Best practices for Bicep

Article • 06/23/2023

This article recommends practices to follow when developing your Bicep files. These practices make your Bicep file easier to understand and use.

Training resources

If you would rather learn about Bicep best practices through step-by-step guidance, see Structure your Bicep code for collaboration.

Parameters

- Use good naming for parameter declarations. Good names make your templates easy to read and understand. Make sure you're using clear, descriptive names, and be consistent in your naming.

- Think carefully about the parameters your template uses. Try to use parameters for settings that change between deployments. Variables and hard-coded values can be used for settings that don't change between deployments.

- Be mindful of the default values you use. Make sure the default values are safe for anyone to deploy. For example, consider using low-cost pricing tiers and SKUs so that someone deploying the template to a test environment doesn't incur a large cost unnecessarily.

- Use the `@allowed` decorator sparingly. If you use this decorator too broadly, you might block valid deployments. As Azure services add SKUs and sizes, your allowed list might not be up to date. For example, allowing only Premium v3 SKUs might make sense in production, but it prevents you from using the same template in non-production environments.

- It's a good practice to provide descriptions for your parameters. Try to make the descriptions helpful, and provide any important information about what the template needs the parameter values to be.

  You can also use `//` comments to add notes within your Bicep files.

- You can put parameter 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.
- It's a good practice to specify the minimum and maximum character length for parameters that control naming. These limitations help avoid errors later during deployment.

For more information about Bicep parameters, see Parameters in Bicep.

Variables

- When you define a variable, the data type isn't needed. Variables infer the type from the resolve value.
- You can use Bicep functions to create a variable.
- After a variable is defined in your Bicep file, you reference the value using the variable's name.

For more information about Bicep variables, see Variables in Bicep.

Names

- Use lower camel case for names, such as `myVariableName` or `myResource`.
- The uniqueString() function is useful for creating unique resource names. When you provide the same parameters, it returns the same string every time. Passing in the resource group ID means the string is the same on every deployment to the same resource group, but different when you deploy to different resource groups or subscriptions.
- It's a good practice to use template expressions to create resource names, like in this example:

  Bicep

  ```
  param shortAppName string = 'toy'
  param shortEnvironmentName string = 'prod'
  ```

```
param appServiceAppName string =
'${shortAppName}-${shortEnvironmentName}-${uniqueString(resourceGroup().id
)}'
```

Using template expressions to create resource names gives you several benefits:

- Strings generated by `uniqueString()` aren't meaningful. It's helpful to use a template expression to create a name that includes meaningful information, such as a short descriptor of the project or environment name, as well as a random component to make the name more likely to be unique.
- The `uniqueString()` function doesn't guarantee globally unique names. By adding additional text to your resource names, you reduce the likelihood of reusing an existing resource name.
- Sometimes the `uniqueString()` function creates strings that start with a number. Some Azure resources, like storage accounts, don't allow their names to start with numbers. This requirement means it's a good idea to use string interpolation to create resource names. You can add a prefix to the unique string.
- Many Azure resource types have rules about the allowed characters and length of their names. Embedding the creation of resource names in the template means that anyone who uses the template doesn't have to remember to follow these rules themselves.

- Avoid using `name` in a symbolic name. The symbolic name represents the resource, not the resource's name. For example, instead of the following syntax:

Bicep

```
resource cosmosDBAccountName 'Microsoft.DocumentDB/databaseAccounts@2023-
04-15' = {
```

Use:

Bicep

```
resource cosmosDBAccount 'Microsoft.DocumentDB/databaseAccounts@2023-04-
15' = {
```

- Avoid distinguishing variables and parameters by the use of suffixes.

Resource definitions

- Instead of embedding complex expressions directly into resource properties, use variables to contain the expressions. This approach makes your Bicep file easier to read and understand. It avoids cluttering your resource definitions with logic.

- Try to use resource properties as outputs, rather than making assumptions about how resources will behave. For example, if you need to output the URL to an App Service app, use the defaultHostname property of the app instead of creating a string for the URL yourself. Sometimes these assumptions aren't correct in different environments, or the resources change the way they work. It's safer to have the resource tell you its own properties.

- It's a good idea to use a recent API version for each resource. New features in Azure services are sometimes available only in newer API versions.

- When possible, avoid using the reference and resourceId functions in your Bicep file. You can access any resource in Bicep by using the symbolic name. For example, if you define a storage account with the symbolic name toyDesignDocumentsStorageAccount, you can access its resource ID by using the expression `toyDesignDocumentsStorageAccount.id`. By using the symbolic name, you create an implicit dependency between resources.

- Prefer using implicit dependencies over explicit dependencies. Although the `dependsOn` resource property enables you to declare an explicit dependency between resources, it's usually possible to use the other resource's properties by using its symbolic name. This approach creates an implicit dependency between the two resources, and enables Bicep to manage the relationship itself.

- If the resource isn't deployed in the Bicep file, you can still get a symbolic reference to the resource using the `existing` keyword.

Child resources

- Avoid nesting too many layers deep. Too much nesting makes your Bicep code harder to read and work with.

- Avoid constructing resource names for child resources. You lose the benefits that Bicep provides when it understands the relationships between your resources. Use the `parent` property or nesting instead.

Outputs

- Make sure you don't create outputs for sensitive data. Output values can be accessed by anyone who has access to the deployment history. They're not appropriate for handling secrets.

- Instead of passing property values around through outputs, use the existing keyword to look up properties of resources that already exist. It's a best practice to look up keys from other resources in this way instead of passing them around through outputs. You'll always get the most up-to-date data.

For more information about Bicep outputs, see Outputs in Bicep.

Tenant scopes

You can't create policies or role assignments at the tenant scope. However, if you need to grant access or apply policies across your whole organization, you can deploy these resources to the root management group.

Next steps

- For an introduction to Bicep, see Bicep quickstart.
- For information about the parts of a Bicep file, see Understand the structure and syntax of Bicep files.