**Bicep code by using Git**

**git add .**

**git commit --message "Add Cosmos DB account definition"**

After you commit files to your Git repository, you can use the git log CLI command to view the history of a file or even all the commits to the repository.

To view a list of commits, run the following command:

**git log --pretty=oneline**

It's also common to view the commits to a specific file. You can specify the file name when you run the git log command, like this:

**git log deploy/main.bicep**

**git checkout main**

**git merge my-experimental-changes**

**git branch -d my-experimental-changes**

**git remote add origin YOUR\_REPOSITORY\_URL**

**git remote -v**

**git push -u origin main**

**--------**

**git checkout main**

**git merge add-orders-queue**

If you want to reset your changes

**git reset --hard HEAD~1**

* **Connect-AzAccount**
* **$context = Get-AzSubscription -SubscriptionId {Your subscription ID}**
* **Set-AzContext $context**
* **Set-AzDefault -ResourceGroupName [sandbox resource group name]**
* **New-AzResourceGroupDeployment -TemplateFile main.bicep**
* **New-AzResourceGroupDeployment -WhatIf -TemplateFile main.bicep**
* **New-AzResourceGroupDeployment -Mode Complete -Confirm -TemplateFile main.bicep**
* **bicep build main.bicep (the validation and linting)**

**Youtube link:**

https://www.youtube.com/live/MP60ND7Upn4?si=6DU\_2TQRO-O-Kpct

**Build your first Bicep template**

**What is Bicep?**

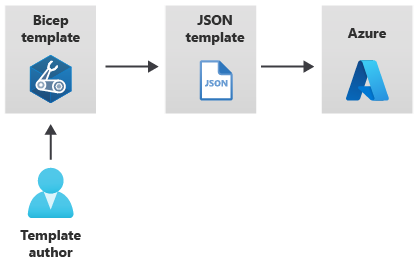
Bicep is a language for declaratively deploying Azure resources. Using Bicep, you can define how your Azure resources should be configured and deployed. You'll define your resources within a Bicep file called a *template*, then submit the template to Azure Resource Manager. Resource Manager then takes responsibility for deploying each resource within the template on your behalf.

**How is Bicep related to ARM templates?**

You might already be familiar with Azure Resource Manager templates (ARM templates), which are files that represent Azure resources. Until Bicep was available, ARM templates had to be written in a special JSON format. One common problem with JSON templates is that they're difficult to work with because they have a complex syntax. It can be hard to get started with writing ARM templates in JSON.

Bicep solves these problems by using a much simpler language designed specifically to help you deploy resources to Azure.

Behind the scenes, Resource Manager still operates based on the same JSON templates. When you submit a Bicep template to Resource Manager, the Bicep tooling converts your template to a JSON format in a process called *transpilation*. This process isn't something you typically have to think about, but you can view the JSON template file that's created from the Bicep file.



**Define a resource:**

The main thing you'll do with Bicep templates is define your Azure resources.

BicepCopy

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' = {

name: 'toylaunchstorage'

location: 'westus3'

sku: {

name: 'Standard\_LRS'

}

kind: 'StorageV2'

properties: {

accessTier: 'Hot'

}

}

Let's look closely at some key parts of this resource definition:

* The resource keyword at the start tells Bicep that you're about to define a resource.
* Next, you give the resource a *symbolic name*. In the example, the resource's symbolic name is storageAccount. Symbolic names are used within Bicep to refer to the resource, but they won't ever show up in Azure.
* Microsoft.Storage/storageAccounts@2022-09-01 is the *resource type* and *API version* of the resource. Microsoft.Storage/storageAccounts tells Bicep that you're declaring an Azure storage account. The date 2022-09-01 is the version of the Azure Storage API that Bicep uses when it creates the resource.
* You have to declare a *resource name*, which is the name the storage account will be assigned in Azure. You'll set a resource name using the name keyword.

**Add a parameter**

In Bicep, you can define a parameter like this:

BicepCopy

**param appServiceAppName string**

Let's look at how each part of this definition works:

* param tells Bicep that you're defining a parameter.
* appServiceAppName is the name of the parameter. If you're deploying the template manually, you might be asked to enter a value, so it's important that the name is clear and understandable. The name is also how you refer to the parameter value within the template, just like with resource symbolic names.
* string is the type of the parameter. You can specify several different types for Bicep parameters, including string for text, int for numbers, and bool for Boolean true or false values. You can also pass in more complex parameters by using the array and object types.

**Here's how you can add a default value:**

**param appServiceAppName string = 'toy-product-launch-1'**

**Use parameter values in the template:**

resource appServiceApp 'Microsoft.Web/sites@2024-04-01' = {

name: appServiceAppName

location: 'eastus'

properties: {

serverFarmId: appServicePlan.id

httpsOnly: true

}

}

**Add a variable:**

You can define a variable like this:

**var appServicePlanName = 'toy-product-launch-plan'**

Variables are defined in a similar way to parameters, but there are a few differences:

* Use the var keyword to tell Bicep you're declaring a variable.
* You must provide a value for a variable.
* Variables don't need types. Bicep can determine the type based on the value that you set.

Bicep has another function called uniqueString() that comes in handy when you're creating resource names. When you use this function, you need to provide a *seed value*, which should be different across different deployments, but consistent across all deployments for the same resources.

**param storageAccountName string = uniqueString(resourceGroup().id)**

**Combined strings:**

Bicep has a feature called *string interpolation* that lets you combine strings. Let's see how it works:

**param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'**

The default value for the storageAccountName parameter now has two parts to it:

* toylaunch is a hard-coded string that helps anyone who looks at the deployed resource in Azure to understand the storage account's purpose.
* ${uniqueString(resourceGroup().id)} is a way of telling Bicep to evaluate the output of the uniqueString(resourceGroup().id) function, then concatenate it into the string.

**@allowed([**

**'nonprod'**

**'prod'**

**])**

**param environmentType string**

**var storageAccountSkuName = (environmentType == 'prod') ? 'Standard\_GRS' : 'Standard\_LRS'**

**var appServicePlanSkuName = (environmentType == 'prod') ? 'P2V3' : 'F1'**

Notice some new syntax here, too. Let's break it down:

* (environmentType == 'prod') evaluates to a Boolean (true or false) value, depending on which allowed value is used for environmentType parameter.
* ? is called a *ternary operator*, and it evaluates an if/then statement. The value after the ? operator is used if the expression is true. If the expression evaluates to false, the value after the colon (:) is used.

**Verify your Bicep file**

*main.bicep* file should look like this example:

BicepCopy

param location string = 'eastus'

param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'

param appServiceAppName string = 'toylaunch${uniqueString(resourceGroup().id)}'

@allowed([

'nonprod'

'prod'

])

param environmentType string

var appServicePlanName = 'toy-product-launch-plan'

var storageAccountSkuName = (environmentType == 'prod') ? 'Standard\_GRS' : 'Standard\_LRS'

var appServicePlanSkuName = (environmentType == 'prod') ? 'P2v3' : 'F1'

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' = {

name: storageAccountName

location: location

sku: {

name: storageAccountSkuName

}

kind: 'StorageV2'

properties: {

accessTier: 'Hot'

}

}

resource appServicePlan 'Microsoft.Web/serverfarms@2024-04-01' = {

name: appServicePlanName

location: location

sku: {

name: appServicePlanSkuName

}

}

resource appServiceApp 'Microsoft.Web/sites@2024-04-01' = {

name: appServiceAppName

location: location

properties: {

serverFarmId: appServicePlan.id

httpsOnly: true

}

}

**Deploy the updated Bicep template**

Run the following Azure PowerShell command in the terminal.

Azure PowerShellCopy

New-AzResourceGroupDeployment `

-Name main `

-TemplateFile main.bicep `

-environmentType nonprod

**Outputs:**

To define an output in a Bicep template, use the output keyword like this:

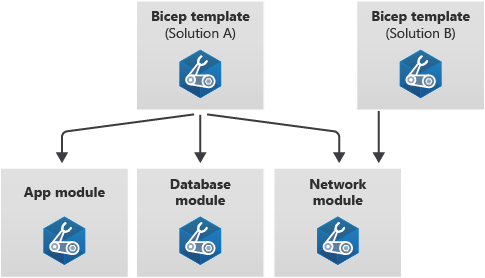
BicepCopy

**output appServiceAppName string = appServiceAppName**

The output definition includes a few key parts:

* The output keyword tells Bicep you're defining an output.
* appServiceAppName is the output's name. When someone deploys the template successfully, the output values include the name you specified so they can access the values they're expecting.
* string is the output type. Bicep outputs support the same types as parameters.
* A value must be specified for each output. Unlike parameters, outputs always need to have values. Output values can be expressions, references to parameters or variables, or properties of resources that are deployed within the file.

**Define a module:**



When you want the template to include a reference to a module file, use the module keyword. A module definition looks similar to a resource declaration, but instead of including a resource type and API version, you'll use the module's file name:

BicepCopy

module myModule 'modules/mymodule.bicep' = {

name: 'MyModule'

params: {

location: location

}

}

Let's look closely at some key parts of this module definition:

* The module keyword tells Bicep you're about to use another Bicep file as a module.
* Just like resources, modules need a *symbolic name* like myModule. You'll use the symbolic name when you refer to the module's outputs in other parts of the template.
* modules/mymodule.bicep is the path to the module file, relative to the template file. Remember, a module file is just a regular Bicep file.
* Just like resources, the name property is mandatory. Azure uses the module name because it creates a separate deployment for each module within the template file. Those deployments have names you can use to identify them.
* You can specify any *parameters* of the module by using the params keyword. When you set the values of each parameter within the template, you can use expressions, template parameters, variables, properties of resources deployed within the template, and outputs from other modules. Bicep will automatically understand the dependencies between the resources.

**Verify your Bicep files:**

After you've completed all of the preceding changes, your *main.bicep* file should look like this example:

Bicep

param location string = 'eastus'

param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'

param appServiceAppName string = 'toylaunch${uniqueString(resourceGroup().id)}'

@allowed([

'nonprod'

'prod'

])

param environmentType string

var storageAccountSkuName = (environmentType == 'prod') ? 'Standard\_GRS' : 'Standard\_LRS'

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' = {

name: storageAccountName

location: location

sku: {

name: storageAccountSkuName

}

kind: 'StorageV2'

properties: {

accessTier: 'Hot'

}

}

module appService 'modules/appService.bicep' = {

name: 'appService'

params: {

location: location

appServiceAppName: appServiceAppName

environmentType: environmentType

}

}

output appServiceAppHostName string = appService.outputs.appServiceAppHostName

Your *appService.bicep* file should look like this example:

BicepCopy

param location string

param appServiceAppName string

@allowed([

'nonprod'

'prod'

])

param environmentType string

var appServicePlanName = 'toy-product-launch-plan'

var appServicePlanSkuName = (environmentType == 'prod') ? 'P2v3' : 'F1'

resource appServicePlan 'Microsoft.Web/serverfarms@2024-04-01' = {

name: appServicePlanName

location: location

sku: {

name: appServicePlanSkuName

}

}

resource appServiceApp 'Microsoft.Web/sites@2024-04-01' = {

name: appServiceAppName

location: location

properties: {

serverFarmId: appServicePlan.id

httpsOnly: true

}

}

output appServiceAppHostName string = appServiceApp.properties.defaultHostName

**Deploy the updated Bicep template:**

Run the following Azure PowerShell command in the terminal.

**Azure PowerShellCopy**

New-AzResourceGroupDeployment `

-Name main `

-TemplateFile main.bicep `

-environmentType nonprod

**Build reusable Bicep templates by using parameters**

https://learn.microsoft.com/en-us/training/modules/build-reusable-bicep-templates-parameters/

With *parameters*, you can provide information to a Bicep template at deployment time. You can make a Bicep template flexible and reusable by declaring parameters within your template.

*Decorators* provide a way to attach constraints and metadata to a parameter, which helps anyone using your templates to understand what information they need to provide.

**Declare a parameter**

In a Bicep template, you declare a parameter by using the param keyword. You can put these declarations anywhere in the template file, although it's usually a good idea to put them at the top of the file so your Bicep code is easy to read.

Here's how you declare a parameter:

BicepCopy

**param environmentName string**

Let's look at how each part works:

* param indicates to Bicep that you're declaring a parameter.
* environmentName refers to the name of the parameter. Although the parameter name can be anything, you should make the name clear and understandable for the template users. Within the same template, you can also refer to the parameter by using its name. Parameter names must be unique. They can't have the same name as a variable or a resource in the same Bicep file.
* string refers to the type of the parameter.

**Add a default value**

You can assign default values to parameters in your templates. By assigning a default value, you're making the parameter optional. If the template is deployed without a specified value for the parameter, the default value is assigned.

Here's how you add a default value:

BicepCopy

**param environmentName string = 'dev'**

The parameter environmentName is assigned a default value of dev.

You can use expressions as default values. Here's an example of a string parameter named location with a default value set to the location of the current resource group:

BicepCopy

**param location string = resourceGroup().location**

**Understand parameter types**

When you declare a parameter, you need to tell Bicep what type of information the parameter will contain. Bicep will ensure that the value assigned to the parameter is compatible with the parameter type.

Parameters in Bicep can be one of the following types:

* string, which lets you enter arbitrary text.
* int, which lets you enter a number.
* bool, which represents a Boolean (true or false) value.
* object and array, which represent structured data and lists.

**Objects**

You can use object parameters to combine structured data together in one place. An object can have multiple properties of different types. You can use objects within resource definitions, within variables, or within expressions in your Bicep file. Here's an example of an object:

BicepCopy

param appServicePlanSku object = {

name: 'F1'

tier: 'Free'

capacity: 1

}

param resourceTags object = {

EnvironmentName: 'Test'

CostCenter: '1000100'

Team: 'Human Resources'

}

This parameter is an object with two string properties, name and tier, and an integer property, capacity. Notice that each of the properties is on its own line.

resource appServicePlan 'Microsoft.Web/serverfarms@2024-04-01' = {

name: appServicePlanName

location: location

tags: resourceTags

sku: {

name: appServicePlanSku.name

tier: appServicePlanSku.tier

capacity: appServicePlanSku.capacity

}

}

resource appServiceApp 'Microsoft.Web/sites@' = {

name: appServiceAppName

location: location

tags: resourceTags

kind: 'app'

properties: {

serverFarmId: appServicePlan.id

}

}

**Arrays**

An array is a list of items. As an example, you might use an array of string values to declare a list of email addresses for an Azure Monitor action group. Or you might use an array of objects to represent a list of subnets for a virtual network

You can create an array parameter that specifies a list of locations:

param cosmosDBAccountLocations array = [

{

locationName: 'australiaeast'

}

{

locationName: 'southcentralus'

}

{

locationName: 'westeurope'

}

]

resource account 'Microsoft.DocumentDB/databaseAccounts@2024-11-15' = {

name: accountName

location: location

properties: {

locations: cosmosDBAccountLocations

}

}

**Specify a list of allowed values**

Sometimes you need to make sure that a parameter has certain values. you can use the @allowed parameter decorator.

BicepCopy

**@allowed([**

**'P1v3'**

**'P2v3'**

**'P3v3'**

**])**

**param appServicePlanSkuName string**

**Restrict parameter length and values**

When you use string parameters, you often need to limit the length of the string.

You can use the @minLength and @maxLength decorators to the minimum and maximum character lengths that you want to allow for a parameter.

Here's an example that declares a string parameter named storageAccountName, whose length can only be between 5 and 24 characters:

BicepCopy

**@minLength(5)**

**@maxLength(24)**

**param storageAccountName string**

**Note**

You can also apply the @minLength and @maxLength decorators to array parameters, to control how many items are allowed to be in the array.

When you work with numeric parameters, you might need values to be in a particular range. For example, your toy company might decide that whenever anybody deploys an App Service plan, they should always deploy at least one instance, but no more than 10 instances of the plan. To meet the requirements, you can use the @minValue and @maxValue decorators to specify the minimum and maximum allowed values. The following example declares the integer parameter appServicePlanInstanceCount whose value can only be between 1 and 10 (inclusive):

BicepCopy

**@minValue(1)**

**@maxValue(10)**

**param appServicePlanInstanceCount int**

**Add descriptions to parameters**

Bicep provides the @description decorator so that you can document your parameters' purpose in a human-readable way.

BicepCopy

**@description('**The locations into which this Cosmos DB account should be configured. This parameter needs to be a list of objects, each of which has a locationName property**.')**

**param cosmosDBAccountLocations array**

**Verify your Bicep file**

After you've completed all of the preceding changes, your Bicep file should look like this example:

BicepCopy

@description('The name of the environment. This must be dev, test, or prod.')

@allowed([

'dev'

'test'

'prod'

])

param environmentName string = 'dev'

@description('The unique name of the solution. This is used to ensure that resource names are unique.')

@minLength(5)

@maxLength(30)

param solutionName string = 'toyhr${uniqueString(resourceGroup().id)}'

@description('The number of App Service plan instances.')

@minValue(1)

@maxValue(10)

param appServicePlanInstanceCount int = 1

@description('The name and tier of the App Service plan SKU.')

param appServicePlanSku object = {

name: 'F1'

tier: 'Free'

}

@description('The Azure region into which the resources should be deployed.')

param location string = 'eastus'

var appServicePlanName = '${environmentName}-${solutionName}-plan'

var appServiceAppName = '${environmentName}-${solutionName}-app'

resource appServicePlan 'Microsoft.Web/serverfarms@2024-04-01' = {

name: appServicePlanName

location: location

sku: {

name: appServicePlanSku.name

tier: appServicePlanSku.tier

capacity: appServicePlanInstanceCount

}

}

resource appServiceApp 'Microsoft.Web/sites@2024-04-01' = {

name: appServiceAppName

location: location

properties: {

serverFarmId: appServicePlan.id

httpsOnly: true

}

}

**Provide values using parameter files:**

**Create parameters files**

Parameters files make it easy to specify parameter values together as a set. Within the parameters file, you provide values for the parameters in your Bicep file. parameters files are created by using a Bicep parameters file with the .bicepparam file extension or a JSON parameters file that contains the parameter values. You can supply a parameters file when you deploy your Bicep template.

{

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

"contentVersion": "1.0.0.0",

"parameters": {

"appServicePlanInstanceCount": {

"value": 3

},

"appServicePlanSku": {

"value": {

"name": "P1v3",

"tier": "PremiumV3"

}

},

"cosmosDBAccountLocations": {

"value": [

{

"locationName": "australiaeast"

},

{

"locationName": "southcentralus"

},

{

"locationName": "westeurope"

}

]

}

}

}

Let's look at each part of the parameters file in more detail:

* $schema helps Azure Resource Manager to understand that this file is a parameters file.
* contentVersion is a property that you can use to keep track of significant changes in your parameters file if you want. Usually, it's set to its default value of 1.0.0.0.
* The parameters section lists each parameter and the value you want to use. The parameter value must be specified as an object. The object has a property called value that defines the actual parameter value to use.

Generally, you'll create a parameters file for each environment. It's a good practice to include the environment name in the name of the parameters file. For example, you might have a parameters file named main.parameters.dev.json for your development environment and one named main.parameters.production.json for your production environment.

**Use parameters files at deployment time**

When you create a new deployment by using the New-AzResourceGroupDeployment cmdlet, you can specify the name of the parameters file you want to use with the -TemplateParameterFile argument:

Azure PowerShellCopy

New-AzResourceGroupDeployment `

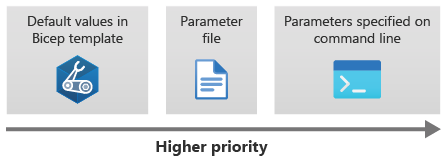
-Name main `

-TemplateFile main.bicep `

-TemplateParameterFile main.parameters.json

**Override parameter values**

You've now learned about three ways to specify parameter values: default values, the command line, and parameters files. It's common to use different approaches to specify different values for the same parameter. You've already seen this approach when you worked with default values. When you create a default value for a parameter, but then specify a different value by using the command line, the command-line value takes precedence. Let's look at how parameters files fit into this order of precedence.



You can see that parameters files override default values, and command-line parameter values override parameters files.

Let's see how this approach works. Here's an example Bicep file that defines three parameters, each with default values:

BicepCopy

param location string = resourceGroup().location

param appServicePlanInstanceCount int = 1

param appServicePlanSku object = {

name: 'F1'

tier: 'Free'

}

Let's look at a parameters file that overrides the value of two of the parameters but doesn't specify a value for the location parameter:

JSONCopy

{

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

"contentVersion": "1.0.0.0",

"parameters": {

"appServicePlanInstanceCount": {

"value": 3

},

"appServicePlanSku": {

"value": {

"name": "P1v3",

"tier": "PremiumV3"

}

}

}

}

When you create the deployment, you override one of the parameter values. You specify the parameter name as if it's an argument to the cmdlet:

Azure PowerShellCopy

New-AzResourceGroupDeployment `

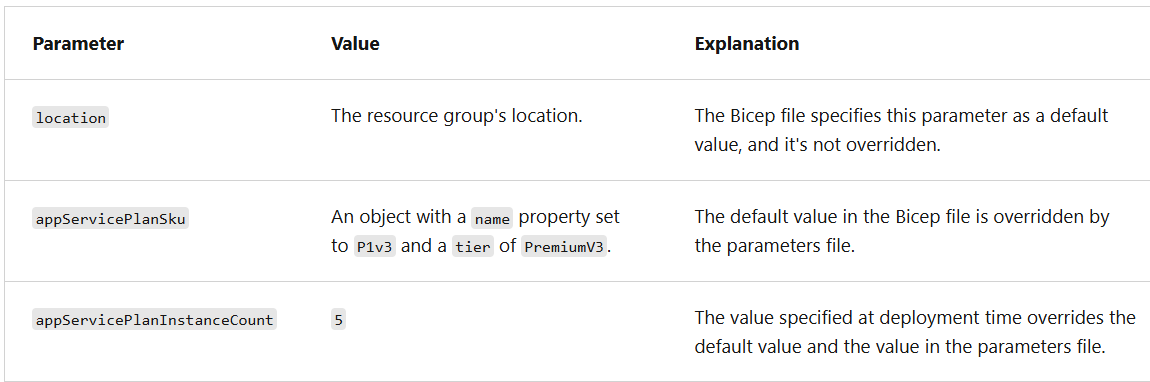
-Name main `

-TemplateFile main.bicep `

-TemplateParameterFile main.parameters.json `

-appServicePlanInstanceCount 5

Let's look at what the values will be.



By using a mixture of the approaches to specify parameter values, you can avoid having to duplicate parameter values in lots of places, while still getting the flexibility to override where you need to.

**Define secure parameters**

The @secure decorator can be applied to string and object parameters that might contain secret values. When you define a parameter as @secure, Azure won't make the parameter values available in the deployment logs. Also, if you create the deployment interactively by using the Azure CLI or Azure PowerShell and you need to enter the values during the deployment, the terminal won't display the text on your screen.

BicepCopy

@secure()

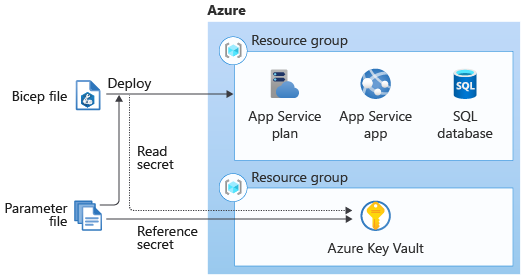
param sqlServerAdministratorLogin string

@secure()

param sqlServerAdministratorPassword string

**Integrate with Azure Key Vault**

Azure Key Vault is a service designed to store and provide access to secrets. You can integrate your Bicep templates with Key Vault by using a parameters file with a reference to a Key Vault secret.



Here's a parameters file that uses Key Vault references to look up the SQL logical server administrator login and password to use:

JSONCopy

{

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

"contentVersion": "1.0.0.0",

"parameters": {

"sqlServerAdministratorLogin": {

"reference": {

"keyVault": {

"id": "/subscriptions/f0750bbe-ea75-4ae5-b24d-a92ca601da2c/resourceGroups/PlatformResources/providers/Microsoft.KeyVault/vaults/toysecrets"

},

"secretName": "sqlAdminLogin"

}

},

"sqlServerAdministratorPassword": {

"reference": {

"keyVault": {

"id": "/subscriptions/f0750bbe-ea75-4ae5-b24d-a92ca601da2c/resourceGroups/PlatformResources/providers/Microsoft.KeyVault/vaults/toysecrets"

},

"secretName": "sqlAdminLoginPassword"

}

}

}

}

**Use Key Vault with modules**

Modules may have parameters that accept secret values, and you can use Bicep's Key Vault integration to provide these values securely. Here's an example Bicep file that deploys a module and provides the value of the ApiKey secret parameter by taking it directly from Key Vault:

BicepCopy

resource keyVault 'Microsoft.KeyVault/vaults@2023-07-01' existing = {

name: keyVaultName

}

module applicationModule 'application.bicep' = {

name: 'application-module'

params: {

apiKey: keyVault.getSecret('ApiKey')

}

}

Notice that in this Bicep file, the Key Vault resource is referenced by using the existing keyword. The keyword tells Bicep that the Key Vault already exists, and this code is a reference to that vault. Bicep won't redeploy it. Also, notice that the module's code uses the getSecret() function in the value for the module's apiKey parameter. This is a special Bicep function that can only be used with secure module parameters. Internally, Bicep translates this expression to the same kind of Key Vault reference you learned about earlier.

**Build flexible Bicep templates by using conditions and loops**

<https://learn.microsoft.com/en-us/training/modules/build-flexible-bicep-templates-conditions-loops/1-introduction>

**Use basic conditions**

When you deploy a resource in Bicep, you can provide the **if** keyword followed by a condition. The condition should resolve to a Boolean (true or false) value. If the value is *true*, the resource is deployed. If the value is *false*, the resource is not deployed.

It's common to create conditions based on the values of parameters that you provide. For example, the following code deploys a storage account only when the deployStorageAccount parameter is set to true:

BicepCopy

**A close-up of a computer screen

Description automatically generated**Notice that the if keyword is on the same line as the resource definition.

**Use expressions as conditions**

The preceding example was quite basic. The deployStorageAccount parameter was of type bool, so it's clear whether it has a value of true or false.

In Bicep, conditions can also include expressions. In the following example, the code deploys a SQL auditing resource only when the environmentName parameter value is equal to Production:

BicepCopy

A close-up of a computer code

Description automatically generatedIt's usually a good idea to create a variable for the expression that you're using as a condition. That way, your template is easier to understand and read. Here's an example:

BicepCopy

**A yellow and red text on a white background

Description automatically generated**

**Depend on conditionally deployed resources**

When you deploy resources conditionally, you sometimes need to be aware of how Bicep evaluates the dependencies between them.

Let's continue writing some Bicep code to deploy SQL auditing settings. The Bicep file also needs to declare a storage account resource, as shown here:

**Note**

You can't define two resources with the same name in the same Bicep file and then conditionally deploy only one of them. The deployment will fail, because Resource Manager views this as a conflict.

If you have several resources, all with the same condition for deployment, consider using Bicep modules. You can create a module that deploys all the resources, then put a condition on the module declaration in your main Bicep file.

**Verify your Bicep file**

After you've completed all of the preceding changes, your Bicep file should look like this example:

BicepCopy

@description('The Azure region into which the resources should be deployed.')

param location string

@secure()

@description('The administrator login username for the SQL server.')

param sqlServerAdministratorLogin string

@secure()

@description('The administrator login password for the SQL server.')

param sqlServerAdministratorLoginPassword string

@description('The name and tier of the SQL database SKU.')

param sqlDatabaseSku object = {

name: 'Standard'

tier: 'Standard'

}

@description('The name of the environment. This must be Development or Production.')

@allowed([

'Development'

'Production'

])

param environmentName string = 'Development'

@description('The name of the audit storage account SKU.')

param auditStorageAccountSkuName string = 'Standard\_LRS'

var sqlServerName = 'teddy${location}${uniqueString(resourceGroup().id)}'

var sqlDatabaseName = 'TeddyBear'

var auditingEnabled = environmentName == 'Production'

var auditStorageAccountName = take('bearaudit${location}${uniqueString(resourceGroup().id)}', 24)

resource sqlServer 'Microsoft.Sql/servers@2024-05-01-preview' = {

name: sqlServerName

location: location

properties: {

administratorLogin: sqlServerAdministratorLogin

administratorLoginPassword: sqlServerAdministratorLoginPassword

}

}

resource sqlDatabase 'Microsoft.Sql/servers/databases@2024-05-01-preview' = {

parent: sqlServer

name: sqlDatabaseName

location: location

sku: sqlDatabaseSku

}

resource auditStorageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' = if (auditingEnabled) {

name: auditStorageAccountName

location: location

sku: {

name: auditStorageAccountSkuName

}

kind: 'StorageV2'

}

resource sqlServerAudit 'Microsoft.Sql/servers/auditingSettings@2024-05-01-preview' = if (auditingEnabled) {

parent: sqlServer

name: 'default'

properties: {

state: 'Enabled'

storageEndpoint: environmentName == 'Production' ? auditStorageAccount.properties.primaryEndpoints.blob : ''

storageAccountAccessKey: environmentName == 'Production' ? auditStorageAccount.listKeys().keys[0].value : ''

}

}

**Deploy the template for the production environment**

In the Visual Studio Code terminal, deploy the template to Azure by running the following Azure PowerShell command:

Azure PowerShellCopy

New-AzResourceGroupDeployment -TemplateFile main.bicep -environmentName Production -location westus3

**Deploy multiple resources by using loops:**

**Use copy loops**

When you define a resource or a module in a Bicep template, you can use the for keyword to create a loop. Place the for keyword in the resource declaration, then specify how you want Bicep to identify each item in the loop. Typically, you loop over an array of objects to create multiple instances of a resource. The following example deploys multiple storage accounts, and their names are specified as parameter values:

BicepCopy

A screen shot of a computer

Description automatically generated

In this example, the loop iterates through each item in the storageAccountNames array. Each time Bicep goes through the loop, it puts the current value into a special variable called storageAccountName, and it's used as the value of the name property. Notice that Bicep requires you put an opening bracket ([) character before the for keyword, and a closing bracket (]) character after the resource definition.

If you deployed this Bicep file, you'd see that three storage accounts were created, with their names specified by the corresponding items in the storageAccountNames array.

**Loop based on a count**

You might sometimes need to loop to create a specific number of resources, and not use an array as the source. Bicep provides the range() function, which creates an array of numbers. For example, if you need to create four storage accounts called sa1 through sa4, you could use a resource definition like this:

BicepCopy

A close-up of a computer screen

Description automatically generated

When you use the range() function, you specify its start value and the number of values you want to create. For example, if you want to create storage accounts with the names sa0, sa1, and sa2, you'd use the function range(0,3).

**Note**

When you use the range() function, you provide two arguments. The first specifies the starting value, and the second tells Bicep the number of values you want. For example, if you use range(3,4) then Bicep returns the values 3, 4, 5, and 6. Make sure you request the right number of values, especially when you use a starting value of 0.

**Access the iteration index**

With Bicep, you can iterate through arrays and retrieve the index of the current element in the array. For example, let's say you want to create a logical server in each location that's specified by an array, and you want the names of the servers to be sqlserver-1, sqlserver-2, and so on. You could achieve this by using the following Bicep code:

Bicep  
A screenshot of a computer

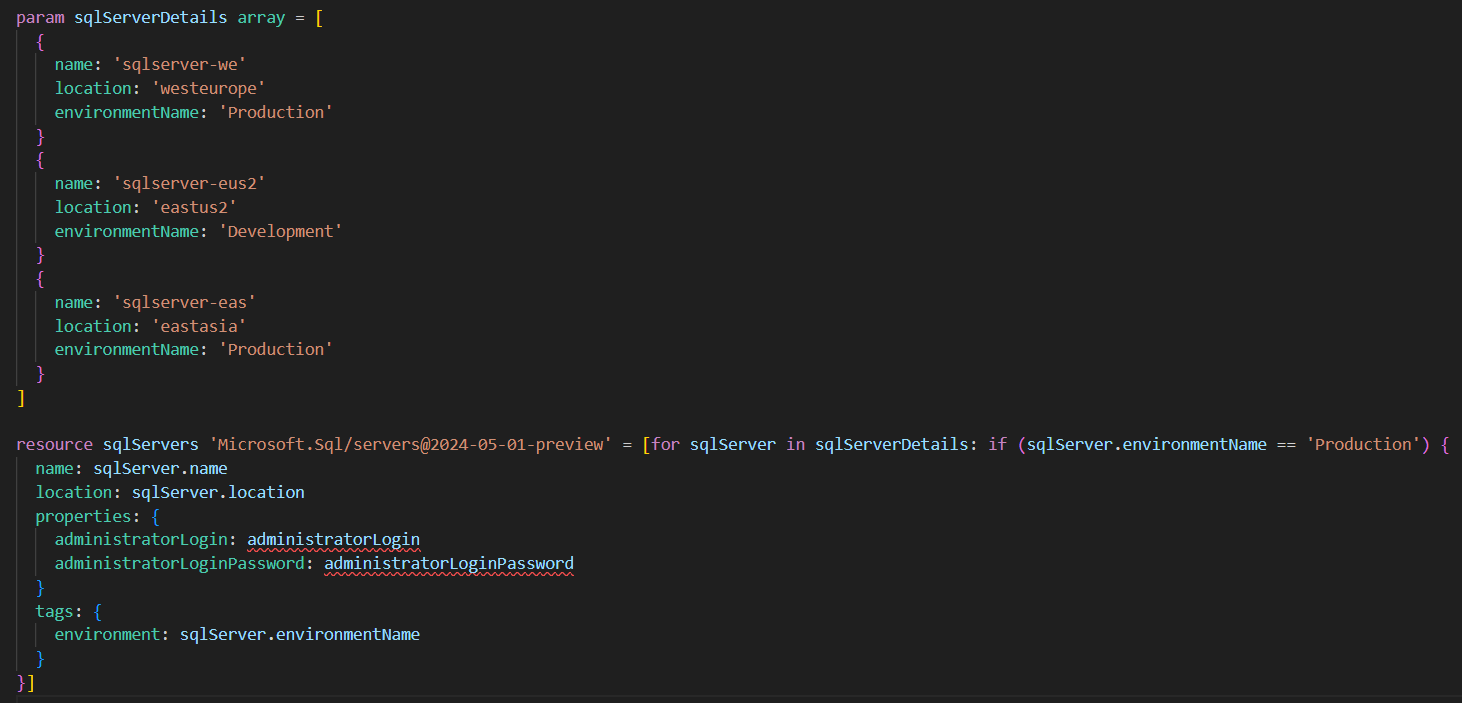
Description automatically generated

Notice that the name property includes the expression i+1. The first value of the i index variable is zero, so you need to add +1 to it if you want your server names to start with 1.

**Filter items with loops**

In some situations, you might want to deploy resources by using copy loops combined with conditions. You can do this by combining the if and for keywords.

In the following example, the code uses an array parameter to define a set of logical servers. A condition is used with the copy loop to deploy the servers only when the environmentName property of the loop object equals Production:



If you deployed the preceding example, you'd see two logical servers, sqlserver-we and sqlserver-eas, but not sqlserver-eus2, because that object's environmentName property doesn't match Production.

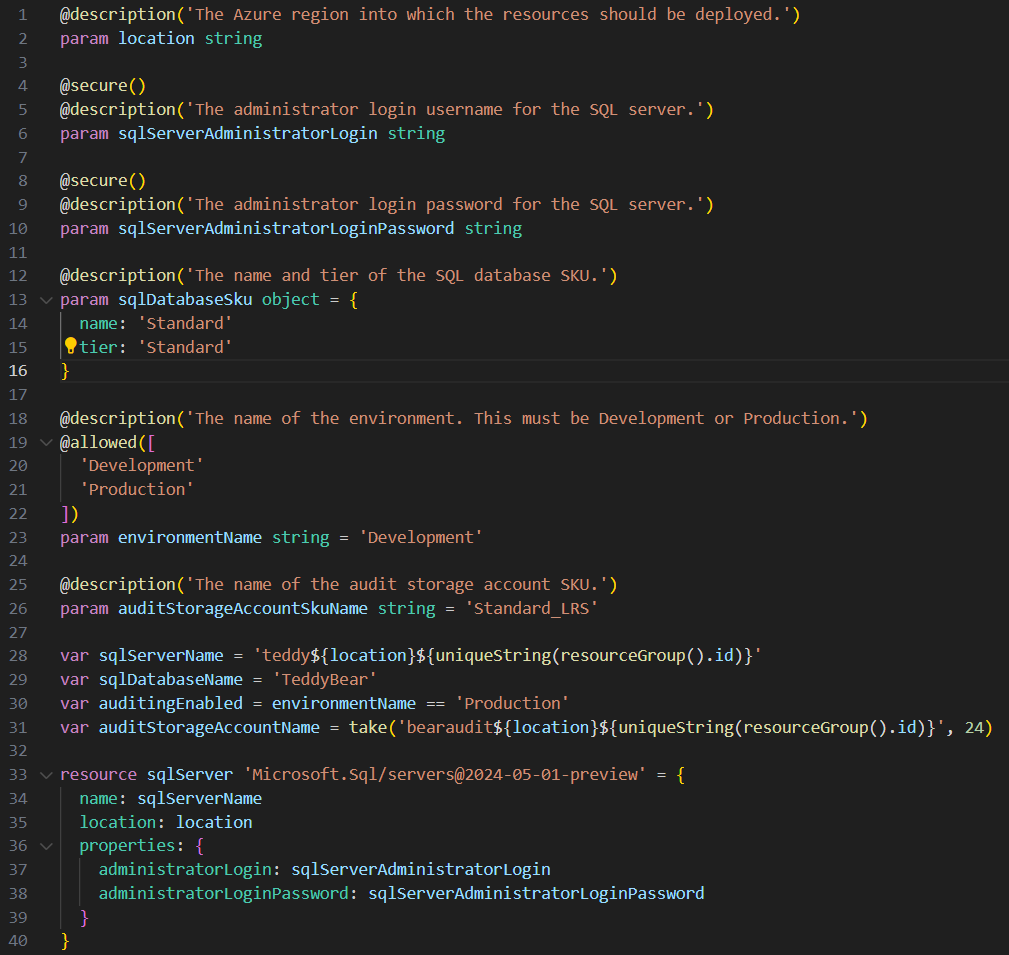
**Deploy multiple instances by using a copy loop**

Create a new *main.bicep* file

A screenshot of a computer

Description automatically generated

Your *database.bicep* file should look like this example:



A screen shot of a computer program

Description automatically generated

**Control loop execution and nest loops**

**Control loop execution**

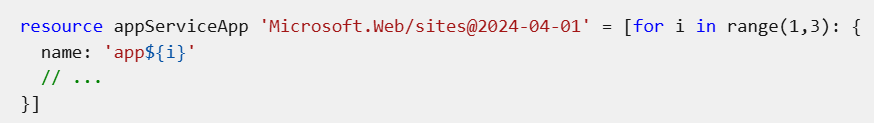
By default, Azure Resource Manager creates resources from loops in parallel and in a non-deterministic order. When you created loops in the previous exercises, both of the Azure SQL logical servers were created at the same time. This helps to reduce the overall deployment time, because all of the resources within the loop are deployed at once.

In some cases, however, you might need to deploy resources in loops sequentially instead of in parallel, or deploy small batches of changes together in parallel. For example, if you have lots of Azure App Service apps in your production environment, you might want to deploy changes to only a small number at a time to prevent the updates from restarting all of them at the same time.

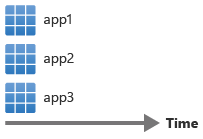
You can control the way your copy loops run in Bicep by using the @batchSize decorator. Put the decorator on the resource or module declaration with the for keyword.

Let's look at an example Bicep definition for a set of App Service applications without the @batchSize decorator:

BicepCopy



All the resources in this loop will be deployed at the same time, in parallel:

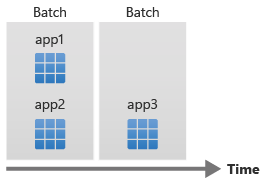


Now let's apply the @batchSize decorator with a value of 2:

A close-up of a computer screen

Description automatically generated

When you deploy the template, Bicep will deploy in batches of two:



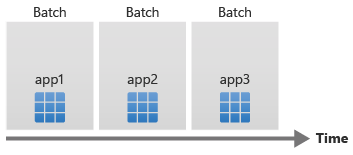
**Note**

Bicep waits for each complete batch to finish before it moves on to the next. In the preceding example, if *app2* finishes its deployment before *app1*, Bicep waits until *app1* finishes before it starts to deploy *app3*.

You can also tell Bicep to run the loop sequentially by setting the @batchSize to 1:

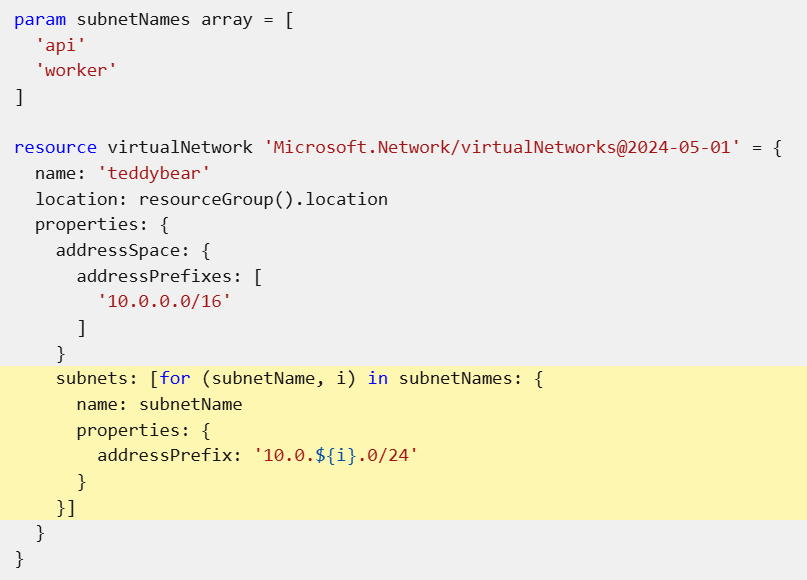
A close-up of a website

Description automatically generatedWhen you deploy the template, Bicep waits for each resource deployment to finish before it starts the next one:



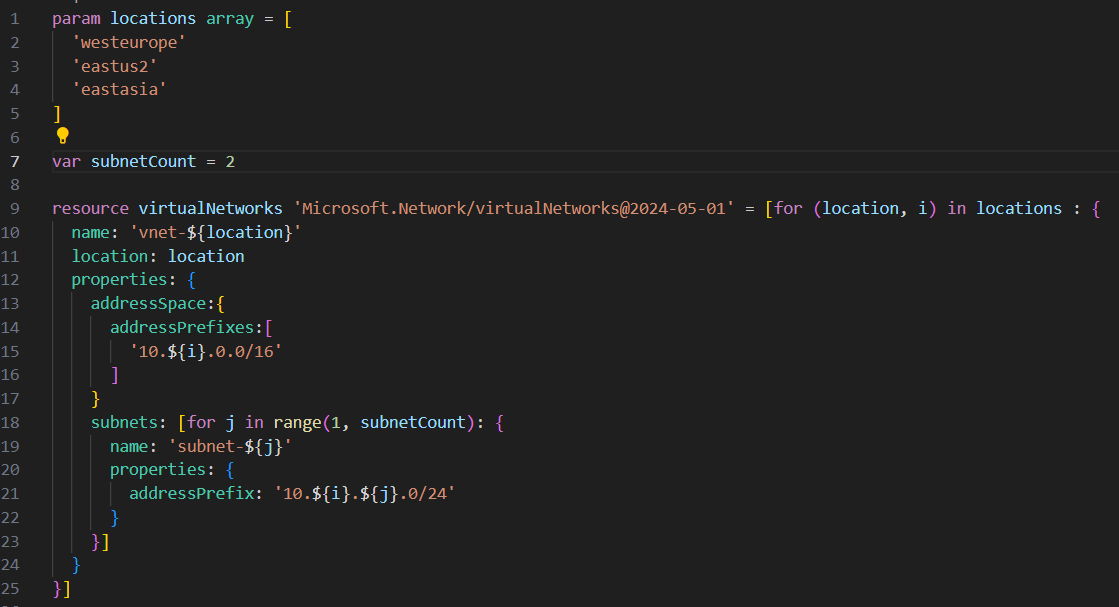
**Use loops with resource properties**

You can use loops to help set resource properties. For example, when you deploy a virtual network, you need to specify its subnets. A subnet has to have two pieces of important information: a name and an address prefix. You can use a parameter with an array of objects so that you can specify different subnets for each environment:



**Nested loops**

You can use a nested loop to deploy the subnets within each virtual network:



A screenshot of a computer

Description automatically generated

**Use variable and output loops**

**Variable loops**

A black and blue text

Description automatically generated

The preceding example creates an array that contains the values item1, item2, item3, item4, and item5.

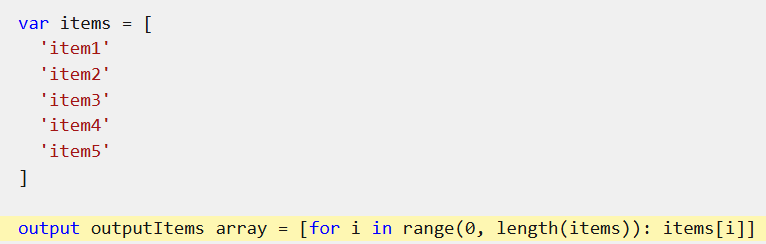
You'd ordinarily use variable loops to create more complex objects that you could then use within a resource declaration. Here's how to use variable loops to create a subnets property:



**Output loops**

You can use Bicep outputs to provide information from your deployments back to the user or tool that started the deployment. Output loops give you the flexibility and power of loops within your outputs.

As you do with other loops, use the for keyword to specify an output loop:



You'll probably need to return information about each storage account that you've created, such as its name and the endpoints that can be used to access it. By using an output loop, you can retrieve this information within your Bicep file.

A screen shot of a computer code

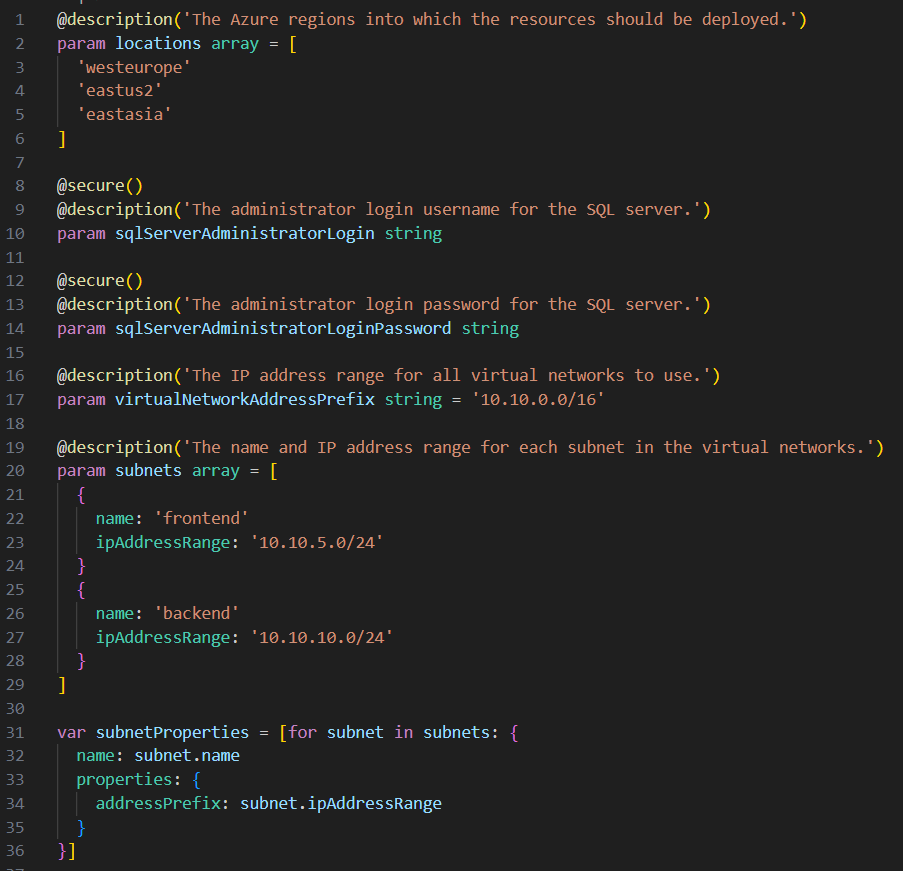
Description automatically generated

**Note**

Currently, Bicep doesn't support directly referencing resources that have been created within a loop from within an output loop. This means that you need to use array indexers to access the resources, as shown in the next example.

**Verify your Bicep file**

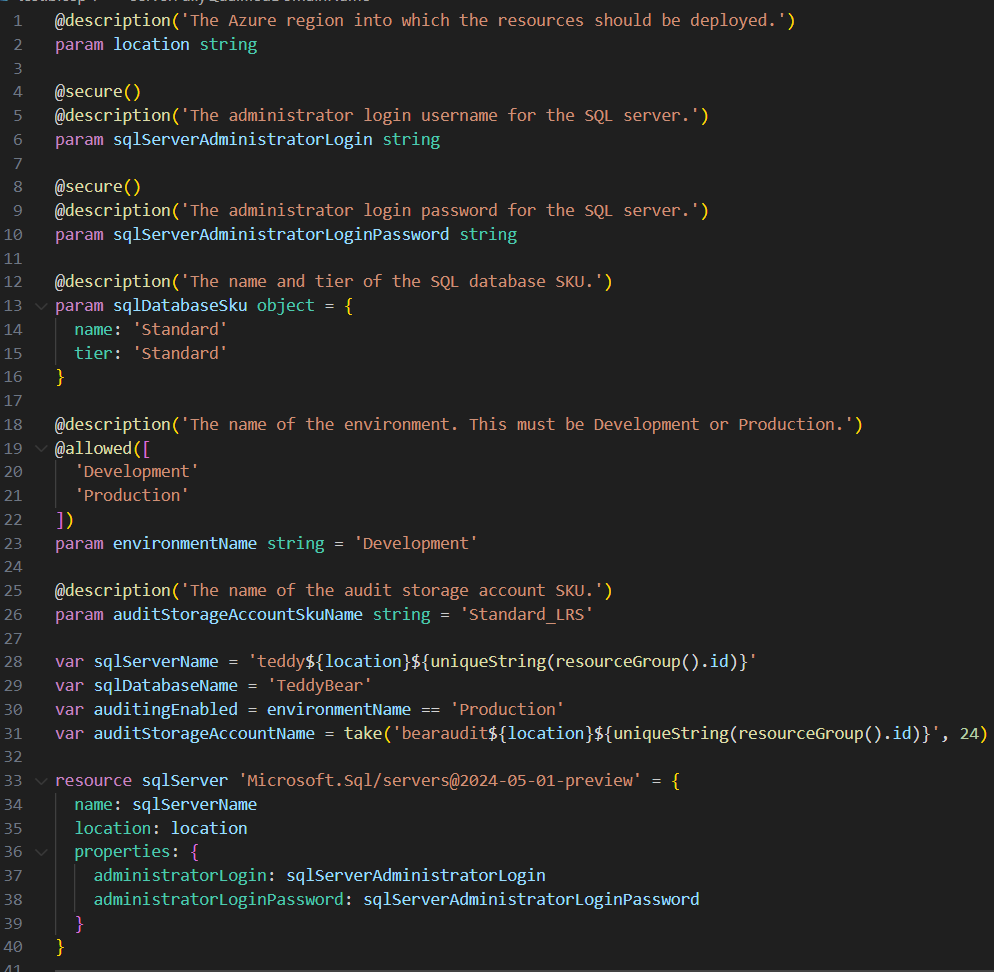
After you've completed all of the preceding changes, your *main.bicep* file should look like this example:



A screen shot of a computer program

Description automatically generated

Your *database.bicep* file should look like this example:



A computer screen with text on it

Description automatically generated

**Azure deployment changes by using what-if**

https://learn.microsoft.com/en-us/training/modules/arm-template-whatif/

**Incremental mode**

The default deployment mode is incremental. In this mode, Resource Manager doesn't delete anything. If resources exist in the resource group but aren't specified in the template, Resource Manager leaves them alone. Resources in the template are added to the resource group if they don't already exist, and if they do exist, Resource Manager updates them to the configuration in the template.

**Complete mode**

You have to explicitly ask for your deployment to run in complete mode. When you use this mode, resources that exist in Azure but that aren't specified in the template are deleted. Complete mode doesn't delete all resources in your resource group. Some resource types are exempt.

**Caution:**

When you run the command in complete mode, whatever resources you have will be removed if they're not defined in the template file.

In Bicep, you can refer to an **existing** resource by using the existing keyword. Referring to a resource in **this way doesn't stop it from being deleted during a deployment in complete mode**. You need to define it as a full-fledged resource.

**Deployment scopes:**

Complete mode is available when you deploy to a resource group. If you use templates to deploy resources to a subscription, management group, or a tenant, you aren't able to use complete mode.

**what a deployment will do by using what-if:**

Azure Resource Manager provides the what-if operation to highlight the changes when you deploy a template. The what-if operation doesn't make any changes to existing resources. Instead, it predicts the changes if the specified template is deployed at a resource group and subscription level.

Using the what-if operation compares the current state model to the desired state model. The what-if operation confirms if the changes made by your template match your expectations *without* applying those changes to real resources or to the state of those resources.

**Control the format of what-if results:**

You can control the amount of text output of the what-if operation by using one of these result formats:

* **FullResourcePayloads**. By including this parameter, you get a *verbose* output that consists of a list of resources that will change. The output also shows details about all the properties that will change in accordance with the template.
* **ResourceIdOnly**. This mode returns a list of resources that will change, but not all the details.

**New-AzResourceGroupDeployment -ResourceGroupName ToyStorage -TemplateFile $templateFileName -WhatIf -WhatIfResultFormat FullResourcePayloads**

The preceding command produces the following results:

Output

Resource and property changes are indicated with this symbol:

~ Modify

The deployment will update the following scope:

Scope: /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/ToyStorage

~ Microsoft.Storage/storageAccounts/bz64gjjpidbuvi [2019-06-01]

~ sku.name: "Standard\_LRS" => "Standard\_GRS"

Resource changes: 1 to modify.

**New-AzResourceGroupDeployment -ResourceGroupName ToyStorage -TemplateFile $templateFileName -WhatIf -WhatIfResultFormat ResourceIdOnly**

The preceding command produces the following results:

Output

Resource and property changes are indicated with this symbol:

! Deploy

The deployment will update the following scope:

Scope: /subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/ToyStorage

! Microsoft.Storage/storageAccounts/bz64gjjpidbuvi

Resource changes: 1 to deploy.

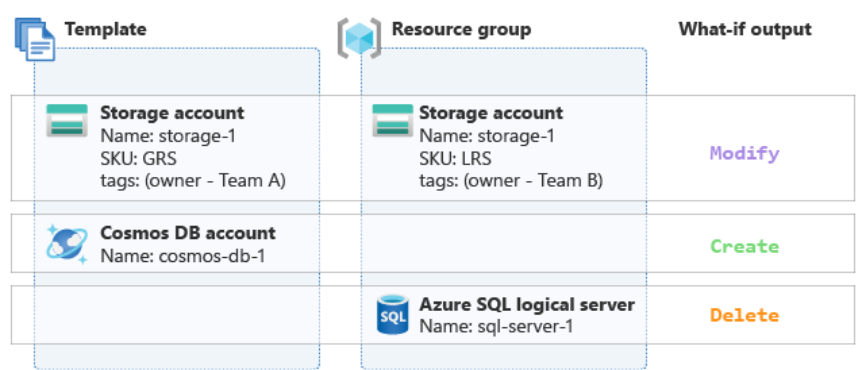
A screenshot of a computer screen

Description automatically generated

**Use what-if results in a script:**

You can get the results by using the **Get-AzResourceGroupDeploymentWhatIfResult** cmdlet. Then, your script can parse the results and perform any custom logic you might need.

**Deployment modes and deletion of resources**



Here's an explanation of what's happening in this example:

* The template is updating a storage account named storage-1, which is already deployed. The SKU is changing from *LRS* to *GRS*, and the owner tag is changing value to *Team A*. The what-if output shows a **Modify** change type for this resource.
* The template is creating a new Azure Cosmos DB account named cosmos-db-1, which doesn't exist in the resource group yet. The what-if output shows a **Create** change type for this resource.
* The template doesn't include the Azure SQL logical server that already exists in the resource group. Because the deployment uses complete mode, the what-if output shows a **Delete** change type for this resource. If the deployment used incremental mode instead of complete mode, the change type would be **Ignore** instead.

**Confirm your deployments:**

It's a good idea to run your deployment commands with the **-Confirm** switch, especially if you're deploying in complete mode. If you use the **-Confirm** switch, you have a chance to stop the operation if you don't like the proposed changes.

* **Connect-AzAccount**
* **$context = Get-AzSubscription -SubscriptionId {Your subscription ID}**
* **Set-AzContext $context**
* **Set-AzDefault -ResourceGroupName [sandbox resource group name]**
* **New-AzResourceGroupDeployment -TemplateFile main.bicep**
* **New-AzResourceGroupDeployment -WhatIf -TemplateFile main.bicep**

**A black screen with white text

Description automatically generated**

**Remove the resources in the template:**

1. In the *main.bicep* file in Visual Studio Code, delete all of the contents of the file but don't delete the file itself.
2. Save the template.

**Deploy by using complete mode and the confirmation option**

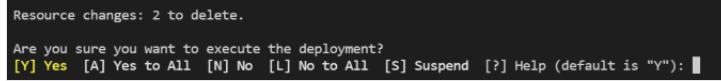
In these next steps, you'll deploy an empty template over your existing environment.

**Warning**

Doing this in real life *will remove* anything you have in the cloud. The following code is interesting as an intellectual experiment, but be careful about using this mode. At minimum, use the -Confirm flag so you can stop this operation if you don't like the proposed changes.

1. Run New-AzResourceGroupDeployment with the -Mode Complete flag to execute the deployment in complete mode:

* **New-AzResourceGroupDeployment -Mode Complete -Confirm -TemplateFile main.bicep**

****

**Create composable Bicep files by using modules**

<https://learn.microsoft.com/en-us/training/modules/create-composable-bicep-files-using-modules/1-introduction>

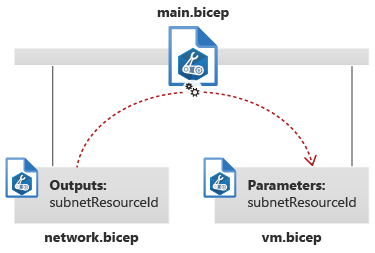
**Create and use Bicep modules**

Modules are independent Bicep files. They typically contain sets of resources that are deployed together. Modules can be consumed from any other Bicep template.

By using modules, you can reuse your Bicep code, and you can make your Bicep files more readable and understandable because they're each focused on a specific job. Your main templates then compose multiple modules together.

**Composability**

After you've created a set of modules, you can compose them together. For example, you might create a module that deploys a virtual network, and another module that deploys a virtual machine. You define parameters and outputs for each module so that you can take the important information from one and send it to the other.



**Nest modules**

Modules can include other modules. By using this nesting technique, you can create some modules that deploy small sets of resources, then compose these into larger modules that define complex topologies of resources. A template combines these pieces into a deployable artifact.

**Use the module in a Bicep template**

You'll use a module in a Bicep template by using the module keyword, like this:

A screen shot of a computer

Description automatically generated

A module definition includes the following components:

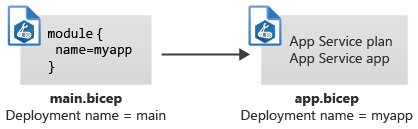
* The module keyword. A symbolic name, like appModule. This name is used within this Bicep file whenever you want to refer to the module. The symbolic name never appears in Azure.
* The module path, like modules/app.bicep. This is typically the path to a Bicep file on your local file system. In a future Microsoft Learn module, you'll learn about how you can share modules by using registries and template specs, which have their own module path formats.
* The name property, which specifies the name of the deployment. You'll learn more about deployments in the next section.
* The params property, where you can specify values for the parameters that the module expects. You'll learn more about module parameters in the next unit.

**How modules work:**

When you deploy a Bicep file by using the Azure CLI or Azure PowerShell, you can optionally specify the name of the deployment. If you don't specify a name, the Azure CLI or Azure PowerShell automatically creates a deployment name for you from the file name of the template. For example, if you deploy a file named *main.bicep*, the default deployment name is main.

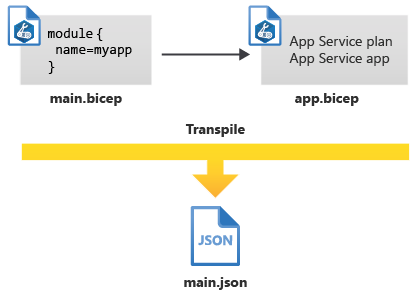
When you use modules, Bicep creates a separate deployment for every module. The name property that you specify for the module becomes the name of the deployment. When you deploy a Bicep file that contains a module, multiple deployment resources are created: one for the parent template and one for each module.

For example, suppose you create a Bicep file named *main.bicep*. It defines a module named myApp. When you deploy the *main.bicep* file, two deployments are created. The first one is named main, and it creates another deployment named myApp that contains your application resources.



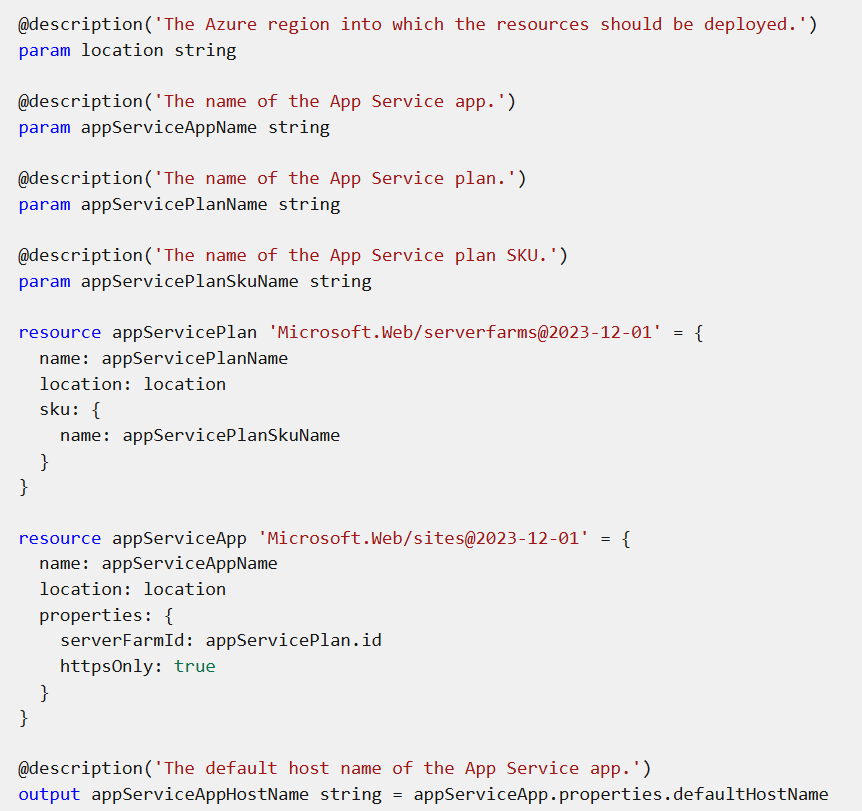
**Generated JSON ARM templates**

When you deploy a Bicep file, Bicep converts it to a JSON ARM template. This conversion is also called *transpilation*. The modules that the template uses are embedded into the JSON file. Regardless of how many modules you include in your template, only a single JSON file will be created.

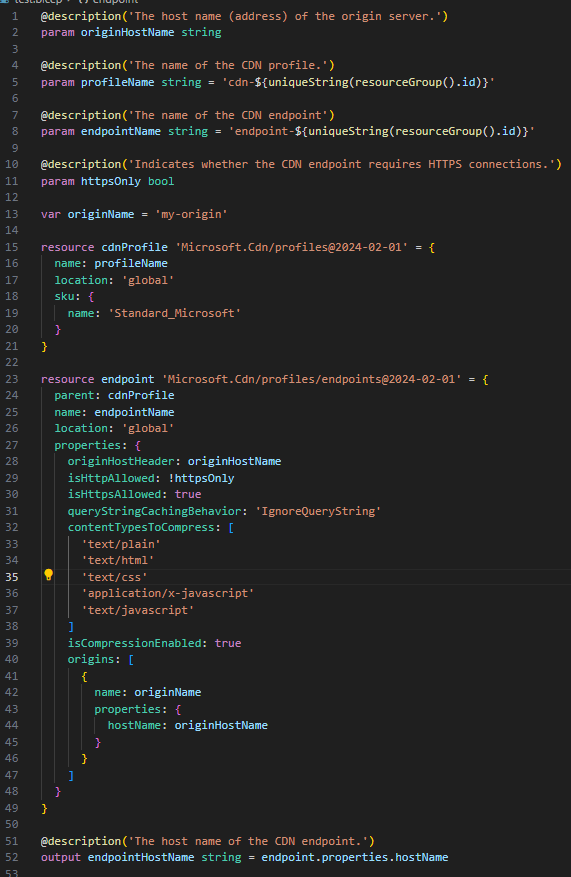


**Create a module for your application**

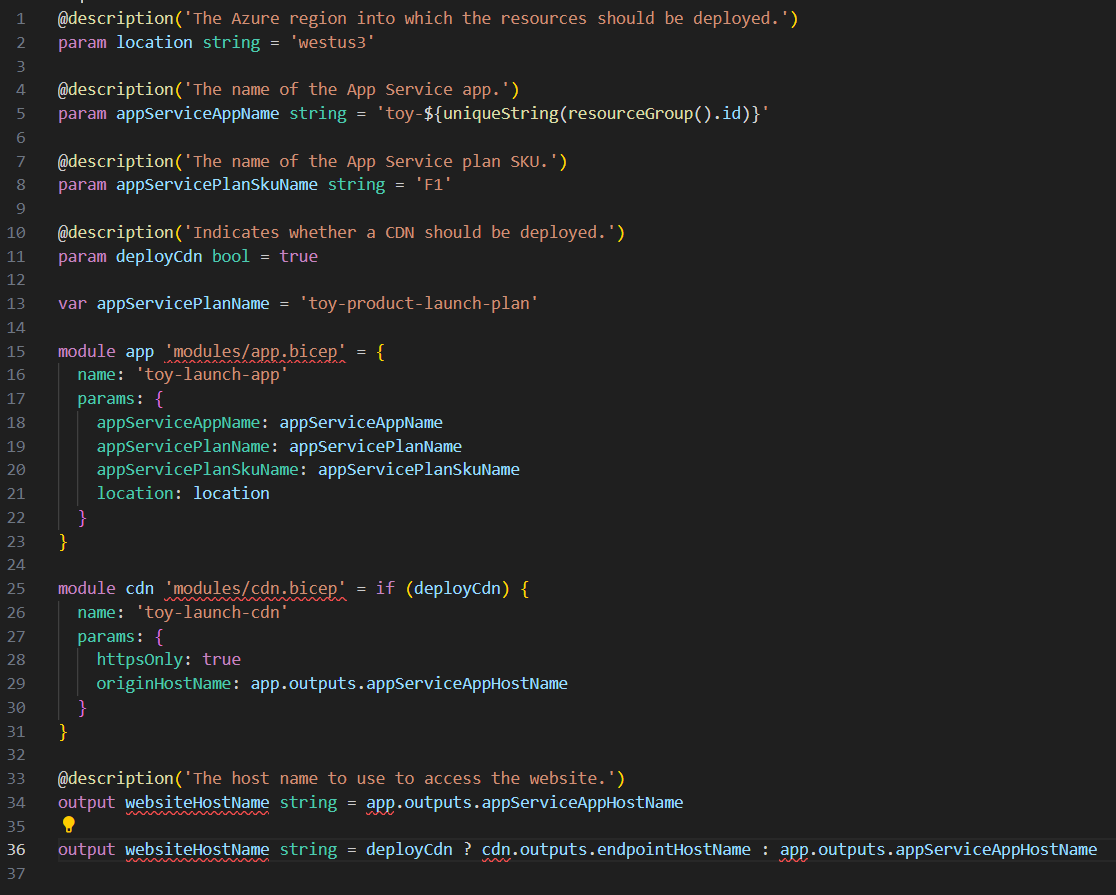
1. Create a new folder called *modules* in the same folder where you created your *main.bicep* file. In the *modules* folder, create a file called *app.bicep*. Save the file.
2. Add the following content into the *app.bicep* file:



1. In the *modules* folder, create a file called *cdn.bicep*. Save the file.
2. Add the following content into the *cdn.bicep* file:



the *main.bicep* file.



**Structure your Bicep code for collaboration**

<https://learn.microsoft.com/en-us/training/modules/structure-bicep-code-collaboration/1-introduction>

refer this document:

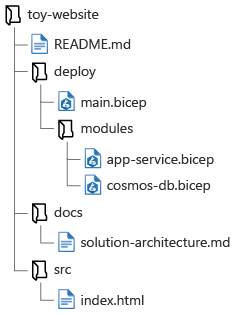
**Manage changes to your Bicep code by using Git**

<https://learn.microsoft.com/en-us/training/modules/manage-changes-bicep-code-git/1-introduction>

If your team doesn't already have a preference, here's a suggestion for how you might do it:

* At the root of your repository, create a *README.md* file. This text file, written in Markdown, describes the repository's contents and gives instructions to help team members work in the repository.
* At the root of your repository, create a *deploy* folder. Inside the folder:
  + Store your main Bicep template, named *main.bicep*.
  + Create a *modules* subfolder, to store your Bicep modules.
  + If you have other scripts or files that are used during deployments, store them in the *deploy* folder.
* At the root of your repository, create a *src* folder for source code. Use it to store application code.
* At the root of your repository, create a *docs* folder. Use it to store documentation about your solution.

Here's an illustration of how this structure might look for your toy company's website:



**Review Azure infrastructure changes by using Bicep and pull requests**

<https://learn.microsoft.com/en-us/training/modules/review-azure-infrastructure-changes-using-bicep-pull-requests/1-introduction>

**Deploy child and extension resources by using Bicep:**

https://learn.microsoft.com/en-us/training/modules/arm-template-whatif/1-introduction

**Define child resources:**

It makes sense to deploy some resources only within the context of their parent. These resources are called *child resources*. There are many child resource types in Azure. Here are a few examples:

| **Name** | **Resource type** |
| --- | --- |
| Virtual network subnets | Microsoft.Network/virtualNetworks/subnets |
| App Service configuration | Microsoft.Web/sites/config |
| SQL databases | Microsoft.Sql/servers/databases |
| Virtual machine extensions | Microsoft.Compute/virtualMachines/extensions |
| Storage blob containers | Microsoft.Storage/storageAccounts/blobServices/containers |
| Azure Cosmos DB containers | Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers |

**Nested resources:**

resource vm 'Microsoft.Compute/virtualMachines@2024-07-01' = {

name: vmName

location: location

properties: {

// ...

}

resource installCustomScriptExtension 'extensions' = {

name: 'InstallCustomScript'

location: location

properties: {

// ...

}

}

}

Notice that the nested resource has a simpler resource type than normal. Even though the fully qualified type name is Microsoft.Compute/virtualMachines/extensions, the nested resource automatically inherits the parent's resource type, so you need to specify only the child resource type, extensions.

Also notice that there's no API version specified for the nested resource. Bicep assumes that you want to use the same API version as the parent resource, although you can override the API version if you want to.

You can refer to a nested resource by using the :: operator. For example, you could create an output that returns the full resource ID of the extension:

output childResourceId string = vm::installCustomScriptExtension.id

**Parent property:**

resource vm 'Microsoft.Compute/virtualMachines@2024-07-01' = {

name: vmName

location: location

properties: {

// ...

}

}

resource installCustomScriptExtension 'Microsoft.Compute/virtualMachines/extensions@2024-07-01' = {

parent: vm

name: 'InstallCustomScript'

location: location

properties: {

// ...

}

}

To refer to a child resource declared with the parent property, use its symbolic name as you would with a normal parent resource:

output childResourceId string = installCustomScriptExtension.id

**Construct the resource name:**

There are some circumstances where you can't use nested resources or the parent keyword.

resource vm 'Microsoft.Compute/virtualMachines@2024-07-01' = {

name: vmName

location: location

properties: {

// ...

}

}

resource installCustomScriptExtension 'Microsoft.Compute/virtualMachines/extensions@2024-07-01' = {

name: '${vm.name}/InstallCustomScript'

location: location

properties: {

// ...

}

}

Notice that this example uses string interpolation to append the virtual machine resource name property to the child resource name. Bicep understands that there's a dependency between your child and parent resources. You could declare the child resource name by using the vmName variable instead. If you do that, though, your deployment could possibly fail because Bicep wouldn't understand that the parent resource needs to be deployed before the child resource:

To resolve this situation, you could manually tell Bicep about the dependency by using the dependsOn keyword, as shown here:

resource vm 'Microsoft.Compute/virtualMachines@2024-07-01' = {

name: vmName

location: location

properties: {

// ...

}

}

resource installCustomScriptExtension 'Microsoft.Compute/virtualMachines/extensions@2024-07-01' = {

name: '${vmName}/InstallCustomScript'

dependsOn: [

vm

]

//...

}

**Refer to existing resources:**

Bicep files often need to refer to resources that were created elsewhere. These resources might be created manually, maybe by a colleague using the Azure portal. Or they might be created in another Bicep file.

Within a Bicep file, you can define a resource that already exists. The declaration looks similar to a normal resource definition, but there are a few key differences. In the following example of an existing resource definition, the definition refers to a storage account named toydesigndocs. The storage account is in the same resource group that your Bicep template is deploying resources to.

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' existing = {

name: 'toydesigndocs'

}

* The existing keyword indicates to Bicep that this resource definition is a reference to an already-created resource, and that Bicep shouldn't try to deploy it.
* The name property is the Azure resource name of the storage account that was previously deployed.
* You don't need to specify the location, sku, or properties, because the template doesn't deploy the resource. It merely references an existing resource. Think of it as a placeholder resource.

**Refer to child resources**

You can refer to an existing child resource, too. Use the same kind of syntax that you used when you deployed a child resource. The following example shows how you can refer to an existing subnet, which is a child resource of a virtual network. The example uses a nested child resource, as shown here:

resource vnet 'Microsoft.Network/virtualNetworks@2024-01-01' existing = {

name: 'toy-design-vnet'

resource managementSubnet 'subnets' existing = {

name: 'management'

}

}

You can then refer to the subnet by using the same :: operator that you use for other nested child resources:

output managementSubnetResourceId string = vnet::managementSubnet.id

**Refer to resources outside the resource group:**

Often, you need to refer to resources in a different resource group. For example, if you have a virtual network in a centralized resource group, you might want to deploy a virtual machine into that virtual network in its own resource group. You can use the scope keyword to refer to existing resources in a different resource group. The following example shows how you could refer to a virtual network named toy-design-vnet within the networking-rg resource group:

resource vnet 'Microsoft.Network/virtualNetworks@2024-01-01' existing = {

scope: resourceGroup('networking-rg')

name: 'toy-design-vnet'

}

Notice that the scope uses the resourceGroup() keyword to refer to the resource group that contains the virtual network.

You can even refer to resources within a different Azure subscription, as long as the subscription is within your Microsoft Entra tenant. If your networking team provisions the virtual network in a different subscription, the template could refer to it, as in this example:

resource vnet 'Microsoft.Network/virtualNetworks@2024-01-01' existing = {

scope: resourceGroup('A123b4567c-1234-1a2b-2b1a-1234abc12345', 'networking-rg')

name: 'toy-design-vnet'

}

Notice that the scope uses the resourceGroup() keyword to refer to the Azure subscription ID (A123b4567c-1234-1a2b-2b1a-1234abc12345) and resource group name that contains the virtual network.

Now that you understand how to refer to existing resources, let's look at how you can use this capability in your templates.

**Add child and extension resources to an existing resource:**

You can add a child resource to an already-created parent resource by using a combination of the existing keyword and the parent keyword. The following example template creates an Azure SQL database within a server that already exists:

resource server 'Microsoft.Sql/servers@2023-08-01-preview' existing = {

name: serverName

}

resource database 'Microsoft.Sql/servers/databases@2023-08-01-preview' = {

parent: server

name: databaseName

location: location

sku: {

name: 'Standard'

tier: 'Standard'

}

}

If you need to deploy an extension resource to an existing resource, you can use the scope keyword. Here's a template that uses the existing keyword and the scope keyword to add a resource lock to a storage account that already exists:

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' existing = {

name: 'toydesigndocs'

}

resource lockResource 'Microsoft.Authorization/locks@2020-05-01' = {

scope: storageAccount

name: 'DontDelete'

properties: {

level: 'CanNotDelete'

notes: 'Prevents deletion of the toy design documents storage account.'

}

}

**Refer to an existing resource's properties:**

Resources often need to refer to the properties of other resources. For example, if you deploy an application, it might need to know the keys or connection information for another resource. By using the existing keyword, you get access to the properties of the resource that you're referring to.

The way that you access the information about a resource depends on the type of information you're getting. If it's a property that isn't secure, you ordinarily use only the properties of the resource. The following example template deploys an Azure Functions application, and uses the access details (*instrumentation key*) for an Application Insights instance that was already created:

resource applicationInsights 'Microsoft.Insights/components@2020-02-02' existing = {

name: applicationInsightsName

}

resource functionApp 'Microsoft.Web/sites@2023-12-01' = {

name: functionAppName

location: location

kind: 'functionapp'

properties: {

siteConfig: {

appSettings: [

// ...

{

name: 'APPINSIGHTS\_INSTRUMENTATIONKEY'

value: applicationInsights.properties.InstrumentationKey

}

]

}

}

}

In this example, because the instrumentation key isn't considered sensitive data, it's available in the properties of the resource. When you need to access secure data, such as the credentials to use to access a resource, use the listKeys() function, as shown in the following code:

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-05-01' existing = {

name: storageAccountName

}

resource functionApp 'Microsoft.Web/sites@2023-12-01' = {

name: functionAppName

location: location

kind: 'functionapp'

properties: {

siteConfig: {

appSettings: [

// ...

{

name: 'StorageAccountKey'

value: storageAccount.listKeys().keys[0].value

}

]

}

}

Notice that the listKeys function returns a keys array. The Bicep code retrieves the value property from the first item in the keys array. Each resource type has different information available from the listKeys() function. The Bicep extension for Visual Studio Code gives you hints to help you to understand the data that each resource's listKeys() function returns.

**Create resource template for Azure Cosmos DB for NoSQL**

**Understand Azure Resource Manager resources**

When creating Bicep files or Azure Resource Manager templates (ARM templates), you need to understand what resource types are available, and what values to use in your template.

Each of the resources available for Azure Cosmos DB is listed under the Microsoft.DocumentDB resource provider:

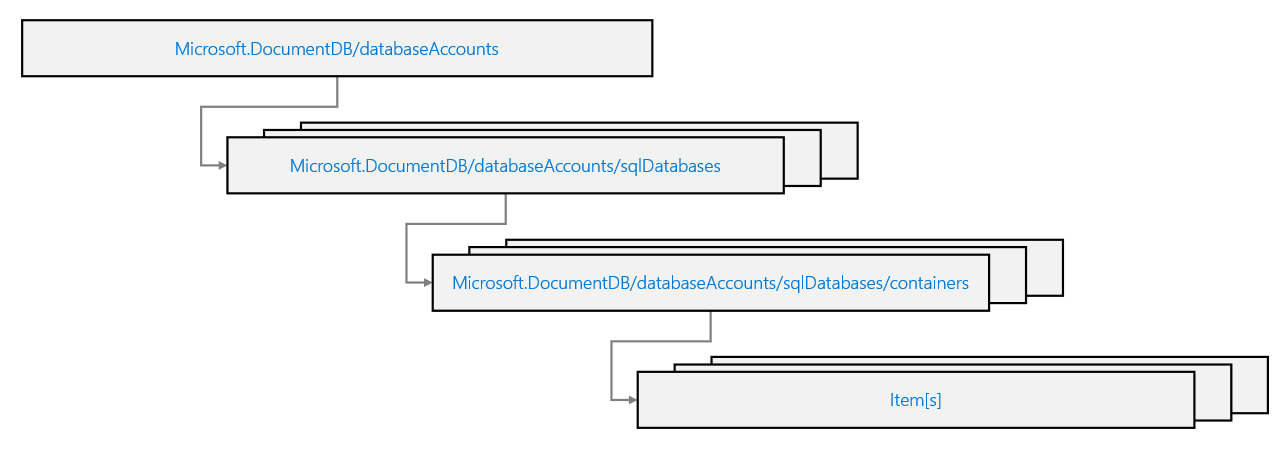
Resource type Description

Microsoft.DocumentDB/databaseAccounts Represents an account

Microsoft.DocumentDB/databaseAccounts/sqlDatabases Represents a NoSQL API database

Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers Represents a NoSQL API container

**Visually, you can think of these resources as a hierarchy.**

****

**Author Azure Resource Manager templates**

Authoring a template for an Azure Cosmos DB for NoSQL account is much like building one from scratch using the portal or from the CLI. There are three primary resources to define in a specific relationship order.

**Account resource**

The first resource type to define is **Microsoft.DocumentDB/databaseAccounts**. This represents an account that is not specific to any API. If the API is not specified, it is inferred to be a NoSQL API account.

An object for this resource must contain, at a minimum, the following properties:

* name
* location
* properties.databaseAccountOfferType
* properties.locations[].locationName

**Database resource**

The next resource is of type **Microsoft.DocumentDB/databaseAccounts/sqlDatabases** and is a child resource of the account. This relationship is defined using the **dependsOn** property.

An object for this resource must contain, at a minimum, the following properties:

* name
* properties.resources.id

**Container resource**

Within a database, you can define multiple child **Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers** resources. Here, you can allocate throughput, configure indexing policy, and set a partition key path.

A container object must contain, at a minimum, the following properties:

* name
* properties.resource.id
* properties.resource.partitionkey.paths[]

A container can also optionally contain the following properties:

* properties.options.throughput
* properties.options.autoscaleSettings.maxThroughput
* properties.resource.indexingPolicy

Here is an example of a container that is named **products**, has **1000 RU/s** autoscale, and a partition key path of **/categoryId**.

**Final template**

Now that all resources are in place, the template file should now contain the following code.

{

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

"contentVersion": "1.0.0.0",

"resources": [

{

"type": "Microsoft.DocumentDB/databaseAccounts",

"apiVersion": "2024-04-15",

"name": "[concat('csmsarm', uniqueString(resourceGroup().id))]",

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

"properties": {

"databaseAccountOfferType": "Standard",

"locations": [

{

"locationName": "westus"

}

]

}

},

{

"type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",

"apiVersion": "2024-04-15",

"name": "[concat('csmsarm', uniqueString(resourceGroup().id), '/cosmicworks')]",

"dependsOn": [

"[resourceId('Microsoft.DocumentDB/databaseAccounts', concat('csmsarm', uniqueString(resourceGroup().id)))]"

],

"properties": {

"resource": {

"id": "cosmicworks"

}

}

},

{

"type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",

"apiVersion": "2021-05-15",

"name": "[concat('csmsarm', uniqueString(resourceGroup().id), '/cosmicworks/products')]",

"dependsOn": [

"[resourceId('Microsoft.DocumentDB/databaseAccounts', concat('csmsarm', uniqueString(resourceGroup().id)))]",

"[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', concat('csmsarm', uniqueString(resourceGroup().id)), 'cosmicworks')]"

],

"properties": {

"resource": {

"id": "products",

"partitionKey": {

"paths": [

"/categoryId"

]

}

},

"options": {

"autoscaleSettings": {

"maxThroughput": 1000

}

}

}

}

]

}

**Configure database or container resources**

Each template resource uses the same resource type and version between both Azure Resource Manager and Bicep templates. If you learn how to build it in one language, you can easily learn it in the other.

**Note**

A Bicep template does not require any "empty" template syntax. You can begin writing your definitions in a blank file.

**Account resource**

The **Microsoft.DocumentDB/databaseAccounts** resource in Bicep must contain the same minimal properties as in an Azure Resource Manager template.

Here is an example of an account that has a unique name with a prefix of **csmsarm** and is deployed to **West US**.

If this resource already exists from a previous deployment, the Azure Resource Manager will just skip the resource and move on to the next. This is very handy when building a template incrementally.

**Database resource**

This example of a **Microsoft.DocumentDB/databaseAccounts/sqlDatabases** resource configures a database resource, a slight difference from the JSON template reviewed in a previous unit.

Bicep also requires resources to define a **parent** property defining relationships as opposed to the verbose **dependsOn** property.

**Container resource**

This **Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers** resource is similar to the JSON equivalent, except it defines a throughput property at this level.

**Final template**

Now that all resources are in place, the template file should now contain the following code.

resource Account 'Microsoft.DocumentDB/databaseAccounts@2024-04-15' = {

name: 'csmsbicep${uniqueString(resourceGroup().id)}'

location: resourceGroup().location

properties: {

databaseAccountOfferType: 'Standard'

locations: [

{

locationName: 'westus'

}

]

}

}

resource Database 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases@2024-04-15' = {

parent: Account

name: 'cosmicworks'

properties: {

resource: {

id: 'cosmicworks'

}

}

}

resource Container 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers@2024-04-15' = {

parent: Database

name: 'customers'

properties: {

resource: {

id: 'customers'

partitionKey: {

paths: [

'/regionId'

]

}

},

options: {

autoscaleSettings: {

maxThroughput: 1000

}

}

}

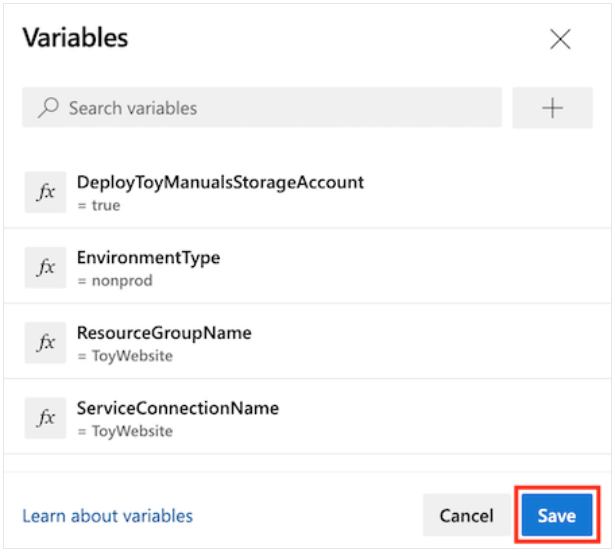
}

**Build your first Bicep deployment pipeline using Azure Pipelines**

https://learn.microsoft.com/en-us/training/modules/build-first-bicep-deployment-pipeline-using-azure-pipelines/1-introduction

**A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated**

**Trigger when multiple branches change:**

-- You can set up triggers to run your pipeline on a specific branch or on sets of branches.

trigger:

branches:

include:

- main

- release/\*

-- You can exclude specific branches, too. You want to run your pipeline automatically on all branches except for the feature branche

trigger:

branches:

include:

- '\*'

exclude:

- feature/\*

**Path filters:**

-- To set up a trigger to respond to changes in a specific folder in your repository, you can use a *path filter*:

trigger:

branches:

include:

- main

paths:

exclude:

- docs

include:

- deploy

**Schedule your pipeline to run automatically:**

A *cron expression* is a specially formatted sequence of characters that sets how often an event will happen. In this example, 0 0 \* \* \* means *run every day at midnight UTC*.

schedules:

- cron: "0 0 \* \* \*"

displayName: Daily environment restore

branches:

include:

- main

**Concurrency control:**

By default, Azure Pipelines allows multiple instances of your pipeline to run simultaneously. This can happen when you make multiple commits to a branch within a short time.

trigger:

batch: true

branches:

include:

- main

**The pipeline steps**

trigger: none

pool:

vmImage: ubuntu-latest

variables:

- name: deploymentDefaultLocation

value: westus3

jobs:

- job:

steps:

- task: AzureResourceManagerTemplateDeployment@3

inputs:

connectedServiceName: $(ServiceConnectionName)

deploymentName: $(Build.BuildNumber)

location: $(deploymentDefaultLocation)

resourceGroupName: $(ResourceGroupName)

csmFile: deploy/main.bicep

overrideParameters: >

-environmentType $(EnvironmentType)

-deployToyManualsStorageAccount $(DeployToyManualsStorageAccount)

**Test your Bicep code by using Azure Pipelines**

<https://learn.microsoft.com/en-us/training/modules/test-bicep-code-using-azure-pipelines/1-introduction>

**Bicep deployment stages:**

**A screen shot of a computer

Description automatically generated**

1. Lint: Use the Bicep linter to verify that the Bicep file is well formed and doesn't contain any obvious errors.
2. Validate: Use the Azure Resource Manager preflight validation process to check for problems that might occur when you deploy.
3. Preview: Use the what-if command to validate the list of changes that will be applied against your Azure environment. Ask a human to manually review the what-if results and approve the pipeline to proceed.
4. Deploy: Submit your deployment to Resource Manager and wait for it to finish.
5. Smoke Test: Run basic post-deployment checks against some of the important resources that you've deployed. These reviews are called *infrastructure smoke tests*.

**Build and lint Bicep code:**

A linter contains a predefined set of rules for each of these categories. Example linter rules include:

* **Unused parameters**: The linter scans for any parameters that aren't used anywhere in the Bicep file. By eliminating unused parameters, you make it easier to deploy your template because you don't have to provide unnecessary values. You also reduce confusion when someone tries to work with your Bicep file.
* **String interpolation**: The linter checks if your file uses the concat() function instead of Bicep string interpolation. String interpolation makes your Bicep files more readable.
* **Default values for secure parameters**: The linter warns you if you set default values for parameters marked with the @secure() decorator. A default value for a secure parameter is a bad practice because it gives the secure parameter a human-readable value, and people might not change it before they deploy.

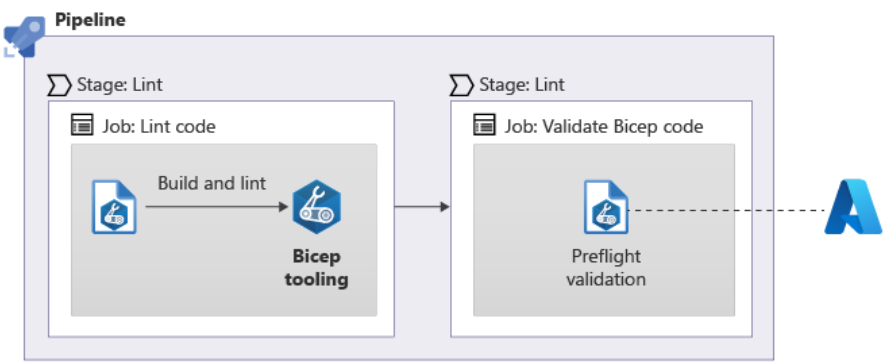
you typically want to run the validation and linting steps before you deploy the file. You can tell Bicep to verify your file by manually building the Bicep file through the Bicep CLI:

**bicep build main.bicep**

**Preflight validation:**

You also should check whether your Bicep template is likely to deploy to your Azure environment successfully. This check is called *preflight validation*, and it runs more checks that need Azure to provide information. These kinds of checks include:

* Are the names that you've specified for your Bicep resources valid?
* Are the names that you've specified for your Bicep resources already taken?
* Are the regions that you're deploying your resources to valid?



**Verify and commit your pipeline definition:**

trigger:

batch: true

branches:

include:

- main

pool:

vmImage: ubuntu-latest

variables:

- name: deploymentDefaultLocation

value: westus3

stages:

- stage: Lint

jobs:

- job: LintCode

displayName: Lint code

steps:

- script: |

az bicep build --file deploy/main.bicep

name: LintBicepCode

displayName: Run Bicep linter

- stage: Validate

jobs:

- job: ValidateBicepCode

displayName: Validate Bicep code

steps:

- task: AzureResourceManagerTemplateDeployment@3

name: RunPreflightValidation

displayName: Run preflight validation

inputs:

connectedServiceName: $(ServiceConnectionName)

location: $(deploymentDefaultLocation)

deploymentMode: Validation

resourceGroupName: $(ResourceGroupName)

csmFile: deploy/main.bicep

overrideParameters: >

-environmentType $(EnvironmentType)

- stage: Deploy

jobs:

- job: Deploy

steps:

- task: AzureResourceManagerTemplateDeployment@3

name: Deploy

displayName: Deploy to Azure

inputs:

connectedServiceName: $(ServiceConnectionName)

deploymentName: $(Build.BuildNumber)

location: $(DeploymentDefaultLocation)

resourceGroupName: $(ResourceGroupName)

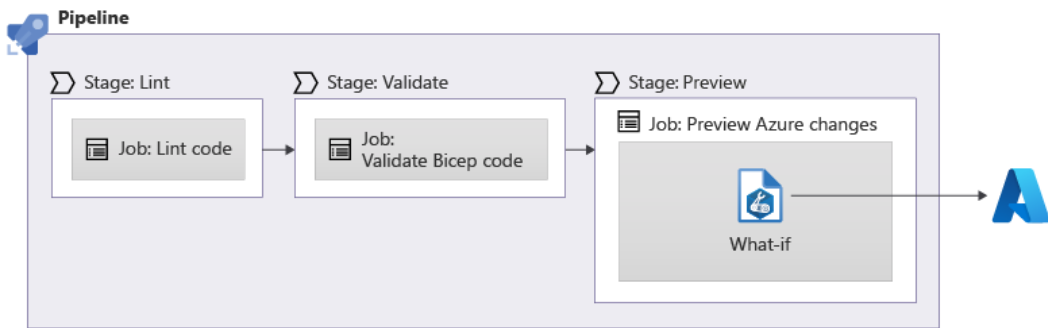
csmFile: deploy/main.bicep

overrideParameters: >

-environmentType $(EnvironmentType)

**The what-if operation:**

When resources are created, updated, or deleted, there's a risk that things can change in a way you didn't expect. It's a good practice to add an extra step to verify which resources will be created, updated, and deleted. This verification adds value to your automation process. When you're deploying to a production environment, it's important to confirm any changes that will happen to your environment.



You can use the az deployment group what-if Azure CLI command from within your pipeline definition to run the what-if step:

In this module, we'll use the Azure CLI to run the what-if operation. If you build your own PowerShell-based pipeline, you can use the New-AzResourceGroupDeployment cmdlet with the -Whatif switch, or you can use the Get-AzResourceGroupDeploymentWhatIfResult cmdlet.

stages:

- stage: Preview

jobs:

- job: Preview

steps:

- task: AzureCLI@2

inputs:

azureSubscription: 'MyServiceConnection'

scriptType: 'bash'

scriptLocation: 'inlineScript'

inlineScript: |

az deployment group what-if \

--resource-group $(ResourceGroupName) \

--template-file deploy/main.bicep

**Environments:**

In Azure Pipelines, an *environment* represents the place to which your solution is deployed. Environments provide features that help when you work with complex deployments.

you use *jobs* to define a sequence of steps within a pipeline stage. When you include environments in your pipeline, you need to use a special type of job called a *deployment job*. A deployment job is similar to a normal job, but it provides some extra functionality. This functionality includes defining the environment that the deployment job uses:

variables:

- name: deploymentDefaultLocation

value: westus3

stages:

- stage: Preview

jobs:

- job: Preview

steps:

- task: AzureCLI@2

inputs:

azureSubscription: 'MyServiceConnection'

scriptType: 'bash'

scriptLocation: 'inlineScript'

inlineScript: |

az deployment group **what-if** \

--resource-group $(ResourceGroupName) \

--template-file deploy/main.bicep

- stage: Deploy

jobs:

- deployment: Deploy

environment: MyAzureEnvironment

strategy:

runOnce:

deploy:

steps:

- checkout: self

- task: AzureResourceManagerTemplateDeployment@3

name: Deploy

displayName: Deploy to Azure

inputs:

connectedServiceName: 'MyServiceConnection'

location: $(deploymentDefaultLocation)

resourceGroupName: $(ResourceGroupName)

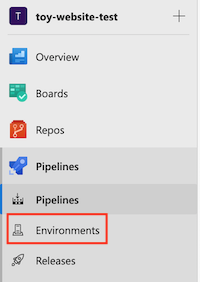
csmFile: deploy/main.bicep

Notice that in the YAML definition for a deployment job, there are some key differences from a normal job:

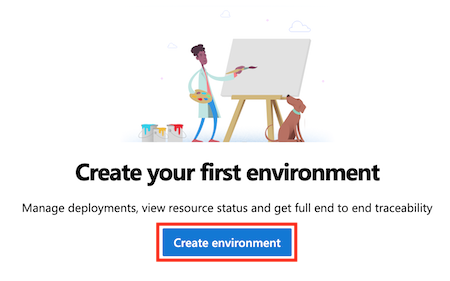
* Instead of beginning with the word job, a deployment job is defined as deployment.
* The environment keyword specifies the name of the environment to target. In the preceding example, the deployment is tracked against an environment named MyAzureEnvironment.
* The strategy keyword specifies how Azure Pipelines runs the deployment steps. Deployment strategies support complex deployment processes, especially where you have multiple production environments. In this module, we use the runOnce deployment strategy. This strategy behaves similarly to the other jobs that you're already used to.

Add an environment

1. In your browser, go to **Pipelines** > **Environments**.



1. Select **Create environment**.



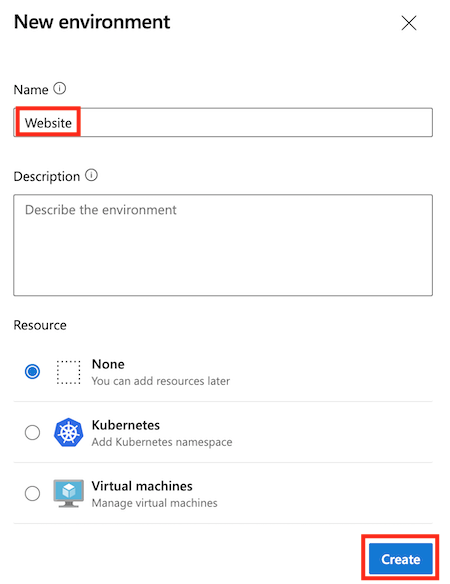
1. Enter **Website** as the environment name.

Leave the description blank. For **Resource**, select **None**.

**Note**

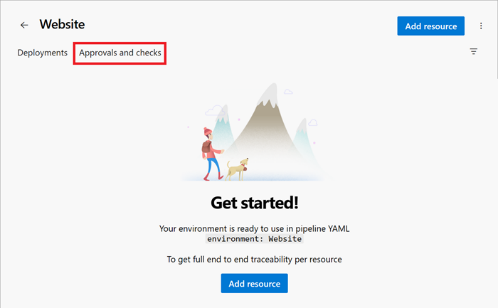
In Azure Pipelines, environments are used to enable deployment features. Some of these features apply only when you're deploying to Kubernetes or to virtual machines. In this module, we don't use these features and you can ignore them.

1. Select **Create**.

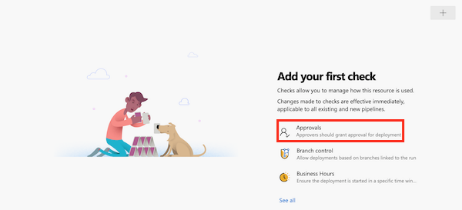


**Add an approval check to the environment**

1. Select the **Approvals and checks** tab at the top-left of the screen.



1. Select **Approvals**.



1. In the **Approvers** text box, type your own name and select yourself.
2. Select the arrow button next to **Advanced**.

Notice that, by default, approvers are allowed to approve the runs that they've triggered. Because you're the only person who will work with this pipeline, leave this checkbox selected.

1. Select **Create**.



**Verify and commit your pipeline definition:**

**trigger:**

**batch: true**

**branches:**

**include:**

**- main**

**pool:**

**vmImage: ubuntu-latest**

**variables:**

**- name: deploymentDefaultLocation**

**value: westus3**

**stages:**

**- stage: Lint**

**jobs:**

**- job: LintCode**

**displayName: Lint code**

**steps:**

**- script: |**

**az bicep build --file deploy/main.bicep**

**name: LintBicepCode**

**displayName: Run Bicep linter**

**- stage: Validate**

**jobs:**

**- job: ValidateBicepCode**

**displayName: Validate Bicep code**

**steps:**

**- task: AzureResourceManagerTemplateDeployment@3**

**name: RunPreflightValidation**

**displayName: Run preflight validation**

**inputs:**

**connectedServiceName: $(ServiceConnectionName)**

**location: $(deploymentDefaultLocation)**

**deploymentMode: Validation**

**resourceGroupName: $(ResourceGroupName)**

**csmFile: deploy/main.bicep**

**overrideParameters: >**

**-environmentType $(EnvironmentType)**

**- stage: Preview**

**jobs:**

**- job: PreviewAzureChanges**

**displayName: Preview Azure changes**

**steps:**

**- task: AzureCLI@2**

**name: RunWhatIf**

**displayName: Run what-if**

**inputs:**

**azureSubscription: $(ServiceConnectionName)**

**scriptType: 'bash'**

**scriptLocation: 'inlineScript'**

**inlineScript: |**

**az deployment group what-if \**

**--resource-group $(ResourceGroupName) \**

**--template-file deploy/main.bicep \**

**--parameters environmentType=$(EnvironmentType)**

**- stage: Deploy**

**jobs:**

**- deployment: DeployWebsite**

**displayName: Deploy website**

**environment: Website**

**strategy:**

**runOnce:**

**deploy:**

**steps:**

**- checkout: self**

**- task: AzureResourceManagerTemplateDeployment@3**

**name: DeployBicepFile**

**displayName: Deploy Bicep file**

**inputs:**

**connectedServiceName: $(ServiceConnectionName)**

**deploymentName: $(Build.BuildNumber)**

**location: $(deploymentDefaultLocation)**

**resourceGroupName: $(ResourceGroupName)**

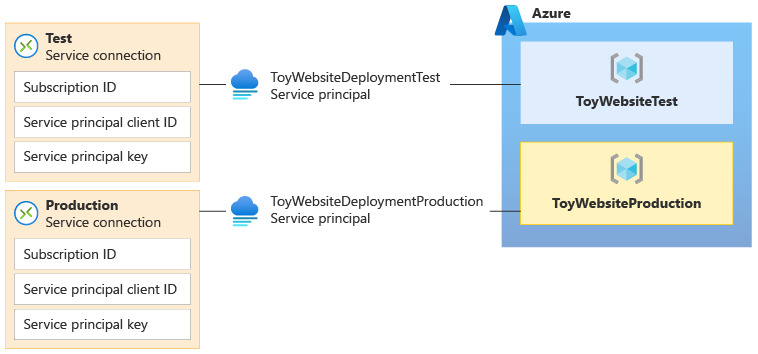
**csmFile: deploy/main.bicep**

**overrideParameters: >**

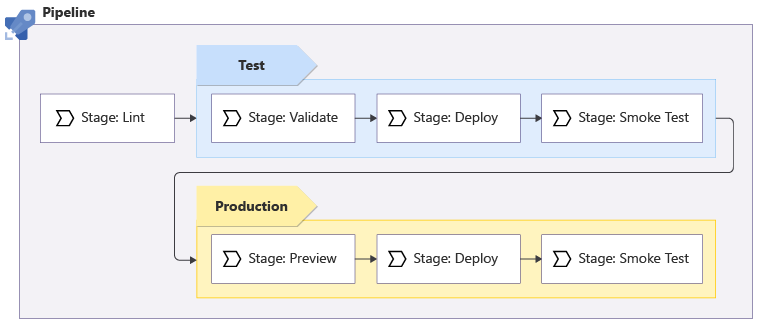
**-environmentType $(EnvironmentType)**

**Manage multiple environments by using Bicep and Azure Pipelines**

<https://learn.microsoft.com/en-us/training/modules/manage-multiple-environments-using-bicep-azure-pipelines/1-introduction>



**Deployment to multiple environments:**

****

**Verify that your *deploy.yml* file now looks like the following code:**

**YAMLCopy**

**parameters:**

**- name: environmentType**

**type: string**

**- name: deploymentDefaultLocation**

**type: string**

**default: westus3**

**stages:**

**- ${{ if ne(parameters.environmentType, 'Production') }}:**

**- stage: Validate\_${{parameters.environmentType}}**

**displayName: Validate (${{parameters.environmentType}} Environment)**

**jobs:**

**- job: ValidateBicepCode**

**displayName: Validate Bicep code**

**variables:**

**- group: ToyWebsite${{parameters.environmentType}}**

**steps:**

**- task: AzureResourceManagerTemplateDeployment@3**

**name: RunPreflightValidation**

**displayName: Run preflight validation**

**inputs:**

**connectedServiceName: ToyWebsite${{parameters.environmentType}}**

**location: ${{parameters.deploymentDefaultLocation}}**

**deploymentMode: Validation**

**resourceGroupName: $(ResourceGroupName)**

**csmFile: deploy/main.bicep**

**overrideParameters: >**

**-environmentType $(EnvironmentType)**

**-reviewApiUrl $(ReviewApiUrl)**

**-reviewApiKey $(ReviewApiKey)**

**- ${{ if eq(parameters.environmentType, 'Production') }}:**

**- stage: Preview\_${{parameters.environmentType}}**

**displayName: Preview (${{parameters.environmentType}} Environment)**

**jobs:**

**- job: PreviewAzureChanges**

**displayName: Preview Azure changes**

**variables:**

**- group: ToyWebsite${{parameters.environmentType}}**

**steps:**

**- task: AzureCLI@2**

**name: RunWhatIf**

**displayName: Run what-if**

**inputs:**

**azureSubscription: ToyWebsite${{parameters.environmentType}}**

**scriptType: 'bash'**

**scriptLocation: 'inlineScript'**

**inlineScript: |**

**az deployment group what-if \**

**--resource-group $(ResourceGroupName) \**

**--template-file deploy/main.bicep \**

**--parameters environmentType=$(EnvironmentType) \**

**reviewApiUrl=$(ReviewApiUrl) \**

**reviewApiKey=$(ReviewApiKey)**

**- stage: Deploy\_${{parameters.environmentType}}**

**displayName: Deploy (${{parameters.environmentType}} Environment)**

**jobs:**

**- deployment: DeployWebsite**

**displayName: Deploy website**

**variables:**

**- group: ToyWebsite${{parameters.environmentType}}**

**environment: ${{parameters.environmentType}}**

**strategy:**

**runOnce:**

**deploy:**

**steps:**

**- checkout: self**

**- task: AzureResourceManagerTemplateDeployment@3**

**name: DeployBicepFile**

**displayName: Deploy Bicep file**

**inputs:**

**connectedServiceName: ToyWebsite${{parameters.environmentType}}**

**deploymentName: $(Build.BuildNumber)**

**location: ${{parameters.deploymentDefaultLocation}}**

**resourceGroupName: $(ResourceGroupName)**

**csmFile: deploy/main.bicep**

**overrideParameters: >**

**-environmentType $(EnvironmentType)**

**-reviewApiUrl $(ReviewApiUrl)**

**-reviewApiKey $(ReviewApiKey)**

**deploymentOutputs: deploymentOutputs**

**- bash: |**

**echo "##vso[task.setvariable variable=appServiceAppHostName;isOutput=true]$(echo $DEPLOYMENT\_OUTPUTS | jq -r '.appServiceAppHostName.value')"**

**name: SaveDeploymentOutputs**

**displayName: Save deployment outputs into variables**

**env:**

**DEPLOYMENT\_OUTPUTS: $(deploymentOutputs)**

**- stage: SmokeTest\_${{parameters.environmentType}}**

**displayName: Smoke Test (${{parameters.environmentType}} Environment)**

**jobs:**

**- job: SmokeTest**

**displayName: Smoke test**

**variables:**

**appServiceAppHostName: $[ stageDependencies.Deploy\_${{parameters.environmentType}}.DeployWebsite.outputs['DeployWebsite.SaveDeploymentOutputs.appServiceAppHostName'] ]**

**steps:**

**- task: PowerShell@2**

**name: RunSmokeTests**

**displayName: Run smoke tests**

**inputs:**

**targetType: inline**

**script: |**

**$container = New-PesterContainer `**

**-Path 'deploy/Website.Tests.ps1' `**

**-Data @{ HostName = '$(appServiceAppHostName)' }**

**Invoke-Pester `**

**-Container $container `**

**-CI**

**- task: PublishTestResults@2**

**name: PublishTestResults**

**displayName: Publish test results**

**condition: always()**

**inputs:**

**testResultsFormat: NUnit**

**testResultsFiles: 'testResults.xml'**

**Open the *azure-pipelines.yml* file:**

**YAMLCopy**

**trigger:**

**batch: true**

**branches:**

**include:**

**- main**

**pool:**

**vmImage: ubuntu-latest**

**stages:**

**# Lint the Bicep file.**

**- stage: Lint**

**jobs:**

**- template: pipeline-templates/lint.yml**

**# Deploy to the test environment.**

**- template: pipeline-templates/deploy.yml**

**parameters:**

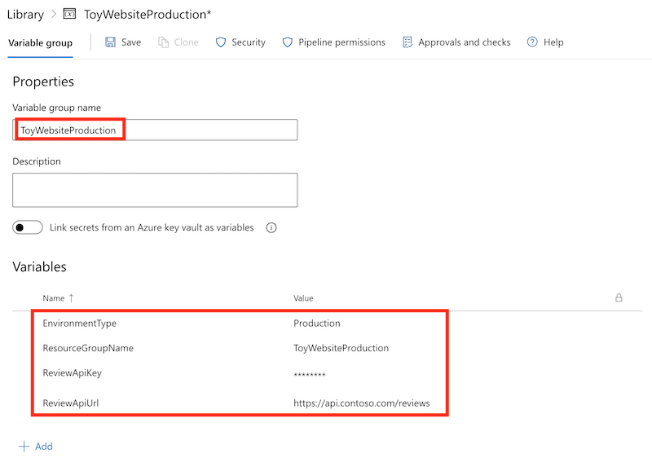
**environmentType: Test**

**# Deploy to the production environment.**

**- template: pipeline-templates/deploy.yml**

**parameters:**

**environmentType: Production**



**Clean up the resources:**

Now that you've completed the exercise, you can remove the resources so you aren't billed for them.

In the Visual Studio Code terminal, run the following commands:

Azure PowerShellCopy

**Remove-AzResourceGroup -Name ToyWebsiteTest -Force**

**Remove-AzResourceGroup -Name ToyWebsiteProduction -Force**

**Migrate Azure resources and JSON ARM templates to use Bicep**

<https://learn.microsoft.com/en-us/training/modules/migrate-azure-resources-bicep/1-introduction>

To accomplish this task, you'll convert and migrate your resources to Bicep. You'll refactor the templates for clarity and test deployments to verify the migration.

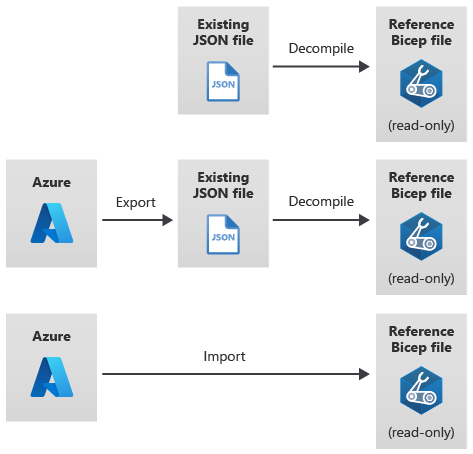
Diagram that shows the recommended workflow for migrating Azure resources to Bicep.

**Convert and migrate your resources to a Bicep file:**

**Convert phase:**

The convert phase consists of two possible steps, which you complete in sequence:

1. Capture a representation of your Azure resources.
2. If necessary, convert the JSON representation to Bicep by using the decompile command.

****

There are two types of operations in Azure: control plane operations and data plane operations. *Control plane* operations are used to manage the resources in your subscription. *Data plane* operations are used to access features that are exposed by a resource.

**Export existing resources to a JSON template:**

The export process is a control plane operation, which means that it exports only the configuration of the Azure resources. For example, when you export a virtual machine, the data on the virtual machine's hard drive isn't exported. And when you export a storage account, the blobs and other contents of the storage account aren't included in the export process.

You need to consider a few things when you export existing resources:

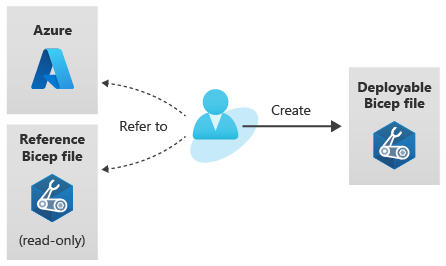
* The exported resource definition is a snapshot of that resource's current state. It includes all changes made to the resource since its initial deployment.
* The exported template might include some default resource properties that are normally omitted from a Bicep definition. For example, the export process might add read-only properties that Azure sets automatically. It doesn't make sense to include these properties because they're read-only. Consider removing these properties from the resource definitions when you migrate to Bicep to keep your Bicep files free of unnecessary code that might cause confusion.
* The exported template probably won't include all the parameters you'll need to make the template reusable. When you export a template, many of the properties will be hard-coded into the template. You'll see how to add parameters later in this module.
* Some resources can't be exported by using this approach, and you need to define them manually in your Bicep file. You'll learn how to re-create these resources later in this unit.

**Migrate phase**

The goal of the *migrate* phase of migrating your resources to Bicep is to create the first draft of your deployable Bicep file and to ensure that it defines all the Azure resources that are in scope for the migration.

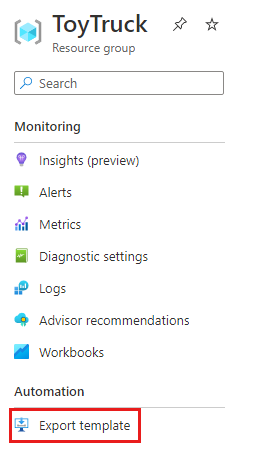
The migrate phase consists of three steps, which you complete in sequence:

1. Create a new empty Bicep file.
2. Copy each resource from your decompiled template.
3. Identify and re-create any missing resources.



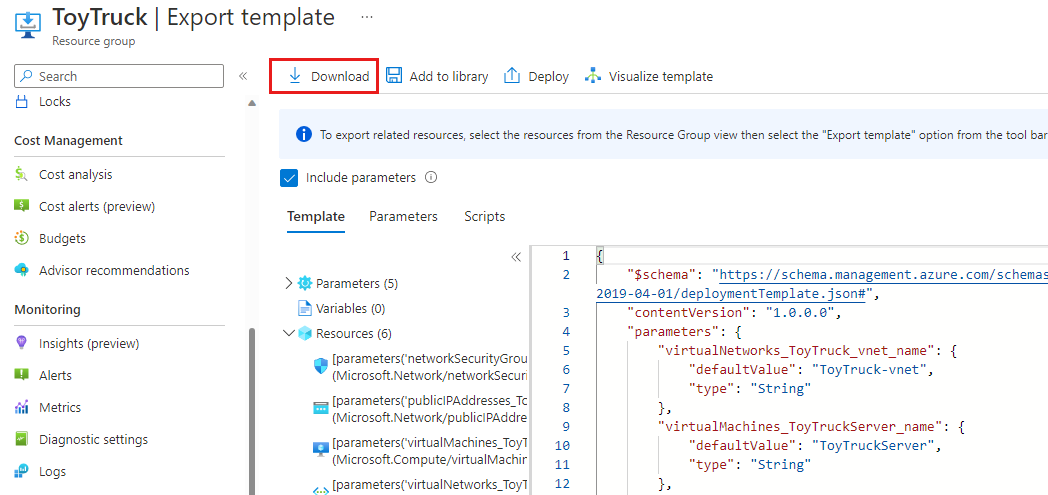
**Export the resource group contents to a JSON template**

1. In the left menu, under **Automation**, select **Export template**:



A JSON template is generated. It might take a minute or two for the process to finish.

1. Select the **Download** button:



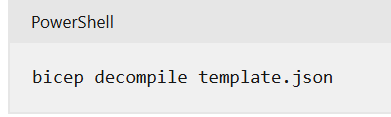
The JSON template and parameters file are downloaded to your computer as a *.zip* file.

1. Unzip the file to a folder that you can access easily, like your desktop.
2. Open Visual Studio Code.

**Decompile the JSON template to Bicep**

Use the decompile command to create a Bicep file from the template:

bicep decompile template.json

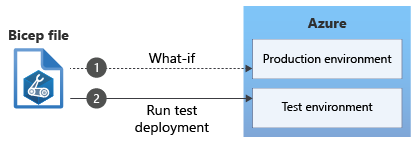


**Test phase**

The goals of the test phase of migrating your resources to Bicep are to verify the integrity of your migrated templates and to do a test deployment.

The test phase consists of two steps that you complete in this order:

1. Run the ARM template deployment what-if operation.
2. Do a test deployment.



The what-if operation provides a preview of the changes that will be made when you deploy your Bicep file. You use a test deployment to compare your original resources with the newly deployed resources.

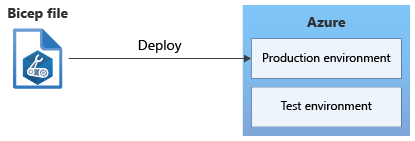
**Deploy phase**

The goal of the deploy phase of migrating your resources to Bicep is to deploy your final Bicep file to production. Before the production deployment, you should consider a few things.

The deploy phase consists of four steps, which you complete in this order:

1. Prepare a rollback plan.
2. Run the what-if operation against production.
3. Deploy the Bicep file manually.
4. Run smoke tests.

These steps help you prepare for any possible problems with production deployments.



**Prepare a rollback plan**

The ability to recover from a failed deployment is crucial. Spend time developing a rollback plan to use if any breaking changes are introduced into your environments. Your plan should take into account your organization's business continuity and disaster recovery (BCDR) strategy. Take inventory of the types of resources that are deployed, like virtual machines, web apps, and databases. You should also consider each resource's data plane. Do you have a way to recover a virtual machine and its data? Do you have a way to recover a database after it's deleted or to recover data from a storage account? A well-developed rollback plan helps keep your downtime to a minimum if any problems arise from a deployment.

**Run what-if**

1. In the Visual Studio Code terminal, run the following command:

Azure PowerShellCopy

New-AzResourceGroupDeployment `

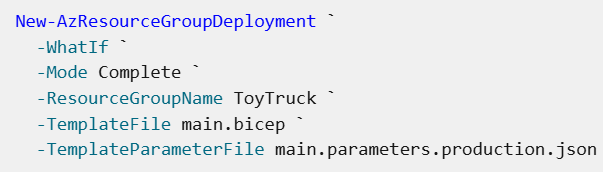
-WhatIf `

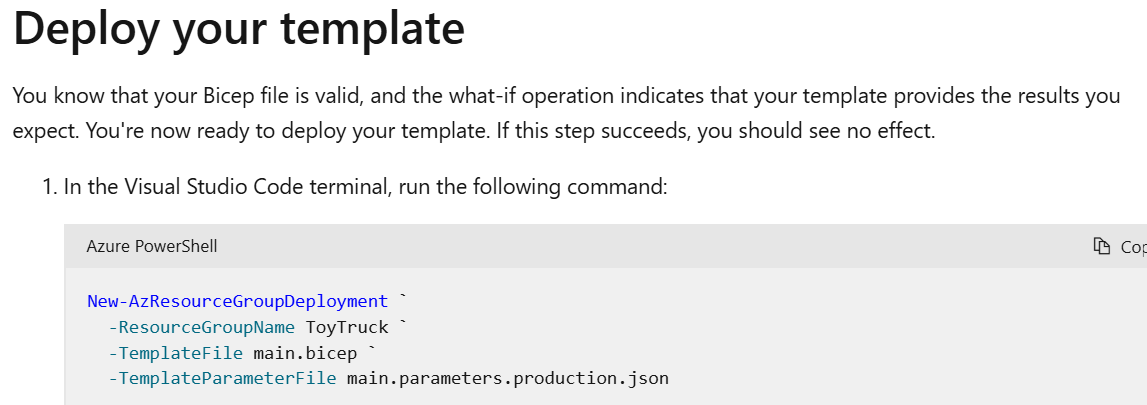
-Mode Complete `

-ResourceGroupName ToyTruck `

-TemplateFile main.bicep `

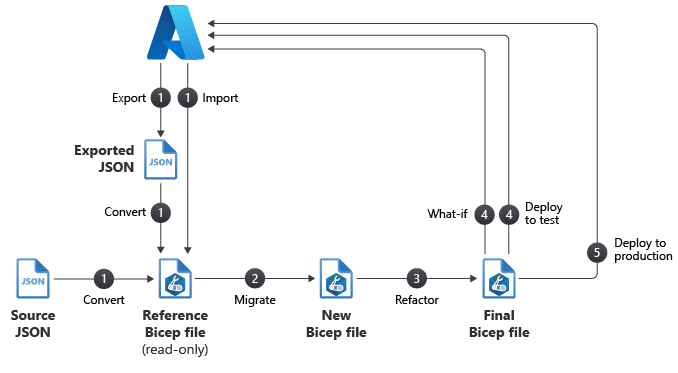
-TemplateParameterFile main.parameters.production.json





**Workflow to migrate your resources to Bicep**

In this module, you learned about the five phases of the recommended workflow for migrating your Azure resources to Bicep. The five phases are convert, migrate, refactor, test, and deploy. This unit provides a summary of the process.



**Phase 1: Convert**

The goal of the convert phase of migrating your resources is to capture an initial representation of your Azure resources. The Bicep file you create in this phase isn't complete, and it's not ready to be used. However, the file gives you a starting point for your migration.

The convert phase consists of two steps:

1. Capture a representation of your Azure resources.
2. Convert the JSON representation to Bicep by using the decompile command.

**Phase 2: Migrate**

The goal of the migrate phase is to create the first draft of your deployable Bicep file and to ensure that it defines all Azure resources that are in scope for the migration.

The migrate phase consists of three steps:

1. Create a new empty Bicep file.
2. Copy each resource from your decompiled template.
3. Identify and re-create any missing resources.

**Phase 3: Refactor**

The main focus of the refactor phase is to improve the quality of your Bicep code. These improvements can include changes, like adding code comments, that align the template with your template standards.

The refactor phase consists of eight steps:

1. Review resource API versions.
2. Review the linter suggestions in your new Bicep file.
3. Revise parameters, variables, and symbolic names.
4. Simplify expressions.
5. Review child and extension resources.
6. Modularize.
7. Add comments and descriptions.
8. Follow Bicep best practices.

**Phase 4: Test**

The goal of the test phase of migrating your resources to Bicep is to verify the integrity of your migrated templates and to do a test deployment.

The test phase consists of two steps:

1. Run the ARM template deployment what-if operation.
2. Do a test deployment.

**Phase 5: Deploy**

The goal of the deploy phase of migrating your resources to Bicep is to deploy your final Bicep file to production. Before the production deployment, you need to consider a few things.

The deploy phase consists of four steps:

1. Prepare a rollback plan.
2. Run the what-if operation against production.
3. Deploy your template manually.
4. Run smoke tests.

**Share Bicep modules by using private registries**

<https://learn.microsoft.com/en-us/training/modules/share-bicep-modules-using-private-registries/1-introduction>

A Bicep *registry* is the place where modules are stored and shared. Anyone can create their own registry. In the future, Microsoft plans to support publishing more types of Bicep content to registries, in addition to modules.

**How do registries compare to template specs?**

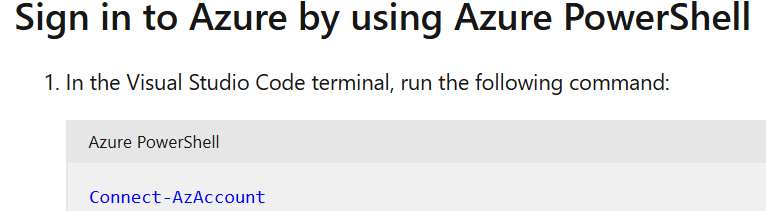
You can save an Azure Resource Manager template (ARM template) as a *template spec*.

* Bicep modules are intended to be combined into a larger deployment. Template specs are designed to be deployable as a complete template. You can deploy template specs by using the Azure portal, along with tooling like the Azure CLI and Azure PowerShell. But if you've created a template spec, Bicep also enables you to use it as a module if you want.
* Template specs are stored in Azure as resources. Modules in registries are stored as *container artifacts*.
* Template specs provide access control capabilities. When you work with a private registry, you need to control the access to your modules in other ways. You'll learn more about access control in a later unit.

**Private registries**

Bicep registries are built on a technology called *container registries*.

Currently, Bicep supports Azure Container Registry. In the future, Microsoft plans to support other registries, such as Docker Hub.



A screenshot of a computer

Description automatically generated

**Create a container registry**

A close-up of a computer screen

Description automatically generated

**List the modules in your registry**

A close-up of a sign

Description automatically generated

**Module paths**

When you've worked with modules in the past, you've probably used the module's file path to refer to it in your templates. When you work with modules and private registries, you need to use a different module path so that Bicep knows how to locate the module in your registry.

Here's an example path for a module in a private Azure container registry:



The path contains four segments:

* **Scheme**: Bicep supports several module types, which are called *schemes*. When you work with Bicep registries, the scheme is br.
* **Registry**: The name of the registry that contains the module you want to use. In the preceding example, the registry name is toycompany.azurecr.io, which is the name of the container registry.
* **Module identifier**: The full path to the module within the registry.
* **Tag**: Tags typically represent versions of modules, because a single module can have multiple versions published. You learn more about tags and versions in the next section.

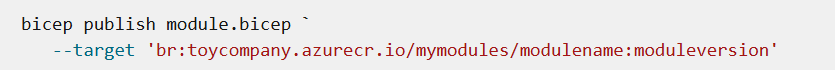
**Versioning schemes**

Your versioning scheme determines how you generate version numbers. Common versioning schemes include:

* *Basic integers* can be used as version numbers. For example, your first version might be called 1, your second version 2, and so forth. Or, you might add a prefix to each version number, such as v1 and v2.
* *Dates* also make good version numbers. For example, if you publish the first version of your module on January 16, 2022, you might name the version 2022-01-16 (using *yyyy-mm-dd* format). When you publish another version on March 3, you could name it 2022-03-03.
* *Semantic versioning* is a versioning system often used in software, where a single version number contains multiple parts. Each part signals different information about the nature of the change.

**Publish your module**

When you create a Bicep module that you want to share, you author the Bicep file as normal. You then *publish* the file to a registry by using the bicep publish command. When you publish, you need to specify the module path to save the module to:



The publish operation performs the same validation steps that happen when you build or deploy a Bicep file. These steps include:

* Checking that your code doesn't have any syntactical errors.
* Verifying that you're specifying valid resource definitions.
* Running the Bicep linter to verify that your code passes a series of quality checks.

**Create a module for a website**

You previously created a module that deploys a website. Here, you save the module file so you can publish it.

1. Open Visual Studio Code.
2. Create a new file named *website.bicep*.
3. Paste the following code into the *website.bicep* file:

**Create a module for a CDN**

Similar to the previous steps, you save a precreated module file so that you can publish it soon.

1. Create a new file named *cdn.bicep*.
2. Paste the following code into the *cdn.bicep* file:

**Publish the modules to your registry**

1. In the Visual Studio Code terminal, run the following commands. Replace YOUR\_CONTAINER\_REGISTRY\_NAME with the name of your private registry.

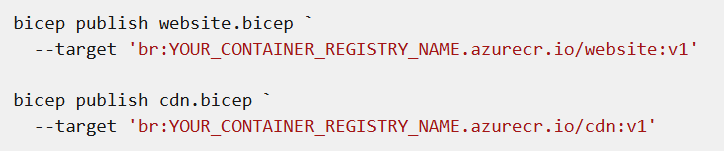
BashCopy

bicep publish website.bicep `

--target 'br:YOUR\_CONTAINER\_REGISTRY\_NAME.azurecr.io/website:v1'

bicep publish cdn.bicep `

--target 'br:YOUR\_CONTAINER\_REGISTRY\_NAME.azurecr.io/cdn:v1'



A screenshot of a computer

Description automatically generated

$context = Get-AzSubscription -SubscriptionId a00abc48-9eec-4214-a081-39ebe163b749

Set-AzContext $context

**Create a registry alias**

You decide to create a registry alias instead of embedding the registry URL in your Bicep file. This approach makes the Bicep file easier to read.

1. In Visual Studio Code, create a new file named *bicepconfig.json*. Create it in the same folder as the *main.bicep* file.
2. Paste the following code into the *bicepconfig.json* file. Replace YOUR\_CONTAINER\_REGISTRY\_NAME with the name of your private registry.

A computer code with red text

Description automatically generated

"Save the file.

**Verify your Bicep file**

After you've completed all of the preceding changes, your *main.bicep* file should look like this example:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

**Build your first Bicep deployment workflow by using GitHub Actions**

<https://learn.microsoft.com/en-us/training/modules/build-first-bicep-deployment-pipeline-using-github-actions/2-understand-github-actions>

**What is a workflow?**

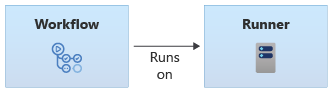
A workflow is a configurable repeatable process that's defined in a file that's used to test and deploy your code. A workflow consists of all the steps, in the proper order, that you need to execute.

the file is stored with your Bicep code in your Git repository in a folder named .github/workflows.

**Runners**

A workflow also needs access to a computer or GPU with the correct operating system and hardware platform so it can execute the deployment actions. GitHub Actions uses *runners*, which are computers that are configured to run deployment steps for a workflow.

GitHub Actions provides multiple types of runners for different operating systems, such as Linux or Windows, and different sets of tools. GitHub manages these runners, so you don't have to maintain any compute infrastructure for the runners. The runners sometimes are called *GitHub-hosted runners* or *hosted runners* because they're hosted on your behalf.

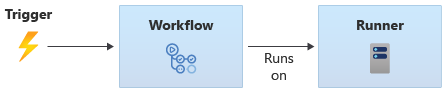


**Note**

You can create a custom runner that's called a *self-hosted runner*. You might create a self-hosted runner if you have specific software that you need to run as part of your workflow or if you need to control precisely how the runner is configured. We don't discuss self-hosted runners in this module, but we provide a link to more information in the Summary section.

**Triggers**

You use a *trigger* to instruct GitHub Actions *when* to run your workflow. You can choose from multiple types of triggers. For now, you'll use a *manual trigger* to tell GitHub Actions when to start running your workflow. Later in this module, you'll learn more about other types of triggers.



**Push event triggers**

A common type of trigger is a *push event trigger*, also called a *continuous integration trigger* or *CI trigger*.

A white background with black text

Description automatically generated

**Trigger when multiple branches change**

A white screen with black text

Description automatically generated

By using the exclude property, you ensure that the workflow isn't automatically triggered for changes to feature branches:

A computer screen shot of a computer code

Description automatically generated

**Note**

You can exclude certain branches by using the ! character. Suppose you want to trigger your workflow for your main branch and all release branches, except for the alpha releases. You can use the ! character to express this:



You can't use branches and branches-ignore together in one trigger, so the ! character provides you with flexibility to control your trigger's behavior.

**Path filters**

To set up a trigger to respond to changes in a specific folder in your repository, you can use a *path filter*:

A computer code with red text

Description automatically generated

**Note**

You can also use paths-ignore, which works in a similar manner to the branches-ignore keyword. However, you can't use paths and paths-ignore in the same trigger.

**Schedule your workflow to run automatically**

Use the schedule keyword, and set the frequency by using a cron expression:

A close-up of a white background

Description automatically generated

**Note**

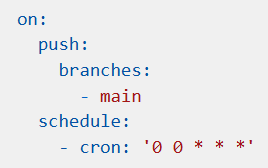
A *cron expression* is a specially formatted sequence of characters that specify how often something should happen. In this example, 0 0 \* \* \* means *run every day at midnight UTC*.

In a YAML file, you need to add quotation marks around strings that contain the \* character, such as cron expressions.

The schedule event always runs your workflow on the default branch of your repository.

**Use multiple triggers**

You can combine triggers and schedules, like in this example:



**Webhook triggers**

GitHub also provides webhook events, which run automatically when certain events happen in your repository. These events include someone creating a branch, updates to your GitHub issues, or changes to pull requests.

**Concurrency control**

By default, GitHub Actions allows multiple instances of your workflow to run simultaneously. This can happen when you make multiple commits to a branch within a short time, or if a previous run hasn't finished when your schedule next triggers.

In some situations, having multiple concurrent runs of your workflow isn't a problem. When you work with deployment workflows, however, it can be challenging to ensure that your workflow runs aren't overwriting your Azure resources or configuration in ways that you don't expect.

To avoid these problems, you can apply *concurrency control*. Use the concurrency keyword, then specify a string that's consistent across all of the runs for your workflow. It's usually a hard-coded string, like in this example:



GitHub Actions then ensures that it waits for any active workflow run to complete before starting a new run.

**Steps**

A *step* represents a single operation that the workflow performs. A step is similar to an individual command that you run in Bash or PowerShell. For most deployments, you execute several steps in a sequence. You define the sequence and all the details of each step in your workflow YAML file.

GitHub Actions offers two types of steps:

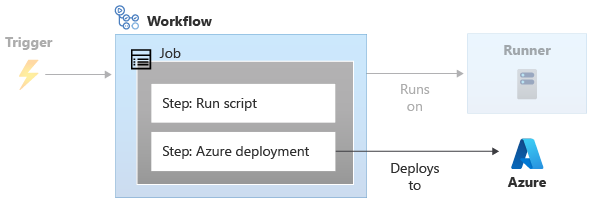
* **Run steps**: You can use a run step to run a single command or a sequence of commands in Bash, PowerShell, or the Windows command shell.
* **Action steps**: An action step is a convenient way to access many different capabilities without writing script statements. For example, there's a built-in task to deploy Bicep files to Azure. Anyone can write an action and share it with other users. A large set of commercial and open-source tasks are available.

**Jobs**

In GitHub Actions, a *job* represents an ordered set of steps. You always have at least one job in a workflow, and it's common to have more than one job when you create complex deployments.

**Note**

You can set each job to run on a different runner.



**Basic workflow example**

Now that you know the basic GitHub Actions concepts, let's look at a simple workflow definition in YAML:

A screenshot of a computer

Description automatically generated

Let's look at each part of the file in detail:

* name is the name of your workflow. The name is shown on the GitHub web interface.
* on tells your workflow when to execute. In this case, on: [workflow\_dispatch] tells GitHub Actions that you want to manually trigger the workflow.
* jobs groups together all the jobs in your workflow.
* say-hello is the name of your first and only job in this workflow.
* runs-on instructs the workflow which runner to use when it runs the job. In this example, the workflow runs on an Ubuntu operating system, which comes from a pool of GitHub-hosted runners.
* steps lists the sequence of steps to run in the job. The example YAML has two steps. Both steps run a simple script to echo some text. Each step has a name value, which is human-readable. You'll see the name in the workflow logs. To create a multi-line script step, use the pipe character (|) as shown in the example. After your step executes, you'll see the outputs in the workflow log.

**Check out your code:**

To check out your code, you can use the actions/checkout@v3 action:

A screenshot of a computer

Description automatically generated

Notice that the workflow includes the uses keyword. The keyword indicates you want to use a predefined action named actions/checkout.

Like Bicep resources, actions are always versioned. In this case, the workflow uses version 3, so @v3 is appended to the action name.

After the workflow includes this action, your repository's code will be checked out onto the runner's file system. You specify the path the files should be stored in by using the path parameter.

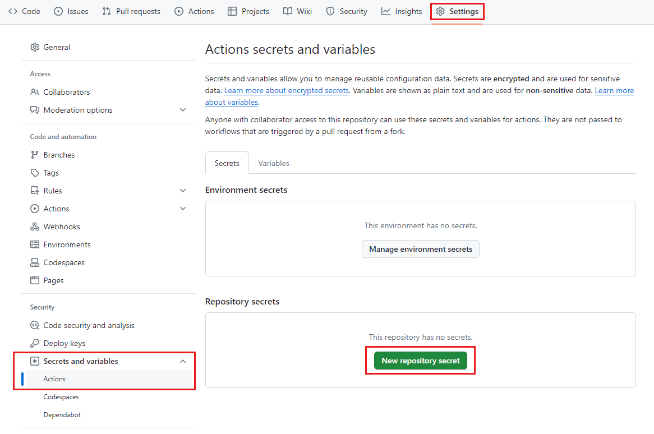
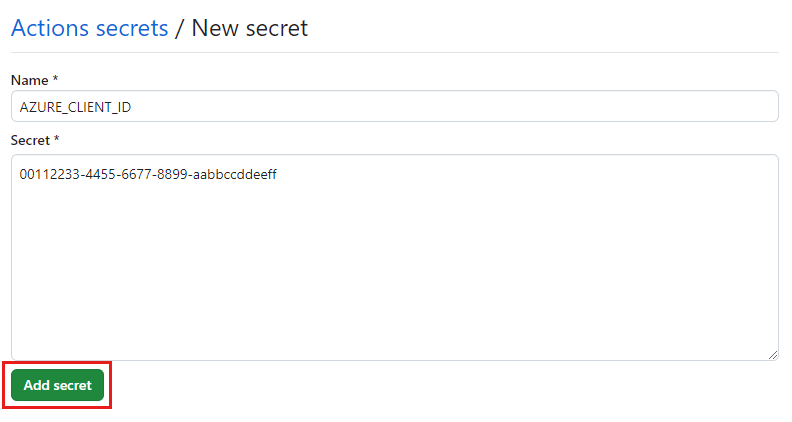
**Default environment variables**

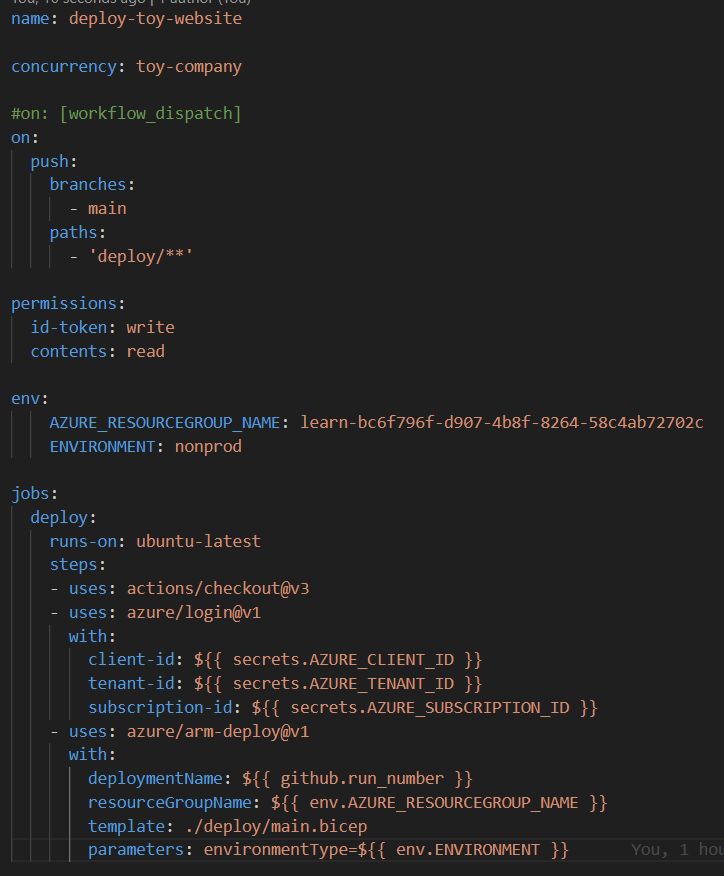
GitHub Actions also uses *default environment variables*. Default environment variables contain predefined information you might want to use in your workflow. Here are some of the default environment variables you can use in your workflow:

* github.sha: The identifier of the Git commit that triggered the workflow to execute.
* github.run\_number: A unique number for each run of a particular workflow in a repository. This number begins at 1 for the workflow's first run and increments with each new run. You might use this variable to name your Azure deployment, so you can track the deployment back to the specific workflow run that triggered it.

Create GitHub secrets

1. In your browser, navigate to your GitHub repository.
2. Select **Settings** > **Secrets and variables** > **Actions**.
3. Select **New repository secret**.



**Test your Bicep code by using GitHub Actions**

<https://learn.microsoft.com/en-us/training/modules/test-bicep-code-using-github-actions/1-introduction>

**Manage multiple environments by using Bicep and GitHub Actions**

<https://learn.microsoft.com/en-us/training/modules/manage-multiple-environments-using-bicep-github-actions/1-introduction>

**Extend Bicep and ARM templates using deployment scripts**

<https://learn.microsoft.com/en-us/training/modules/extend-resource-manager-template-deployment-scripts/1-introduction>

**What are deployment scripts?**

deploymentScripts resources are either PowerShell or Bash scripts that run in a Docker container as part of your template deployment. The default container images have either the Azure CLI or Azure PowerShell available. These scripts run during the ARM template processing, so you can add custom behavior to the deployment process.

Deployment scripts use a [managed identity](https://learn.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview) to authenticate to Azure. A managed identity is a service principal whose credential and lifecycle are managed by the Azure platform.

**How deployment scripts work**

A deploymentScripts resource takes a user-provided script (either from the template or by URI) and possibly some supporting scripts, and runs them in an [Azure container instance](https://learn.microsoft.com/en-us/azure/container-instances). That container instance is assigned the managed identity that you provide. The scripts and their output are stored in a [file share for an Azure storage account](https://learn.microsoft.com/en-us/azure/storage/files/storage-files-introduction).

When the template deployment runs, it checks whether there's an existing deploymentScripts resource in the targeted resource group. If so, it compares the properties. If everything matches, nothing new happens. If the resource doesn't exist or has been changed, Azure Resource Manager creates a new container instance and runs the deployment scripts inside that container instance. Any defined output will be passed back to Azure Resource Manager for use later in the deployment.

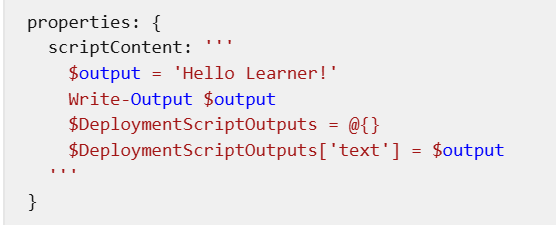
**Deployment script structure**

To add a custom behavior to an ARM template, you start with the deploymentScripts resource. At a minimum, you need to provide common details like:

* A name for the deploymentScripts resource.
* The type and apiVersion values.
* The location (location value) where the supporting resources will be created.
* An empty properties object. You'll get to that shortly.

Two deploymentScripts-specific values are required:

* kind: The type of script to run (either AzurePowerShell or AzureCLI).
* identity: The managed identity that the container instance will use. You can create the managed identity ahead of time and specify it like the following example, or you can create it in the template and reference it there (which is what you'll do in the next exercise).



Notice that the scriptContent uses a multi-line string. In Bicep, you can specify a multi-line string by using three quotes together (''') before and after your string.

**Our full template would look something like:**

****

userAssignedIdentities: {

'/subscriptions/01234567-89AB-CDEF-0123-456789ABCDEF/resourcegroups/deploymenttest/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myscriptingid': {}

}

**Include script files**

Embedding scripts inline in templates can be cumbersome, hard to read and understand, and difficult to change. Bicep uses the loadTextContent() function to embed an external text file in your deployment. When Bicep transpiles your template into JSON, it embeds the external file into the template it emits.

Let's say you have a PowerShell file named *myscript.ps1* in the same folder as your Bicep template. You can tell Bicep to embed the file like this:

A screen shot of a computer

Description automatically generated

**Verify your template**

Your template should look like:

var storageAccountName = 'storage${uniqueString(resourceGroup().id)}'

var storageBlobContainerName = 'config'

var userAssignedIdentityName = 'configDeployer'

var roleAssignmentName = guid(resourceGroup().id, 'contributor')

var contributorRoleDefinitionId = resourceId('Microsoft.Authorization/roleDefinitions', 'b24988ac-6180-42a0-ab88-20f7382dd24c')

var deploymentScriptName = 'CopyConfigScript'

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-01-01' = {

name: storageAccountName

tags: {

displayName: storageAccountName

}

location: resourceGroup().location

kind: 'StorageV2'

sku: {

name: 'Standard\_LRS'

tier: 'Standard'

}

properties: {

allowBlobPublicAccess: true

encryption: {

services: {

blob: {

enabled: true

}

}

keySource: 'Microsoft.Storage'

}

supportsHttpsTrafficOnly: true

}

resource blobService 'blobServices' existing = {

name: 'default'

}

}

resource blobContainer 'Microsoft.Storage/storageAccounts/blobServices/containers@2019-04-01' = {

parent: storageAccount::blobService

name: storageBlobContainerName

properties: {

publicAccess: 'Blob'

}

}

resource userAssignedIdentity 'Microsoft.ManagedIdentity/userAssignedIdentities@2018-11-30' = {

name: userAssignedIdentityName

location: resourceGroup().location

}

resource roleAssignment 'Microsoft.Authorization/roleAssignments@2020-04-01-preview' = {

name: roleAssignmentName

properties: {

roleDefinitionId: contributorRoleDefinitionId

principalId: userAssignedIdentity.properties.principalId

principalType: 'ServicePrincipal'

}

}

resource deploymentScript 'Microsoft.Resources/deploymentScripts@2020-10-01' = {

name: deploymentScriptName

location: resourceGroup().location

kind: 'AzurePowerShell'

identity: {

type: 'UserAssigned'

userAssignedIdentities: {

'${userAssignedIdentity.id}': {}

}

}

properties: {

azPowerShellVersion: '3.0'

scriptContent: '''

Invoke-RestMethod -Uri 'https://raw.githubusercontent.com/Azure/azure-docs-json-samples/master/mslearn-arm-deploymentscripts-sample/appsettings.json' -OutFile 'appsettings.json'

$storageAccount = Get-AzStorageAccount -ResourceGroupName 'learndeploymentscript\_exercise\_1' | Where-Object { $\_.StorageAccountName -like 'storage\*' }

$blob = Set-AzStorageBlobContent -File 'appsettings.json' -Container 'config' -Blob 'appsettings.json' -Context $storageAccount.Context

$DeploymentScriptOutputs = @{}

$DeploymentScriptOutputs['Uri'] = $blob.ICloudBlob.Uri

$DeploymentScriptOutputs['StorageUri'] = $blob.ICloudBlob.StorageUri

'''

retentionInterval: 'P1D'

}

dependsOn: [

roleAssignment

blobContainer

]

}

output fileUri string = deploymentScript.properties.outputs.Uri

**Deploy the template**

Set-Location -Path templates

Connect-AzAccount

Get-AzSubscription

Set-AzContext -SubscriptionId {Your subscription ID}

$resourceGroupName = 'learndeploymentscript\_exercise\_1'

New-AzResourceGroup -Location eastus -Name $resourceGroupName

**Deploy the template to Azure**

A screenshot of a computer code

Description automatically generated

**Using command-line arguments**

The first option for passing data into the deploymentScripts resources is to customize the arguments property. The arguments property takes a string of arguments just like the ones you'd supply at the command line. These arguments are supplied to the command property of the Azure container instance that will run the script.

A screenshot of a computer program

Description automatically generated

**Verify your template**

Your template should look similar to:

@description('List of files to copy to application storage account.')

param filesToCopy array = [

'appsettings.json'

]

var storageAccountName = 'storage${uniqueString(resourceGroup().id)}'

var storageBlobContainerName = 'config'

var userAssignedIdentityName = 'configDeployer'

var roleAssignmentName = guid(resourceGroup().id, 'contributor')

var contributorRoleDefinitionId = resourceId('Microsoft.Authorization/roleDefinitions', 'b24988ac-6180-42a0-ab88-20f7382dd24c')

var deploymentScriptName = 'CopyConfigScript'

resource storageAccount 'Microsoft.Storage/storageAccounts@2023-01-01' = {

name: storageAccountName

tags: {

displayName: storageAccountName

}

location: resourceGroup().location

kind: 'StorageV2'

sku: {

name: 'Standard\_LRS'

tier: 'Standard'

}

properties: {

allowBlobPublicAccess: true

encryption: {

services: {

blob: {

enabled: true

}

}

keySource: 'Microsoft.Storage'

}

supportsHttpsTrafficOnly: true

}

resource blobService 'blobServices' existing = {

name: 'default'

}

}

resource blobContainer 'Microsoft.Storage/storageAccounts/blobServices/containers@2019-04-01' = {

parent: storageAccount::blobService

name: storageBlobContainerName

properties: {

publicAccess: 'Blob'

}

}

resource userAssignedIdentity 'Microsoft.ManagedIdentity/userAssignedIdentities@2018-11-30' = {

name: userAssignedIdentityName

location: resourceGroup().location

}

resource roleAssignment 'Microsoft.Authorization/roleAssignments@2020-04-01-preview' = {

name: roleAssignmentName

properties: {

roleDefinitionId: contributorRoleDefinitionId

principalId: userAssignedIdentity.properties.principalId

principalType: 'ServicePrincipal'

}

}

resource deploymentScript 'Microsoft.Resources/deploymentScripts@2020-10-01' = {

name: deploymentScriptName

location: resourceGroup().location

kind: 'AzurePowerShell'

identity: {

type: 'UserAssigned'

userAssignedIdentities: {

'${userAssignedIdentity.id}': {}

}

}

properties: {

arguments: '-File \'${string(filesToCopy)}\''

environmentVariables: [

{

name: 'ResourceGroupName'

value: resourceGroup().name

}

{

name: 'StorageAccountName'

value: storageAccountName

}

{

name: 'StorageContainerName'

value: storageBlobContainerName

}

]

azPowerShellVersion: '3.0'

scriptContent: '''

param([string]$File)

$fileList = $File -replace '(\[|\])' -split ',' | ForEach-Object { $\_.trim() }

$storageAccount = Get-AzStorageAccount -ResourceGroupName $env:ResourceGroupName -Name $env:StorageAccountName -Verbose

$count = 0

$DeploymentScriptOutputs = @{}

foreach ($fileName in $fileList) {

Write-Host "Copying $fileName to $env:StorageContainerName in $env:StorageAccountName."

Invoke-RestMethod -Uri "https://raw.githubusercontent.com/Azure/azure-docs-json-samples/master/mslearn-arm-deploymentscripts-sample/$fileName" -OutFile $fileName

$blob = Set-AzStorageBlobContent -File $fileName -Container $env:StorageContainerName -Blob $fileName -Context $storageAccount.Context

$DeploymentScriptOutputs[$fileName] = @{}

$DeploymentScriptOutputs[$fileName]['Uri'] = $blob.ICloudBlob.Uri

$DeploymentScriptOutputs[$fileName]['StorageUri'] = $blob.ICloudBlob.StorageUri

$count++

}

Write-Host "Finished copying $count files."

'''

retentionInterval: 'P1D'

}

dependsOn: [

roleAssignment

blobContainer

]

}

output fileUri object = deploymentScript.properties.outputs

output storageAccountName string = storageAccountName

**Create a parameters file**

Now that you've got the template set, you can validate the new deployment script by using a parameters file with new files specified.

1. Create an *azuredeploy.parameters.json* file.
2. Edit the file to have two filesToCopy specified:



**Deploy the template to Azure**

Deploy the template by using Azure PowerShell commands in the terminal.

A screen shot of a computer program

Description automatically generated

**Deploy resources to subscriptions, management groups, and tenants by using Bicep**

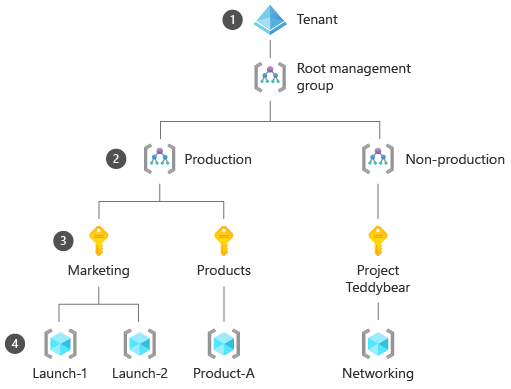
<https://learn.microsoft.com/en-us/training/modules/deploy-resources-scopes-bicep/1-introduction>

Some Azure resources are deployed outside resource groups, and are instead deployed at various *scopes*: a subscription, a management group, or even your whole tenant.

* Set the scope of a Bicep file by using the targetScope keyword.
* Use modules and the scope keyword to deploy your resources to various scopes within a single deployment.
* Target specific scopes by using the resourceGroup(), subscription(), managementGroup(), and tenant() functions.

**Understand deployment scopes:**

Azure has a hierarchical resource structure with multiple levels of management.

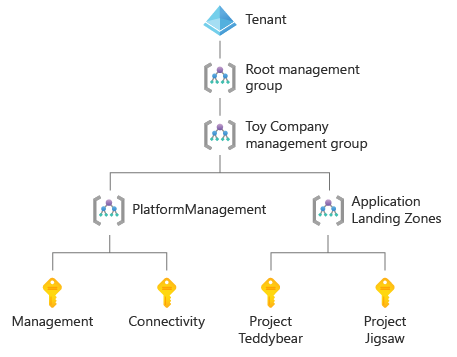


 Your *tenant* corresponds to your Microsoft Entra instance. An organization ordinarily has only one Microsoft Entra instance. This instance acts as the root of the resource hierarchy.

 *Management groups* provide a way to organize Azure subscriptions. Each tenant has a single root management group, and you can establish your own hierarchy of management groups under it. You might create separate management groups for the various parts of your organization, or for subscriptions that have their own security or governance requirements. You can apply policy and access-control restrictions to management groups, and all subscriptions below that management group in the hierarchy inherit these restrictions. Management groups aren't deployed to regions, and they have no impact on your resources' locations.

 *Subscriptions* act as billing accounts, and they contain resource groups and resources. Like management groups, subscriptions have no location and don't restrict where your resources are deployed.

 *Resource groups* are logical containers for your resources. With resource groups, you can manage and control related resources as a single unit. Resources such as virtual machines, Azure App Service plans, storage accounts, and virtual networks must be put into a resource group. Resource groups are created in a location so that Azure can track the metadata for the resources in the group, but resources inside the group can be deployed to other locations.



**Subscription-scoped resources**

You might deploy resources to a subscription when:

* You need to create a new resource group. A resource group is really only a subscription-scoped resource.
* You need to grant access to all the resources within a subscription. For example, if your HR department has an Azure subscription that contains all the department's Azure resources, you might create role assignments to allow everybody in the HR department to read the contents of the subscription.
* You're using Azure Policy, and you want to define or apply a policy to all resources within the subscription. For example, your toy company's R&D department has asked you to deploy a policy that restricts the list of virtual machine SKUs that can be created within the team's subscription.

**Management group-scoped resources**

You might deploy resources to a management group when:

* You need to grant access to all the resources within any subscriptions that fall under the management group hierarchy. For example, your cloud operations team might require access to every subscription in your organization. You can create a role assignment at your root management group, which grants your cloud operations team access to everything in Azure.
* You need to apply policies across your entire organization. For example, your organization might have a policy that resources can't be created in certain geographic regions, under any circumstances. You might apply a policy to your root management group that will block the creation of resources in that region.

**Tenant-scoped resources**

You might deploy resources to your tenant when:

* You need to create Azure subscriptions. When you use management groups, subscriptions sit under management groups in the resource hierarchy, but a subscription is deployed as a tenant-scoped resource.

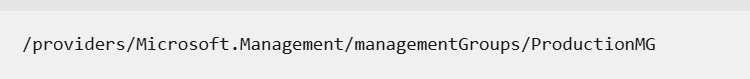
**Note**

Not all Azure customers can create subscriptions by using infrastructure as code. Depending on your billing relationship with Microsoft, this might not be possible. For more information, see [**Create Azure subscriptions programmatically**](https://learn.microsoft.com/en-us/azure/cost-management-billing/manage/programmatically-create-subscription).

* You're creating or configuring management groups. Azure creates single root management group when you enable management groups for your tenant, and you can create multiple levels of management groups under it. You can use Bicep to define your whole management group hierarchy. You can also assign subscriptions to management groups.

With Bicep, you can submit deployments to the tenant scope. [Tenant-scoped deployments require special permission](https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/deploy-to-tenant#required-access).

Here's an example resource ID for a management group:



When a resource is deployed at a management group scope, its resource ID includes the management group ID. Here's an example resource ID for a role definition that has been created at a management group scope

/providers/Microsoft.Management/managementGroups/ProductionMG/providers/Microsoft.Authorization/roleDefinitions/00000000-0000-0000-0000-000000000000

Another role definition might be defined at a subscription scope, so its resource ID looks a little different:

Copy

/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/providers/Microsoft.Authorization/roleDefinitions/00000000-00

**Specify the target scope for a Bicep file**

Use the targetScope keyword to tell Bicep that the resources in the file are for a specific scope.

A close-up of a computer screen

Description automatically generated

You can set the targetScope for your file to resourceGroup, subscription, managementGroup, or tenant. If you don't specify a target scope, Bicep assumes the resourceGroup scope.

**Create resource groups**

creating a resource group, which is a subscription-scoped resource:

A screenshot of a computer

Description automatically generated

**Submit a deployment**

When you initiate a deployment, you need to tell Azure at which scope you want to deploy it. This means that you use a different Azure CLI command for each deployment scope, as shown here:

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Azure stores metadata about each deployment. Unlike deployments at the resource group scope, there's some information you need to provide when you deploy at other scopes so that Azure can store the metadata correctly:

* **Location**: The deployment metadata has to be stored in a location that you specify. You don't need to specify a location for resource group scope deployments, because the deployment metadata uses the same location as the resource group. However, when you create a deployment at the subscription, management group, or tenant scope, you need to specify the Azure region that the deployment metadata is stored in. The resources for your deployment in these scopes aren't always created in the same location that you specified for the metadata.
* **Name**: All deployments in Azure have a name. You can ask Azure for information about a deployment by using its name. When you use the Azure CLI or Azure PowerShell to submit a deployment, you don't need to specify the name. But if you don't, the template file's filename is used as the deployment name.

**Create a subscription-scoped template**

Notice that the policy definition applies only to resources where:

* The resource type is equal to Microsoft.Compute/virtualMachines.
* The sku.name property *either* begins with Standard\_F or Standard\_G.

When you try to create a resource that matches these conditions, Azure will deny the resource creation.

**Specify the scope for a module**

Here's an example Bicep file that's deployed with a targetScope of a subscription, but uses a module to deploy some resources to a resource group

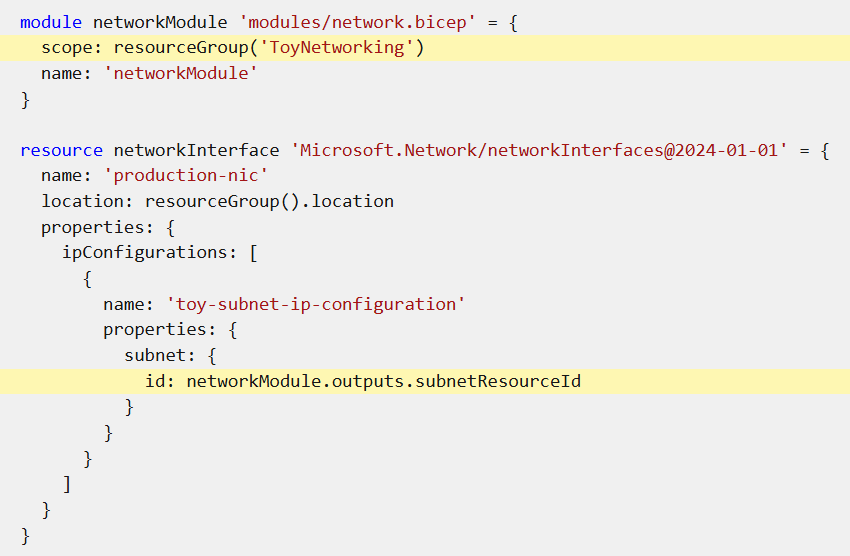
A screenshot of a computer program

Description automatically generated

**Note**

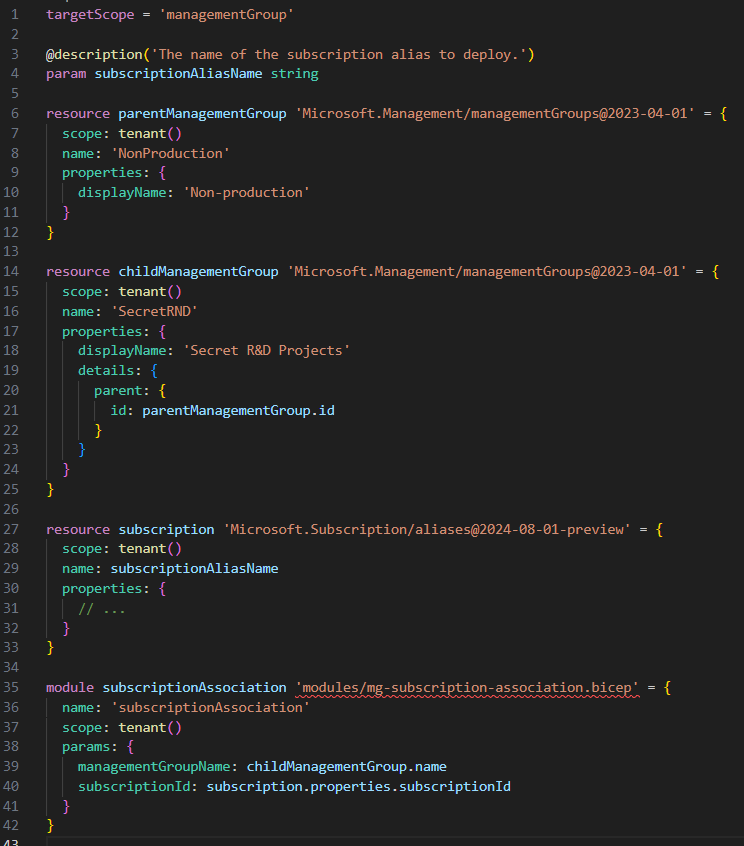
One exception is that Bicep files with a targetScope of resourceGroup or subscription can't include a module with a scope of managementGroup.

**Deploy across multiple resource groups**

 A screenshot of a computer

Description automatically generated

**Create a management group and subscription hierarchy**



**Exercise - Deploy resources to multiple scopes by using modules**

**Verify your template**

Your *main.bicep* file should look like the following:

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Clean up the resources**

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

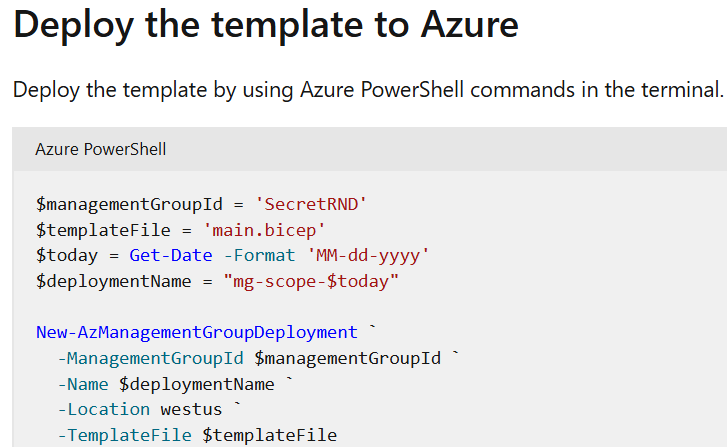
**Exercise - Deploy resources to a management group**

**Create a management group**

A close-up of a computer screen

Description automatically generated





**Manage end-to-end deployment scenarios by using Bicep and Azure Pipelines**

<https://learn.microsoft.com/en-us/training/modules/manage-end-end-deployment-scenarios-using-bicep-azure-pipelines/>

**Publish reusable Bicep code by using Azure Pipelines**

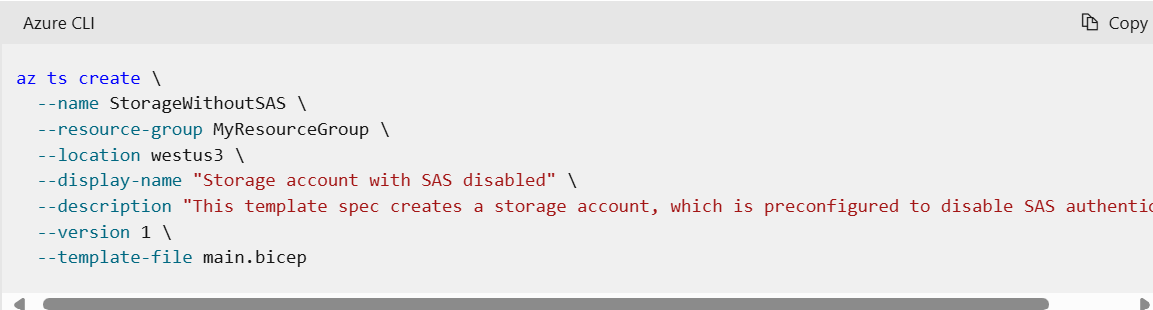
https://learn.microsoft.com/en-us/training/modules/publish-reusable-bicep-code-using-azure-pipelines/

**Publish Bicep code from a deployment pipeline**

**Template specs and modules**

* **Template specs**, which are optimized for the deployment of complete solutions. For example, suppose you've defined a set of security-hardened resources to deploy a complete virtual machine according to your company's specifications. You could publish this code as a template spec. Your colleagues could then use your template spec to deploy a complete virtual machine, even from the Azure portal.
* **Modules**, which are designed to be components of other deployments. For example, suppose you've created a Bicep file that creates a storage account. You're likely to need storage accounts in many other deployments, so you could publish the Bicep file to a registry and use it as a module throughout your organization's deployments.

**Publish template specs and modules from a pipeline**



You can convert this Azure CLI command to a pipeline step:

A screenshot of a computer

Description automatically generated

Similarly, when you publish a Bicep module from your own computer by using the Azure CLI, you use a command like the following one:

A close-up of a white background

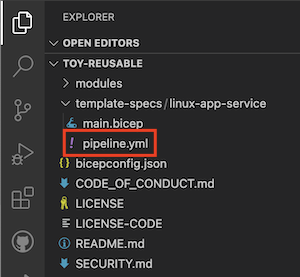
Description automatically generated

You can convert this Azure CLI command to a pipeline step, too:

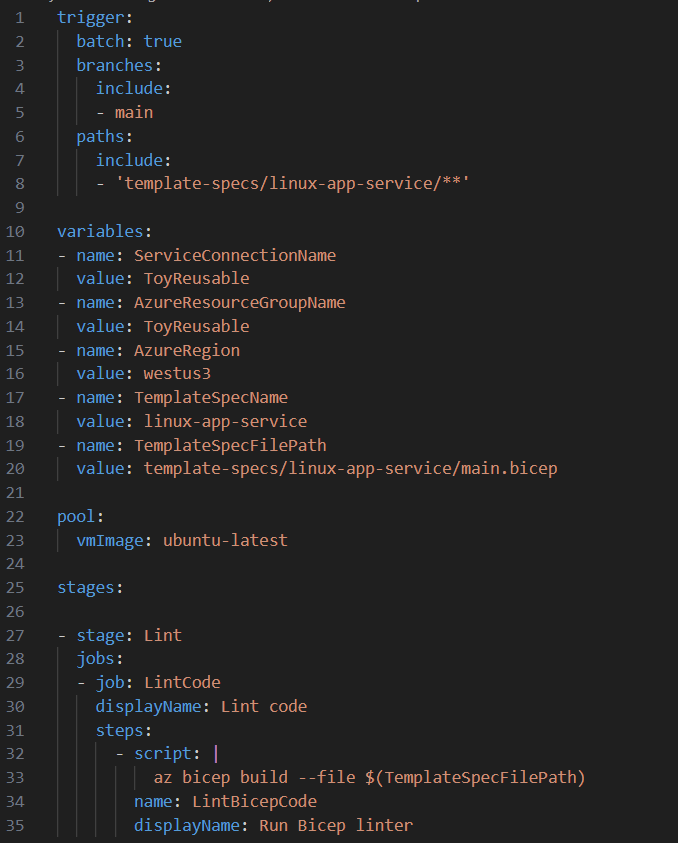
A screenshot of a computer program

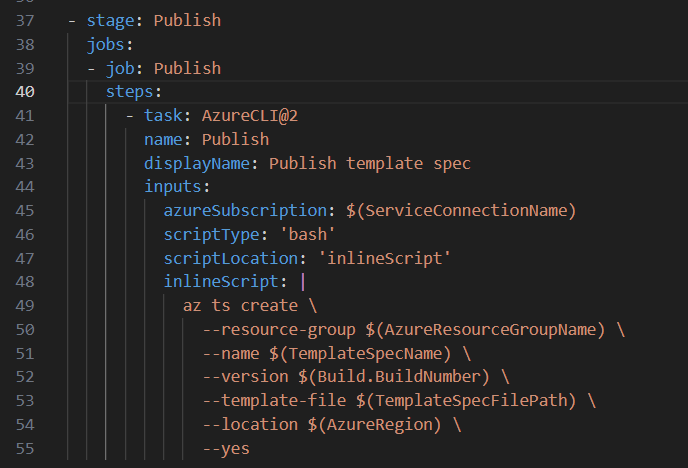
Description automatically generated

**Exercise - Publish a template spec**



**Verify and commit your pipeline definition**

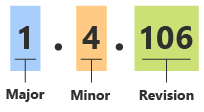




**Design a pipeline and versioning strategy**

**Version numbers**

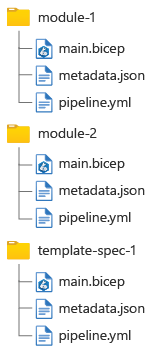
A multipart versioning system consists of a *major* version, *minor* version, and *revision* number, similar to the following example:



* **Whenever you make a breaking change, you should increment the major version number.** For example, suppose you add a new mandatory parameter or remove a parameter from your Bicep file. These are examples of breaking changes, because Bicep requires mandatory parameters to be specified at deployment time and doesn't allow setting values for nonexistent parameters. So, you would update the major version number.
* **Whenever you add something new to the code, but it isn't a breaking change, you should increment the minor version number.** For example, suppose you add a new optional parameter with a default value. Optional parameters aren't breaking changes, so you should update the minor version number.
* **Whenever you make backward-compatible bug fixes or other changes that don't affect how the code works, you should increment the revision number.** For example, suppose you refactor your Bicep code to make better use of variables and expressions. If the refactoring doesn't change your Bicep code's behavior at all, you update the revision number.
* **Your pipeline can also automatically set the revision number.** The pipeline's build number can be used as the revision number. This convention helps ensure that your version numbers are always unique, even if you don't update the other components of your version numbers.

**Versions and pipelines:**

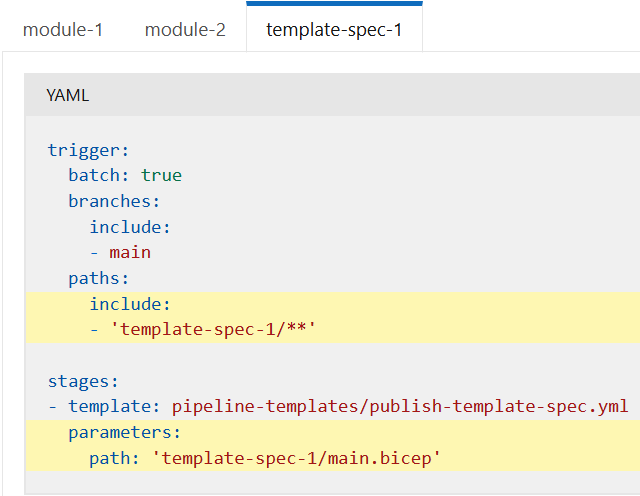
One approach is to store a *metadata file* with your Bicep code, as illustrated in the following diagram:

****

**How many pipelines?**

By using *path filters* in Azure Pipelines, you can create separate pipelines for each module or template spec within your repository. This approach helps you avoid publishing a new version of every Bicep file within the repository every time you change one file.

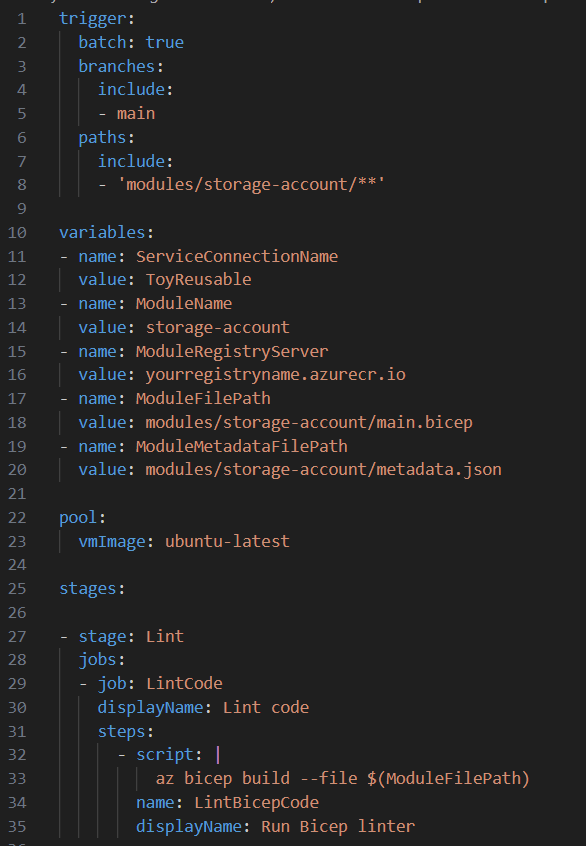
** **

****

Suppose you change only the *module-2/main.bicep* file. Only the pipeline for module 2 runs. But if you change multiple files in the same commit, each of the relevant pipelines is triggered.

**Verify and commit your pipeline definition**

Verify that your *storage\_account\_module.yml* file looks like the following example:



A black screen with text

Description automatically generated

**Publish reusable Bicep code by using GitHub Actions**

<https://learn.microsoft.com/en-us/training/modules/publish-reusable-bicep-code-using-github-actions/1-introduction>

Azure template specs enable you to create reusable deployments for complete solutions. Bicep modules and registries enable you to create components that you can add to other Bicep deployments.

**Automate Azure infrastructure change reviews by using Bicep and GitHub**

<https://learn.microsoft.com/en-us/training/modules/automate-azure-infrastructure-change-reviews-using-bicep-github/1-introduction>

**Pull request validation**

When you review a pull request for Bicep code, you might have some common steps that you go through to assess the change. These steps might include:

* Checking to see whether the Bicep file has any errors or linter warnings.
* Ensuring that any resources previously defined in the Bicep file continue to work.
* Testing any newly defined resources to ensure that they're deployed successfully and that they work as you expect.
* **Linting** your Bicep code helps to ensure that the code is semantically correct, and that it follows some baseline recommended Bicep practices.
* **Preflight validation** helps to build your confidence that the code will be deployed successfully without having to actually deploying the file. Depending on the resource type, preflight validation might look for issues such as invalid resource names, invalid regions for specific resources, and whether you've specified a configuration that can't be deployed successfully.
* **What-if**, which lists the changes that will be made to your Azure environment as a result of the deployment.
* **Deployments**, to actually deploy your resources and ensure that there are no deployment errors.
* **Testing** your resources after deployment, to ensure that they're configured according to your business requirements.

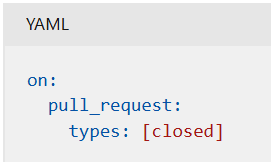
**Pull request lifecycle triggers**

you can define a workflow that runs automatically whenever a pull request is opened, synchronized, or reopened by specifying the pull\_request trigger without any additional configuration:

A close-up of a computer screen

Description automatically generated

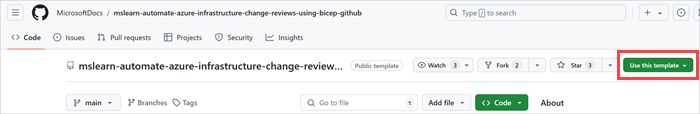
You can also specify pull request events that trigger a workflow. For example, the following workflow runs automatically whenever a pull request is closed:



**Exercise - Set up your environment**

On the GitHub site, create a repository from the template by doing the following steps:

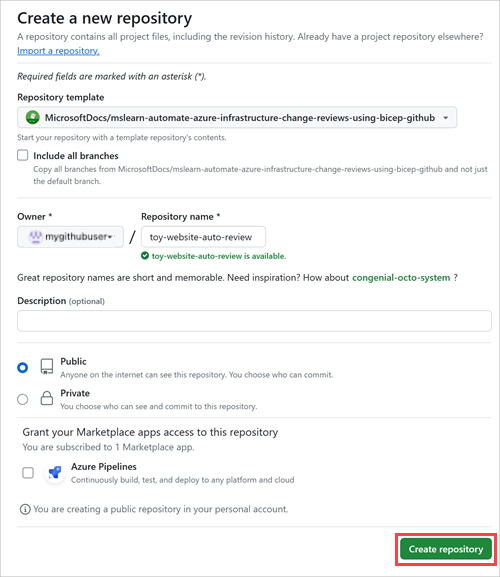
1. Select **Use this template** > **Create a new repository**.



1. Note the name of your GitHub username or organization. In the example, the GitHub user name is *mygithubuser*. You'll need this name soon.
2. Enter a name for your new project, such as *toy-website-auto-review*.
3. Select the **Public** option.

When you create your own repositories, you might want to make them private. In this module, you'll work with some features of GitHub that work only with public repositories and GitHub Enterprise accounts.

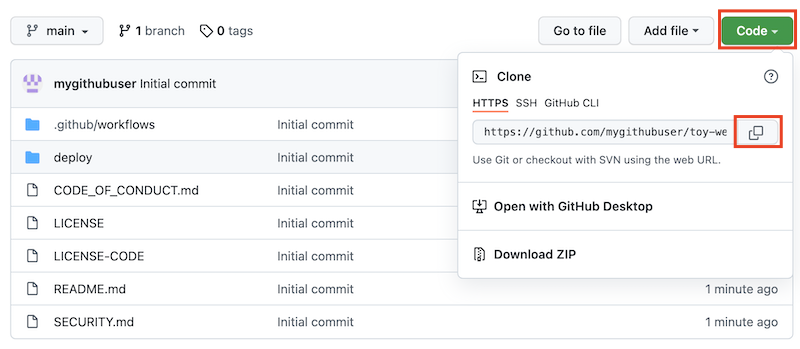
1. Select **Create repository**.



**Clone the repository**

You now have a copy of the template repository in your own account. Clone this repository locally so you can start to work in it.

1. Select **Code**, and then select the copy icon.



1. Open Visual Studio Code.
2. Open a Visual Studio Code terminal window by selecting **Terminal** > **New Terminal**. The window usually opens at the bottom of the screen.
3. In the terminal, go to the directory where you want to clone the GitHub repository on your local computer. For example, to clone the repository to the *toy-website-auto-review* folder, run the following command:

BashCopy

cd toy-website-auto-review

1. Type git clone and then paste the URL you copied earlier, which looks something like this:

BashCopy

git clone https://github.com/mygithubuser/toy-website-auto-review.git

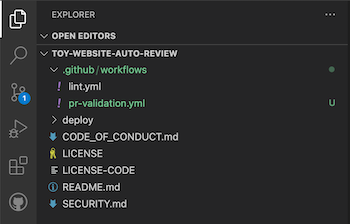
1. Reopen Visual Studio Code in the repository folder by running the following command in the Visual Studio Code terminal:

BashCopy

code -r toy-website-auto-review

**Create a workflow to run when pull requests are created and updated**

1. In Visual Studio Code, create a new file named *pr-validation.yml* in the *.github/workflows* folder.



1. Add the following code to the file:

A screenshot of a computer

Description automatically generated

This code ensures that the workflow is executed whenever a pull request is created or updated.

**Note**

You're working directly against your repository's *main* branch. You wouldn't ordinarily do this, but in this exercise you'll work against *main* to simplify the steps. In your own projects, it's important to set up branch-protection rules to protect your main branch.

**Add a lint job to your workflow**

Whenever you open or edit a pull request, you want to run a linting step for your Bicep files. There's a reusable *lint* workflow in the repository that you can call from this workflow.

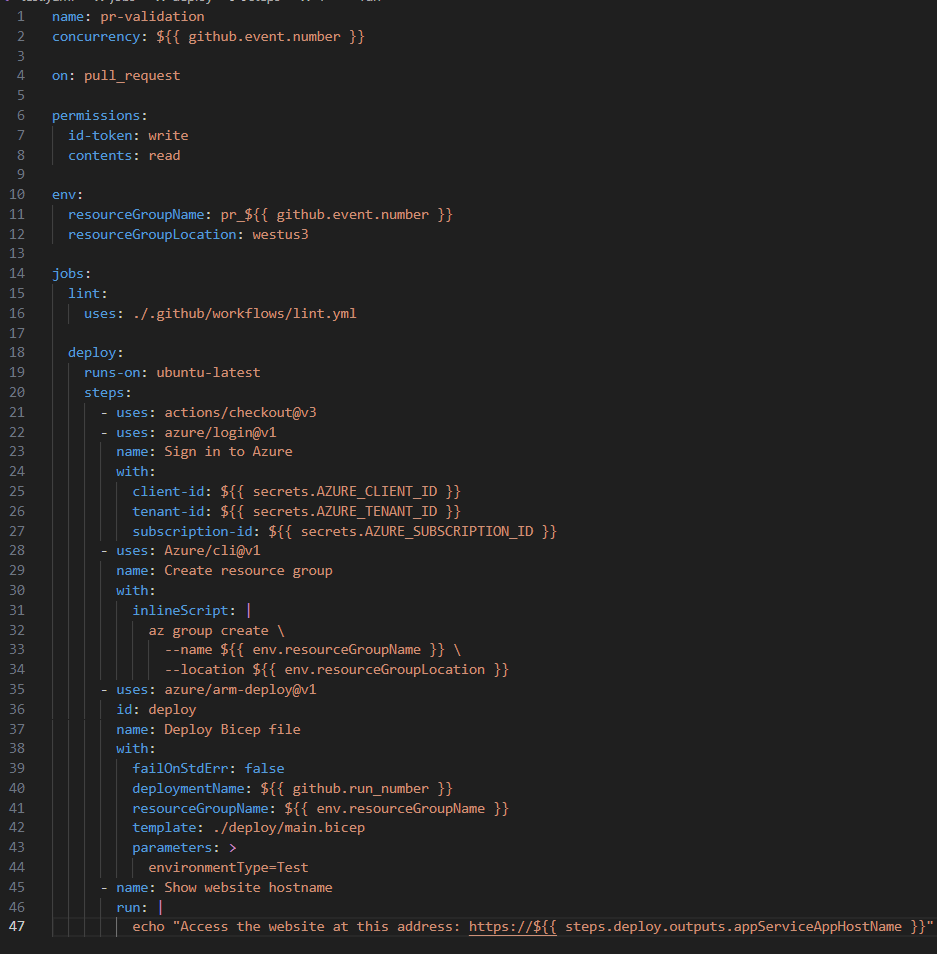
1. At the end of the existing file contents, to reuse the *lint* workflow that's defined in your repository, add the following lines:

A white screen with red text

Description automatically generated

**Exercise - Create ephemeral environments for pull requests**

Verify that your *pr-validation.yml* file looks like the following:



Verify that your *pr-closed.yml* file looks like the following:

A screenshot of a computer program

Description automatically generated

**Manage end-to-end deployment scenarios by using Bicep and GitHub Actions**

<https://learn.microsoft.com/en-us/training/modules/manage-end-end-deployment-scenarios-using-bicep-github-actions/1-introduction>

**The control plane and the data plane**

Many Azure resources provide two different *planes* for access. The *control plane* deploys and configures the resource. The *data plane* enables you to access the resource's functionality.

**How to perform data-plane operations**

When you create a deployment workflow that interacts with the data plane of your resources, you can use any of three common approaches:

* Resource Manager deployment scripts
* Workflow scripts
* Workflow actions

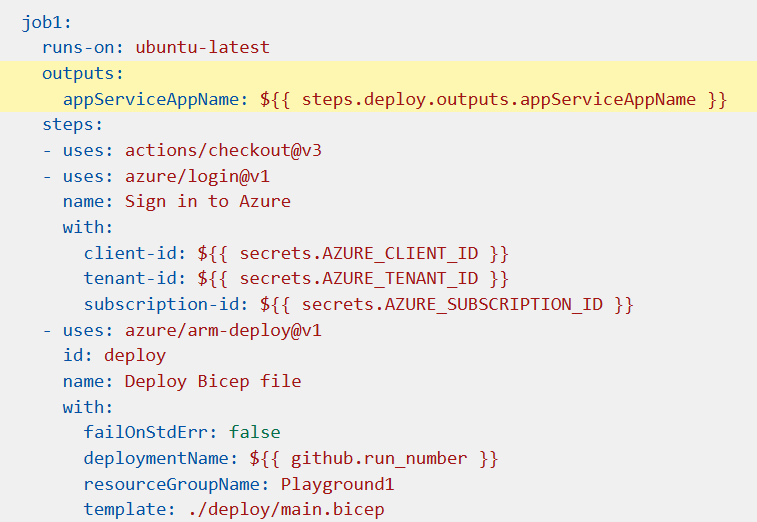
*Resource Manager deployment scripts* are defined within your Bicep file. They run Bash or PowerShell scripts, and they can interact with Azure CLI commands and Azure PowerShell cmdlets. You create a managed identity for the deployment script to use to authenticate to Azure, and Azure automatically provisions and manages the other resources it needs to run the deployment script.

Workflow actions and scripts give you flexibility and control. They also allow you to access

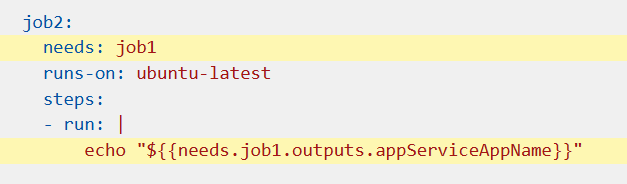
*workflow artifacts*

**Outputs:**

When you access variables that were created in another job, you need to publish the variable to make it accessible to the job that reads it.



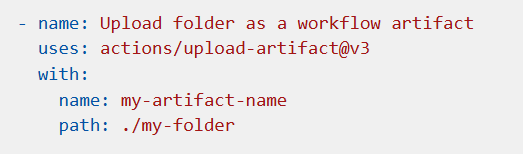
Then, in a later job, you create an explicit dependency on the job that created the variable by including the needs keyword, and you refer to the variable by using the name of the published variable:



**Workflow artifacts**

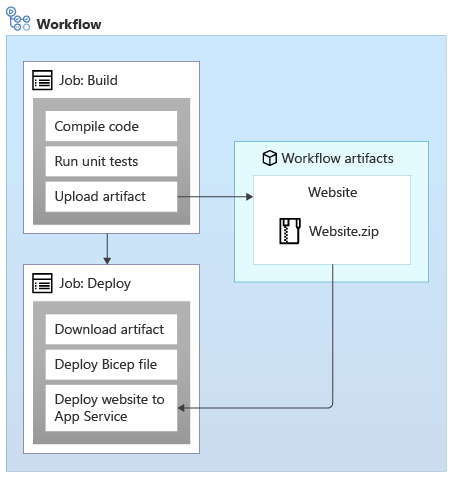
The artifacts that are generated in your workflow aren't stored in your Git repository. They're derived from the source code but aren't code themselves, so they don't belong in a source-control repository. They're created on the workflow runner's file system. A new runner is created for each workflow job, so you need a way to share the files between jobs and runners.

*Workflow artifacts* provide a way to store files in GitHub Actions, and they're associated with your workflow's particular run. You use the actions/upload-artifact workflow action to instruct GitHub Actions to upload a file or folder from the runner's file system as a workflow artifact:

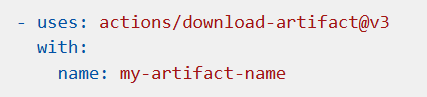


The path property is the location that contains your compiled code or output files on the job runner's file system. The contents at this location will be uploaded to the artifact. You can specify a single file, multiple files, or a folder.

Each artifact has a name, which you specify by using the name property.

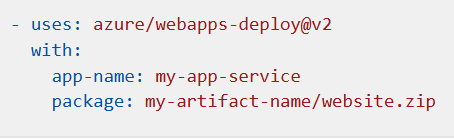


Use the actions/download-artifact action to download all of the workflow artifacts: Or, specify an artifact name to download just a specific artifact:

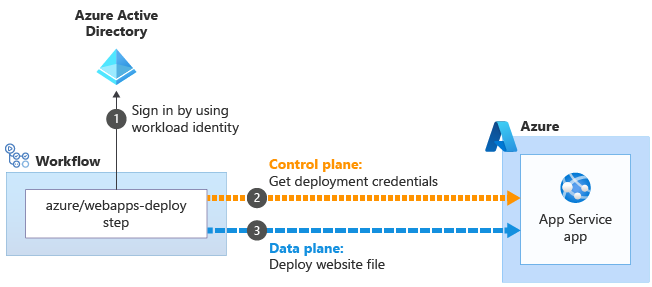


**Deploy to Azure App Service:**

The most common approach is to use the azure/webapps-deploy action:



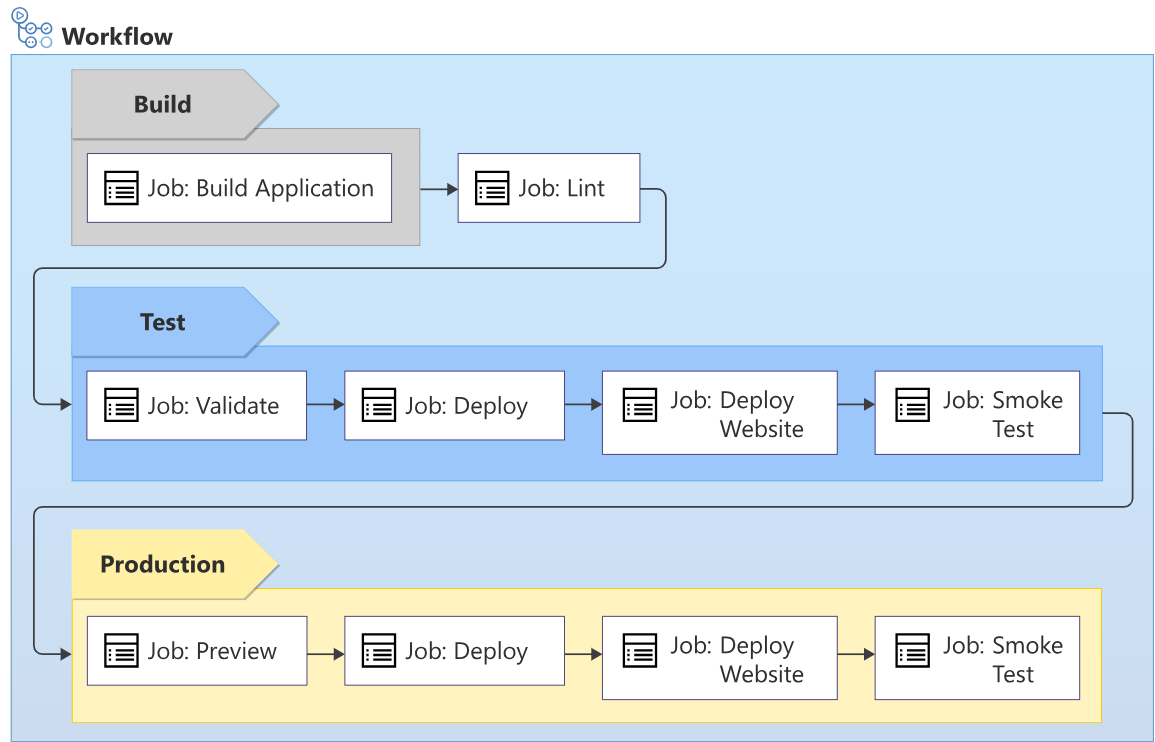
App Service has its own data plane authentication system that it uses for deployments. The azure/webapps-deploy action handles the authentication process automatically for you:



The azure/webapps-deploy action uses the identity that's associated with your job's active Azure session, which you signed in by using a workload identity . The action creates and downloads the necessary credentials for deployment . It then uses the deployment credentials when it communicates with the App Service data plane API

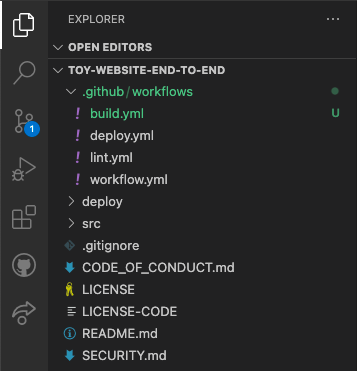
**Your deployment workflow**

In the next exercise, you'll update your deployment workflow to add new jobs to build your website's application and deploy it to each environment:



**Exercise - Deploy a web application**

1. Open Visual Studio Code.
2. In the *.github/workflows* folder, create a new file named *build.yml*.



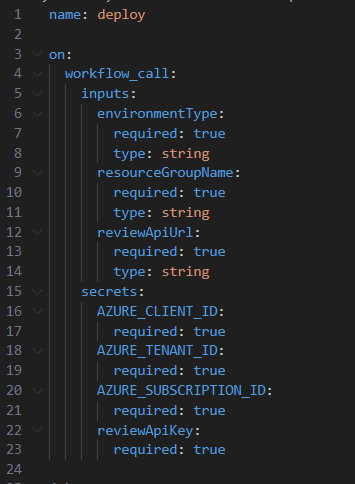
1. Add the following content to the *build.yml* workflow file:

A screenshot of a computer program

Description automatically generated

**Verify the deploy.yml file contents, and commit your changes**

1. Verify that your *deploy.yml* file looks like the following example:

 A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

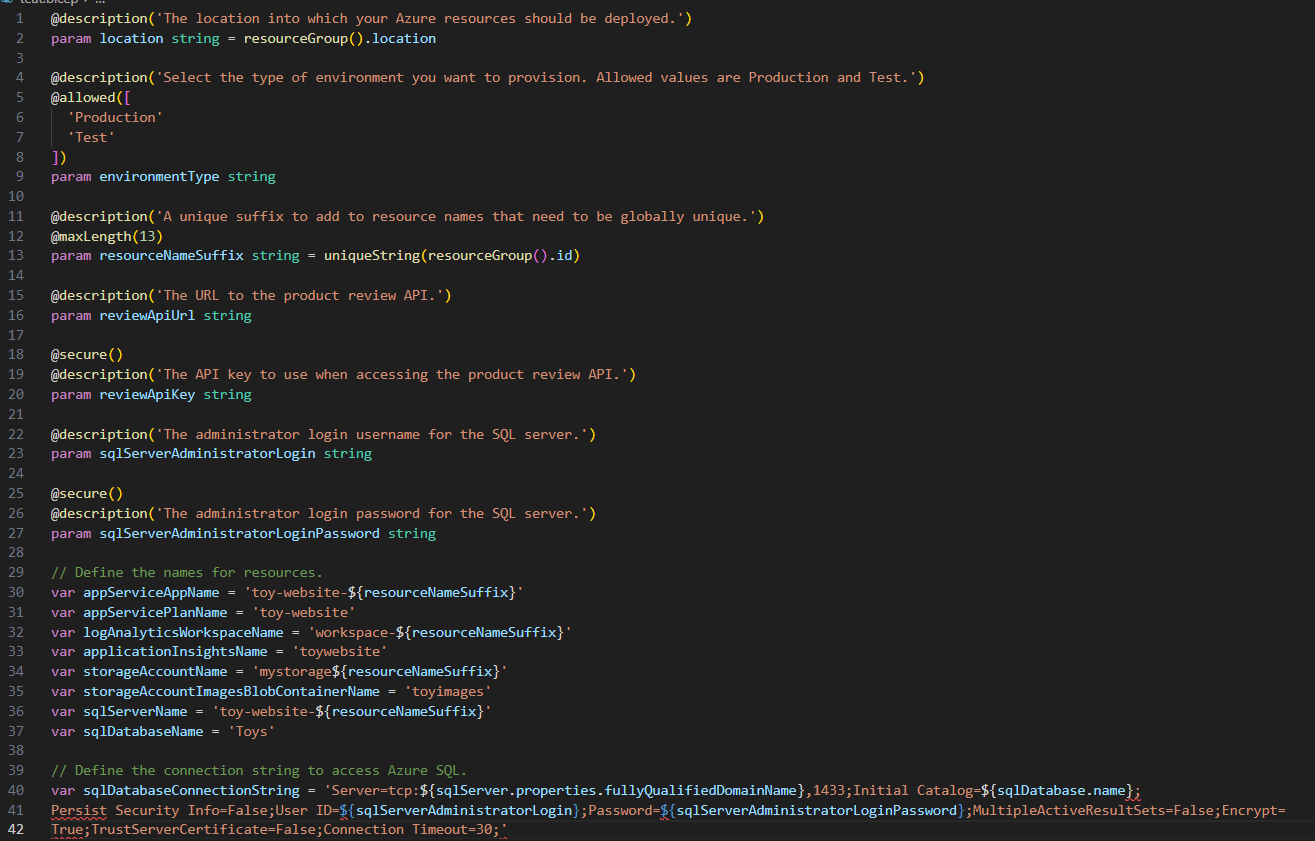
**Idempotence:**

For example, when you redeploy a Bicep file that you've already deployed, Azure Resource Manager compares the file you submitted with the existing state of your Azure resources. If there are no changes, Resource Manager doesn't do anything. The ability to re-execute an operation repeatedly is called *idempotence*.

**Exercise - Seed a storage account and database**

**Verify files and commit your changes**

1. Verify that your *main.bicep* file looks like this:



A screen shot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

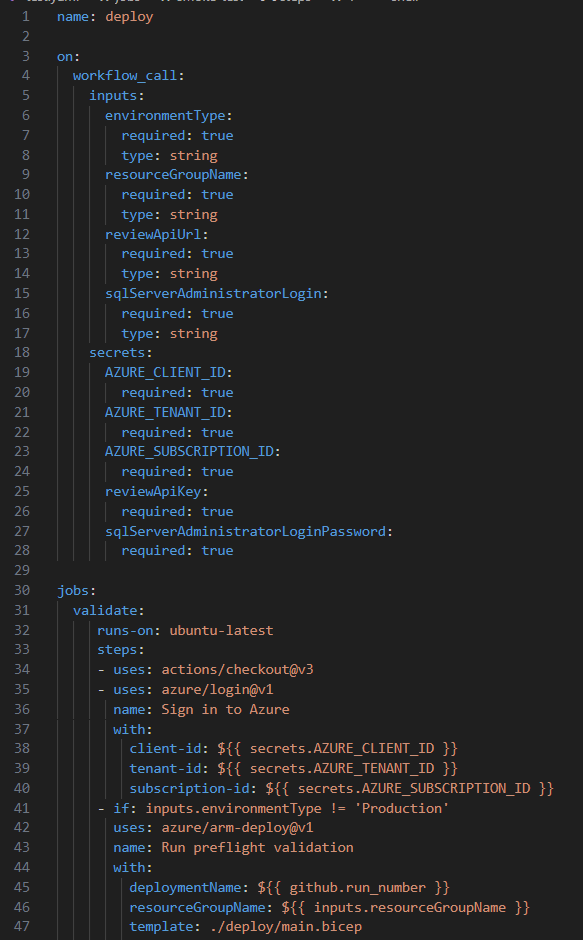
A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Verify that your *deploy.yml* file looks like this:

 A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated



A screen shot of a computer program

Description automatically generated

**Publish libraries of reusable infrastructure code by using template specs**

<https://learn.microsoft.com/en-us/training/modules/arm-template-specs/1-introduction>

**Understand template specs**

When you work with template specs, you still send the template to Azure. But instead of deploying it, Azure saves it for you to use in the future. Then, you can go back later and tell Azure to deploy the template spec. You can even use the same template spec repeatedly to deploy more environments.

Template specs are stored within Azure, so you don't need to maintain shared template files yourself. You manage who can use and modify your template specs with Azure role-based access controls. Without template specs, you need to choose a storage location, like Azure Storage, to keep your template files.

**How do template specs compare to Bicep modules?**

When you work with Bicep, you can create reusable *modules* to define sets of resources in a single file. Template specs and Bicep modules are both ways of adding reusability to your templates, but they're optimized for different things:

* Template specs are designed to be deployable as a complete template. You can deploy template specs by using Azure portal and tooling like the Azure CLI and Azure PowerShell. Bicep modules are intended to be combined into a larger deployment. However, if you create a template spec, Bicep also enables you to use it as a module if you want.
* Template specs provide versioning and access control capabilities. You need to manage the versions and security for your Bicep code yourself.
* Template specs are stored in Azure as a resource. You need to store Bicep modules somewhere that you control, like a version control system such as Git, or your file system.
* Bicep modules retain all of the original Bicep code, including comments, symbolic names, and whitespace. When you create a template spec by using Bicep, your Bicep code is converted to JSON and some of this information is lost. So you need to keep the source Bicep file somewhere else as well.

**How template specs work**

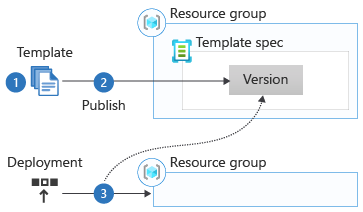
A template spec is an Azure resource, just like a storage account or virtual machine. It must be created within a resource group, although the template itself can deploy resources to a subscription, management, or tenant scope.

When you work with template specs, you create two resources:

* The *template spec* is the container resource. It contains one or multiple versions.
* *Template spec versions* contain the actual template to deploy.

/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4/resourceGroups/SharedTemplates/providers/Microsoft.Resources/templateSpecs/StorageWithoutSAS

the workflow that you follow when you use template specs:

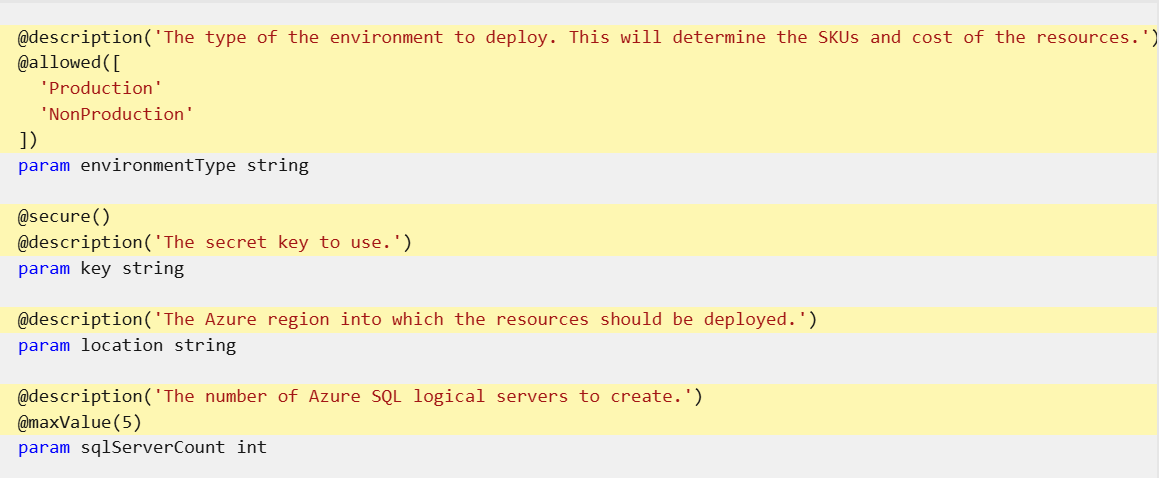


 You create a template the way you're used to. There's nothing special about a template that you create for a template spec. You declare resources, create parameters and variables, use functions, and so forth.

 When your template is ready, you create a template spec resource. You then publish your template to the template spec as a version. The tooling that you use to create template specs enables you to do these steps all in one operation. Your template spec is stored in Azure as a resource, after you publish it. You can view it, edit it, and control access to it just like any other Azure resource. You can publish your template spec from your local machine or from a deployment pipeline.

 Whenever you want to deploy your template spec, you refer to the template spec version's resource ID from the deployment. You can deploy it to any resource group, or even to another subscription or scope. Azure reads the template spec and uses that as the template for the deployment.

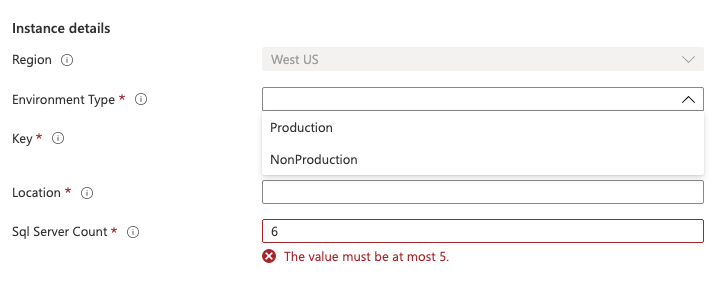
**Create a template**

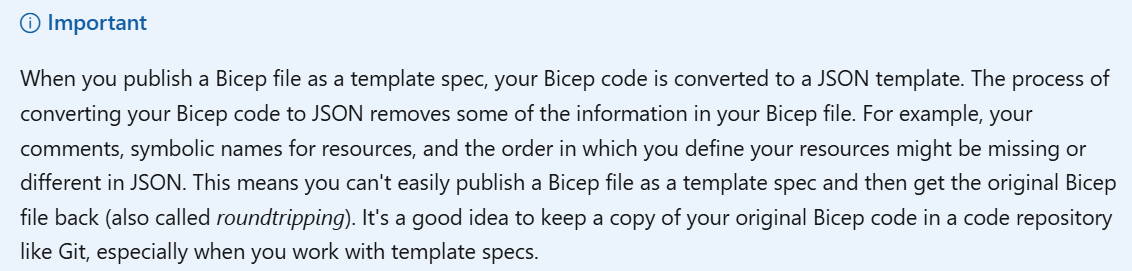


When someone deploys a template spec by using the Azure portal, the portal:

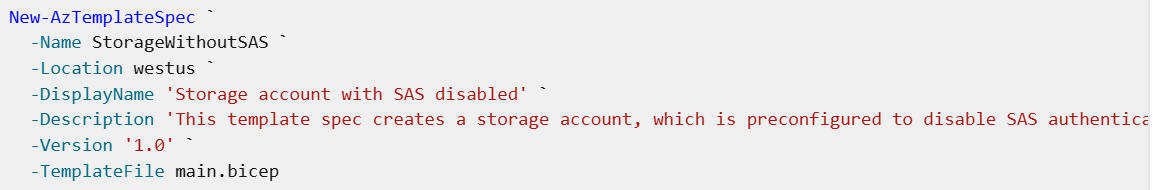
* Shows the parameter name and description.
* Hides the text entry for secure parameters.
* Enforces the allowed values, length limits, and value limits that you define.

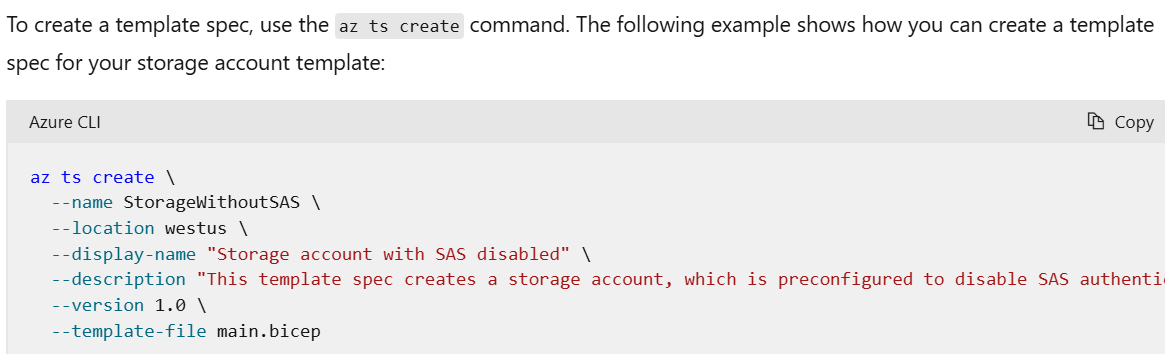
This screenshot illustrates the entry of parameter values:





To create a template spec, use the New-AzTemplateSpec cmdlet.





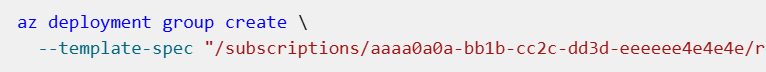
Let's look at each of the parameters:

* -Name is the resource name of the template spec, which can't include spaces.
* -Location is the location in which the template spec metadata should be created. You can deploy the template spec into any region though.
* -DisplayName is a human-readable name, which can include spaces.
* -Description is a human-readable description, which you can use to provide detail about the contents of the template spec and when someone might use it.
* -Version is the version of the template spec. You learn about versions later in this module.
* -TemplateFile is the path to the ARM template to create the template spec for.

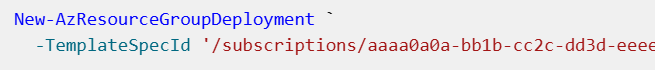
**Create a deployment by using a template spec**

To deploy a template spec to a resource group, you use the same az deployment group create

az deployment group create --template-spec "/subscriptions/aaaa0a0a-bb1b-cc2c-dd3d-eeeeee4e4e4e/resourceGroups/SharedTemplates/providers/Microsoft.Resources/templateSpecs/StorageWithoutSAS"



To deploy a template spec to a resource group, you use the same New-AzResourceGroupDeployment

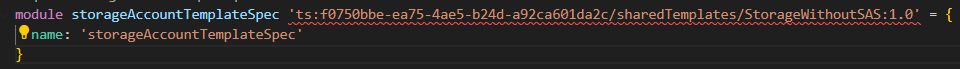


A screenshot of a computer

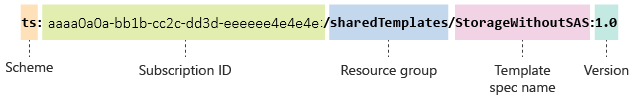
Description automatically generated A screenshot of a computer

Description automatically generated

**Use a template spec as a Bicep module**



Notice that the module path uses a special format:



There are three components to the module path, separated by the colon (:) character:

* **Scheme:** Bicep supports several types of module, which are called *schemes*. When you use a template spec as a module, you use ts as the scheme.
* **Subscription ID, resource group name, and template spec name:** These values should specify the location of the template spec resource that you previously published. You use forward slashes (/) to separate the subscription ID, resource group name, and template spec name. This section of the module path isn't the full resource ID of the template spec - it's just a few of the components of the resource ID.
* **Version:** The template spec version needs to be included.

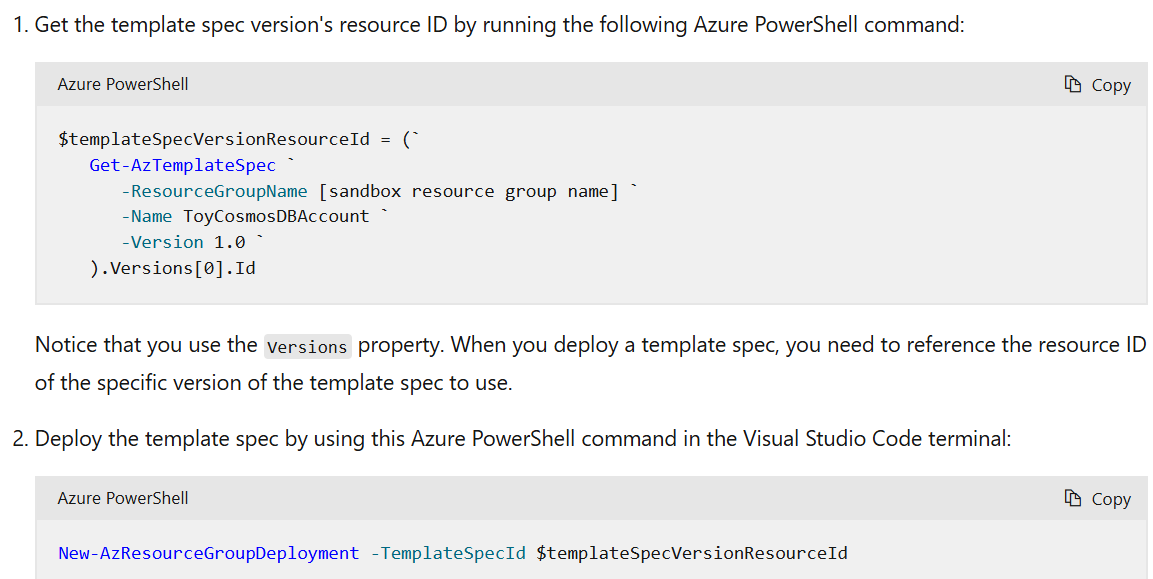
1. Create a new file called *main.bicep*.



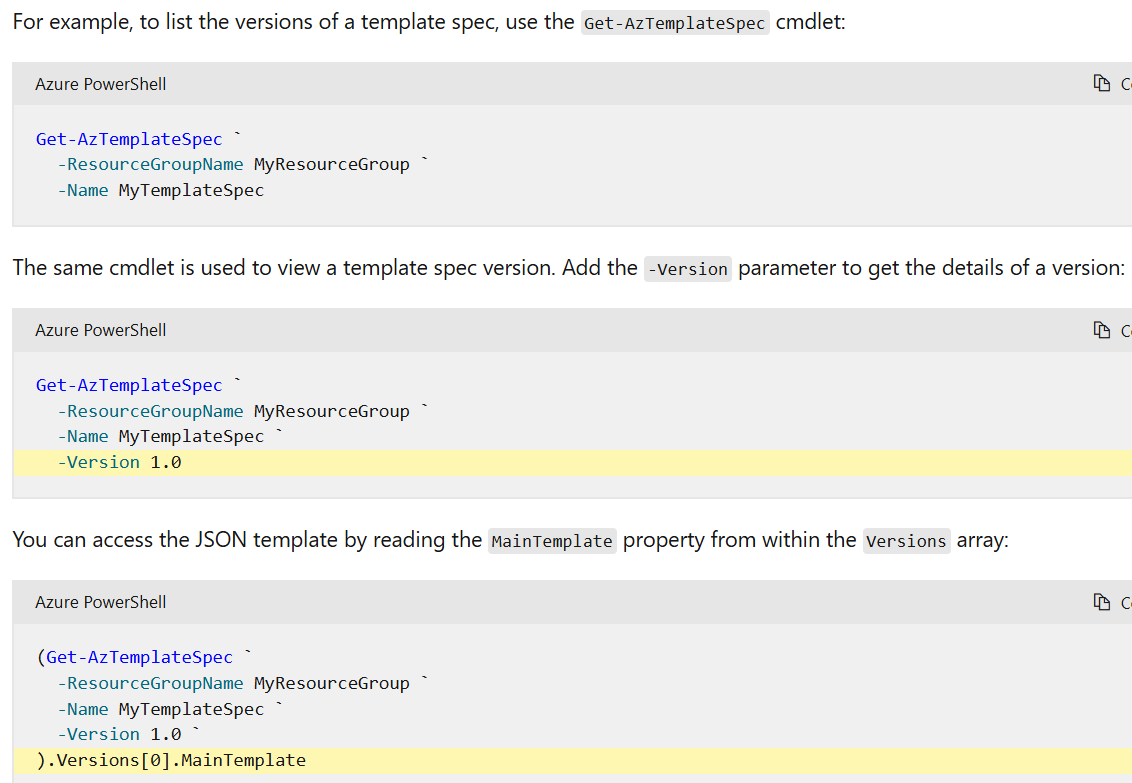
**Publish the template as a template spec**

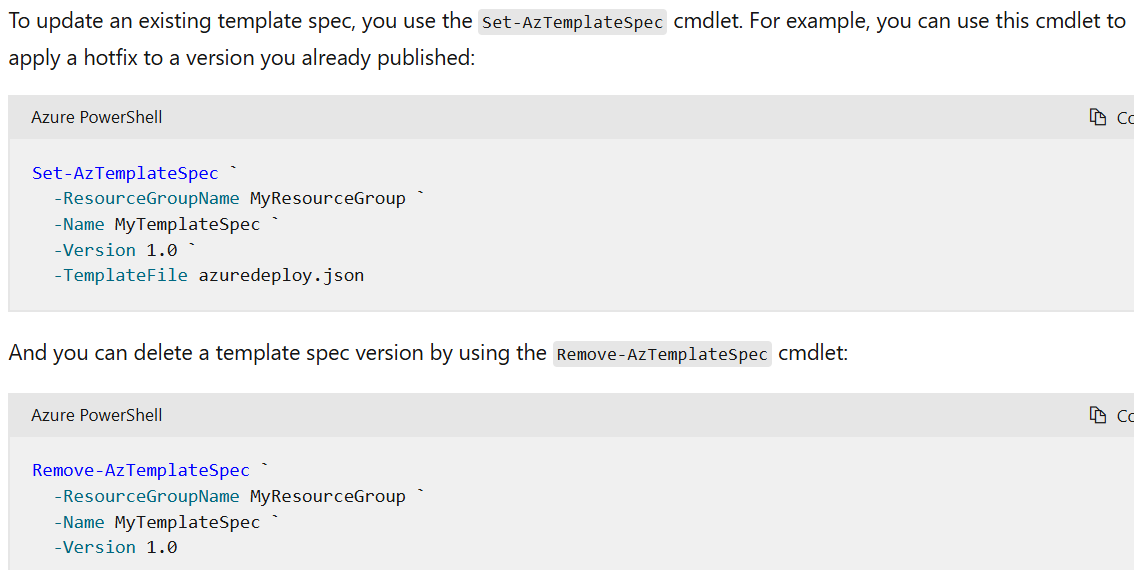
A close-up of a computer screen

Description automatically generated



**Making changes to a template spec**



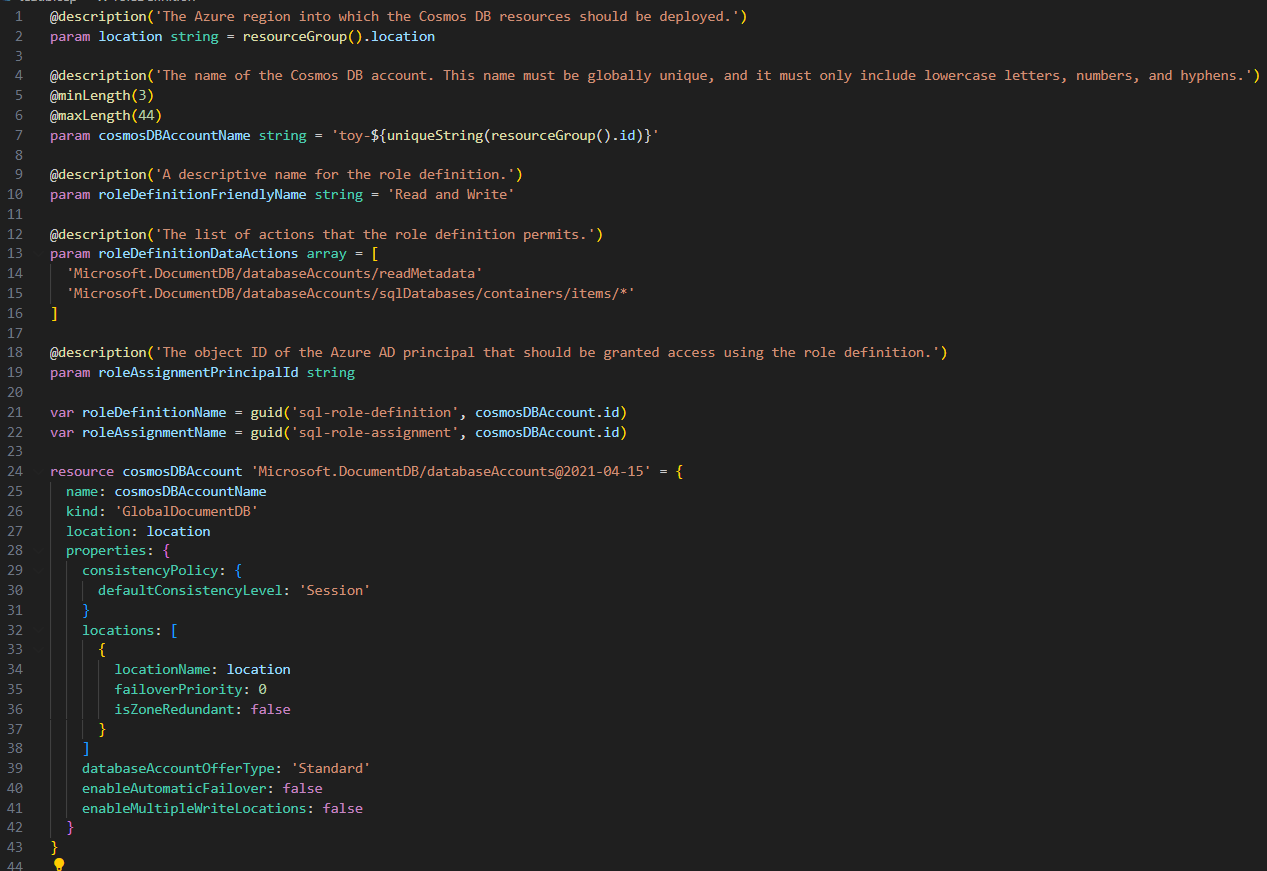


A screenshot of a computer

Description automatically generated

**Update the template**

1. In Visual Studio Code, open the *main.bicep* file.
2. Update the *main.bicep* file to include the following changes:



A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Publish libraries of reusable infrastructure code by using template specs**

https://learn.microsoft.com/en-us/training/modules/arm-template-specs/9-summary