw data sent by a publisher that needs to be processed by a subscriber.
* Azure Service Bus and Azure Queue message are message broker services.

**Thought experiment**

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find answers to this thought experiment in the next section.

Your organization has several Line-Of-Business (LOB) applications deployed on Azure and on-premises environments. The information managed by some of these LOB applications overlaps between applications. All your LOB applications allow you to use SOAP or REST API for connecting to the applications.

Your organization needs to implement some business processes that require sharing information between the LOB applications. Answer the following questions about connecting Azure services and third-party applications:

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que1a) You need to implement a business process that requires that an application deployed in Azure share information with an application deployed in your company’s on-premises datacenter. How can you implement this business process?

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que2a) Your company needs to share some information managed by one of the LOB applications with a partner. The LOB application uses a SOAP API for accessing the data. You need to ensure that the partner is authenticated before accessing the information. Your partner needs to get the information from your application in JSON format, so you also need to ensure that the information provided by your application is published using a REST API. Which service should you use?

[**3.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que3a) One of the LOB applications of your company is becoming obsolete. Your company decides to develop a new web application for replacing the legacy LOB application. You are designing the architecture for the new web application. You need to implement a decoupled 311architecture that needs to process millions of events per second. Which service should you use?

**Thought experiment answers**

This section contains the solution to the thought experiment. Each answer explains why the answer choice is correct.

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que1) You should use Azure Logic Apps for implementing the business process. Azure Logic Apps allows you to create workflows that can be used to implement your business process. You can connect Azure Logic Apps with your on-premises LOB applications by using the on-premises data gateway. You also need to create custom connectors for Azure Logic Apps being able to work with your LOB applications.

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que2) You should use the API Management service. This service allows you to share your backed APIs with partners and external developers securely. Using the APIM policies, you can also convert the XML message provided by the SOAP API to JSON documents needed for REST APIs. You can use Azure AD, mutual certificate authentication, or API keys for authenticating access to the API.

[**3.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch05.xhtml#ch05que3) You should use Azure Event Hub. This service is specially designed to ingest millions of events per second. Once the service has ingested the events, you forward the event to other services like Azure Storage, Azure Data Lake, or Azure Event Grid. The critical point here for choosing Azure Event Hub instead of Event Grid is the number of events that need to be ingested. Another clue for choosing Event Hub instead of Azure Queue Storage or Azure Service Bus is you need to process events instead of messages. Azure Queue or Azure Service Bus are services aimed to use in message-driven architectures.