Bicep Cheat Sheat

What is Bicep?

Bicep enables you to define your Azure resources in a simple templating language and then use the template to automate the process of deploying these resources across multiple environments and applications.

Where can I use Bicep?

Bicep can only be used to deploy resources onto Microsoft Azure.
Other cloud providers don't support Bicep as a template language.

What benefits will it provide?

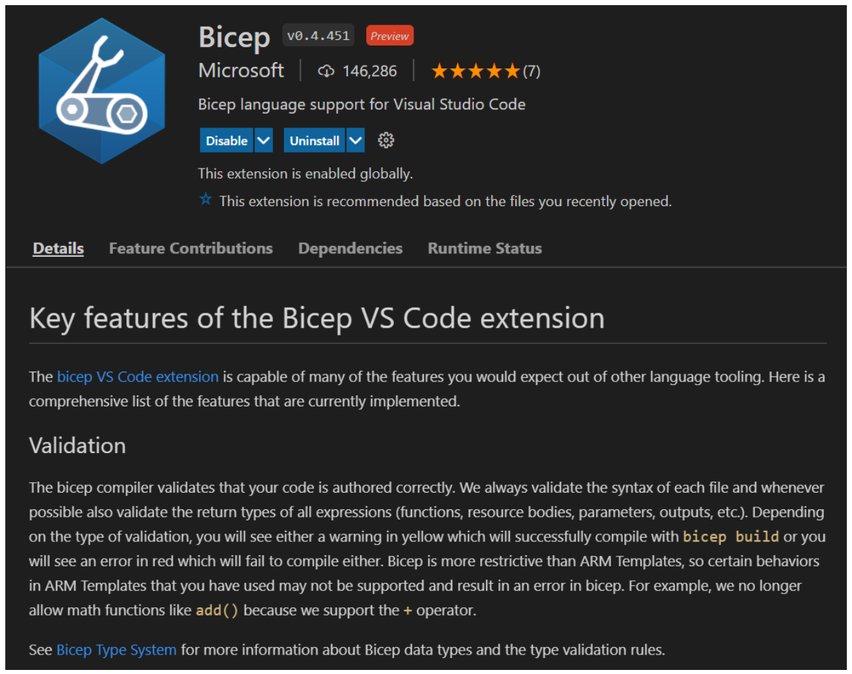
Bicep enables you to scale your solutions and to deliver with higher quality and consistency.

What do I need to install to use Bicep?

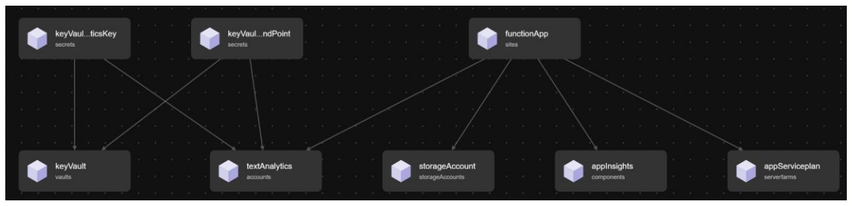
You need:

1. An Azure account to which you have contributor access. If you don't have an Azure account you can get a free trial;
2. Either:
   - The latest Azure CLI tools installed locally, or;
   - Yhe latest version of Azure PowerShell installed locally.

In addition, we highly recommend that you install Visual Studio Code and use the Bicep extension for Visual Studio Code. All code samples in this cheat sheet were prepared using Visual Studio Code.



The Visual Studio code Bicep extension also provides a tool to visualise dependencies in your Bicep template:



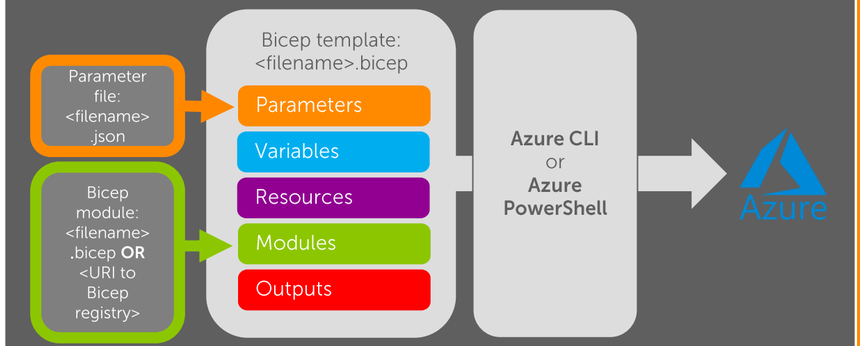
Where can I get more resources?

All of the documentation for Bicep is available online:
https://docs.microsoft.com/en-us/azure/azure-resource-manager/bicep/
The Bicep Github site provides hundreds of examples of templates that you can re-use:
https://github.com/Azure/bicep
There is an excellent Microsoft Learn module that includes practical exercises:
https://docs.microsoft.com/en-us/learn/paths/bicep-deploy/

Overview

The following illustration provides an overview how Bicep templates are structured and deployed via the Azure CLI or Azure PowerShell.

Each of the elements in the diagram are expanded further in the colour coded sections in this cheat sheet.



Parameters

Use parameters to create flexible and reusable Bicep templates. Users will be prompted for a value where no default is provided. Use the **param** keyword to create a parameter, the most basic example is:

```
param keyVaultSku string = 'Standard'
```

The following types are supported:

Type	Example
string	`param environmentName string = 'dev'`
int	`param appServicePlanInstanceCount int = 3`
bool	`param enablePublicAccess bool = true`
array	`param locations array = [ 'eastus' 'euwest' ]`
object	`param subNet object = { name: 'subnetName' addressPrefix: '10.0.0.0/24' }`

You can set the default value of parameter dynamically based on the context of deployment:

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

Decorators (denoted by **@**) can be used to augment parameters:

Decorator	Purpose	Example
@secure	Prevent Azure from capturing values in the deployment logs.	`@secure()` `param sqlServerAdministratorPassword string`
@allowed	Limit the parameter to one of a pre-defined set of values.	`@allowed([ 'dev' 'test' 'prod' ])` `param environmentName string = 'dev'`
@minLength @maxLength	Enforce a minimum length and/or maximum length in characters.	`@minLength(1)` `@maxLength(8)` `param appName string`
@description	Provide a helpful description for the parameter.	`@description('Runtime language for Function.')` `param functionRuntime string = 'python'`

- Use parameters only for settings that change between deployments. Be mindful of the default values you use.
- Use camel case for parameter names. Make your templates easy to read and understand by using clear, descriptive names for parameters. Provide @description to provide any important information about what the template needs the parameter values to be.
- Put parameters at the top of the file so your Bicep code is easy to read.
- Use the @allowed decorator sparingly. If you use this decorator too broadly, you might block valid deployments if you don't keep the list up to date.

Parameter files

Parameter files make it easy to specify parameter values together as a set. Use this approach when you have many parameters or when you need to automate your deployments.

Parameter files use JavaScript Object Notation (JSON) as follows:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/
                                deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "appServicePlanInstanceCount": {
        "value": 3
      }
    }
}
```

Element	Element value in example above	Purpose
Schema declaration.	`"$schema"`	Helps Azure Resource Manager to understand that this file is a parameter file.
Version.	`"contentVersion"`	Is a property that you can use to keep track of significant changes in your parameter file.
Parameters section.	`"parameters"`	The parameters section lists each parameter and the value you want to use.
Parameter declaration.	`"appServicePlanInstanceCount": { "value": 3 }`	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 parameter file for each environment.
- It's a good practice to include the environment name in the name of the parameter file. For example, you might have a parameter file named main.parameters.dev.json for your development environment and one named main.parameters.production.json for your production environment.

Variables

Use variables to capture complicated expressions. Then simplify your development by using variable as needed throughout your Bicep file.

Declare a variable using the **var** keyword. Unlike parameters, there is no need to define the type as this is inferred by the value assigned. Variables support the same types as parameters:

```
var functionRuntime = 'python'
```

Use variables and string operations standardise creation of unique composite names for resources:

```
@minLength(1)
@maxLength(4)
@description('A short prefix to be used in resource names.')
param appNamePrefix string
// Generate unique suffix using resource group ID as seed
param appNameSuffix string = uniqueString(resourceGroup().id)
@allowed([
  'dev'
  'tst'
  'prd'
])
param environmentName string = 'dev'

var appName = '${appNamePrefix}-${environmentName}-${appNameSuffix}'
var appServicePlanName = 'spln-${appName}'
var storageAccountName = 'stor${replace(appName, '-', '')}'
```

Applying a conditions to select appropriate for the properties of a resource.

```
var storageSkuName = (environmentName == 'prod') ? 'Standard_GRS' : 'Standard_LRS'
```

- Use camel case for variable names. Make your templates easy to read and understand by using clear, descriptive names for variables.
- Use string interpolation to create resource name variables. Add a prefix to the unique string generated by the **uniqueString()** function to avoid creates name that starts with a number as this is not allowed by some resource types.

Resources

Resources are deployed denoted using the **resource** keyword as follows:

```
resource  storageAccount 'Microsoft.Storage/storageAccounts@2021-02-01' = {
    name: 'appdatastorage'
    ...
}
```

Element	Element value in example above	Purpose
Symbolic name.	`storageAccount`	used within Bicep to refer to the resource during deployment, they don't show up in Azure.
Resource provider / resource type.	`Microsoft.Storage/storageAccounts`	Is the type of resource that you want to deploy. The following site lists all resource types that are available to deploy via Bicep and provides full definition of the template for each type: Azure Resource Manager template reference
API version.	`2021-02-01`	The version resource provider API that will be used to create the resource.
Resource name.	`name: 'appdatastorage'`	The name of the resource as it will appear in Azure Portal. This must comply with the naming convention for that specific resource.

Access any resource in Bicep by using the symbolic name. Use resource properties or functions to generate outputs:

Use case	Example
Access a specific property in a resource.	`applicationInsights.properties.InstrumentationKey`
Use the getSecret() function to retrieve the value of specific secret in key vault.	`keyVault.getSecret('ApiKey')`

If the resource already exists and isn't being deployed in the Bicep file, you can still get a symbolic reference to the resource using the **existing** keyword:

```
resource keyVault 'Microsoft.KeyVault/vaults@2021-04-01-preview' existing = {
  name: 'mykeyvault'
}
```

- To make your Bicep file easier to read and understand, use variables to contain complex expressions. This will avoid cluttering your resource definitions with logic.
- Use resource properties as outputs, rather than making assumptions about how resources will behave. It's safer to have the resource tell you its own properties.
- Use a recent API version for each resource. New features in Azure services are sometimes available only in newer API versions.

Resource dependencies

In most cases dependencies between resources are **implicit** - Bicep will manage them without you needing to specify them. In the example below, the **keyVaultEndPointName** resource will not be created until after:

- the parent **keyVault** resource has been created, **AND**;
- the **textAnalytics** resource which is references has been created.

```
resource keyVaultEndPointName 'Microsoft.KeyVault/vaults/secrets@2019-09-01' = {
  parent: keyVault
  name: 'endPointName'
  properties: {
    value: textAnalytics.properties.endpoint
  }
}
```

You can create **explicit** dependencies by using the dependsOn element:

```
resource otherZone 'Microsoft.Network/dnszones@2018-05-01' = {
  name: 'myZone'
  location: 'global'
  dependsOn: [
    dnsZone
  ]
}
```

- Setting unnecessary explicit dependencies using **dependsOn** slows deployment time because Resource Manager can't deploy those resources in parallel.
- Even though explicit dependencies are sometimes required, the need for them is rare. In most cases you have a symbolic reference available to imply the dependency between resources. If you find yourself using dependsOn you should consider if there is a way to get rid of it.

Modules

Every Bicep file can be consumed as a module. A module only exposes parameters and outputs as a contract to other Bicep files. Use the **module** keyword to consume a module.

Modules can be consumed from a local file system. In the example below, a separate Bicep module is called which takes care of the complexities of setting up the storage account resource:

```
param namePrefix string
param location string = resourceGroup().location

module stgModule './storage.bicep' = {
  name: 'storageDeploy'
  params: {
    storagePrefix: namePrefix
    location: location
  }
}
output storageEndpoint object = stgModule.outputs.storageEndpoint
```

Modules can also be consumed from a Bicep registry is hosted on Azure Container Registry (ACR). For the example above, the following syntax would be adopted:

```
module stgModule 'br:<registry-name>.azurecr.io/bicep/modules/storage:v1' = {
  …
  }
```

Element	Element value in example above	Purpose
Symbolic name.	`stgModule`	Identifier for the module.
Module file **OR** Bicep registry URL.	`'./storage.bicep'` **OR** `'br:<registry-name>.azurecr.io/bicep/modules/storage:v1'`	Module files must be referenced by using relative paths. The Windows backslash (\) character is unsupported. Paths can contain spaces.
Name property.	`name: 'storageDeploy'`	The name property is used when Bicep generates the template as the name of the nested deployment resource.
Parameters object.	`params: { storagePrefix: namePrefix location: location }`	The params property contains any parameters to pass to the module file. These parameters should match the parameters defined in the Bicep file.
Module outputs.	`stgModule.outputs.storageEndpoint`	Use this syntax to get access to an output value from a module.

- Use modules to break down a complex solution into more manageable parts and to abstract away complex details of the resource declarations, which can increase readability.
- Design your modules with re-use in mind, give each module a clear purpose and adopt parameters (inputs) and outputs that make sense. Make it as self-contained as possible.
- In general, it's better for modules to combine multiple related resources. For example, to define a database server and all its child databases.
- A module should not output secrets.

Outputs

Use outputs to pass data generated during deployment of the Bicep template back to whoever or whatever is executing it.

Use the **output** keyword to declare an output:

```
output appName string =  appServiceAppName
```

Element	Element value in example above	Purpose
Name of output.	`appName`	When a template is deployed successfully, the output value will be tagged with the name that you specify here so that it can be accessed symbolically.
Type.	`string`	Bicep outputs support the same types as parameters.
Value.	`appServiceAppName`	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..

Output the properties of resources by using the resources symbolic reference in Bicep:

```
output textAnalyticsEndPoint string =  textAnalytics.properties.endpoint
```

- Don't create outputs for secret values like connection strings or keys. Anyone with access to your resource group can read outputs from templates.

Deployment: Azure CLI

If you're deploying to a resource group that doesn't exist, create the resource group:

```
az group create --name rg-myapp-dev-westeu --location "West Europe"
```

Use the **New-AzResourceGroupDeployment** PowerShell command to deploy the Bicep:

```
az deployment group create \
  --name AppDeployDevelopment \
  --resource-group rg-myapp-dev-westeu \
  --template-file ./main.bicep \
  --parameters appName=myapp ./mapp.dev.parameters.json
  --confirm-with-what-if
```

Element	Element value in example above	Purpose
Name of deployment.	`--name`	Sets the name of the deployment as it will appear in the activity log in the Azure portal.
Resource group.	`--resource-group`	The name of the resource group into which the resources should be deployed.
Bicep template.	`--template-file`	The path to the bicep template file that should be deployed.
Bicep parameter.	`--parameters`	The value of any parameters (e.g. a parameter named "appName") for the Bicep template can be set in line and/or the path to a file containing the parameters can be provided.
Preview changes and confirm.	`--confirm-with-what-if`	This flag allows you to preview the changes before committing them into your environment.

Deployment: PowerShell

If you're deploying to a resource group that doesn't exist, create the resource group:

```
New-AzResourceGroup -Name rg-app-dev-westeu -Location "West Europe"
```

Use the **New-AzResourceGroupDeployment** PowerShell command to deploy the Bicep:

```
New-AzResourceGroupDeployment `
  -Name AppDeployDevelopment `
  -ResourceGroupName rg-myapp-dev-westeu `
  -TemplateFile ./main.bicep `
  -appName "myapp" `
  -TemplateParameterFile ./mapp.dev.parameters.json `
  -Confirm
```

Element	Element value in example above	Purpose
Name of deployment.	`-Name`	Sets the name of the deployment as it will appear in the activity log in the Azure portal.
Resource group.	`-ResourceGroupName`	The name of the resource group into which the resources should be deployed.
Bicep template.	`-TemplateFile`	The path to the bicep template file that should be deployed.
Bicep parameter.	`-appName`	The value of any parameters (e.g. a parameter named "appName") for the Bicep template can be set in line in the PowerShell command.
Bicep parameter file.	`-TemplateParameterFile`	The path to the parameter file.
Preview changes and confirm.	`-Confirm`	This flag allows you to preview the changes before committing them into your environment.