# Azure Resource Manager

When a user sends a request from any of the Azure tools, APIs, or SDKs, Resource Manager receives the request. It authenticates and authorizes the request. Resource Manager sends the request to the Azure service, which takes the requested action. Because all requests are handled through the same API, you see consistent results and capabilities in all the different tools.

The following image shows the role Azure Resource Manager plays in handling Azure requests.



## Why choose Azure Resource Manager templates?

If you're trying to decide between using Azure Resource Manager templates and one of the other infrastructure as code services, consider the following advantages of using templates:

* **Declarative syntax**: Azure Resource Manager templates allow you to create and deploy an entire Azure infrastructure declaratively. For example, you can deploy not only virtual machines, but also the network infrastructure, storage systems, and any other resources you may need.
* **Repeatable results**: Repeatedly deploy your infrastructure throughout the development lifecycle and have confidence your resources are deployed in a consistent manner. Templates are idempotent, which means you can deploy the same template many times and get the same resource types in the same state. You can develop one template that represents the desired state, rather than developing lots of separate templates to represent updates.
* **Orchestration**: You don't have to worry about the complexities of ordering operations. Resource Manager orchestrates the deployment of interdependent resources so they're created in the correct order. When possible, Resource Manager deploys resources in parallel so your deployments finish faster than serial deployments. You deploy the template through one command, rather than through multiple imperative commands.

## Template file

Within your template, you can write template expressions that extend the capabilities of JSON. These expressions make use of the [functions](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/template-functions) provided by Resource Manager.

The template has the following sections:

* [Parameters](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/parameters) - Provide values during deployment that allow the same template to be used with different environments.
* [Variables](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/variables) - Define values that are reused in your templates. They can be constructed from parameter values.
* [User-defined functions](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/user-defined-functions) - Create customized functions that simplify your template.
* [Resources](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/resource-declaration) - Specify the resources to deploy.
* [Outputs](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/outputs) - Return values from the deployed resources.

# Deploy multi-tiered solutions

Completed 100 XP

* 3 minutes

With Resource Manager, you can create a template (in JSON format) that defines the infrastructure and configuration of your Azure solution. By using a template, you can repeatedly deploy your solution throughout its lifecycle and have confidence your resources are deployed in a consistent state.

When you deploy a template, Resource Manager converts the template into REST API operations. For example, when Resource Manager receives a template with the following resource definition:

JSON

"resources": [

{

"type": "Microsoft.Storage/storageAccounts",

"apiVersion": "2019-04-01",

"name": "mystorageaccount",

"location": "westus",

"sku": {

"name": "Standard\_LRS"

},

"kind": "StorageV2",

"properties": {}

}

]

It converts the definition to the following REST API operation, which is sent to the Microsoft.Storage resource provider:

HTTP

PUT

https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Storage/storageAccounts/mystorageaccount?api-version=2019-04-01

REQUEST BODY

{

"location": "westus",

"sku": {

"name": "Standard\_LRS"

},

"kind": "StorageV2",

"properties": {}

}

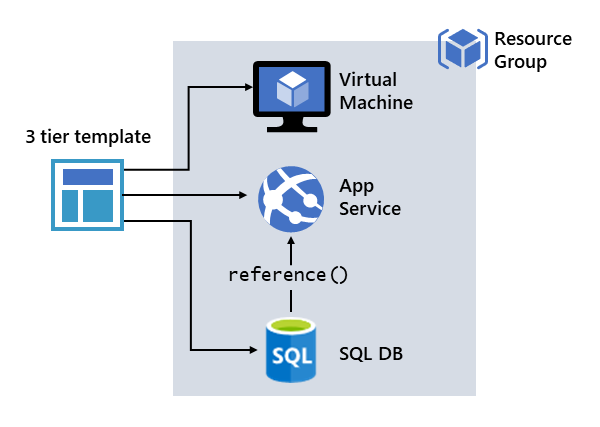
Notice that the **apiVersion** you set in the template for the resource is used as the API version for the REST operation. You can repeatedly deploy the template and have confidence it will continue to work. By using the same API version, you don't have to worry about breaking changes that might be introduced in later versions.

You can deploy a template using any of the following options:

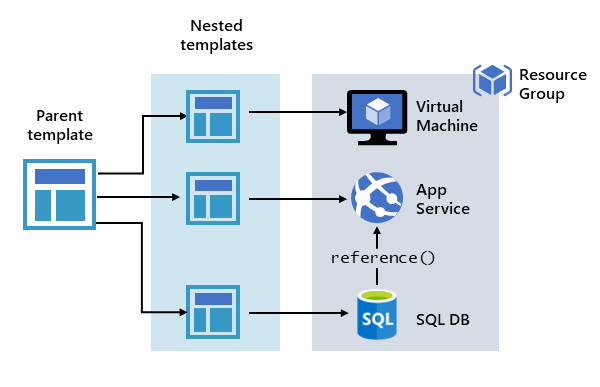
* Azure portal
* Azure CLI
* PowerShell
* REST API
* Button in GitHub repository
* Azure Cloud Shell

## Defining multi-tiered templates

How you define templates and resource groups is entirely up to you and how you want to manage your solution. For example, you can deploy a three tier application through a single template to a single resource group.



But, you don't have to define your entire infrastructure in a single template. Often, it makes sense to divide your deployment requirements into a set of targeted, purpose-specific templates. You can easily reuse these templates for different solutions. To deploy a particular solution, you create a master template that links all the required templates. The following image shows how to deploy a three tier solution through a parent template that includes three nested templates.



If you envision your tiers having separate lifecycles, you can deploy your three tiers to separate resource groups. The resources can still be linked to resources in other resource groups.

Azure Resource Manager analyzes dependencies to ensure resources are created in the correct order. If one resource relies on a value from another resource (such as a virtual machine needing a storage account for disks), you set a dependency. For more information, see Defining dependencies in Azure Resource Manager templates.

You can also use the template for updates to the infrastructure. For example, you can add a resource to your solution and add configuration rules for the resources that are already deployed. If the template specifies creating a resource but that resource already exists, Azure Resource Manager performs an update instead of creating a new asset. Azure Resource Manager updates the existing asset to the same state as it would be as new.

Resource Manager provides extensions for scenarios when you need additional operations such as installing particular software that isn't included in the setup. If you're already using a configuration management service, like DSC, Chef or Puppet, you can continue working with that service by using extensions.

Finally, the template becomes part of the source code for your app. You can check it in to your source code repository and update it as your app evolves. You can edit the template through Visual Studio.

## Share templates

After creating your template, you may wish to share it with other users in your organization. [Template specs](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/template-specs) enable you to store a template as a resource type. You use role-based access control to manage access to the template spec. Users with read access to the template spec can deploy it, but not change the template.

This approach means you can safely share templates that meet your organization's standards.

# Explore conditional deployment

Sometimes you need to optionally deploy a resource in an Azure Resource Manager template (Azure Resource Manager template). Use the condition element to specify whether the resource is deployed. The value for the condition resolves to true or false. When the value is true, the resource is created. When the value is false, the resource isn't created. The value can only be applied to the whole resource.

Note

Conditional deployment doesn't cascade to [child resources](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/child-resource-name-type). If you want to conditionally deploy a resource and its child resources, you must apply the same condition to each resource type.

## New or existing resource

You can use conditional deployment to create a new resource or use an existing one. The following example shows how to use condition to deploy a new storage account or use an existing storage account. It contains a parameter named newOrExisting which is used as a condition in the resources section.

JSON

{

"$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",

"contentVersion": "1.0.0.0",

"parameters": {

"storageAccountName": {

"type": "string"

},

"location": {

"type": "string",

"defaultValue": "[resourceGroup().location]"

},

"newOrExisting": {

"type": "string",

"defaultValue": "new",

"allowedValues": [

"new",

"existing"

]

}

},

"functions": [],

"resources": [

{

"condition": "[equals(parameters('newOrExisting'), 'new')]",

"type": "Microsoft.Storage/storageAccounts",

"apiVersion": "2019-06-01",

"name": "[parameters('storageAccountName')]",

"location": "[parameters('location')]",

"sku": {

"name": "Standard\_LRS",

"tier": "Standard"

},

"kind": "StorageV2",

"properties": {

"accessTier": "Hot"

}

}

]

}

When the parameter **newOrExisting** is set to **new**, the condition evaluates to true. The storage account is deployed. However, when **newOrExisting** is set to **existing**, the condition evaluates to false and the storage account isn't deployed.

## Runtime functions

If you use a reference or list function with a resource that is conditionally deployed, the function is evaluated even if the resource isn't deployed. You get an error if the function refers to a resource that doesn't exist.

Use the if function to make sure the function is only evaluated for conditions when the resource is deployed.

You set a resource as dependent on a conditional resource exactly as you would any other resource. When a conditional resource isn't deployed, Azure Resource Manager automatically removes it from the required dependencies.

# Set the correct deployment mode

Completed 100 XP

* 3 minutes

When deploying your resources, you specify that the deployment is either an incremental update or a complete update. The difference between these two modes is how Resource Manager handles existing resources in the resource group that aren't in the template. The default mode is incremental.

For both modes, Resource Manager tries to create all resources specified in the template. If the resource already exists in the resource group and its settings are unchanged, no operation is taken for that resource. If you change the property values for a resource, the resource is updated with those new values. If you try to update the location or type of an existing resource, the deployment fails with an error. Instead, deploy a new resource with the location or type that you need.

## Complete mode

In complete mode, Resource Manager **deletes** resources that exist in the resource group that aren't specified in the template.

If your template includes a resource that isn't deployed because condition evaluates to false, the result depends on which REST API version you use to deploy the template. If you use a version earlier than 2019-05-10, the resource **isn't deleted**. With 2019-05-10 or later, the resource **is deleted**. The latest versions of Azure PowerShell and Azure CLI delete the resource.

Be careful using complete mode with copy loops. Any resources that aren't specified in the template after resolving the copy loop are deleted.

## Incremental mode

In incremental mode, Resource Manager **leaves unchanged** resources that exist in the resource group but aren't specified in the template.

However, when redeploying an existing resource in incremental mode, the outcome is different. Specify all properties for the resource, not just the ones you're updating. A common misunderstanding is to think properties that aren't specified are left unchanged. If you don't specify certain properties, Resource Manager interprets the update as overwriting those values.

## Example result

To illustrate the difference between incremental and complete modes, consider the following table.

| **Resource Group contains** | **Template contains** | **Incremental result** | **Complete result** |
| --- | --- | --- | --- |
| Resource A Resource B Resource C | Resource A Resource B Resource D | Resource A Resource B Resource C Resource D | Resource A Resource B Resource D |

When deployed in **incremental** mode, Resource D is added to the existing resource group. When deployed in **complete** mode, Resource D is added and Resource C is deleted.

## Set deployment mode

To set the deployment mode when deploying with PowerShell, use the Mode parameter.

PowerShell

New-AzResourceGroupDeployment `

-Mode Complete `

-Name ExampleDeployment `

-ResourceGroupName ExampleResourceGroup `

-TemplateFile c:\MyTemplates\storage.json

To set the deployment mode when deploying with Azure CLI, use the mode parameter.

Azure CLI

az deployment group create \

--mode Complete \

--name ExampleDeployment \

--resource-group ExampleResourceGroup \

--template-file storage.json

# Azure App Service

Completed 100 XP

* 3 minutes

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. You can develop in your favorite programming language, be it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Applications run and scale with ease on both Windows and Linux-based environments.

## Built-in auto scale support

Baked into Azure App Service is the ability to scale up/down or scale out/in. Depending on the usage of the web app, you can scale the resources of the underlying machine that is hosting your web app up/down . Resources include the number of cores or the amount of RAM available. Scaling out/in is the ability to increase, or decrease, the number of machine instances that are running your web app.

## Continuous integration/deployment support

The Azure portal provides out-of-the-box continuous integration and deployment with Azure DevOps, GitHub, Bitbucket, FTP, or a local Git repository on your development machine. 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.

## Deployment slots

When you deploy your web app, web app on Linux, mobile back end, or API app to Azure App Service, you can use a separate deployment slot instead of the default production slot when you're running in the Standard, Premium, or Isolated App Service plan tier. Deployment slots are live apps with their own host names. App content and configurations elements can be swapped between two deployment slots, including the production slot.

## App Service on Linux

App Service can also host web apps natively on Linux for supported application stacks. It can also run custom Linux containers (also known as Web App for Containers). App Service on Linux supports a number of language specific built-in images. Just deploy your code. Supported languages include: Node.js, Java (JRE 8 & JRE 11), PHP, Python, .NET Core, and Ruby. If the runtime your application requires is not supported in the built-in images, you can deploy it with a custom container.

The languages, and their supported versions, are updated on a regular basis. You can retrieve the current list by using the following command in the Cloud Shell.

Bash

az webapp list-runtimes --os-type linux

### Limitations

App Service on Linux does have some limitations:

* App Service on Linux is not supported on Shared pricing tier.
* You can't mix Windows and Linux apps in the same App Service plan.
* Historically, you could not mix Windows and Linux apps in the same resource group. However, all resource groups created on or after January 21, 2021 do support this scenario. Support for resource groups created before January 21, 2021 will be rolled out across Azure regions (including National cloud regions) soon.
* The Azure portal shows only features that currently work for Linux apps. As features are enabled, they're activated on the portal.

# Examine Azure App Service plans

Completed 100 XP

* 3 minutes

In App Service, an app (Web Apps, API Apps, or Mobile Apps) always runs in an App Service plan. An App Service plan defines a set of compute resources for a web app to run. One or more apps can be configured to run on the same computing resources (or in the same App Service plan). In addition, Azure Functions also has the option of running in an 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, PremiumV3, Isolated)

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**: Both **Free** and **Shared** share the resource pools of your apps with the apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources can't scale out.
* **Dedicated compute**: The **Basic**, **Standard**, **Premium**, **PremiumV2**, and **PremiumV3** 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. It 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 virtual machines as other App Service apps. Some apps might belong to other customers. These tiers are intended to be used only for development and testing purposes.

## How does my app run and scale?

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

* An app 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. 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.

You can potentially save money by putting multiple apps into one App Service plan. However, since apps in the same App Service plan all share the same compute resources you need to understand the capacity of the existing App Service plan and the expected load for the new app.

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 in 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.

# Deploy to App Service

Completed 100 XP

* 3 minutes

Every development team has unique requirements that can make implementing an efficient deployment pipeline difficult on any cloud service. App Service supports both automated and manual deployment.

## Automated deployment

Automated deployment, or continuous integration, is a process used to push out new features and bug fixes in a fast and repetitive pattern with minimal impact on end users.

Azure supports automated deployment directly from several sources. The following options are available:

* **Azure DevOps**: You can push your code to Azure DevOps, build your code in the cloud, run the tests, generate a release from the code, and finally, push your code to an Azure Web App.
* **GitHub**: Azure supports automated deployment directly from GitHub. When you connect your GitHub repository to Azure for automated deployment, any changes you push to your production branch on GitHub will be automatically deployed for you.
* **Bitbucket**: With its similarities to GitHub, you can configure an automated deployment with Bitbucket.

## Manual deployment

There are a few options that you can use to manually push your code to Azure:

* **Git**: App Service web apps feature a Git URL that you can add as a remote repository. Pushing to the remote repository will deploy your app.
* **CLI**: webapp up is a feature of the az command-line interface that packages your app and deploys it. Unlike other deployment methods, az webapp up can create a new App Service web app for you if you haven't already created one.
* **Zip deploy**: Use curl or a similar HTTP utility to send a ZIP of your application files to App Service.
* **FTP/S**: FTP or FTPS is a traditional way of pushing your code to many hosting environments, including App Service.

## Use deployment slots

Whenever possible, use deployment slots when deploying a new production build. When using a Standard App Service Plan tier or better, you can deploy your app to a staging environment and then swap your staging and production slots. The swap operation warms up the necessary worker instances to match your production scale, thus eliminating downtime.