

Microsoft Official Course



AZ-203T02

Develop Azure Platform
as a Service compute
solutions

AZ-203T02

**Develop Azure Platform as a
Service compute solutions**

MCT USE ONLY. STUDENT USE PROHIBITED



Contents

■	Welcome to the course	1
	Start Here	1
■	Create Azure App Service Web Apps	5
	Azure App Service core concepts	5
	Creating an Azure App Service Web App	21
	Creating background tasks by using WebJobs in Azure App Service	29
	Review questions	35
■	Create Azure App Service mobile apps	37
	Getting Started with mobile apps in App Service	37
	Enabling push notifications for your app	43
	Enabling offline sync for your app	51
	Review questions	56
■	Create Azure App Service API apps	57
	Creating APIs	57
	Using Swagger to document an API	70
	Review questions	80
■	Implement Azure functions	81
	Azure Functions overview	81
	Develop Azure Functions using Visual Studio	94
	Implement Durable Functions	100
	Review questions	125



Welcome to the course

Start Here

Welcome

Welcome to the **Develop Azure Platform as a Service compute solutions** course. This course is part of a series of courses to help you prepare for the **AZ-203: Developing Solutions for Microsoft Azure**¹ certification exam.

Who should take this exam?

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

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

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

Exam study areas

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

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

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

AZ-203 Study Areas	Weight
Develop for Azure storage	15-20%
Implement Azure security	10-15%
Monitor, troubleshoot, and optimize Azure solutions	15-20%
Connect to and consume Azure, and third-party, services	20-25%

✓ This course will focus on preparing you for the **Develop Azure Platform as a Service compute solutions** area of the AZ-203 certification exam.

Course description

In this course students will gain the knowledge and skills needed to implement Azure Platform as a Service features and services in their development solutions. Students will learn how to create and manage Azure App Service resources, integrate push and offline sync in their mobile apps, and how to document an API. Students will also learn how to create and test Azure Functions.

Level: Intermediate

Audience:

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

Course Syllabus

Module 1: Create App Service web apps

- Azure App Service core concepts
- Creating an Azure App Service web app
- Creating background tasks by using WebJobs in Azure App Service

Module 2: Creating Azure App Service mobile apps

- Getting started with mobile apps in App Service
- Enable push notifications for your app
- Enable offline sync for your app

Module 3: Create Azure App Service API apps

- Creating APIs
- Using Swagger to document an API

Module 4: Implement Azure Functions

- Azure Functions overview
- Develop Azure Functions using Visual Studio

- Implement durable functions



Create Azure App Service Web Apps

Azure App Service core concepts

Web Apps Overview

Azure App Service web apps (or just Web Apps) is a service for hosting web applications, REST APIs, and mobile back ends. You can develop in your favorite language, be it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Applications run and scale with ease on Windows-based environments.

Web Apps not only adds the power of Microsoft Azure to your application, such as security, load balancing, autoscaling, and automated management. You can also take advantage of its DevOps capabilities, such as continuous deployment from VSTS, GitHub, Docker Hub, and other sources, package management, staging environments, custom domain, and SSL certificates.

With App Service, you pay for the Azure compute resources you use. The compute resources you use is determined by the *App Service plan* that you run your Web Apps on.

Key features of App Service Web Apps

Here are some key features of App Service Web Apps:

- **Multiple languages and frameworks** - Web Apps has first-class support for ASP.NET, ASP.NET Core, Java, Ruby, Node.js, PHP, or Python. You can also run PowerShell and other scripts or executables as background services.
- **DevOps optimization** - Set up continuous integration and deployment with Visual Studio Team Services, GitHub, BitBucket, Docker Hub, or Azure Container Registry. Promote updates through test and staging environments. Manage your apps in Web Apps by using Azure PowerShell or the cross-platform command-line interface (CLI).
- **Global scale with high availability** - Scale up or out manually or automatically. Host your apps anywhere in Microsoft's global datacenter infrastructure, and the App Service SLA promises high availability.
- **Connections to SaaS platforms and on-premises data** - Choose from more than 50 connectors for enterprise systems (such as SAP), SaaS services (such as Salesforce), and internet services (such as Facebook). Access on-premises data using Hybrid Connections and Azure Virtual Networks.

- **Security and compliance** - App Service is ISO, SOC, and PCI compliant. Authenticate users with Azure Active Directory or with social login (Google, Facebook, Twitter, and Microsoft). Create IP address restrictions and manage service identities.
- **Application templates** - Choose from an extensive list of application templates in the Azure Marketplace, such as WordPress, Joomla, and Drupal.
- **Visual Studio integration** - Dedicated tools in Visual Studio streamline the work of creating, deploying, and debugging.
- **API and mobile features** - Web Apps provides turn-key CORS support for RESTful API scenarios, and simplifies mobile app scenarios by enabling authentication, offline data sync, push notifications, and more.
- **Serverless code** - Run a code snippet or script on-demand without having to explicitly provision or manage infrastructure, and pay only for the compute time your code actually uses.

Besides Web Apps in App Service, Azure offers other services that can be used for hosting websites and web applications. For most scenarios, Web Apps is the best choice. For microservice architecture, consider Service Fabric. If you need more control over the VMs that your code runs on, consider Azure Virtual Machines.

Azure App Service plans

In App Service, an app runs in an *App Service plan*. An App Service plan defines a set of compute resources for a web app to run. These compute resources are analogous to the *server farm* in conventional web hosting. One or more apps can be configured to run on the same computing resources (or in the same App Service plan).

When you create an App Service plan in a certain region (for example, West Europe), a set of compute resources is created for that plan in that region. Whatever apps you put into this App Service plan run on these compute resources as defined by your App Service plan. Each App Service plan defines:

- Region (West US, East US, etc.)
- Number of VM instances
- Size of VM instances (Small, Medium, Large)
- Pricing tier (Free, Shared, Basic, Standard, Premium, PremiumV2, Isolated, Consumption)

The *pricing tier* of an App Service plan determines what App Service features you get and how much you pay for the plan. There are a few categories of pricing tiers:

- **Shared compute:** **Free** and **Shared**, the two base tiers, runs an app on the same Azure VM as other App Service apps, including apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources cannot scale out.
- **Dedicated compute:** The **Basic**, **Standard**, **Premium**, and **PremiumV2** tiers run apps on dedicated Azure VMs. Only apps in the same App Service plan share the same compute resources. The higher the tier, the more VM instances are available to you for scale-out.
- **Isolated:** This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks, which provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities.
- **Consumption:** This tier is only available to *function apps*. It scales the functions dynamically depending on workload.

Note: App Service Free and Shared (preview) hosting plans are base tiers that run on the same Azure VM as other App Service apps. Some apps may belong to other customers. These tiers are intended to be used only for development and testing purposes.

Each tier also provides a specific subset of App Service features. These features include custom domains and SSL certificates, autoscaling, deployment slots, backups, Traffic Manager integration, and more. The higher the tier, the more features are available. To find out which features are supported in each pricing tier, see App Service plan details.

How does my app run and scale?

In the **Free** and **Shared** tiers, an app receives CPU minutes on a shared VM instance and cannot scale out. In other tiers, an app runs and scales as follows.

When you create an app in App Service, it is put into an App Service plan. When the app runs, it runs on all the VM instances configured in the App Service plan. If multiple apps are in the same App Service plan, they all share the same VM instances. If you have multiple deployment slots for an app, all deployment slots also run on the same VM instances. If you enable diagnostic logs, perform backups, or run WebJobs, they also use CPU cycles and memory on these VM instances.

In this way, the App Service plan is the scale unit of the App Service apps. If the plan is configured to run five VM instances, then all apps in the plan run on all five instances. If the plan is configured for autoscaling, then all apps in the plan are scaled out together based on the autoscale settings.

What if my app needs more capabilities or features?

Your App Service plan can be scaled up and down at any time. It is as simple as changing the pricing tier of the plan. You can choose a lower pricing tier at first and scale up later when you need more App Service features.

For example, you can start testing your web app in a Free App Service plan and pay nothing. When you want to add your custom DNS name to the web app, just scale your plan up to Shared tier. Later, when you want to add a custom SSL certificate, scale your plan up to Basic tier. When you want to have staging environments, scale up to Standard tier. When you need more cores, memory, or storage, scale up to a bigger VM size in the same tier.

The same works in the reverse. When you feel you no longer need the capabilities or features of a higher tier, you can scale down to a lower tier, which saves you money.

If your app is in the same App Service plan with other apps, you may want to improve the app's performance by isolating the compute resources. You can do it by moving the app into a separate App Service plan.

Should I put an app in a new plan or an existing plan?

Since you pay for the computing resources your App Service plan allocates, you can potentially save money by putting multiple apps into one App Service plan. You can continue to add apps to an existing plan as long as the plan has enough resources to handle the load. However, keep in mind that apps in the same App Service plan all share the same compute resources. To determine whether the new app has the necessary resources, you need to understand the capacity of the existing App Service plan, and the expected load for the new app. Overloading an App Service plan can potentially cause downtime for your new and existing apps.

Isolate your app into a new App Service plan when:

- The app is resource-intensive.
- You want to scale the app independently from the other apps the existing plan.
- The app needs resource in a different geographical region.

This way you can allocate a new set of resources for your app and gain greater control of your apps.

Authentication and authorization in Azure App Service

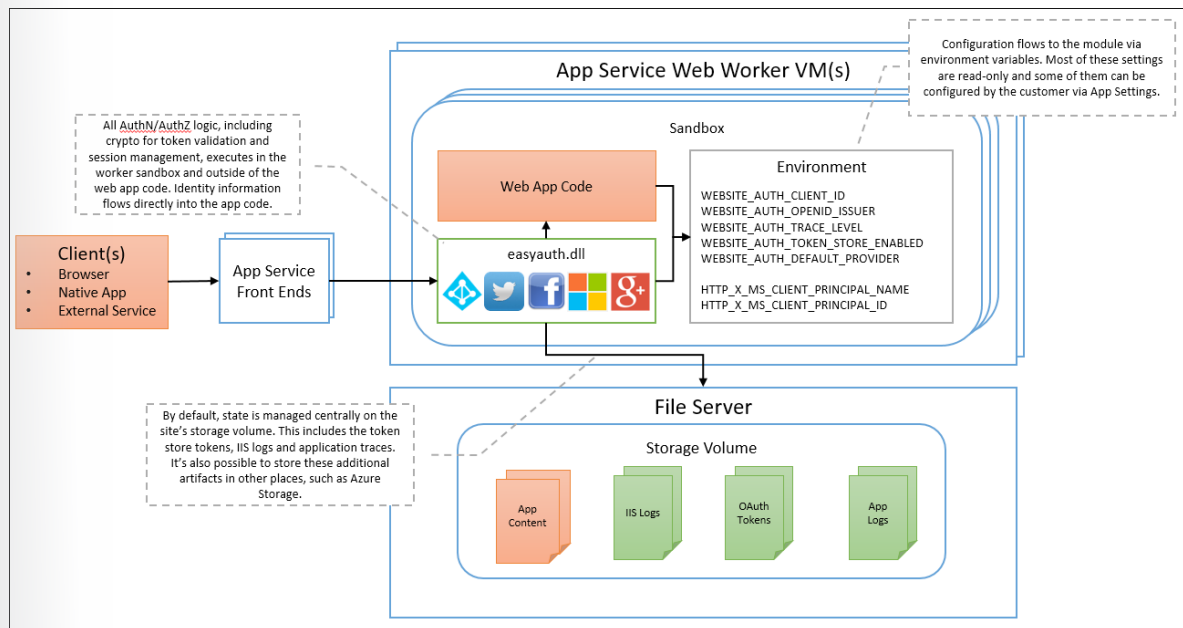
Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your web app, API, and mobile back end, and also Azure Functions.

Secure authentication and authorization require deep understanding of security, including federation, encryption, JSON web tokens (JWT) management, grant types, and so on. App Service provides these utilities so that you can spend more time and energy on providing business value to your customer.

Note: You're not required to use App Service for authentication and authorization. Many web frameworks are bundled with security features, and you can use them if you like. If you need more flexibility than App Service provides, you can also write your own utilities.

How it works

The authentication and authorization module runs in the same sandbox as your application code. When it's enabled, every incoming HTTP request passes through it before being handled by your application code.



This module handles several things for your app:

- Authenticates users with the specified provider
- Validates, stores, and refreshes tokens

- Manages the authenticated session
- Injects identity information into request headers

The module runs separately from your application code and is configured using app settings. No SDKs, specific languages, or changes to your application code are required.

User claims

For all language frameworks, App Service makes the user's claims available to your code by injecting them into the request headers. For ASP.NET 4.6 apps, App Service populates `ClaimsPrincipal.Current` with the authenticated user's claims, so you can follow the standard .NET code pattern, including the `[Authorize]` attribute. Similarly, for PHP apps, App Service populates the `$_SERVER['REMOTE_USER']` variable.

For Azure Functions, `ClaimsPrincipal.Current` is not hydrated for .NET code, but you can still find the user claims in the request headers.

For more information, see **Access user claims**¹.

Token store

App Service provides a built-in token store, which is a repository of tokens that are associated with the users of your web apps, APIs, or native mobile apps. When you enable authentication with any provider, this token store is immediately available to your app. If your application code needs to access data from these providers on the user's behalf, such as:

- post to the authenticated user's Facebook timeline
- read the user's corporate data from the Azure Active Directory Graph API or even the Microsoft Graph

You typically must write code to collect, store, and refresh these tokens in your application. With the token store, you just retrieve the tokens when you need them and tell App Service to refresh them when they become invalid.

The id tokens, access tokens, and refresh tokens cached for the authenticated session, and they're accessible only by the associated user.

If you don't need to work with tokens in your app, you can disable the token store.

Logging and tracing

If you enable application logging, you will see authentication and authorization traces directly in your log files. If you see an authentication error that you didn't expect, you can conveniently find all the details by looking in your existing application logs. If you enable failed request tracing, you can see exactly what role the authentication and authorization module may have played in a failed request. In the trace logs, look for references to a module named `EasyAuthModule_32/64`.

Identity providers

App Service uses federated identity, in which a third-party identity provider manages the user identities and authentication flow for you. Five identity providers are available by default:

¹ <https://docs.microsoft.com/en-us/azure/app-service/app-service-authentication-how-to#access-user-claims>

Provider	Sign-in endpoint
Azure Active Directory	<code>/.auth/login/aad</code>
Microsoft Account	<code>/.auth/login/microsoftaccount</code>
Facebook	<code>/.auth/login/facebook</code>
Google	<code>/.auth/login/google</code>
Twitter	<code>/.auth/login/twitter</code>

When you enable authentication and authorization with one of these providers, its sign-in endpoint is available for user authentication and for validation of authentication tokens from the provider. You can provide your users with any number of these sign-in options with ease. You can also integrate another identity provider or your own custom identity solution.

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- **Without provider SDK:** The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*. This case applies to web apps. It also applies to native apps that sign users in using the Mobile Apps client SDK because the SDK opens a web view to sign users in with App Service authentication.
- **With provider SDK:** The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This case applies to REST APIs, Azure Functions, and JavaScript browser clients, as well as web apps that need more flexibility in the sign-in process. It also applies to native mobile apps that sign users in using the provider's SDK.

Note: Calls from a trusted browser app in App Service calls another REST API in App Service or Azure Functions can be authenticated using the server-directed flow. For more information, see **Customize authentication and authorization in App Service**².

The table below shows the steps of the authentication flow.

Step	Without provider SDK	With provider SDK
1. Sign user in	Redirects client to <code>/.auth/login/<provider></code> .	Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation.
2. Post-authentication	Provider redirects client to <code>/.auth/login/<provider>/callback</code> .	Client code posts token from provider to <code>/.auth/login/<provider></code> for validation.
3. Establish authenticated session	App Service adds authenticated cookie to response.	App Service returns its own authentication token to client code.

² <https://docs.microsoft.com/en-us/azure/app-service/app-service-authentication-how-to>

Step	Without provider SDK	With provider SDK
4. Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in X-ZUMO-AUTH header (automatically handled by Mobile Apps client SDKs).

For client browsers, App Service can automatically direct all unauthenticated users to `/.auth/login/<provider>`. You can also present users with one or more `/.auth/login/<provider>` links to sign in to your app using their provider of choice.

Authorization behavior

In the Azure portal, you can configure App Service authorization with a number of behaviors.

Allow all requests (default)

Authentication and authorization are not managed by App Service (turned off).

Choose this option if you don't need authentication and authorization, or if you want to write your own authentication and authorization code.

Allow only authenticated requests

The option is **Log in with <provider>**. App Service redirects all anonymous requests to `/.auth/login/<provider>` for the provider you choose. If the anonymous request comes from a native mobile app, the returned response is an HTTP 401 Unauthorized.

With this option, you don't need to write any authentication code in your app. Finer authorization, such as role-specific authorization, can be handled by inspecting the user's claims.

Allow all requests, but validate authenticated requests

The option is **Allow Anonymous requests**. This option turns on authentication and authorization in App Service, but defers authorization decisions to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers.

This option provides more flexibility in handling anonymous requests. For example, it lets you present multiple sign-in providers to your users. However, you must write code.

OS and runtime patching in Azure App Service

App Service is a Platform-as-a-Service, which means that the OS and application stack are managed for you by Azure; you only manage your application and its data. More control over the OS and application stack is available to you in Azure Virtual Machines. With that in mind, it is nevertheless helpful for you as an App Service user to know more information, such as:

- How and when are OS updates applied?
- How is App Service patched against significant vulnerabilities (such as zero-day)?
- Which OS and runtime versions are running your apps?

For security reasons, certain specifics of security information are not published. However, the article aims to alleviate concerns by maximizing transparency on the process, and how you can stay up-to-date on security-related announcements or runtime updates.

How and when are OS updates applied?

Azure manages OS patching on two levels, the physical servers and the guest virtual machines (VMs) that run the App Service resources. Both are updated monthly, which aligns to the monthly **Patch Tuesday**³ schedule. These updates are applied automatically, in a way that guarantees the high-availability SLA of Azure services.

How does Azure deal with significant vulnerabilities?

When severe vulnerabilities require immediate patching, such as zero-day vulnerabilities, the high-priority updates are handled on a case-by-case basis. Stay current with critical security announcements in Azure by visiting **Azure Security Blog**⁴.

When are supported language runtimes updated, added, or deprecated?

New stable versions of supported language runtimes (major, minor, or patch) are periodically added to App Service instances. Some updates overwrite the existing installation, while others are installed side by side with existing versions. An overwrite installation means that your app automatically runs on the updated runtime. A side-by-side installation means you must manually migrate your app to take advantage of a new runtime version. For more information, see one of the subsections below.

Runtime updates and deprecations are announced here:

- <https://azure.microsoft.com/updates/?product=app-service>
- <https://github.com/Azure/app-service-announcements/issues>

Note: Information here applies to language runtimes that are built into an App Service app. A custom runtime you upload to App Service, for example, remains unchanged unless you manually upgrade it.

New patch updates

Patch updates to .NET, PHP, Java SDK, or Tomcat/Jetty version are applied automatically by overwriting the existing installation with the new version. Node.js patch updates are installed side by side with the existing versions (similar to major and minor versions in the next section). New Python patch versions can be installed manually through site extensions, side by side with the built-in Python installations.

New major and minor versions

When a new major or minor version is added, it is installed side by side with the existing versions. You can manually upgrade your app to the new version. If you configured the runtime version in a configuration file (such as `web.config` and `package.json`), you need to upgrade with the same method. If you used an App Service setting to configure your runtime version, you can change it in the Azure portal or by running an Azure CLI command in the Cloud Shell, as shown in the following examples:

```
az webapp config set --net-framework-version v4.7 --resource-group <groupname> --name <appname>
az webapp config set --php-version 7.0 --resource-group <groupname> --name <appname>
az webapp config appsettings set --settings WEBSITE_NODE_DEFAULT_VERSION
```

³ <https://technet.microsoft.com/security/bulletins.aspx>

⁴ <https://azure.microsoft.com/blog/topics/security/>

```

SION=8.9.3 --resource-group <groupname> --name <appname>
az webapp config set --python-version 3.4 --resource-group <groupname>
--name <appname>
az webapp config set --java-version 1.8 --java-container Tomcat --java-con-
tainer-version 9.0 --resource-group <groupname> --name <appname>

```

Deprecated versions

When an older version is deprecated, the removal date is announced so that you can plan your runtime version upgrade accordingly.

How can I query OS and runtime update status on my instances?

While critical OS information is locked down from access, the **Kudu console**⁵ enables you to query your App Service instance regarding the OS version and runtime versions.

The following table shows how to the versions of Windows and of the language runtime that are running your apps:

Information	Where to find it
Windows version	See <a href="https://<appname>.scm.azurewebsites.net/Env.cshtml">https://<appname>.scm.azurewebsites.net/Env.cshtml (under System info)
.NET version	At <a href="https://<appname>.scm.azurewebsites.net/DebugConsole">https://<appname>.scm.azurewebsites.net/DebugConsole , run the following command in the command prompt: powershell -command "gci 'Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Net Framework Setup\NDP\CDF'"
.NET Core version	At <a href="https://<appname>.scm.azurewebsites.net/DebugConsole">https://<appname>.scm.azurewebsites.net/DebugConsole , run the following command in the command prompt: dotnet --version
PHP version	At <a href="https://<appname>.scm.azurewebsites.net/DebugConsole">https://<appname>.scm.azurewebsites.net/DebugConsole , run the following command in the command prompt: php --version
Default Node.js version	In the Cloud Shell, run the following command: az webapp config appsettings list --resource-group <groupname> --name <appname> --query "[?name=='WEBSITE_NODE_DEFAULT_VERSION']"
Python version	At <a href="https://<appname>.scm.azurewebsites.net/DebugConsole">https://<appname>.scm.azurewebsites.net/DebugConsole , run the following command in the command prompt: python --version

⁵ <https://github.com/projectkudu/kudu/wiki/Kudu-console>

Note: Access to registry location `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Component Based Servicing\Packages`, where information on “KB” patches is stored, is locked down.

Inbound and outbound IP addresses in Azure App Service

Azure App Service is a multi-tenant service, except for App Service Environments. Apps that are not in an App Service environment (not in the Isolated tier) share network infrastructure with other apps. As a result, the inbound and outbound IP addresses of an app can be different, and can even change in certain situations.

App Service Environments use dedicated network infrastructures, so apps running in an App Service environment get static, dedicated IP addresses both for inbound and outbound connections.

When inbound IP changes

Regardless of the number of scaled-out instances, each app has a single inbound IP address. The inbound IP address may change when you perform one of the following actions:

- Delete an app and recreate it in a different resource group.
- Delete the last app in a resource group and region combination and recreate it.
- Delete an existing SSL binding, such as during certificate renewal.

Get static inbound IP

Sometimes you might want a dedicated, static IP address for your app. To get a static inbound IP address, you need to configure an IP-based SSL binding. If you don't actually need SSL functionality to secure your app, you can even upload a self-signed certificate for this binding. In an IP-based SSL binding, the certificate is bound to the IP address itself, so App Service provisions a static IP address to make it happen.

When outbound IPs change

Regardless of the number of scaled-out instances, each app has a set number of outbound IP addresses at any given time. Any outbound connection from the App Service app, such as to a back-end database, uses one of the outbound IP addresses as the origin IP address. You can't know beforehand which IP address a given app instance will use to make the outbound connection, so your back-end service must open its firewall to all the outbound IP addresses of your app.

The set of outbound IP addresses for your app changes when you scale your app between the lower tiers (**Basic**, **Standard**, and **Premium**) and the **Premium V2** tier.

You can find the set of all possible outbound IP addresses your app can use, regardless of pricing tiers, by looking for the `possibleOutboundIPAddresses` property.

Find outbound IPs

To find the outbound IP addresses currently used by your app in the Azure portal, click **Properties** in your app's left-hand navigation.

You can find the same information by running the following command in the Cloud Shell.

```
az webapp show --resource-group <group_name> --name <app_name> --query
outboundIpAddresses --output tsv
```

To find all possible outbound IP addresses for your app, regardless of pricing tiers, run the following command in the Cloud Shell.

```
az webapp show --resource-group <group_name> --name <app_name> --query
possibleOutboundIpAddresses --output tsv
```

Azure App Service Hybrid Connections

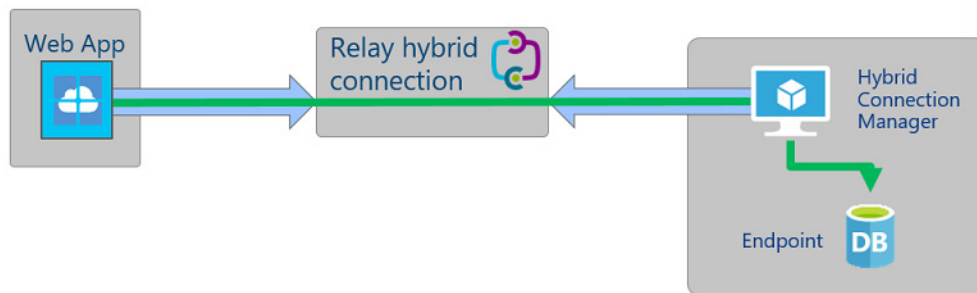
Hybrid Connections is both a service in Azure and a feature in Azure App Service. As a service, it has uses and capabilities beyond those that are used in App Service. To learn more about Hybrid Connections and their usage outside App Service, see **Azure Relay Hybrid Connections**⁶.

Within App Service, Hybrid Connections can be used to access application resources in other networks. It provides access from your app to an application endpoint. It does not enable an alternate capability to access your application. As used in App Service, each Hybrid Connection correlates to a single TCP host and port combination. This means that the Hybrid Connection endpoint can be on any operating system and any application, provided you are accessing a TCP listening port. The Hybrid Connections feature does not know or care what the application protocol is, or what you are accessing. It is simply providing network access.

How it works

The Hybrid Connections feature consists of two outbound calls to Azure Service Bus Relay. There is a connection from a library on the host where your app is running in App Service. There is also a connection from the Hybrid Connection Manager (HCM) to Service Bus Relay. The HCM is a relay service that you deploy within the network hosting the resource you are trying to access.

Through the two joined connections, your app has a TCP tunnel to a fixed host:port combination on the other side of the HCM. The connection uses TLS 1.2 for security and shared access signature (SAS) keys for authentication and authorization.



When your app makes a DNS request that matches a configured Hybrid Connection endpoint, the outbound TCP traffic will be redirected through the Hybrid Connection.

Note: This means that you should try to always use a DNS name for your Hybrid Connection. Some client software does not do a DNS lookup if the endpoint uses an IP address instead.

⁶ <https://docs.microsoft.com/azure/service-bus-relay/relay-hybrid-connections-protocol/>

App Service Hybrid Connection benefits

There are a number of benefits to the Hybrid Connections capability, including:

- Apps can access on-premises systems and services securely.
- The feature does not require an internet-accessible endpoint.
- It is quick and easy to set up.
- Each Hybrid Connection matches to a single host:port combination, helpful for security.
- It normally does not require firewall holes. The connections are all outbound over standard web ports.
- Because the feature is network level, it is agnostic to the language used by your app and the technology used by the endpoint.
- It can be used to provide access in multiple networks from a single app.

Things you cannot do with Hybrid Connections

Things you cannot do with Hybrid Connections include:

- Mount a drive.
- Use UDP.
- Access TCP-based services that use dynamic ports, such as FTP Passive Mode or Extended Passive Mode.
- Support LDAP, because it can require UDP.
- Support Active Directory, because you cannot domain join an App Service worker.

Controlling App Service traffic by using Azure Traffic Manager

You can use Azure Traffic Manager to control how requests from web clients are distributed to apps in Azure App Service. When App Service endpoints are added to an Azure Traffic Manager profile, Azure Traffic Manager keeps track of the status of your App Service apps (running, stopped, or deleted) so that it can decide which of those endpoints should receive traffic.

Routing methods

Azure Traffic Manager uses four different routing methods. These methods are described in the following list as they pertain to Azure App Service.

- **Priority:** use a primary app for all traffic, and provide backups in case the primary or the backup apps are unavailable.
- **Weighted:** distribute traffic across a set of apps, either evenly or according to weights, which you define.
- **Performance:** when you have apps in different geographic locations, use the “closest” app in terms of the lowest network latency.
- **Geographic:** direct users to specific apps based on which geographic location their DNS query originates from.

For more information, see [Traffic Manager routing methods](#)⁷.

App Service and Traffic Manager Profiles

To configure the control of App Service app traffic, you create a profile in Azure Traffic Manager that uses one of the three load balancing methods described previously, and then add the endpoints (in this case, App Service) for which you want to control traffic to the profile. Your app status (running, stopped, or deleted) is regularly communicated to the profile so that Azure Traffic Manager can direct traffic accordingly.

When using Azure Traffic Manager with Azure, keep in mind the following points:

- For app only deployments within the same region, App Service already provides failover and round-robin functionality without regard to app mode.
- For deployments in the same region that use App Service in conjunction with another Azure cloud service, you can combine both types of endpoints to enable hybrid scenarios.
- You can only specify one App Service endpoint per region in a profile. When you select an app as an endpoint for one region, the remaining apps in that region become unavailable for selection for that profile.
- The App Service endpoints that you specify in an Azure Traffic Manager profile appears under the **Domain Names** section on the Configure page for the app in the profile, but is not configurable there.
- After you add an app to a profile, the **Site URL** on the Dashboard of the app's portal page displays the custom domain URL of the app if you have set one up. Otherwise, it displays the Traffic Manager profile URL (for example, `contoso.trafficmanager.net`). Both the direct domain name of the app and the Traffic Manager URL are visible on the app's Configure page under the **Domain Names** section.
- Your custom domain names work as expected, but in addition to adding them to your apps, you must also configure your DNS map to point to the Traffic Manager URL.
- You can only add apps that are in standard or premium mode to an Azure Traffic Manager profile.

Azure App Service Local Cache overview

Note: Local cache is not supported in Function apps or containerized App Service apps, such as on App Service on Linux.

Azure web app content is stored on Azure Storage and is surfaced up in a durable manner as a content share. This design is intended to work with a variety of apps and has the following attributes:

- The content is shared across multiple virtual machine (VM) instances of the web app.
- The content is durable and can be modified by running web apps.
- Log files and diagnostic data files are available under the same shared content folder.
- Publishing new content directly updates the content folder. You can immediately view the same content through the SCM website and the running web app (typically some technologies such as ASP.NET do initiate a web app restart on some file changes to get the latest content).

⁷ <https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-routing-methods>

While many web apps use one or all of these features, some web apps just need a high-performance, read-only content store that they can run from with high availability. These apps can benefit from a VM instance of a specific local cache.

The Azure App Service Local Cache feature provides a web role view of your content. This content is a write-but-discard cache of your storage content that is created asynchronously on-site startup. When the cache is ready, the site is switched to run against the cached content. Web apps that run on Local Cache have the following benefits:

- They are immune to latencies that occur when they access content on Azure Storage.
- They are immune to the planned upgrades or unplanned downtimes and any other disruptions with Azure Storage that occur on servers that serve the content share.
- They have fewer app restarts due to storage share changes.

How the local cache changes the behavior of App Service

- *D:\home* points to the local cache, which is created on the VM instance when the app starts up. *D:\local* continues to point to the temporary VM-specific storage.
- The local cache contains a one-time copy of the */site* and */siteextensions* folders of the shared content store, at *D:\home\site* and *D:\home\siteextensions*, respectively. The files are copied to the local cache when the app starts up. The size of the two folders for each app is limited to 300 MB by default, but you can increase it up to 2 GB.
- The local cache is read-write. However, any modification is discarded when the app moves virtual machines or gets restarted. Do not use the local cache for apps that store mission-critical data in the content store.
- *D:\home\LogFiles* and *D:\home\Data* contain log files and app data. The two subfolders are stored locally on the VM instance, and are copied to the shared content store periodically. Apps can persist log files and data by writing them to these folders. However, the copy to the shared content store is best-effort, so it is possible for log files and data to be lost due to a sudden crash of a VM instance.
- Log streaming is affected by the best-effort copy. You could observe up to a one-minute delay in the streamed logs.
- In the shared content store, there is a change in the folder structure of the LogFiles and Data folders for apps that use the local cache. There are now subfolders in them that follow the naming pattern of "unique identifier" + time stamp. Each of the subfolders corresponds to a VM instance where the app is running or has run.
- Other folders in *D:\home* remain in the local cache and are not copied to the shared content store.
- App deployment through any supported method publishes directly to the durable shared content store. To refresh the *D:\home\site* and *D:\home\siteextensions* folders in the local cache, the app needs to be restarted.
- The default content view of the SCM site continues to be that of the shared content store.

About App Service Environments

Overview

The Azure App Service Environment is an Azure App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale. This capability can host your:

- Windows web apps
- Linux web apps (in Preview)
- Docker containers (in Preview)
- Mobile apps
- Functions

App Service environments (ASEs) are appropriate for application workloads that require:

- Very high scale
- Isolation and secure network access
- High memory utilization

Customers can create multiple ASEs within a single Azure region or across multiple Azure regions. This flexibility makes ASEs ideal for horizontally scaling stateless application tiers in support of high RPS workloads.

ASEs are isolated to running only a single customer's applications and are always deployed into a virtual network. Customers have fine-grained control over inbound and outbound application network traffic. Applications can establish high-speed secure connections over VPNs to on-premises corporate resources.

- ASE comes with its own pricing tier, learn how the *Isolated offering* helps drive hyper-scale and security.
- *App Service Environments v2* provide a surrounding to safeguard your apps in a subnet of your network and provides your own private deployment of Azure App Service.
- Multiple ASEs can be used to scale horizontally.
- ASEs can be used to configure security architecture, as shown in the AzureCon Deep Dive. To see how the security architecture shown in the AzureCon Deep Dive was configured, see the ***article on how to implement a layered security architecture***⁸ with App Service environments.
- Apps running on ASEs can have their access gated by upstream devices, such as web application firewalls (WAFs).

Dedicated environment

An ASE is dedicated exclusively to a single subscription and can host 100 App Service Plan instances. The range can span 100 instances in a single App Service plan to 100 single-instance App Service plans, and everything in between.

⁸ <https://docs.microsoft.com/en-us/azure/app-service/environment/app-service-app-service-environment-layered-security>

An ASE is composed of front ends and workers. Front ends are responsible for HTTP/HTTPS termination and automatic load balancing of app requests within an ASE. Front ends are automatically added as the App Service plans in the ASE are scaled out.

Workers are roles that host customer apps. Workers are available in three fixed sizes:

- One vCPU/3.5 GB RAM
- Two vCPU/7 GB RAM
- Four vCPU/14 GB RAM

Customers do not need to manage front ends and workers. All infrastructure is automatically added as customers scale out their App Service plans. As App Service plans are created or scaled in an ASE, the required infrastructure is added or removed as appropriate.

There is a flat monthly rate for an ASE that pays for the infrastructure and doesn't change with the size of the ASE. In addition, there is a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU. For information on pricing for an ASE, see the **App Service pricing**⁹ page and review the available options for ASEs.

⁹ <http://azure.microsoft.com/pricing/details/app-service/>

Creating an Azure App Service Web App

Using shell commands to create an App Service Web App

Using scripts to deploy, configure, and manage Web Apps can make developing and testing Web Apps faster and more efficient. Below are some of the current options for a shell-based experience.

Azure Cloud Shell

Azure Cloud Shell is an interactive, browser-accessible shell for managing Azure resources. It provides the flexibility of choosing the shell experience that best suits the way you work. Linux users can opt for a Bash experience, while Windows users can opt for PowerShell.

Cloud Shell enables access to a browser-based command-line experience built with Azure management tasks in mind. Leverage Cloud Shell to work untethered from a local machine in a way only the cloud can provide.

Cloud Shell is managed by Microsoft so it comes with popular command-line tools and language support. Cloud Shell also securely authenticates automatically for instant access to your resources through the Azure CLI 2.0 or Azure PowerShell cmdlets.

View the full **list of tools installed in Cloud Shell**¹⁰.

Azure CLI

The Azure CLI 2.0 is Microsoft's cross-platform command line experience for managing Azure resources. You can use it in your browser with Azure Cloud Shell, or install it on macOS, Linux, or Windows and run it from the command line.

Azure CLI 2.0 is optimized for managing and administering Azure resources from the command line, and for building automation scripts that work against the Azure Resource Manager. Using the Azure CLI 2.0, you can create VMs within Azure as easily as typing the following command:

```
az vm create -n MyLinuxVM -g MyResourceGroup --image UbuntuLTS
```

Use the Cloud Shell to run the CLI in your browser, or install it on macOS, Linux, or Windows. Read the **Get Started**¹¹ article to begin using the CLI. For information about the latest release, see the **release notes**¹².

A detailed **reference**¹³ is also available that documents how to use each individual Azure CLI 2.0 command.

Azure PowerShell

Azure PowerShell provides a set of cmdlets that use the Azure Resource Manager model for managing your Azure resources. You can use it in your browser with Azure Cloud Shell, or you can install it on your local machine and use it in any PowerShell session.

¹⁰ <https://docs.microsoft.com/en-us/azure/cloud-shell/features#tools>

¹¹ <https://docs.microsoft.com/en-us/cli/azure/get-started-with-azure-cli?view=azure-cli-latest>

¹² <https://docs.microsoft.com/en-us/cli/azure/release-notes-azure-cli?view=azure-cli-latest>

¹³ <https://docs.microsoft.com/en-us/cli/azure/reference-index>

Use the Cloud Shell to run the Azure PowerShell in your browser, or **install**¹⁴ it on own computer. Then read the **Get Started**¹⁵ article to begin using it. For information about the latest release, see the **release notes**¹⁶.

Creating a Web App with Azure CLI

Creating an Azure Services Web App with Azure CLI follows a consistent pattern:

1. Create the resource group
2. Create the App Service Plan
3. Create the web app
4. Deploy the app

The command for the last step will change depending on the whether you are deploying with FTP, GitHub, or some other source. There may be additional commands that are needed if you need to, for example, set the credentials for a local Git repository.

Command	Notes
<code>az group create</code>	Creates a resource group in which all resources are stored.
<code>az appservice plan create</code>	Creates an App Service plan.
<code>az webapp create</code>	Creates an Azure web app.
<code>az webapp deployment source config</code>	Get the details for available web app deployment profiles.

Sample script

Below is a sample script for creating a web app with a source deployment from GitHub. For more samples please see the **Azure CLI Samples**¹⁷ page.

```
#!/bin/bash

# Replace the following URL with a public GitHub repo URL
gitrepo=https://github.com/Azure-Samples/php-docs-hello-world
webappname=mywebapp$RANDOM

# Create a resource group.
az group create --location westeurope --name myResourceGroup

# Create an App Service plan in `FREE` tier.
az appservice plan create --name $webappname --resource-group myResource-
Group --sku FREE

# Create a web app.
az webapp create --name $webappname --resource-group myResourceGroup --plan
$webappname
```

¹⁴ <https://docs.microsoft.com/en-us/powershell/azure/install-azurermps-6.5.0>

¹⁵ <https://docs.microsoft.com/en-us/powershell/azure/get-started-azureps?view=azurermps-6.5.0>

¹⁶ <https://docs.microsoft.com/en-us/powershell/azure/release-notes-azureps?view=azurermps-6.5.0>

¹⁷ <https://docs.microsoft.com/en-us/azure/app-service/app-service-cli-samples>

```
# Deploy code from a public GitHub repository.
az webapp deployment source config --name $webappname --resource-group
myResourceGroup \
--repo-url $gitrepo --branch master --manual-integration

# Copy the result of the following command into a browser to see the web
app.
echo http://$webappname.azurewebsites.net
```

Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

Creating a Web App with Azure PowerShell

Creating an Azure Services Web App with Azure PowerShell follows a consistent pattern:

1. Create the resource group
2. Create the App Service Plan
3. Create the web app
4. Deploy the app

The command for the last step will change depending on the whether you are deploying with FTP, GitHub, or some other source. There may be additional commands that are needed if you need to, for example, set the credentials for a local Git repository.

Command	Notes
New-AzureRmResourceGroup	Creates a resource group in which all resources are stored.
New-AzureRmAppServicePlan	Creates an App Service plan.
New-AzureRmWebApp	Creates an Azure web app.
Set-AzureRmResource	Modifies a resource in a resource group.

Sample script

Below is a sample script for creating a web app with a source deployment from GitHub. For more samples please see the **Azure PowerShell Samples**¹⁸ page.

```
# Replace the following URL with a public GitHub repo URL
$gitrepo="https://github.com/Azure-Samples/app-service-web-dotnet-get-
started.git"
$webappname="mywebapp$(Get-Random) "
$location="West Europe"

# Create a resource group.
```

¹⁸ <https://docs.microsoft.com/en-us/azure/app-service/app-service-powershell-samples>

```
New-AzureRmResourceGroup -Name myResourceGroup -Location $location

# Create an App Service plan in Free tier.
New-AzureRmAppServicePlan -Name $webappname -Location $location -Resource-
GroupName myResourceGroup -Tier Free

# Create a web app.
New-AzureRmWebApp -Name $webappname -Location $location -AppServicePlan
$webappname -ResourceGroupName myResourceGroup

# Configure GitHub deployment from your GitHub repo and deploy once.
$PropertiesObject = @{
    repoUrl = "$gitrepo";
    branch = "master";
    isManualIntegration = "true";
}
Set-AzureRmResource -PropertyObject $PropertiesObject -ResourceGroupName
myResourceGroup -ResourceType Microsoft.Web/sites/sourcecontrols -Resource-
Name $webappname/web -ApiVersion 2015-08-01 -Force
```

Clean up deployment

After the script sample has been run, the following command can be used to remove the resource group, web app, and all related resources.

```
Remove-AzureRmResourceGroup -Name myResourceGroup -Force
```

Create a Web App by using the Azure Portal

There are several ways you can create a web app. You can use the Azure portal, the Azure CLI, a script, or an IDE. Here, we are going to use the portal because it's a graphical experience, which makes it a great learning tool. The portal helps you discover available features, add additional resources, and customize existing resources.

How to create a web app

When it's time to host your own app, you visit the Azure portal and create a **Web App**. By creating a **Web App** in the Azure portal, you are actually creating a set of hosting resources in App Service, which you can use to host any web-based application that is supported by Azure, whether it be ASP.NET Core, Node.js, PHP, etc. The figure below shows how easy it is to configure the framework/language used by the app.

The Azure portal provides a template to create a web app. This template requires the following fields:

- **App name:** The name of the web app.
- **Subscription:** A valid and active subscription.
- **Resource group:** A valid resource group. The sections below explain in detail what a resource group is.
- **OS:** The operating system. The options are: Windows, Linux, and Docker containers. On Windows, you can host any type of application from a variety of technologies. The same applies to Linux hosting, though on Linux, any ASP.NET apps must be ASP.Net Core on the .NET Core framework. The final option is Docker containers, where you can deploy your containers directly over containers hosted and maintained by Azure.
- **App Service plan/location:** A valid Azure App Service plan. The sections below explain in detail what an App Service plan is.
- **Applications Insights:** You can turn on the Azure Application Insights option and benefit from the monitoring and metric tools that the Azure portal offers to help you keep an eye on the performance of your apps.

The Azure portal gives you the upper hand in managing, monitoring, and controlling your web app through the many available tools.

Deployment slots

Using the Azure portal, you can easily add **deployment slots** to an App Service web app. For instance, you can create a **staging** deployment slot where you can push your code to test on Azure. Once you are

happy with your code, you can easily **swap** the staging deployment slot with the production slot. You do all this with a few simple mouse clicks in the Azure portal.

Home > All resources > BestBike1234 - Deployment slots

BestBike1234 - Deployment slots
App Service

Search (Ctrl+ /)

Deployment

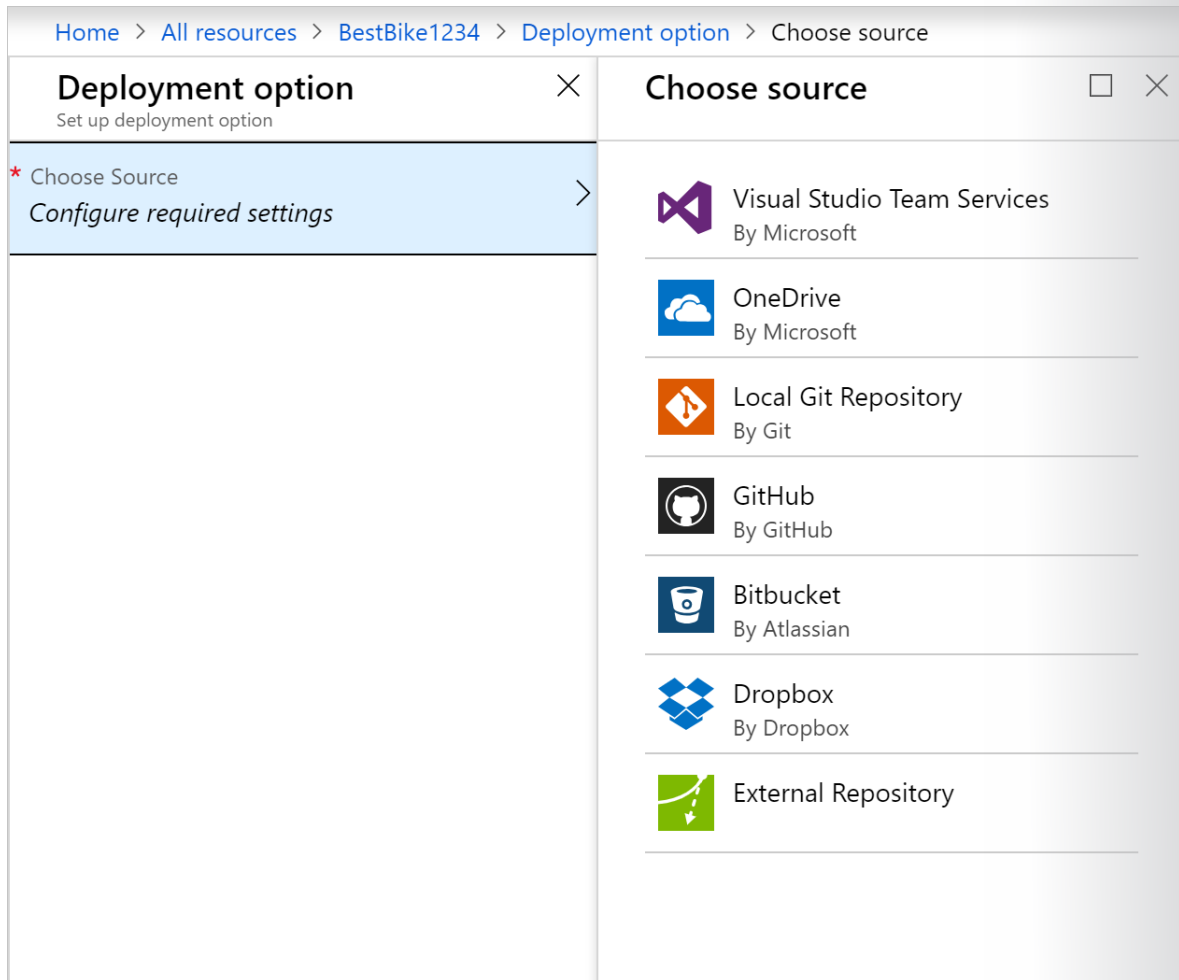
- Quickstart
- Deployment credentials
- Deployment slots**
- Deployment options
- Deployment Center (Preview)

+ Add Slot Swap

NAME	STATUS	APP SERVICE PLAN
bestbike1234-staging	Running	BestBike1234-app-service-plan

Continuous integration/deployment support

The Azure portal provides out-of-the-box continuous integration and deployment with Visual Studio Team Services, GitHub, Bitbucket, Dropbox, OneDrive, or a local Git repository on your development machine. You connect your web app with any of the above sources and App Service will do the rest for you by auto-syncing code and any future changes on the code into the web app. Furthermore, with Visual Studio Team Services, you can define your own build and release process that ends up compiling your source code, running the tests, building a release, and finally pushing the release into a web app every time you commit the code. All that happens implicitly without any need to intervene.



Integrated Visual Studio publishing and FTP publishing

In addition to being able to set up continuous integration/deployment for your web app, you can always benefit from the tight integration with Visual Studio to publish your web app to Azure via Web Deploy technology. Also, Azure supports FTP, although you are better off not using FTP for publishing because it lacks some capability in Web Deploy to pick and choose only those files that were changed or added, and not just publish everything to Azure!

Built-in auto scale support (automatic scale-out based on real-world load)

Baked into the web app is the ability to scale up/down or scale out. Depending on the usage of the web app, you can scale your app up/down by increasing/decreasing the resources of the underlying machine that is hosting your web app. Resources can be number of cores or the amount of RAM available.

Scaling out, on the other hand, is the ability to increase the number of machine instances that are running your web app.

What is a resource group?

A resource group is a method of grouping interdependent resources and services such as virtual machines, web apps, databases, and more for a given application and environment. Think of it as a folder, a place to group elements of your app.

Resource groups allow you to easily manage and delete resources. They also provide a way to monitor, control access, provision, and manage billing for collections of resources that are required to run an application or are used by a client.

Creating background tasks by using WebJobs in Azure App Service

Overview of WebJobs

WebJobs is a feature of Azure App Service that enables you to run a program or script in the same context as a web app, API app, or mobile app. There is no additional cost to use WebJobs.

The Azure WebJobs SDK can be used with WebJobs to simplify many programming tasks. Azure Functions provides another way to run programs and scripts. For a comparison between WebJobs and Functions, see **Choose between Flow, Logic Apps, Functions, and WebJobs**¹⁹.

WebJob Types

There are two types of WebJobs, *continuous* and *triggered*. The following table describes the differences.

Continuous	Triggered
Starts immediately when the WebJob is created. To keep the job from ending, the program or script typically does its work inside an endless loop. If the job does end, you can restart it.	Starts only when triggered manually or on a schedule.
Runs on all instances that the web app runs on. You can optionally restrict the WebJob to a single instance.	Runs on a single instance that Azure selects for load balancing.
Supports remote debugging.	Doesn't support remote debugging.

Note: A web app can time out after 20 minutes of inactivity. Only requests to the scm (deployment) site or to the web app's pages in the portal reset the timer. Requests to the actual site don't reset the timer. If your app runs continuous or scheduled WebJobs, enable **Always On** to ensure that the WebJobs run reliably. This feature is available only in the Basic, Standard, and Premium pricing tiers.

Supported file types for scripts or programs

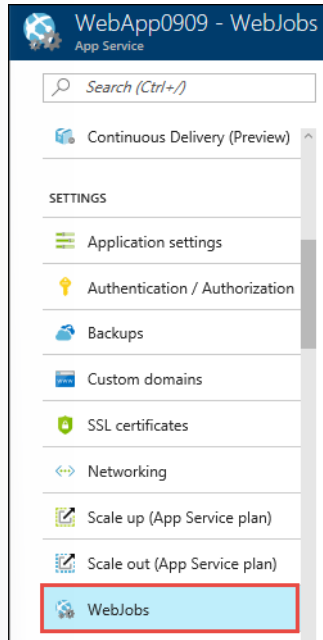
The following file types are supported:

- .cmd, .bat, .exe (using Windows cmd)
- .ps1 (using PowerShell)
- .sh (using Bash)
- .php (using PHP)
- .py (using Python)
- .js (using Node.js)
- .jar (using Java)

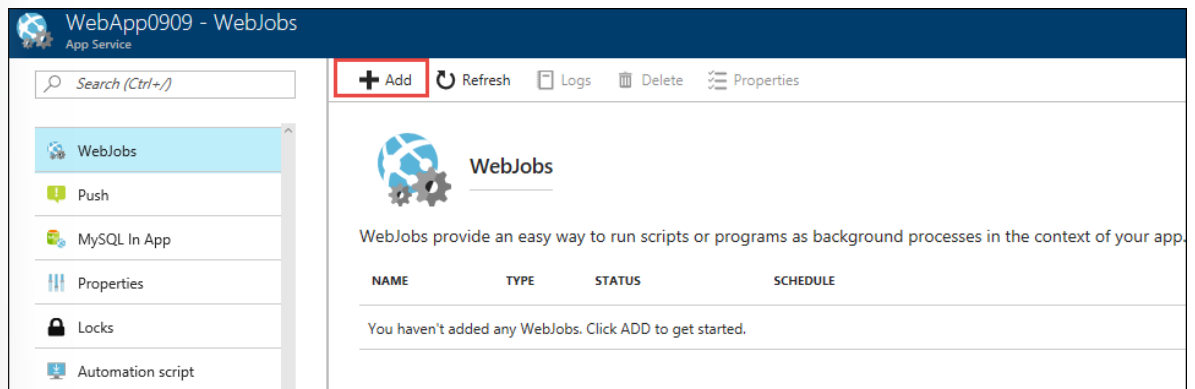
¹⁹ <https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs>

Creating a continuous WebJob

1. In the **Azure portal**²⁰, go to the **App Service** page of your App Service web app, API app, or mobile app.
2. Select **WebJobs**.



- 3.
4. In the **WebJobs** page, select **Add**.



- 5.
6. Use the **Add WebJob** settings as specified in the table

²⁰ <https://portal.azure.com/>

Add WebJob

webapp0909

*

Name ⓘ


myContinuousWebJob

✓

*

File Upload

ConsoleApp1.zip



Type ⓘ

Continuous

▼

Scale ⓘ

Multi Instance

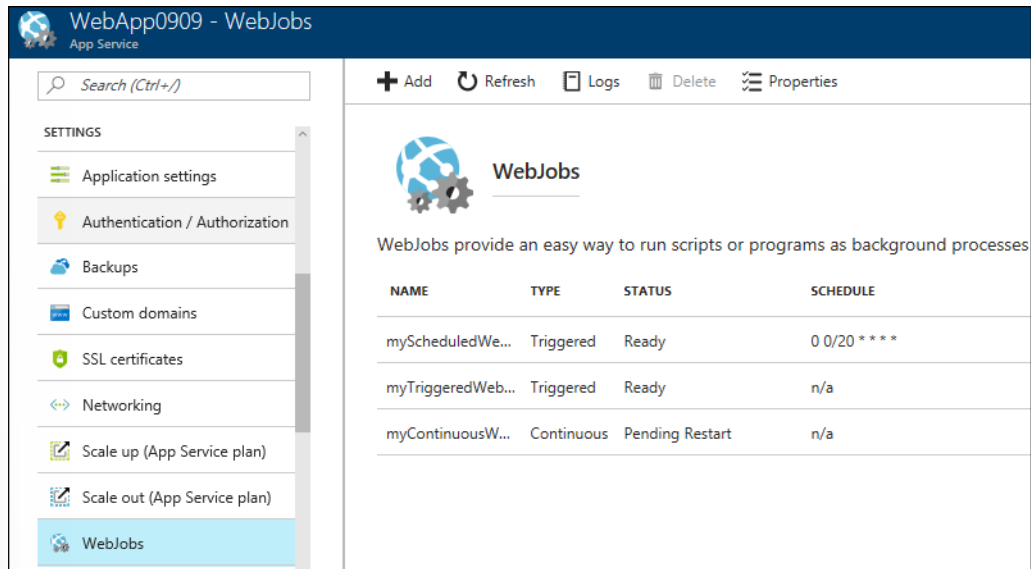
▼

OK

7.

Setting	Sample Value	Description
Name	myContinuousWebJob	A name that is unique within an App Service app. Must start with a letter or a number and cannot contain special characters other than "-" and "_" .
File Upload	ConsoleApp.zip	A .zip file that contains your executable or script file as well as any supporting files needed to run the program or script. The supported executable or script file types are listed in <i>Overview of Webjobs</i> section of this course.
Type	Continuous	Choose the type of of WebJob you want to create, this example uses <i>continuous</i>
Scale	Multi instance	Available only for Continuous WebJobs. Determines whether the program or script runs on all instances or just one instance. The option to run on multiple instances doesn't apply to the Free or Shared pricing tiers.

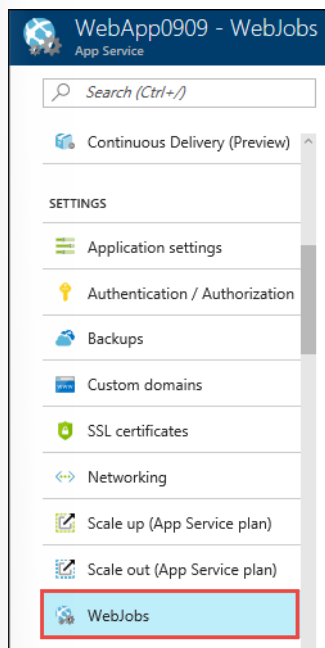
8. Click **OK**.
9. The new WebJob appears on the **WebJobs** page.



10.

Creating a manually triggered WebJob

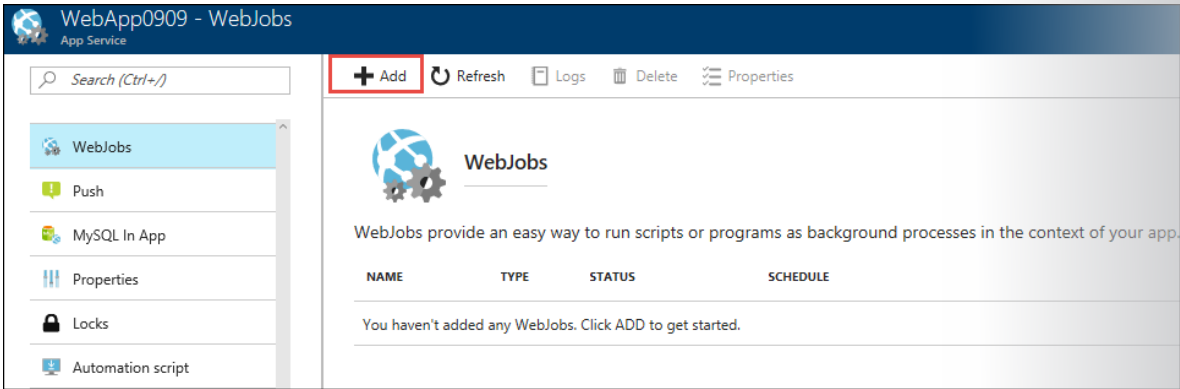
1. In the **Azure portal**²¹, go to the **App Service** page of your App Service web app, API app, or mobile app.
2. Select **WebJobs**.



3.

4. In the **WebJobs** page, select **Add**.

²¹ <https://portal.azure.com/>



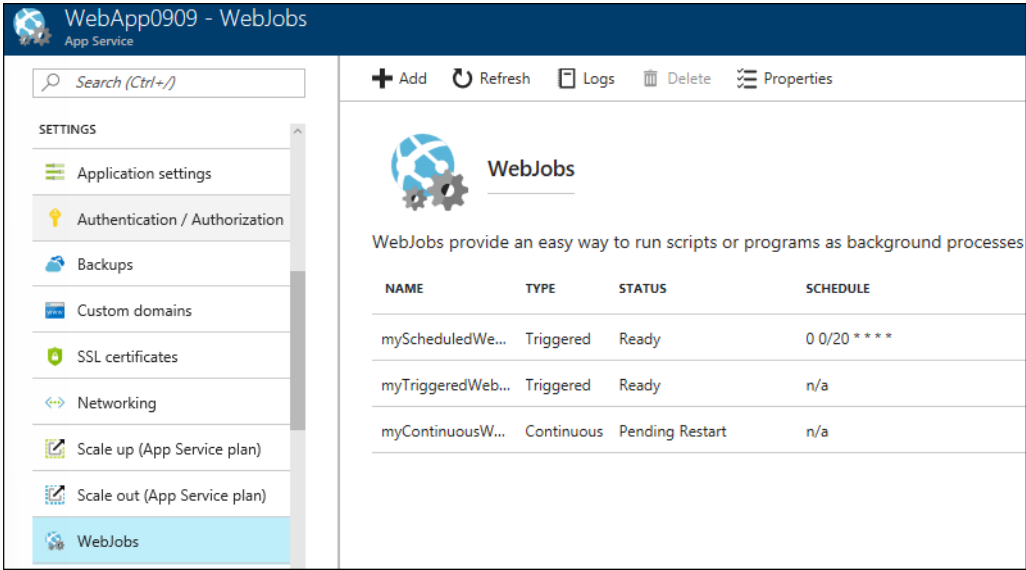
- 5.
6. Use the **Add WebJob** settings as specified in the table

- 7.

Setting	Sample Value	Description
Name	myContinuousWebJob	A name that is unique within an App Service app. Must start with a letter or a number and cannot contain special characters other than "-" and "_".
File Upload	ConsoleApp.zip	A .zip file that contains your executable or script file as well as any supporting files needed to run the program or script. The supported executable or script file types are listed in <i>Overview of Webjobs</i> section of this course.
Type	Triggered	Choose the type of of WebJob you want to create, this example uses <i>Triggered</i> .

Triggers	Manual	
----------	--------	--

- 8. Click **OK**.
- 9. The new WebJob appears on the **WebJobs** page.



10.

Review questions

Module 1 review questions

Azure App Service plans

In App Service, an app runs in an App Service plan. An App Service plan defines a set of compute resources for a web app to run. These compute resources are analogous to the server farm in conventional web hosting. One or more apps can be configured to run on the same computing resources (or in the same App Service plan).

Which App Service plan is available only to function apps?

> Click to see suggested answer

There are a few categories of pricing tiers:

- **Consumption:** This tier is only available to function apps. It scales the functions dynamically depending on workload.
- **Shared compute:** Free and Shared, the two base tiers, runs an app on the same Azure VM as other App Service apps, including apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources cannot scale out.
- **Dedicated compute:** The Basic, Standard, Premium, and PremiumV2 tiers run apps on dedicated Azure VMs. Only apps in the same App Service plan share the same compute resources. The higher the tier, the more VM instances are available to you for scale-out.
- **Isolated:** This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks, which provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities.

Managing web client requests

Web clients are sending requests to your app configured with multiple endpoints. What would you use to control how those requests are routed, and what are the available routing methods?

> Click to see suggested answer

You can use Azure Traffic Manager to control how requests from web clients are distributed to apps in Azure App Service. When App Service endpoints are added to an Azure Traffic Manager profile, Azure Traffic Manager keeps track of the status of your App Service apps (running, stopped, or deleted) so that it can decide which of those endpoints should receive traffic.

Routing methods

Azure Traffic Manager uses four different routing methods. These methods are described in the following list as they pertain to Azure App Service.

- **Priority:** use a primary app for all traffic, and provide backups in case the primary or the backup apps are unavailable.

- **Weighted:** distribute traffic across a set of apps, either evenly or according to weights, which you define.
- **Performance:** when you have apps in different geographic locations, use the “closest” app in terms of the lowest network latency.
- **Geographic:** direct users to specific apps based on which geographic location their DNS query originates from.

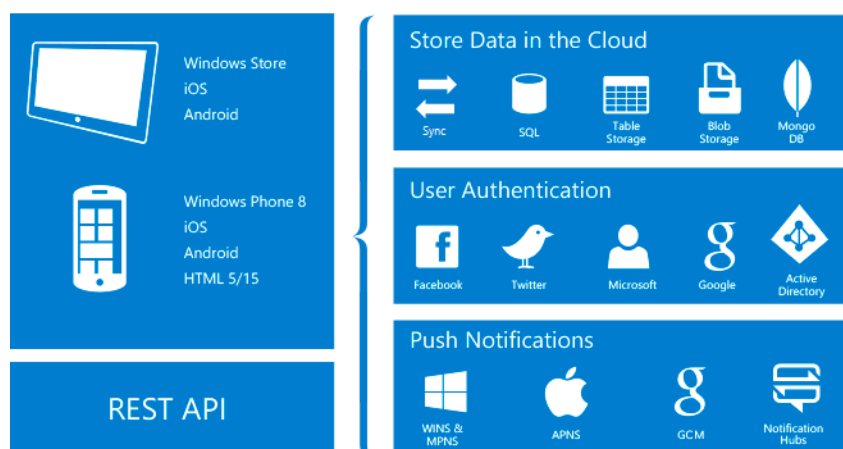
Create Azure App Service mobile apps

Getting Started with mobile apps in App Service

About mobile apps in App Service

Azure App Service is a fully managed platform as a service (PaaS) offering for professional developers. The service brings a rich set of capabilities to web, mobile, and integration scenarios.

The Mobile Apps feature of Azure App Service gives enterprise developers and system integrators a mobile-application development platform that's highly scalable and globally available.



Why Mobile Apps?

With the Mobile Apps feature, you can:

- **Build native and cross-platform apps:** Whether you're building native iOS, Android, and Windows apps or cross-platform Xamarin or Cordova (PhoneGap) apps, you can take advantage of App Service by using native SDKs.

- **Connect to your enterprise systems:** With the Mobile Apps feature, you can add corporate sign-in in minutes, and connect to your enterprise on-premises or cloud resources.
- **Build offline-ready apps with data sync:** Make your mobile workforce more productive by building apps that work offline, and use Mobile Apps to sync data in the background when connectivity is present with any of your enterprise data sources or software as a service (SaaS) APIs.
- **Push notifications to millions in seconds:** Engage your customers with instant push notifications on any device, personalized to their needs, and sent when the time is right.

Mobile Apps Features

The following features are important to cloud-enabled mobile development:

- **Authentication and authorization:** Support for identity providers, including Azure Active Directory for enterprise authentication, plus social providers such as Facebook, Google, Twitter, and Microsoft accounts. Mobile Apps offers an OAuth 2.0 service for each provider. You can also integrate the SDK for the identity provider for provider-specific functionality.
- **Data access:** Mobile Apps provides a mobile-friendly OData v3 data source that's linked to Azure SQL Database or an on-premises SQL server. Because this service can be based on Entity Framework, you can easily integrate with other NoSQL and SQL data providers, including Azure Table storage, MongoDB, Azure Cosmos DB, and SaaS API providers such as Office 365 and Salesforce.com.
- **Offline sync:** The client SDKs make it easy to build robust and responsive mobile applications that operate with an offline dataset. You can sync this dataset automatically with the back-end data, including conflict-resolution support.
- **Push notifications:** The client SDKs integrate seamlessly with the registration capabilities of Azure Notification Hubs, so you can send push notifications to millions of users simultaneously.
- **Client SDKs:** There is a complete set of client SDKs that cover native development (iOS, Android, and Windows), cross-platform development (Xamarin.iOS and Xamarin.Android, Xamarin.Forms), and hybrid application development (Apache Cordova). Each client SDK is available with an MIT license and is open-source.

Using the Mobile App backend service in an app

This tutorial shows you how to add a cloud-based backend service to a Universal Windows Platform (UWP) app by using an Azure mobile app backend. You will create both a new mobile app backend and a simple Todo list app that stores app data in Azure. The steps for creating a mobile app for any platform follow a similar pattern:

1. Create a new Azure mobile app backend
2. Configure the server project and connect to a database
3. Deploy and run the app

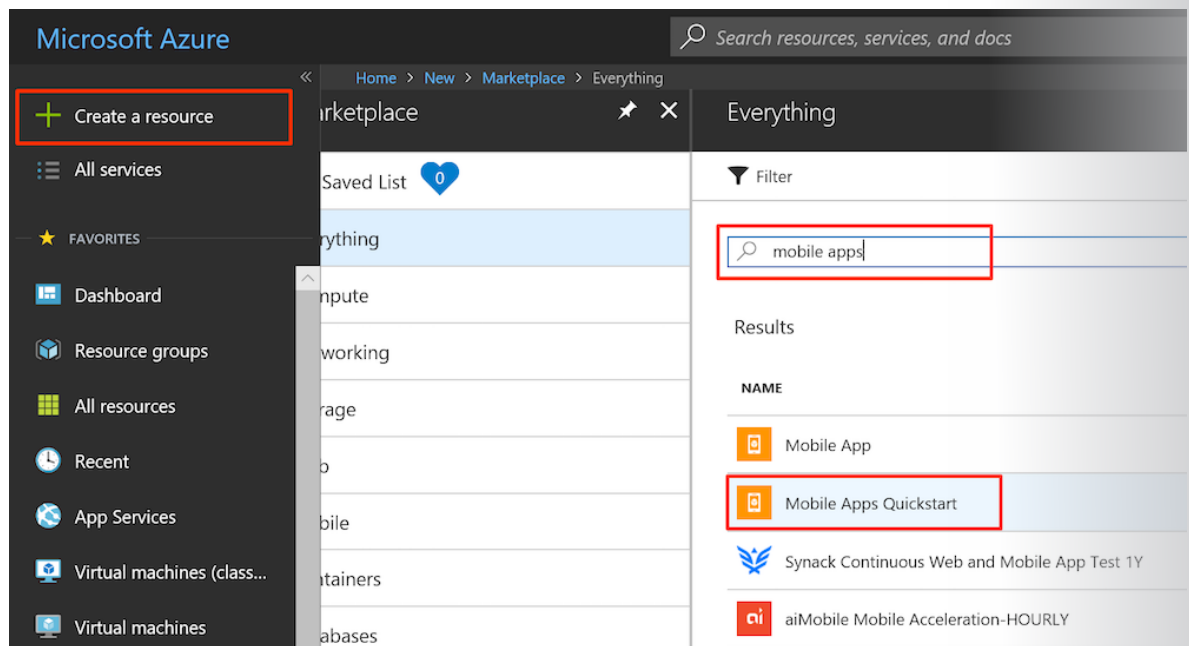
Prerequisites

The following step-by-step instructions show you how to use the Mobile Apps Quickstart feature to create a mobile app backend for the Windows platform. If you want to try this on your own, you will need the following:

- An active Azure account. If you don't have an account, you can sign up for an Azure trial and get up to 10 free mobile apps that you can keep using even after your trial ends. For details, see **Azure Free Trial**¹.
- **Visual Studio Community 2015**² or a later version.

Create a new Azure mobile backend

1. Sign in to the **Azure portal**³.
2. Click **Create a resource**.
3. In the search box, type **Mobile Apps**.



4. In the results list, select **Mobile Apps Quickstart**, and then select **Create**.
5. Choose a unique **App name**. This will also be part of the domain name for your App Service.
6. Under **Resource Group**, select an existing resource group or create a new one (using the same name as your app).
7. Click **Create**. Wait a few minutes for the service to be deployed successfully before proceeding. Watch the Notifications (bell) icon in the portal header for status updates.

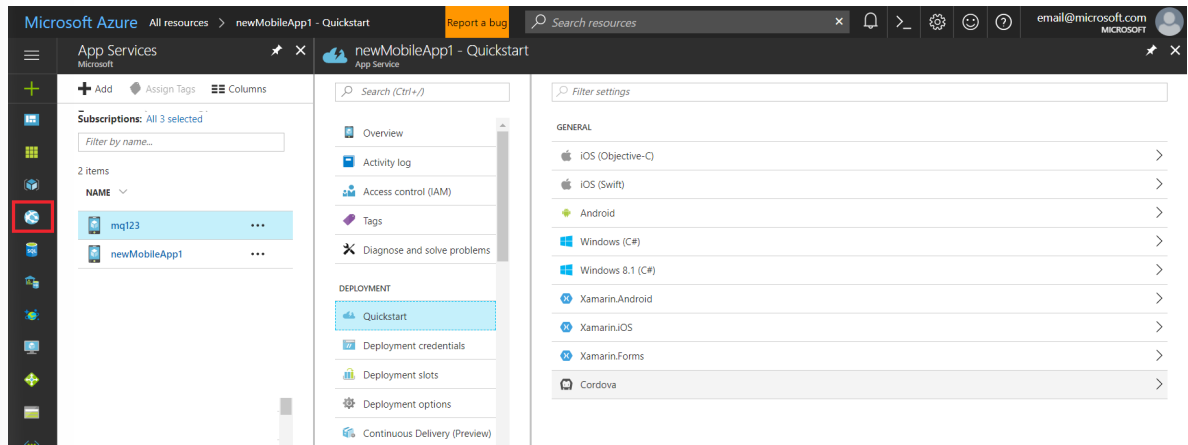
¹ <https://azure.microsoft.com/pricing/free-trial/>

² <https://go.microsoft.com/fwlink/p/?LinkId=534203>

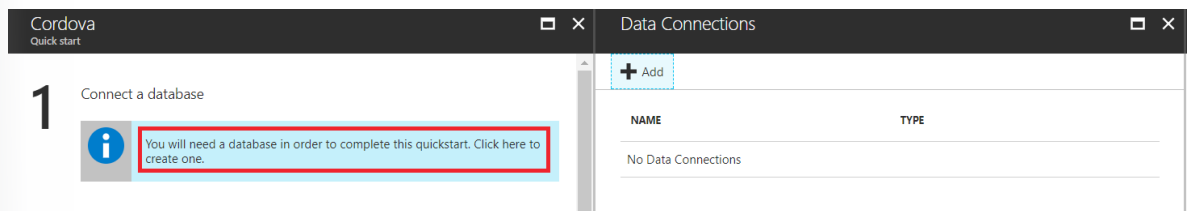
³ <https://portal.azure.com/>

Configure the server project

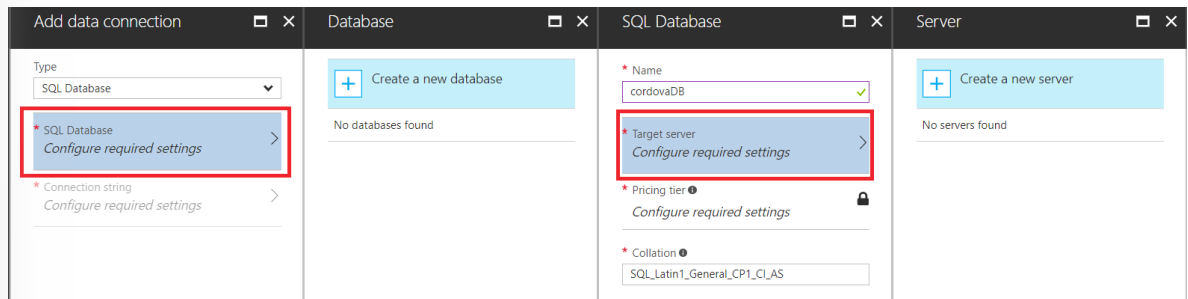
1. Click the **App Services** button, select your Mobile Apps back end, select **Quickstart**, and then select your client platform (iOS, Android, Xamarin, Cordova, Windows (C#)).



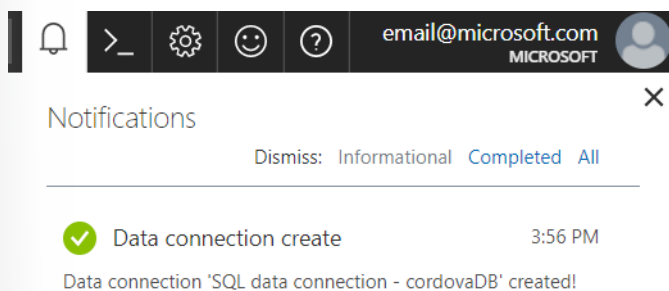
- 2.
3. If a database connection is not configured, create one by doing the following:



- 4.
5. a. Create a new SQL database and server.



- 6.
7. b. Wait until the data connection is successfully created.



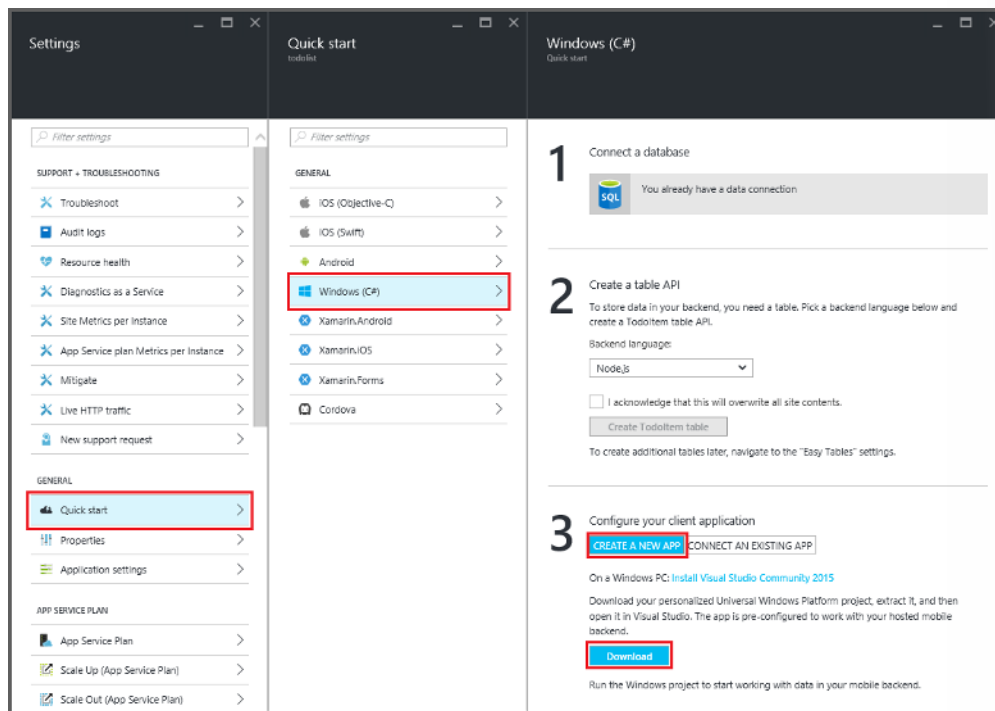
- 8.
9. Under 2. Create a table API, select Node.js for Backend language.

10. Accept the acknowledgment, and then select **Create TodoItem table**. This action creates a new to-do item table in your database.
11. Note: Switching an existing back end to Node.js overwrites all contents. To create a .NET back end instead, see **Work with the .NET back-end server SDK for Mobile Apps**⁴.

Download and run the client project

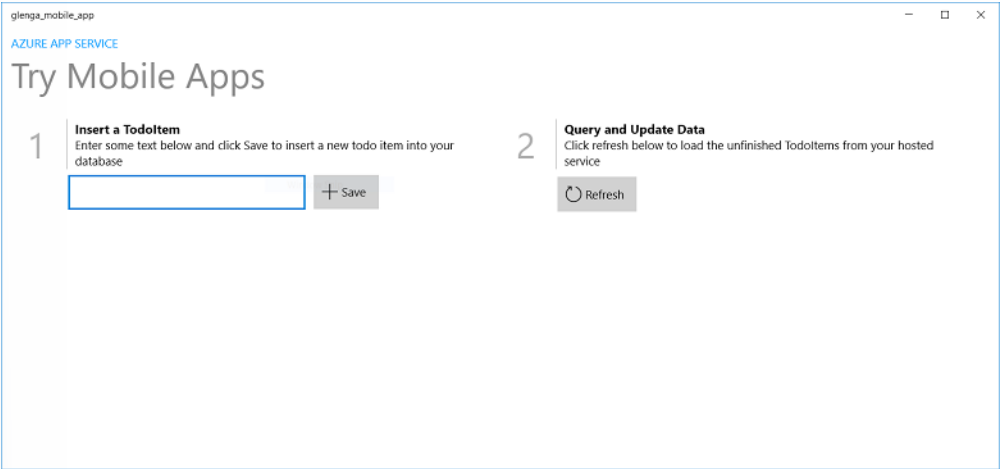
Once you have configured your Mobile App backend, you can either create a new client app or modify an existing app to connect to Azure. In this section, you download a UWP app template project that is customized to connect to your Mobile App backend.

1. Back in the **Quick start** blade for your Mobile App backend, click **Create a new app > Download**, then extract the compressed project files to your local computer.



- 2.
3. (Optional) Add the UWP app project to the same solution as the server project. This makes it easier to debug and test both the app and the backend in the same Visual Studio solution, if you choose to do so. To add a UWP app project to the solution, you must be using Visual Studio 2015 or a later version.
4. With the UWP app as the startup project, press the F5 key to deploy and run the app.
5. In the app, type meaningful text, such as *Complete the tutorial*, in the **Insert a TodoItem** text box, and then click **Save**.

⁴ <https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-dotnet-backend-how-to-use-server-sdk#create-app>



6.

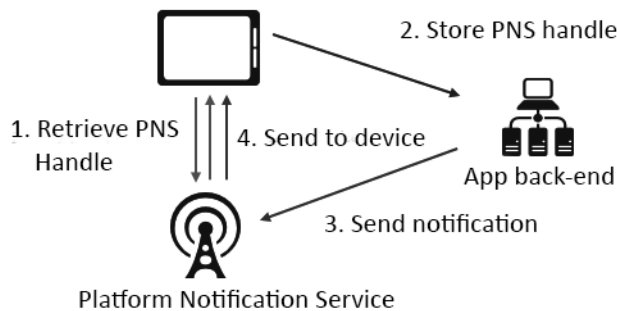
Enabling push notifications for your app

Overview of push notifications

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

At a high level, here is how push works:

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



5.

Adding push notifications to our sample project

This section of the course shows you how to enable push notifications in your app. This builds on the instructions in the previous section covering how to create the mobile backend. The steps for enabling push notifications follow a similar pattern for whichever platform you are targeting.

1. Configure a Notification Hub
2. Register your app for push notifications
3. Configure the back end to send push notifications
4. Update the server to send push notifications
5. Add push notifications to your app

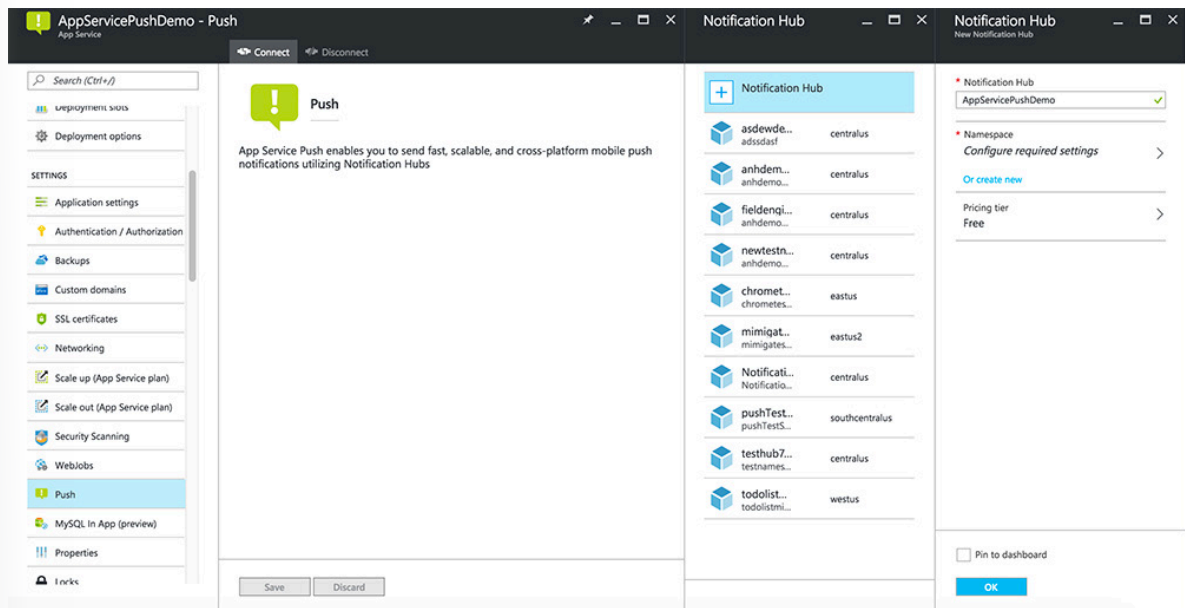
If you do not use the downloaded quick start server project, you will need the push notification extension package. See **Work with the .NET backend server SDK for Azure Mobile Apps**⁵ for more information.

⁵ <https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-dotnet-backend-how-to-use-server-sdk>

Configure a Notification Hub

The Mobile Apps feature of Azure App Service uses Azure Notification Hubs to send pushes, so you will be configuring a notification hub for your mobile app.

1. In the Azure portal, go to **App Services**, and then select your app back end. Under **Settings**, select **Push**.
2. To add a notification hub resource to the app, select **Connect**. You can either create a hub or connect to an existing one.



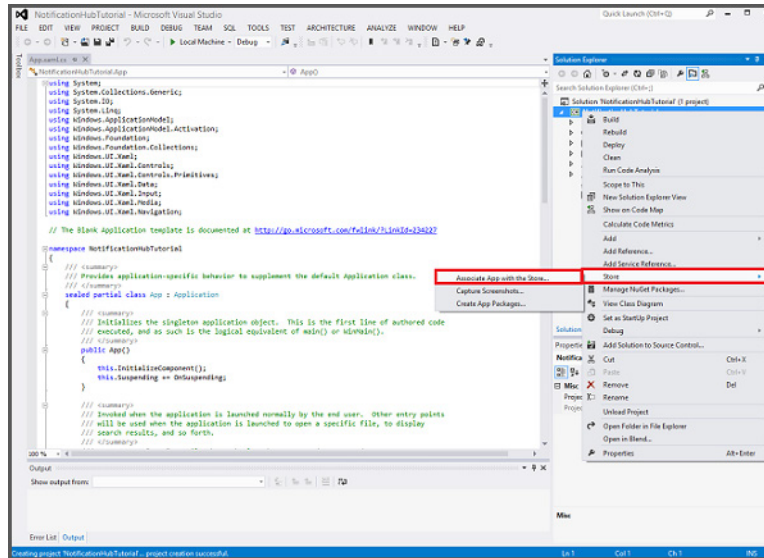
3.

Now you have connected a notification hub to your Mobile Apps back-end project. Later you configure this notification hub to connect to a platform notification system (PNS) to push to devices.

Register your app for push notifications

You need to submit your app to the Microsoft Store, then configure your server project to integrate with Windows Notification Services (WNS) to send push.

1. In Visual Studio Solution Explorer, right-click the UWP app project, click **Store > Associate App with the Store....**



- 2.
3. In the wizard, click **Next**, sign in with your Microsoft account, type a name for your app in **Reserve a new app name**, then click **Reserve**.
4. After the app registration is successfully created, select the new app name, click **Next**, and then click **Associate**. This adds the required Microsoft Store registration information to the application manifest.
5. Navigate to the **Windows Dev Center**⁶, sign-in with your Microsoft account, click the new app registration in **My apps**, then expand **Services > Push notifications**.
6. In the **Push notifications** page, click **Live Services site** under **Microsoft Azure Mobile Services**.
7. In the registration page, make a note of the value under **Application secrets** and the **Package SID**, which you will next use to configure your mobile app backend.

⁶ <https://dev.windows.com/en-us/overview>

Application Id
000000004418C017

Application Secrets [Learn More](#)

[Generate New Password](#)

Secret	Version	Status
6cpdsfg23hedfdd2234ggfOw4	Version 0	Current

Platforms

[Add Platform](#)

Web [Delete](#)

☒ Allow Implicit Flow
☒ Restrict token issuing to this app
Limits the issuing of JSON Web Tokens (JWT) for your domain to exclusively this application.

Target Domain
This is the domain that other apps will use when they request a JWT for your app on Windows (such as www.contoso.com).

Redirect URIs [Add Uri](#)

[Click here for help integrating your application with Microsoft.](#)

Windows Store

Package SID
This is the unique identifier for your Windows Store app.
ms-app://s-1-15-2-3896011764-999207426-1841428445-2442315578-2322321175-1779425796-3197542241

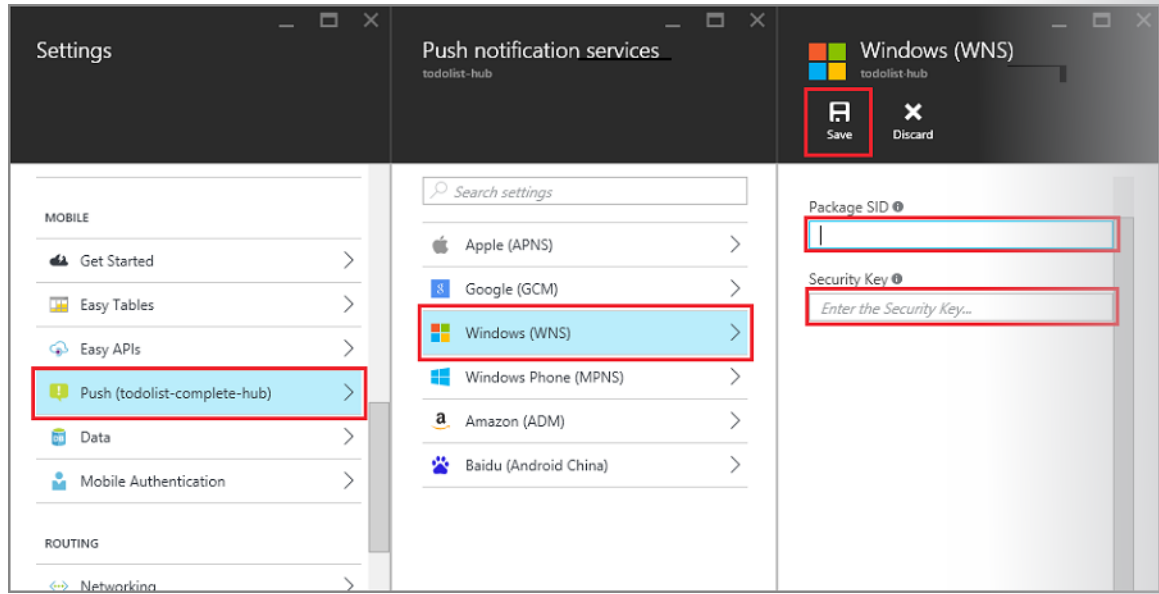
Application Identity
To set your application's identity values manually, open the AppManifest.xml file in a text editor and set these attributes of the <identity> element using the values shown here.

8.

Important: The client secret and package SID are important security credentials. Do not share these values with anyone or distribute them with your app. The Application Id is used with the secret to configure Microsoft Account authentication.

Configure the backend to send push notifications

1. In the Azure portal, select **Browse All > App Services**. Then select your Mobile Apps back end. Under **Settings**, select **App Service Push**. Then select your notification hub name.
2. Go to **Windows (WNS)**. Then enter the **Security key** (client secret) and **Package SID** that you obtained from the Live Services site. Next, select **Save**.



3.

Your back end is now configured to use WNS to send push notifications.

Update the server to send push notifications

Use the procedure below that matches your backend project type—either .NET backend or Node.js backend.

.NET backend project

1. In Visual Studio, right-click the server project and click **Manage NuGet Packages**, search for Microsoft.Azure.NotificationHubs, then click **Install**. This installs the Notification Hubs client library.
2. Expand **Controllers**, open `TodoltemController.cs`, and add the following using statements:

```
using System.Collections.Generic;
using Microsoft.Azure.NotificationHubs;
using Microsoft.Azure.Mobile.Server.Config;
```

3. In the **PostTodoltem** method, add the following code after the call to **InsertAsync**:

```
// Get the settings for the server project.
HttpConfiguration config = this.Configuration;
MobileAppSettingsDictionary settings =
    this.Configuration.GetMobileAppSettingsProvider().GetMobileAppSettings();

// Get the Notification Hubs credentials for the Mobile App.
string notificationHubName = settings.NotificationHubName;
string notificationHubConnection = settings
    .Connections[MobileAppSettingsKeys.NotificationHubConnectionString].
ConnectionString;

// Create the notification hub client.
```

```

NotificationHubClient hub = NotificationHubClient
    .CreateClientFromConnectionString(notificationHubConnection, notificationHubName);

// Define a WNS payload
var windowsToastPayload = @"<toast><visual><binding template=""ToastText01""><text id=""1"">"
    + item.Text + @"</text></binding></visual></toast>";
try
{
    // Send the push notification.
    var result = await hub.SendWindowsNativeNotificationAsync(windowsToastPayload);

    // Write the success result to the logs.
    config.Services.GetTraceWriter().Info(result.State.ToString());
}
catch (System.Exception ex)
{
    // Write the failure result to the logs.
    config.Services.GetTraceWriter().Error(ex.Message, null, "Push.SendAsync Error");
}

```

4. This code tells the notification hub to send a push notification after a new item is insertion.
5. Republish the server project.

Node.js backend project

1. If you haven't already done so, **download the quickstart project⁷** or else use the **online editor in the Azure portal⁸**.
2. Replace the existing code in the `todoitem.js` file with the following:

```

var azureMobileApps = require('azure-mobile-apps'),
    promises = require('azure-mobile-apps/src/utilities/promises'),
    logger = require('azure-mobile-apps/src/logger');

var table = azureMobileApps.table();

table.insert(function (context) {
    // For more information about the Notification Hubs JavaScript SDK,
    // see http://aka.ms/nodejshubs
    logger.info('Running TodoItem.insert');

    // Define the WNS payload that contains the new item Text.
    var payload = "<toast><visual><binding template=\ToastText01\><text

```

⁷ <https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-node-backend-how-to-use-server-sdk#download-quickstart>

⁸ <https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-node-backend-how-to-use-server-sdk#online-editor>

```

id=\"1\">\"
                                + context.item.text + "</text></binding></
visual></toast>";

// Execute the insert. The insert returns the results as a Promise,
// Do the push as a post-execute action within the promise flow.
return context.execute()
    .then(function (results) {
        // Only do the push if configured
        if (context.push) {
            // Send a WNS native toast notification.
            context.push.wns.sendToast(null, payload, function (error) {
                if (error) {
                    logger.error('Error while sending push notification: ',
error);
                } else {
                    logger.info('Push notification sent successfully!');
                }
            });
        }
        // Don't forget to return the results from the context.execute()
        return results;
    })
    .catch(function (error) {
        logger.error('Error while running context.execute: ', error);
    });
});

module.exports = table;

```

3. This sends a WNS toast notification that contains the `item.text` when a new todo item is inserted.
4. When editing the file on your local computer, republish the server project.

Add push notifications to your app

Next, your app must register for push notifications on start-up. When you have already enabled authentication, make sure that the user signs-in before trying to register for push notifications.

1. Open the **App.xaml.cs** project file and add the following `using` statements:

```

using System.Threading.Tasks;
using Windows.Networking.PushNotifications;

```

2. In the same file, add the following **InitNotificationsAsync** method definition to the **App** class:

```

private async Task InitNotificationsAsync()
{
    // Get a channel URI from WNS.
    var channel = await PushNotificationChannelManager
        .CreatePushNotificationChannelForApplicationAsync();

    // Register the channel URI with Notification Hubs.

```

```
        await App.MobileService.GetPush().RegisterAsync(channel.Uri);  
    }
```

3. This code retrieves the ChannelURI for the app from WNS, and then registers that ChannelURI with your App Service Mobile App.
4. At the top of the **OnLaunched** event handler in **App.xaml.cs**, add the **async** modifier to the method definition and add the following call to the new **InitNotificationsAsync** method, as in the following example:

```
protected async override void OnLaunched(LaunchActivatedEventArgs e)  
{  
    await InitNotificationsAsync();  
  
    // ...  
}
```

5. This guarantees that the short-lived ChannelURI is registered each time the application is launched.
6. Rebuild your UWP app project. Your app is now ready to receive toast notifications.

Enabling offline sync for your app

Overview of offline sync

Offline sync allows end users to interact with a mobile app - viewing, adding, or modifying data - even when there is no network connection. Changes are stored in a local database. Once the device is back online, these changes are synced with the remote backend.

What is a sync context?

A *sync context* is associated with a mobile client object (such as `IMobileServiceClient` or `MSClient`) and tracks changes that are made with sync tables. The sync context maintains an *operation queue*, which keeps an ordered list of CUD operations (Create, Update, Delete) that is later sent to the server.

A local store is associated with the sync context using an initialize method such as `IMobileServicesSyncContext.InitializeAsync(localstore)` in the .NET client SDK.

How offline synchronization works

When using sync tables, your client code controls when local changes are synchronized with an Azure Mobile App backend. Nothing is sent to the backend until there is a call to push local changes. Similarly, the local store is populated with new data only when there is a call to pull data.

- **Push:** Push is an operation on the sync context and sends all CUD changes since the last push. Note that it is not possible to send only an individual table's changes, because otherwise operations could be sent out of order. Push executes a series of REST calls to your Azure Mobile App backend, which in turn modifies your server database.
- **Pull:** Pull is performed on a per-table basis and can be customized with a query to retrieve only a subset of the server data. The Azure Mobile client SDKs then insert the resulting data into the local store.
- **Implicit Pushes:** If a pull is executed against a table that has pending local updates, the pull first executes a `push()` on the sync context. This push helps minimize conflicts between changes that are already queued and new data from the server.
- **Incremental Sync:** the first parameter to the pull operation is a query name that is used only on the client. If you use a non-null query name, the Azure Mobile SDK performs an incremental sync. Each time a pull operation returns a set of results, the latest `updatedAt` timestamp from that result set is stored in the SDK local system tables. Subsequent pull operations retrieve only records after that timestamp.
- To use incremental sync, your server must return meaningful `updatedAt` values and must also support sorting by this field. However, since the SDK adds its own sort on the `updatedAt` field, you cannot use a pull query that has its own `orderBy` clause.
- The query name can be any string you choose, but it must be unique for each logical query in your app. Otherwise, different pull operations could overwrite the same incremental sync timestamp and your queries can return incorrect results.
- If the query has a parameter, one way to create a unique query name is to incorporate the parameter value. For instance, if you are filtering on `userid`, your query name could be as follows (in C#):


```
await todoTable.PullAsync("todoItems" + userid,
    syncTable.Where(u => u.UserId == userid));
```

- If you want to opt out of incremental sync, pass null as the query ID. In this case, all records are retrieved on every call to PullAsync, which is potentially inefficient.
- **Purging:** You can clear the contents of the local store using `IMobileServiceSyncTable.PurgeAsync`. Purging may be necessary if you have stale data in the client database, or if you wish to discard all pending changes.
- A purge clears a table from the local store. If there are operations awaiting synchronization with the server database, the purge throws an exception unless the force purge parameter is set.
- As an example of stale data on the client, suppose in the "todo list" example, Device1 only pulls items that are not completed. A todoitem "Buy milk" is marked completed on the server by another device. However, Device1 still has the "Buy milk" todoitem in local store because it is only pulling items that are not marked complete. A purge clears this stale item.

Next

The rest of this lesson shows you how to add offline support to a Universal Windows Platform (UWP) app using an Azure Mobile App backend. The steps for enabling offline sync follow a similar pattern for whichever platform you are targeting.

1. Ensure the client app supports offline features
2. Ensure the app can disconnect from the backend
3. Ensure the app can reconnect to your Mobile App backend

Requirements

The next part of this course builds on a UWP app project. If you want to follow along in your own project be sure to have:

- Visual Studio 2013 running on Windows 8.1 or later.
- Complete the online tutorial **Create a Windows app**⁹. You'll be modifying this app to enable offline sync.
- **Azure Mobile Services SQLite Store**¹⁰
- **SQLite for Universal Windows Platform development**¹¹

Updating the client app to support offline features

Azure Mobile App offline features allow you to interact with a local database when you are in an offline scenario. To use these features in your app, you initialize a SyncContext to a local store. Then reference

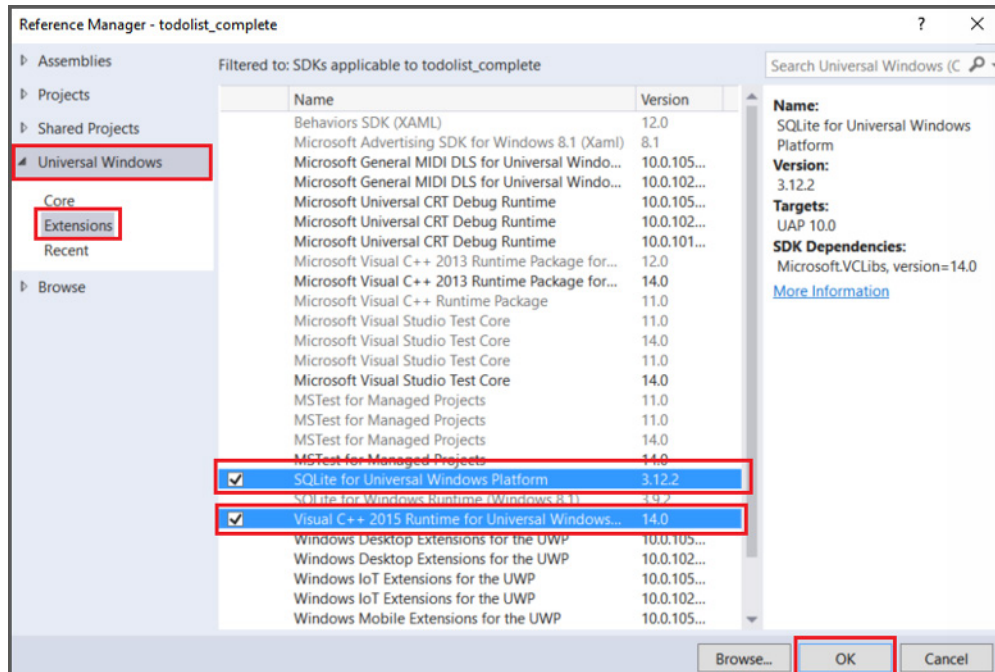
⁹ <https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-windows-store-dotnet-get-started>

¹⁰ <https://www.nuget.org/packages/Microsoft.Azure.Mobile.Client.SQLiteStore/>

¹¹ <https://marketplace.visualstudio.com/items?itemName=SQLiteDevelopmentTeam.SQLiteforUniversalWindowsPlatform>

your table through the `IMobileServiceSyncTable` interface. SQLite is used as the local store on the device.

1. Install the **SQLite runtime for the Universal Windows Platform**¹².
2. In Visual Studio, open the NuGet package manager for the UWP app project that you completed in the Create a Windows app tutorial. Search for and install the **Microsoft.Azure.Mobile.Client.SQLite-Store** NuGet package.
3. In Solution Explorer, right-click **References > Add Reference... > Universal Windows > Extensions**, then enable both **SQLite for Universal Windows Platform** and **Visual C++ 2015 Runtime for Universal Windows Platform** apps.



- 4.
5. Open the `MainPage.xaml.cs` file and uncomment the `#define OFFLINE_SYNC_ENABLED` definition.
6. In Visual Studio, press the F5 key to rebuild and run the client app. The app works the same as it did before you enabled offline sync. However, the local database is now populated with data that can be used in an offline scenario.

Update the app to disconnect from the backend

In this section, you break the connection to your Mobile App backend to simulate an offline situation. When you add data items, your exception handler tells you that the app is in offline mode. In this state, new items added in the local store and will be synced to the mobile app backend when push is next run in a connected state.

1. Edit `App.xaml.cs` in the shared project. Comment out the initialization of the **MobileServiceClient** and add the following line, which uses an invalid mobile app URL:

```
public static MobileServiceClient MobileService = new MobileServiceClient(
    "https://your-service.azurewebsites.fail");
```

¹² <http://sqlite.org/2016/sqlite-uwp-3120200.vsix>

1. You can also demonstrate offline behavior by disabling wifi and cellular networks on the device or use airplane mode.
2. Press **F5** to build and run the app. Notice your sync failed on refresh when the app launched.
3. Enter new items and notice that push fails with a `CancelledByNetworkError` status each time you click **Save**. However, the new todo items exist in the local store until they can be pushed to the mobile app backend. In a production app, if you suppress these exceptions the client app behaves as if it's still connected to the mobile app backend.
4. Close the app and restart it to verify that the new items you created are persisted to the local store.
5. (Optional) In Visual Studio, open **Server Explorer**. Navigate to your database in **Azure->SQL Databases**. Right-click your database and select **Open in SQL Server Object Explorer**. Now you can browse to your SQL database table and its contents. Verify that the data in the backend database has not changed.
6. (Optional) Use a REST tool such as Fiddler or Postman to query your mobile backend, using a GET query in the form `https://<your-mobile-app-backend-name>.azurewebsites.net/tables/ToDoItem`.

Update the app to reconnect to the backend

In this section, you reconnect the app to the mobile app backend. These changes simulate a network reconnection on the app.

When you first run the application, the `OnNavigatedTo` event handler calls `InitLocalStoreAsync`. This method in turn calls `SyncAsync` to sync your local store with the backend database. The app attempts to sync on startup.

1. Open `App.xaml.cs` in the shared project, and uncomment your previous initialization of `MobileServiceClient` to use the correct the mobile app URL.
2. Press the **F5** key to rebuild and run the app. The app syncs your local changes with the Azure Mobile App backend using push and pull operations when the `OnNavigatedTo` event handler executes.
3. (Optional) View the updated data using either SQL Server Object Explorer or a REST tool like Fiddler. Notice the data has been synchronized between the Azure Mobile App backend database and the local store.
4. In the app, click the check box beside a few items to complete them in the local store.

`UpdateCheckedToDoItem` calls `SyncAsync` to sync each completed item with the Mobile App backend. `SyncAsync` calls both push and pull. However, **whenever you execute a pull against a table that the client has made changes to, a push is always executed automatically**. This behavior ensures all tables in the local store along with relationships remain consistent. This behavior may result in an unexpected push.

API summary

To support the offline features of mobile services, we used the `IMobileServiceSyncTable` interface and initialized `MobileServiceClient.SyncContext` with a local SQLite database. When offline, the normal CRUD operations for Mobile Apps work as if the app is still connected while the operations occur against the local store. The following methods are used to synchronize the local store with the server:

- **PushAsync:** Because this method is a member of `IMobileServicesSyncContext`, changes across all tables are pushed to the backend. Only records with local changes are sent to the server.

- **PullAsync:** A pull is started from a `IMobileServiceSyncTable`. When there are tracked changes in the table, an implicit push is run to make sure that all tables in the local store along with relationships remain consistent. The *pushOtherTables* parameter controls whether other tables in the context are pushed in an implicit push. The *query* parameter takes an `IMobileServiceTableQuery` or OData query string to filter the returned data. The *queryId* parameter is used to define incremental sync.
- **PurgeAsync:** Your app should periodically call this method to purge stale data from the local store. Use the *force* parameter when you need to purge any changes that have not yet been synced.

Review questions

Module 2 review questions

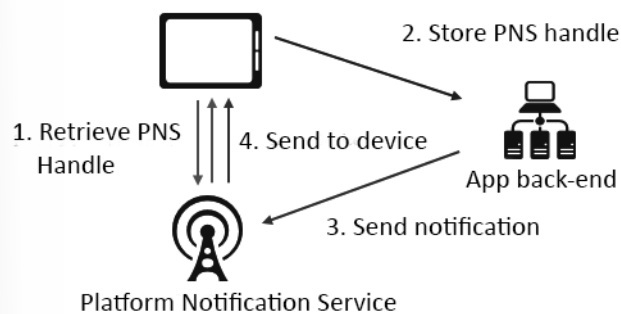
App Service push notifications

Push notifications are delivered through platform-specific infrastructures called Platform Notification Systems (PNSes). They offer barebone push functionalities to deliver a message to a device with a provided handle, and have no common interface. Can you describe how a push works?

> Click to see suggested answer

At a high level, here is how push works:

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



5.

Offline sync

What is the general pattern you need to follow to enable offline sync regardless of the app platform?

> Click to see suggested answer

The steps for enabling offline sync follow a similar pattern for whichever platform you are targeting.

1. Ensure the client app supports offline features
2. Ensure the app can disconnect from the backend
3. Ensure the app can reconnect to your Mobile App backend



Create Azure App Service API apps

Creating APIs

API Management overview

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

Overview

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

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

The system is made up of the following components:

- The **API gateway** is the endpoint that:
 - Accepts API calls and routes them to your backends.
 - Verifies API keys, JWT tokens, certificates, and other credentials.

- Enforces usage quotas and rate limits.
- Transforms your API on the fly without code modifications.
- Caches backend responses where set up.
- Logs call metadata for analytics purposes.
- The **Azure portal** is the administrative interface where you set up your API program. Use it to:
 - Define or import API schema.
 - Package APIs into products.
 - Set up policies like quotas or transformations on the APIs.
 - Get insights from analytics.
 - Manage users.
- The **Developer portal** serves as the main web presence for developers, where they can:
 - Read API documentation.
 - Try out an API via the interactive console.
 - Create an account and subscribe to get API keys.
 - Access analytics on their own usage.

APIs and operations

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

Products

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

Groups are used to manage the visibility of products to developers. Products grant visibility to groups, and developers can view and subscribe to the products that are visible to the groups in which they belong.

Groups

Groups are used to manage the visibility of products to developers. API Management has the following immutable system groups:

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

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

Developers

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

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

Policies

Policies are a powerful capability of API Management that allow the Azure portal to change the behavior of the API through configuration. Policies are a collection of statements that are executed sequentially on the request or response of an API. Popular statements include format conversion from XML to JSON and call rate limiting to restrict the number of incoming calls from a developer, and many other policies are available.

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

Developer portal

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

API Management terminology

- **Backend API** - An HTTP service that implements your API and its operations.
- **Frontend API/APIM API** - An APIM API does not host APIs, it creates facades for your APIs in order to customize the facade according to your needs without touching the back end API.
- **APIM product** - a product contains one or more APIs as well as a usage quota and the terms of use. You can include a number of APIs and offer them to developers through the Developer portal.
- **APIM API operation** - Each APIM API represents a set of operations available to developers. Each APIM API contains a reference to the back end service that implements the API, and its operations map to the operations implemented by the back end service.
- **Version** - Sometimes you want to publish new or different API features to some users, while others want to stick with the API that currently works for them.
- **Revision** - When your API is ready to go and starts to be used by developers, you usually need to take care in making changes to that API and at the same time not to disrupt callers of your API. It's also useful to let developers know about the changes you made.
- **Developer portal** - Your customers (developers) should use the Developer portal to access your APIs. The Developer portal can be customized.

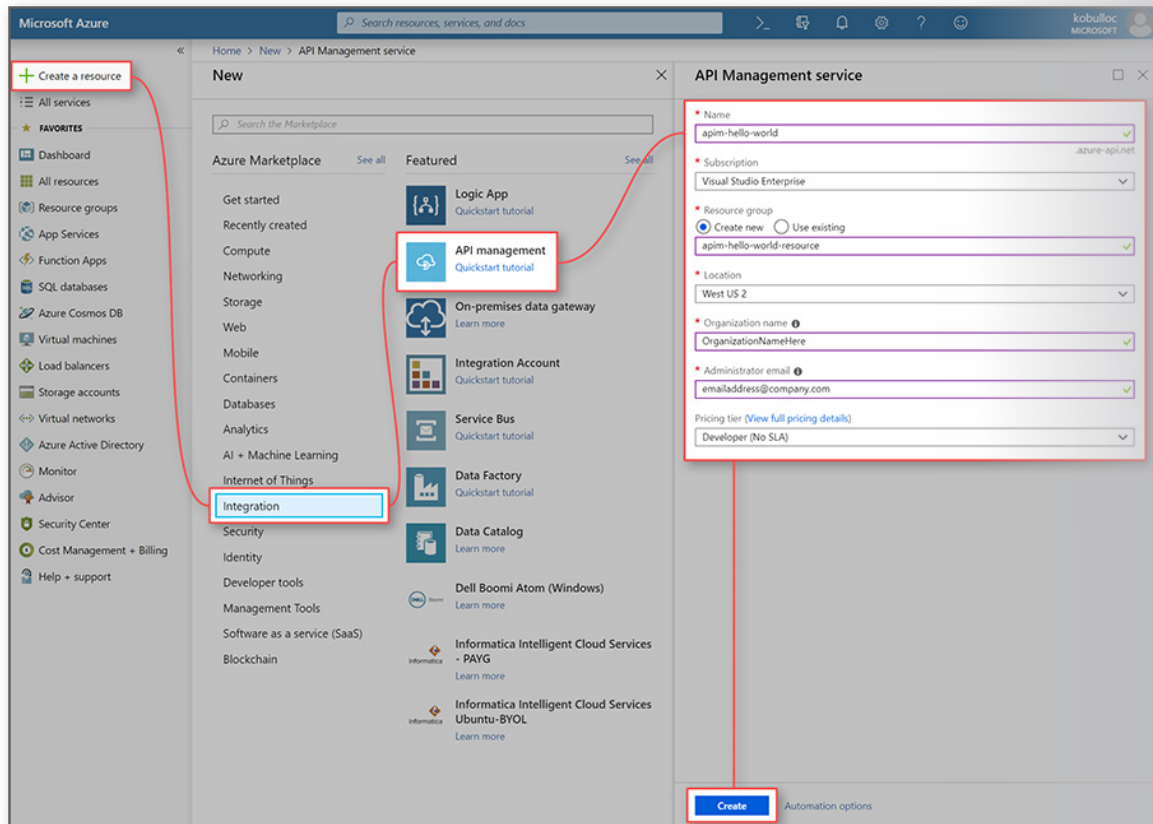
Create an Azure API Management service instance

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

Steps to create an APIM instance in the Azure Portal

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

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



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

API Management service

* Name

* Subscription
Visual Studio Ultimate with MSDN

* Resource group
☐ Create new ☒ Use existing

* Location
West US

* Organization name

* Administrator email

Pricing tier ([View full pricing details](#))
Developer (No SLA)

[Create](#) [Automation options](#)

5.

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

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

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

Clean up resources

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

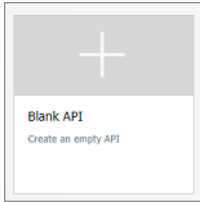
1. In the Azure portal, select **All services**.
2. Input `resource groups` in the search box and click on the result.
3. Find your resource group and click on it.
4. Click **Delete resource group**.
5. Confirm the deletion by inputting the name of your resource group.
6. Click **Delete**.

Create a new API

The steps in this tutorial show how to use the Azure portal to add an API manually to the API Management (APIM) instance you created earlier. A common scenario when you would want to create a blank API and define it manually is when you want to mock the API.

Create an API

1. Login to the Azure Portal and select the APIM instance you created earlier.
2. Select **APIs** from under **API MANAGEMENT**.
3. From the left menu, select **+ Add API**.
4. Select **Blank API** from the list.



- 5.
6. Enter settings for the API.

Name	Value	Description
Display name	"Blank API"	This name is displayed in the Developer portal.
Web Service URL (optional)	"http://httpbin.org"	If you want to mock an API, you might not enter anything. In this case, we enter http://httpbin.org. This is a public testing service.
URL scheme	"HTTPS"	<p>In this case, even though the back end has non-secure HTTP access, we specify a secure HTTPS APIM access to the back end.</p> <p>This kind of scenario (HTTPS to HTTP) is called HTTPS termination. You might do it if your API exists within a virtual network (where you know the access is secure even if HTTPS is not used).</p> <p>You might want to use "HTTPS termination" to save on some CPU cycles.</p>
URL suffix	"hbin"	The suffix is a name that identifies this specific API in this APIM instance. It has to be unique in this APIM instance.

Name	Value	Description
Products	"Unlimited"	<p>Publish the API by associating the API with a product. If you want for the API to be published and be available to developers, add it to a product. You can do it during API creation or set it later.</p> <p>Products are associations of one or more APIs. You can include a number of APIs and offer them to developers through the developer portal.</p> <p>Developers must first subscribe to a product to get access to the API. When they subscribe, they get a subscription key that is good for any API in that product. If you created the APIM instance, you are an administrator already, so you are subscribed to every product by default.</p> <p>By default, each API Management instance comes with two sample products: Starter and Unlimited.</p>

7. Select **Create**.

At this point, you have no operations in APIM that map to the operations in your back-end API. If you call an operation that is exposed through the back end but not through the APIM, you get a **404**.

Note: By default, when you add an API, even if it is connected to some back-end service, APIM will not expose any operations until you whitelist them. To whitelist an operation of your back-end service, create an APIM operation that maps to the back-end operation.

Add and test an operation

This section shows how to add a `/get` operation in order to map it to the back end `http://httpbin.org/get` operation.

Add an operation

1. Select the API you created in the previous step.
2. Click + **Add Operation**.
3. In the **URL**, select **GET** and enter `/get` in the resource.
4. Enter `FetchData` for **Display name**.
5. Select **Save**.

Test an operation

Test the operation in the Azure portal. Alternatively, you can test it in the **Developer portal**.

1. Select the **Test** tab.
2. Select **FetchData**.
3. Press **Send**.

The response that the `http://httpbin.org/get` operation generates appears.

Add and test a parameterized operation

This section shows how to add an operation that takes a parameter. In this case, we map the operation to `http://httpbin.org/status/200`.

Add the operation

1. Select the API you created in the previous step.
2. Click + **Add Operation**.
3. In the **URL**, select **GET** and enter `/status/{code}` in the resource. Optionally, you can provide some information associated with this parameter. For example, enter `Number` for **TYPE**, `200` (default) for **VALUES**.
4. Enter `GetStatus` for **Display name**.
5. Select **Save**.

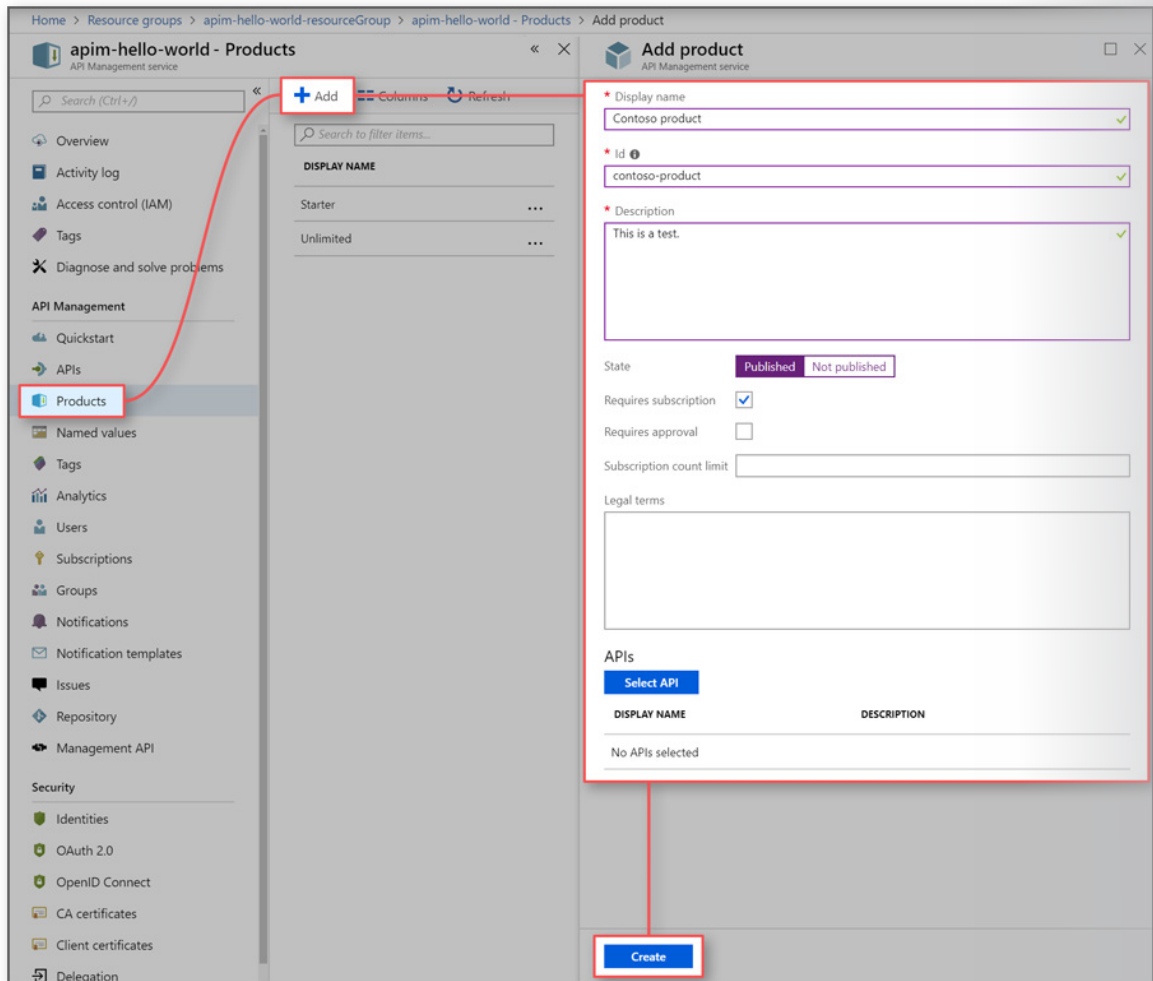
Test the operation

Test the operation in the Azure portal. Alternatively, you can test it in the Developer portal.

1. Select the **Test** tab.
2. Select **GetStatus**. By default the code value is set to `200`. You can change it to test other values. For example, type `418`.
3. Press **Send**. The response that the `"http://httpbin.org/status/200"` operation generates appears.

Create and publish a product

In Azure API Management, a product contains one or more APIs as well as a usage quota and the terms of use. Once a product is published, developers can subscribe to the product and begin to use the product's APIs.



1. Select **Products** in the menu on the left to display the **Products** page.
2. Select **+ Add**. When you add a product, you need to supply the following information:

Name	Description
Display name	The name as you want it to be shown in the Developer portal .
Name	A descriptive name of the product.
Description	The Description field allows you to provide detailed information about the product such as its purpose, the APIs it provides access to, and other useful information.
State	Press Published if you want to publish the product. Before the APIs in a product can be called, the product must be published. By default new products are unpublished, and are visible only to the Administrators group.
Requires subscription	Check Require subscription if a user is required to subscribe to use the product.

Name	Description
Requires approval	Check Require approval if you want an administrator to review and accept or reject subscription attempts to this product. If the box is unchecked, subscription attempts are auto-approved.
Subscription count limit	To limit the count of multiple simultaneous subscriptions, enter the subscription limit.
Legal terms	You can include the terms of use for the product which subscribers must accept in order to use the product.
APIs	Products are associations of one or more APIs. You can include a number of APIs and offer them to developers through the developer portal. You can add an existing API during the product creation. You can add an API to the product later, either from the Products Settings page or while creating an API.

3. Select **Create** to create the new product.

Add more configurations

You can continue configuring the product after saving it by choosing the **Settings** tab.

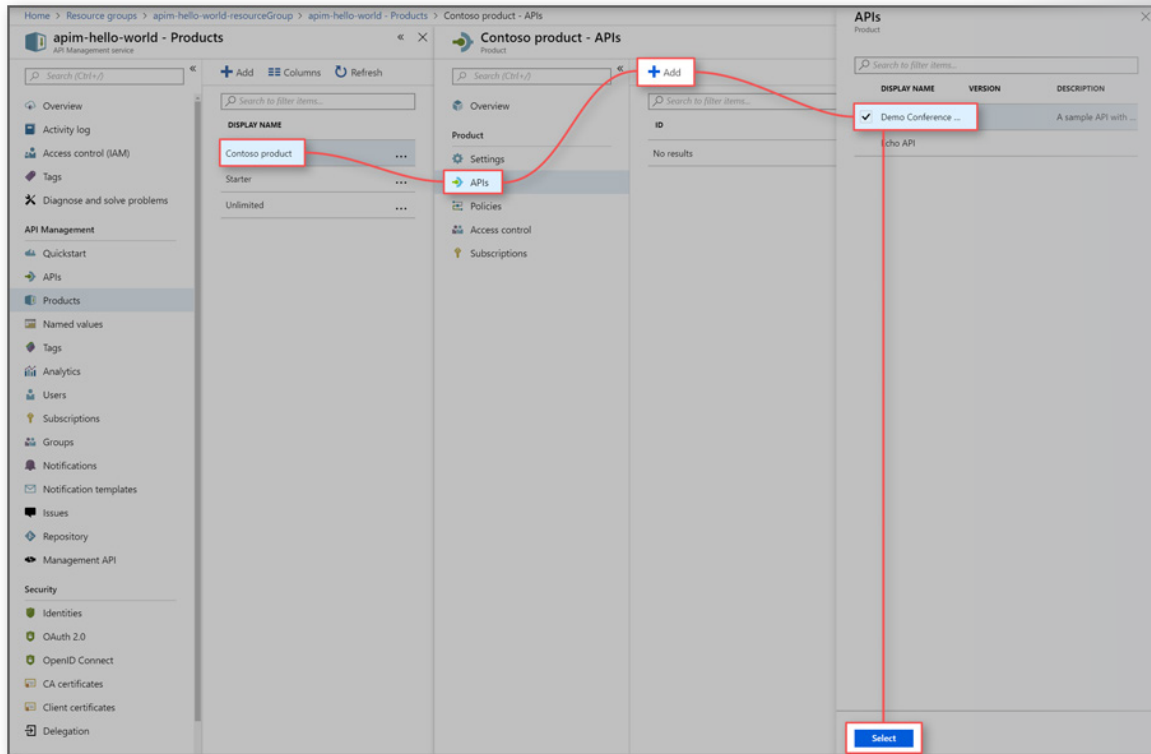
View/add subscribers to the product from the **Subscriptions** tab. Set visibility of a product for developers or guest from the **Access control** tab.

Add APIs to a product

Products are associations of one or more APIs. You can include a number of APIs and offer them to developers through the developer portal. You can add an existing API during the product creation. You can add an API to the product later, either from the Products **Settings** page or while creating an API.

Developers must first subscribe to a product to get access to the API. When they subscribe, they get a subscription key that is good for any API in that product. If you created the APIM instance, you are an administrator already, so you are subscribed to every product by default.

Add an API to an existing product



1. From the **Products** tab, select a product.
2. Navigate to the **APIs** tab.
3. Click **+ Add**.
4. Choose an API and click **Select**.

Tip: You can create or update user's subscription to a Product with custom subscription keys through REST API or PowerShell command.

Using Swagger to document an API

Getting started with Swashbuckle

This section of the course focuses on Swashbuckle to generate Swagger objects in ASP.NET Core. There are three main components to Swashbuckle:

- `Swashbuckle.AspNetCore.Swagger`: a Swagger object model and middleware to expose `SwaggerDocument` objects as JSON endpoints.
- `Swashbuckle.AspNetCore.SwaggerGen`: a Swagger generator that builds `SwaggerDocument` objects directly from your routes, controllers, and models. It's typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.
- `Swashbuckle.AspNetCore.SwaggerUI`: an embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing the Web API functionality. It includes built-in test harnesses for the public methods.

Package installation

Here's how to install the `Swashbuckle.AspNetCore` package in Visual studio:

- From the Package Manager Console window:
 - Go to **View > Other Windows > Package Manager Console**
 - Navigate to the directory in which the `TodoApi.csproj` file exists
 - Execute the following command:
`Install-Package Swashbuckle.AspNetCore`
- From the **Manage NuGet Packages** dialog:
 - Right-click the project in **Solution Explorer > Manage NuGet Packages**
 - Set the **Package source** to "nuget.org"
 - Enter "Swashbuckle.AspNetCore" in the search box
 - Select the "Swashbuckle.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger middleware

Add the Swagger generator to the services collection in the `Startup.ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}
```

```
}
```

Import the following namespace to use the `Info` class:

```
using Swashbuckle.AspNetCore.Swagger;
```

In the `Startup.Configure` method, enable the middleware for serving the generated JSON document and the Swagger UI:

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

Launch the app, and navigate to `http://localhost:<port>/swagger/v1/swagger.json`. The generated document describing the endpoints appears as shown in **Swagger specification (swagger.json)**¹.

The Swagger UI can be found at `http://localhost:<port>/swagger`. Explore the API via Swagger UI and incorporate it in other programs.

Tip: To serve the Swagger UI at the app's root (`http://localhost:<port>/`), set the `RoutePrefix` property to an empty string:

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    c.RoutePrefix = string.Empty;
});
```

Documenting the object model - API info and description

The configuration action passed to the `AddSwaggerGen` method adds information such as the author, license, and description:

```
// Register the Swagger generator, defining 1 or more Swagger documents
services.AddSwaggerGen(c =>
{
```

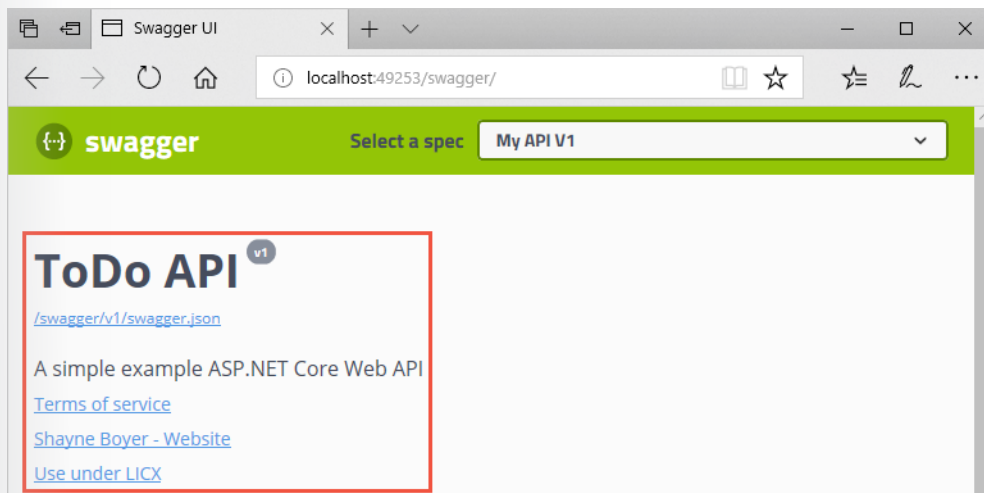
¹ <https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-2.1#swagger-specification-swaggerjson>

```

c.SwaggerDoc("v1", new Info
{
    Version = "v1",
    Title = "ToDo API",
    Description = "A simple example ASP.NET Core Web API",
    TermsOfService = "None",
    Contact = new Contact
    {
        Name = "Shayne Boyer",
        Email = string.Empty,
        Url = "https://twitter.com/spboyer"
    },
    License = new License
    {
        Name = "Use under LICX",
        Url = "https://example.com/license"
    }
});
});

```

The Swagger UI displays the version's information:



Enabling XML comments

XML comments can be enabled in Visual Studio using the following approach:

- Right-click the project in **Solution Explorer** and select **Edit <project_name>.csproj**.
- Manually add the highlighted lines to the .csproj file:

```

<PropertyGroup>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
    <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>

```

Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

```
warning CS1591: Missing XML comment for publicly visible type or member
'TodoController.GetAll()'
```

To suppress warnings project-wide, define a semicolon-delimited list of warning codes to ignore in the project file. Appending the warning codes to `$(NoWarn)`; applies the C# default values too.

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

To suppress warnings only for specific members, enclose the code in `#pragma warning` preprocessor directives. This approach is useful for code that shouldn't be exposed via the API docs. In the following example, warning code CS1591 is ignored for the entire `Program` class. Enforcement of the warning code is restored at the close of the class definition. Specify multiple warning codes with a comma-delimited list.

```
namespace TodoApi
{
    #pragma warning disable CS1591
    public class Program
    {
        public static void Main(string[] args) =>
            BuildWebHost(args).Run();

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
    #pragma warning restore CS1591
}
```

Configure Swagger to use the generated XML file. For Linux or non-Windows operating systems, file names and paths can be case-sensitive. For example, a `TodoApi.XML` file is valid on Windows but not CentOS.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
```

```

        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = "https://twitter.com/spboyer"
            },
            License = new License
            {
                Name = "Use under LICX",
                Url = "https://example.com/license"
            }
        }
    });

    // Set the comments path for the Swagger JSON and UI.
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.
xml";
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
});
}

```

In the preceding code, Reflection is used to build an XML file name matching that of the Web API project. The `AppContext.BaseDirectory` property is used to construct a path to the XML file.

Adding triple-slash comments to an action enhances the Swagger UI by adding the description to the section header. Add a `<summary>` element above the `Delete` action:

```

/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TODOItems.Find(id);

    if (todo == null)
    {
        return NotFound();
    }

    _context.TODOItems.Remove(todo);
    _context.SaveChanges();

    return NoContent();
}

```

The Swagger UI displays the inner text of the preceding code's `<summary>` element:

DELETE `/api/ToDo/{id}` Deletes a specific TodoItem.

Parameters Try it out

Name	Description
id * required integer (path)	

Responses Response content type: application/json

Code	Description
200	Success

The UI is driven by the generated JSON schema:

```
"delete": {
  "tags": [
    "ToDo"
  ],
  "summary": "Deletes a specific TodoItem.",
  "operationId": "ApiToDoByIdDelete",
  "consumes": [],
  "produces": [],
  "parameters": [
    {
      "name": "id",
      "in": "path",
      "description": "",
      "required": true,
      "type": "integer",
      "format": "int64"
    }
  ],
  "responses": {
    "200": {
      "description": "Success"
    }
  }
}
```

Add a `<remarks>` element to the Create action method documentation. It supplements information specified in the `<summary>` element and provides a more robust Swagger UI. The `<remarks>` element content can consist of text, JSON, or XML.

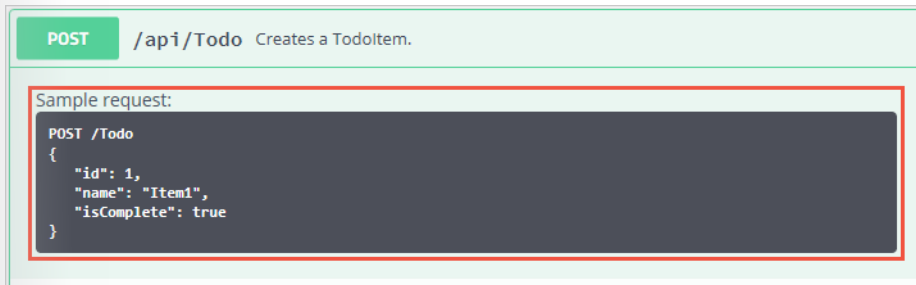

```

    /// <summary>
    /// Creates a TodoItem.
    /// </summary>
    /// <remarks>
    /// Sample request:
    ///
    ///     POST /Todo
    ///     {
    ///         "id": 1,
    ///         "name": "Item1",
    ///         "isComplete": true
    ///     }
    ///
    /// </remarks>
    /// <param name="item"></param>
    /// <returns>A newly created TodoItem</returns>
    /// <response code="201">Returns the newly created item</response>
    /// <response code="400">If the item is null</response>
    [HttpPost]
    [ProducesResponseType(201)]
    [ProducesResponseType(400)]
    public ActionResult<TodoItem> Create(TodoItem item)
    {
        _context.TODOItems.Add(item);
        _context.SaveChanges();

        return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
    }

```

Notice the UI enhancements with these additional comments:



Decorating the model with attributes

Decorate the model with attributes, found in the `System.ComponentModel.DataAnnotations` namespace, to help drive the Swagger UI components.

Add the `[Required]` attribute to the `Name` property of the `TodoItem` class:

```

using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models

```

```

{
    public class TodoItem
    {
        public long Id { get; set; }

        [Required]
        public string Name { get; set; }

        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}

```

The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

```

"definitions": {
  "TodoItem": {
    "required": [
      "name"
    ],
    "type": "object",
    "properties": {
      "id": {
        "format": "int64",
        "type": "integer"
      },
      "name": {
        "type": "string"
      },
      "isComplete": {
        "default": false,
        "type": "boolean"
      }
    }
  }
},

```

Add the `[Produces("application/json")]` attribute to the API controller. Its purpose is to declare that the controller's actions support a response content type of *application/json*:

```

[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;

```

The **Response Content Type** drop-down selects this content type as the default for the controller's GET actions:

The screenshot shows the Swagger UI for the endpoint `GET /api/ToDo`. The 'Parameters' section is empty, indicating no parameters are defined for this endpoint. The 'Responses' section is visible at the bottom, with a dropdown menu for 'Response content type' currently set to 'application/json'. A red rectangle highlights the 'Response content type' dropdown.

As the usage of data annotations in the Web API increases, the UI and API help pages become more descriptive and useful.

Describing response types

Consuming developers are most concerned with what's returned—specifically response types and error codes (if not standard). The response types and error codes are denoted in the XML comments and data annotations.

The `Create` action returns an HTTP 201 status code on success. An HTTP 400 status code is returned when the posted request body is null. Without proper documentation in the Swagger UI, the consumer lacks knowledge of these expected outcomes. Fix that problem by adding the highlighted lines in the following example:

```

/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /ToDo
///     {
///         "id": 1,
///         "name": "Item1",
///         "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TODOItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}

```

```
}
```

The Swagger UI now clearly documents the expected HTTP response codes:

The image shows the 'Responses' section of a Swagger UI. At the top, there's a header with 'Responses' on the left and 'Response content type' with a dropdown menu set to 'application/json' on the right. Below this is a table with two columns: 'Code' and 'Description'. The first row has the code '201' and a description 'Returns the newly created item'. Below the description for '201', there are two tabs: 'Example Value' (selected) and 'Model'. Under the 'Example Value' tab, a JSON object is displayed:

```
{  "id": 0,  "name": "string",  "isComplete": false}
```

. The second row has the code '400' and a description 'If the item is null'. A red rectangular box highlights the '201' response details, including the description and the example JSON value.

Code	Description
201	<p>Returns the newly created item</p> <p>Example Value Model</p> <pre>{ "id": 0, "name": "string", "isComplete": false}</pre>
400	<p>If the item is null</p>

Review questions

Module 3 review questions

API Management groups

Groups are used to manage the visibility of products to developers. Products grant visibility to groups, and developers can view and subscribe to the products that are visible to the groups in which they belong. Can you name the three immutable system groups and their capabilities?

> Click to see suggested answer

API Management has the following immutable system groups:

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

API Debug information

What do you need to enable in order to provide debug information for undocumented public types and members?

> Click to see suggested answer

Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

```
warning CS1591: Missing XML comment for publicly visible type or member  
'TodoController.GetAll()'
```



Implement Azure functions

Azure Functions overview

Introduction to Azure Functions

Azure Functions is a solution for easily running small pieces of code, or “functions,” in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, F#, Node.js, Java, or PHP. Pay only for the time your code runs and trust Azure to scale as needed. Azure Functions lets you develop serverless applications on Microsoft Azure.

What can I do with Functions?

Functions is a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices. Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule.

Functions provides templates to get you started with key scenarios, including the following:

- **HTTPTrigger** - Trigger the execution of your code by using an HTTP request.
- **TimerTrigger** - Execute cleanup or other batch tasks on a predefined schedule.
- **GitHub webhook** - Respond to events that occur in your GitHub repositories.
Generic webhook - Process webhook HTTP requests from any service that supports webhooks.
- **CosmosDBTrigger** - Process Azure Cosmos DB documents when they are added or updated in collections in a NoSQL database.
- **BlobTrigger** - Process Azure Storage blobs when they are added to containers. You might use this function for image resizing.
- **QueueTrigger** - Respond to messages as they arrive in an Azure Storage queue.
- **EventHubTrigger** - Respond to events delivered to an Azure Event Hub. Particularly useful in application instrumentation, user experience or workflow processing, and Internet of Things (IoT) scenarios.

- **ServiceBusQueueTrigger** - Connect your code to other Azure services or on-premises services by listening to message queues.
- **ServiceBusTopicTrigger** - Connect your code to other Azure services or on-premises services by subscribing to topics.

Azure Functions supports *triggers*, which are ways to start execution of your code, and *bindings*, which are ways to simplify coding for input and output data.

Integrations

Azure Functions integrates with various Azure and 3rd-party services. These services can trigger your function and start execution, or they can serve as input and output for your code. The following service integrations are supported by Azure Functions:

- Azure Cosmos DB
- Azure Event Hubs
- Azure Event Grid
- Azure Notification Hubs
- Azure Service Bus (queues and topics)
- Azure Storage (blob, queues, and tables)
- On-premises (using Service Bus)
- Twilio (SMS messages)

How much does Functions cost?

Azure Functions has two kinds of pricing plans. Choose the one that best fits your needs:

- **Consumption plan** - When your function runs, Azure provides all of the necessary computational resources. You don't have to worry about resource management, and you only pay for the time that your code runs.
- **App Service plan** - Run your functions just like your web apps. When you are already using App Service for your other applications, you can run your functions on the same plan at no additional cost.

For more information about hosting plans, see [Azure Functions hosting plan comparison](#)¹.

Azure Functions scale and hosting concepts

Azure Functions runs in two different modes: Consumption plan and Azure App Service plan. The Consumption plan automatically allocates compute power when your code is running. Your app is scaled out when needed to handle load, and scaled down when code is not running. You don't have to pay for idle VMs or reserve capacity in advance.

When you create a function app, you choose the hosting plan for functions in the app. In either plan, an instance of the *Azure Functions host* executes the functions. The type of plan controls:

- How host instances are scaled out.
- The resources that are available to each host.

¹ <https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale>

Important: You must choose the type of hosting plan during the creation of the function app. You can't change it afterward.

On an App Service plan, you can scale between tiers to allocate different amount of resources. On the Consumption plan, Azure Functions automatically handles all resource allocation.

Consumption plan

When you're using a Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. This serverless plan scales automatically, and you're charged for compute resources only when your functions are running. On a Consumption plan, a function execution times out after a configurable period of time.

Note: The default timeout for functions on a Consumption plan is 5 minutes. The value can be increased for the Function App up to a maximum of 10 minutes by changing the property `functionTimeout` in the `host.json` project file.

Billing is based on number of executions, execution time, and memory used. Billing is aggregated across all functions within a function app.

The Consumption plan is the default hosting plan and offers the following benefits:

- Pay only when your functions are running.
- Scale out automatically, even during periods of high load.

App Service plan

In the dedicated App Service plan, your function apps run on dedicated VMs on Basic, Standard, Premium, and Isolated SKUs, which is the same as other App Service apps. Dedicated VMs are allocated to your function app, which means the functions host can be always running. App Service plans support Linux.

Consider an App Service plan in the following cases:

- You have existing, underutilized VMs that are already running other App Service instances.
- Your function apps run continuously, or nearly continuously. In this case, an App Service Plan can be more cost-effective.
- You need more CPU or memory options than what is provided on the Consumption plan.
- Your code needs to run longer than the maximum execution time allowed on the Consumption plan, which is up to 10 minutes.
- You require features that are only available on an App Service plan, such as support for App Service Environment, VNET/VPN connectivity, and larger VM sizes.
- You want to run your function app on Linux, or you want to provide a custom image on which to run your functions.

A VM decouples cost from number of executions, execution time, and memory used. As a result, you won't pay more than the cost of the VM instance that you allocate. With an App Service plan, you can manually scale out by adding more VM instances, or you can enable autoscale.

When running JavaScript functions on an App Service plan, you should choose a plan that has fewer vCPUs.

Always On

If you run on an App Service plan, you should enable the **Always on** setting so that your function app runs correctly. On an App Service plan, the functions runtime goes idle after a few minutes of inactivity, so only HTTP triggers will “wake up” your functions. Always on is available only on an App Service plan. On a Consumption plan, the platform activates function apps automatically.

Storage account requirements

On either a Consumption plan or an App Service plan, a function app requires a general Azure Storage account, which supports Azure Blob, Queue, Files, and Table storage. This is because Functions relies on Azure Storage for operations such as managing triggers and logging function executions, but some storage accounts do not support queues and tables. These accounts, which include blob-only storage accounts (including premium storage) and general-purpose storage accounts with zone-redundant storage replication, are filtered-out from your existing **Storage Account** selections when you create a function app.

How the Consumption plan works

In the Consumption plan, the scale controller automatically scales CPU and memory resources by adding additional instances of the Functions host, based on the number of events that its functions are triggered on. Each instance of the Functions host is limited to 1.5 GB of memory. An instance of the host is the function app, meaning all functions within a function app share resource within an instance and scale at the same time. Function apps that share the same Consumption plan are scaled independently.

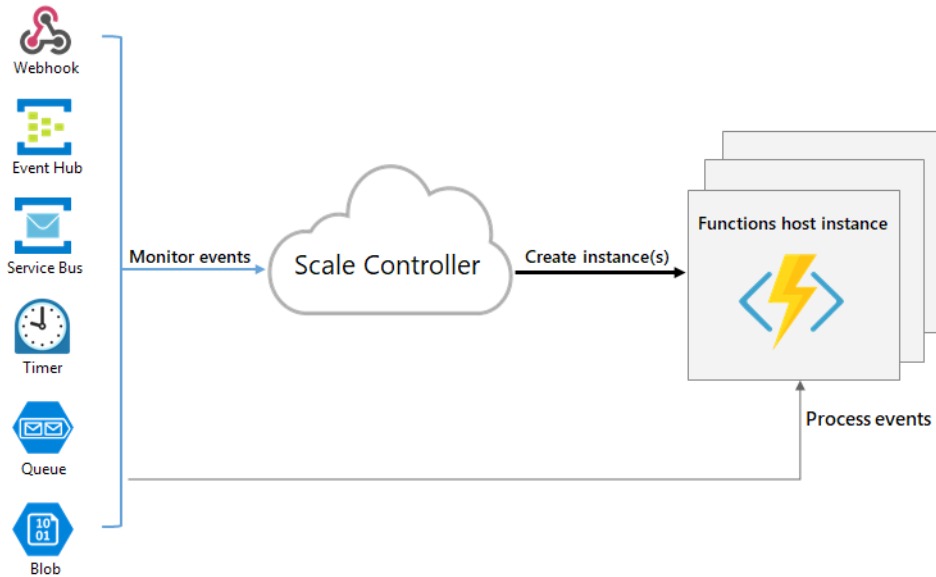
When you use the Consumption hosting plan, function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and cannot be recovered.

Note: When you're using a blob trigger on a Consumption plan, there can be up to a 10-minute delay in processing new blobs. This delay occurs when a function app has gone idle. After the function app is running, blobs are processed immediately. To avoid this cold-start delay, use an App Service plan with **Always On** enabled, or use the Event Grid trigger.

Runtime scaling

Azure Functions uses a component called the scale controller to monitor the rate of events and determine whether to scale out or scale in. The scale controller uses heuristics for each trigger type. For example, when you're using an Azure Queue storage trigger, it scales based on the queue length and the age of the oldest queue message.

The unit of scale is the function app. When the function app is scaled out, additional resources are allocated to run multiple instances of the Azure Functions host. Conversely, as compute demand is reduced, the scale controller removes function host instances. The number of instances is eventually scaled down to zero when no functions are running within a function app.



Understanding scaling behaviors

Scaling can vary on a number of factors, and scale differently based on the trigger and language selected. However there are a few aspects of scaling that exist in the system today:

- A single function app only scales up to a maximum of 200 instances. A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions.
- New instances will only be allocated at most once every 10 seconds.

Different triggers may also have different scaling limits as well.

Azure Functions triggers and bindings concepts

This section is a conceptual overview of triggers and bindings in Azure Functions. Features that are common to all bindings and all supported languages are described here.

Overview

A *trigger* defines how a function is invoked. A function must have exactly one trigger. Triggers have associated data, which is usually the payload that triggered the function.

Input and output *bindings* provide a declarative way to connect to data from within your code. Bindings are optional and a function can have multiple input and output bindings.

Triggers and bindings let you avoid hardcoding the details of the services that you're working with. Your function receives data (for example, the content of a queue message) in function parameters. You send data (for example, to create a queue message) by using the return value of the function. In C# and C# script, alternative ways to send data are `out` parameters and collector objects.

When you develop functions by using the Azure portal, triggers and bindings are configured in a *function.json* file. The portal provides a UI for this configuration but you can edit the file directly by changing to the **Advanced editor**.

When you develop functions by using Visual Studio to create a class library, you configure triggers and bindings by decorating methods and parameters with attributes.

Example trigger and binding

Suppose you want to write a new row to Azure Table storage whenever a new message appears in Azure Queue storage. This scenario can be implemented using an Azure Queue storage trigger and an Azure Table storage output binding.

Here's a *function.json* file for this scenario.

```
{
  "bindings": [
    {
      "name": "order",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "name": "$return",
      "type": "table",
      "direction": "out",
      "tableName": "outTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"
    }
  ]
}
```

The first element in the `bindings` array is the Queue storage trigger. The `type` and `direction` properties identify the trigger. The `name` property identifies the function parameter that receives the queue message content. The name of the queue to monitor is in `queueName`, and the connection string is in the app setting identified by `connection`.

The second element in the `bindings` array is the Azure Table Storage output binding. The `type` and `direction` properties identify the binding. The `name` property specifies how the function provides the new table row, in this case by using the function return value. The name of the table is in `tableName`, and the connection string is in the app setting identified by `connection`.

To view and edit the contents of *function.json* in the Azure portal, click the **Advanced editor** option on the **Integrate** tab of your function.

Note: The value of `connection` is the name of an app setting that contains the connection string, not the connection string itself. Bindings use connection strings stored in app settings to enforce the best practice that *function.json* does not contain service secrets.

Here's C# script code that works with this trigger and binding. Notice that the name of the parameter that provides the queue message content is `order`; this name is required because the `name` property value in *function.json* is `order`.

```
#r "Newtonsoft.Json"

using Microsoft.Extensions.Logging;
```

```

using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and
write to Table storage
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, ILogger log)
{
    return new Person() {
        PartitionKey = "Orders",
        RowKey = Guid.NewGuid().ToString(),
        Name = order["Name"].ToString(),
        MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}

```

Register binding extensions

In some development environments, you have to explicitly register a binding that you want to use. Binding extensions are provided in NuGet packages, and to register an extension you install a package. The following table indicates when and how you register binding extensions.

Development environment	Registration in Functions 1.x	Registration in Functions 2.x
Azure portal	Automatic	Automatic with prompt
Local using Azure Functions Core Tools	Automatic	Use Core Tools CLI commands
C# class library using Visual Studio 2017	Use NuGet tools	Use NuGet tools
C# class library using Visual Studio Code	N/A	Use .NET Core CLI

The following binding types are exceptions that don't require explicit registration because they are automatically registered in all versions and environments: HTTP and timer.

Binding direction

All triggers and bindings have a `direction` property in the `function.json` file:

- For triggers, the direction is always `in`
- Input and output bindings use `in` and `out`
- Some bindings support a special direction `inout`. If you use `inout`, only the **Advanced editor** is available in the **Integrate** tab.

When you use attributes in a class library to configure triggers and bindings, the direction is provided in an attribute constructor or inferred from the parameter type.

Using the function return value

In languages that have a return value, you can bind an output binding to the return value:

- In a C# class library, apply the output binding attribute to the method return value.
- In other languages, set the `name` property in *function.json* to `$return`.

If there are multiple output bindings, use the return value for only one of them.

In C# and C# script, alternative ways to send data to an output binding are `out` parameters and collector objects.

C# example

Here's C# code that uses the return value for an output binding:

```
[FunctionName("QueueTrigger")]
[return: Blob("output-container/{id}")]
public static string Run([QueueTrigger("inputqueue")]WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return json;
}
```

Binding dataType property

In .NET, use the parameter type to define the data type for input data. For instance, use `string` to bind to the text of a queue trigger, a byte array to read as binary and a custom type to deserialize to a POJO object.

For languages that are dynamically typed such as JavaScript, use the `dataType` property in the *function.json* file. For example, to read the content of an HTTP request in binary format, set `dataType` to `binary`:

```
{
  "type": "httpTrigger",
  "name": "req",
  "direction": "in",
  "dataType": "binary"
}
```

Other options for `dataType` are `stream` and `string`.

Binding expressions and patterns

One of the most powerful features of triggers and bindings is binding expressions. In the *function.json* file and in function parameters and code, you can use expressions that resolve to values from various sources.

Most expressions are identified by wrapping them in curly braces. For example, in a queue trigger function, `{queueTrigger}` resolves to the queue message text. If the `path` property for a blob output binding is `container/{queueTrigger}` and the function is triggered by a queue message `HelloWorld`, a blob named `HelloWorld` is created.

Types of binding expressions

- App settings
- Trigger file name
- Trigger metadata
- JSON payloads
- New GUID
- Current date and time

Binding expressions - app settings

As a best practice, secrets and connection strings should be managed using app settings, rather than configuration files. This limits access to these secrets and makes it safe to store files such as *function.json* in public source control repositories.

App settings are also useful whenever you want to change configuration based on the environment. For example, in a test environment, you may want to monitor a different queue or blob storage container.

App setting binding expressions are identified differently from other binding expressions: they are wrapped in percent signs rather than curly braces. For example if the blob output binding path is `%Environment%/newblob.txt` and the `Environment` app setting value is `Development`, a blob will be created in the `Development` container.

When a function is running locally, app setting values come from the *local.settings.json* file.

Note that the `connection` property of triggers and bindings is a special case and automatically resolves values as app settings, without percent signs.

Binding expressions - trigger file name

The `path` for a Blob trigger can be a pattern that lets you refer to the name of the triggering blob in other bindings and function code. The pattern can also include filtering criteria that specify which blobs can trigger a function invocation.

Binding expressions - trigger metadata

In addition to the data payload provided by a trigger (such as the content of the queue message that triggered a function), many triggers provide additional metadata values. These values can be used as input parameters in C# and F# or properties on the `context.bindings` object in JavaScript.

For example, an Azure Queue storage trigger supports the following properties:

- `QueueTrigger` - triggering message content if a valid string
- `DequeueCount`
- `ExpirationTime`
- `Id`

- InsertionTime
- NextVisibleTime
- PopReceipt

These metadata values are accessible in *function.json* file properties. For example, suppose you use a queue trigger and the queue message contains the name of a blob you want to read. In the *function.json* file, you can use `queueTrigger` metadata property in the blob `path` property.

Binding expressions - JSON payloads

When a trigger payload is JSON, you can refer to its properties in configuration for other bindings in the same function and in function code.

The following example shows the *function.json* file for a webhook function that receives a blob name in JSON: `{"BlobName": "HelloWorld.txt"}`. A Blob input binding reads the blob, and the HTTP output binding returns the blob contents in the HTTP response. Notice that the Blob input binding gets the blob name by referring directly to the `BlobName` property `"path": "strings/{BlobName}"`:

```
{
  "bindings": [
    {
      "name": "info",
      "type": "httpTrigger",
      "direction": "in",
      "webHookType": "genericJson"
    },
    {
      "name": "blobContents",
      "type": "blob",
      "direction": "in",
      "path": "strings/{BlobName}",
      "connection": "AzureWebJobsStorage"
    },
    {
      "name": "res",
      "type": "http",
      "direction": "out"
    }
  ]
}
```

Binding expressions - create GUIDs

The `{rand-guid}` binding expression creates a GUID. The following blob path in a *function.json* file creates a blob with a name like `50710cb5-84b9-4d87-9d83-a03d6976a682.txt`.

```
{
  "type": "blob",
  "name": "blobOutput",
  "direction": "out",
  "path": "my-output-container/{rand-guid}"
}
```

```
}
```

Binding expressions - current time

The binding expression `DateTime` resolves to `DateTime.UtcNow`. The following blob path in a *function.json* file creates a blob with a name like *2018-02-16T17-59-55Z.txt*.

```
{
  "type": "blob",
  "name": "blobOutput",
  "direction": "out",
  "path": "my-output-container/{DateTime}"
}
```

Optimize the performance and reliability of Azure Functions

This section provides guidance to improve the performance and reliability of your serverless function apps.

General best practices

The following are best practices in how you build and architect your serverless solutions using Azure Functions.

Avoid long running functions

Large, long-running functions can cause unexpected timeout issues. A function can become large due to many Node.js dependencies. Importing dependencies can also cause increased load times that result in unexpected timeouts. Dependencies are loaded both explicitly and implicitly. A single module loaded by your code may load its own additional modules.

Whenever possible, refactor large functions into smaller function sets that work together and return responses fast. For example, a webhook or HTTP trigger function might require an acknowledgment response within a certain time limit; it is common for webhooks to require an immediate response. You can pass the HTTP trigger payload into a queue to be processed by a queue trigger function. This approach allows you to defer the actual work and return an immediate response.

Cross function communication

Durable Functions and Azure Logic Apps are built to manage state transitions and communication between multiple functions.

If not using Durable Functions or Logic Apps to integrate with multiple functions, it is generally a best practice to use storage queues for cross function communication. The main reason is storage queues are cheaper and much easier to provision.

Individual messages in a storage queue are limited in size to 64 KB. If you need to pass larger messages between functions, an Azure Service Bus queue could be used to support message sizes up to 256 KB in the Standard tier, and up to 1 MB in the Premium tier.

Service Bus topics are useful if you require message filtering before processing.

Event hubs are useful to support high volume communications.

Write functions to be stateless

Functions should be stateless and idempotent if possible. Associate any required state information with your data. For example, an order being processed would likely have an associated `state` member. A function could process an order based on that state while the function itself remains stateless.

Idempotent functions are especially recommended with timer triggers. For example, if you have something that absolutely must run once a day, write it so it can run any time during the day with the same results. The function can exit when there is no work for a particular day. Also if a previous run failed to complete, the next run should pick up where it left off.

Write defensive functions

Assume your function could encounter an exception at any time. Design your functions with the ability to continue from a previous fail point during the next execution. Consider a scenario that requires the following actions:

1. Query for 10,000 rows in a db.
2. Create a queue message for each of those rows to process further down the line.

Depending on how complex your system is, you may have: involved downstream services behaving badly, networking outages, or quota limits reached, etc. All of these can affect your function at any time. You need to design your functions to be prepared for it.

How does your code react if a failure occurs after inserting 5,000 of those items into a queue for processing? Track items in a set that you've completed. Otherwise, you might insert them again next time. This can have a serious impact on your work flow.

If a queue item was already processed, allow your function to be a no-op.

Scalability best practices

There are a number of factors which impact how instances of your function app scale. The details were covered earlier in this lesson. The following are some best practices to ensure optimal scalability of a function app.

Share and manage connections

Re-use connections to external resources whenever possible. See how to manage connections in Azure Functions.

Don't mix test and production code in the same function app

Functions within a function app share resources. For example, memory is shared. If you're using a function app in production, don't add test-related functions and resources to it. It can cause unexpected overhead during production code execution.

Be careful what you load in your production function apps. Memory is averaged across each function in the app.

If you have a shared assembly referenced in multiple .Net functions, put it in a common shared folder. Reference the assembly with a statement similar to the following example if using C# Scripts (.csx):

```
#r "..\Shared\MyAssembly.dll".
```

Otherwise, it is easy to accidentally deploy multiple test versions of the same binary that behave differently between functions.

Don't use verbose logging in production code. It has a negative performance impact.

Use async code but avoid blocking calls

Asynchronous programming is a recommended best practice. However, always avoid referencing the `Result` property or calling `Wait` method on a `Task` instance. This approach can lead to thread exhaustion.

Tip: If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`.

Receive messages in batch whenever possible

Some triggers like Event Hub enable receiving a batch of messages on a single invocation. Batching messages has much better performance. You can configure the max batch size in the `host.json` file as detailed in the [host.json reference documentation](#)².

For C# functions you can change the type to a strongly-typed array. For example, instead of `EventData sensorEvent` the method signature could be `EventData[] sensorEvent`. For other languages you'll need to explicitly set the cardinality property in your `function.json` to `many` in order to enable batching as shown here.

Configure host behaviors to better handle concurrency

The `host.json` file in the function app allows for configuration of host runtime and trigger behaviors. In addition to batching behaviors, you can manage concurrency for a number of triggers. Often adjusting the values in these options can help each instance scale appropriately for the demands of the invoked functions.

Settings in the hosts file apply across all functions within the app, within a single instance of the function. For example, if you had a function app with 2 HTTP functions and concurrent requests set to 25, a request to either HTTP trigger would count towards the shared 25 concurrent requests. If that function app scaled to 10 instances, the 2 functions would effectively allow 250 concurrent requests (10 instances * 25 concurrent requests per instance).

² <https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json>

Develop Azure Functions using Visual Studio

Getting started

Azure Functions Tools for Visual Studio 2017 is an extension for Visual Studio that lets you develop, test, and deploy C# functions to Azure.

The Azure Functions Tools provides the following benefits:

- Edit, build, and run functions on your local development computer.
- Publish your Azure Functions project directly to Azure.
- Use WebJobs attributes to declare function bindings directly in the C# code instead of maintaining a separate function.json for binding definitions.
- Develop and deploy pre-compiled C# functions. Pre-compiled functions provide a better cold-start performance than C# script-based functions.
- Code your functions in C# while having all of the benefits of Visual Studio development.

Important: Don't mix local development with portal development in the same function app. When you publish from a local project to a function app, the deployment process overwrites any functions that you developed in the portal.

Prerequisites

Azure Functions Tools is included in the Azure development workload of Visual Studio 2017 version 15.5, or a later version. Make sure you include the Azure development workload in your Visual Studio 2017 installation. Also make sure that your Visual Studio is up-to-date and that you are using the most recent version of the Azure Functions tools.

You will also need:

- An active Azure subscription
- An Azure Storage account.

Creating an Azure Functions project

The Azure Functions project template in Visual Studio creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio, select **New > Project** from the File menu.
2. In the **New Project** dialog, select **Installed**, expand **Visual C# > Cloud**, select **Azure Functions**, type a **Name** for your project, and click **OK**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.
3. Use the settings specified in the table below.

Setting	Suggested Value	Description
---------	-----------------	-------------

Version	Azure Functions v1	This creates a function project that uses the version 1 runtime of Azure Functions. The version 2 runtime, which supports .NET Core, is currently in preview.
Template	HTTP trigger	This creates a function triggered by an HTTP request.
Storage account	Storage emulator	An HTTP trigger doesn't use the Storage account connection. All other trigger types require a valid Storage account connection string.
Access rights	Anonymous	The created function can be triggered by any client without providing a key. This authorization setting makes it easy to test your new function.

4. Click **OK** to create the function project and HTTP triggered function.

The project template creates a C# project, installs the `Microsoft.NET.Sdk.Functions` NuGet package, and sets the target framework. Functions 1.x targets the .NET Framework, and Functions 2.x targets .NET Standard. The new project has the following files:

- **host.json:** Lets you configure the Functions host. These settings apply both when running locally and in Azure.
- **local.settings.json:** Maintains settings used when running functions locally. These settings are not used by Azure, they are used by the Azure Functions Core Tools. Use this file to specify app settings for variables required by your functions. Add a new item to the **Values** array for each connection required by the functions bindings in your project.

Configure the project for local development

The Functions runtime uses an Azure Storage account internally. For all trigger types other than HTTP and webhooks, you must set the **Values.AzureWebJobsStorage** key to a valid Azure Storage account connection string. Your function app can also use the Azure storage emulator for the **AzureWebJobsStorage** connection setting that is required by the project. To use the emulator, set the value of **AzureWebJobsStorage** to `UseDevelopmentStorage=true`. You must change this setting to an actual storage connection before deployment.

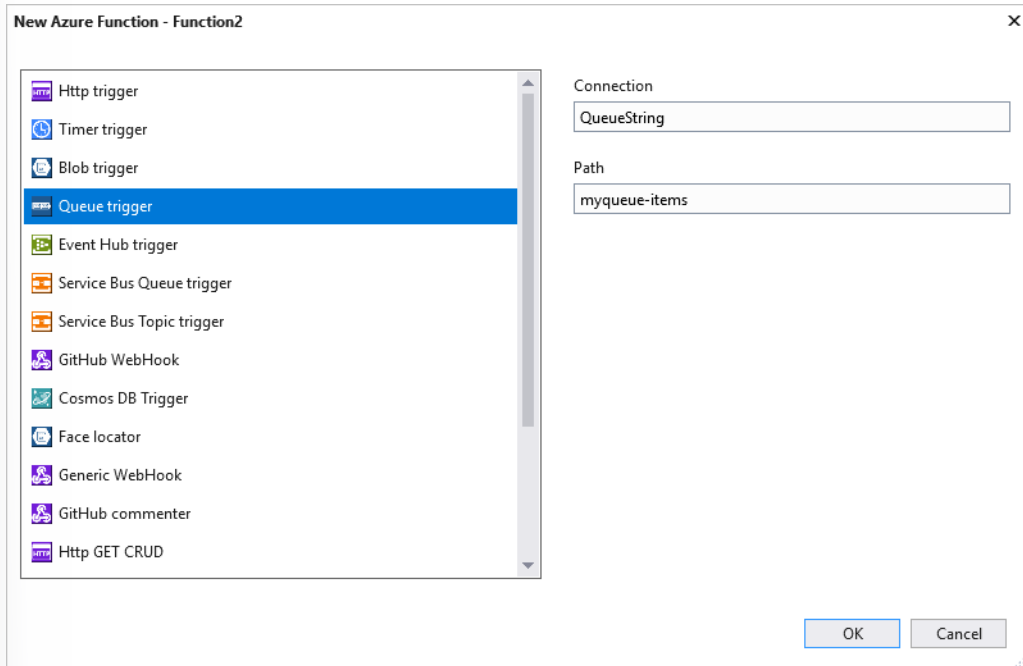
To set the storage account connection string:

- In Visual Studio, open **Cloud Explorer**, expand **Storage Account > Your Storage Account**, then select **Properties** and copy the **Primary Connection String** value.
- In your project, open the `local.settings.json` file and set the value of the **AzureWebJobsStorage** key to the connection string you copied.
- Repeat the previous step to add unique keys to the **Values** array for any other connections required by your functions.

Creating a function

In pre-compiled functions, the bindings used by the function are defined by applying attributes in the code. When you use the Azure Functions Tools to create your functions from the provided templates, these attributes are applied for you.

1. In **Solution Explorer**, right-click on your project node and select **Add > New Item**. Select **Azure Function**, type a **Name** for the class, and click **Add**.
2. Choose your trigger, set the binding properties, and click **Create**. The following example shows the settings when creating a Queue storage triggered function.



- 3.
4. This trigger example uses a connection string with a key named **QueueStorage**. This connection string setting must be defined in the *local.settings.json* file.
5. Examine the newly added class. You see a static **Run** method, that is attributed with the **Function-Name** attribute. This attribute indicates that the method is the entry point for the function.
6. For example, the following C# class represents a basic Queue storage triggered function:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

namespace FunctionApp1
{
    public static class Function1
    {
        [FunctionName("QueueTriggerCSharp")]
        public static void Run([QueueTrigger("myqueue-items", Connection =
"QueueStorage")]string myQueueItem, TraceWriter log)
        {
            log.Info($"C# Queue trigger function processed: {myQueue-
Item}");
        }
    }
}
```

```

    }
}

```

7. A binding-specific attribute is applied to each binding parameter supplied to the entry point method. The attribute takes the binding information as parameters. In the previous example, the first parameter has a **QueueTrigger** attribute applied, indicating queue triggered function. The queue name and connection string setting name are passed as parameters to the **QueueTrigger** attribute.

You can use the above procedure to add more functions to your function app project. Each function in the project can have a different trigger, but a function must have exactly one trigger.

Add bindings to the Azure Function

As with triggers, input and output bindings are added to your function as binding attributes. Add bindings to a function as follows:

1. Make sure you have configured the project for local development.
2. Add the appropriate NuGet extension package for the specific binding. The binding-specific NuGet package requirements are found in the reference article for the binding. For example, find package requirements for the Event Hubs trigger in the **Event Hubs binding reference article**³.
3. If there are app settings that the binding needs, add them to the **Values** collection in the local setting file. These values are used when the function runs locally. When the function runs in the function app in Azure, the function app settings are used.
4. Add the appropriate binding attribute to the method signature. In the following example, a queue message triggers the function, and the output binding creates a new queue message with the same text in a different queue.

```

public static class SimpleExampleWithOutput
{
    [FunctionName("CopyQueueMessage")]
    public static void Run(
        [QueueTrigger("myqueue-items-source", Connection = "AzureWebJobsStorage")] string myQueueItem,
        [Queue("myqueue-items-destination", Connection = "AzureWebJobsStorage")] out string myQueueItemCopy,
        TraceWriter log)
    {
        log.Info($"CopyQueueMessage function processed: {myQueueItem}");
        myQueueItemCopy = myQueueItem;
    }
}

```

5. The connection to Queue storage is obtained from the AzureWebJobsStorage setting. For more information, see the reference article for the specific binding.

³ <https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs>

Testing and publishing Azure Functions

Azure Functions Core Tools lets you run Azure Functions project on your local development computer. You are prompted to install these tools the first time you start a function from Visual Studio.

To test your function, press **F5**. If prompted, accept the request from Visual Studio to download and install Azure Functions Core (CLI) tools. You may also need to enable a firewall exception so that the tools can handle HTTP requests.

With the project running, you can test your code as you would test deployed function. For more information, see **Strategies for testing your code in Azure Functions**⁴. When running in debug mode, breakpoints are hit in Visual Studio as expected.

Publish to Azure

1. In **Solution Explorer**, right-click the project and select **Publish**.
2. Select **Azure Function App**, choose **Create New**, and then select **Publish**.
3. If you haven't already connected Visual Studio to your Azure account, select **Add an account...**
4. In the Create App Service dialog, use the Hosting settings as specified in the table below:

Setting	Suggested Value	Description
App Name	Globally unique name	Name that uniquely identifies your new function app.
Subscription	Choose your subscription	The Azure subscription to use.
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose New to create a new resource group.
App Service Plan	Consumption plan	Make sure to choose the Consumption under Size after you click New to create a plan. Also, choose a Location in a region near you or near other services your functions access.
Storage Account	General purpose storage account	An Azure storage account is required by the Functions runtime. Click New to create a general purpose storage account, or use an existing one.

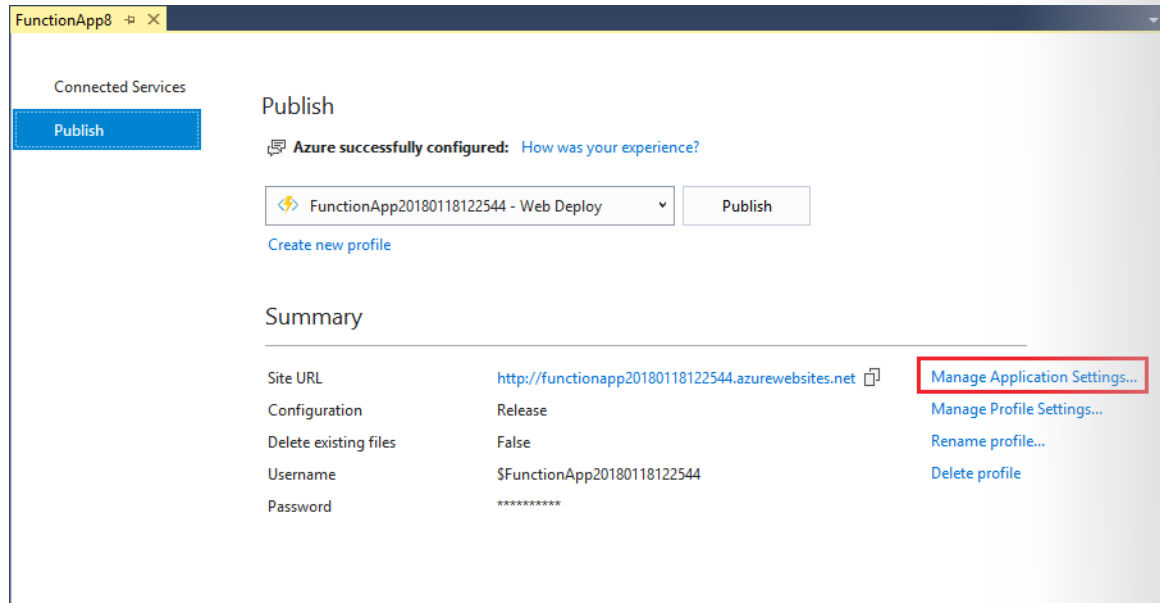
5. Click **Create** to create a function app and related resources in Azure with these settings and deploy your function project code.
6. After the deployment is complete, make a note of the **Site URL** value, which is the address of your function app in Azure.

⁴ <https://docs.microsoft.com/en-us/azure/azure-functions/functions-test-a-function>

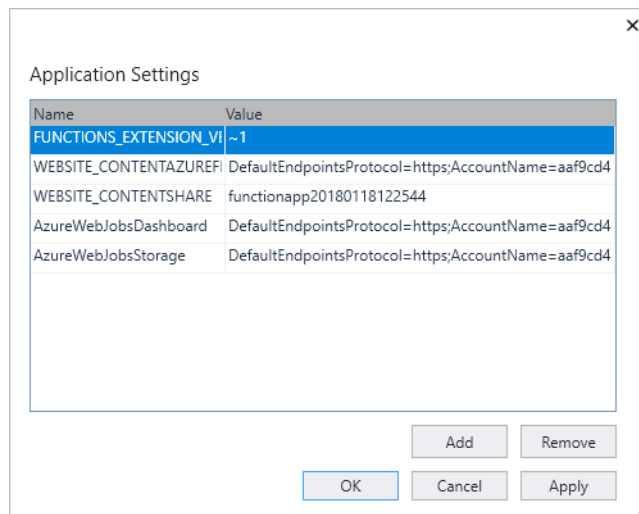
Function app settings

Any settings you added in the `local.settings.json` must be also added to the function app in Azure. These settings are not uploaded automatically when you publish the project.

The easiest way to upload the required settings to your function app in Azure is to use the **Manage Application Settings...** link that is displayed after you successfully publish your project.



This displays the **Application Settings** dialog for the function app, where you can add new application settings or modify existing ones.



You can also manage application settings in one of these other ways:

- Using the Azure portal.
- Using the `--publish-local-settings` publish option in the Azure Functions Core Tools.
- Using the Azure CLI.

Implement Durable Functions

Durable Functions overview

Durable Functions is an extension of Azure Functions and Azure WebJobs that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

The extension lets you define stateful workflows in a new type of function called an *orchestrator function*. Here are some of the advantages of orchestrator functions:

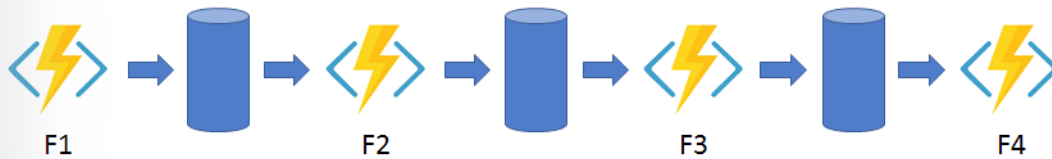
- They define workflows in code. No JSON schemas or designers are needed.
- They can call other functions synchronously and asynchronously. Output from called functions can be saved to local variables.
- They automatically checkpoint their progress whenever the function awaits. Local state is never lost if the process recycles or the VM reboots.

Note: Durable Functions is an advanced extension for Azure Functions that is not appropriate for all applications. The rest of this section assumes that you have a strong familiarity with Azure Functions concepts and the challenges involved in serverless application development.

The primary use case for Durable Functions is simplifying complex, stateful coordination problems in serverless applications. The following sections describe some typical application patterns that can benefit from Durable Functions.

Pattern #1: Function chaining

Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function.



Durable Functions allows you to implement this pattern concisely in code.

C# script

```
public static async Task<object> Run(DurableOrchestrationContext ctx)
{
    try
    {
        var x = await ctx.CallActivityAsync<object>("F1");
        var y = await ctx.CallActivityAsync<object>("F2", x);
        var z = await ctx.CallActivityAsync<object>("F3", y);
        return await ctx.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // error handling/compensation goes here
    }
}
```

```

    }
}

```

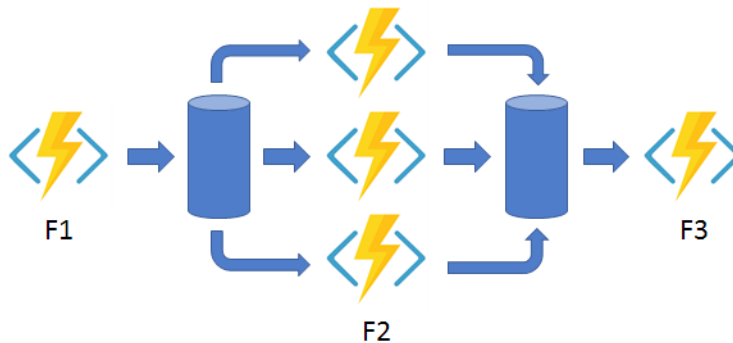
Note: There are subtle differences while writing a precompiled durable function in C# vs the C# script sample shown before. A C# precompiled function would require durable parameters to be decorated with respective attributes. An example is `[OrchestrationTrigger]` attribute for `DurableOrchestrationContext` parameter. If the parameters are not properly decorated, the runtime would not be able to inject the variables to the function and would give error.

The values "F1", "F2", "F3", and "F4" are the names of other functions in the function app. Control flow is implemented using normal imperative coding constructs. That is, code executes top-down and can involve existing language control flow semantics, like conditionals, and loops. Error handling logic can be included in try/catch/finally blocks.

The `ctx` parameter (`DurableOrchestrationContext`) provides methods for invoking other functions by name, passing parameters, and returning function output. Each time the code calls `await`, the Durable Functions framework checkpoints the progress of the current function instance. If the process or VM recycles midway through the execution, the function instance resumes from the previous `await` call. More on this restart behavior later.

Pattern #2: Fan-out/fan-in

Fan-out/fan-in refers to the pattern of executing multiple functions in parallel, and then waiting for all to finish. Often some aggregation work is done on results returned from the functions.



With normal functions, fanning out can be done by having the function send multiple messages to a queue. However, fanning back in is much more challenging. You'd have to write code to track when the queue-triggered functions end and store function outputs. The Durable Functions extension handles this pattern with relatively simple code.

C# script

```

public static async Task Run(DurableOrchestrationContext ctx)
{
    var parallelTasks = new List<Task<int>>();

    // get a list of N work items to process in parallel
    object[] workBatch = await ctx.CallActivityAsync<object[]>("F1");
    for (int i = 0; i < workBatch.Length; i++)
    {

```

```

        Task<int> task = ctx.CallActivityAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // aggregate all N outputs and send result to F3
    int sum = parallelTasks.Sum(t => t.Result);
    await ctx.CallActivityAsync("F3", sum);
}

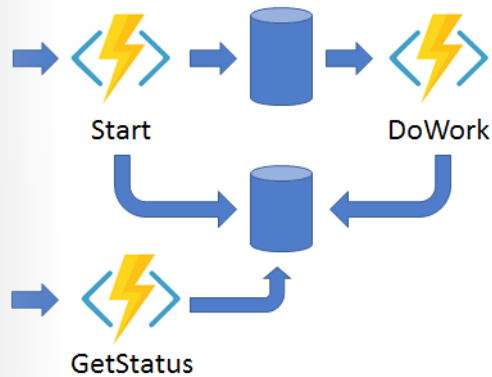
```

The fan-out work is distributed to multiple instances of function F2, and the work is tracked by using a dynamic list of tasks. The .NET `Task.WhenAll` API is called to wait for all of the called functions to finish. Then the F2 function outputs are aggregated from the dynamic task list and passed on to the F3 function.

The automatic checkpointing that happens at the `await` call on `Task.WhenAll` ensures that any crash or reboot midway through does not require a restart of any already completed tasks.

Pattern #3: Async HTTP APIs

The third pattern is all about the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having the long-running action triggered by an HTTP call, and then redirecting the client to a status endpoint that they can poll to learn when the operation completes.



Durable Functions provides built-in APIs that simplify the code you write for interacting with long-running function executions. Once an instance is started, the extension exposes webhook HTTP APIs that query the orchestrator function status. The following example shows the REST commands to start an orchestrator and to query its status. For clarity, some details are omitted from the example.

```

> curl -X POST https://myfunc.azurewebsites.net/orchestrators/DoWork -H
"Content-Length: 0" -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location: https://myfunc.azurewebsites.net/admin/extensions/DurableTaskEx-
tension/b79baf67f717453ca9e86c5da21e03ec

{"id":"b79baf67f717453ca9e86c5da21e03ec", ...}

```

```
> curl https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location: https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec

{"runtimeStatus":"Running","lastUpdatedTime":"2017-03-16T21:20:47Z", ...}

> curl https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExtension/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 200 OK
Content-Length: 175
Content-Type: application/json

{"runtimeStatus":"Completed","lastUpdatedTime":"2017-03-16T21:20:57Z", ...}
```

Because the state is managed by the Durable Functions runtime, you don't have to implement your own status tracking mechanism.

Even though the Durable Functions extension has built-in webhooks for managing long-running orchestrations, you can implement this pattern yourself using your own function triggers (such as HTTP, queue, or Event Hub) and the `orchestrationClient` binding. For example, you could use a queue message to trigger termination. Or you could use an HTTP trigger protected by an Azure Active Directory authentication policy instead of the built-in webhooks that use a generated key for authentication.

```
// HTTP-triggered function to start a new orchestrator function instance.
public static async Task<HttpResponseMessage> Run(
    HttpRequestMessage req,
    DurableOrchestrationClient starter,
    string functionName,
    ILogger log)
{
    // Function name comes from the request URL.
    // Function input comes from the request content.
    dynamic eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, eventData);

    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

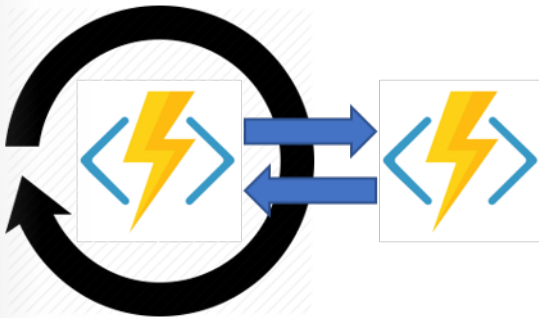
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

The `DurableOrchestrationClientstarter` parameter is a value from the `orchestrationClient` output binding, which is part of the Durable Functions extension. It provides methods for starting, sending events to, terminating, and querying for new or existing orchestrator function instances. In the previous example, an HTTP triggered-function takes in a `functionName` value from the incoming URL and passes that value to `StartNewAsync`. This binding API then returns a response that contains a `Location` header and additional information about the instance that can later be used to look up the status of the started instance or terminate it.

Pattern #4: Monitoring

The monitor pattern refers to a flexible recurring process in a workflow - for example, polling until certain conditions are met. A regular timer-trigger can address a simple scenario, such as a periodic cleanup job, but its interval is static and managing instance lifetimes becomes complex. Durable Functions enables flexible recurrence intervals, task lifetime management, and the ability to create multiple monitor processes from a single orchestration.

An example would be reversing the earlier async HTTP API scenario. Instead of exposing an endpoint for an external client to monitor a long-running operation, the long-running monitor consumes an external endpoint, waiting for some state change.



Using Durable Functions, multiple monitors that observe arbitrary endpoints can be created in a few lines of code. The monitors can end execution when some condition is met, or be terminated by the `DurableOrchestrationClient`, and their wait interval can be changed based on some condition (i.e. exponential backoff.) The following code implements a basic monitor.

C# script

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    int jobId = ctx.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (ctx.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await ctx.CallActivityAsync<string>("GetJobStatus",
jobId);
        if (jobStatus == "Completed")
        {
            // Perform action when condition met
            await ctx.CallActivityAsync("SendAlert", machineId);
            break;
        }

        // Orchestration will sleep until this time
        var nextCheck = ctx.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await ctx.CreateTimer(nextCheck, CancellationToken.None);
    }
}
```

```

    // Perform further work here, or let the orchestration end
}

```

When a request is received, a new orchestration instance is created for that job ID. The instance polls a status until a condition is met and the loop is exited. A durable timer is used to control the polling interval. Further work can then be performed, or the orchestration can end. When the `ctx.CurrentUtcDateTime` exceeds the `expiryTime`, the monitor ends.

Pattern #5: Human interaction

Many processes involve some kind of human interaction. The tricky thing about involving humans in an automated process is that people are not always as highly available and responsive as cloud services. Automated processes must allow for this, and they often do so by using timeouts and compensation logic.

One example of a business process that involves human interaction is an approval process. For example, approval from a manager might be required for an expense report that exceeds a certain amount. If the manager does not approve within 72 hours (maybe they went on vacation), an escalation process kicks in to get the approval from someone else (perhaps the manager's manager).



This pattern can be implemented using an orchestrator function. The orchestrator would use a durable timer to request approval and escalate in case of timeout. It would wait for an external event, which would be the notification generated by some human interaction.

C# script

```

public static async Task Run(DurableOrchestrationContext ctx)
{
    await ctx.CallActivityAsync("RequestApproval");
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = ctx.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = ctx.CreateTimer(dueTime, timeoutCts.Token);

        Task<bool> approvalEvent = ctx.WaitForExternalEvent<bool>("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTimeout))
        {
            timeoutCts.Cancel();
            await ctx.CallActivityAsync("ProcessApproval", approvalEvent.

```

```
Result);  
    }  
    else  
    {  
        await ctx.CallActivityAsync("Escalate");  
    }  
}  
}
```

The durable timer is created by calling `ctx.CreateTimer`. The notification is received by `ctx.WaitForExternalEvent`. And `Task.WhenAny` is called to decide whether to escalate (timeout happens first) or process approval (approval is received before timeout).

An external client can deliver the event notification to a waiting orchestrator function using either the built-in HTTP APIs or by using `DurableOrchestrationClient.RaiseEventAsync` API from another function:

```
public static async Task Run(string instanceId, DurableOrchestrationClient  
client)  
{  
    bool isApproved = true;  
    await client.RaiseEventAsync(instanceId, "ApprovalEvent", isApproved);  
}
```

The technology

Behind the scenes, the Durable Functions extension is built on top of the Durable Task Framework, an open-source library on GitHub for building durable task orchestrations. Much like how Azure Functions is the serverless evolution of Azure WebJobs, Durable Functions is the serverless evolution of the Durable Task Framework. The Durable Task Framework is used heavily within Microsoft and outside as well to automate mission-critical processes. It's a natural fit for the serverless Azure Functions environment.

Event sourcing, checkpointing, and replay

Orchestrator functions reliably maintain their execution state using a design pattern known as Event Sourcing. Instead of directly storing the current state of an orchestration, the durable extension uses an append-only store to record the full series of actions taken by the function orchestration. This has many benefits, including improving performance, scalability, and responsiveness compared to “dumping” the full runtime state. Other benefits include providing eventual consistency for transactional data and maintaining full audit trails and history. The audit trails themselves enable reliable compensating actions.

The use of Event Sourcing by this extension is transparent. Under the covers, the `await` operator in an orchestrator function yields control of the orchestrator thread back to the Durable Task Framework dispatcher. The dispatcher then commits any new actions that the orchestrator function scheduled (such as calling one or more child functions or scheduling a durable timer) to storage. This transparent commit action appends to the execution history of the orchestration instance. The history is stored in a storage table. The commit action then adds messages to a queue to schedule the actual work. At this point, the orchestrator function can be unloaded from memory. Billing for it stops if you're using the Azure Functions Consumption Plan. When there is more work to do, the function is restarted and its state is reconstructed.

Once an orchestration function is given more work to do (for example, a response message is received or a durable timer expires), the orchestrator wakes up again and re-executes the entire function from the

start in order to rebuild the local state. If during this replay the code tries to call a function (or do any other async work), the Durable Task Framework consults with the execution history of the current orchestration. If it finds that the activity function has already executed and yielded some result, it replays that function's result, and the orchestrator code continues running. This continues happening until the function code gets to a point where either it is finished or it has scheduled new async work.

Orchestrator code constraints

The replay behavior creates constraints on the type of code that can be written in an orchestrator function. For example, orchestrator code must be deterministic, as it will be replayed multiple times and must produce the same result each time.

Language support

Currently C# (Functions v1 and v2), F# and JavaScript (Functions v2 only) are the only supported languages for Durable Functions. This includes orchestrator functions and activity functions. In the future, we will add support for all languages that Azure Functions supports. See the Azure Functions GitHub repository issues list to see the latest status of our additional language support work.

Storage and scalability

The Durable Functions extension uses Azure Storage queues, tables, and blobs to persist execution history state and trigger function execution. The default storage account for the function app can be used, or you can configure a separate storage account. You might want a separate account due to storage throughput limits. The orchestrator code you write does not need to (and should not) interact with the entities in these storage accounts. The entities are managed directly by the Durable Task Framework as an implementation detail.

Orchestrator functions schedule activity functions and receive their responses via internal queue messages. When a function app runs in the Azure Functions Consumption plan, these queues are monitored by the Azure Functions Scale Controller and new compute instances are added as needed. When scaled out to multiple VMs, an orchestrator function may run on one VM while activity functions it calls run on several different VMs. You can find more details on the scale behavior of Durable Functions in Performance and scale.

Table storage is used to store the execution history for orchestrator accounts. Whenever an instance rehydrates on a particular VM, it fetches its execution history from table storage so that it can rebuild its local state. One of the convenient things about having the history available in Table storage is that you can take a look and see the history of your orchestrations using tools such as Microsoft Azure Storage Explorer.

Storage blobs are used primarily as a leasing mechanism to coordinate the scale-out of orchestration instances across multiple VMs. They are also used to hold data for large messages which cannot be stored directly in tables or queues.

Create a Durable Function in C#

In this tutorial, you learn how to use the Visual Studio 2017 tools for Azure Functions to locally create and test a "hello world" durable function. This function will orchestrate and chain together calls to other functions. You then publish the function code to Azure. These tools are available as part of the Azure development workload in Visual Studio 2017.

Prerequisites

To complete this tutorial:

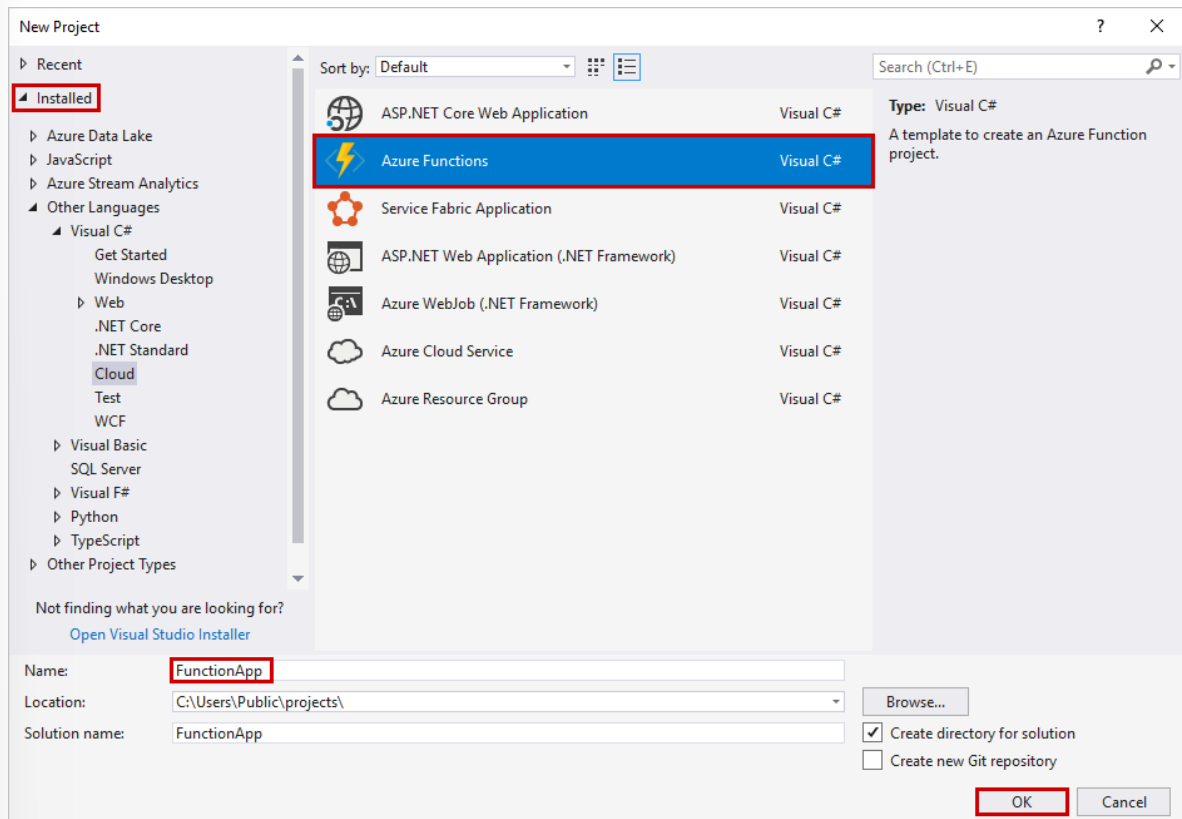
- Install **Visual Studio 2017**⁵ and ensure that the **Azure development** workload is also installed.
- Make sure you have the latest **Azure Functions tools**⁶.
- Verify you have the **Azure Storage Emulator**⁷ installed and running.

If you don't have an Azure subscription, create a free account before you begin.

Create a function app project

The Azure Functions project template in Visual Studio creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio, select **New > Project** from the **File** menu.
2. In the **New Project dialog**, select **Installed**, expand **Visual C# > Cloud**, select **Azure Functions**, type a **Name** for your project, and click **OK**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other nonalphanumeric characters.

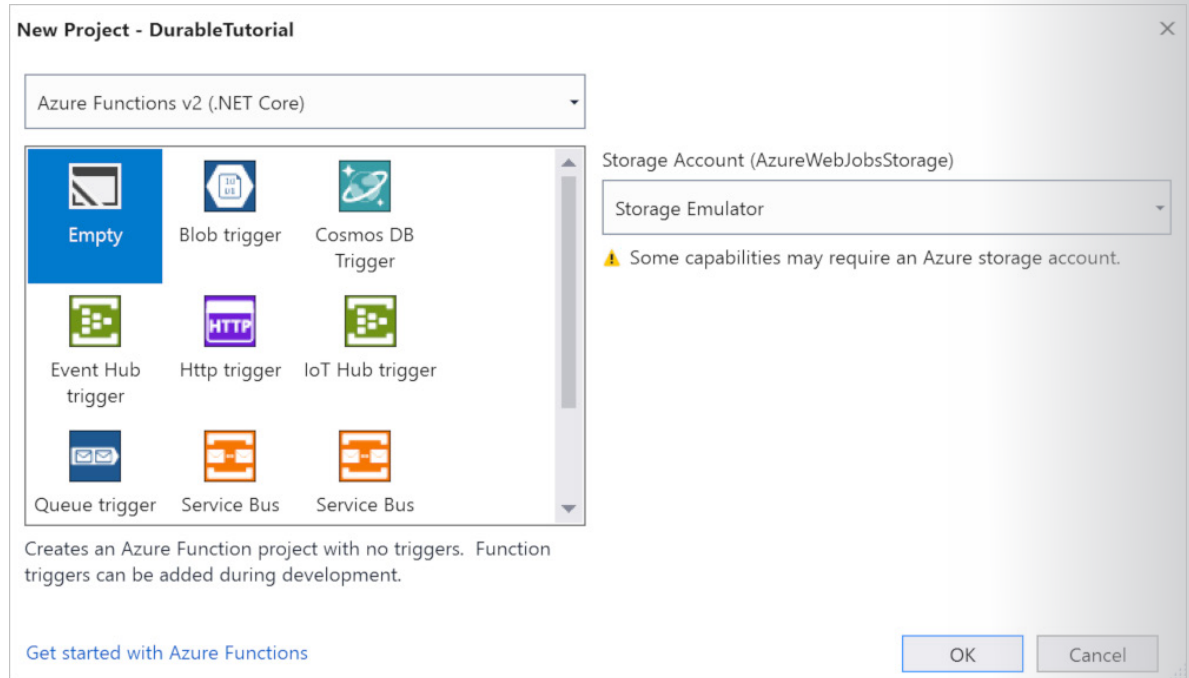


- 3.
4. Use the settings specified in the table that follows the image.

⁵ <https://azure.microsoft.com/downloads/>

⁶ <https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-vs#check-your-tools-version>

⁷ <https://docs.microsoft.com/en-us/azure/storage/common/storage-use-emulator>



5.

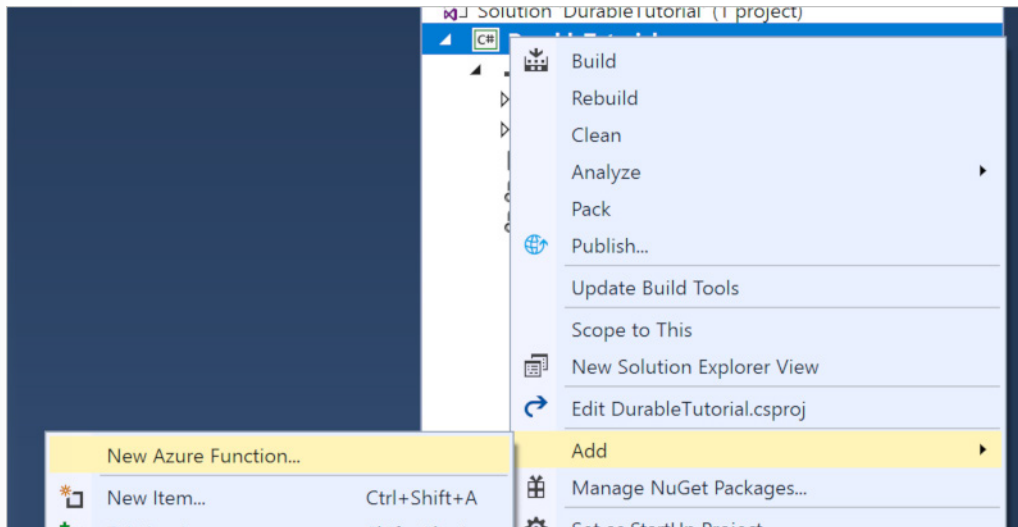
Setting	Suggested value	Description
Version	Azure Functions 2.x (.NET Core)	Creates a function project that uses the version 2.x runtime of Azure Functions which supports .NET Core. Azure Functions 1.x supports the .NET Framework.
Template	Empty	This creates an empty function app.
Storage account	Storage Emulator	A storage account is required for durable function state management.

6. Click **OK** to create an empty function project.

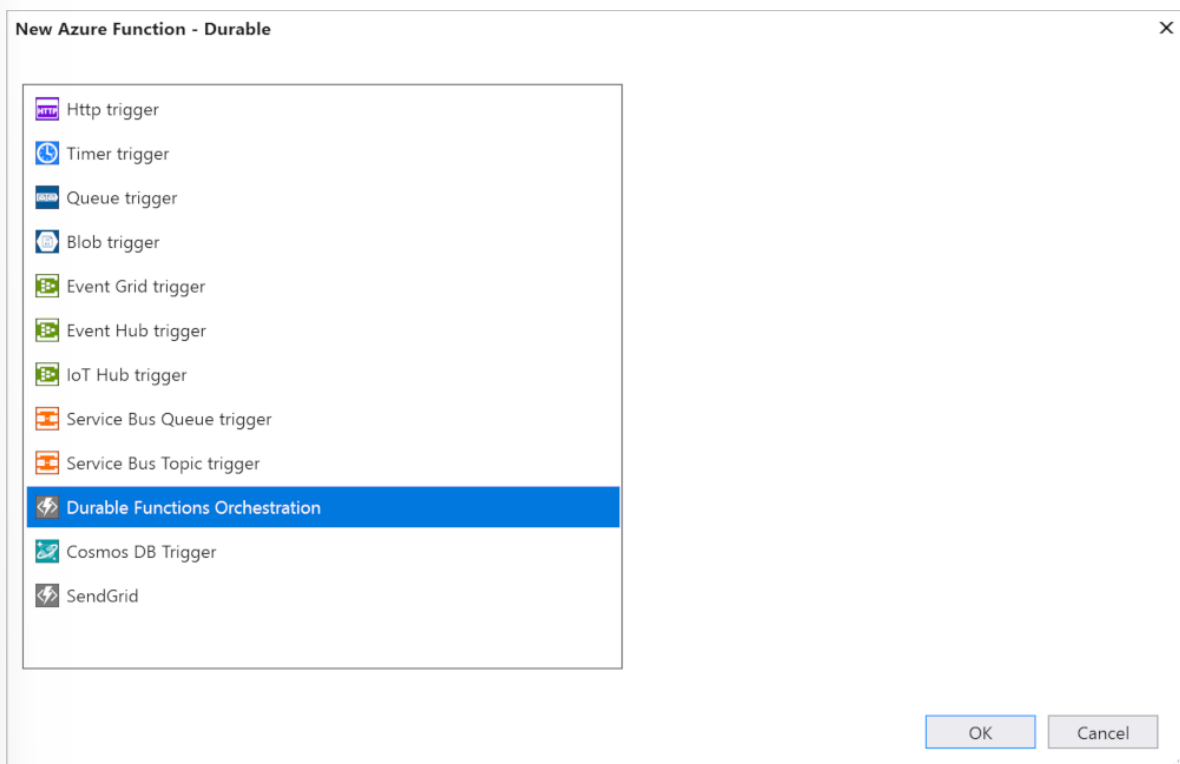
Add functions to the app

Visual Studio creates an empty function app project. It contains the basic configuration files needed for an app, but doesn't yet contain any functions. We'll need to add a durable function template to the project.

1. Right-click the project in Visual Studio and select **Add > New Azure Function**.



- 2.
3. Verify **Azure Function** is selected from the add menu, and give your C# file a name. Press **Add**.
4. Select the **Durable Functions Orchestration** template and click **Ok**.



- 5.
- A new durable function will be added to the app. Open the new file to view the contents. This durable function is a simple function chaining example.
- The `RunOrchestrator` method is associated with the orchestrator function. This function will start, create a list, and add the result of three functions calls to the list. When the three function calls are completed, it will return the list. The function it is calling is the `SayHello` method (default it will be called `"_Hello"`).
 - The `SayHello` function will return a hello.

- The `HttpStart` method describes the function that will start instances of the orchestration. It is associated with an HTTP trigger that will start a new instance of the orchestrator and return back a check status response.

Now that you've created your function project and a durable function, you can test it on your local computer.

Test the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. You are prompted to install these tools the first time you start a function from Visual Studio.

1. To test your function, press **F5**. If prompted, accept the request from Visual Studio to download and install Azure Functions Core (CLI) tools. You may also need to enable a firewall exception so that the tools can handle HTTP requests.
2. Copy the URL of your function from the Azure Functions runtime output.

```
Name: . ExtensionVersion: 1.6.2. SequenceNumber: 1.
[11/8/2018 7:05:31 AM] Host started (2709ms)
[11/8/2018 7:05:31 AM] Job host started
Hosting environment: Production
Content root path: C:\Users\jeffhollan\source\repos\DurableTutorial\DurableTutorial\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...

Http Functions:

    Durable_HttpStart: [GET,POST] http://localhost:7071/api/Durable_HttpStart

[11/8/2018 7:05:38 AM] Host lock lease acquired by instance ID '00000000000000000000000075A5073E'.
```

- 3.
4. Paste the URL for the HTTP request into your browser's address bar and execute the request. The following shows the response in the browser to the local GET request returned by the function:

```
localhost:7071/api/Durable_HttpStart

{"id":"d495cb0ac10d4e13b22729c37e335190","statusQueryGetUri":"http://localhost:7071/runtime/webhooks/durabletask/instances/d495cb0ac10d4e13b22729c37e335190?taskHub=DurableFunctionsHub&connection=Storage&code=/yA1085y3Lsi7cofK1au6X3FZZMKxENddnoQSYszvMPa66uvXtwyYA==","sendEventPostUri":"http://localhost:7071/runtime/webhooks/durabletask/instances/d495cb0ac10d4e13b22729c37e335190/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection=Storage&code=/yA1085y3Lsi7cofK1au6X3FZZMKxENddnoQSYszvMPa66uvXtwyYA==","terminatePostUri":"http://localhost:7071/runtime/webhooks/durabletask/instances/d495cb0ac10d4e13b22729c37e335190/terminate?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code=/yA1085y3Lsi7cofK1au6X3FZZMKxENddnoQSYszvMPa66uvXtwyYA==","rewindPostUri":"http://localhost:7071/runtime/webhooks/durabletask/instances/d495cb0ac10d4e13b22729c37e335190/rewind?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code=/yA1085y3Lsi7cofK1au6X3FZZMKxENddnoQSYszvMPa66uvXtwyYA=="}

```

- 5.
6. The response is the initial result from the HTTP function letting us know the durable orchestration has started successfully. It is not yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.
7. Copy the URL value for `statusQueryGetUri` and pasting it in the browser's address bar and execute the request.
8. The request will query the orchestration instance for the status. You should get an eventual response that looks like the following. This shows us the instance has completed, and includes the outputs or results of the durable function.

```
{
  "instanceId": "d495cb0ac10d4e13b22729c37e335190",
  "runtimeStatus": "Completed",
  "input": null,
```

```
"customStatus": null,
"output": [
  "Hello Tokyo!",
  "Hello Seattle!",
  "Hello London!"
],
"createdTime": "2018-11-08T07:07:40Z",
"lastUpdatedTime": "2018-11-08T07:07:52Z"
}
```

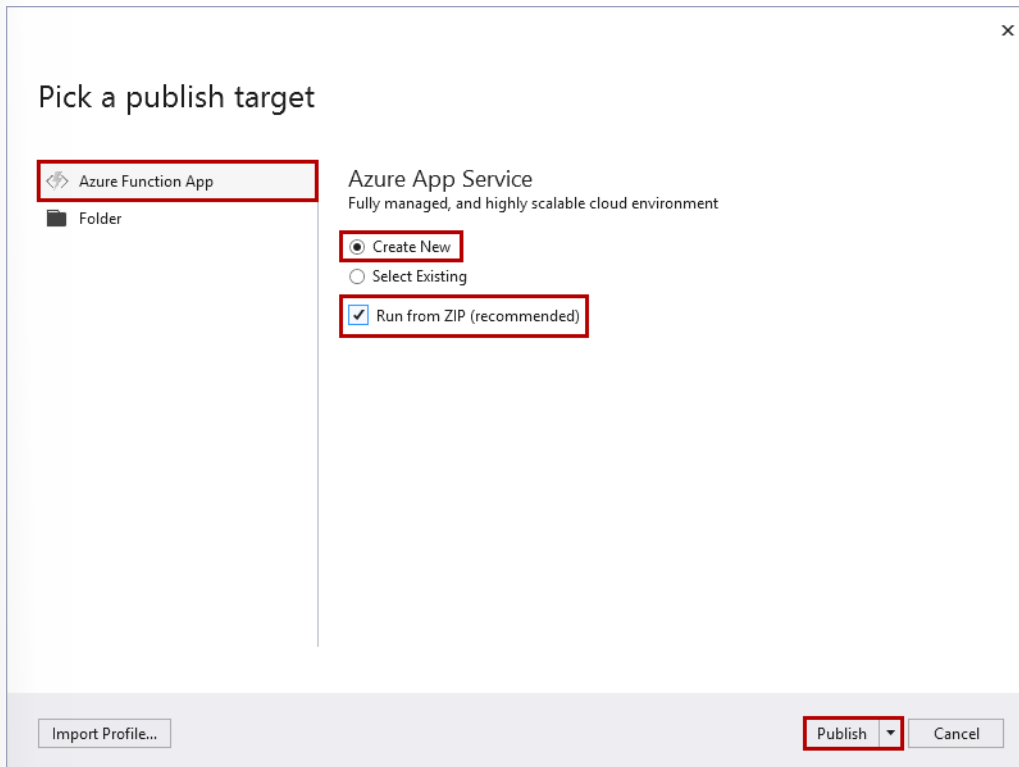
9. To stop debugging, press **Shift + F5**.

After you have verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Publish the project to Azure

You must have a function app in your Azure subscription before you can publish your project. You can create a function app right from Visual Studio.

1. In **Solution Explorer**, right-click the project and select **Publish**.
2. Select **Azure Function App**, choose **Create New**, and then select **Publish**.



- 3.
4. When you enable **Run from Zip**, your function app in Azure is run directly from the deployment package.
5. **Caution:** When you choose **Select Existing**, all files in the existing function app in Azure are overwritten by files from the local project. Only use this option when republishing updates to an existing function app.

6. If you haven't already connected Visual Studio to your Azure account, select **Add an account....**
7. In the **Create App Service** dialog, use the **Hosting** settings as specified in the table below the image:

Create App Service
Host your web and mobile applications, REST APIs, and more in Azure

App Name
myDemoFunctionApp20181016

Subscription
Microsoft Azure Internal Consumption

Resource Group
myDemos* **New...**

Hosting Plan
myDemoPlan20181016* (Central US, Y1) **New...**

Storage Account
mydemofunctionapp2018101* (Central US) **New...**

Export...

Create Cancel

Explore additional Azure services
Create a SQL Database
Create a storage account

Clicking the Create button will create the following Azure resources
Storage Account - mydemofunctionapp2018101
Hosting Plan - myDemoPlan20181016
App Service - myDemoFunctionApp20181016

8.

Setting	Suggested value	Description
App Name	Globally unique name	Name that uniquely identifies your new function app.
Subscription	Choose your subscription	The Azure subscription to use.
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose New to create a new resource group.
App Service Plan	Consumption plan	Make sure to choose the Consumption under Size after you click New to create a serverless plan. Also, choose a Location in a region near you or near other services your functions access. When you run in a plan other than Consumption , you must manage the scaling of your function app.

Setting	Suggested value	Description
Storage Account	General purpose storage account	An Azure storage account is required by the Functions runtime. Click New to create a general purpose storage account. You can also use an existing account that meets the storage account requirements.

- Click **Create** to create a function app and related resources in Azure with these settings and deploy your function project code.
- After the deployment is complete, make a note of the **Site URL** value, which is the address of your function app in Azure.

Publish

Publish your app to Azure or another host. [Learn more](#)

myDemoFunctionApp20181016 - Zip Deploy [Publish](#)

[New Profile...](#) [Actions ▼](#)

Site URL	https://mydemofunctiona...	Manage Application Settings...
Configuration	Release	Manage Profile Settings...
Delete existing files	True	
Username	\$myDemoFunctionApp20181016	
Password	*****	

11.

Test your function in Azure

- Copy the base URL of the function app from the Publish profile page. Replace the `localhost:port` portion of the URL you used when testing the function locally with the new base URL.
- The URL that calls your durable function HTTP trigger should be in the following format:

```
http://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>_HttpStart
```

- Paste this new URL for the HTTP request into your browser's address bar. You should get the same status response as before when using the published app.

Fan-out/fan-in Durable Function example

Fan-out/fan-in refers to the pattern of executing multiple functions concurrently and then performing some aggregation on the results. This lesson explains a sample that uses Durable Functions to implement a fan-out/fan-in scenario. The sample is a durable function that backs up all or some of an app's site content into Azure Storage.

Scenario overview

In this sample, the functions upload all files under a specified directory recursively into blob storage. They also count the total number of bytes that were uploaded.

It's possible to write a single function that takes care of everything. The main problem you would run into is scalability. A single function execution can only run on a single VM, so the throughput will be limited by the throughput of that single VM. Another problem is reliability. If there's a failure midway through, or if the entire process takes more than 5 minutes, the backup could fail in a partially-completed state. It would then need to be restarted.

A more robust approach would be to write two regular functions: one would enumerate the files and add the file names to a queue, and another would read from the queue and upload the files to blob storage. This is better in terms of throughput and reliability, but it requires you to provision and manage a queue. More importantly, significant complexity is introduced in terms of state management and coordination if you want to do anything more, like report the total number of bytes uploaded.

A Durable Functions approach gives you all of the mentioned benefits with very low overhead.

The functions

This lesson explains the following functions in the sample app:

- `E2_BackupSiteContent`
- `E2_GetFileList`
- `E2_CopyFileToBlob`

The following sections explain the configuration and code that are used for C# scripting. The code for Visual Studio development is shown at the end of the lesson.

The cloud backup orchestration (Visual Studio Code and Azure portal sample code)

The `E2_BackupSiteContent` function uses the standard `function.json` for orchestrator functions.

```
{
  "bindings": [
    {
      "name": "backupContext",
      "type": "orchestrationTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

Here is the code that implements the orchestrator function:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task<long> Run(DurableOrchestrationContext backupCon-
text)
{
    string rootDirectory = Environment.ExpandEnvironmentVariables(backup-
Context.GetInput<string>() ?? "");
```



```

        if (string.IsNullOrEmpty(rootDirectory))
        {
            throw new ArgumentException("A directory path is required as an
input.");
        }

        if (!Directory.Exists(rootDirectory))
        {
            throw new DirectoryNotFoundException($"Could not find a directory
named '{rootDirectory}'.");
        }

        string[] files = await backupContext.CallActivityAsync<string[]>(
            "E2_GetFileList",
            rootDirectory);

        var tasks = new Task<long>[files.Length];
        for (int i = 0; i < files.Length; i++)
        {
            tasks[i] = backupContext.CallActivityAsync<long>(
                "E2_CopyFileToBlob",
                files[i]);
        }

        await Task.WhenAll(tasks);

        long totalBytes = tasks.Sum(t => t.Result);
        return totalBytes;
    }

```

JavaScript (Functions v2 only)

```

const df = require("durable-functions");

module.exports = df.orchestrator(function*(context) {
    const rootDirectory = context.df.getInput();
    if (!rootDirectory) {
        throw new Error("A directory path is required as an input.");
    }

    const files = yield context.df.callActivity("E2_GetFileList", rootDirec-
tory);

    // Backup Files and save Promises into array
    const tasks = [];
    for (const file of files) {
        tasks.push(context.df.callActivity("E2_CopyFileToBlob", file));
    }

    // wait for all the Backup Files Activities to complete, sum total

```

```

bytes
    const results = yield context.df.Task.all(tasks);
    const totalBytes = results.reduce((prev, curr) => prev + curr, 0);

    // return results;
    return totalBytes;
});

```

This orchestrator function essentially does the following:

1. Takes a `rootDirectory` value as an input parameter.
2. Calls a function to get a recursive list of files under `rootDirectory`.
3. Makes multiple parallel function calls to upload each file into Azure Blob Storage.
4. Waits for all uploads to complete.
5. Returns the sum total bytes that were uploaded to Azure Blob Storage.

Notice the `await Task.WhenAll(tasks);` (C#) and `yield context.df.Task.all(tasks);` (JS) line. All the calls to the `E2_CopyFileToBlob` function were *not* awaited. This is intentional to allow them to run in parallel. When we pass this array of tasks to `Task.WhenAll`, we get back a task that won't complete *until all the copy operations have completed*. If you're familiar with the Task Parallel Library (TPL) in .NET, then this is not new to you. The difference is that these tasks could be running on multiple VMs concurrently, and the Durable Functions extension ensures that the end-to-end execution is resilient to process recycling.

Tasks are very similar to the JavaScript concept of promises. However, `Promise.all` has some differences from `Task.WhenAll`. The concept of `Task.WhenAll` has been ported over as part of the `durable-functions` JavaScript module and is exclusive to it.

After awaiting from `Task.WhenAll` (or yielding from `context.df.Task.all`), we know that all function calls have completed and have returned values back to us. Each call to `E2_CopyFileToBlob` returns the number of bytes uploaded, so calculating the sum total byte count is a matter of adding all those return values together.

Helper activity functions

The helper activity functions, as with other samples, are just regular functions that use the `activityTrigger` trigger binding. For example, the `function.json` file for `E2_GetFileList` looks like the following:

```

{
  "bindings": [
    {
      "name": "rootDirectory",
      "type": "activityTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}

```

And here is the implementation:

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Microsoft.Extensions.Logging"

public static string[] Run(string rootDirectory, ILogger log)
{
    string[] files = Directory.GetFiles(rootDirectory, "*", SearchOption.
AllDirectories);
    log.LogInformation($"Found {files.Length} file(s) under {rootDirecto-
ry}.");

    return files;
}
```

JavaScript (Functions v2 only)

```
const readdirp = require("readdirp");

module.exports = function (context, rootDirectory) {
    context.log(`Searching for files under '${rootDirectory}'...`);
    const allFilePaths = [];

    readdirp(
        {root: rootDirectory, entryType: 'all'},
        function (fileInfo) {
            if (!fileInfo.stat.isDirectory()) {
                allFilePaths.push(fileInfo.fullPath);
            }
        },
        function (err, res) {
            if (err) {
                throw err;
            }

            context.log(`Found ${allFilePaths.length} under ${rootDirecto-
ry}.`);

            context.done(null, allFilePaths);
        }
    );
};
```

The JavaScript implementation of `E2_GetFileList` uses the `readdirp` module to recursively read the directory structure.

Note: You might be wondering why you couldn't just put this code directly into the orchestrator function. You could, but this would break one of the fundamental rules of orchestrator functions, which is that they should never do I/O, including local file system access.

The `function.json` file for `E2_CopyFileToBlob` is similarly simple:

```
{
  "bindings": [
    {
      "name": "filePath",
      "type": "activityTrigger",
      "direction": "in"
    }
  ],
  "disabled": false
}
```

The C# implementation is also pretty straightforward. It happens to use some advanced features of Azure Functions bindings (that is, the use of the `Binder` parameter), but you don't need to worry about those details for the purpose of this walkthrough.

C#

```
#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"
#r "Microsoft.Azure.WebJobs.Extensions.Storage"
#r "Microsoft.Extensions.Logging"
#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Blob;

public static async Task<long> Run(
    string filePath,
    Binder binder,
    ILogger log)
{
    long byteCount = new FileInfo(filePath).Length;

    // strip the drive letter prefix and convert to forward slashes
    string blobPath = filePath
        .Substring(Path.GetPathRoot(filePath).Length)
        .Replace('\\', '/');
    string outputLocation = $"backups/{blobPath}";

    log.LogInformation($"Copying '{filePath}' to '{outputLocation}'. Total
bytes = {byteCount}.");

    // copy the file contents into a blob
    using (Stream source = File.Open(filePath, FileMode.Open, FileAccess.
Read, FileShare.ReadWrite))
    using (Stream destination = await binder.BindAsync<CloudBlobStream>(
        new BlobAttribute(outputLocation)))
    {
        await source.CopyToAsync(destination);
    }

    return byteCount;
}
```

```

}
```

JavaScript (Functions v2 only)

The JavaScript implementation does not have access to the `Binder` feature of Azure Functions, so the Azure Storage SDK for Node takes its place.

```

const fs = require("fs");
const path = require("path");
const storage = require("azure-storage");

module.exports = function (context, filePath) {
  const container = "backups";
  const root = path.parse(filePath).root;
  const blobPath = filePath
    .substring(root.length)
    .replace("\\", "/");
  const outputLocation = `backups/${blobPath}`;
  const blobService = storage.createBlobService(process.env['AzureWebJobsStorage']);

  blobService.createContainerIfNotExists(container, (error) => {
    if (error) {
      throw error;
    }
  })

  fs.stat(filePath, function (error, stats) {
    if (error) {
      throw error;
    }
    context.log(`Copying '${filePath}' to '${outputLocation}'. Total
bytes = ${stats.size}.`);

    const readStream = fs.createReadStream(filePath);

    blobService.createBlockBlobFromStream(container, blobPath,
readStream, stats.size, function (error) {
      if (error) {
        throw error;
      }
      context.done(null, stats.size);
    });
  });
};
```

The implementation loads the file from disk and asynchronously streams the contents into a blob of the same name in the “backups” container. The return value is the number of bytes copied to storage, that is then used by the orchestrator function to compute the aggregate sum.

Note: This is a perfect example of moving I/O operations into an `activityTrigger` function. Not only can the work be distributed across many different VMs, but you also get the benefits of checkpointing the progress. If the host process gets terminated for any reason, you know which uploads have already completed.

Run the sample

You can start the orchestration by sending the following HTTP POST request.

```
POST http://{host}/orchestrators/E2_BackupSiteContent
Content-Type: application/json
Content-Length: 20

"D:\\home\\LogFiles"
```

Note: The `HttpStart` function that you are invoking only works with JSON-formatted content. For this reason, the `Content-Type: application/json` header is required and the directory path is encoded as a JSON string. Moreover, HTTP snippet assumes there is an entry in the `host.json` file which removes the default `api/` prefix from all HTTP trigger functions URLs. You can find the markup for this configuration in the `host.json` file in the samples.

This HTTP request triggers the `E2_BackupSiteContent` orchestrator and passes the string `D:\\home\\LogFiles` as a parameter. The response provides a link to get the status of the backup operation:

```
HTTP/1.1 202 Accepted
Content-Length: 719
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/
b4e9bdcc435d460f8dc008115ff0a8a9?taskHub=DurableFunction-
sHub&connection=Storage&code={systemKey}

(...trimmed...)
```

Depending on how many log files you have in your function app, this operation could take several minutes to complete. You can get the latest status by querying the URL in the `Location` header of the previous HTTP 202 response.

```
GET http://{host}/admin/extensions/DurableTaskExtension/instances/b4e9bdcc-
c435d460f8dc008115ff0a8a9?taskHub=DurableFunctionsHub&con-
nection=Storage&code={systemKey}

HTTP/1.1 202 Accepted
Content-Length: 148
Content-Type: application/json; charset=utf-8
Location: http://{host}/admin/extensions/DurableTaskExtension/instances/
b4e9bdcc435d460f8dc008115ff0a8a9?taskHub=DurableFunction-
sHub&connection=Storage&code={systemKey}

{"runtimeStatus":"Running","input":"D:\\home\\LogFiles","output":null,"cre-
atedTime":"2017-06-29T18:50:55Z","lastUpdatedTime":"2017-06-29T18:51:16Z"}
```

In this case, the function is still running. You are able to see the input that was saved into the orchestrator state and the last updated time. You can continue to use the `Location` header values to poll for completion. When the status is “Completed”, you see an HTTP response value similar to the following:

```
HTTP/1.1 200 OK
Content-Length: 152
Content-Type: application/json; charset=utf-8

{"runtimeStatus":"Completed","input":"D:\\home\\LogFiles","output":452071,"createdTime":"2017-06-29T18:50:55Z","lastUpdatedTime":"2017-06-29T18:51:26Z"}
```

Now you can see that the orchestration is complete and approximately how much time it took to complete. You also see a value for the `output` field, which indicates that around 450 KB of logs were uploaded.

Visual Studio sample code

Here is the orchestration as a single C# file in a Visual Studio project:

```
// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for
// license information.

using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;
using Microsoft.WindowsAzure.Storage.Blob;

namespace VSSample
{
    public static class BackupSiteContent
    {
        [FunctionName("E2_BackupSiteContent")]
        public static async Task<long> Run(
            [OrchestrationTrigger] DurableOrchestrationContext backupContext)
        {
            string rootDirectory = backupContext.GetInput<string>()?.Trim();

            if (string.IsNullOrEmpty(rootDirectory))
            {
                rootDirectory = Directory.GetParent(typeof(BackupSiteContent).Assembly.Location).FullName;
            }

            string[] files = await backupContext.CallActivityAsync<string[]>(
                "E2_GetFileList",
                rootDirectory);
        }
    }
}
```

```

var tasks = new Task<long>[files.Length];
for (int i = 0; i < files.Length; i++)
{
    tasks[i] = backupContext.CallActivityAsync<long>(
        "E2_CopyFileToBlob",
        files[i]);
}

await Task.WhenAll(tasks);

long totalBytes = tasks.Sum(t => t.Result);
return totalBytes;
}

[FunctionName("E2_GetFileList")]
public static string[] GetFileList(
    [ActivityTrigger] string rootDirectory,
    ILogger log)
{
    log.LogInformation($"Searching for files under '{rootDirectory}'...");
    string[] files = Directory.GetFiles(rootDirectory, "*",
        SearchOption.AllDirectories);
    log.LogInformation($"Found {files.Length} file(s) under {rootDirectory}.");

    return files;
}

[FunctionName("E2_CopyFileToBlob")]
public static async Task<long> CopyFileToBlob(
    [ActivityTrigger] string filePath,
    Binder binder,
    ILogger log)
{
    long byteCount = new FileInfo(filePath).Length;

    // strip the drive letter prefix and convert to forward slashes
    string blobPath = filePath
        .Substring(Path.GetPathRoot(filePath).Length)
        .Replace('\\', '/');
    string outputLocation = $"backups/{blobPath}";

    log.LogInformation($"Copying '{filePath}' to '{outputLocation}'. Total bytes = {byteCount}.");

    // copy the file contents into a blob
    using (Stream source = File.Open(filePath, FileMode.Open, FileAccess.Read, FileShare.Read))
    using (Stream destination = await binder.BindAsync<CloudBlob-

```



```
Stream>(
    new BlobAttribute(outputLocation, FileAccess.Write))
{
    await source.CopyToAsync(destination);
}

return byteCount;
}
}
```

Review questions

Module 4 review questions

Azure Functions consumption plans

Azure Functions run within a service plan. What service plans support Azure Functions and how do they differ?

> Click to see suggested answer

Azure Functions runs in two different modes: Consumption plan and Azure App Service plan. When you're using a Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. This serverless plan scales automatically, and you're charged for compute resources only when your functions are running. On a Consumption plan, a function execution times out after a configurable period of time.

In the dedicated App Service plan, your function apps run on dedicated VMs on Basic, Standard, Premium, and Isolated SKUs, which is the same as other App Service apps. Dedicated VMs are allocated to your function app, which means the functions host can be always running. App Service plans support Linux.

Durable Functions

The primary use case for Durable Functions is simplifying complex, stateful coordination problems in serverless applications. Function chaining and Fan-in/fan-out are two of the typical application patterns that can benefit from Durable Functions.

Can you briefly describe how they operate?

> Click to see suggested answer

- Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function.
- Fan-out/fan-in refers to the pattern of executing multiple functions in parallel, and then waiting for all to finish. Often some aggregation work is done on results returned from the functions.