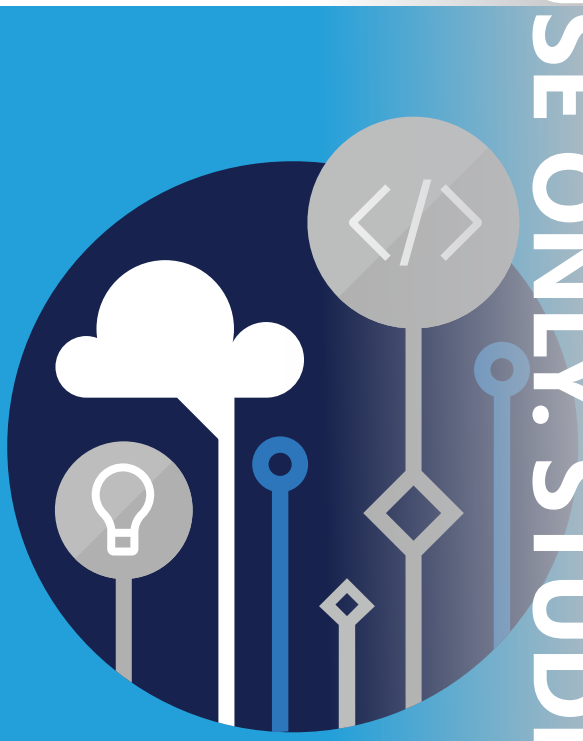
Microsoft

Microsoft Official Course



AZ-300T04

Creating and Deploying Apps

AZ-300T04
Creating and Deploying Apps

Contents

Module 0 Start Here

Welcome to Creating and Deploying Apps

Welcome to Creating and Deploying Apps

Course Overview: Welcome to Creating and Deploying Apps

Welcome to *Creating and Deploying Apps* (AZ-300t4). This course is part of a series of six courses to help students prepare for Microsoft's Azure Solutions Architect technical certification exam AZ-300: Microsoft Azure Architect Technologies. These courses are designed for IT professionals and developers with experience and knowledge across various aspects of IT operations, including networking, virtualization, identity, security, business continuity, disaster recovery, data management, budgeting, and governance.

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

You will see how Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Service Fabric also addresses the significant challenges in developing and managing cloud native applications. Using Azure Service Fabric, developers and administrators can avoid complex infrastructure problems and focus on implementing mission-critical, demanding workloads that are scalable, reliable, and manageable. Service Fabric represents the next-generation platform for building and managing enterprise-class, tier-1, cloud-scale applications running in containers.

You'll see how Azure Kubernetes Service (AKS) makes it simple to deploy a managed Kubernetes cluster in Azure. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you.

Lastly, you will get an overview of how Azure Functions are used as a solution for easily running small pieces of code, or "functions," in the cloud. 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.

The course outline is as follows:

Module 1 - Creating Web Applications using PaaS

This module provides and overview of Azure App Service Web Apps for hosting web applications, REST APIs, and a mobile back end.

Topics include the following:

- Using shell commands to create an App Service Web App
- Creating Background Tasks
- Using Swagger to document an API

Also, you'll see how Logic Apps assists in building solutions that integrate apps, data, systems, and services across enterprises or organizations by automating tasks and business processes as workflows.

Module 1 Online Lab - Implementing and Managing Application Services (Topic: Implementing Azure Logic Apps)

Module 2 - Creating Apps and Services Running on Service Fabric

This module provides an overview of Azure Service Fabric as a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. This module also addresses the challenges in developing and managing cloud native applications.

Additional topics include:

- Creating a reliable service
- Creating a Reliable Actors app
- Working with Reliable collections

Module 3 - Using Azure Kubernetes Service. This module focuses on the Azure Kubernetes Service (AKS) for deploying and managing a Kubernetes cluster in Azure. Topics include how to reduce operational overhead of managing Kubernetes by offloading much of that responsibility to Azure, such as health monitoring and maintenance.

Additional topics include:

- Azure Container Registry
- Azure Container Instances

Module 4 - Understanding 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.

This module includes:

- Azure Functions scale and hosting concepts
- Develop, Test, and Publish Azure Functions using Visual Studio
- Implement Durable Functions

What You'll Learn:

- Use shell commands to create an App Service Web App
- Create Background Tasks
- Use Swagger to document an API
- Create a reliable service
- Create a Reliable Actors app
- Hands-on with Reliable collections
- Understand the Azure Container Registry
- Use Azure Container instances

Prerequisites:

Successful Cloud Solutions Architects begin this role with practical experience with operating systems, virtualization, cloud infrastructure, storage structures, billing, and networking.

Module 1 Module Creating Web Apps using PaaS

Introduction to Web Apps

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

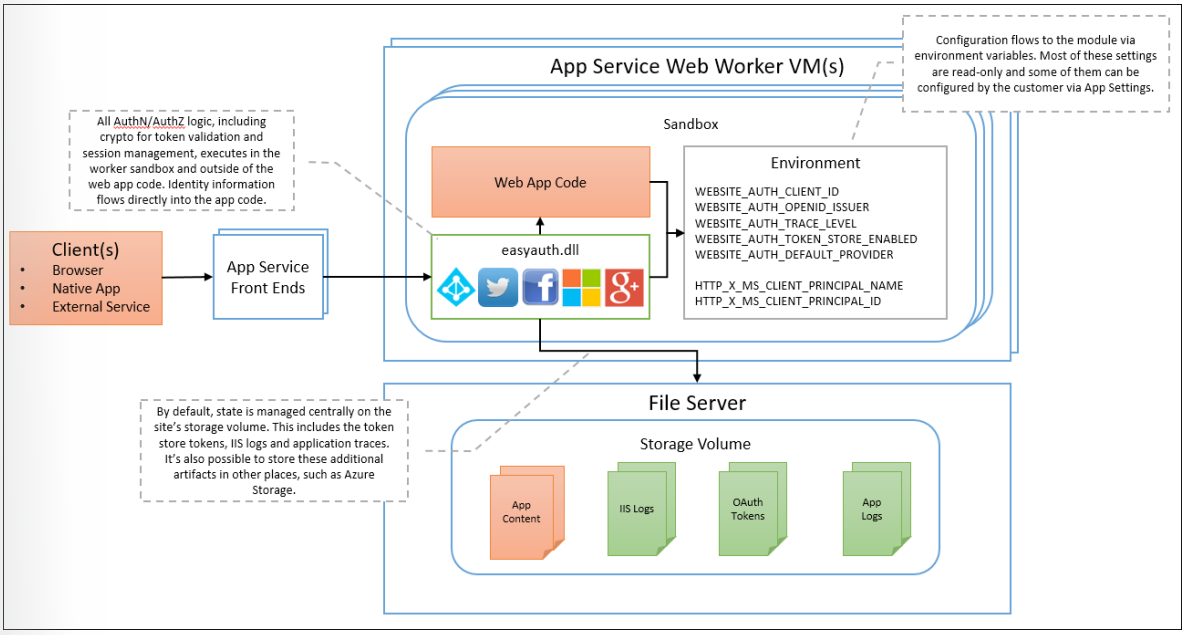
Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your web app, API, and mobile back end, and also Azure Functions.

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	`/.auth/login/aad`
Microsoft Account	`/.auth/login/microsoftaccount`
Facebook	`/.auth/login/facebook`
Google	`/.auth/login/google`
Twitter	`/.auth/login/twitter`

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 `/.auth/login/<provider>`.	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 `/.auth/login/<provider>/callback`.	Client code posts token from provider to `/.auth/login/<provider>` 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.

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**[3].

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.

3 https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-routing-methods

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***[4] 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.

4 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***[5] page and review the available options for ASEs.

5 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**[6].

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 installit on macOS, Linux, or Windows. Read the **Get Started**[7] article to begin using the CLI. For information about the latest release, see the **release notes**[8].

A detailed **reference**[9] 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.

6 https://docs.microsoft.com/en-us/azure/cloud-shell/features#tools
7 https://docs.microsoft.com/en-us/cli/azure/get-started-with-azure-cli?view=azure-cli-latest
8 https://docs.microsoft.com/en-us/cli/azure/release-notes-azure-cli?view=azure-cli-latest
9 https://docs.microsoft.com/en-us/cli/azure/reference-index

Use the Cloud Shell to run the Azure PowerShell in your browser, or **install**[10] it on own computer. Then read the **Get Started**[11] article to begin using it. For information about the latest release, see the **release notes**[12].

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
`az group create`	Creates a resource group in which all resources are stored.
`az appservice plan create`	Creates an App Service plan.
`az webapp create`	Creates an Azure web app.
`az webapp deployment source config`	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**[13] 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
```

10 https://docs.microsoft.com/en-us/powershell/azure/install-azurerm-ps?view=azurermps-6.5.0
11 https://docs.microsoft.com/en-us/powershell/azure/get-started-azureps?view=azurermps-6.5.0
12 https://docs.microsoft.com/en-us/powershell/azure/release-notes-azureps?view=azurermps-6.5.0
13 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**[14] 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.
```

14 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.

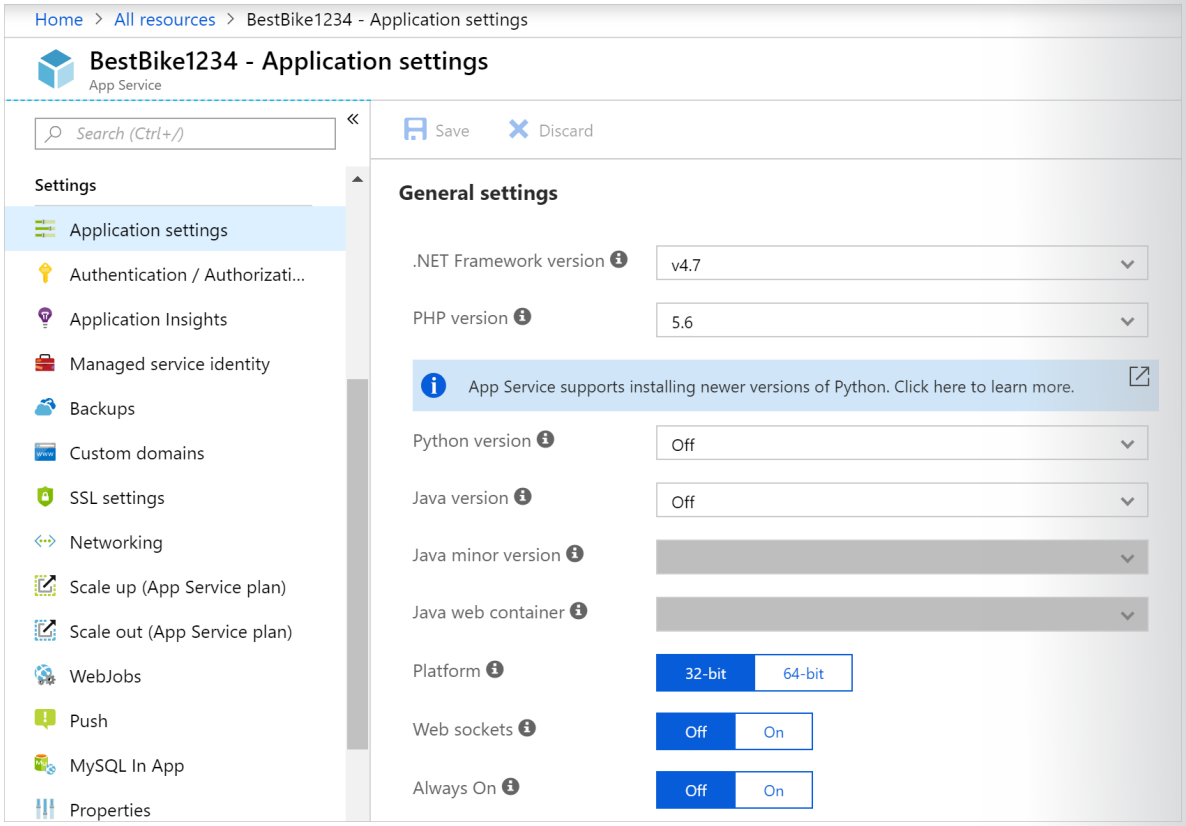
```
Remove-AzureRmResourceGroup -Name myResourceGroup -Force
```

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:

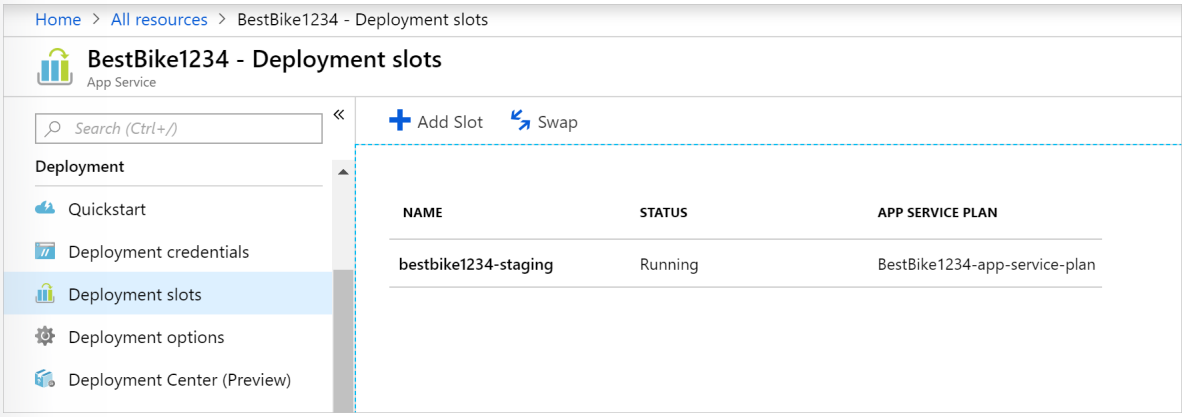
- **App name:** The name of the web app.
- **Subscription:** A valid and active subscription.
- **Resource group:** A valid resource group. The sections below explain in detail what a resource group is.
- **OS:** The operating system. The options are: Windows, Linux, and Docker containers. On Windows, 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

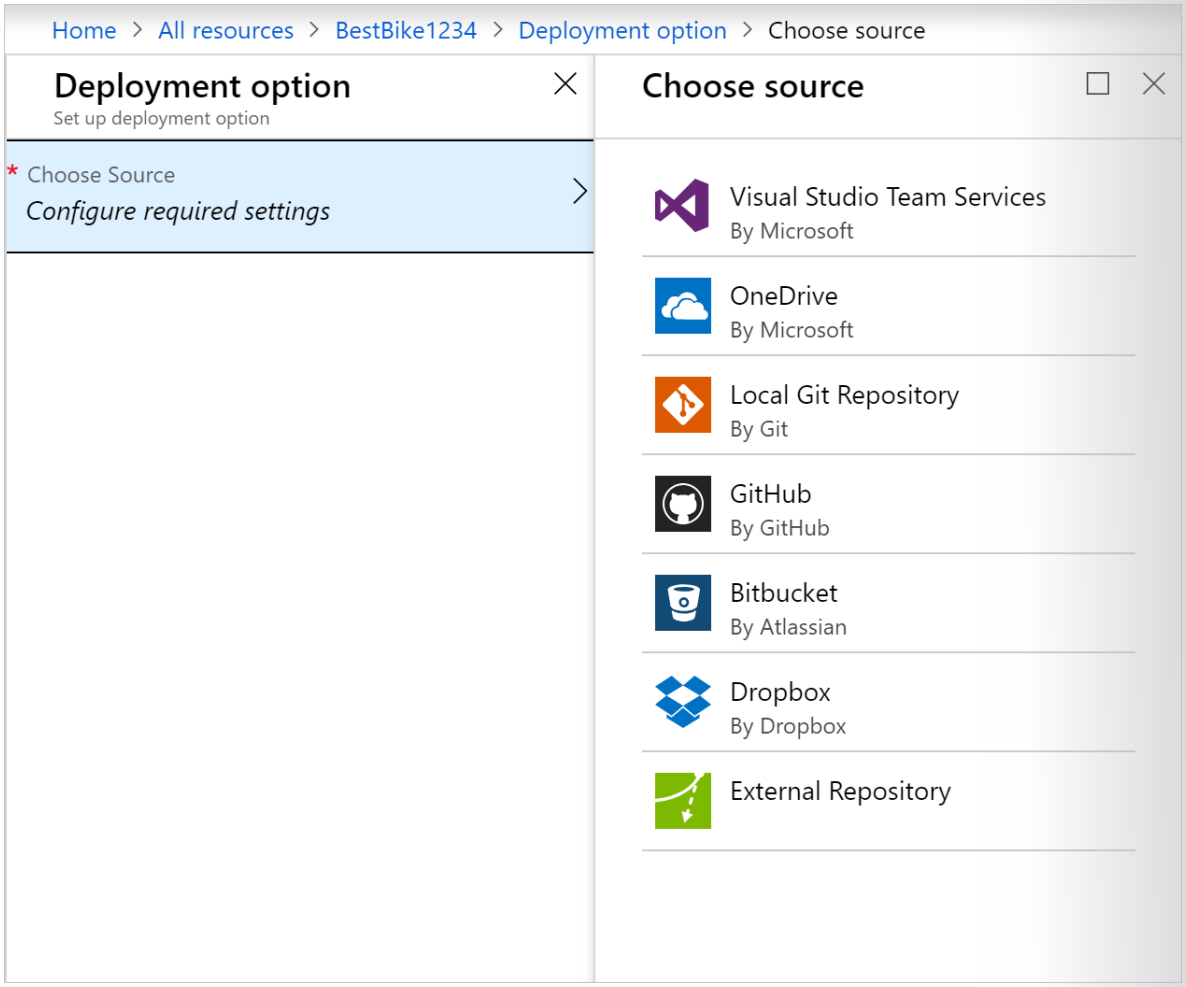
Using the Azure portal, you can easily add **deployment slots** to an App Service web app. For instance, you can create a **staging** deployment slot where you can push your code to test on Azure. Once you are

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.



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

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**[15].

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:

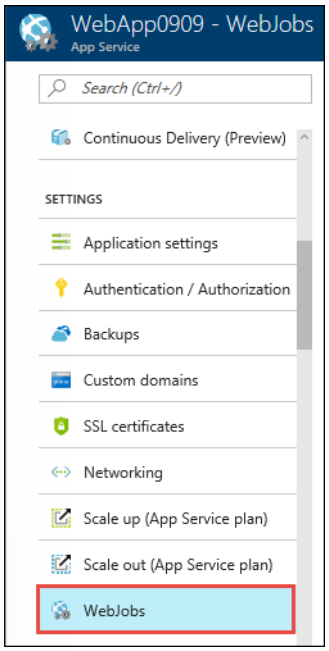
- .cmd, .bat, .exe (using Windows cmd)
- .ps1 (using PowerShell)
- .sh (using Bash)
- .php (using PHP)
- .py (using Python)
- .js (using Node.js)
- .jar (using Java)

Creating a continuous WebJob

1. In the **Azure portal**[16], go to the **App Service** page of your App Service web app, API app, or mobile app.

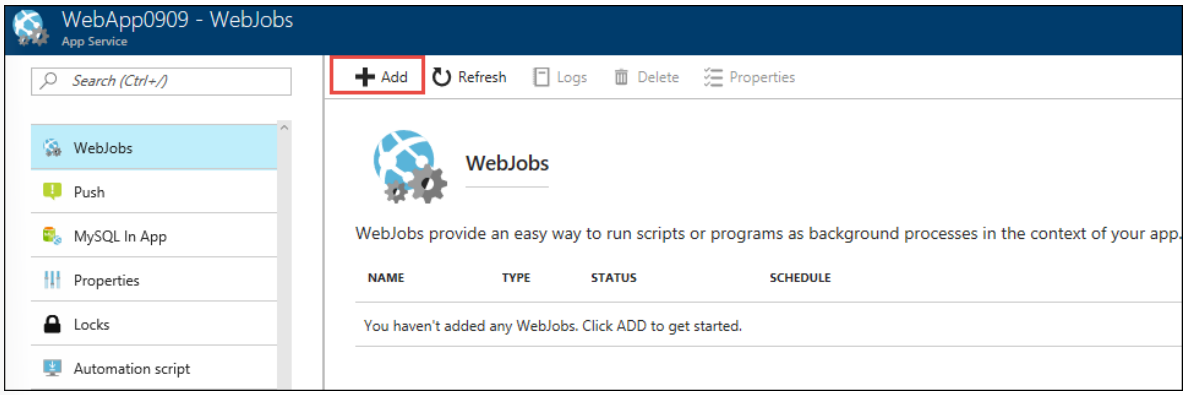
15 https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs
16 https://portal.azure.com/

2. Select **WebJobs**.



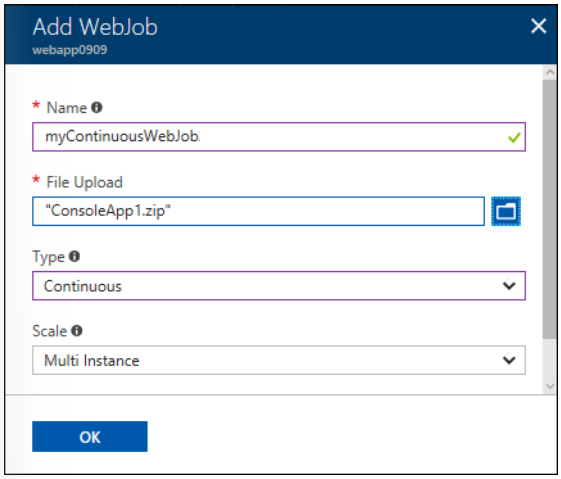
3.

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



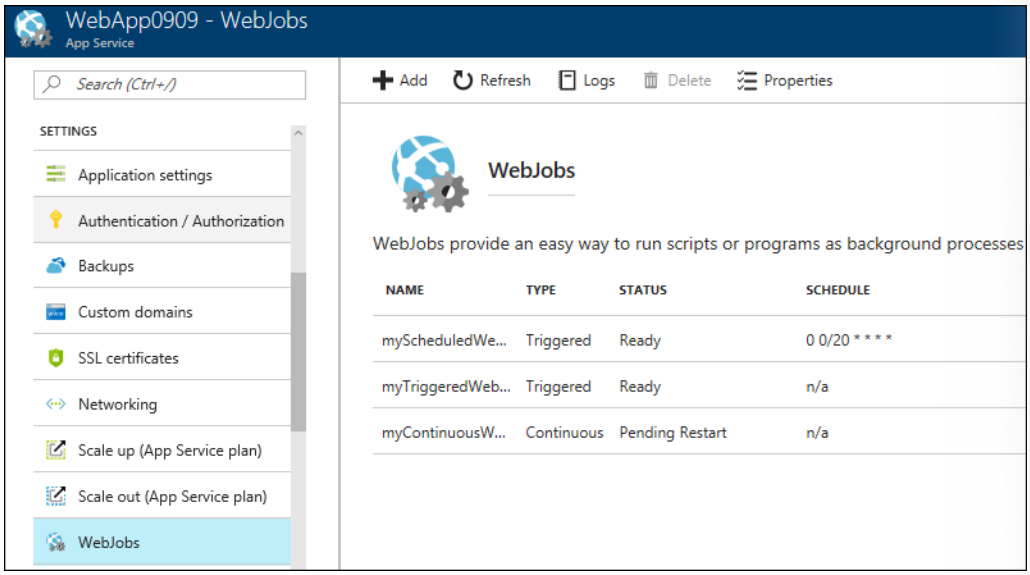
5.

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 *Overview of Webjobs* section of this course.
Type	Continuous	Choose the type of of WebJob you want to create, this example uses *continuous*
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.

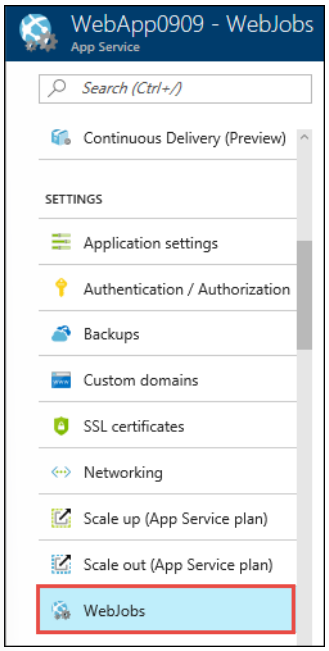
8. Click **OK**.
9. The new WebJob appears on the **WebJobs** page.
10. 

Creating a manually triggered WebJob

1. In the **Azure portal**[17], go to the **App Service** page of your App Service web app, API app, or mobile app.

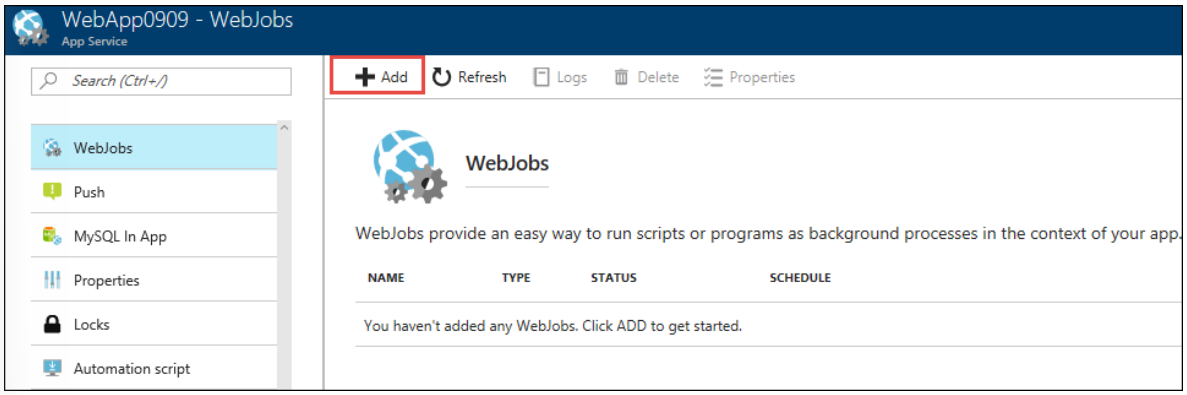
17 https://portal.azure.com/

2. Select **WebJobs**.



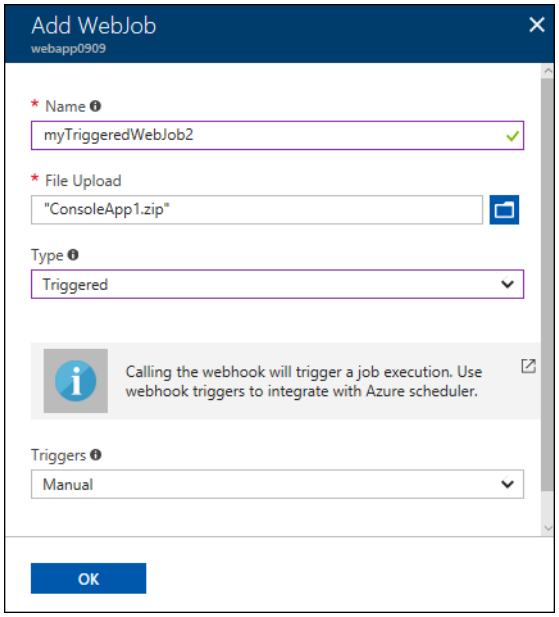
3.

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



5.

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

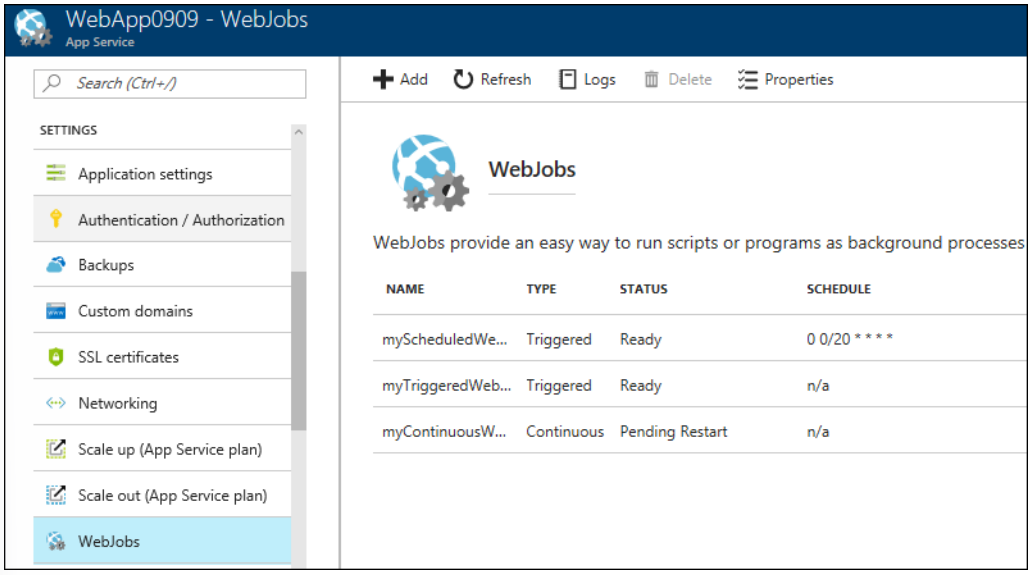


7.

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 *Overview of Webjobs* section of this course.
Type	Triggered	Choose the type of of WebJob you want to create, this example uses *Triggered*.
Triggers	Manual	

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

10.



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:
C#

```
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)**[18].

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

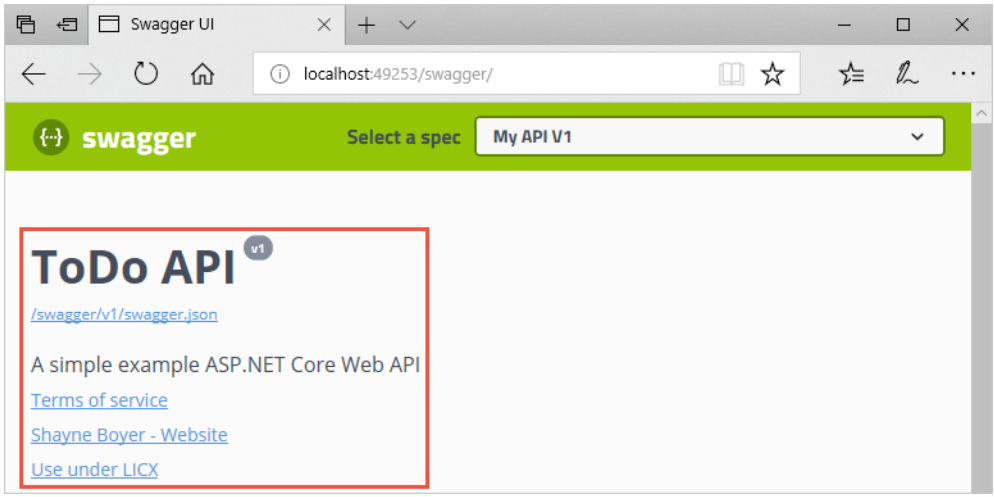
The configuration action passed to the `AddSwaggerGen` method adds information such as the author, license, and description:

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

18 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:

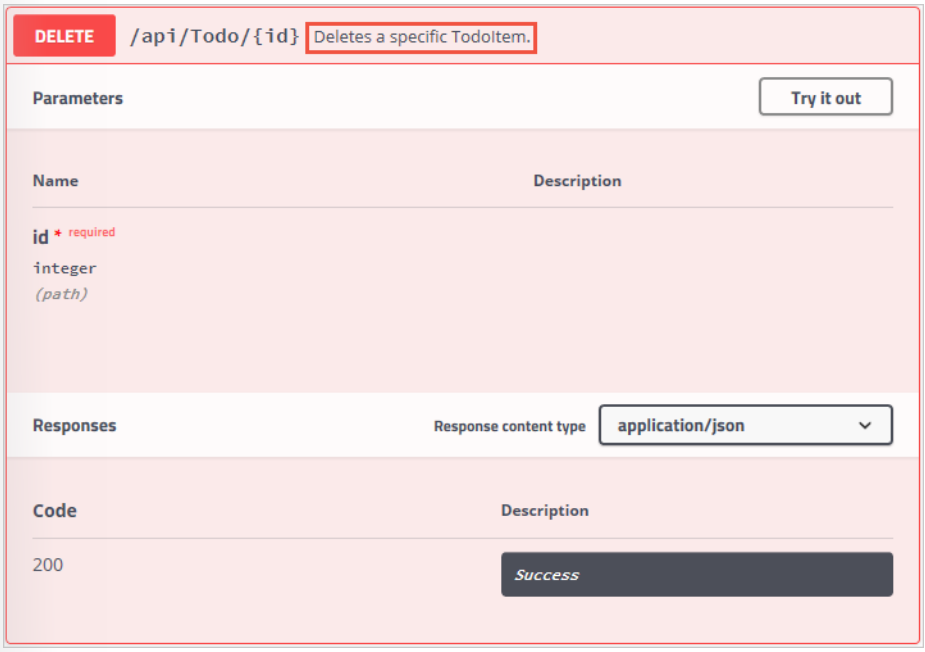
```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);

    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:



The UI is driven by the generated JSON schema:

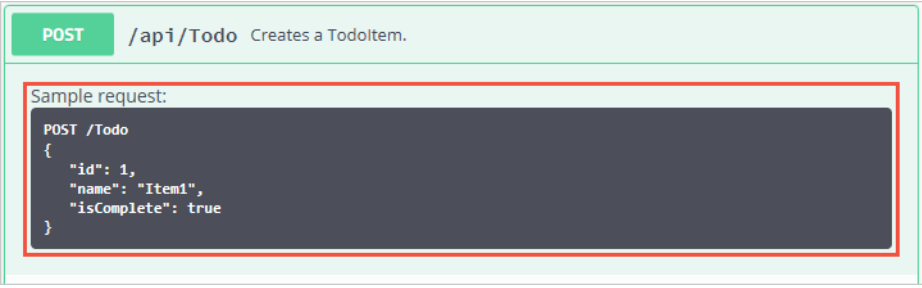
```
"delete": {
    "tags": [
        "Todo"
    ],
    "summary": "Deletes a specific TodoItem.",
    "operationId": "ApiTodoByIdDelete",
    "consumes": [],
    "produces": [],
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "",
            "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; }
    }
}
```

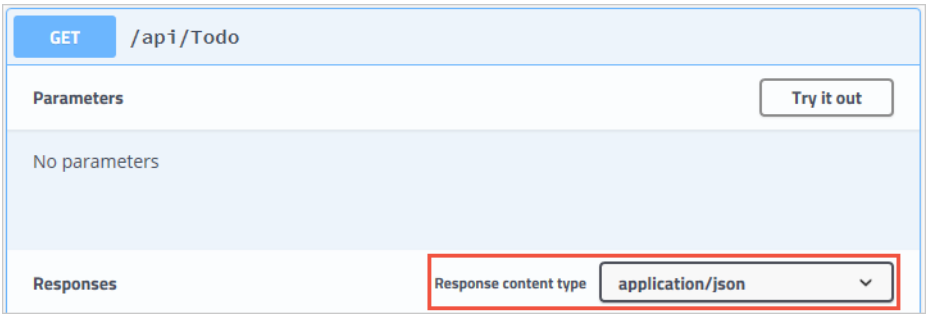
The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

```
"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:



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.

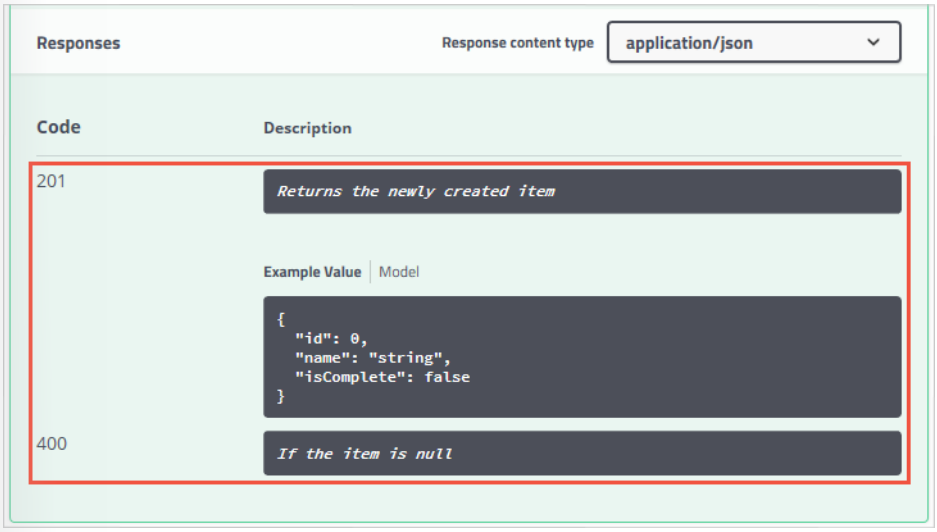
The `Create` action returns an HTTP 201 status code on success. An HTTP 400 status code is returned when the posted request body is null. Without proper documentation in the Swagger UI, the consumer lacks knowledge of these expected outcomes. Fix that problem by adding the highlighted lines in the following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



Creating an App Service Logic App

Azure Logic Apps explained

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

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

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

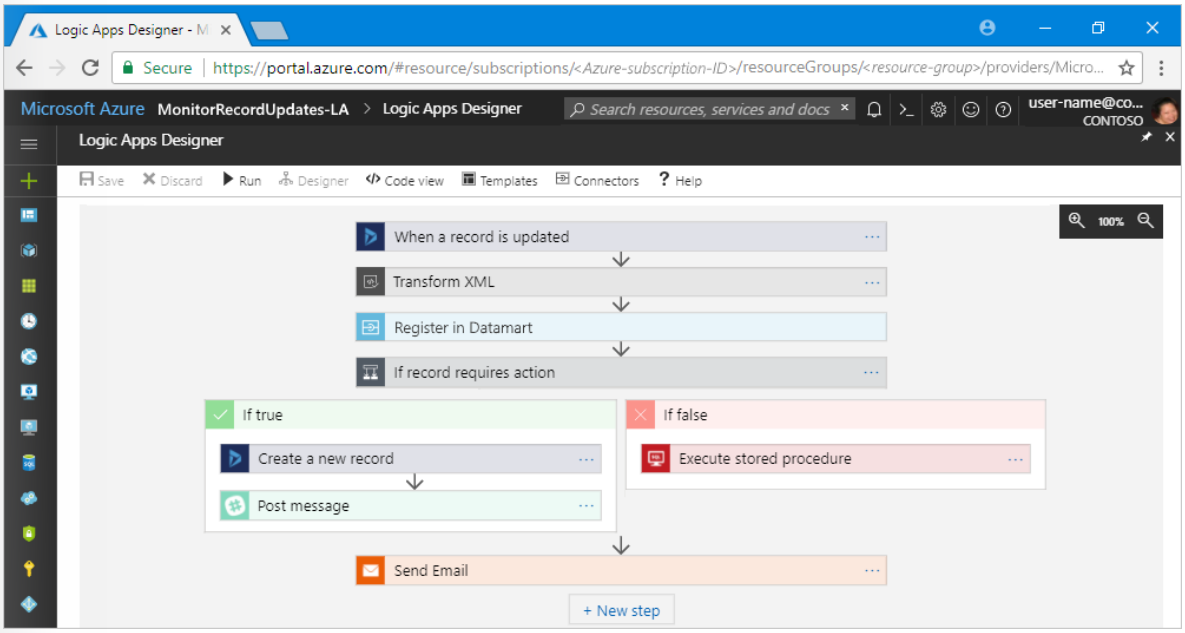
To build integration solutions with logic apps, choose from a growing gallery that has **200+ connectors**[19], including other Azure services such as Service Bus, Functions, and Storage; SQL, Office 365, Dynamics, BizTalk, Salesforce, SAP, Oracle DB, file shares, and many more. These connectors provide triggers, actions, or both for creating logic apps that securely access and process data in real time

How does Logic Apps work?

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

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

19 https://docs.microsoft.com/en-us/azure/connectors/apis-list



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

B2B scenarios and the Enterprise Integration Pack

For business-to-business (B2B) workflows and seamless communication with Azure Logic Apps, you can enable enterprise integration scenarios with Microsoft's cloud-based solution, the Enterprise Integration Pack. Organizations can exchange messages electronically, even if they use different protocols and formats. The pack transforms different formats into a format that organizations' systems can interpret and process. Organizations can exchange messages through industry-standard protocols, including AS2, X12, and EDIFACT. You can also secure messages with both encryption and digital signatures.

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

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

Why should you use enterprise integration?

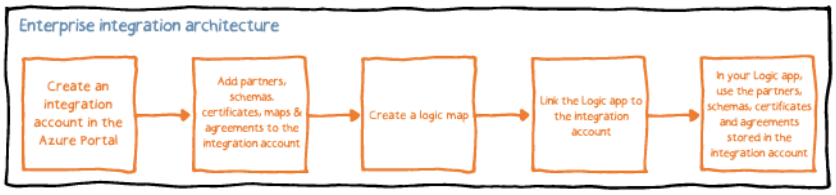
- With enterprise integration, you can store all your artifacts in one place – your integration account.

- You can build B2B workflows and integrate with third-party software-as-service (SaaS) apps, on-premises apps, and custom apps by using the Azure Logic Apps engine and all its connectors.
- You can create custom code for your logic apps with Azure functions.

How to get started with enterprise integration

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

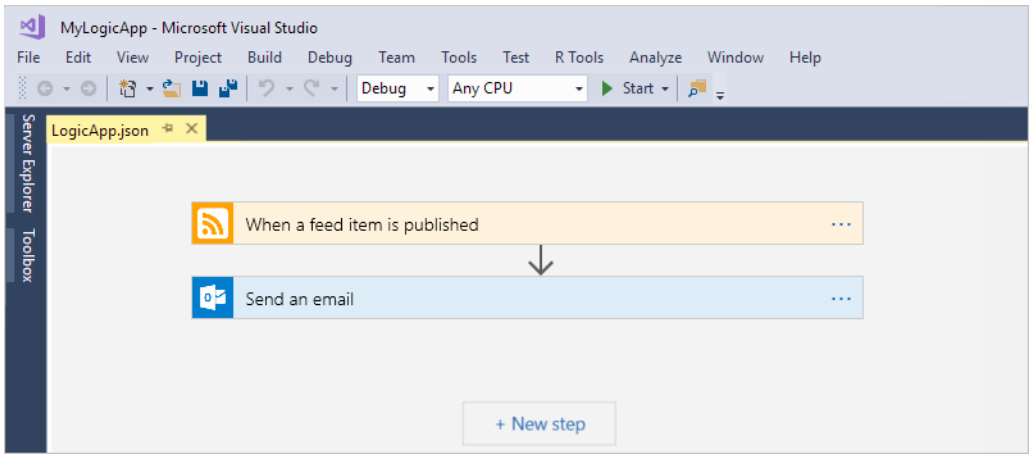
Here are the high-level steps you must take before you can create apps in the Azure portal:



Before you begin

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

This section of the course will teach you how to create a logic app monitors that website's RSS feed and sends email for each new item posted on the site. When you're done, your logic app looks like this high-level workflow:



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

- If you don't have an Azure subscription, **sign up for a free Azure account**[20].

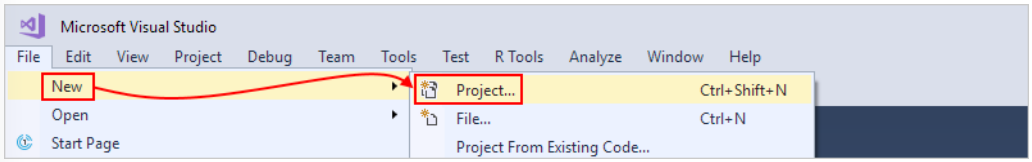
20 https://azure.microsoft.com/free/

- Download and install these tools, if you don't have them already:
  - **Visual Studio 2017 or Visual Studio 2015 - Community edition or greater**[21]. This section uses Visual Studio Community 2017, which is free.
  - **Microsoft Azure SDK for .NET (2.9.1 or later)**[22] and **Azure PowerShell**[23].
  - **Azure Logic Apps Tools for Visual Studio 2017**[24] or the **Visual Studio 2015**[25] version
- An email account that's supported by Logic Apps, such as Office 365 Outlook, Outlook.com, or Gmail. For other providers, **review the connectors list here**[26]. This logic app uses Office 365 Outlook. If you use a different provider, the overall steps are the same, but your UI might slightly differ.
- Access to the web while using the embedded Logic App Designer
- The designer requires an internet connection to create resources in Azure and to read the properties and data from connectors in your logic app. For example, if you use the Dynamics CRM Online connector, the designer checks your CRM instance for available default and custom properties.

Create an Azure resource group project

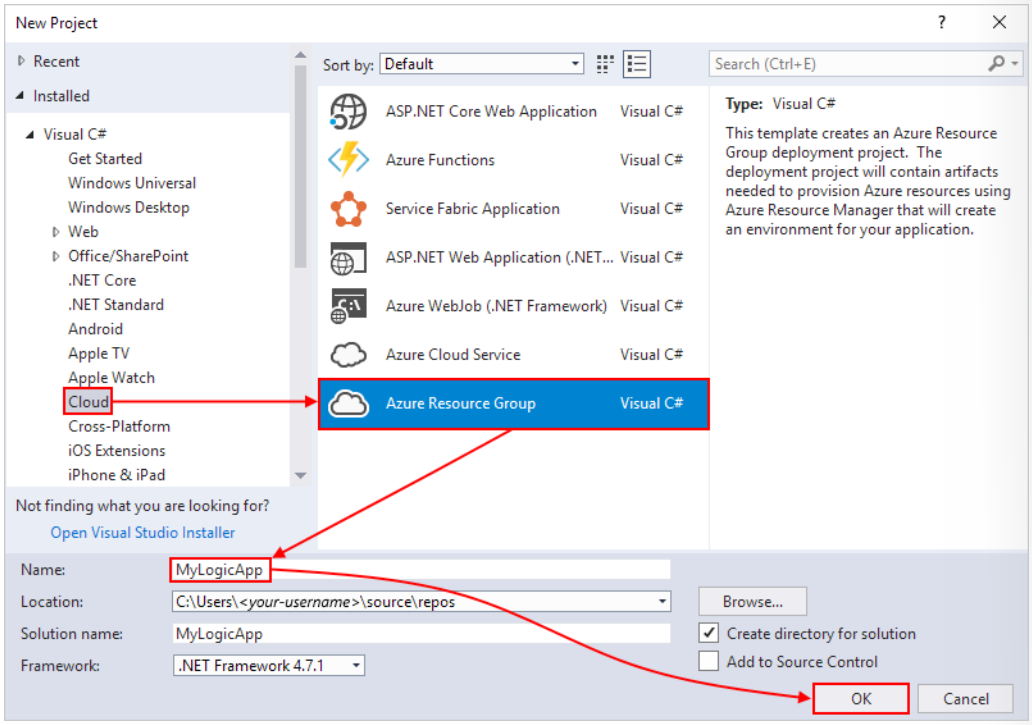
To get started, create an Azure Resource Group project.

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



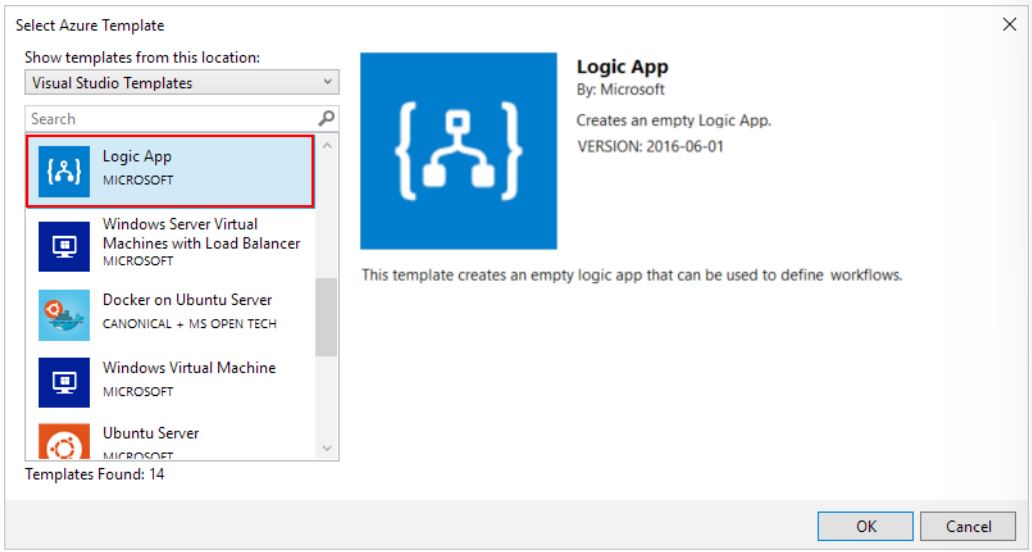
3.
4. Under **Installed**, select **Visual C#**. Select **Cloud > Azure Resource Group**. Name your project, for example:

21 https://www.visualstudio.com/downloads
22 https://azure.microsoft.com/downloads/
23 https://github.com/Azure/azure-powershell#installation
24 https://marketplace.visualstudio.com/items?itemName=VinaySinghMSFT.AzureLogicAppsToolsforVisualStudio-18551
25 https://marketplace.visualstudio.com/items?itemName=VinaySinghMSFT.AzureLogicAppsToolsforVisualStudio
26 https://docs.microsoft.com/connectors/



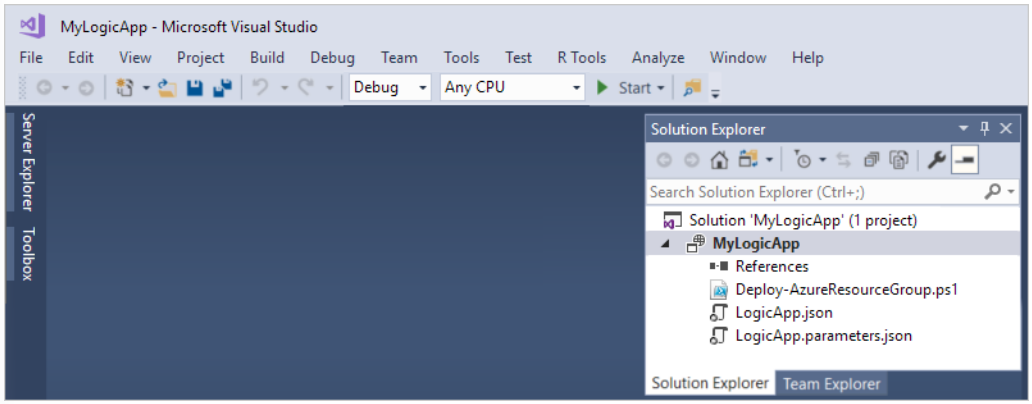
5.

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



7.

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



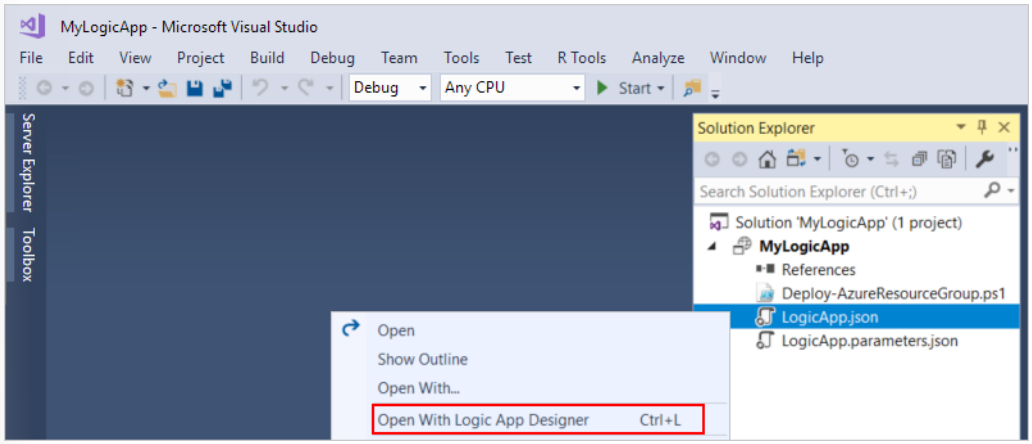
9.

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

Create a blank Logic App

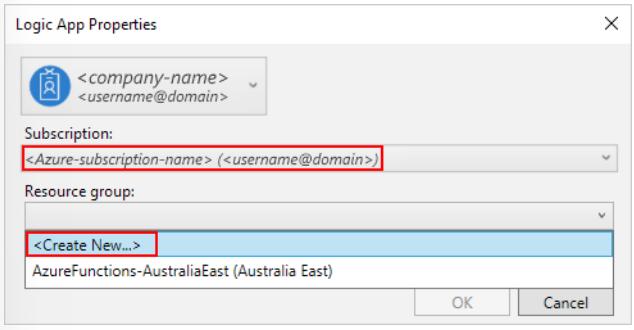
After you create your Azure Resource Group project, create and build your logic app starting from the Blank Logic App template.

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



2.

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

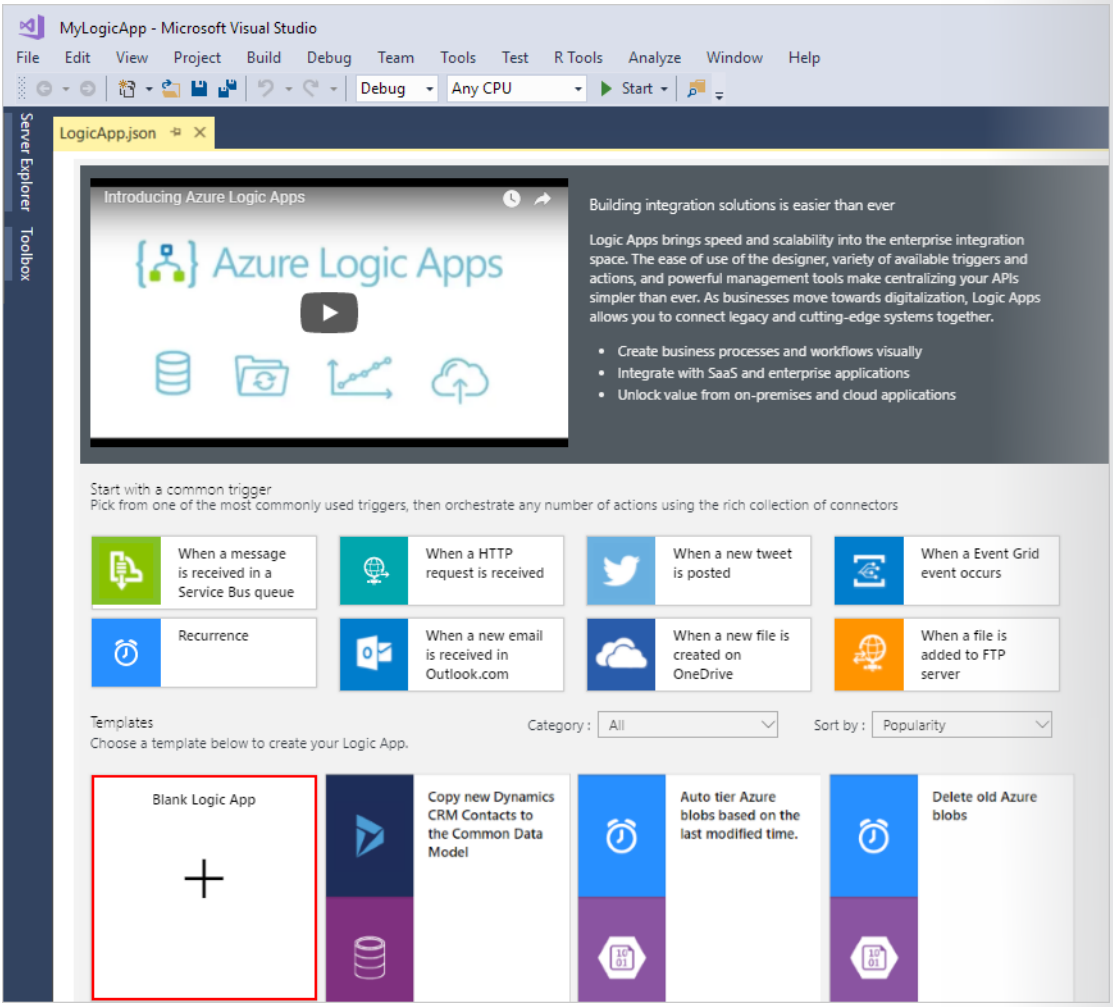


4.

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

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

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



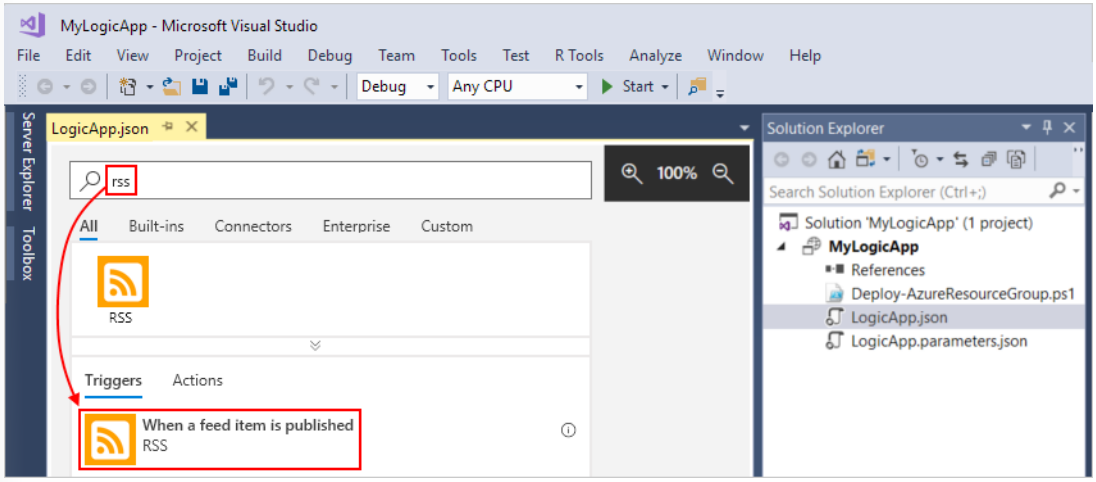
7.

Build the Logic App workflow

Creating the trigger

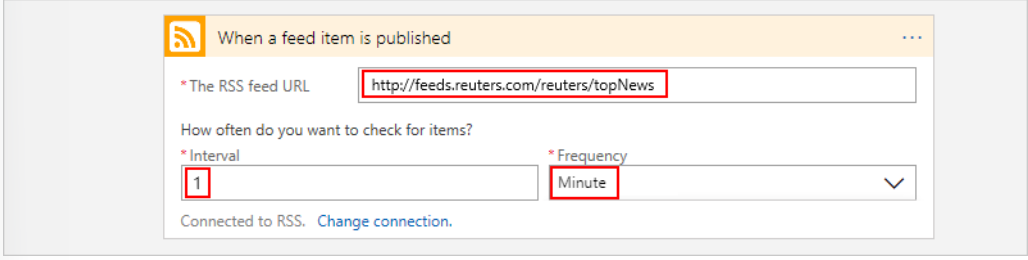
Next, add a trigger that fires when a new RSS feed item appears. Every logic app must start with a trigger, which fires when specific criteria is met. Each time the trigger fires, the Logic Apps engine creates a logic app instance that runs your workflow.

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



2.

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



4.

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

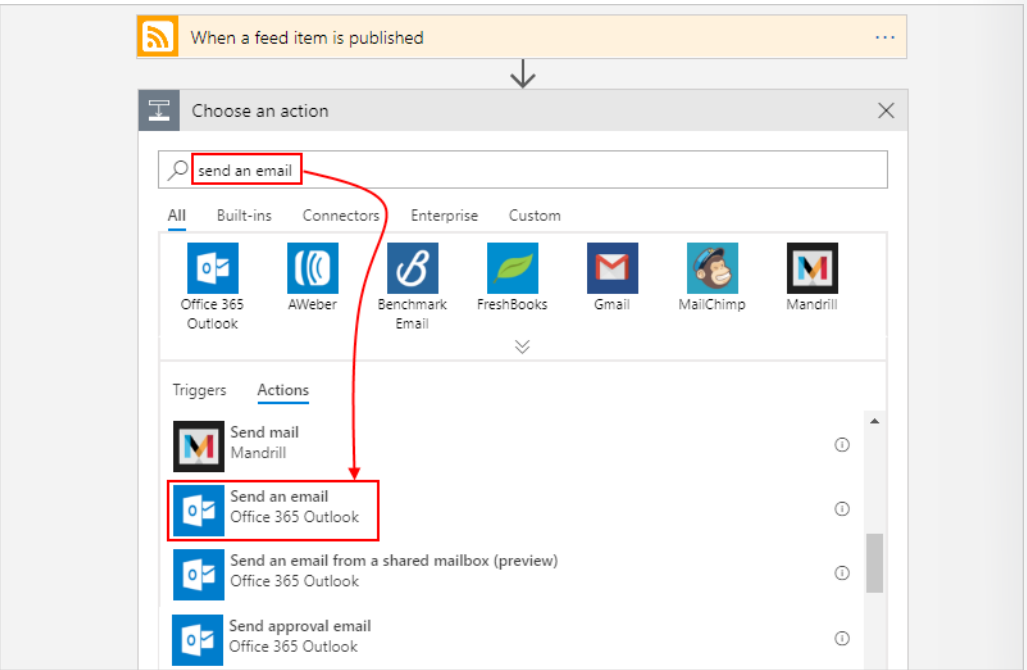
5. Together, the interval and frequency define the schedule for your logic app's trigger. This logic app checks the feed every minute.

6. Save your project.

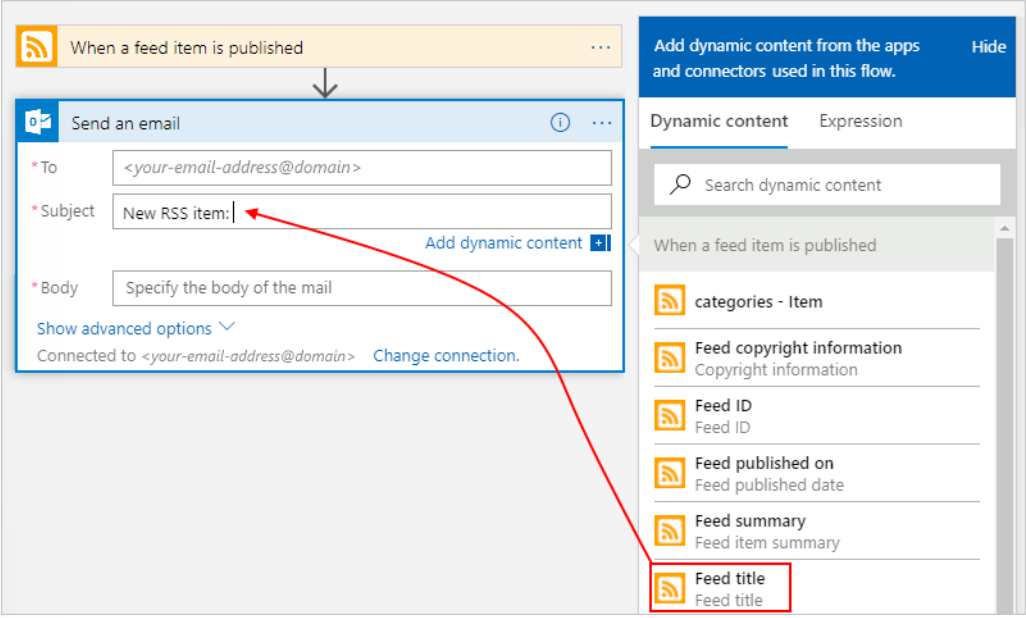
Your logic app now has a trigger, but won't actually do anything until you add an action.

Adding an action

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

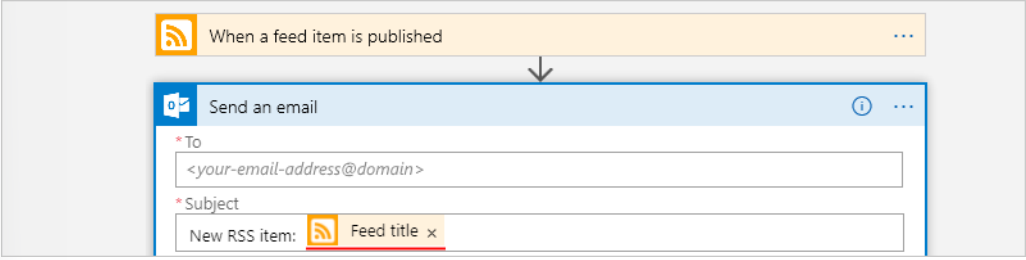


3.
4. To filter the actions list to a specific app or service, you can select that app or service first:
   - For Azure work or school accounts, select Office 365 Outlook.
   - For personal Microsoft accounts, select Outlook.com.
5. If asked for credentials, sign in to your email account so that Logic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



10.

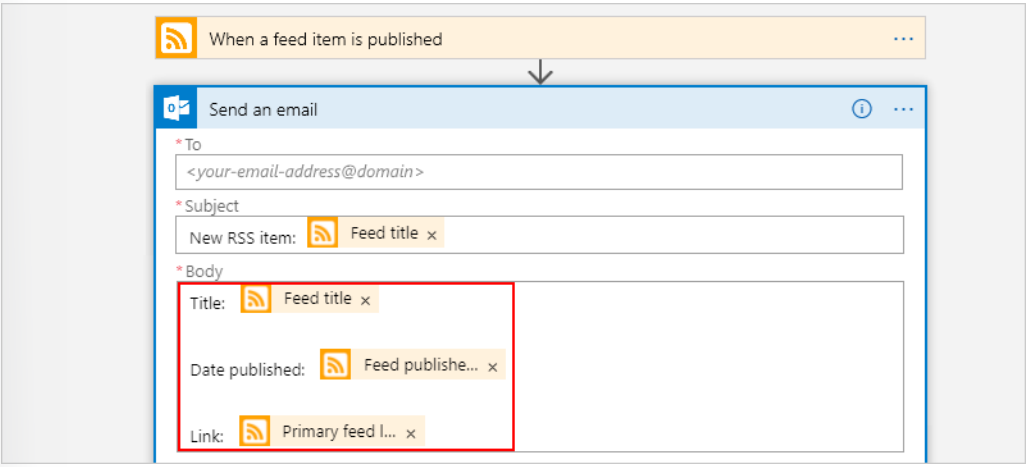
11. When you're done, the email subject looks like this example:



12.

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

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

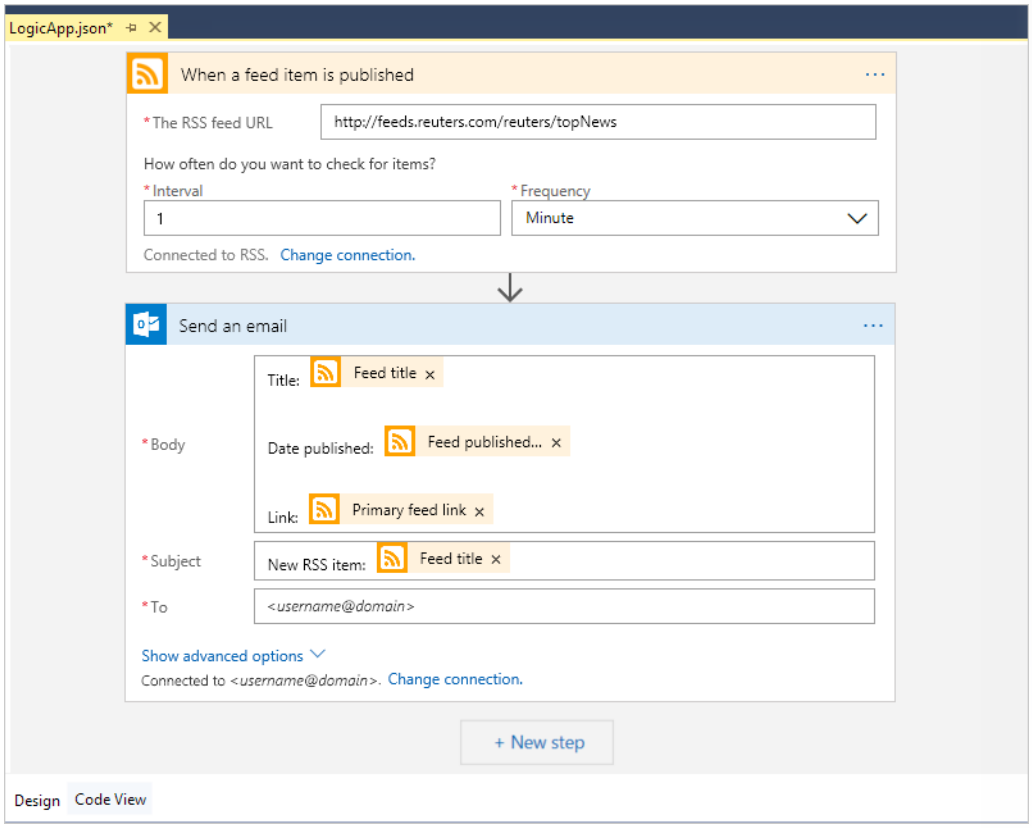


15.

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

16. Save your logic app.

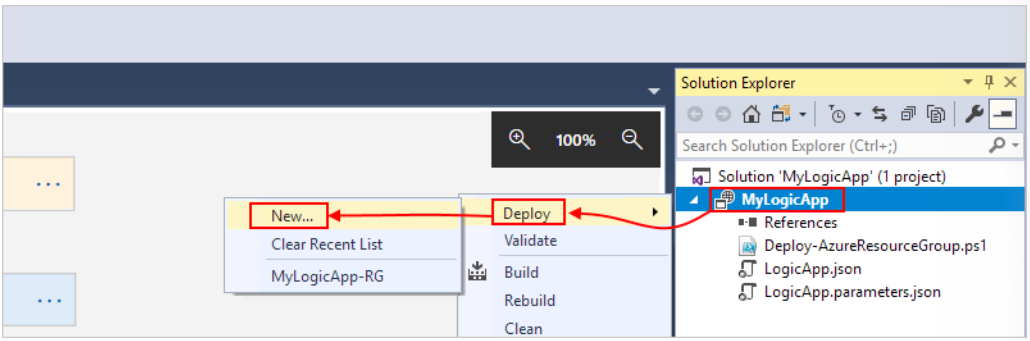
When you're done, your logic app looks like this example:



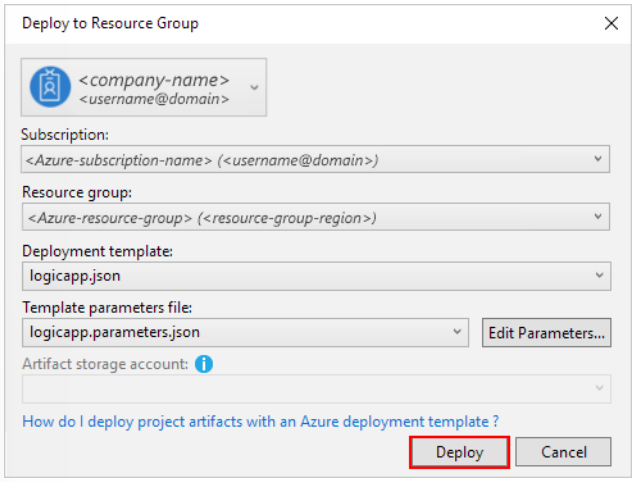
Deploy the Logic App to Azure

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

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

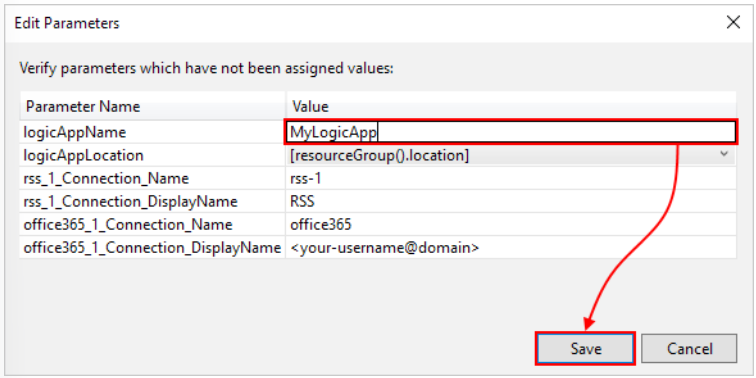
2. 

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



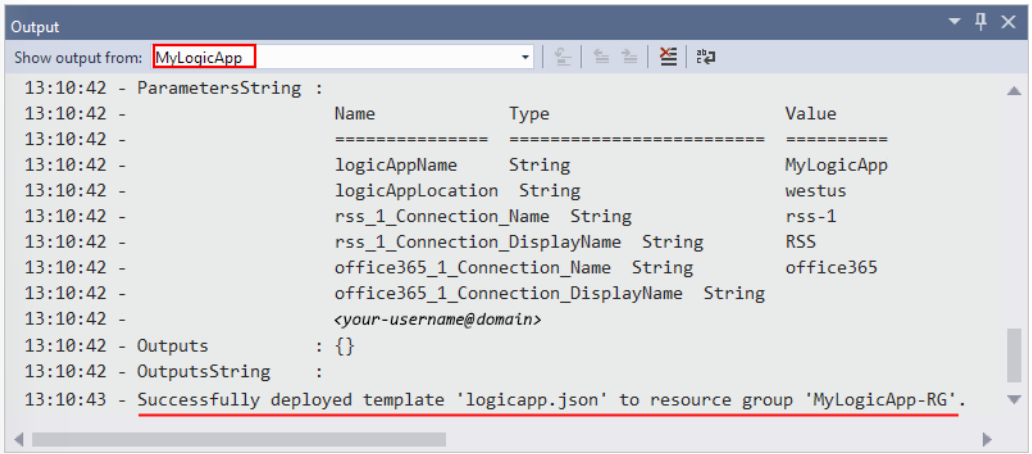
4.

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



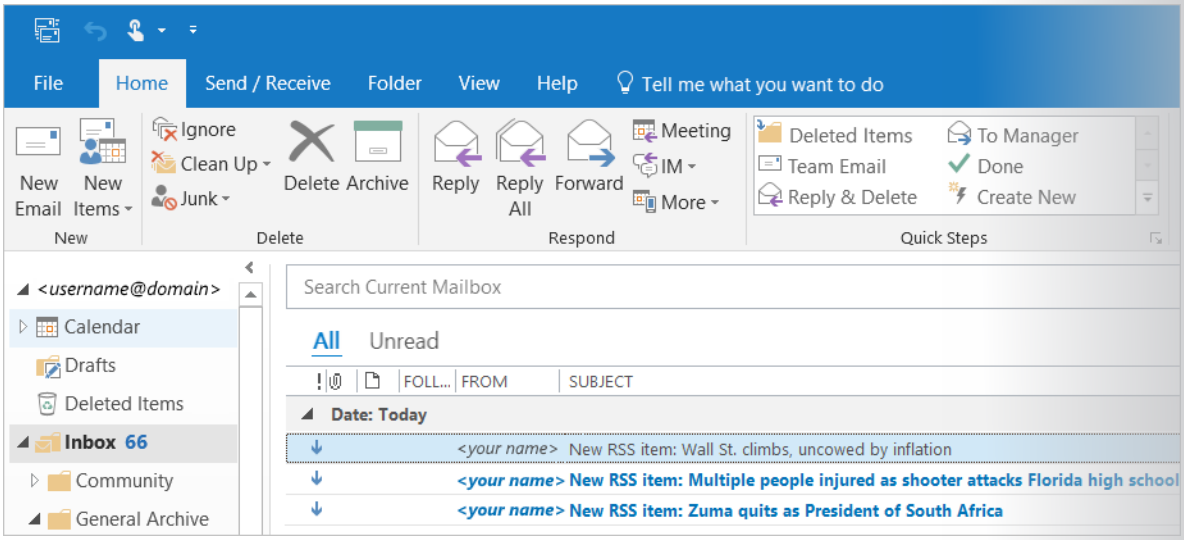
6.

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



8.

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



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

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

Clean up resources

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

1. Sign in to the **Azure portal**[27] with the same account used to create your logic app.
2. On the main Azure menu, select **Resource groups**. Select the resource group for your logic app, and then select **Overview**.
3. On the **Overview** page, choose **Delete resource group**. Enter the resource group name as confirmation, and choose **Delete**.
4. <img src="../..\Linked_Image_Files\M03L0206delete-resource-group.png" alt=""Resource groups" > "Overview" > "Delete resource group"" title="">

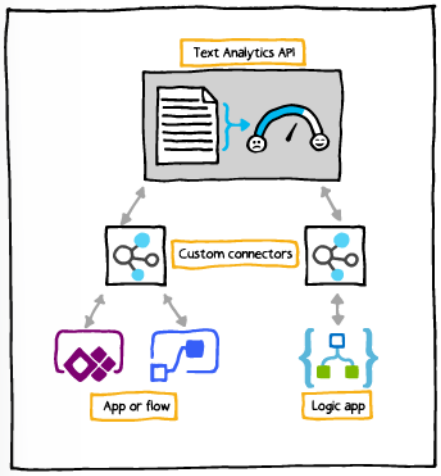
Custom connectors overview

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

Sometimes though, you might want to call APIs, services, and systems that aren't available as prebuilt connectors. To support more tailored scenarios, you can build *custom connectors* with their own triggers and actions. These connectors are *function-based* - data is returned based on calling specific functions in

27 https://portal.azure.com/

the underlying service. The following diagram shows a custom connector for an API that detects sentiment in text.

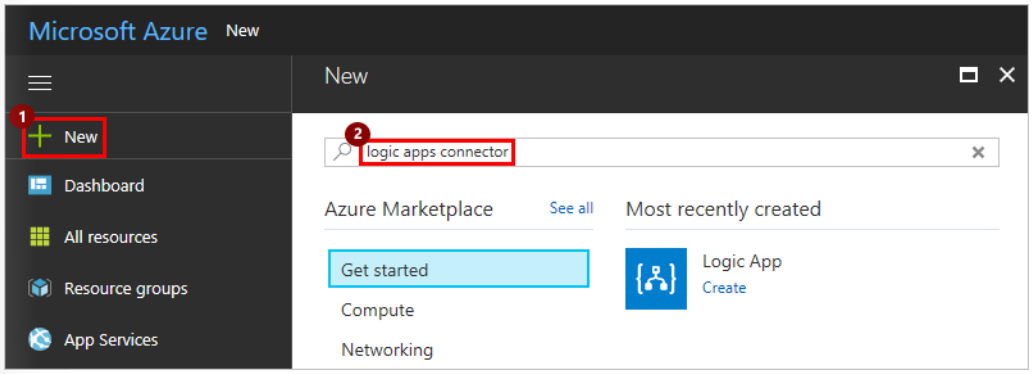


The following diagram shows the high-level tasks involved in creating and using custom connectors:

Create a custom connector in Logic Apps

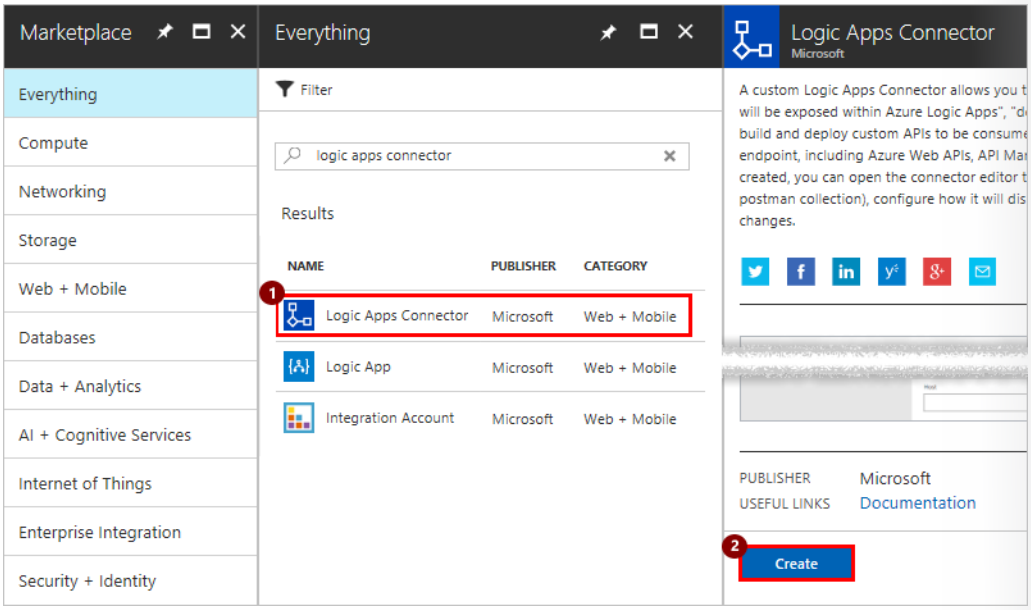
In Logic Apps, you first create a custom connector (covered in this topic), and then you define the behavior of the connector using an OpenAPI definition or a Postman collection (covered in subsequent topics).

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



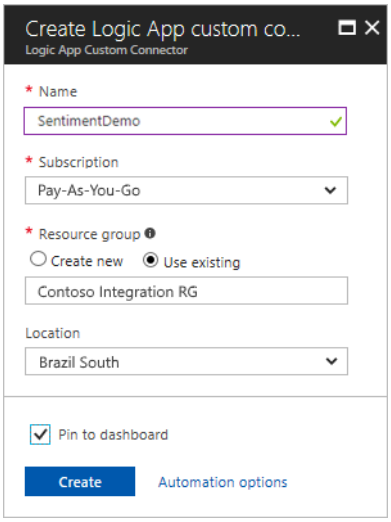
2.

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



4.

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



6.

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

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

Next Steps

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

Import the OpenAPI definition

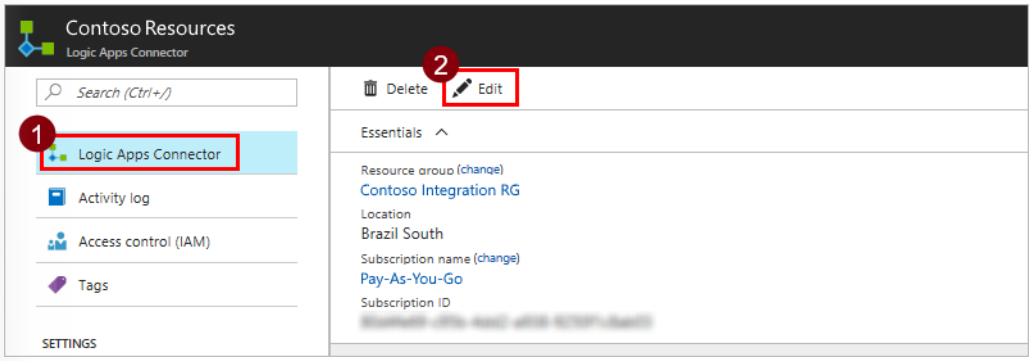
To create a custom connector, you must describe the API you want to connect to so that the connector understands the API's operations and data structures.

Prerequisites

- An **OpenAPI definition**[28] that describes the example API. When creating a custom connector, the OpenAPI definition must be less than 1 MB.
- An **API key**[29] for the Cognitive Services Text Analytics API
- One of the following subscriptions:
  - **Azure**[30], if you're using Logic Apps
  - **Microsoft Flow**[31]
  - **PowerApps**[32]

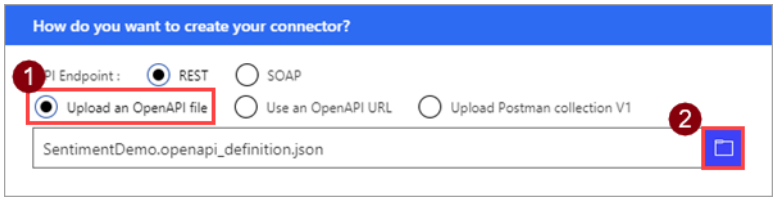
You're now ready to work with the OpenAPI definition you downloaded. All the required information is contained in the definition, and you can review and update this information as you go through the custom connector wizard.

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



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

28 https://procsi.blob.core.windows.net/docs/SentimentDemo.openapi_definition.json
29 https://docs.microsoft.com/en-us/connectors/custom-connectors/index#get-an-api-key
30 https://azure.microsoft.com/get-started/
31 https://docs.microsoft.com/flow/sign-up-sign-in
32 https://docs.microsoft.com/powerapps/signup-for-powerapps

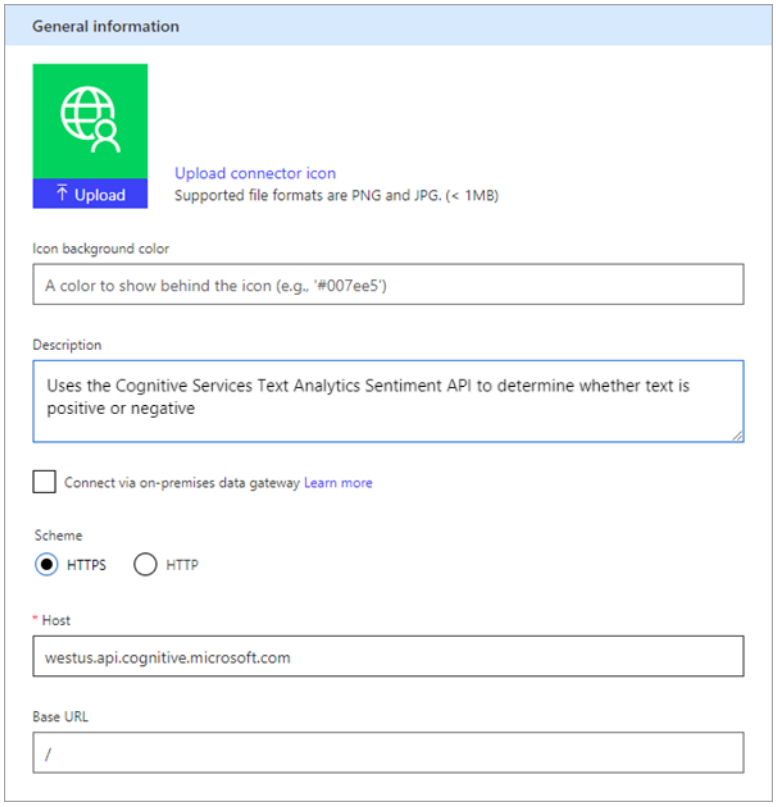


5.

Review general details

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

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



3.

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

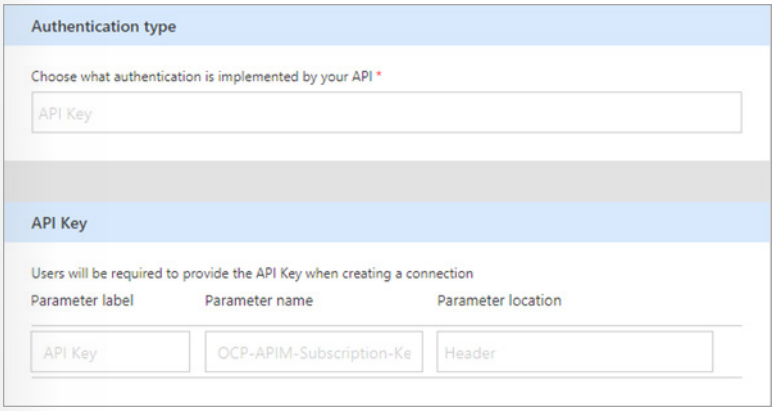
```
"info": {
    "version": "1.0.0",
    "title": "SentimentDemo",
    "description": "Uses the Cognitive Services Text Analytics Sentiment
API to determine whether text is positive or negative"
},
```

```
"host": "westus.api.cognitive.microsoft.com",
"basePath": "/",
"schemes": [
    "https"
]
```

Review authentication type

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

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



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

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

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

Logic App deployment template overview

After a logic app has been created, you might want to create it as an Azure Resource Manager template. This way, you can easily deploy the logic app to any environment or resource group where you might need it. For more about Resource Manager templates, see **authoring Azure Resource Manager templates**[33] and **deploying resources by using Azure Resource Manager templates**[34].

33 https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates
34 https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-template-deploy

A logic app has three basic components:

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

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

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

Create a Logic App deployment template

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

A few other tools can assist you as you create a logic app deployment template. You can author by hand, that is, by using the resources already discussed here to create parameters as needed. Another option is to use a **logic app template creator**[35] PowerShell module. This open-source module first evaluates the logic app and any connections that it is using, and then generates template resources with the necessary parameters for deployment. For example, if you have a logic app that receives a message from an Azure Service Bus queue and adds data to an Azure SQL database, the tool preserves all the orchestration logic and parameterizes the SQL and Service Bus connection strings so that they can be set at deployment.

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

Install the logic app template PowerShell module

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

For the module to work with any tenant and subscription access token, we recommend that you use it with the **ARMClient**[36] command-line tool. This **blog post**[37] discusses ARMClient in more detail.

Generate a logic app template by using PowerShell

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

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

35 https://github.com/jeffhollan/LogicAppTemplateCreator
36 https://github.com/projectkudu/ARMClient
37 http://blog.davidebbo.com/2015/01/azure-resource-manager-client.html

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

Add parameters to a Logic App template

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

Add references for dependent resources to Visual Studio deployment templates

When you want your logic app to reference dependent resources, you can use **Azure Resource Manager template functions**[38] in your logic app deployment template. For example, you might want your logic app to reference an Azure Function or integration account that you want to deploy alongside your logic app. Follow these guidelines about how to use parameters in your deployment template so that the Logic App Designer renders correctly.

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

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

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

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

And where you would use parameters:

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

38 https://docs.microsoft.com/azure/azure-resource-manager/resource-group-template-functions

```
    },
    "runAfter":{}
}
```

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

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

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

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

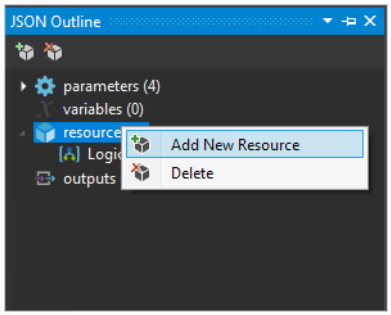
```
"parameters": {
    "IntegrationAccount": {
    "type":"string",
    "minLength":1,
    "defaultValue":"/subscriptions/<subscriptionID>/resourceGroups/<re-
sourceGroupName>/providers/Microsoft.Logic/integrationAccounts/<integra-
tionAccountName>"
    }
},
```

Adding your Logic App to an existing Resource Group

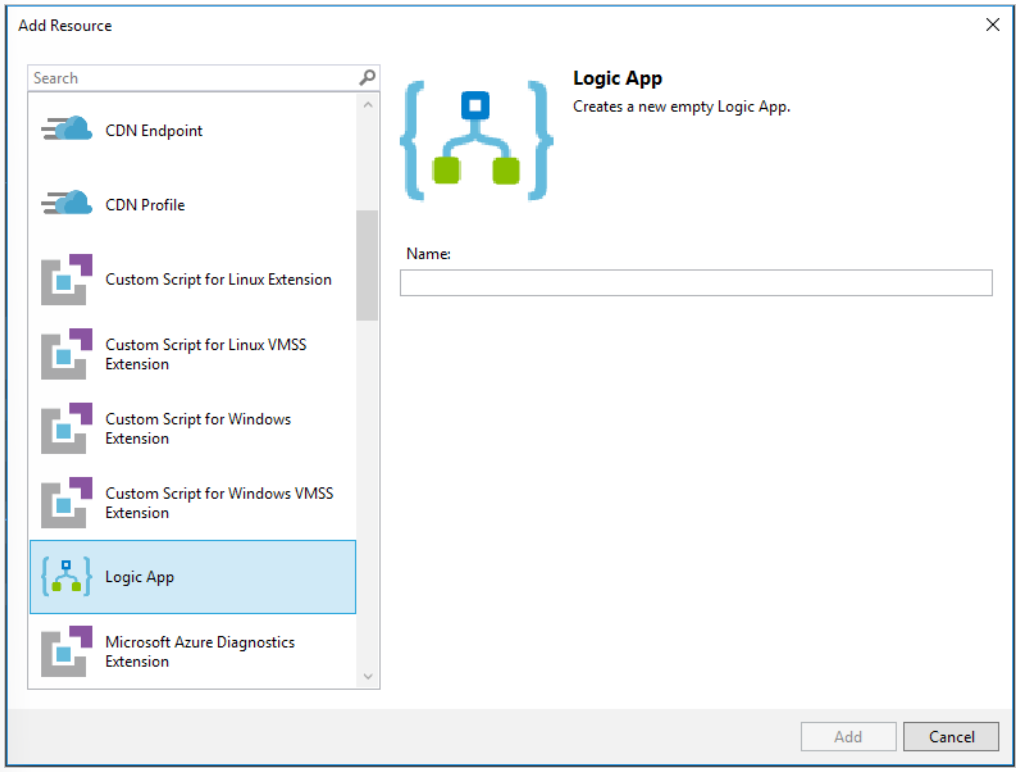
If you have an existing Resource Group project, you can add your logic app to that project in the JSON Outline window. You can also add another logic app alongside the app you previously created.

1. Open the `<template>.json` file.

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



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



6.

Deploy a logic app template

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

Authorize OAuth connections

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

Online Lab - Implementing and Managing Application Services

Lab Steps

Online Lab - Implementing and Managing Application Services

Topic: Implementing Azure Logic Apps

Scenario

Adatum Corporation wants to implement custom monitoring of changes to a resource group.

Objectives

After completing this lab, you will be able to:

- Create an Azure logic app
- Configure integration of an Azure logic app and an Azure event grid

Lab Setup

Estimated Time: 45 minutes

User Name: **Student**

Password: **Pa55w.rd**

Exercise 1: Set up the lab environment that consists of an Azure storage account and an Azure logic app

The main tasks for this exercise are as follows:

1. Create an Azure storage account
2. Create an Azure logic app
3. Create an Azure AD service principal
4. Assign the Reader role to the Azure AD service principal
5. Register the Microsoft.EventGrid resource provider

Task 1: Create a storage account in Azure

1. From the lab virtual machine, start Microsoft Edge and browse to the Azure portal at **http://portal.azure.com** and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.

2. From Azure Portal, create a new storage account with the following settings:
   - Subscription: the name of the target Azure subscription
   - Resource group: a new resource group named **az3000701-LabRG**
   - Storage account name: any valid, unique name between 3 and 24 characters consisting of lower-case letters and digits
   - Location: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Performance: **Standard**
   - Account kind: **Storage (general purpose v1)**
   - Replication: **Locally-redundant storage (LRS)**
   - Secure transfer required: **Enabled**
   - Allow access from: **All networks**
   - Hierarchical namespace: **Disabled**
3. **Note**: Do not wait for the deployment to complete but instead proceed to the next task.

Task 2: Create an Azure logic app

1. From Azure Portal, create an instance of **Logic App** with the following settings:
   - Name: **logicapp3000701**
   - Subscription: the name of the target Azure subscription
   - Resource group: the name of a new resource group **az3000702-LabRG**
   - Location: the same Azure region into which you deployed the storage account in the previous task
   - Log Analytics: **Off**
2. Wait until the logic app is provisioned. This will take about a minute.

Task 3: Create an Azure AD service principal

1. In the Azure portal, in the Microsoft Edge window, start a **PowerShell** session within the **Cloud Shell**.
2. If you are presented with the **You have no storage mounted** message, configure storage using the following settings:
   - Subsciption: the name of the target Azure subscription
   - Cloud Shell region: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Resource group: the name of a new resource group **az3000700-LabRG**
   - Storage account: a name of a new storage account
   - File share: a name of a new file share
3. From the Cloud Shell pane, run the following to create a new Azure AD application that you will associate with the service principal you create in the subsequent steps of this task:
4. `$password = 'Pa55w.rd1234' $securePassword = ConvertTo-SecureString -Force -AsPlainText -String $password $aadApp30007 = New-AzureRmADApplication`

```
-DisplayName 'aadApp30007' -HomePage 'http://aadApp30007' -IdentifierUris
'http://aadApp30007' -Password $securePassword
```

5. From the Cloud Shell pane, run the following to create a new Azure AD service principal associated with the application you created in the previous step:
6. `New-AzureRmADServicePrincipal -ApplicationId $aadApp30007.ApplicationId.Guid`
7. In the output of the **New-AzureRmADServicePrincipal** command, note the value of the **ApplicationId** property. You will need this in the next exercise of this lab.
8. From the Cloud Shell pane, run the following to identify the value of the **Id** property of the current Azure subscription and the value of the **TenantId** property of the Azure AD tenant associated with that subscription (you will also need them in the next exercise of this lab):
9. `Get-AzureRmSubscription`
10. Close the Cloud Shell pane.

Task 4: Assign the Reader role to the Azure AD service principal

1. In the Azure portal, navigate to the blade displaying properties of your Azure subscription.
2. On the Azure subscription blade, click **Access control (IAM)**.
3. Assign the **Reader** role within the scope of the Azure subscription to the **aadApp30007** service principal.

Task 5: Register the Microsoft.EventGrid resource provider

1. In the Azure portal, in the Microsoft Edge window, reopen the **PowerShell** session within the **Cloud Shell**.
2. From the Cloud Shell pane, run the following to register the Microsoft.EventGrid resource provider:
3. `Register-AzureRmResourceProvider -ProviderNamespace Microsoft.EventGrid`
4. Close the Cloud Shell pane.

Result: After you completed this exercise, you have created a storage account, a logic app that you will configure in the next exercise of this lab, and an Azure AD service principal that you will reference during that configuration.

Exercise 2: Configure Azure logic app to monitor changes to a resource group.

The main tasks for this exercise are as follows:

1. Add a trigger to the Azure logic app
2. Add an action to the Azure logic app
3. Identify the callback URL of the Azure logic app
4. Configure an event subscription
5. Test the logic app

Task 1: Add a trigger to the Azure logic app

1. In the the Azure portal, navigate to the **Logic App Designer** blade of the newly provisioned Azure logic app.
2. Click **Blank Logic App**. This will create a blank designer workspace and display a list of connectors and triggers to add to the workspace.
3. Search for **Event Grid** triggers and, in the list of results, click the **When a resource event occurs (preview) Azure Event Grid** trigger to add it to the designer workspace.
4. In the **Azure Event Grid** tile, click the **Connect with Service Principal** link, specify the following values, and click **Create**:
   - Connection Name: **egc30007**
   - Client ID: the **ApplicationId** property you identified in the previous exercise
   - Client Secret: **Pa55w.rd1234**
   - Tenant: the **TenantId** property you identified in the previous exercise
5. In the **When a resource event occurs** tile, specify the following values:
   - Subscription: the subscription **Id** property you identified in the previous exercise
   - Resource Type: **Microsoft.Resources.resourceGroups**
   - Resource Name: */subscriptions/subscriptionId*/resourceGroups/az3000701-LabRG**, where ***subscriptionId*** is the subscription **Id** property you identified in the previous exercise
   - Event Type Item - 1: **Microsoft.Resources.ResourceWriteSuccess**
   - Event Type Item - 2: **Microsoft.Resources.ResourceDeleteSuccess**
6. Click **Show advanced options**, in the **Subscription Name** text box, type **event-subscription-az3000701** and click **Save**.

Task 2: Add an action to the Azure logic app

1. In the the Azure portal, on the Logic App Designer blade of the newly provisioned Azure logic app, click **+ New step**.
2. In the **Choose an action** pane, in the **Search connectors and actions** text box, type **Outlook**.
3. In the list of results, click **Outlook.com**.
4. In the list of actions for **Outlook.com**, click **Outlook.com - Send an email**.
5. In the **Outlook.com** pane, click **Sign in**.
6. When prompted, authenticate by using the Microsoft Account you are using in this lab.
7. When prompted for the consent to grant Azure Logic App permissions to access Outlook resources, click **Yes**.
8. In the **Send an email** pane, specify the following settings and click **Save**:
   - To: the name of your Microsoft Account
   - Subject: type **Resource updated:** and, in the **Dynamic Content** column to the right of the **Send an email** pane, click **Subject**.
   - Body: type **Resource group**: in the Dynamic Content column to the right of the Send an email pane, click **Topic**, type **Event type**:, in the Dynamic Content column to the right of the Send an

email pane, click **Event Type**, type **Event ID**:, in the Dynamic Content column to the right of the Send an email pane, click **ID**, type **Event Time**:, and in the Dynamic Content column to the right of the Send an email pane, click **Event Time.**

Task 3: Identify the callback URL of the Azure logic app

1. In the Azure portal, navigate to the **logicapp3000701** blade and, in the **Summary** section, click **See trigger history**.
2. On the **When*a*resource*event*occurs** blade, copy the value of the **Callback url [POST]** text box.

Task 4: Configure an event subscription

1. In the Azure portal, navigate to the **az3000701-LabRG** resource group and, in the vertical menu, click **Events**.
2. On the **az3000701-LabRG - Events** blade, click **Web Hook**.
3. On the **Create Event Subscription** blade, clear the **Subscribe to all event types** checkbox and, in the **Defined Event Types** drop down list, ensure that only the checkboxes next to the **Resource Write Success** and **Resource Delete Success** are selected.
4. In the **Endpoint Type** drop down list, select the **Web Hook** entry and then click the **Select an endpoint** link.
5. On the **Select Web Hook** blade, in the **Subscriber Endpoint**, paste the value of the **Callback url [POST]** of the Azure logic app you copied in the previous task and click **Confirm Selection**.
6. In the **Name** text box within the **EVENT SUBSCRIPTION DETAILS** section, type **event-subscription-az3000701**.
7. Click **Create**.

Task 5: Test the logic app

1. In the Azure portal, navigate to the **az3000701-LabRG** resource group and, in the vertical menu, click **Overview**.
2. In the list of resources, click the Azure storage account you created in the first exercise.
3. On the storage account blade, in the vertical menu, click **Configuration**.
4. On the configuration blade, set the **Secure transfer required** setting to **Disabled**
5. Navigate to the **logicapp3000701** blade, click **Refresh**, and note that the **Runs history** includes the entry corresponding to configuration change of the Azure storage account.
6. Navigate to the inbox of the email account you configured in this exercise and verify that includes an email generated by the logic app.

Result: After you completed this exercise, you have configured an Azure logic app to monitor changes to a resource group.

Review Questions

Module 1 Review Questions

App Service Environments

You have web applications that periodically require additional resources. At other times, the applications are idle. You plan to deploy the web applications to an Azure App Service Environment (ASE).

How many App Service plans and instances can you run in your subscription? What benefits can you realize by using Azure ASE?

Suggested Answer ↓

An App Service environment (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.

ASEs are appropriate for application workloads that require:

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

Azure Cloud Shell

You have web applications that periodically require additional resources. At other times, the applications are idle. You plan to deploy the web applications to an Azure App Service Environment (ASE).

You need to manage the ASE from the command line. What tool should you use?

Suggested Answer ↓

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.

App Service Environments

You develop a web application that queries a database on a daily schedule and produces a report. You decide that a WebJob will produce the desired result.

What type of WebJob is required? Can you remotely debug the WebJob?

Suggested Answer ↓

There are two types of WebJobs, continuous and triggered. Continuous WebJobs start as soon as you create the WebJob and runs on all instances that the web application runs on. Continuous WebJobs support remote debugging.

In contrast, you must manually start triggered WebJobs or run them on a schedule. Triggered WebJobs run on a single instance (selected by Azure) and do not support remote debugging.

Module 2 Module Creating Apps and Services Running on Service Fabric

Understanding Azure Service Fabric

Azure Service Fabric overview

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Service Fabric also addresses the significant challenges in developing and managing cloud native applications. Developers and administrators can avoid complex infrastructure problems and focus on implementing mission-critical, demanding workloads that are scalable, reliable, and manageable. Service Fabric represents the next-generation platform for building and managing these enterprise-class, tier-1, cloud-scale applications running in containers.

Applications composed of microservices

Service Fabric enables you to build and manage scalable and reliable applications composed of microservices that run at high density on a shared pool of machines, which is referred to as a cluster. It provides a sophisticated, lightweight runtime to build distributed, scalable, stateless, and stateful microservices running in containers. It also provides comprehensive application management capabilities to provision, deploy, monitor, upgrade/patch, and delete deployed applications including containerized services.

Service Fabric powers many Microsoft services today, including Azure SQL Database, Azure Cosmos DB, Cortana, Microsoft Power BI, Microsoft Intune, Azure Event Hubs, Azure IoT Hub, Dynamics 365, Skype for Business, and many core Azure services.

Service Fabric is tailored to create cloud native services that can start small, as needed, and grow to massive scale with hundreds or thousands of machines.

Today's Internet-scale services are built of microservices. Examples of microservices include protocol gateways, user profiles, shopping carts, inventory processing, queues, and caches. Service Fabric is a microservices platform that gives every microservice (or container) a unique name that can be either stateless or stateful.

Service Fabric provides comprehensive runtime and lifecycle management capabilities to applications that are composed of these microservices. It hosts microservices inside containers that are deployed and

activated across the Service Fabric cluster. A move from virtual machines to containers makes possible an order-of-magnitude increase in density. Similarly, another order of magnitude in density becomes possible when you move from containers to microservices in these containers. For example, a single cluster for Azure SQL Database comprises hundreds of machines running tens of thousands of containers that host a total of hundreds of thousands of databases. Each database is a Service Fabric stateful microservice.

Container deployment and orchestration

Service Fabric is Microsoft's container orchestrator deploying microservices across a cluster of machines. Microservices can be developed in many ways from using the Service Fabric programming models, ASP.NET Core, to deploying any code of your choice. Importantly, you can mix both services in processes and services in containers in the same application. If you just want to deploy and manage containers, Service Fabric is a perfect choice as a container orchestrator.

Stateless and stateful microservices for Service Fabric

Service Fabric enables you to build applications that consist of microservices or containers. Stateless microservices (such as protocol gateways and web proxies) do not maintain a mutable state outside a request and its response from the service. Azure Cloud Services worker roles are an example of a stateless service. Stateful microservices (such as user accounts, databases, devices, shopping carts, and queues) maintain a mutable, authoritative state beyond the request and its response. Today's Internet-scale applications consist of a combination of stateless and stateful microservices.

A key differentiation with Service Fabric is its strong focus on building stateful services, either with the built-in programming models or with containerized stateful services.

Service Fabric app scenarios

Azure Service Fabric offers a reliable and flexible platform that enables you to write and run many types of business applications and services. These applications and microservices can be stateless or stateful, and they are resource-balanced across virtual machines to maximize efficiency. The unique architecture of Service Fabric enables you to perform near real-time data analysis, in-memory computation, parallel transactions, and event processing in your applications. You can easily scale your applications up or down (really in or out), depending on your changing resource requirements.

Design applications composed of stateless and stateful microservices

Building applications with Azure Cloud Service worker roles is an example of a stateless service. In contrast, stateful microservices maintain their authoritative state beyond the request and its response. This provides high availability and consistency of the state through simple APIs that provide transactional guarantees backed by replication. Service Fabric's stateful services democratize high availability, bringing it to all types of applications, not just databases and other data stores. This is a natural progression. Applications have already moved from using purely relational databases for high availability to NoSQL databases. Now the applications themselves can have their “hot” state and data managed within them for additional performance gains without sacrificing reliability, consistency, or availability.

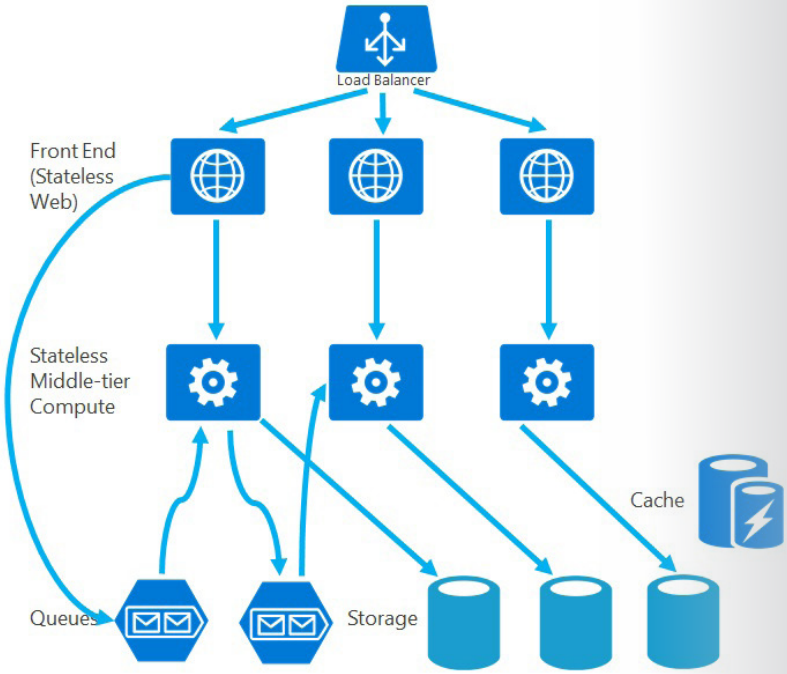
When building applications consisting of microservices, you typically have a combination of stateless web apps (ASP.NET, Node.js, etc.) calling onto stateless and stateful business middle-tier services, all deployed into the same Service Fabric cluster using the Service Fabric deployment commands. Each of these

services is independent with regard to scale, reliability, and resource usage, greatly improving agility in development and lifecycle management.

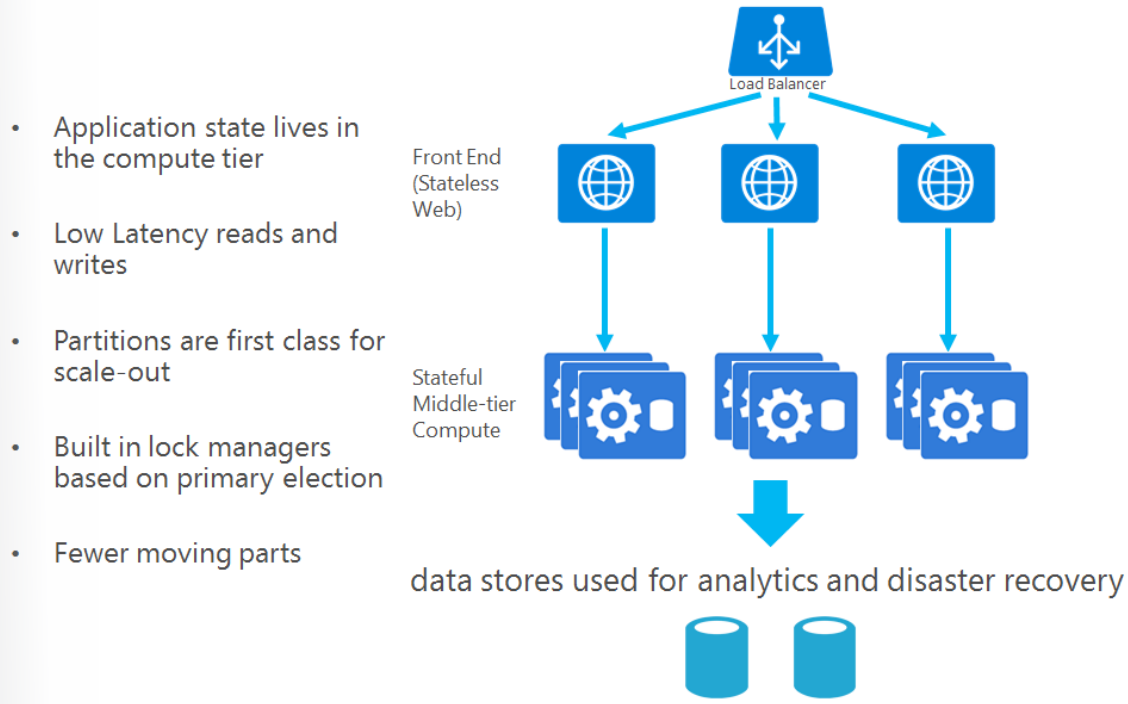
Stateful microservices simplify application designs because they remove the need for the additional queues and caches that have traditionally been required to address the availability and latency requirements of purely stateless applications. Since stateful services are naturally highly available and low latency, this means that there are fewer moving parts to manage in your application as a whole. The diagrams below illustrate the differences between designing an application that is stateless and one that is stateful. By taking advantage of the Reliable Services and Reliable Actors programming models, stateful services reduce application complexity while achieving high throughput and low latency.

An application built using stateless services

- Scale with partitioned storage
- Increase reliability with queues
- Reduce read latency with caches
- Write your own lock managers for state consistency
- Many moving parts each managed differently



An application built using stateful services



Reliable Services concepts

An Azure Service Fabric application contains one or more services that run your code. This section of the course shows you how to create both stateless and stateful Service Fabric applications with Reliable Services.

Basic concepts

To get started with Reliable Services, you only need to understand a few basic concepts:

- **Service type:** This is your service implementation. It is defined by the class you write that extends StatelessService and any other code or dependencies used therein, along with a name and a version number.
- **Named service instance:** To run your service, you create named instances of your service type, much like you create object instances of a class type. A service instance has a name in the form of a URI using the “fabric:/” scheme, such as "fabric:/MyApp/MyService".
- **Service host:** The named service instances you create need to run inside a host process. The service host is just a process where instances of your service can run.
- **Service registration:** Registration brings everything together. The service type must be registered with the Service Fabric runtime in a service host to allow Service Fabric to create instances of it to run.

Preparing your development environments on Windows

To build and run Azure Service Fabric applications on your Windows development machine, install the Service Fabric runtime, SDK, and tools. You also need to enable execution of the Windows PowerShell scripts included in the SDK.

Prerequisites

The following operating system versions are supported for development:

- Windows 7
- Windows 8/Windows 8.1
- Windows Server 2012 R2
- Windows Server 2016
- Windows 10

Windows 7 support:

- Windows 7 only includes Windows PowerShell 2.0 by default. Service Fabric PowerShell cmdlets requires PowerShell 3.0 or higher. You can **download Windows PowerShell 5.0**[1] from the Microsoft Download Center.
- Service Fabric Reverse Proxy is not available on Windows 7.

Install the SDK and tools

The Service Fabric Tools are part of the Azure Development workload in Visual Studio 2017. Enable this workload as part of your Visual Studio installation. In addition, you need to install the Microsoft Azure Service Fabric SDK and runtime using Web Platform Installer.

- **Install the Microsoft Azure Service Fabric SDK**[2]

For Visual Studio 2015, the Service Fabric tools are installed together with the SDK and runtime using the Web Platform Installer:

- **Install the Microsoft Azure Service Fabric SDK and Tools**[3]

If you only need the SDK, you can install this package:

- **Install the Microsoft Azure Service Fabric SDK**[4]

Enable PowerShell script execution

Service Fabric uses Windows PowerShell scripts for creating a local development cluster and for deploying applications from Visual Studio. By default, Windows blocks these scripts from running. To enable them, you must modify your PowerShell execution policy. Open PowerShell as an administrator and enter the following command:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force -Scope CurrentUser
```

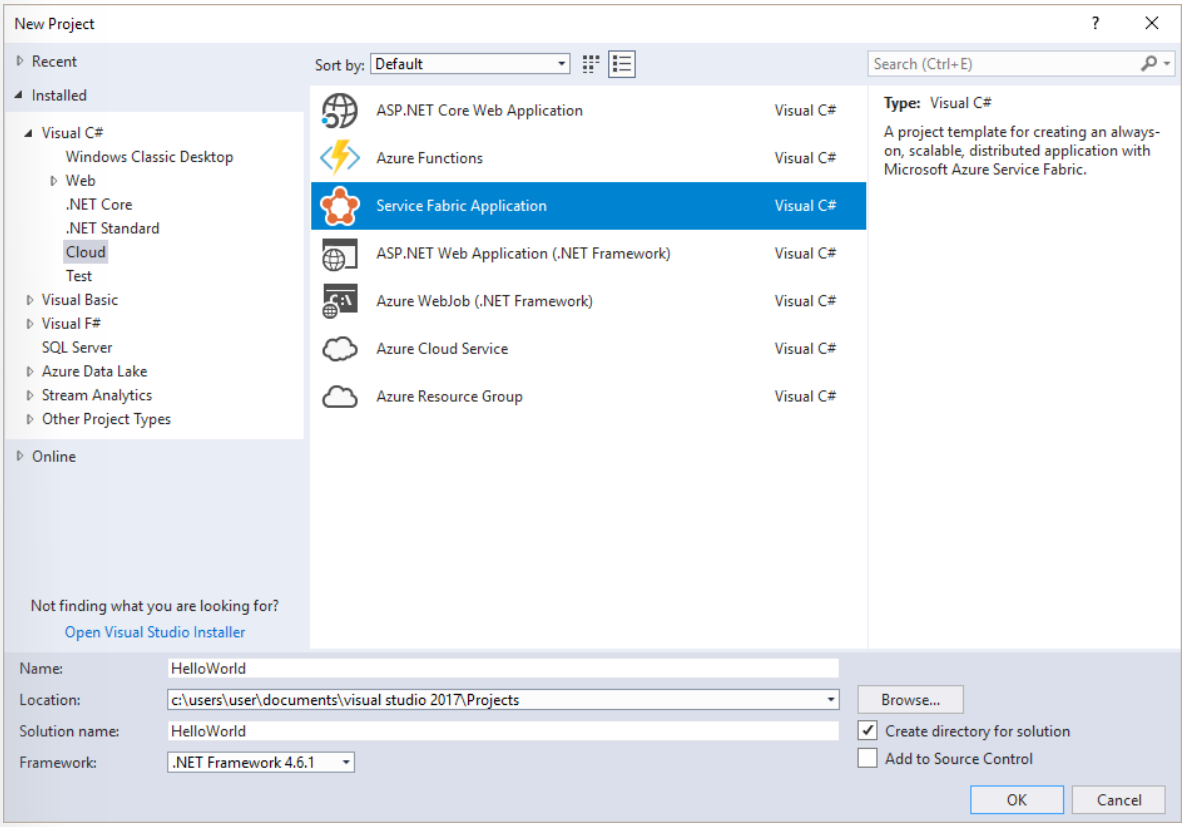
1 https://www.microsoft.com/en-us/download/details.aspx?id=50395
2 http://www.microsoft.com/web/handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric-CoreSDK
3 http://www.microsoft.com/web/handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric-VS2015
4 http://www.microsoft.com/web/handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric-CoreSDK

Creating a reliable service

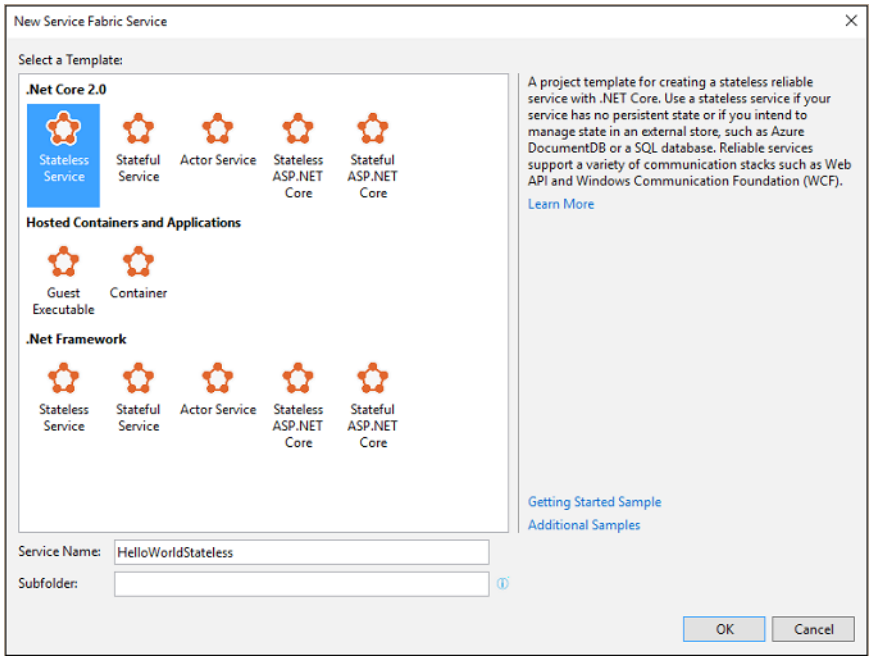
Creating a stateless service in Visual Studio

A stateless service is a type of service that is currently the norm in cloud applications. It is considered stateless because the service itself does not contain data that needs to be stored reliably or made highly available. If an instance of a stateless service shuts down, all of its internal state is lost. In this type of service, state must be persisted to an external store, such as Azure Tables or a SQL database, for it to be made highly available and reliable.

Launch Visual Studio 2015 or Visual Studio 2017 as an administrator, and create a new Service Fabric application project named *HelloWorld*:



Then create a stateless service project using **.Net Core 2.0** named *HelloWorldStateless*:



Your solution now contains two projects:

- *HelloWorld*. This is the application project that contains your services. It also contains the application manifest that describes the application, as well as a number of PowerShell scripts that help you to deploy your application.
- *HelloWorldStateless*. This is the service project. It contains the stateless service implementation.

Implement the service

Open the **HelloWorldStateless.cs** file in the service project. In Service Fabric, a service can run any business logic. The service API provides two entry points for your code:

- An open-ended entry point method, called *RunAsync*, where you can begin executing any workloads, including long-running compute workloads.

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    ...
}
```

- A communication entry point where you can plug in your communication stack of choice, such as ASP.NET Core. This is where you can start receiving requests from users and other services.

```
protected override IEnumerable<ServiceInstanceListener> CreateServiceIn-
stanceListeners()
{
    ...
}
```

We will be focusing on the `RunAsync()` entry point method. This is where you can immediately start running your code. The project template includes a sample implementation of `RunAsync()` that increments a rolling count.

Note: For details about how to work with a communication stack, see **Service Fabric Web API services with OWIN self-hosting**[5]

RunAsync

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    //       or remove this RunAsync override if it's not needed in your
service.

    long iterations = 0;

    while (true)
    {
        cancellationToken.ThrowIfCancellationRequested();

        ServiceEventSource.Current.ServiceMessage(this.Context, "Work-
ing-{0}", ++iterations);

        await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    }
}
```

The platform calls this method when an instance of a service is placed and ready to execute. For a stateless service, that simply means when the service instance is opened. A cancellation token is provided to coordinate when your service instance needs to be closed. In Service Fabric, this open/close cycle of a service instance can occur many times over the lifetime of the service as a whole. This can happen for various reasons, including:

- The system moves your service instances for resource balancing.
- Faults occur in your code.
- The application or system is upgraded.
- The underlying hardware experiences an outage.

This orchestration is managed by the system to keep your service highly available and properly balanced.

`RunAsync()` should not block synchronously. Your implementation of `RunAsync` should return a Task or await on any long-running or blocking operations to allow the runtime to continue. Note in the `while(true)` loop in the previous example, a Task-returning `await Task.Delay()` is used. If your workload must block synchronously, you should schedule a new Task with `Task.Run()` in your `RunAsync` implementation.

Cancellation of your workload is a cooperative effort orchestrated by the provided cancellation token. The system will wait for your task to end (by successful completion, cancellation, or fault) before it moves

5 https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-communication-webapi

on. It is important to honor the cancellation token, finish any work, and exit `RunAsync()` as quickly as possible when the system requests cancellation.

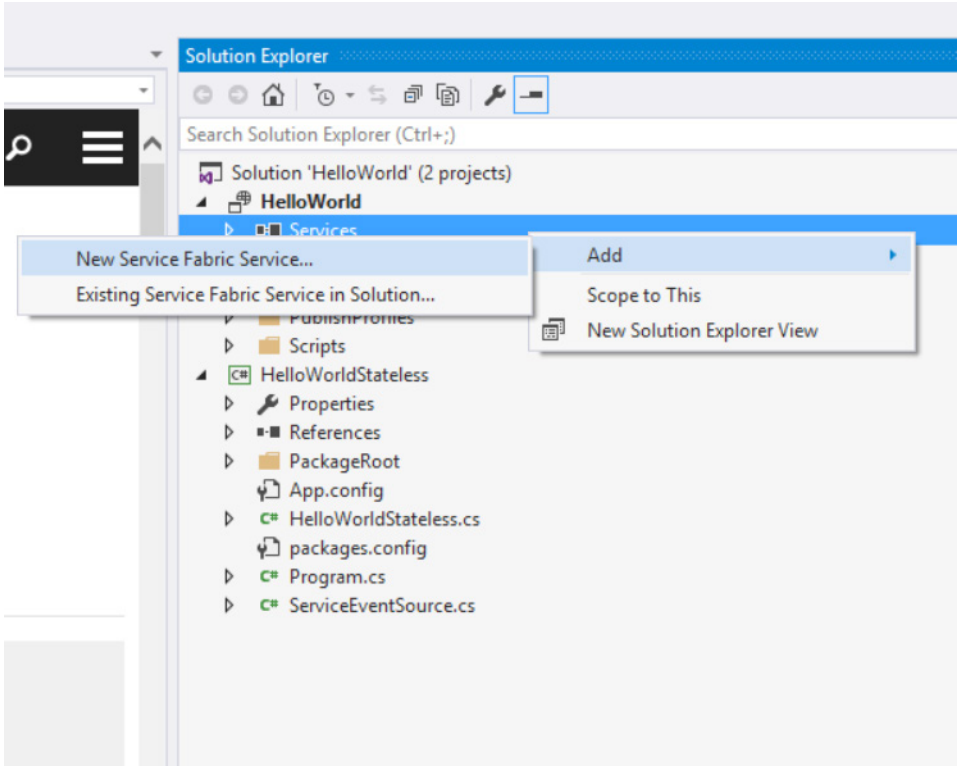
In this stateless service example, the count is stored in a local variable. But because this is a stateless service, the value that's stored exists only for the current lifecycle of its service instance. When the service moves or restarts, the value is lost.

Creating a stateful service in Visual Studio

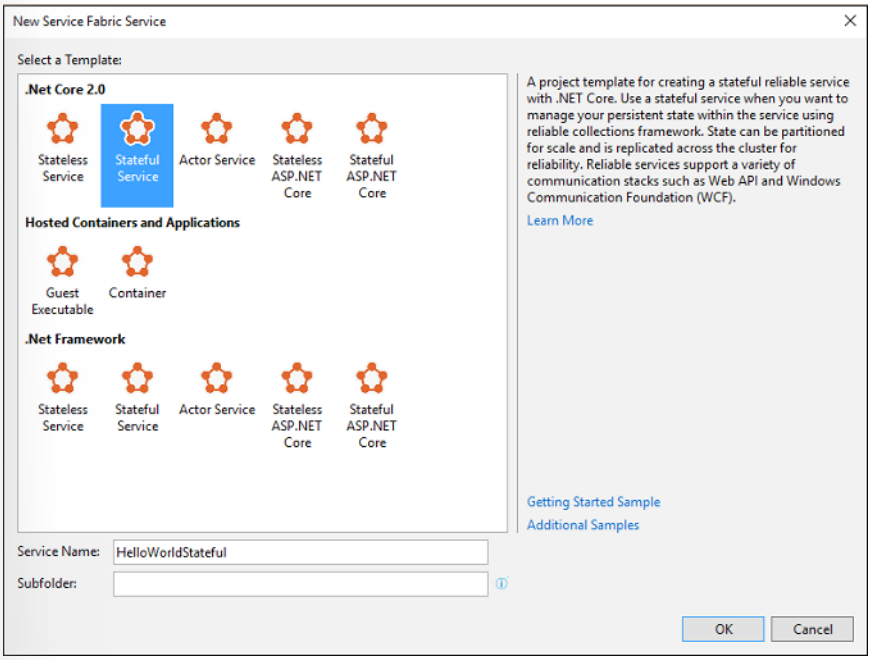
Service Fabric introduces a new kind of service that is stateful. A stateful service can maintain state reliably within the service itself, co-located with the code that's using it. State is made highly available by Service Fabric without the need to persist state to an external store.

To convert a counter value from stateless to highly available and persistent, even when the service moves or restarts, you need a stateful service.

In the same HelloWorld application, you can add a new service by right-clicking on the Services references in the application project and selecting **Add -> New Service Fabric Service**.



Select **.Net Core 2.0 -> Stateful Service** and name it *HelloWorldStateful*. Click **OK**.



Your application should now have two services: the stateless service *HelloWorldStateless* and the stateful service *HelloWorldStateful*.

A stateful service has the same entry points as a stateless service. The main difference is the availability of a *state provider* that can store state reliably. Service Fabric comes with a state provider implementation called Reliable Collections, which lets you create replicated data structures through the Reliable State Manager. A stateful Reliable Service uses this state provider by default.

Open **HelloWorldStateful.cs** in *HelloWorldStateful*, which contains the following `RunAsync` method:

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    //       or remove this RunAsync override if it's not needed in your
service.

    var myDictionary = await this.StateManager.GetOrAddAsync<IReliableDic-
tionary<string, long>>("myDictionary");

    while (true)
    {
        cancellationToken.ThrowIfCancellationRequested();

        using (var tx = this.StateManager.CreateTransaction())
        {
            var result = await myDictionary.TryGetValueAsync(tx, "Coun-
ter");

            ServiceEventSource.Current.ServiceMessage(this.Context, "Cur-
rent Counter Value: {0}",
                result.HasValue ? result.Value.ToString() : "Value does not
exist.");
```

```
            await myDictionary.AddOrUpdateAsync(tx, "Counter", 0, (key,
value) => ++value);

            // If an exception is thrown before calling CommitAsync, the
transaction aborts, all
            // changes are discarded, and nothing is saved to the secondary
replicas.
            await tx.CommitAsync();
        }

        await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    }
```

RunAsync

`RunAsync()` operates similarly in stateful and stateless services. However, in a stateful service, the platform performs additional work on your behalf before it executes `RunAsync()`. This work can include ensuring that the Reliable State Manager and Reliable Collections are ready to use.

Reliable Collections and the Reliable State Manager

```
var myDictionary = await this.StateManager.GetOrAddAsync<IReliableDiction-
ary<string, long>>("myDictionary");
```

IReliableDictionary is a dictionary implementation that you can use to reliably store state in the service. With Service Fabric and Reliable Collections, you can store data directly in your service without the need for an external persistent store. Reliable Collections make your data highly available. Service Fabric accomplishes this by creating and managing multiple replicas of your service for you. It also provides an API that abstracts away the complexities of managing those replicas and their state transitions.

Reliable Collections can store any .NET type, including your custom types, with a couple of caveats:

- Service Fabric makes your state highly available by replicating state across nodes, and Reliable Collections store your data to local disk on each replica. This means that everything that is stored in Reliable Collections must be serializable. By default, Reliable Collections use DataContract for serialization, so it's important to make sure that your types are supported by the Data Contract Serializer when you use the default serializer.
- Objects are replicated for high availability when you commit transactions on Reliable Collections. Objects stored in Reliable Collections are kept in local memory in your service. This means that you have a local reference to the object.
- It is important that you do not mutate local instances of those objects without performing an update operation on the reliable collection in a transaction. This is because changes to local instances of objects will not be replicated automatically. You must re-insert the object back into the dictionary or use one of the update methods on the dictionary.

The Reliable State Manager manages Reliable Collections for you. You can simply ask the Reliable State Manager for a reliable collection by name at any time and at any place in your service. The Reliable State Manager ensures that you get a reference back. We don't recommended that you save references to reliable collection instances in class member variables or properties. Special care must be taken to ensure

that the reference is set to an instance at all times in the service lifecycle. The Reliable State Manager handles this work for you, and it's optimized for repeat visits.

Transactional and asynchronous operations

```
using (ITransaction tx = this.StateManager.CreateTransaction())
{
    var result = await myDictionary.TryGetValueAsync(tx, "Counter-1");

    await myDictionary.AddOrUpdateAsync(tx, "Counter-1", 0, (k, v) => ++v);

    await tx.CommitAsync();
}
```

Reliable Collections have many of the same operations that their `System.Collections.Generic` and `System.Collections.Concurrent` counterparts do, except LINQ. Operations on Reliable Collections are asynchronous. This is because write operations with Reliable Collections perform I/O operations to replicate and persist data to disk.

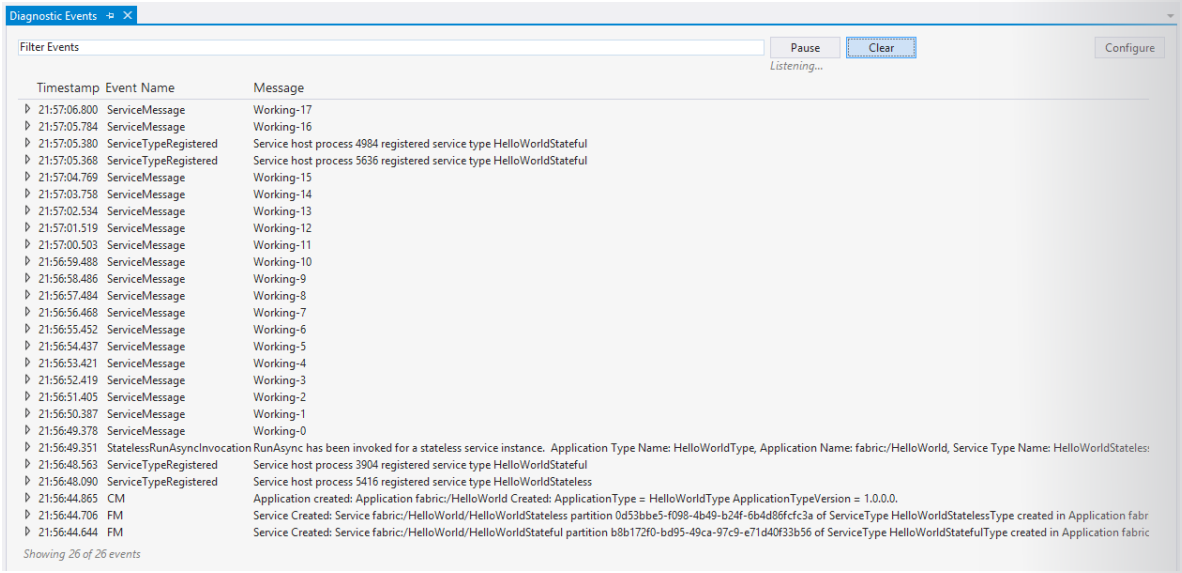
Reliable Collection operations are transactional, so that you can keep state consistent across multiple Reliable Collections and operations. For example, you may dequeue a work item from a Reliable Queue, perform an operation on it, and save the result in a Reliable Dictionary, all within a single transaction. This is treated as an atomic operation, and it guarantees that either the entire operation will succeed or the entire operation will roll back. If an error occurs after you dequeue the item but before you save the result, the entire transaction is rolled back and the item remains in the queue for processing.

Run the application

We now return to the *HelloWorld* application. You can now build and deploy your services. When you press **F5**, your application will be built and deployed to your local cluster.

After the services start running, you can view the generated Event Tracing for Windows (ETW) events in a **Diagnostic Events** window. Note that the events displayed are from both the stateless service and the stateful service in the application. You can pause the stream by clicking the **Pause** button. You can then examine the details of a message by expanding that message.

Note: Before you run the application, make sure that you have a local development cluster running.



Creating a Reliable Actors app

Introduction to Service Fabric Reliable Actors

Reliable Actors is a Service Fabric application framework based on the **Virtual Actor**[6] pattern. The Reliable Actors API provides a single-threaded programming model built on the scalability and reliability guarantees provided by Service Fabric.

What are Actors?

An actor is an isolated, independent unit of compute and state with single-threaded execution. The actor pattern is a computational model for concurrent or distributed systems in which a large number of these actors can execute simultaneously and independently of each other. Actors can communicate with each other and they can create more actors.

When to use Reliable Actors

Service Fabric Reliable Actors is an implementation of the actor design pattern. As with any software design pattern, the decision whether to use a specific pattern is made based on whether or not a software design problem fits the pattern.

Although the actor design pattern can be a good fit to a number of distributed systems problems and scenarios, careful consideration of the constraints of the pattern and the framework implementing it must be made. As general guidance, consider the actor pattern to model your problem or scenario if:

- Your problem space involves a large number (thousands or more) of small, independent, and isolated units of state and logic.
- You want to work with single-threaded objects that do not require significant interaction from external components, including querying state across a set of actors.
- Your actor instances won't block callers with unpredictable delays by issuing I/O operations.

Actors in Service Fabric

In Service Fabric, actors are implemented in the Reliable Actors framework: An actor-pattern-based application framework built on top of Service Fabric Reliable Services. Each Reliable Actor service you write is actually a partitioned, stateful Reliable Service.

Every actor is defined as an instance of an actor type, identical to the way a .NET object is an instance of a .NET type. For example, there may be an actor type that implements the functionality of a calculator and there could be many actors of that type that are distributed on various nodes across a cluster. Each such actor is uniquely identified by an actor ID.

Actor Lifetime

Service Fabric actors are virtual, meaning that their lifetime is not tied to their in-memory representation. As a result, they do not need to be explicitly created or destroyed. The Reliable Actors runtime automatically activates an actor the first time it receives a request for that actor ID. If an actor is not used for a period of time, the Reliable Actors runtime garbage-collects the in-memory object. It will also maintain

6 http://research.microsoft.com/en-us/projects/orleans/

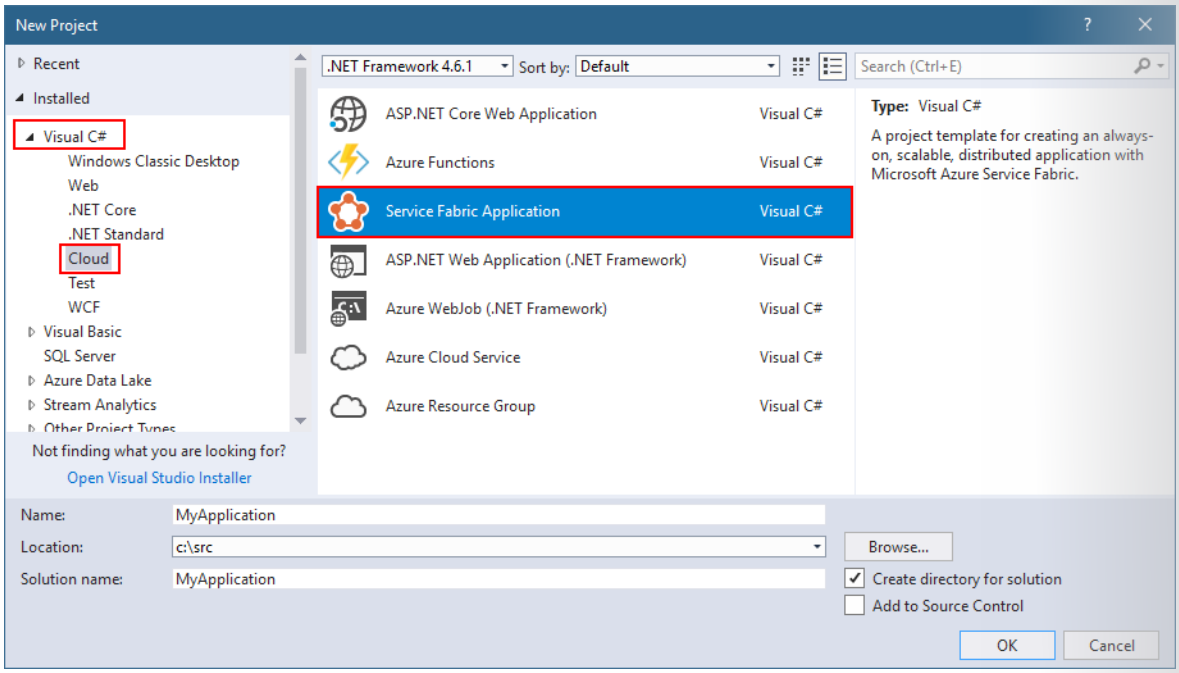
knowledge of the actor's existence should it need to be reactivated later. For more details, see **Actor lifecycle and garbage collection**[7].

This virtual actor lifetime abstraction carries some caveats as a result of the virtual actor model, and in fact the Reliable Actors implementation deviates at times from this model.

- An actor is automatically activated (causing an actor object to be constructed) the first time a message is sent to its actor ID. After some period of time, the actor object is garbage collected. In the future, using the actor ID again, causes a new actor object to be constructed. An actor's state outlives the object's lifetime when stored in the state manager.
- Calling any actor method for an actor ID activates that actor. For this reason, actor types have their constructor called implicitly by the runtime. Therefore, client code cannot pass parameters to the actor type's constructor, although parameters may be passed to the actor's constructor by the service itself. The result is that actors may be constructed in a partially-initialized state by the time other methods are called on it, if the actor requires initialization parameters from the client. There is no single entry point for the activation of an actor from the client.
- Although Reliable Actors implicitly create actor objects; you do have the ability to explicitly delete an actor and its state.

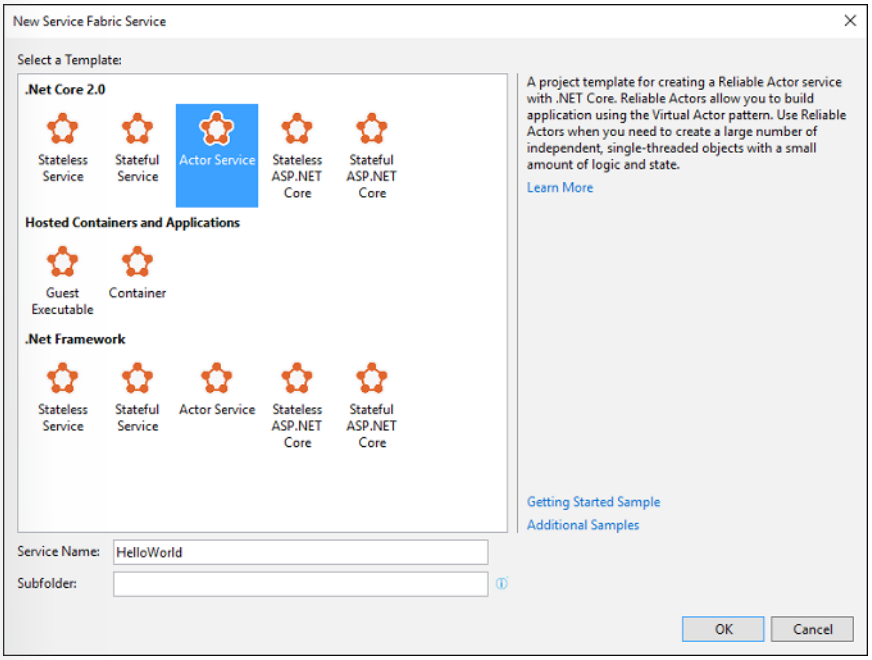
Creating the project in Visual Studio

Launch Visual Studio 2015 or later as an administrator, and then create a new **Service Fabric Application** project:

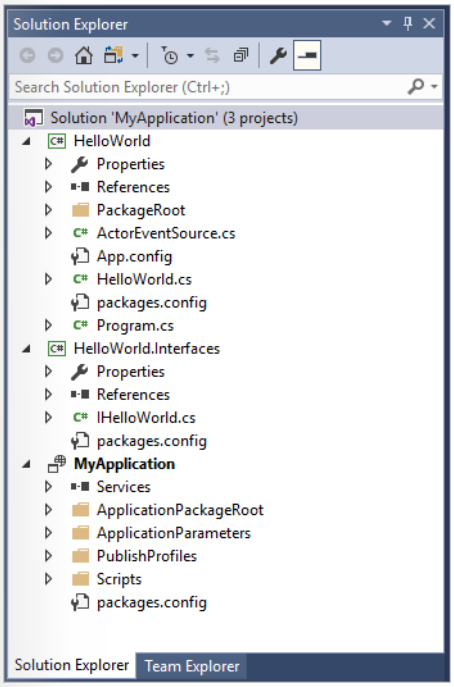


In the next dialog box, choose **Actor Service** under **.Net Core 2.0** and enter a name for the service.

7 https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-lifecycle



The created project shows the following structure:



Examine the solution

The solution contains three projects:

- **The application project (MyApplication).** This project packages all of the services together for deployment. It contains the ApplicationManifest.xml and PowerShell scripts for managing the application.

- **The interface project (HelloWorld.Interfaces).** This project contains the interface definition for the actor. Actor interfaces can be defined in any project with any name. The interface defines the actor contract that is shared by the actor implementation and the clients calling the actor. Because client projects may depend on it, it typically makes sense to define it in an assembly that is separate from the actor implementation.
- **The actor service project (HelloWorld).** This project defines the Service Fabric service that is going to host the actor. It contains the implementation of the actor, *HellowWorld.cs*. An actor implementation is a class that derives from the base type `Actor` and implements the interfaces defined in the *MyActor.Interfaces* project. An actor class must also implement a constructor that accepts an `ActorService` instance and an `ActorId` and passes them to the base `Actor` class.
- This project also contains *Program.cs*, which registers actor classes with the Service Fabric runtime using `ActorRuntime.RegisterActorAsync<T>()`. The `HelloWorld` class is already registered. Any additional actor implementations added to the project must also be registered in the `Main()` method.

Customzing the actor

The project template defines some methods in the `IHelloWorld` interface and implements them in the `HelloWorld` actor implementation. Replace those methods so the actor service returns a simple "Hello World" string.

In the *HelloWorld.Interfaces* project, in the *IHelloWorld.cs* file, replace the interface definition as follows:

```
public interface IHelloWorld : IActor
{
    Task<string> GetHelloWorldAsync();
}
```

In the HelloWorld project, in HelloWorld.cs, replace the entire class definition as follows:

```
[StatePersistence(StatePersistence.Persisted)]
internal class HelloWorld : Actor, IHelloWorld
{
    public HelloWorld(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task<string> GetHelloWorldAsync()
    {
        return Task.FromResult("Hello from my reliable actor!");
    }
}
```

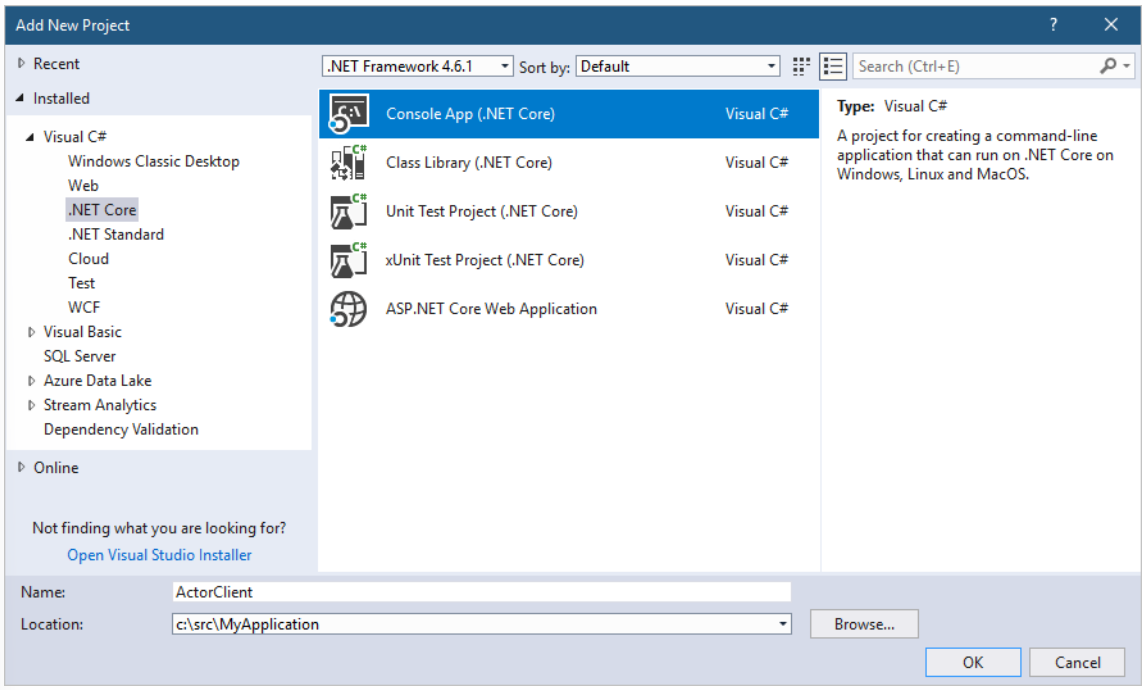
Press **Ctrl-Shift-B** to build the project and make sure everything compiles.

Adding a client

Create a simple console application to call the actor service.

1. Right-click on the solution in **Solution Explorer > Add > New Project....**

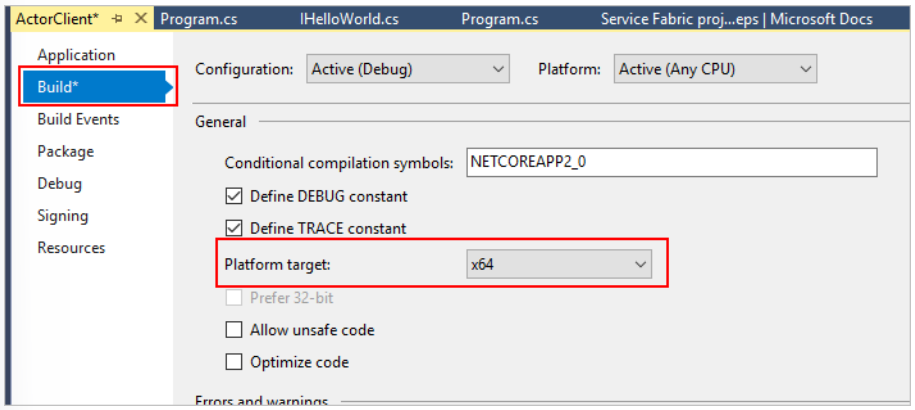
2. Under the **.NET Core** project types, choose **Console App (.NET Core)**. Name the project ActorClient.



3.

4. **Note:** A console application is not the type of app you would typically use as a client in Service Fabric, but it makes a convenient example for debugging and testing using the local Service Fabric cluster.

5. The console application must be a 64-bit application to maintain compatibility with the interface project and other dependencies. In Solution Explorer, right-click the **ActorClient** project, and then click **Properties**. On the **Build** tab, set **Platform target** to **x64**.



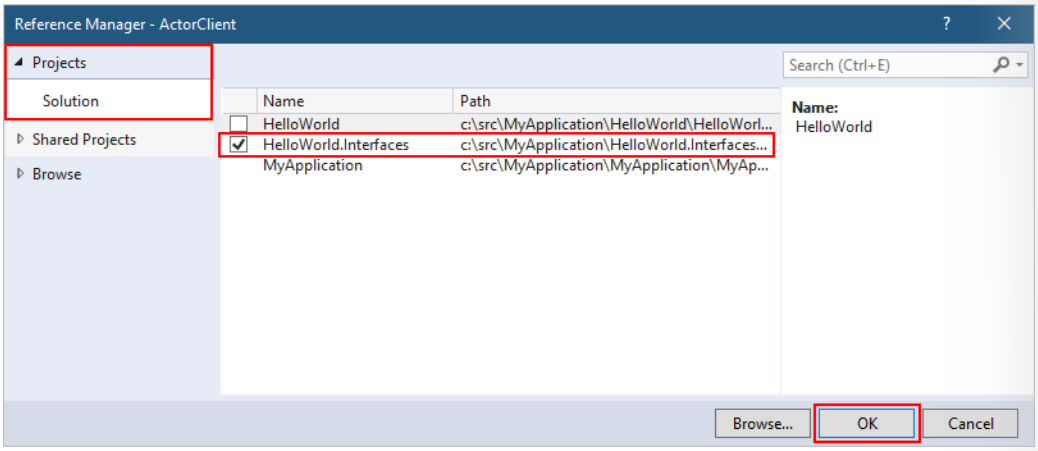
6.

7. The client project requires the reliable actors NuGet package. Click **Tools > NuGet Package Manager > Package Manager Console**. In the Package Manager Console, enter the following command:

```
Install-Package Microsoft.ServiceFabric.Actors -IncludePrerelease -Project-
Name ActorClient
```

8. The NuGet package and all its dependencies are installed in the ActorClient project.

9. The client project also requires a reference to the interfaces project. In the ActorClient project, right-click **Dependencies** and then click **Add reference...**. Select **Projects > Solution** (if not already selected), and then tick the checkbox next to **HelloWorld.Interfaces**. Click **OK**.



10.

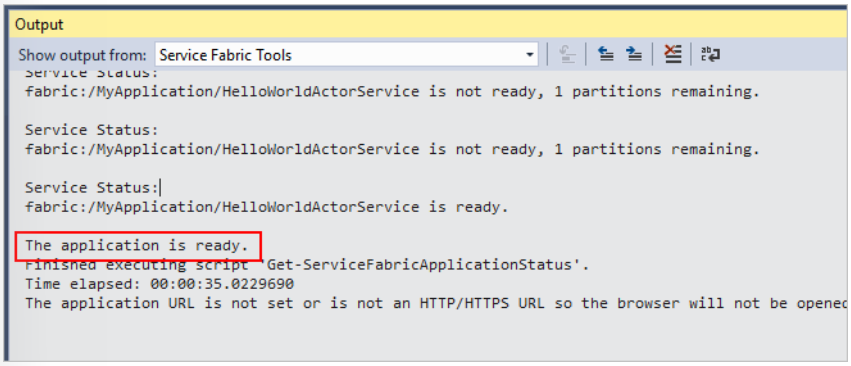
11. In the ActorClient project, replace the entire contents of *Program.cs* with the following code:

```
using System;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Actors;
using Microsoft.ServiceFabric.Actors.Client;
using HelloWorld.Interfaces;

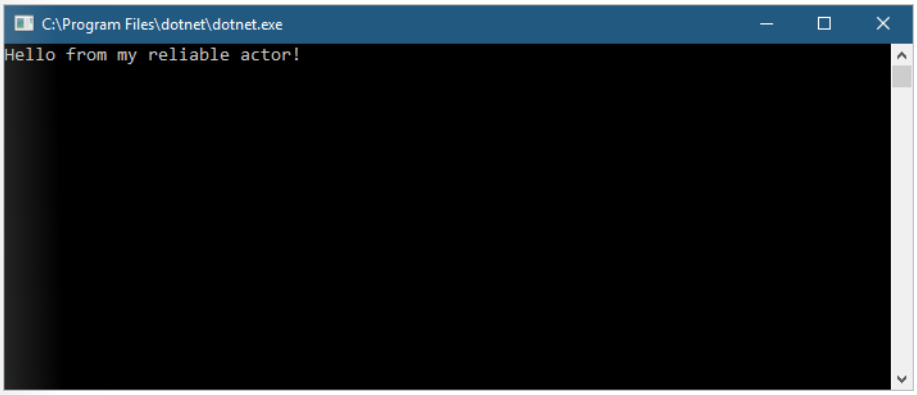
namespace ActorClient
{
    class Program
    {
        static void Main(string[] args)
        {
            IHelloWorld actor = ActorProxy.Create<IHelloWorld>(ActorId.
CreateRandom(), new Uri("fabric:/MyApplication/HelloWorldActorService"));
            Task<string> retval = actor.GetHelloWorldAsync();
            Console.Write(retval.Result);
            Console.ReadLine();
        }
    }
}
```

Running and debugging

Press **F5** to build, deploy, and run the application locally in the Service Fabric development cluster. During the deployment process, you can see the progress in the **Output** window.



When the output contains the text, *The application is ready*, it's possible to test the service using the ActorClient application. In **Solution Explorer**, right-click on the **ActorClient** project, then click **Debug > Start new instance**. The command line application should display the output from the actor service.



The Service Fabric Actors runtime emits some **events and performance counters related to actor methods**[8]. They are useful in diagnostics and performance monitoring.

8 https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-diagnostics#actor-method-events-and-performance-counters

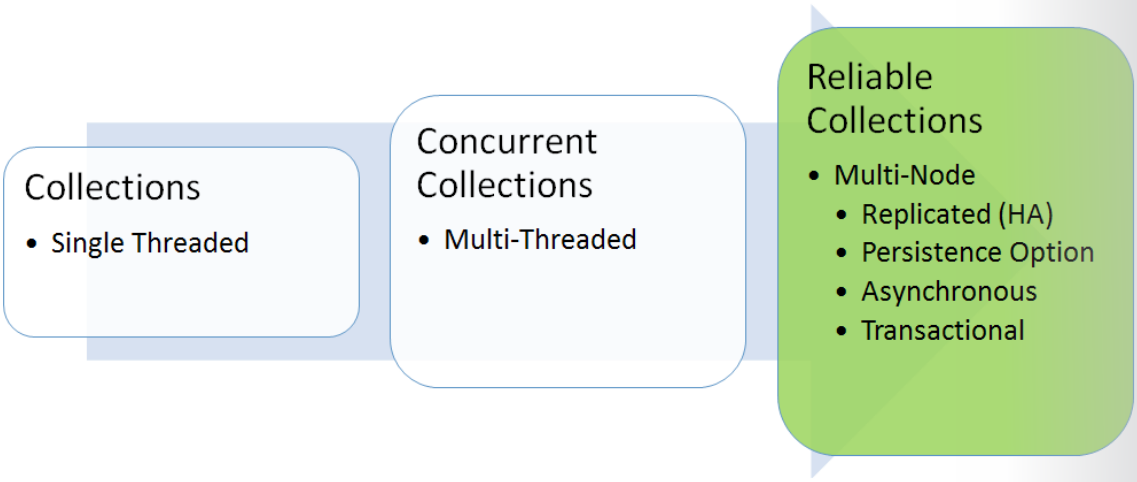
Working with Reliable Collections

Reliable Collections overview

Reliable Collections enable you to write highly available, scalable, and low-latency cloud applications as though you were writing single computer applications. The classes in the **Microsoft.ServiceFabric.Data.Collections** namespace provide a set of collections that automatically make your state highly available. Developers need to program only to the Reliable Collection APIs and let Reliable Collections manage the replicated and local state.

The key difference between Reliable Collections and other high-availability technologies (such as Redis, Azure Table service, and Azure Queue service) is that the state is kept locally in the service instance while also being made highly available. This means that:

- All reads are local, which results in low latency and high-throughput reads.
- All writes incur the minimum number of network IOs, which results in low latency and high-throughput writes.



Reliable Collections can be thought of as the natural evolution of the **System.Collections** classes: a new set of collections that are designed for the cloud and multi-computer applications without increasing complexity for the developer. As such, Reliable Collections are:

- Replicated: State changes are replicated for high availability.
- Persisted: Data is persisted to disk for durability against large-scale outages (for example, a datacenter power outage).
- Asynchronous: APIs are asynchronous to ensure that threads are not blocked when incurring IO.
- Transactional: APIs utilize the abstraction of transactions so you can manage multiple Reliable Collections within a service easily.

Reliable Collections provide strong consistency guarantees out of the box to make reasoning about application state easier. Strong consistency is achieved by ensuring transaction commits finish only after the entire transaction has been logged on a majority quorum of replicas, including the primary. To achieve weaker consistency, applications can acknowledge back to the client/requester before the asynchronous commit returns.

The Reliable Collections APIs are an evolution of concurrent collections APIs (found in the **System.Collections.Concurrent** namespace):

- Asynchronous: Returns a task since, unlike concurrent collections, the operations are replicated and persisted.
- No out parameters: Uses `ConditionalValue<T>` to return a bool and a value instead of out parameters. `ConditionalValue<T>` is like `Nullable<T>` but does not require T to be a struct.
- Transactions: Uses a transaction object to enable the user to group actions on multiple Reliable Collections in a transaction.

Today, Microsoft.ServiceFabric.Data.Collections contains three collections:

- Reliable Dictionary: Represents a replicated, transactional, and asynchronous collection of key/value pairs. Similar to ConcurrentDictionary, both the key and the value can be of any type.
- Reliable Queue: Represents a replicated, transactional, and asynchronous strict first-in, first-out (FIFO) queue. Similar to ConcurrentQueue, the value can be of any type.
- Reliable Concurrent Queue: Represents a replicated, transactional, and asynchronous best effort ordering queue for high throughput. Similar to the **ConcurrentQueue**, the value can be of any type.

Working with Reliable Collections

Service Fabric offers a stateful programming model available to .NET developers via Reliable Collections. Specifically, Service Fabric provides reliable dictionary and reliable queue classes. When you use these classes, your state is partitioned (for scalability), replicated (for availability), and transacted within a partition (for ACID semantics). Let's look at a typical usage of a reliable dictionary object and see what its actually doing.

```
///retry:

try {
   // Create a new Transaction object for this partition
   using (ITransaction tx = base.StateManager.CreateTransaction()) {
      // AddAsync takes key's write lock; if >4 secs, TimeoutException
      // Key & value put in temp dictionary (read your own writes),
      // serialized, redo/undo record is logged & sent to
      // secondary replicas
      await m_dic.AddAsync(tx, key, value, cancellationToken);

      // CommitAsync sends Commit record to log & secondary replicas
      // After quorum responds, all locks released
      await tx.CommitAsync();
   }
   // If CommitAsync not called, Dispose sends Abort
   // record to log & all locks released
}
catch (TimeoutException) {
   await Task.Delay(100, cancellationToken); goto retry;
}
```

All operations on reliable dictionary objects (except for ClearAsync which is not undoable), require an `ITransaction` object. This object has associated with it any and all changes you're attempting to make

to any reliable dictionary and/or reliable queue objects within a single partition. You acquire an `ITransaction` object by calling the partition's StateManager's CreateTransaction method.

In the code above, the `ITransaction` object is passed to a reliable dictionary's `AddAsync` method. Internally, dictionary methods that accepts a key take a reader/writer lock associated with the key. If the method modifies the key's value, the method takes a write lock on the key and if the method only reads from the key's value, then a read lock is taken on the key. Since `AddAsync` modifies the key's value to the new, passed-in value, the key's write lock is taken. So, if 2 (or more) threads attempt to add values with the same key at the same time, one thread will acquire the write lock and the other threads will block. By default, methods block for up to 4 seconds to acquire the lock; after 4 seconds, the methods throw a *TimeoutException*. Method overloads exist allowing you to pass an explicit timeout value if you'd prefer.

Usually, you write your code to react to a *TimeoutException* by catching it and retrying the entire operation (as shown in the code above). In the sample code, we're just calling `Task.Delay` passing 100 milliseconds each time. But, in reality, you might be better off using some kind of exponential back-off delay instead.

Once the lock is acquired, `AddAsync` adds the key and value object references to an internal temporary dictionary associated with the ITransaction object. This is done to provide you with read-your-own-writes semantics. That is, after you call `AddAsync`, a later call to TryGetValueAsync (using the same ITransaction object) will return the value even if you have not yet committed the transaction. Next, `AddAsync` serializes your key and value objects to byte arrays and appends these byte arrays to a log file on the local node. Finally, `AddAsync` sends the byte arrays to all the secondary replicas so they have the same key/value information. Even though the key/value information has been written to a log file, the information is not considered part of the dictionary until the transaction that they are associated with has been committed.

In the code above, the call to `CommitAsync` commits all of the transaction's operations. Specifically, it appends commit information to the log file on the local node and also sends the commit record to all the secondary replicas. Once a quorum (majority) of the replicas has replied, all data changes are considered permanent and any locks associated with keys that were manipulated via the `ITransaction` object are released so other threads/transactions can manipulate the same keys and their values.

If `CommitAsync` is not called (usually due to an exception being thrown), then the `ITransaction` object gets disposed. When disposing an uncommitted ITransaction object, Service Fabric appends abort information to the local node's log file and nothing needs to be sent to any of the secondary replicas. And then, any locks associated with keys that were manipulated via the transaction are released.

Common pitfalls and how to avoid them

Now that you understand how the reliable collections work internally, let's take a look at some common misuses of them. See the code below:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // AddAsync serializes the name/user, logs the bytes,
   // & sends the bytes to the secondary replicas.
   await m_dic.AddAsync(tx, name, user);

   // The line below updates the property's value in memory only; the
   // new value is NOT serialized, logged, & sent to secondary replicas.
   user.LastLogin = DateTime.UtcNow;  // Corruption!

   await tx.CommitAsync();
}
```

When working with a regular .NET dictionary, you can add a key/value to the dictionary and then change the value of a property (such as LastLogin). However, this code will not work correctly with a reliable dictionary. Remember from the earlier discussion, the call to `AddAsync` serializes the key/value objects to byte arrays and then saves the arrays to a local file and also sends them to the secondary replicas. If you later change a property, this changes the property's value in memory only; it does not impact the local file or the data sent to the replicas. If the process crashes, what's in memory is thrown away. When a new process starts or if another replica becomes primary, then the old property value is what is available.

The correct way to write the code is simply to reverse the two lines:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   user.LastLogin = DateTime.UtcNow;  // Do this BEFORE calling AddAsync
   await m_dic.AddAsync(tx, name, user);
   await tx.CommitAsync();
}
```

Here is another example showing a common mistake:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // Use the user's name to look up their data
   ConditionalValue<User> user =
      await m_dic.TryGetValueAsync(tx, name);

   // The user exists in the dictionary, update one of their properties.
   if (user.HasValue) {
      // The line below updates the property's value in memory only; the
      // new value is NOT serialized, logged, & sent to secondary replicas.
      user.Value.LastLogin = DateTime.UtcNow; // Corruption!
      await tx.CommitAsync();
   }
}
```

Again, with regular .NET dictionaries, the code above works fine and is a common pattern: the developer uses a key to look up a value. If the value exists, the developer changes a property's value. However, with reliable collections, this code exhibits the same problem as already discussed: **you MUST not modify an object once you have given it to a reliable collection**.

The correct way to update a value in a reliable collection, is to get a reference to the existing value and consider the object referred to by this reference immutable. Then, create a new object which is an exact copy of the original object. Now, you can modify the state of this new object and write the new object into the collection so that it gets serialized to byte arrays, appended to the local file and sent to the replicas. After committing the change(s), the in-memory objects, the local file, and all the replicas have the same exact state. All is good!

The code below shows the correct way to update a value in a reliable collection:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // Use the user's name to look up their data
   ConditionalValue<User> currentUser =
      await m_dic.TryGetValueAsync(tx, name);

   // The user exists in the dictionary, update one of their properties.
   if (currentUser.HasValue) {
      // Create new user object with the same state as the current user
```

```
object.
      // NOTE: This must be a deep copy; not a shallow copy. Specifically,
only
      // immutable state can be shared by currentUser & updatedUser object
graphs.
      User updatedUser = new User(currentUser);

      // In the new object, modify any properties you desire
      updatedUser.LastLogin = DateTime.UtcNow;

      // Update the key's value to the updateUser info
      await m_dic.SetValue(tx, name, updatedUser);

      await tx.CommitAsync();
   }
}
```

Define immutable data types to prevent programmer error

Ideally, we'd like the compiler to report errors when you accidentally produce code that mutates state of an object that you are supposed to consider immutable. But, the C# compiler does not have the ability to do this. So, to avoid potential programmer bugs, we highly recommend that you define the types you use with reliable collections to be immutable types. Specifically, this means that you stick to core value types (such as numbers [Int32, UInt64, etc.], DateTime, Guid, TimeSpan, and the like). And, of course, you can also use String. It is best to avoid collection properties as serializing and deserializing them can frequently can hurt performance. However, if you want to use collection properties, we highly recommend the use of .NET's immutable collections library (`System.Collections.Immutable`). This library is available for download from **http://nuget.org**. We also recommend sealing your classes and making fields read-only whenever possible.

The UserInfo type below demonstrates how to define an immutable type taking advantage of aforementioned recommendations.

```
[DataContract]
// If you don't seal, you must ensure that any derived classes are also
immutable
public sealed class UserInfo {
   private static readonly IEnumerable<ItemId> NoBids = ImmutableList<Item-
Id>.Empty;

   public UserInfo(String email, IEnumerable<ItemId> itemsBidding = null) {
      Email = email;
      ItemsBidding = (itemsBidding == null) ? NoBids : itemsBidding.ToIm-
mutableList();
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      // Convert the deserialized collection to an immutable collection
      ItemsBidding = ItemsBidding.ToImmutableList();
```

```
    }

    [DataMember]
    public readonly String Email;

    // Ideally, this would be a readonly field but it can't be because OnDe-
serialized
    // has to set it. So instead, the getter is public and the setter is
private.
    [DataMember]
    public IEnumerable<ItemId> ItemsBidding { get; private set; }

    // Since each UserInfo object is immutable, we add a new ItemId to the
ItemsBidding
    // collection by creating a new immutable UserInfo object with the added
ItemId.
    public UserInfo AddItemBidding(ItemId itemId) {
       return new UserInfo(Email, ((ImmutableList<ItemId>)ItemsBidding).
Add(itemId));
    }
}
```

The `ItemId` type is also an immutable type as shown here:

```
[DataContract]
public struct ItemId {

    [DataMember] public readonly String Seller;
    [DataMember] public readonly String ItemName;
    public ItemId(String seller, String itemName) {
       Seller = seller;
       ItemName = itemName;
    }
}
```

Schema versioning (upgrades)

Internally, Reliable Collections serialize your objects using .NET's DataContractSerializer. The serialized objects are persisted to the primary replica's local disk and are also transmitted to the secondary replicas. As your service matures, it's likely you'll want to change the kind of data (schema) your service requires. You must approach versioning of your data with great care. First and foremost, you must always be able to deserialize old data. Specifically, this means your deserialization code must be infinitely backward compatible: Version 333 of your service code must be able to operate on data placed in a reliable collection by version 1 of your service code 5 years ago.

Furthermore, service code is upgraded one upgrade domain at a time. So, during an upgrade, you have two different versions of your service code running simultaneously. You must avoid having the new version of your service code use the new schema as old versions of your service code might not be able to handle the new schema. When possible, you should design each version of your service to be forward compatible by 1 version. Specifically, this means that V1 of your service code should be able to simply

ignore any schema elements it does not explicitly handle. However, it must be able to save any data it doesn't explicitly know about and simply write it back out when updating a dictionary key or value.

Warning: While you can modify the schema of a key, you must ensure that your key's hash code and equals algorithms are stable. If you change how either of these algorithms operate, you will not be able to look up the key within the reliable dictionary ever again.

Alternatively, you can perform what is typically referred to as a 2-phase upgrade. With a 2-phase upgrade, you upgrade your service from V1 to V2: V2 contains the code that knows how to deal with the new schema change but this code doesn't execute. When the V2 code reads V1 data, it operates on it and writes V1 data. Then, after the upgrade is complete across all upgrade domains, you can somehow signal to the running V2 instances that the upgrade is complete. (One way to signal this is to roll out a configuration upgrade; this is what makes this a 2-phase upgrade.) Now, the V2 instances can read V1 data, convert it to V2 data, operate on it, and write it out as V2 data. When other instances read V2 data, they do not need to convert it, they just operate on it, and write out V2 data.

Review Questions

Module 2 Review Questions

Service Fabric

You manage a customer facing web application for your company. The web application was originally deployed in an Azure virtual machine (VM). As demand for the application grew, your development team created containers for increased performance.

The application must now serve even more users.

What should you consider next?

Suggested Answer ↓

Service Fabric is tailored to create cloud native services that can start small, as needed, and grow to massive scale with hundreds or thousands of machines.

Service Fabric is a microservices platform that gives every microservice (or container) a unique name that can be either stateless or stateful.

Service Fabric provides comprehensive runtime and lifecycle management capabilities to applications that are composed of these microservices. It hosts microservices inside containers that are deployed and activated across the Service Fabric cluster.

Stateful Microservices in Service Fabric

You are designing a customer-facing web application for your company. The web application uses Azure Service Fabric.

The application must be highly available. You decide to use a stateful microservices design.

What is the benefit of using this type of design?

Suggested Answer ↓

Stateful microservices maintain their authoritative state beyond the request and its response. This provides high availability and consistency of the state through simple APIs that provide transactional guarantees backed by replication. Service Fabric's stateful services democratize high availability, bringing it to all types of applications, not just databases and other data stores.

Stateful microservices simplify application designs because they remove the need for the additional queues and caches that have traditionally been required to address the availability and latency requirements of purely stateless applications. Since stateful services are naturally highly available and low latency, this means that there are fewer moving parts to manage in your application as a whole.

Stateless Microservice

You are designing a customer-facing web application for your company. The web application uses Azure Service Fabric.

You decide to use a stateful microservices design.

What happens with data in the application if the service is restarted or interrupted?

Suggested Answer ↓

In a stateless service the data is stored in a local variable. The value that is stored exists only for the current lifecycle of its service instance.

Module 3 Module Using Azure Kubernetes Service

Creating an Azure Kubernetes Service Cluster

What is Azure Kubernetes Service?

Kubernetes is a rapidly evolving platform that manages container-based applications and their associated networking and storage components. The focus is on the application workloads, not the underlying infrastructure components. Kubernetes provides a declarative approach to deployments, backed by a robust set of APIs for management operations.

You can build and run modern, portable, microservices-based applications that benefit from Kubernetes orchestrating and managing the availability of those application components. Kubernetes supports both stateless and stateful applications as teams progress through the adoption of microservices-based applications.

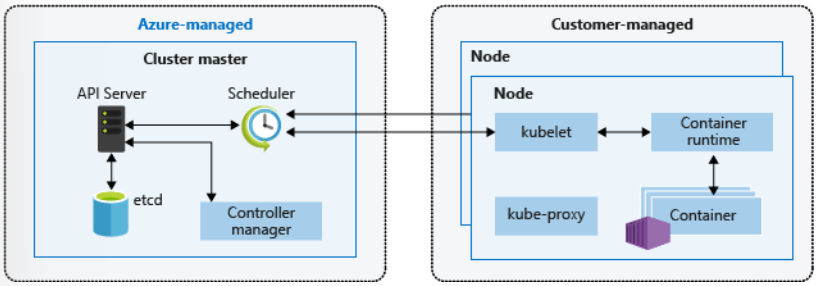
As an open platform, Kubernetes allows you to build your applications with your preferred programming language, OS, libraries, or messaging bus. Existing continuous integration and continuous delivery (CI/CD) tools can integrate with Kubernetes to schedule and deploy releases.

Azure Kubernetes Service (AKS) provides a managed Kubernetes service that reduces the complexity for deployment and core management tasks, including coordinating upgrades. The AKS cluster masters are managed by the Azure platform, and you only pay for the AKS nodes that run your applications. AKS is built on top of the open-source Azure Container Service Engine (acs-engine).

Kubernetes cluster architecture

A Kubernetes cluster is divided into two components:

- *Cluster master* nodes provide the core Kubernetes services and orchestration of application workloads.
- *Nodes* run your application workloads.



Cluster master

When you create an AKS cluster, a cluster master is automatically created and configured. This cluster master is provided as a managed Azure resource abstracted from the user. There is no cost for the cluster master, only the nodes that are part of the AKS cluster.

The cluster master includes the following core Kubernetes components:

- **kube-apiserver** - The API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as `kubectl` or the Kubernetes dashboard.
- **etcd** - To maintain the state of your Kubernetes cluster and configuration, the highly available *etcd* is a key value store within Kubernetes.
- **kube-scheduler** - When you create or scale applications, the Scheduler determines what nodes can run the workload and starts them.
- **kube-controller-manager** - The Controller Manager oversees a number of smaller Controllers that perform actions such as replicating pods and handling node operations.

AKS provides a single-tenant cluster master, with a dedicated API server, Scheduler, etc. You define the number and size of the nodes, and the Azure platform configures the secure communication between the cluster master and nodes. Interaction with the cluster master occurs through Kubernetes APIs, such as `kubectl` or the Kubernetes dashboard.

This managed cluster master means that you do not need to configure components like a highly available etcd store, but it also means that you cannot access the cluster master directly. Upgrades to Kubernetes are orchestrated through the Azure CLI or Azure portal, which upgrades the cluster master and then the nodes. To troubleshoot possible issues, you can review the cluster master logs through Azure Log Analytics.

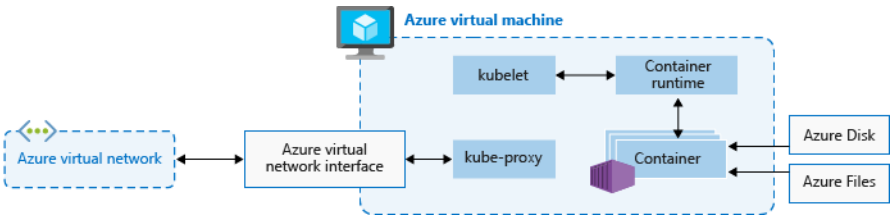
If you need to configure the cluster master in a particular way or need direct access to them, you can deploy your own Kubernetes cluster using `aks-engine`.

Nodes and node pools

To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has one or more nodes, which is an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime:

- The `kubelet` is the Kubernetes agent that processes the orchestration requests from the cluster master and scheduling of running the requested containers.
- Virtual networking is handled by the *kube-proxy* on each node. The proxy routes network traffic and manages IP addressing for services and pods.

- The *container runtime* is the component that allows containerized applications to run and interact with additional resources such as the virtual network and storage. In AKS, Docker is used as the container runtime.



The Azure VM size for your nodes defines how many CPUs, how much memory, and the size and type of storage available (such as high-performance SSD or regular HDD). If you anticipate a need for applications that require large amounts of CPU and memory or high-performance storage, plan the node size accordingly. You can also scale up the number of nodes in your AKS cluster to meet demand.

In AKS, the VM image for the nodes in your cluster is currently based on Ubuntu Linux. When you create an AKS cluster or scale up the number of nodes, the Azure platform creates the requested number of VMs and configures them. There is no manual configuration for you to perform.

If you need to use a different host OS, container runtime, or include custom packages, you can deploy your own Kubernetes cluster using `aks-engine`. The upstream `aks-engine` releases features and provide configuration options before they are officially supported in AKS clusters. For example, if you wish to use Windows containers or a container runtime other than Docker, you can use `aks-engine` to configure and deploy a Kubernetes cluster that meets your current needs.

Resource reservations

You don't need to manage the core Kubernetes components on each node, such as the *kubelet*, *kube-proxy*, and *kube-dns*, but they do consume some of the available compute resources. To maintain node performance and functionality, the following compute resources are reserved on each node:

- **CPU** - 60ms
- **Memory** - 20% up to 4 GiB

These reservations mean that the amount of available CPU and memory for your applications may appear less than the node itself contains. If there are resource constraints due to the number of applications that you run, these reservations ensure CPU and memory remains available for the core Kubernetes components. The resource reservations cannot be changed.

For example:

- **Standard DS2 v2** node size contains 2 vCPU and 7 GiB memory
  - 20% of 7 GiB memory = 1.4 GiB
  - A total of (7 - 1.4) = 5.6 GiB memory is available for the node
- **Standard E4s v3** node size contains 4 vCPU and 32 GiB memory
  - 20% of 32 GiB memory = 6.4 GiB, but AKS only reserves a maximum of 4 GiB
  - A total of (32 - 4) = 28 GiB is available for the node

The underlying node OS also requires some amount of CPU and memory resources to complete its own core functions.

Node pools

Nodes of the same configuration are grouped together into *node pools*. A Kubernetes cluster contains one or more node pools. The initial number of nodes and size are defined when you create an AKS cluster, which creates a *default node pool*. This default node pool in AKS contains the underlying VMs that run your agent nodes.

When you scale or upgrade an AKS cluster, the action is performed against the default node pool. For upgrade operations, running containers are scheduled on other nodes in the node pool until all the nodes are successfully upgraded.

Pods

Kubernetes uses *pods* to run an instance of your application. A pod represents a single instance of your application. Pods typically have a 1:1 mapping with a container, although there are advanced scenarios where a pod may contain multiple containers. These multi-container pods are scheduled together on the same node, and allow containers to share related resources.

When you create a pod, you can define *resource limits* to request a certain amount of CPU or memory resources. The Kubernetes Scheduler tries to schedule the pods to run on a node with available resources to meet the request. You can also specify maximum resource limits that prevent a given pod from consuming too much compute resource from the underlying node. A best practice is to include resource limits for all pods to help the Kubernetes Scheduler understand what resources are needed and permitted.

A pod is a logical resource, but the container(s) are where the application workloads run. Pods are typically ephemeral, disposable resources, and individually scheduled pods miss some of the high availability and redundancy features Kubernetes provides. Instead, pods are usually deployed and managed by Kubernetes *Controllers*, such as the Deployment Controller.

Deployments and YAML manifests

A deployment represents one or more identical pods, managed by the Kubernetes Deployment Controller. A deployment defines the number of replicas (pods) to create, and the Kubernetes Scheduler ensures that if pods or nodes encounter problems, additional pods are scheduled on healthy nodes.

You can update deployments to change the configuration of pods, container image used, or attached storage. The Deployment Controller drains and terminates a given number of replicas, creates replicas from the new deployment definition, and continues the process until all replicas in the deployment are updated.

Most stateless applications in AKS should use the deployment model rather than scheduling individual pods. Kubernetes can monitor the health and status of deployments to ensure that the required number of replicas run within the cluster. When you only schedule individual pods, the pods are not restarted if they encounter a problem, and are not rescheduled on healthy nodes if their current node encounters a problem.

If an application requires a quorum of instances to always be available for management decisions to be made, you don't want an update process to disrupt that ability. *Pod Disruption Budgets* can be used to define how many replicas in a deployment can be taken down during an update or node upgrade. For example, if you have 5 replicas in your deployment, you can define a pod disruption of 4 to only permit one replica from being deleted/rescheduled at a time. As with pod resource limits, a best practice is to define pod disruption budgets on applications that require a minimum number of replicas to always be present.

Deployments are typically created and managed with `kubectl create` or `kubectl apply`. To create a deployment, you define a manifest file in the YAML (YAML Ain't Markup Language) format. The following example creates a basic deployment of the NGINX web server. The deployment specifies 3 replicas to be created, and that port 80 be open on the container. Resource requests and limits are also defined for CPU and memory.

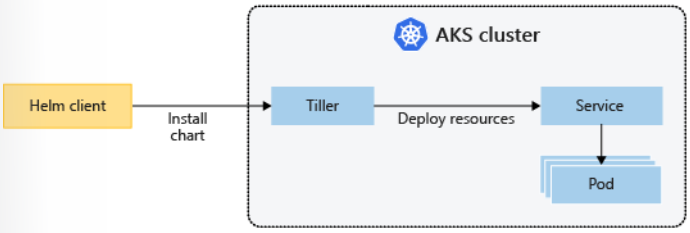
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.2
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 250m
            memory: 64Mi
          limits:
            cpu: 500m
            memory: 256Mi
```

More complex applications can be created by also including services such as load balancers within the YAML manifest.

Package management with Helm

A common approach to managing applications in Kubernetes is with Helm. You can build and use existing public Helm charts that contain a packaged version of application code and Kubernetes YAML manifests to deploy resources. These Helm charts can be stored locally, or often in a remote repository, such as an Azure Container Registry Helm chart repo.

To use Helm, a server component called *Tiller* is installed in your Kubernetes cluster. The Tiller manages the installation of charts within the cluster. The Helm client itself is installed locally on your computer, or can be used within the Azure Cloud Shell. You can search for or create Helm charts with the client, and then install them to your Kubernetes cluster.



For more information, see **Install applications with Helm in Azure Kubernetes Service (AKS)**[1].

StatefulSets and DaemonSets

The Deployment Controller uses the Kubernetes Scheduler to run a given number of replicas on any available node with available resources. This approach of using deployments may be sufficient for stateless applications, but not for applications that require a persistent naming convention or storage. For applications that require a replica to exist on each node, or selected nodes, within a cluster, the Deployment Controller doesn't look at how replicas are distributed across the nodes.

There are two Kubernetes resources that let you manage these types of applications:

- *StatefulSets* - Maintain the state of applications beyond an individual pod lifecycle, such as storage.
- *DaemonSets* - Ensure a running instance on each node, early in the Kubernetes bootstrap process.

StatefulSets

Modern application development often aims for stateless applications, but *StatefulSets* can be used for stateful applications, such as applications that include database components. A StatefulSet is similar to a deployment in that one or more identical pods are created and managed. Replicas in a StatefulSet follow a graceful, sequential approach to deployment, scale, upgrades, and terminations. With a StatefulSet, the naming convention, network names, and storage persist as replicas are rescheduled.

You define the application in YAML format using `kind: StatefulSet`, and the StatefulSet Controller then handles the deployment and management of the required replicas. Data is written to persistent storage, provided by Azure Managed Disks or Azure Files. With StatefulSets, the underlying persistent storage remains even when the StatefulSet is deleted.

Replicas in a StatefulSet are scheduled and run across any available node in an AKS cluster. If you need to ensure that at least one pod in your Set runs on a node, you can instead use a DaemonSet.

DaemonSets

For specific log collection or monitoring needs, you may need to run a given pod on all, or selected, nodes. A DaemonSet is again used to deploy one or more identical pods, but the DaemonSet Controller ensures that each node specified runs an instance of the pod.

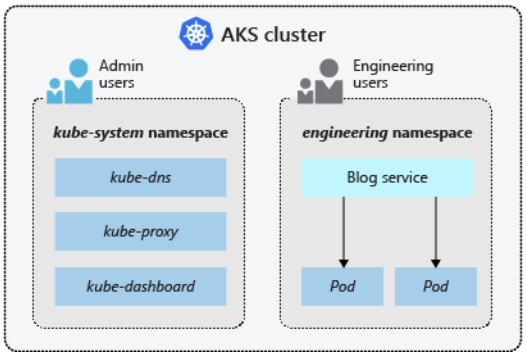
The DaemonSet Controller can schedule pods on nodes early in the cluster boot process, before the default Kubernetes scheduler has started. This ability ensures that the pods in a DaemonSet are started before traditional pods in a Deployment or StatefulSet are scheduled.

Like StatefulSets, a DaemonSet is defined as part of a YAML definition using `kind: DaemonSet`.

1 https://docs.microsoft.com/en-us/azure/aks/kubernetes-helm

Namespaces

Kubernetes resources, such as pods and Deployments, are logically grouped into a *namespace*. These groupings provide a way to logically divide an AKS cluster and restrict access to create, view, or manage resources. You can create namespaces to separate business groups, for example. Users can only interact with resources within their assigned namespaces.



When you create an AKS cluster, the following namespaces are available:

- *default* - This namespace is where pods and deployments are created by default when none is provided. In smaller environments, you can deploy applications directly into the default namespace without creating additional logical separations. When you interact with the Kubernetes API, such as with kubectl get pods, the default namespace is used when none is specified.
- *kube-system* - This namespace is where core resources exist, such as network features like DNS and proxy, or the Kubernetes dashboard. You typically don't deploy your own applications into this namespace.
- *kube-public* - This namespace is typically not used, but can be used for resources to be visible across the whole cluster, and can viewed by any users.

AKS security Concepts for Apps and Clusters

To protect your customer data as you run application workloads in Azure Kubernetes Service (AKS), the security of your cluster is a key consideration. Kubernetes includes security components such as network policies and Secrets. Azure then adds in components such as network security groups and orchestrated cluster upgrades. These security components are combined to keep your AKS cluster running the latest OS security updates and Kubernetes releases, and with secure pod traffic and access to sensitive credentials.

This section introduces the core concepts that secure your applications in AKS:

- Master components security
- Node security
- Cluster upgrades
- Network security
- Kubernetes Secrets

Master security

In AKS, the Kubernetes master components are part of the managed service provided my Microsoft. Each AKS cluster has their own single-tenanted, dedicated Kubernetes master to provide the API Server, Scheduler, etc. This master is managed and maintained by Microsoft

By default, the Kubernetes API server uses a public IP address and with fully qualified domain name (FQDN). You can control access to the API server using Kubernetes role-based access controls and Azure Active Directory.

Node security

AKS nodes are Azure virtual machines that you manage and maintain. The nodes run an optimized Ubuntu Linux distribution with the Docker container runtime. When an AKS cluster is created or scaled up, the nodes are automatically deployed with the latest OS security updates and configurations.

The Azure platform automatically applies OS security patches to the nodes on a nightly basis. If an OS security update requires a host reboot, that reboot is not automatically performed. You can manually reboot the nodes, or a common approach is to use **Kured**[2], an open-source reboot daemon for Kubernetes. Kured runs as a [DaemonSet][aks-daemonset] and monitors each node for the presence of a file indicating that a reboot is required. Reboots are managed across the cluster using the same cordon and drain process as a cluster upgrade.

Nodes are deployed into a private virtual network subnet, with no public IP addresses assigned. For troubleshooting and management purposes, SSH is enabled by default. This SSH access is only available using the internal IP address. Azure network security group rules can be used to further restrict IP range access to the AKS nodes. Deleting the default network security group SSH rule and disabling the SSH service on the nodes prevents the Azure platform from performing maintenance tasks.

To provide storage, the nodes use Azure Managed Disks. For most VM node sizes, these are Premium disks backed by high-performance SSDs. The data stored on managed disks is automatically encrypted at rest within the Azure platform. To improve redundancy, these disks are also securely replicated within the Azure datacenter.

Cluster upgrades

For security and compliance, or to use the latest features, Azure provides tools to orchestrate the upgrade of an AKS cluster and components. This upgrade orchestration includes both the Kubernetes master and agent components. You can view a list of available Kubernetes versions for your AKS cluster. To start the upgrade process, you specify one of these available versions. Azure then safely cordons and drains each AKS node and performs the upgrade.

Cordon and drain

During the upgrade process, AKS nodes are individually cordoned from the cluster so new pods are not scheduled on them. The nodes are then drained and upgraded as follows:

- Existing pods are gracefully terminated and scheduled on remaining nodes.
- The node is rebooted, the upgrade process completed, and then joins back into the AKS cluster.
- Pods are scheduled to run on them again.

2 https://github.com/weaveworks/kured

- The next node in the cluster is cordoned and drained using the same process until all nodes are successfully upgraded.

Network security

For connectivity and security with on-premises networks, you can deploy your AKS cluster into existing Azure virtual network subnets. These virtual networks may have an Azure Site-to-Site VPN or Express Route connection back to your on-premises network. Kubernetes ingress controllers can be defined with private, internal IP addresses so services are only accessible over this internal network connection.

Azure network security groups

To filter the flow of traffic in virtual networks, Azure uses network security group rules. These rules define the source and destination IP ranges, ports, and protocols that are allowed or denied access to resources. Default rules are created to allow TLS traffic to the Kubernetes API server and for SSH access to the nodes. As you create services with load balancers, port mappings, or ingress routes, AKS automatically modifies the network security group for traffic to flow appropriately.

Kubernetes Secrets

A Kubernetes *Secret* is used to inject sensitive data into pods, such as access credentials or keys. You first create a Secret using the Kubernetes API. When you define your pod or deployment, a specific Secret can be requested. Secrets are only provided to nodes that have a scheduled pod that requires it, and the Secret is stored in *tmpfs*, not written to disk. When the last pod on a node that requires a Secret is deleted, the Secret is deleted from the node's tmpfs. Secrets are stored within a given namespace and can only be accessed by pods within the same namespace.

The use of Secrets reduces the sensitive information that is defined in the pod or service YAML manifest. Instead, you request the Secret stored in Kubernetes API Server as part of your YAML manifest. This approach only provides the specific pod access to the Secret.

AKS Access and Identity

There are different ways to authenticate with and secure Kubernetes clusters. Using role-based access controls (RBAC), you can grant users or groups access to only the resources they need. With Azure Kubernetes Service (AKS), you can further enhance the security and permissions structure by using Azure Active Directory. These approaches help you secure your application workloads and customer data.

This section introduces the core concepts that help you authenticate and assign permissions in AKS:

- Kubernetes service accounts
- Azure Active Directory integration
- Role-based access controls (RBAC)
- Roles and ClusterRoles
- RoleBindings and ClusterRoleBindings

Kubernetes service accounts

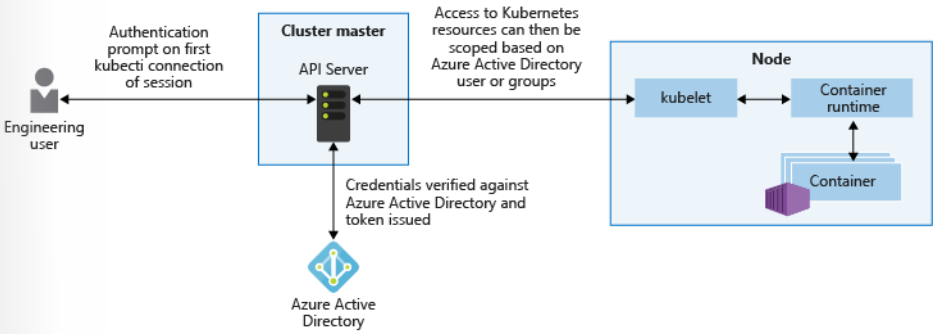
One of the primary user types in Kubernetes is a service account. A service account exists in, and is managed by, the Kubernetes API. The credentials for service accounts are stored as Kubernetes secrets,

which allows them to be used by authorized pods to communicate with the API Server. Most API requests provide an authentication token for a service account or a normal user account.

Normal user accounts allow more traditional access for human administrators or developers, not just services and processes. Kubernetes itself does not provide an identity management solution where regular user accounts and passwords are stored. Instead, external identity solutions can be integrated into Kubernetes. For AKS clusters, this integrated identity solution is Azure Active Directory.

Azure Active Directory integration

The security of AKS clusters can be enhanced with the integration of Azure Active Directory (AD). Built on decades of enterprise identity management, Azure AD is a multi-tenant, cloud-based directory, and identity management service that combines core directory services, application access management, and identity protection. With Azure AD, you can integrate on-premises identities into AKS clusters to provide a single source for account management and security.



With Azure AD-integrated AKS clusters, you can grant users or groups access to Kubernetes resources within a namespace or across the cluster. To obtain a `kubectl` configuration context, a user can run the `az aks get-credentials` command. When a user then interacts with the AKS cluster with `kubectl`, they are prompted to sign in with their Azure AD credentials. This approach provides a single source for user account management and password credentials. The user can only access the resources as defined by the cluster administrator.

Azure AD authentication in AKS clusters uses OpenID Connect, an identity layer built on top of the OAuth 2.0 protocol. OAuth 2.0 defines mechanisms to obtain and use access tokens to access protected resources, and OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. For more information on OpenID Connect, see the **Open ID Connect documentation**[3]. To verify the authentication tokens obtained from Azure AD through OpenID Connect, AKS clusters use Kubernetes Webhook Token Authentication. For more information, see the **Webhook Token Authentication documentation**[4].

Role-based access controls (RBAC)

To provide granular filtering of the actions that users can perform, Kubernetes uses role-based access controls (RBAC). This control mechanism lets you assign users, or groups of users, permission to do things like create or modify resources, or view logs from running application workloads. These permissions can be scoped to a single namespace, or granted across the entire AKS cluster. With Kubernetes RBAC, you create roles to define permissions, and then assign those *roles* to users with *role bindings*.

3 https://docs.microsoft.com/en-us/azure/active-directory/develop/v1-protocols-openid-connect-code
4 https://kubernetes.io/docs/reference/access-authn-authz/authentication/#webhook-token-authentication

Azure role-based access controls (RBAC)

One additional mechanism for controlling access to resources is Azure role-based access controls (RBAC). Kubernetes RBAC is designed to work on resources within your AKS cluster, and Azure RBAC is designed to work on resources within your Azure subscription. With Azure RBAC, you create a *role definition* that outlines the permissions to be applied. A user or group is then assigned this role definition for a particular *scope*, which could be an individual resource, a resource group, or across the subscription.

Roles and ClusterRoles

Before you assign permissions to users with Kubernetes RBAC, you first define those permissions as a *Role*. Kubernetes roles grant permissions. There is no concept of a *deny* permission.

Roles are used to grant permissions within a namespace. If you need to grant permissions across the entire cluster, or to cluster resources outside a given namespace, you can instead use *ClusterRoles*.

A ClusterRole works in the same way to grant permissions to resources, but can be applied to resources across the entire cluster, not a specific namespace.

RoleBindings and ClusterRoleBindings

Once roles are defined to grant permissions to resources, you assign those Kubernetes RBAC permissions with a *RoleBinding*. If your AKS cluster integrates with Azure Active Directory, bindings are how those Azure AD users are granted permissions to perform actions within the cluster.

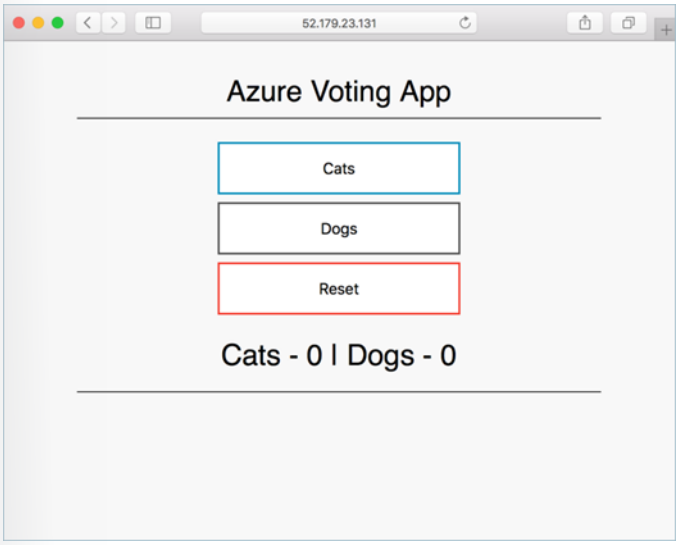
Role bindings are used to assign roles for a given namespace. This approach lets you logically segregate a single AKS cluster, with users only able to access the application resources in their assigned namespace. If you need to bind roles across the entire cluster, or to cluster resources outside a given namespace, you can instead use *ClusterRoleBindings*.

A ClusterRoleBinding works in the same way to bind roles to users, but can be applied to resources across the entire cluster, not a specific namespace. This approach lets you grant administrators or support engineers access to all resources in the AKS cluster.

Deploy an AKS cluster using Azure CLI

In this section of the coruse, an AKS cluster is deployed using the Azure CLI. A multi-container application consisting of web front end and a Redis instance is then run on the cluster. Once completed, the application is accessible over the internet. You'll need a basic understanding of Kubernetes concepts, for detailed information on Kubernetes see the **Kubernetes documentation**[5]. Once completed, the application is accessible over the internet.

5 https://kubernetes.io/docs/home/



Create a resource group

Login to the Azure Portal (https://portal.azure.com) and launch the Azure Cloud Shell.

Create a resource group with the `az group create` command. An Azure resource group is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are asked to specify a location. This location is where your resources run in Azure.

The following example creates a resource group named *myAKSCluster* in the eastus location.

```
az group create --name myAKSCluster --location eastus
```

Output:

```
{
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resource-
Groups/myAKSCluster",
  "location": "eastus",
  "managedBy": null,
  "name": "myAKSCluster",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

Create AKS cluster

Use the `az aks create` command to create an AKS cluster. The following example creates a cluster named myAKSCluster with one node. Container health monitoring is also enabled using the *–enable-addons monitoring* parameter.

```
az aks create --resource-group myAKSCluster --name myAKSCluster --node-
count 1 --enable-addons monitoring --generate-ssh-keys
```

After several minutes, the command completes and returns JSON-formatted information about the cluster.

Connect to the cluster

To manage a Kubernetes cluster, use `kubectl`, the Kubernetes command-line client.

If you're using Azure Cloud Shell, `kubectl` is already installed. If you want to install it locally, use the `az aks install-cli` command.

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. This step downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myAKSCluster --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes. It can take a few minutes for the nodes to appear.

```
kubectl get nodes
```

Output:

```
NAME                              STATUS    ROLES     AGE       VERSION
k8s-myAKSCluster-36346190-0   Ready     agent     2m        v1.7.7
```

Run the application

A Kubernetes manifest file defines a desired state for the cluster, including what container images should be running. For this example, a manifest is used to create all objects needed to run the Azure Vote application. This manifest includes two Kubernetes deployments, one for the Azure Vote Python applications, and the other for a Redis instance. Also, two Kubernetes Services are created, an internal service for the Redis instance, and an external service for accessing the Azure Vote application from the internet.

Create a file named `azure-vote.yaml` and copy into it the following YAML code. If you are working in Azure Cloud Shell, this file can be created using `vi` or `Nano` as if working on a virtual or physical system.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
```

```
      containers:
      - name: azure-vote-back
        image: redis
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 6379
          name: redis
--- #separator
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
  - port: 6379
  selector:
    app: azure-vote-back
--- #separator
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
      - name: azure-vote-front
        image: microsoft/azure-vote-front:v1
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 80
        env:
```

```
        - name: REDIS
          value: "azure-vote-back"
--- #separator
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Use the `kubectl apply` command to run the application.

```
kubectl apply -f azure-vote.yaml
```

Output:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the `kubectl get service` command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

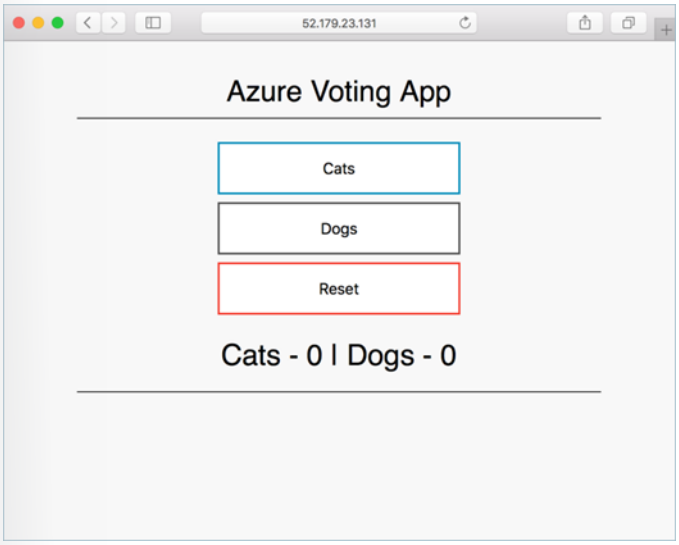
Initially the EXTERNAL-IP for the azure-vote-front service appears as pending.

```
NAME               TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
azure-vote-front   LoadBalancer   10.0.37.27   <pending>     80:30572/TCP
6s
```

Once the EXTERNAL-IP address has changed from pending to an IP address, use CTRL-C to stop the kubectl watch process.

```
azure-vote-front   LoadBalancer   10.0.37.27   52.179.23.131   80:30572/TCP
2m
```

Now browse to the external IP address to see the Azure Vote App.



Monitor health and logs

Monitor health and logs

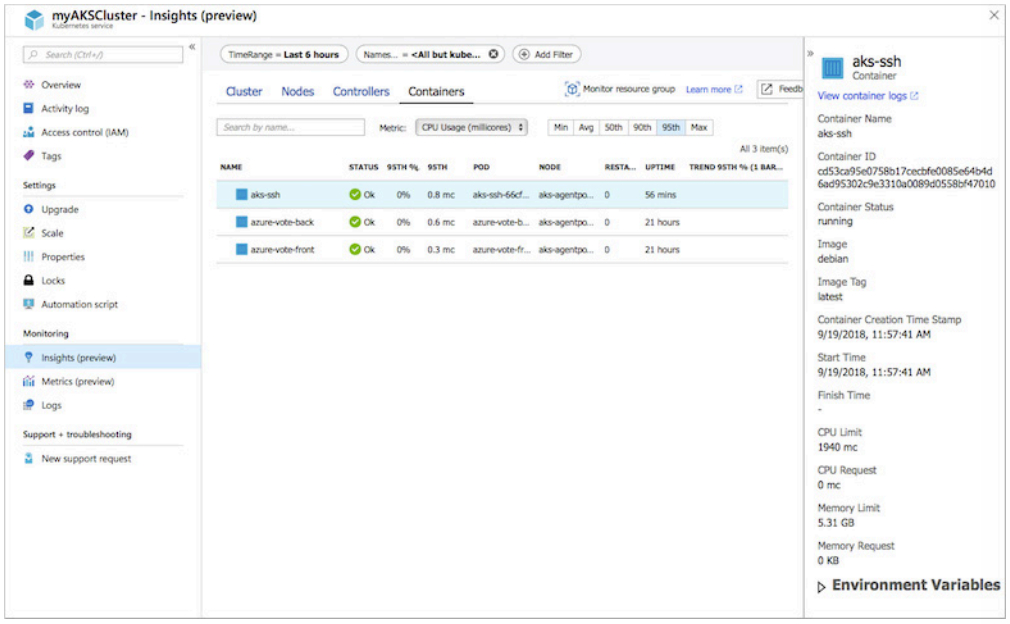
When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see, see **Monitor Azure Kubernetes Service health**[6].

It may take a few minutes for this data to populate in the Azure portal. To see current status, uptime, and resource usage for the Azure Vote pods, complete the following steps:

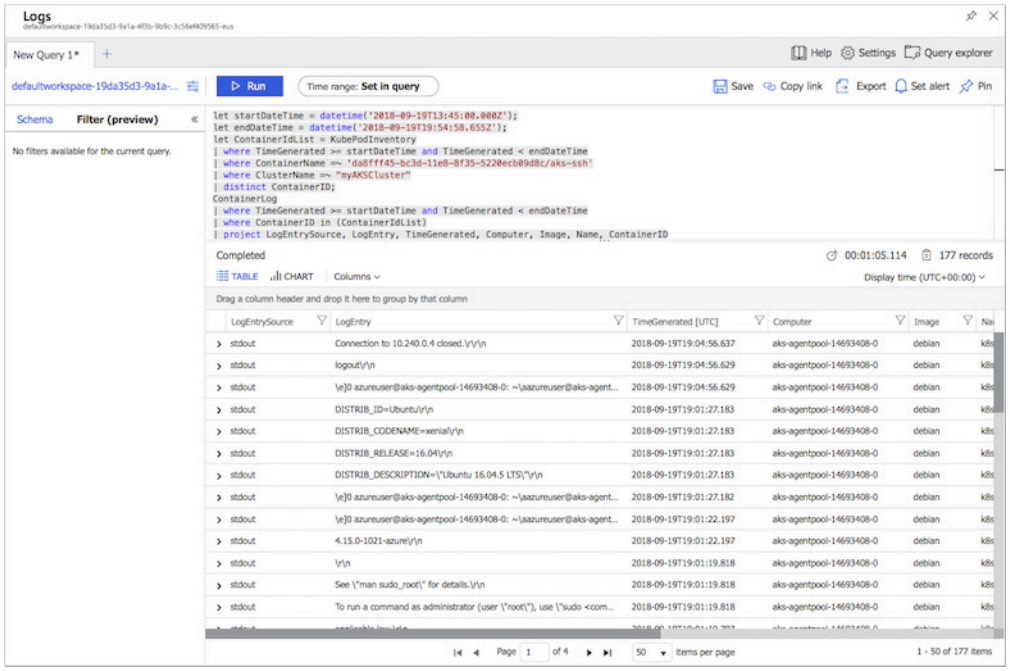
1. Open a web browser to the Azure portal https://portal.azure.com.
2. Select your resource group, such as *myResourceGroup*, then select your AKS cluster, such as *myAKSCluster*.
3. Under **Monitoring** on the left-hand side, choose **Insights (preview)**
4. Across the top, choose to **+ Add Filter**
5. Select **Namespace** as the property, then choose *<All but kube-system>*
6. Choose to view the **Containers**.

The *azure-vote-back* and *azure-vote-front* containers are displayed, as shown in the following example:

6 https://docs.microsoft.com/en-us/azure/monitoring/monitoring-container-health



To see logs for the `azure-vote-front` pod, select the **View container logs** link on the right-hand side of the containers list. These logs include the *stdout* and *stderr* streams from the container.



Delete cluster

When the cluster is no longer needed, delete the cluster resource, which deletes all associated resources. This operation can be completed in the Azure portal by selecting the **Delete** button on the AKS cluster dashboard. Alternatively, the az aks delete command can be used in the Cloud Shell:

```
az aks delete --resource-group myResourceGroup --name myAKSCluster --no-
wait
```

Note: When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see **AKS service principal considerations and deletion**[7].

Get the code

In this tutorial, pre-created container images have been used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

https://github.com/Azure-Samples/azure-voting-app-redis

apiVersion: v1
kind: Service
metadata:
name: azure-vote-back
spec:
ports:

- port: 6379
  selector:
  app: azure-vote-back

apiVersion: apps/v1beta1
kind: Deployment
metadata:
name: azure-vote-front
spec:
replicas: 1
template:
metadata:
labels:
app: azure-vote-front
spec:
containers:
- name: azure-vote-front
image: microsoft/azure-vote-front:v1
ports:
- containerPort: 80
env:
- name: REDIS
value: “azure-vote-back”

apiVersion: v1
kind: Service
metadata:
name: azure-vote-front
spec:

7 https://docs.microsoft.com/en-us/azure/aks/kubernetes-service-principal#additional-considerations

type: LoadBalancer
ports:

- port: 80
  selector:
  app: azure-vote-front

```
Use the `kubectl apply` command to run the application.

```azurecli
kubectl apply -f azure-vote.yaml
```

Output:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the `kubectl get service` command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

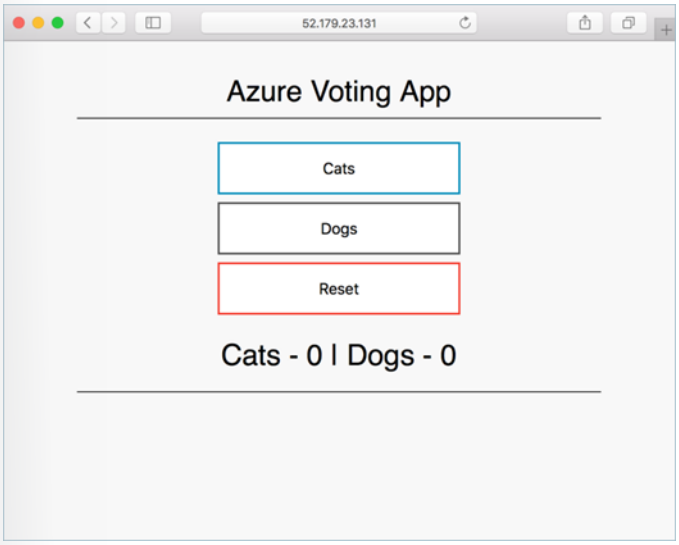
Initially the EXTERNAL-IP for the azure-vote-front service appears as pending.

```
NAME               TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
azure-vote-front   LoadBalancer   10.0.37.27   <pending>     80:30572/TCP
6s
```

Once the EXTERNAL-IP address has changed from pending to an IP address, use CTRL-C to stop the kubectl watch process.

```
azure-vote-front   LoadBalancer   10.0.37.27   52.179.23.131   80:30572/TCP
2m
```

Now browse to the external IP address to see the Azure Vote App.
```



Monitor health and logs

When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see Monitor Azure Kubernetes Service health.

To see current status, uptime, and resource usage for the Azure Vote pods, complete the following steps:

1. Open a web browser to the Azure portal.
2. Select your resource group, such as *myResourceGroup*, then select your AKS cluster, such as *myAKSCluster*.
   3.Choose **Monitor container health >** select the **default** namespace > then select **Containers**.

To see logs for the `azure-vote-front` pod, select the **View Logs** link on the right-hand side of the list of containers. These logs include the *stdout* and *stderr* streams from the container.

Deploy an AKS cluster using Azure Portal

Now, we'll cover deploying an AKS cluster using the Azure portal. The first step is to sign in to the Azure portal: https://portal.azure.com.

Create an AKS cluster

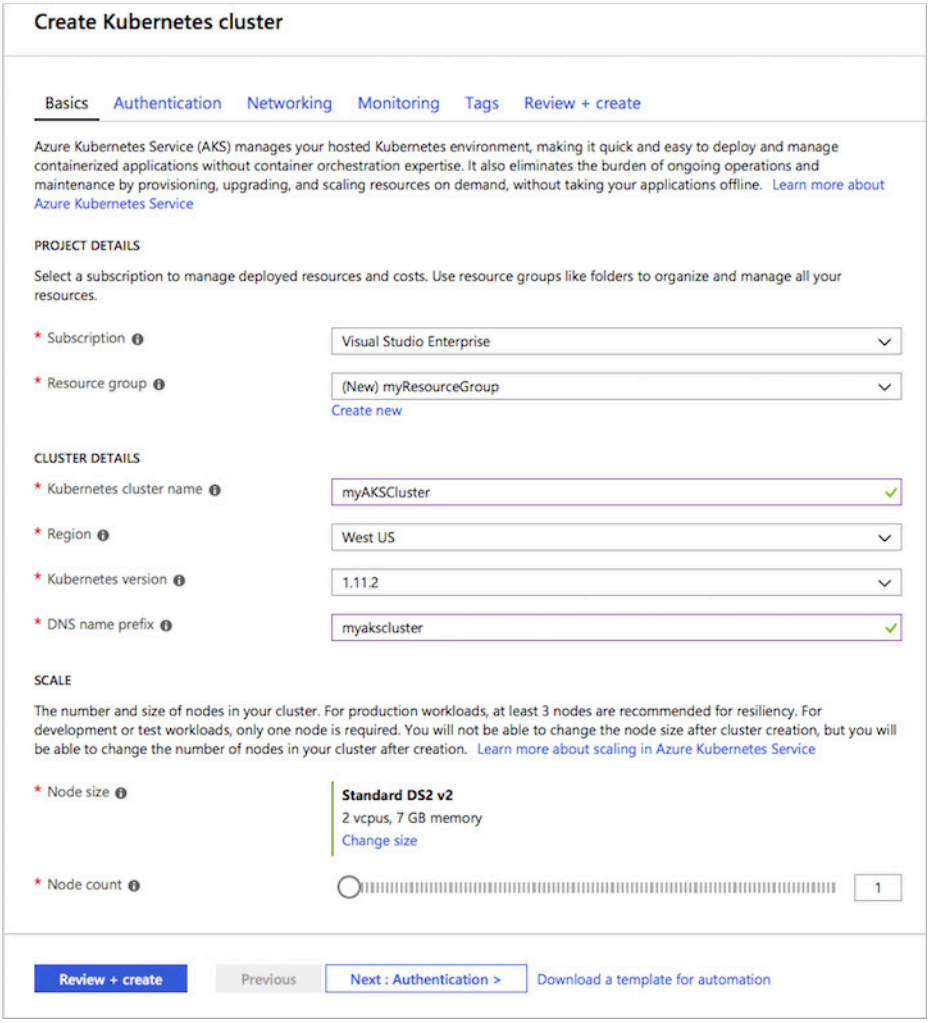
In the top left-hand corner of the Azure portal, select **Create a resource > Kubernetes Service**.

To create an AKS cluster, complete the following steps:

1. **Basics** - Configure the following options:
   - PROJECT DETAILS: Select an Azure subscription, then select or create an Azure resource group, such as *myResourceGroup*. Enter a **Kubernetes cluster name**, such as *myAKSCluster*.
   - CLUSTER DETAILS: Select a region, Kubernetes version, and DNS name prefix for the AKS cluster.

- SCALE: Select a VM size for the AKS nodes. The VM size cannot be changed once an AKS cluster has been deployed.

    - Select the number of nodes to deploy into the cluster. For this tutorial, set **Node count** to 1. Node count can be adjusted after the cluster has been deployed.



1.

2. Select **Next: Authentication** when complete.
3. **Authentication:** Configure the following options:

    - Create a new service principal or *Configure* to use an existing one. When using an existing SPN, you need to provide the SPN client ID and secret.
    - Enable the option for Kubernetes role-based access controls (RBAC). These controls provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

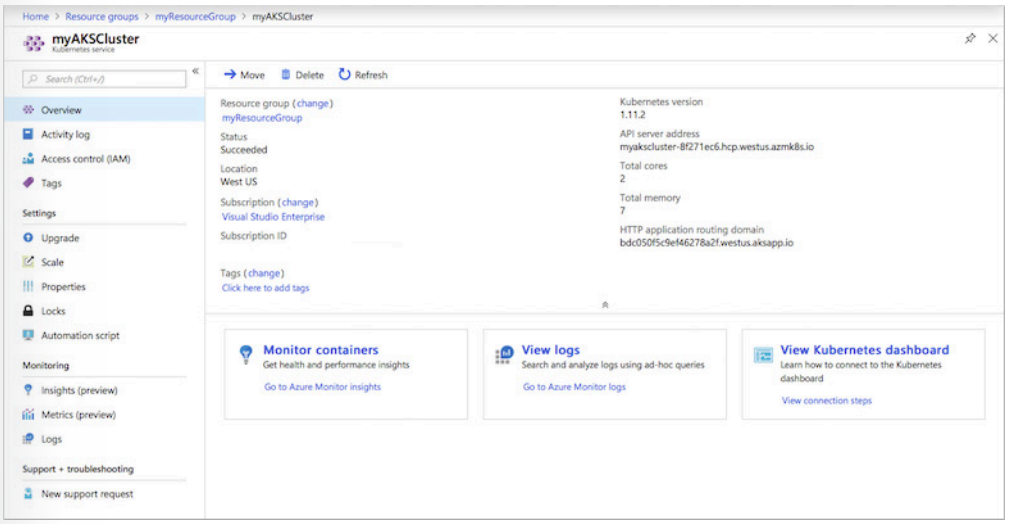
4. Select **Next: Networking** when complete.
5. **Networking:** Configure the following networking options, which should be set as default:

    - **Http application routing** - Select **Yes** to configure an integrated ingress controller with automatic public DNS name creation.

- **Network configuration** - Select the **Basic** network configuration using the `kubenet` Kubernetes plugin, rather than advanced networking configuration using Azure CNI.

6. Select **Next: Monitoring** when complete.
7. When deploying an AKS cluster, Azure Container Insights can be configured to monitor health of the AKS cluster and pods running on the cluster.
   - Select **Yes** to enable container monitoring and select an existing Log Analytics workspace, or create a new one.
   - Select **Review + create** and then **Create** when ready.

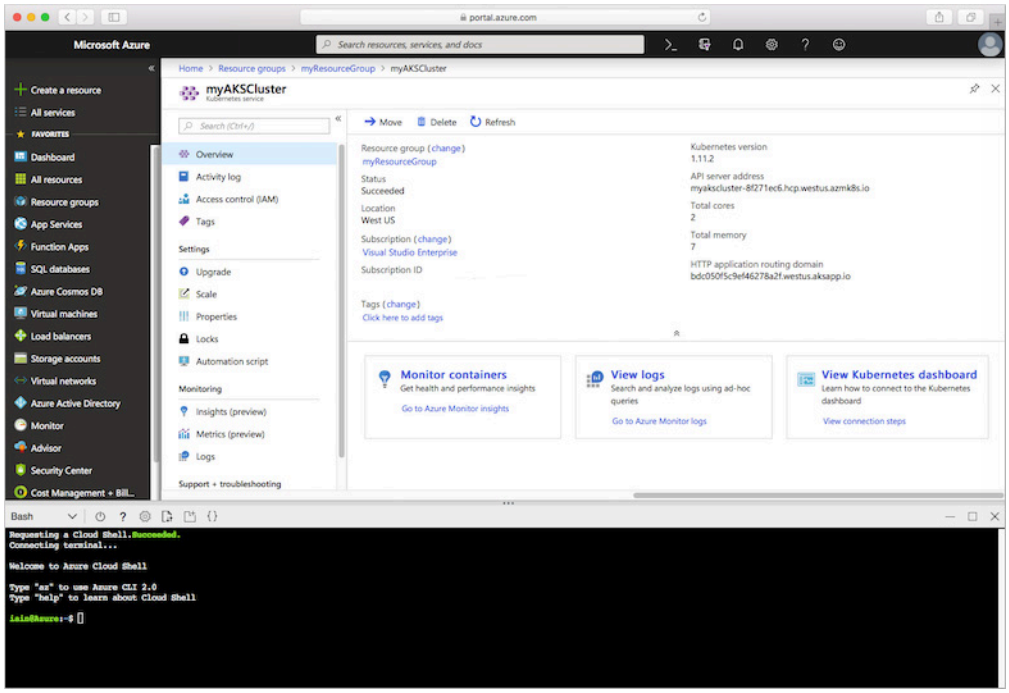
It takes a few minutes to create the AKS cluster and to be ready for use. Browse to the AKS cluster resource group, such as *myResourceGroup*, and select the AKS resource, such as *myAKSCluster*. The AKS cluster dashboard is shown, as in the following example screenshot:



Connect to the cluster

To manage a Kubernetes cluster, use `kubectl`, the Kubernetes command-line client. The `kubectl` client is pre-installed in the Azure Cloud Shell.

Open Cloud Shell using the button on the top right-hand corner of the Azure portal.



Use the az `aks get-credentials` command to configure `kubectl` to connect to your Kubernetes cluster. The following example gets credentials for the cluster name *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single node created in the previous steps.

```
NAME                       STATUS    ROLES     AGE       VERSION
aks-agentpool-14693408-0   Ready     agent     10m       v1.11.2
```

Run the application

Kubernetes manifest files define a desired state for a cluster, including what container images should be running. In this quickstart, a manifest is used to create all the objects needed to run a sample Azure Vote application. These objects include two Kubernetes deployments - one for the Azure Vote front end, and the other for a Redis instance. Also, two Kubernetes Services are created - an internal service for the Redis instance, and an external service for accessing the Azure Vote application from the internet.

Create a file named `azure-vote.yaml` and copy into it the following YAML code. If you are working in Azure Cloud Shell, create the file using `vi` or `Nano`, as if working on a virtual or physical system.

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      containers:
      - name: azure-vote-back
        image: redis
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 6379
          name: redis
--- #separator
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
  - port: 6379
  selector:
    app: azure-vote-back
--- #separator
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
      - name: azure-vote-front
```

```
        image: microsoft/azure-vote-front:v1
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 80
        env:
        - name: REDIS
          value: "azure-vote-back"
--- #separator
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Use the `kubectl apply` command to run the application.

```
kubectl apply -f azure-vote.yaml
```

The following example output shows the Kubernetes resources created on your AKS cluster:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the `kubectl get service` command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

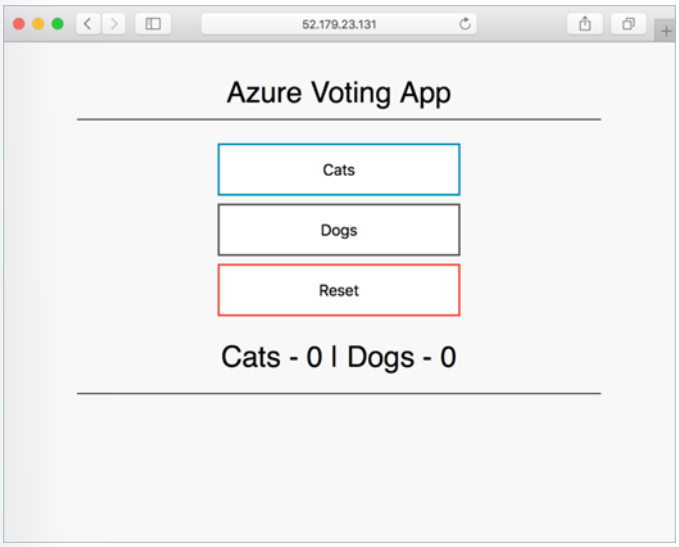
Initially the *EXTERNAL-IP* for the *azure-vote-front* service appears as pending.

```
NAME               TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
azure-vote-front   LoadBalancer   10.0.37.27   <pending>     80:30572/TCP
6s
```

Once the *EXTERNAL-IP* address has changed from pending to an IP address, use `CTRL-C` to stop the kubectl watch process.

```
azure-vote-front    LoadBalancer    10.0.37.27    52.179.23.131    80:30572/TCP
2m
```

Open a web browser to the external IP address of your service to see the Azure Vote App, as shown in the following example:



Monitor health and logs

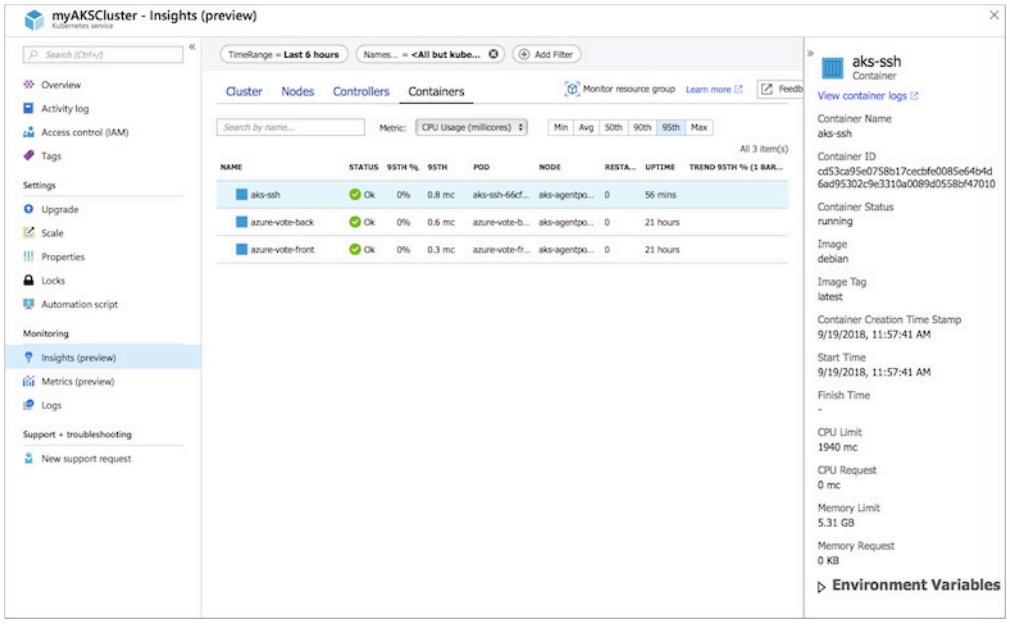
When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see, see **Monitor Azure Kubernetes Service health**[8].

It may take a few minutes for this data to populate in the Azure portal. To see current status, uptime, and resource usage for the Azure Vote pods, complete the following steps:

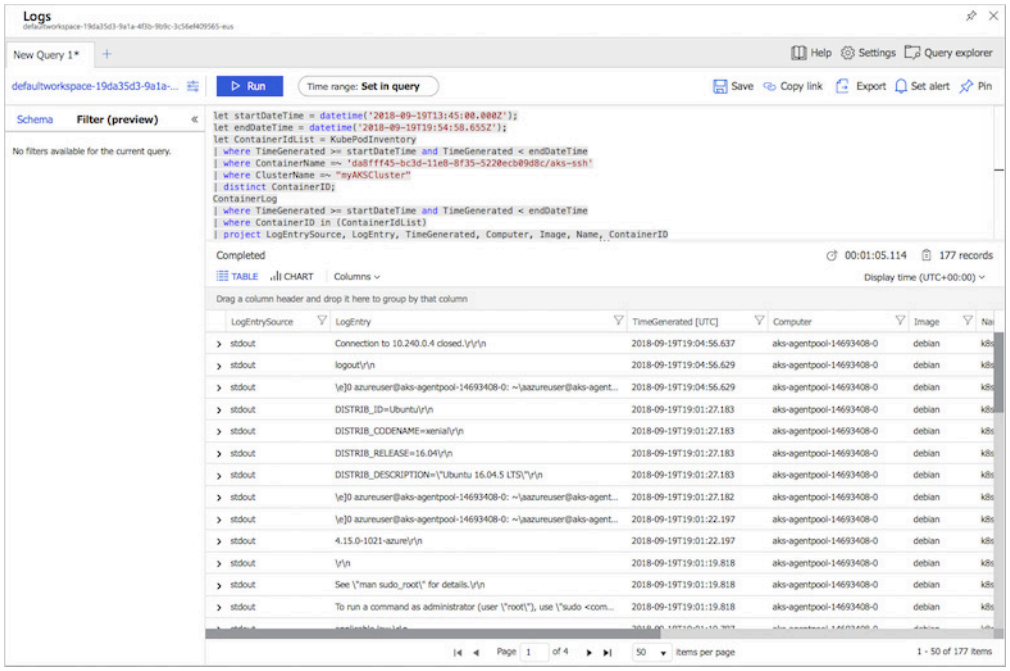
1. Open a web browser to the Azure portal https://portal.azure.com.
2. Select your resource group, such as *myResourceGroup*, then select your AKS cluster, such as *myAKS-Cluster*.
3. Under **Monitoring** on the left-hand side, choose **Insights (preview)**
4. Across the top, choose to **+ Add Filter**
5. Select **Namespace** as the property, then choose *<All but kube-system>*
6. Choose to view the **Containers**.

The *azure-vote-back* and *azure-vote-front* containers are displayed, as shown in the following example:

8 https://docs.microsoft.com/en-us/azure/monitoring/monitoring-container-health



To see logs for the `azure-vote-front` pod, select the **View container logs** link on the right-hand side of the containers list. These logs include the *stdout* and *stderr* streams from the container.



Delete cluster

When the cluster is no longer needed, delete the cluster resource, which deletes all associated resources. This operation can be completed in the Azure portal by selecting the **Delete** button on the AKS cluster dashboard. Alternatively, the az aks delete command can be used in the Cloud Shell:

```
az aks delete --resource-group myResourceGroup --name myAKSCluster --no-
wait
```

Note: When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see **AKS service principal considerations and deletion**[9].

Get the code

In this tutorial, pre-created container images have been used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

https://github.com/Azure-Samples/azure-voting-app-redis

apiVersion: v1
kind: Service
metadata:
name: azure-vote-back
spec:
ports:

- port: 6379
  selector:
  app: azure-vote-back

apiVersion: apps/v1beta1
kind: Deployment
metadata:
name: azure-vote-front
spec:
replicas: 1
template:
metadata:
labels:
app: azure-vote-front
spec:
containers:
- name: azure-vote-front
image: microsoft/azure-vote-front:v1
ports:
- containerPort: 80
env:
- name: REDIS
value: “azure-vote-back”

apiVersion: v1
kind: Service
metadata:
name: azure-vote-front
spec:

9 https://docs.microsoft.com/en-us/azure/aks/kubernetes-service-principal#additional-considerations

type: LoadBalancer
ports:

- port: 80
  selector:
  app: azure-vote-front

```
Use the `kubectl apply` command to run the application.

```azurecli
kubectl apply -f azure-vote.yaml
```

Output:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the `kubectl get service` command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

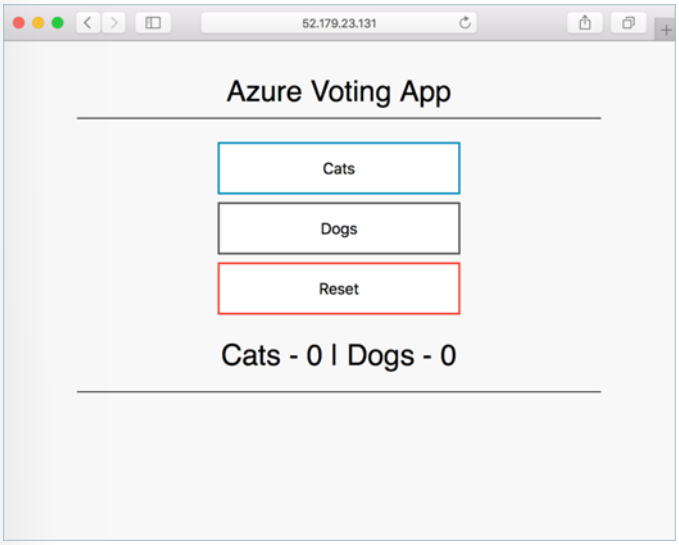
Initially the EXTERNAL-IP for the azure-vote-front service appears as pending.

```
NAME               TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
azure-vote-front   LoadBalancer   10.0.37.27   <pending>     80:30572/TCP
6s
```

Once the EXTERNAL-IP address has changed from pending to an IP address, use CTRL-C to stop the kubectl watch process.

```
azure-vote-front   LoadBalancer   10.0.37.27   52.179.23.131   80:30572/TCP
2m
```

Now browse to the external IP address to see the Azure Vote App.
```



Monitor health and logs

When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see Monitor Azure Kubernetes Service health.

To see current status, uptime, and resource usage for the Azure Vote pods, complete the following steps:

1. Open a web browser to the Azure portal.
2. Select your resource group, such as *myResourceGroup*, then select your AKS cluster, such as *myAKSCluster*.
   3.Choose **Monitor container health >** select the **default** namespace > then select **Containers**.

To see logs for the `azure-vote-front` pod, select the **View Logs** link on the right-hand side of the list of containers. These logs include the *stdout* and *stderr* streams from the container.

Developer Best Practices for Managing Resources in AKS

As you develop and run applications in Azure Kubernetes Service (AKS), there are a few key areas to consider. How you manage your application deployments can negatively impact the end-user experience of services that you provide. To help you succeed, keep in mind some best practices you can follow as you develop and run applications in AKS.

In this section, you learn:

- What are pod resource requests and limits
- Ways to develop and deploy applications with Dev Spaces and Visual Studio Code
- How to use the `kube-advisor` tool to check for issues with deployments

Define pod resource requests and limits

Best practice guidance - Set pod requests and limits on all pods in your YAML manifests. If the AKS cluster uses *resource quotas*, your deployment may be rejected if you don't define these values.

A primary way to manage the compute resources within an AKS cluster is to use pod requests and limits. These requests and limits let the Kubernetes scheduler know what compute resources a pod should be assigned.

- **Pod requests** define a set amount of CPU and memory that the pod needs. These requests should be the amount of compute resources the pod needs to provide an acceptable level of performance.
    - When the Kubernetes scheduler tries to place a pod on a node, the pod requests are used to determine which node has sufficient resources available.
    - Monitor the performance of your application to adjust these requests to make sure you don't define less resources that required to maintain an acceptable level of performance.
- **Pod limits** are the maximum amount of CPU and memory that a pod can use. These limits help prevent one or two runaway pods from taking too much CPU and memory from the node. This scenario would reduce the performance of the node and other pods that run on it.
    - Don't set a pod limit higher than your nodes can support. Each AKS node reserves a set amount of CPU and memory for the core Kubernetes components. Your application may try to consume too many resources on the node for other pods to successfully run.
    - Again, monitor the performance of your application at different times during the day or week. Determine when the peak demand is, and align the pod limits to the resources required to meet the application's needs.

In your pod specifications, it's best practice to define these requests and limits. If you don't include these values, the Kubernetes scheduler doesn't understand what resources are needed. The scheduler may schedule the pod on a node without sufficient resources to provide acceptable application performance. The cluster administrator may set *resource quotas* on a namespace that requires you to set resource requests and limits.

When you define a CPU request or limit, the value is measured in CPU units. *1.0* CPU equates to one underlying virtual CPU core on the node. The same measurement is used for GPUs. You can also define a fractional request or limit, typically in millicpu. For example, *100m* is *0.1* of an underlying virtual CPU core.

In the following basic example for a single NGINX pod, the pod requests *100m* of CPU time, and *128Mi* of memory. The resource limits for the pod are set to *250m* CPU and *256Mi* memory:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:1.15.5
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
```

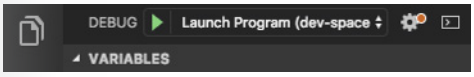
```
    limits:
      cpu: 250m
      memory: 256Mi
```

For more information about resource measurements and assignments, see **Managing compute resources for containers**[10].

Develop and debug applications against an AKS cluster

Best practice guidance - Development teams should deploy and debug against an AKS cluster using Dev Spaces. This development model makes sure that role-based access controls, network, or storage needs are implemented before the app is deployed to production.

With Azure Dev Spaces, you develop, debug, and test applications directly against an AKS cluster. Developers within a team work together to build and test throughout the application lifecycle. You can continue to use existing tools such as Visual Studio or Visual Studio Code. An extension is installed for Dev Spaces that gives an option to run and debug the application in an AKS cluster:

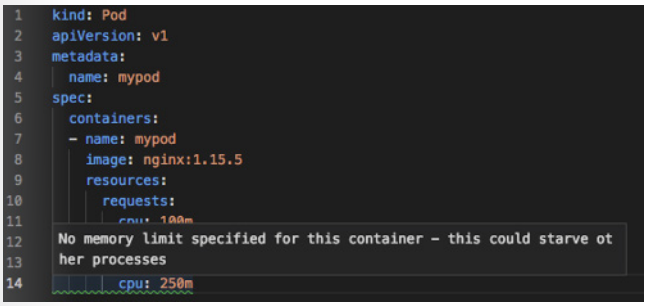


This integrated development and test process with Dev Spaces reduces the need for local test environments, such as `minikube`. Instead, you develop and test against an AKS cluster. This cluster can be secured and isolated as noted in previous section on the use of namespaces to logically isolate a cluster. When your apps are ready to deploy to production, you can confidently deploy as your development was all done against a real AKS cluster.

Use the Visual Studio Code extension for Kubernetes

Best practice guidance - Install and use the VS Code extension for Kubernetes when you write YAML manifests. You can also use the extension for integrated deployment solution, which may help application owners that infrequently interact with the AKS cluster.

The **Visual Studio Code extension for Kubernetes**[11] helps you develop and deploy applications to AKS. The extension provides intellisense for Kubernetes resources, and Helm charts and templates. You can also browse, deploy, and edit Kubernetes resources from within VS Code. The extension also provides an intellisense check for resource requests or limits being set in the pod specifications:



10 https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/
11 https://github.com/Azure/vscode-kubernetes-tools

Regularly check for application issues with kube-advisor

Best practice guidance - Regularly run the latest version of `kube-advisor` to detect issues in your cluster. If you apply resource quotas on an existing AKS cluster, run `kube-advisor` first to find pods that don't have resource requests and limits defined.

The `kube-advisor` tool scans a Kubernetes cluster and reports on issues that it finds. One useful check is to identify pods that don't have resource requests and limits in place.

In an AKS cluster that hosts many development teams and applications, it can be hard to track pods without these resource requests and limits set. As a best practice, regularly run `kube-advisor` on your AKS clusters.

Azure Container Registry

Azure Container Registry overview

Azure Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images.

Use container registries in Azure with your existing container development and deployment pipelines. Use Azure Container Registry Build (ACR Build) to build container images in Azure. Build on demand, or fully automate builds with source code commit and base image update build triggers.

For background about Docker and containers, see the **Docker overview**[12].

Use cases

Pull images from an Azure container registry to various deployment targets:

- **Scalable orchestration systems** that manage containerized applications across clusters of hosts, including Kubernetes, DC/OS, and Docker Swarm.
- **Azure services** that support building and running applications at scale, including Azure Kubernetes Service (AKS), App Service, Batch, Service Fabric, and others.

Developers can also push to a container registry as part of a container development workflow. For example, target a container registry from a continuous integration and deployment tool such as Azure DevOps Services or Jenkins.

Configure ACR Tasks to automatically rebuild application images when their base images are updated. Use ACR Tasks to automate image builds when your team commits code to a Git repository.

Key concepts

- **Registry** - Create one or more container registries in your Azure subscription. Registries are available in three SKUs: Basic, Standard, and Premium, each of which support webhook integration, registry authentication with Azure Active Directory, and delete functionality. Take advantage of local, network-close storage of your container images by creating a registry in the same Azure location as your deployments. Use the geo-replication feature of Premium registries for advanced replication and container image distribution scenarios. A fully qualified registry name has the form `myregistry.azurecr.io`.
- You control access to a container registry using an Azure Active Directory-backed service principal or a provided admin account. Run the standard docker login command to authenticate with a registry.
- **Repository** - A registry contains one or more repositories, which store groups of container images. Azure Container Registry supports multilevel repository namespaces. With multilevel namespaces, you can group collections of images related to a specific app, or a collection of apps to specific development or operational teams. For example:
  - `myregistry.azurecr.io/aspnetcore:1.0.1` represents a corporate-wide image
  - `myregistry.azurecr.io/warrantydept/dotnet-build` represents an image used to build .NET apps, shared across the warranty department

12 https://docs.docker.com/engine/docker-overview/

- `myregistry.azurecr.io/warrantydept/customersubmissions/web` represents a web image, grouped in the customer submissions app, owned by the warranty department

- **Image** - Stored in a repository, each image is a read-only snapshot of a Docker-compatible container. Azure container registries can include both Windows and Linux images. You control image names for all your container deployments. Use standard Docker commands to push images into a repository, or pull an image from a repository. In addition to container images, Azure Container Registry stores related content formats such as Helm charts, used to deploy applications to Kubernetes.
- **Container** - A container defines a software application and its dependencies wrapped in a complete filesystem including code, runtime, system tools, and libraries. Run Docker containers based on Windows or Linux images that you pull from a container registry. Containers running on a single machine share the operating system kernel. Docker containers are fully portable to all major Linux distros, macOS, and Windows.

Azure Container Registry Tasks

Azure Container Registry Tasks (ACR Tasks) is a suite of features within Azure Container Registry that provides streamlined and efficient Docker container image builds in Azure. Use ACR Tasks to extend your development inner-loop to the cloud by offloading `docker build` operations to Azure. Configure build tasks to automate your container OS and framework patching pipeline, and build images automatically when your team commits code to source control.

Multi-step tasks, a preview feature of ACR Tasks, provides step-based task definition and execution for building, testing, and patching container images in the cloud. Task steps define individual container image build and push operations. They can also define the execution of one or more containers, with each step using the container as its execution environment.

Deploy an image to ACR using Azure CLI

Azure Container Registry is a managed Docker container registry service used for storing private Docker container images. This guide details creating an Azure Container Registry instance using the Azure CLI, pushing a container image into the registry and finally deploying the container from your registry into Azure Container Instances (ACI).

Create a resource group

Create a resource group with the `az group create` command. An Azure resource group is a logical container into which Azure resources are deployed and managed.

The following example creates a resource group named *myResourceGroup* in the eastus location.
Azure CLI

```
az group create --name myResourceGroup --location eastus
```

Create a container registry

In this guide you create a *Basic* registry. Azure Container Registry is available in several different SKUs, described briefly in the following table. For extended details on each, see **Container registry SKUs**[13].

SKU	Description

13 https://docs.microsoft.com/en-us/azure/container-registry/container-registry-skus

Basic	A cost-optimized entry point for developers learning about Azure Container Registry. Basic registries have the same programmatic capabilities as Standard and Premium (Azure Active Directory authentication integration, image deletion, and web hooks), however, there are size and usage constraints.
Standard	The Standard registry offers the same capabilities as Basic, but with increased storage limits and image throughput. Standard registries should satisfy the needs of most production scenarios.
Premium	Premium registries have higher limits on constraints, such as storage and concurrent operations, including enhanced storage capabilities to support high-volume scenarios. In addition to higher image throughput capacity, Premium adds features like geo-replication for managing a single registry across multiple regions, maintaining a network-close registry to each deployment.

Create an ACR instance using the `az acr create` command. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the following example, *myContainerRegistry007* is used. Update this to a unique value.

```
az acr create --resource-group myResourceGroup --name myContainerRegis-
try007 --sku Basic
```

When the registry is created, the output is similar to the following:

```
{
  "adminUserEnabled": false,
  "creationDate": "2017-09-08T22:32:13.175925+00:00",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resource-
Groups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/
myContainerRegistry007",
  "location": "eastus",
  "loginServer": "myContainerRegistry007.azurecr.io",
  "name": "myContainerRegistry007",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "status": null,
  "storageAccount": null,
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

Throughout the rest of this guide `<acrName>` is a placeholder for the container registry name.

Log in to ACR

Before pushing and pulling container images, you must log in to the ACR instance. To do so, use the `az acr login` command.

```
az acr login --name <acrName>
```

The command returns a Login Succeeded message once completed.

Push image to ACR

To push an image to an Azure Container registry, you must first have an image. If you don't yet have any local container images, run the following command to pull an existing image from Docker Hub.

```
docker pull microsoft/aci-helloworld
```

Before you can push an image to your registry, you must tag it with the fully qualified name of your ACR login server. Run the following command to obtain the full login server name of the ACR instance.

```
az acr list --resource-group myResourceGroup --query "[].{acrLogin-
Server:loginServer}" --output table
```

Tag the image using the docker tag command. Replace `<acrLoginServer>` with the login server name of your ACR instance.

```
docker tag microsoft/aci-helloworld <acrLoginServer>/aci-helloworld:v1
```

Finally, use docker push to push the image to the ACR instance. Replace `<acrLoginServer>` with the login server name of your ACR instance.

```
docker push <acrLoginServer>/aci-helloworld:v1
```

List container images

The following example lists the repositories in a registry:

```
az acr repository list --name <acrName> --output table
```

Output:

```
Result
----------------
aci-helloworld
```

The following example lists the tags on the **aci-helloworld** repository.

```
az acr repository show-tags --name <acrName> --repository aci-helloworld
--output table
```

Output:

```
Result
--------
v1
```

Deploy image to ACI

In order to deploy a container instance from the registry you created, you must provide the registry credentials when you deploy it. In production scenarios you should use a service principal for container registry access, but to keep this quickstart brief, enable the admin user on your registry with the following command:

```
az acr update --name <acrName> --admin-enabled true
```

Once admin is enabled the username is the same as your registry name and you can retrieve the password with this command:

```
az acr credential show --name <acrName> --query "passwords[0].value"
```

To deploy your container image with 1 CPU core and 1 GB of memory, run the following command. Replace `<acrName>`, `<acrLoginServer>`, and `<acrPassword>` with the values you obtained from previous commands.

```
az container create --resource-group myResourceGroup --name acr-quickstart
--image <acrLoginServer>/aci-helloworld:v1 --cpu 1 --memory 1 --regis-
try-username <acrName> --registry-password <acrPassword> --dns-name-label
aci-demo --ports 80
```

You should get an initial response back from Azure Resource Manager with details on your container. To monitor the status of your container and check and see when it is running, repeat the `az container show`. It should take less than a minute.

```
az container show --resource-group myResourceGroup --name acr-quickstart
--query instanceView.state
```

View the application

Once the deployment to ACI is successful, retrieve the container's FQDN with the `az container show` command:

```
az container show --resource-group myResourceGroup --name acr-quickstart
--query ipAddress.fqdn
```

Example output: `"aci-demo.eastus.azurecontainer.io"`

To see the running application, navigate to the public IP address in your favorite browser.

Azure Container Instances

Azure Container Intances Overview

Containers are becoming the preferred way to package, deploy, and manage cloud applications. Azure Container Instances offers the fastest and simplest way to run a container in Azure, without having to manage any virtual machines and without having to adopt a higher-level service.

Azure Container Instances is a great solution for any scenario that can operate in isolated containers, including simple applications, task automation, and build jobs. For scenarios where you need full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades, we recommend Azure Kubernetes Service (AKS).

Below is an overview of the features of Azure Container Instances.

Feature	Description
Fast startup times	Containers offer significant startup benefits over virtual machines. Azure Container Instances can start containers in Azure in seconds, without the need to provision and manage VMs.
Public IP connectivity and DNS name	Azure Container Instances enables exposing your containers directly to the internet with an IP address and a fully qualified domain name (FQDN). When you create a container instance, you can specify a custom DNS name label so your application is reachable at *customlabel.azureregion.azurecontainer.io*.
Hypervisor-level security	Historically, containers have offered application dependency isolation and resource governance but have not been considered sufficiently hardened for hostile multi-tenant usage. Azure Container Instances guarantees your application is as isolated in a container as it would be in a VM.
Custom sizes	As demand for resources increases, the nodes of an AKS cluster can be scaled out to match. If resource demand drops, nodes can be removed by scaling in the cluster. AKS scale operations can be completed using the Azure portal or the Azure CLI. For compute-intensive jobs such as machine learning, Azure Container Instances can schedule Linux containers to use NVIDIA Tesla GPU resources (preview).
Persistent storage	To retrieve and persist state with Azure Container Instances, we offer direct mounting of Azure Files shares.

Linux and Windows containers	Azure Container Instances can schedule both Windows and Linux containers with the same API. Simply specify the OS type when you create your container groups. Some features are currently restricted to Linux containers. While we work to bring feature parity to Windows containers, you can find current platform differences in **Quotas and region availability for Azure Container Instances** *(https://docs.microsoft.com/en-us/azure/container-instances/container-instances-quotas)*. Azure Container Instances supports Windows images based on Long-Term Servicing Channel (LTSC) versions. Windows Semi-Annual Channel (SAC) releases like 1709 and 1803 are unsupported.
Co-scheduled groups	Azure Container Instances supports scheduling of multi-container groups that share a host machine, local network, storage, and lifecycle. This enables you to combine your main application container with other supporting role containers, such as logging sidecars.
Virtual network deployment (preview)	Currently in preview, this feature of Azure Container Instances enables deployment of container instances into an Azure virtual network. By deploying container instances into a subnet within your virtual network, they can communicate securely with other resources in the virtual network, including those that are on premises (through VPN gateway or ExpressRoute).

Important: Certain features of Azure Container Instances are in preview, and some **limitations apply**[14]. Previews are made available to you on the condition that you agree to the **supplemental terms of use**[15]. Some aspects of these features may change prior to general availability (GA).

Create Container for Deployment to ACI

When using the Virtual Kubelet provider for Azure Container Instances, both Linux and Windows containers can be scheduled on a container instance as if it is a standard Kubernetes node. This configuration allows you to take advantage of both the capabilities of Kubernetes and the management value and cost benefit of container instances.

Note: Virtual Kubelet is an experimental open source project and should be used as such. To contribute, file issues, and read more about virtual kubelet, see the **Virtual Kubelet GitHub project**[16].

14 https://docs.microsoft.com/en-us/azure/container-instances/container-instances-vnet#preview-limitations
15 https://azure.microsoft.com/support/legal/preview-supplemental-terms/
16 https://github.com/virtual-kubelet/virtual-kubelet

Prerequisites

This guide assumes that you have an AKS cluster. You also need the Azure CLI version 2.0.33 or later. Run `az --version` to find the version.

To install the Virtual Kubelet, **Helm**[17] is also required.

For RBAC-enabled clusters

If your AKS cluster is RBAC-enabled, you must create a service account and role binding for use with Tiller.

A *ClusterRoleBinding* must also be created for the Virtual Kubelet. To create a binding, create a file named *rbac-virtualkubelet.yaml* and paste the following definition:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: virtual-kubelet
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
```

Apply the binding with `kubectl apply` and specify your *rbac-virtualkubelet.yaml* file, as shown in the following example:

```
$ kubectl apply -f rbac-virtual-kubelet.yaml

clusterrolebinding.rbac.authorization.k8s.io/virtual-kubelet created
```

You can now continue to installing the Virtual Kubelet into your AKS cluster.

Installation

Use the `az aks install-connector` command to install Virtual Kubelet. The following example deploys both the Linux and Windows connector.

```
az aks install-connector --resource-group myAKSCluster --name myAKSCluster
--connector-name virtual-kubelet --os-type Both
```

These arguments are available for the `aks install-connector` command.

Argument	Description	Required
`--connector-name`	Name of the ACI Connector.	Yes
`--name`	Name of the managed cluster.	Yes

17 https://docs.helm.sh/using_helm/#installing-helm

`--resource-group`	Name of resource group.	Yes
`--os-type`	Container instances operating system type. Allowed values: Both, Linux, Windows. Default: Linux.	No
`--aci-resource-group`	The resource group in which to create the ACI container groups.	No
`--location`	The location to create the ACI container groups.	No
`--service-principal`	Service principal used for authentication to Azure APIs.	No
`--client-secret`	Secret associated with the service principal.	No
`--chart-url`	URL of a Helm chart that installs ACI Connector.	No
`--image-tag`	The image tag of the virtual kubelet container image.	No

Validate Virtual Kubelet

To validate that Virtual Kubelet has been installed, return a list of Kubernetes nodes using the `kubectl get nodes` command.

```
$ kubectl get nodes

NAME                                STATUS    ROLES     AGE       VER-
SION
aks-nodepool1-23443254-0            Ready     agent     16d
v1.9.6
aks-nodepool1-23443254-1            Ready     agent     16d
v1.9.6
aks-nodepool1-23443254-2            Ready     agent     16d
v1.9.6
virtual-kubelet-virtual-kubelet-linux   Ready     agent     4m
v1.8.3
virtual-kubelet-virtual-kubelet-win     Ready     agent     4m
v1.8.3
```

Run Linux container

Create a file named *virtual-kubelet-linux.yaml* and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Linux Virtual Kubelet node. Take note that a `nodeSelector` and `toleration` are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
```

```
    template:
      metadata:
        labels:
          app: aci-helloworld
      spec:
        containers:
        - name: aci-helloworld
          image: microsoft/aci-helloworld
          ports:
          - containerPort: 80
        nodeSelector:
          kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-linux
        tolerations:
        - key: azure.com/aci
          effect: NoSchedule
```

Run the application with the `kubectl create` command.

```
kubectl create -f virtual-kubelet-linux.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the `aci-helloworld` pod has been scheduled on the `virtual-kubelet-virtual-kubelet-linux` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE
IP             NODE
aci-helloworld-2559879000-8vmjw     1/1       Running   0          39s
52.179.3.180   virtual-kubelet-virtual-kubelet-linux
```

Run Windows container

Create a file named *virtual-kubelet-windows.yaml* and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Windows Virtual Kubelet node. Take note that a `nodeSelector` and `toleration` are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nanoserver-iis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nanoserver-iis
    spec:
      containers:
      - name: nanoserver-iis
        image: nanoserver/iis
```

```
    ports:
    - containerPort: 80
  nodeSelector:
    kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-win
  tolerations:
  - key: azure.com/aci
    effect: NoSchedule
```

Run the application with the `kubectl create` command.

```
kubectl create -f virtual-kubelet-windows.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the nanoserver-iis pod has been scheduled on the `virtual-kubelet-virtual-kubelet-win` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE
IP             NODE
nanoserver-iis-868bc8d489-tq4st     1/1       Running   8          21m
138.91.121.91   virtual-kubelet-virtual-kubelet-win
```

Deploy a Container to ACI

In the last tutorial a container image was create and pushed to the Azure Container Registry. Now, you will deploy the container to Azure Container Instances.

In this tutorial, you:

- Deploy the container from Azure Container Registry to Azure Container Instances
- View the running application in the browser
- Display the container's logs

Before you begin

You must satisfy the following requirements to complete this tutorial:

Azure CLI: You must have Azure CLI version 2.0.29 or later installed on your local computer. Run `az --version` to find the version. If you need to install or upgrade, see **Install the Azure CLI**[18].

Docker: This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic docker commands. For a primer on Docker and container basics, see the [Docker overview]https://docs.docker.com/engine/docker-overview/.

Docker Engine: To complete this tutorial, you need Docker Engine installed locally. Docker provides packages that configure the Docker environment on **macOS**[19], **Windows**[20], and **Linux**[21].

18 https://docs.microsoft.com/cli/azure/install-azure-cli
19 https://docs.docker.com/docker-for-mac/
20 https://docs.docker.com/docker-for-windows/
21 https://docs.docker.com/engine/installation/#supported-platforms

Important: Because the Azure Cloud shell does not include the Docker daemon, you must install both the Azure CLI and Docker Engine on your local computer to complete this tutorial. You cannot use the Azure Cloud Shell for this tutorial.

Deploy the container using the Azure CLI

In this section, you use the Azure CLI to deploy the image you have already built and pushed to Azure Container Registry. Be sure you've completed those steps before proceeding.

Get registry credentials

When you deploy an image that's hosted in a private container registry you must supply the registry's credentials.

First, get the full name of the container registry login server (replace `<acrName>` with the name of your registry):

```
az acr show --name <acrName> --query loginServer
```

Next, get the container registry password:

```
az acr credential show --name <acrName> --query "passwords[0].value"
```

Deploy container

Now, use the az container create command to deploy the container. Replace `<acrLoginServer>` and `<acrPassword>` with the values you obtained from the previous two commands. Replace `<acrName>` with the name of your container registry and `<aciDnsLabel>` with desired DNS name.

```
az container create --resource-group myResourceGroup --name aci-tutori-
al-app --image <acrLoginServer>/aci-tutorial-app:v1 --cpu 1 --memory 1
--registry-login-server <acrLoginServer> --registry-username <acrName>
--registry-password <acrPassword> --dns-name-label <aciDnsLabel> --ports 80
```

Within a few seconds, you should receive an initial response from Azure. The `--dns-name-label` value must be unique within the Azure region you create the container instance. Modify the value in the preceding command if you receive a **DNS name label** error message when you execute the command.

Verify deployment progress

To view the state of the deployment, use `az container show`:

```
az container show --resource-group myResourceGroup --name aci-tutorial-app
--query instanceView.state
```

Repeat the `az container show` command until the state changes from *Pending* to *Running*, which should take under a minute. When the container is *Running*, proceed to the next step.

View the application and container logs

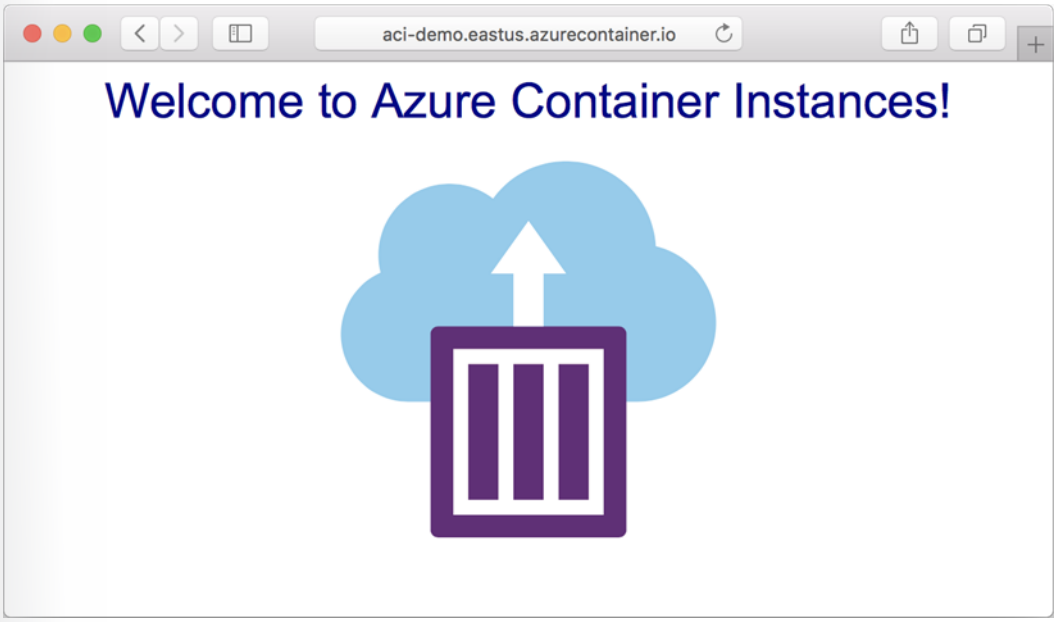
Once the deployment succeeds, display the container's fully qualified domain name (FQDN) with the `az container show` command:

```
az container show --resource-group myResourceGroup --name aci-tutorial-app
--query ipAddress.fqdn
```

For example:

```
$ az container show --resource-group myResourceGroup --name aci-tutori-
al-app --query ipAddress.fqdn
"aci-demo.eastus.azurecontainer.io"
```

To see the running application, navigate to the displayed DNS name in your favorite browser:



You can also view the log output of the container:

```
az container logs --resource-group myResourceGroup --name aci-tutorial-app
```

Example output:

```
$ az container logs --resource-group myResourceGroup --name aci-tutori-
al-app
listening on port 80
::ffff:10.240.0.4 - - [21/Jul/2017:06:00:02 +0000] "GET / HTTP/1.1" 200 1663
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36"
::ffff:10.240.0.4 - - [21/Jul/2017:06:00:02 +0000] "GET /favicon.ico
HTTP/1.1" 404 150 "http://aci-demo.eastus.azurecontainer.io/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/59.0.3071.115 Safari/537.36"
```

Clean up resources

If you no longer need any of the resources you created in this tutorial series, you can execute the az group delete command to remove the resource group and all resources it contains. This command deletes the container registry you created, as well as the running container, and all related resources.

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

Implement an application using Virtual Kublet

Azure Container Instances (ACI) provide a hosted environment for running containers in Azure. When using ACI, there is no need to manage the underlying compute infrastructure, Azure handles this management for you. When running containers in ACI, you are charged by the second for each running container.

When using the Virtual Kubelet provider for Azure Container Instances, both Linux and Windows containers can be scheduled on a container instance as if it is a standard Kubernetes node. This configuration allows you to take advantage of both the capabilities of Kubernetes and the management value and cost benefit of container instances.

Note: Virtual Kubelet is an experimental open source project and should be used as such. To contribute, file issues, and read more about virtual kubelet, see the **Virtual Kubelet GitHub project**[22].

Prerequisites

This guide assumes that you have an AKS cluster. You also need the Azure CLI version 2.0.33 or later. Run `az --version` to find the version.

To install the Virtual Kubelet, **Helm**[23] is also required.

For RBAC-enabled clusters

If your AKS cluster is RBAC-enabled, you must create a service account and role binding for use with Tiller.

A *ClusterRoleBinding* must also be created for the Virtual Kubelet. To create a binding, create a file named *rbac-virtualkubelet.yaml* and paste the following definition:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: virtual-kubelet
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
```

22 https://github.com/virtual-kubelet/virtual-kubelet
23 https://docs.helm.sh/using_helm/#installing-helm

Apply the binding with `kubectl apply` and specify your *rbac-virtualkubelet.yaml* file, as shown in the following example:

```
$ kubectl apply -f rbac-virtual-kubelet.yaml

clusterrolebinding.rbac.authorization.k8s.io/virtual-kubelet created
```

You can now continue to installing the Virtual Kubelet into your AKS cluster.

Installation

Use the `az aks install-connector` command to install Virtual Kubelet. The following example deploys both the Linux and Windows connector.

```
az aks install-connector --resource-group myAKSCluster --name myAKSCluster
--connector-name virtual-kubelet --os-type Both
```

These arguments are available for the `aks install-connector` command.

Argument	Description	Required
`--connector-name`	Name of the ACI Connector.	Yes
`--name`	Name of the managed cluster.	Yes
`--resource-group`	Name of resource group.	Yes
`--os-type`	Container instances operating system type. Allowed values: Both, Linux, Windows. Default: Linux.	No
`--aci-resource-group`	The resource group in which to create the ACI container groups.	No
`--location`	The location to create the ACI container groups.	No
`--service-principal`	Service principal used for authentication to Azure APIs.	No
`--client-secret`	Secret associated with the service principal.	No
`--chart-url`	URL of a Helm chart that installs ACI Connector.	No
`--image-tag`	The image tag of the virtual kubelet container image.	No

Validate Virtual Kubelet

To validate that Virtual Kubelet has been installed, return a list of Kubernetes nodes using the `kubectl get nodes` command.

```
$ kubectl get nodes

NAME                                STATUS    ROLES     AGE       VER-
SION
aks-nodepool1-23443254-0            Ready     agent     16d
```

```
v1.9.6
aks-nodepool1-23443254-1                Ready     agent     16d
v1.9.6
aks-nodepool1-23443254-2                Ready     agent     16d
v1.9.6
virtual-kubelet-virtual-kubelet-linux   Ready     agent     4m
v1.8.3
virtual-kubelet-virtual-kubelet-win     Ready     agent     4m
v1.8.3
```

Run Linux container

Create a file named *virtual-kubelet-linux.yaml* and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Linux Virtual Kubelet node. Take note that a `nodeSelector` and `toleration` are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: aci-helloworld
    spec:
      containers:
      - name: aci-helloworld
        image: microsoft/aci-helloworld
        ports:
        - containerPort: 80
      nodeSelector:
        kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-linux
      tolerations:
      - key: azure.com/aci
        effect: NoSchedule
```

Run the application with the `kubectl create` command.

```
kubectl create -f virtual-kubelet-linux.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the `aci-helloworld` pod has been scheduled on the `virtual-kubelet-virtual-kubelet-linux` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE
IP             NODE
aci-helloworld-2559879000-8vmjw     1/1       Running   0          39s
```

```
52.179.3.180   virtual-kubelet-virtual-kubelet-linux
```

Run Windows container

Create a file named* virtual-kubelet-windows.yaml* and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Windows Virtual Kubelet node. Take note that a `nodeSelector` and `toleration` are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nanoserver-iis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nanoserver-iis
    spec:
      containers:
      - name: nanoserver-iis
        image: nanoserver/iis
        ports:
        - containerPort: 80
      nodeSelector:
        kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-win
      tolerations:
      - key: azure.com/aci
        effect: NoSchedule
```

Run the application with the `kubectl create` command.

```
kubectl create -f virtual-kubelet-windows.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the nanoserver-iis pod has been scheduled on the `virtual-kubelet-virtual-kubelet-win` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE
IP             NODE
nanoserver-iis-868bc8d489-tq4st     1/1       Running   8          21m
138.91.121.91   virtual-kubelet-virtual-kubelet-win
```

Review Questions

Module 3 Review Questions

Managing Kubernetes Service Clusters

You manage several Azure subscriptions for an organization.

You have a multi-container application that consists of a web front-end and a Redis instance that runs on an Azure Kubernetes Service (AKS) cluster.

You need to manage the AKS cluster by using the command line.

What tool should you use? How should you deploy the tool?

Suggested Answer ↓

To manage a Kubernetes cluster, use kubectl, the Kubernetes command-line client.

If you are using Azure Cloud Shell, kubectl is already installed. If you want to install it locally, use the **az aks install-cli** command.

To configure kubectl to connect to your Kubernetes cluster, use the **az aks get-credentials** command. This step downloads credentials and configures the Kubernetes CLI to use them

Azure Container Registry

You manage a multi-container application based on Docker Registry 2.0.

You deploy Azure Container Registry (ACR) and create Azure container registries to store your private Docker container images.

What are the key concepts of ACR you must consider?

Suggested Answer ↓

Registry - Create one or more container registries in your Azure subscription. Registries are available in three SKUs: Basic, Standard, and Premium, each of which support webhook integration, registry authentication with Azure Active Directory, and delete functionality.

Repository - A registry contains one or more repositories, which are groups of container images.

Image - Stored in a repository, each image is a read-only snapshot of a Docker container.

Container - A container defines a software application and its dependencies wrapped in a complete filesystem including code, runtime, system tools, and libraries. Run Docker containers based on Windows or Linux images that you pull from a container registry.

Azure Container Registry SKUs

You manage a multi-container application based on Docker Registry 2.0.

You need to deploy Azure Container Registry (ACR) and create Azure container registries to store your private Docker container images.

Your application requires geo-replication of content.

What version of ACR should you use and why?

Suggested Answer ↓

Azure Container Registry is available in the following SKUS: Basic, Standard, and Premium. Standard offers higher storage limits and throughput than basic. Premium accounts add geo-replication and other features to help you manage high-volume scenarios.

Module 4 Module Understanding 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.

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

- The resources that are available to each host.

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.

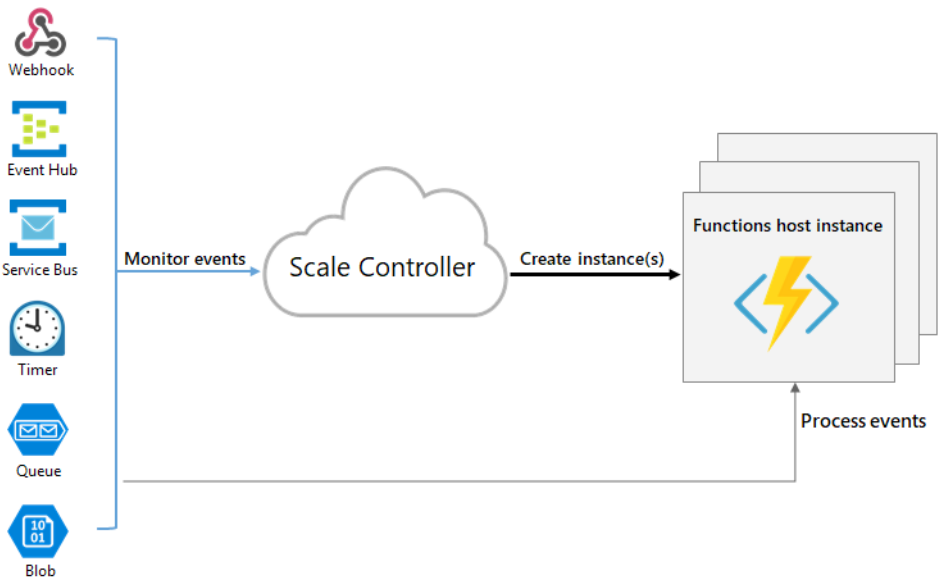
When you use the Consumption hosting plan, function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and cannot be recovered.

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, ILog-
ger 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 POCO 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 easlier 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-complied 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:

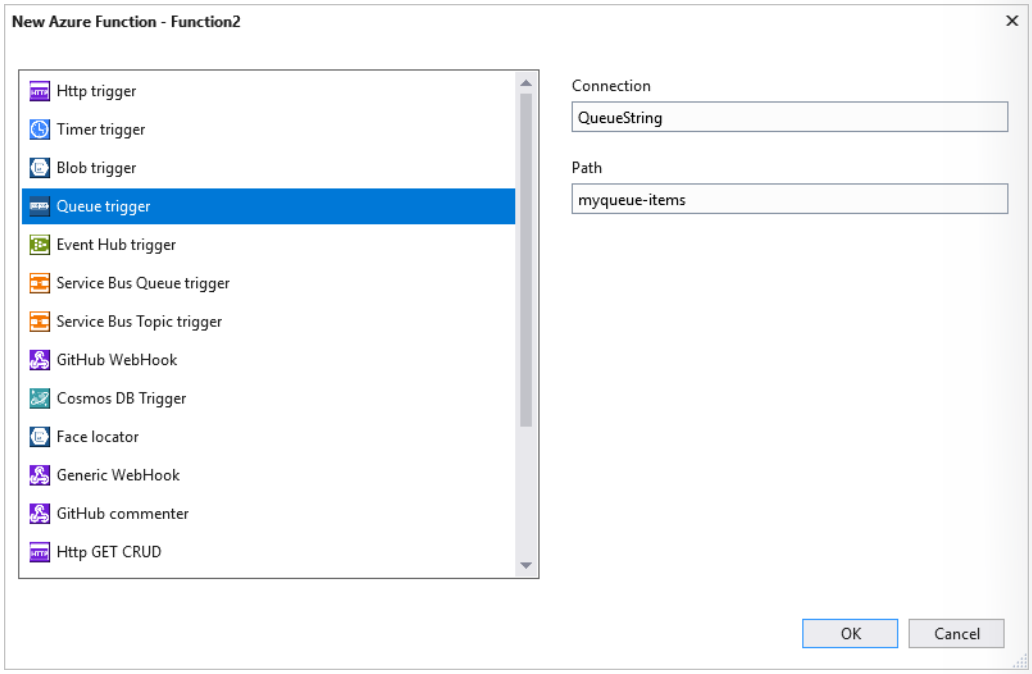
- An active Azure subscription
- An Azure Storage account.

Creating an Azure Functions project

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 **FunctionName** 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 = "AzureWebJobsS-
torage")] string myQueueItem,
        [Queue("myqueue-items-destination", Connection = "AzureWebJobsStor-
age")] 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.

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.

3 https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs
4 https://docs.microsoft.com/en-us/azure/azure-functions/functions-test-a-function

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:

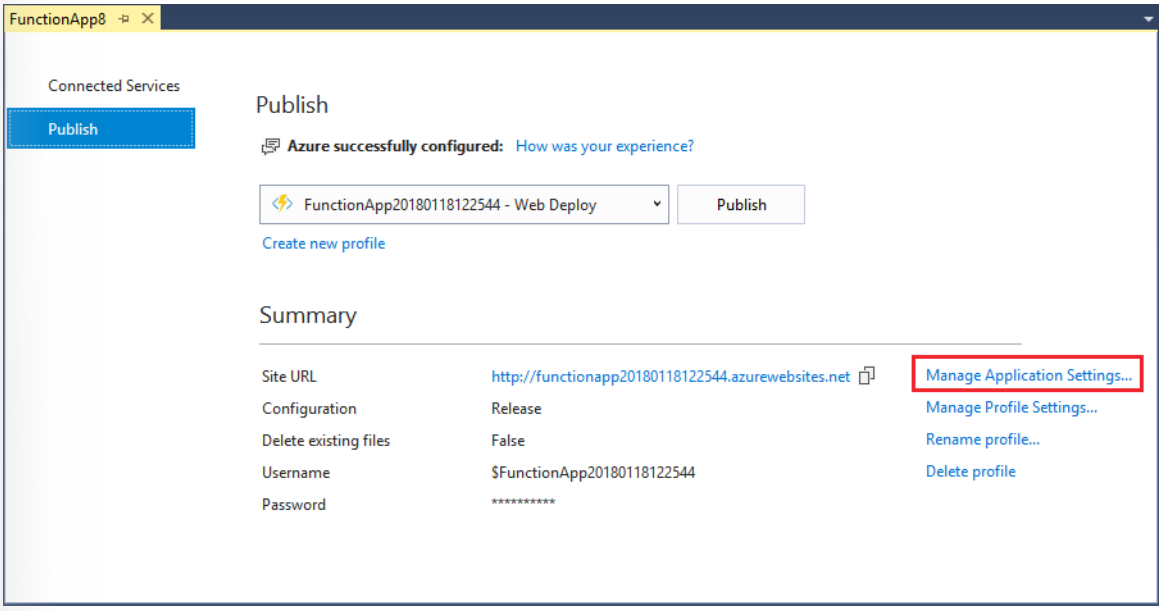
Setting	Suggested Value	Description
App Name	Globally unique name	Name that uniquely identifies your new function app.
Subscription	Choose your subscription	The Azure subscription to use.
Resource Group	myResourceGroup	Name of the resource group in which to create your function app. Choose **New** to 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.

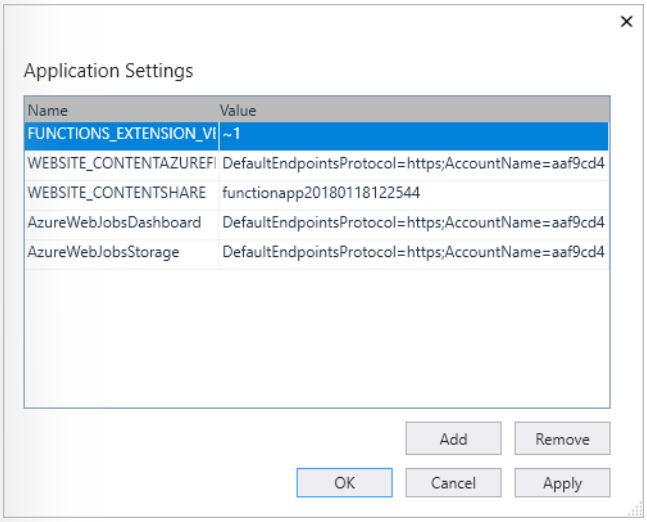
Function app settings

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:

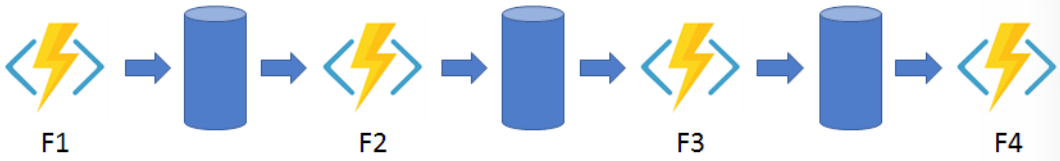
- They define workflows in code. No JSON schemas or designers are needed.
- They can call other functions synchronously and asynchronously. Output from called functions can be saved to local variables.
- They automatically checkpoint their progress whenever the function awaits. Local state is never lost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

```
    }
}
```

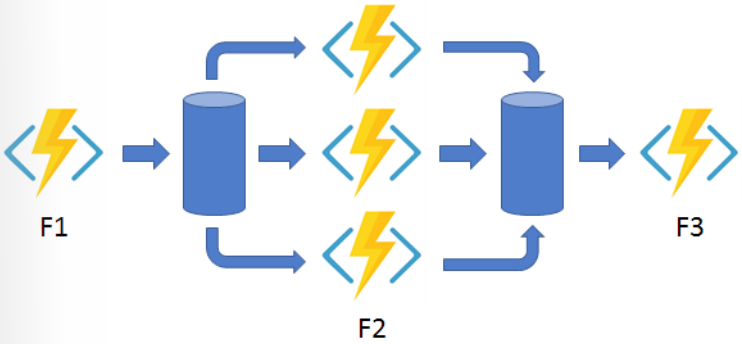
Note: There are subtle differences while writing a precompiled durable function in C# vs the C# script sample shown before. A C# precompiled function would require durable parameters to be decorated with respective attributes. An example is `[OrchestrationTrigger]` attribute for `DurableOrchestrationContext` 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);

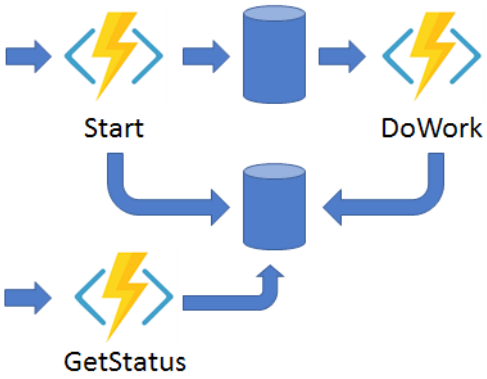
        // aggregate all N outputs and send result to F3
        int sum = parallelTasks.Sum(t => t.Result);
        await ctx.CallActivityAsync("F3", sum);
    }
```

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/DurableTaskExten-
sion/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location: https://myfunc.azurewebsites.net/admin/extensions/DurableTaskEx-
tension/b79baf67f717453ca9e86c5da21e03ec

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

> curl https://myfunc.azurewebsites.net/admin/extensions/DurableTaskExten-
sion/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, eventDa-
ta);

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

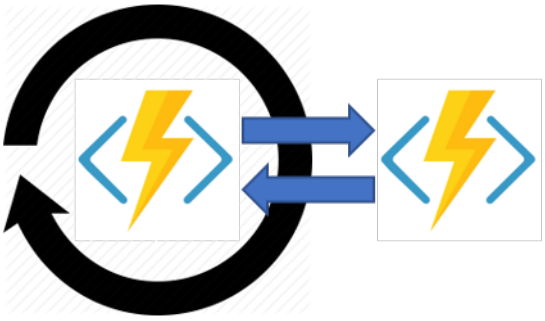
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

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>("Approva-
lEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTime-
out))
        {
            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:

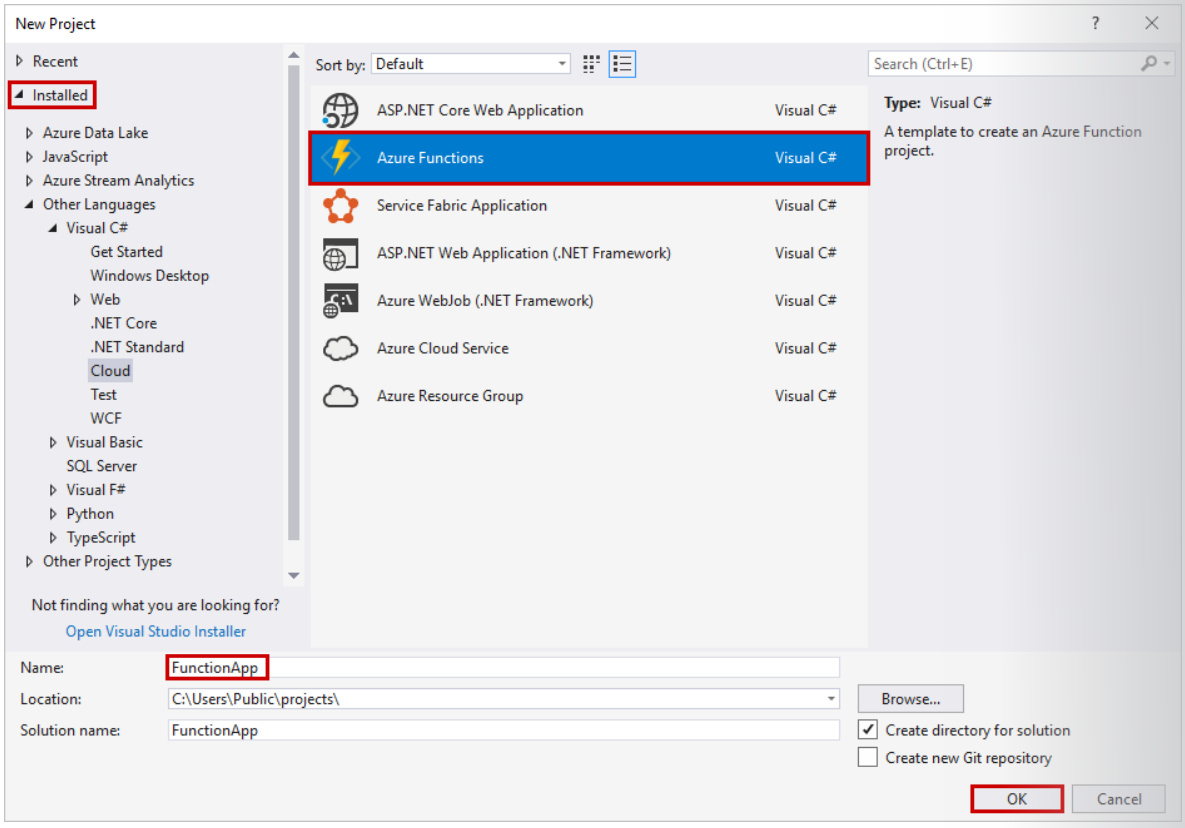
- Install **Visual Studio 2017**[5] and ensure that the **Azure development** workload is also installed.
- Make sure you have the latest **Azure Functions tools**[6].
- Verify you have the **Azure Storage Emulator**[7] installed and running.

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

Create a function app project

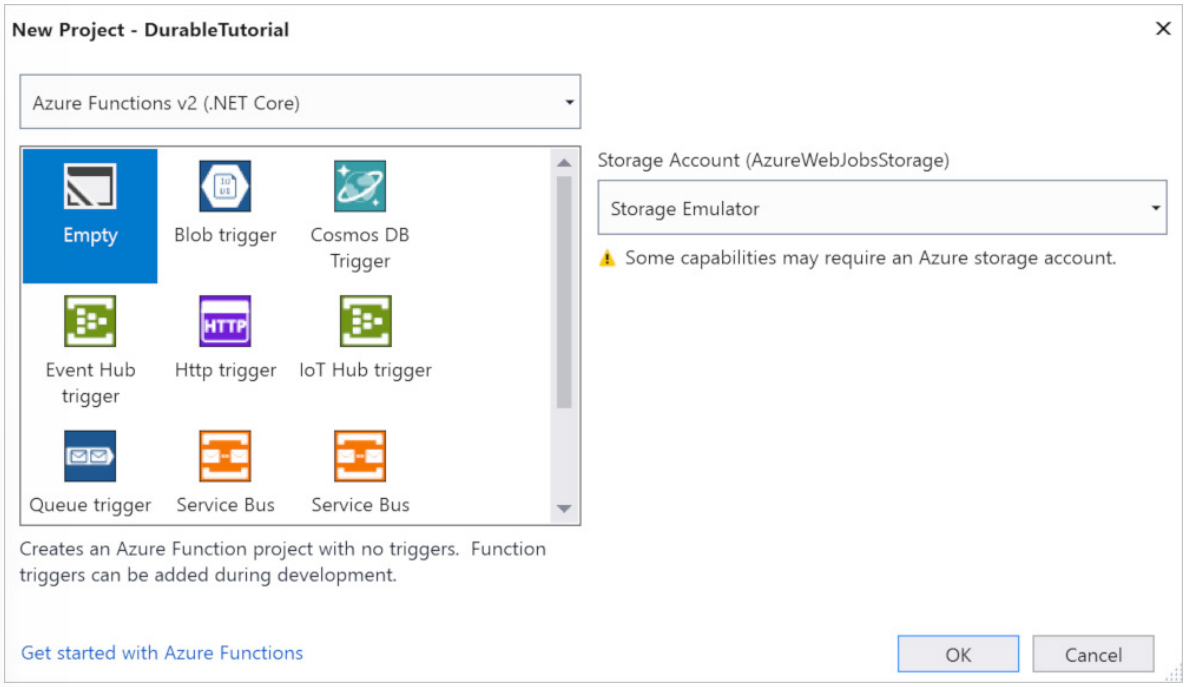
The Azure Functions project template in Visual Studio creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

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.

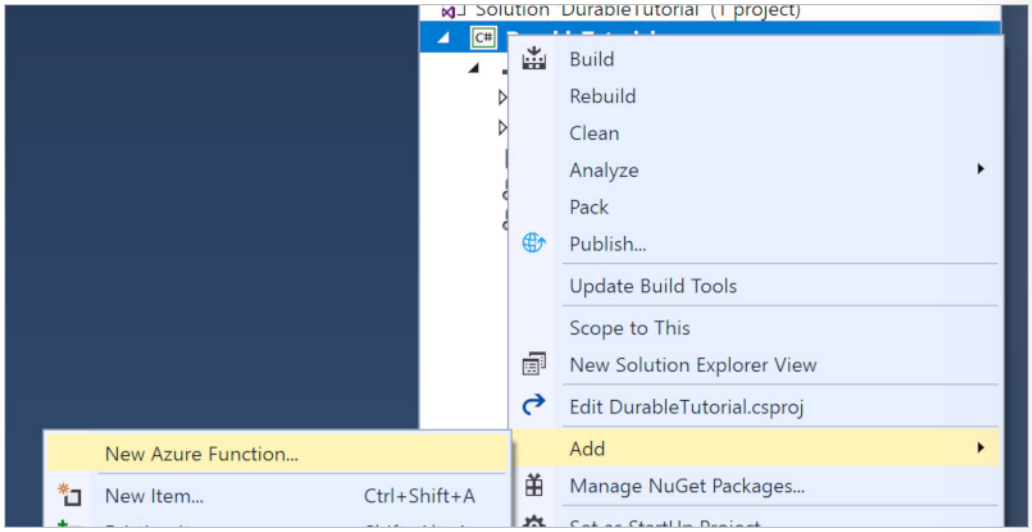
Setting	Suggested value	Description
Version	Azure Functions 2.x (.NET Core)	Creates a function project that uses the version 2.x runtime of Azure Functions which supports .NET Core. Azure Functions 1.x supports the .NET Framework.
Template	Empty	This creates an empty function app.
Storage account	Storage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

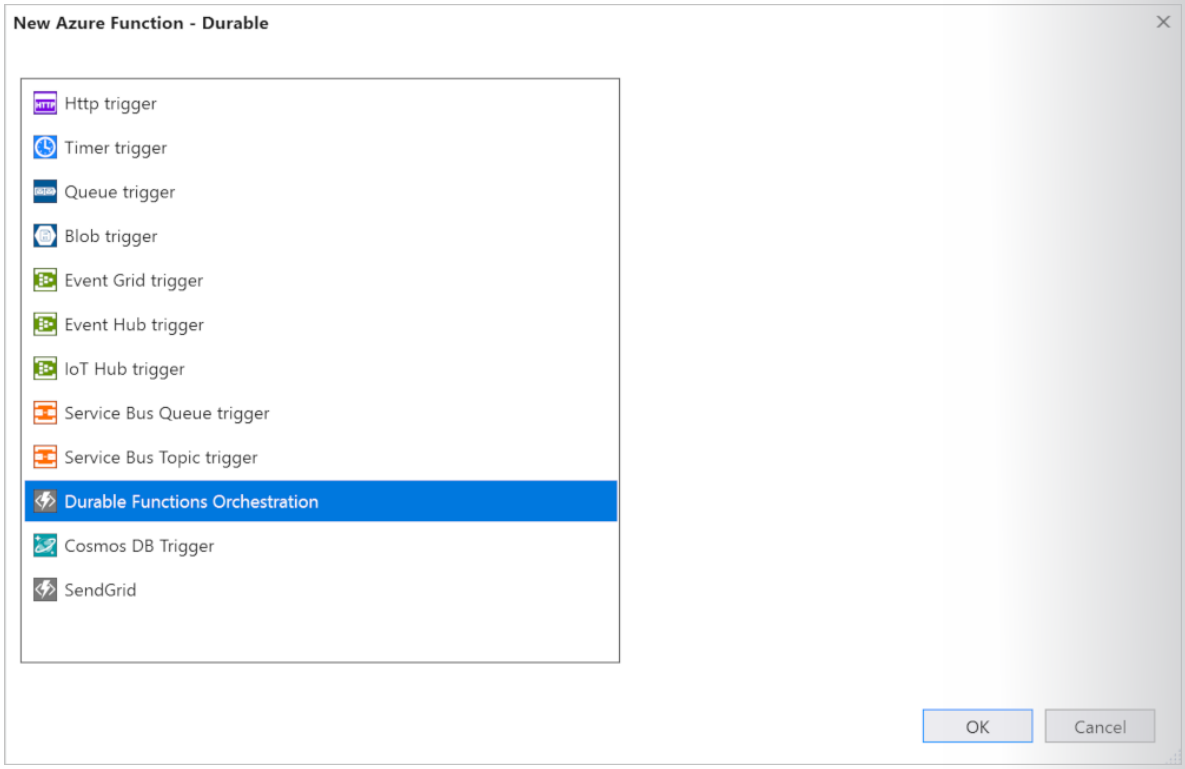
Visual Studio creates an empty function app project. It contains the basic configuration files needed for an app, but doesn't yet contain any functions. We'll need to add a durable function template to the project.

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.

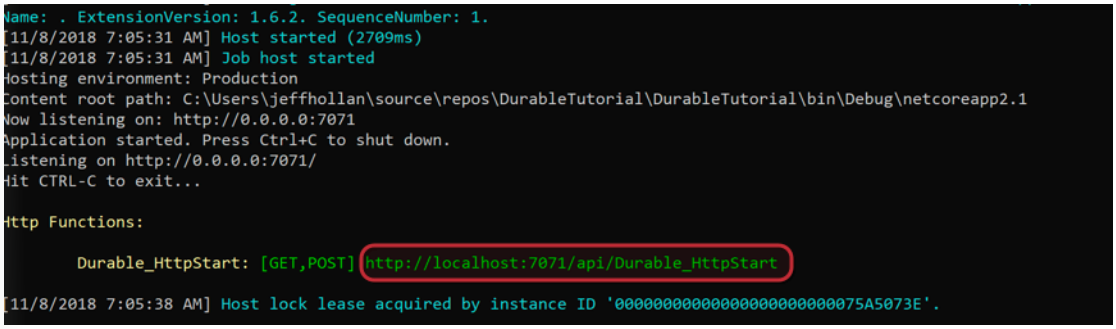
- The `HttpStart` method describes the function that will start instances of the orchestration. It is associated with an HTTP trigger that will start a new instance of the orchestrator and return back a check status response.

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

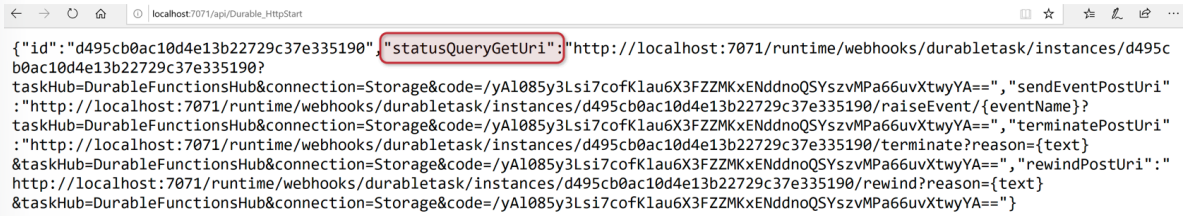
Test the function locally

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.



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:



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"
}
```

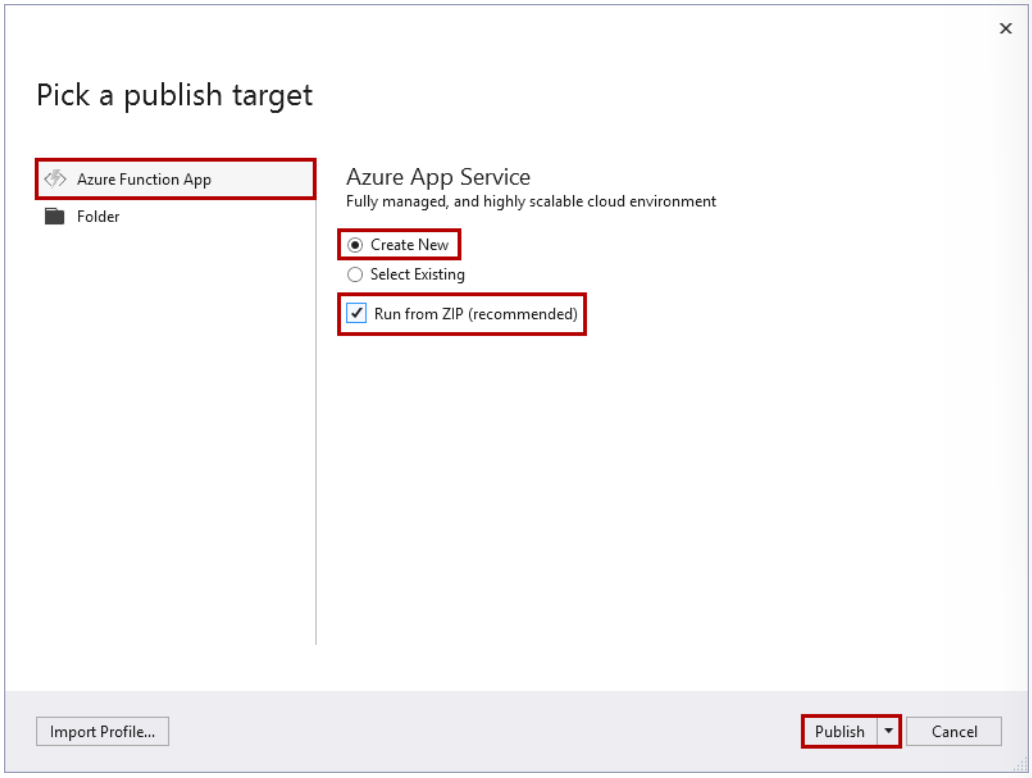
9. To stop debugging, press **Shift + F5**.

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

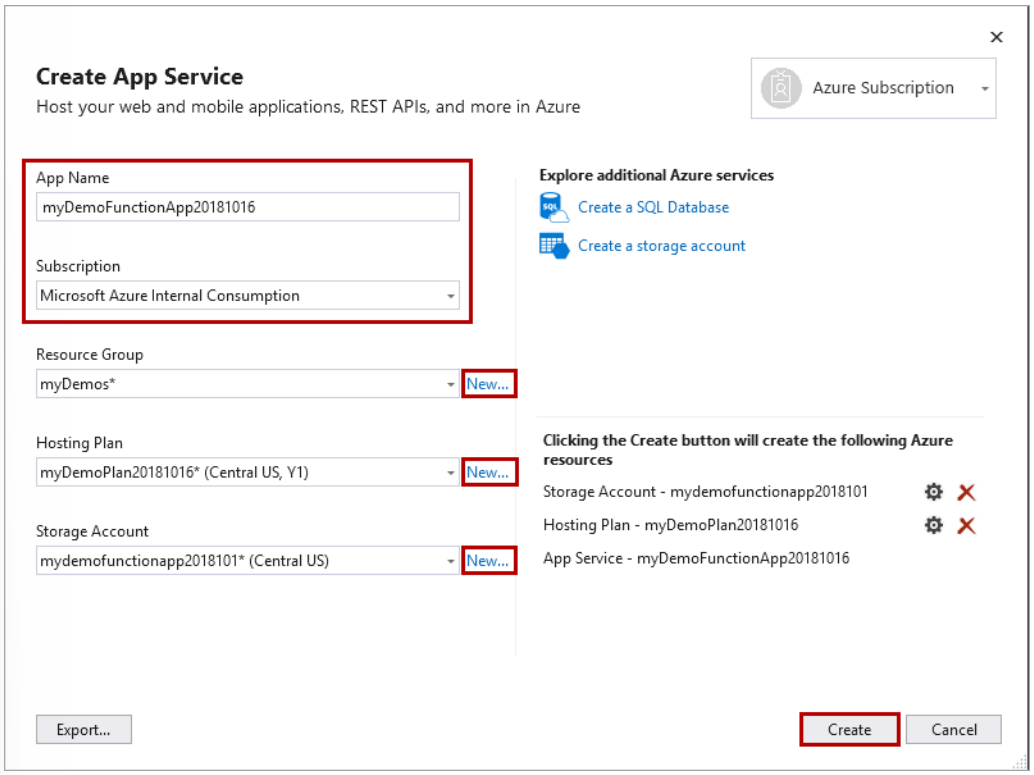
You must have a function app in your Azure subscription before you can publish your project. You can create a function app right from Visual Studio.

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:

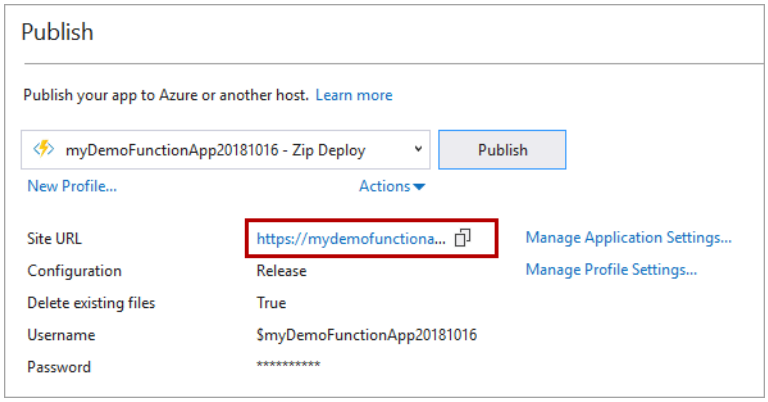


8.

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 regionnear 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.

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



11.

Test your function in Azure

1. 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.
2. The URL that calls your durable function HTTP trigger should be in the following format:

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

3. 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['AzureWeb-
JobsStorage']);

    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/b4e9bdc-
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","out-
put":452071,"createdTime":"2017-06-29T18:50:55Z","lastUpdated-
Time":"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 backupCon-
text)
        {
            string rootDirectory = backupContext.GetInput<string>()?.
Trim();
            if (string.IsNullOrEmpty(rootDirectory))
            {
                rootDirectory = Directory.GetParent(typeof(BackupSiteCon-
tent).Assembly.Location).FullName;
            }

            string[] files = await backupContext.CallActivityAsyn-
c<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($"Searching for files under '{rootDirecto-
ry}'...");
            string[] files = Directory.GetFiles(rootDirectory, "*",
SearchOption.AllDirectories);
            log.LogInformation($"Found {files.Length} file(s) under {rootDi-
rectory}.");

            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, File-
Access.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 2 Review Questions

Azure Redis Cache

Azure Redis Cache is a managed service based on Redis Cache that provides you secure nodes as a service. There are only two tiers for this service currently available. What are they and how do they differ?

> Click to see suggested answer

- **Basic:** Includes a single node.
- **Standard:** Includes two nodes in the Primary/Replica configuration and also includes replication support and a Service Level Agreement (SLA).

Normalized units

How can normalized units help you determine, and plan for, your database needs?

> Click to see suggested answer

In a world of hyperscale database services, it can be difficult to determine how much performance you need or how powerful an allocated database is. To help ease this challenge, many cloud vendors have provided normalized units of measurements that can be used to compare database tiers.

For example, if your application uses 20 database units today, 40 database units will guarantee you approximately double your performance, while 10 database units will guarantee you half of your performance.