

# Introduction

Completed 100 XP

- 2 minutes

JSON Azure Resource Manager templates (ARM templates) allow you to specify your project's infrastructure in a declarative and reusable way. The templates can be versioned and saved in the same source control as your development project.

Suppose you're managing a software team that's developing an inventory system for your partner companies. You plan to deploy this product to Azure, and each partner company will have its own solution. Different policies for each deployment will be implemented through different Azure storage accounts. You decide to use the practice of *infrastructure as code* by using ARM templates. This approach lets you track the different versions and ensure that your infrastructure deployments for each environment are consistent and flexible.

In this module, you're introduced to ARM template structure. You also practice creating and deploying an ARM template to Azure.

## Note

Bicep is a new language for defining your Azure resources. It has a simpler authoring experience than JSON, along with other features that help improve the quality of your infrastructure as code. We recommend that anyone new to infrastructure as code on Azure use Bicep instead of JSON. To learn about Bicep, see [Deploy and manage resources in Azure by using Bicep](#).

## Learning objectives

In this module, you will:

- Implement an JSON ARM template by using Visual Studio Code.
- Declare resources and add flexibility to your template by adding parameters and outputs.

## Prerequisites

- Familiarity with Azure, including the Azure portal, subscriptions, resource groups, and resource definitions.

- An Azure account. You can get a free account [here](#).
  - [Visual Studio Code](#) installed locally.
  - The [Azure Resource Manager Tools for Visual Studio Code](#) extension installed locally.
  - Either:
    - The latest [Azure CLI](#) tools installed locally.
    - The latest [Azure PowerShell](#) installed locally.
- 

## Next unit: Explore Azure Resource Manager template structure

# Explore Azure Resource Manager template structure

Completed 100 XP

- 7 minutes

In this unit, you learn about using Azure Resource Manager templates (ARM templates) to implement infrastructure as code. You survey the sections of an ARM template, learn how to deploy your ARM template to Azure, and delve into detail on the *resources* section of the ARM template.

## What is infrastructure as code?

*Infrastructure as code* enables you to describe, through code, the infrastructure that you need for your application.

With infrastructure as code, you can maintain both your application code and everything you need to deploy your application in a central code repository. The advantages to infrastructure as code are:

- Consistent configurations
- Improved scalability
- Faster deployments
- Better traceability

# What is an ARM template?

ARM templates are JavaScript Object Notation (JSON) files that define the infrastructure and configuration for your deployment. The template uses a *declarative syntax*. The declarative syntax is a way of building the structure and elements that outline what resources will look like without describing its control flow. Declarative syntax is different than *imperative syntax*, which uses commands for the computer to perform. Imperative scripting focuses on specifying each step in deploying the resources.

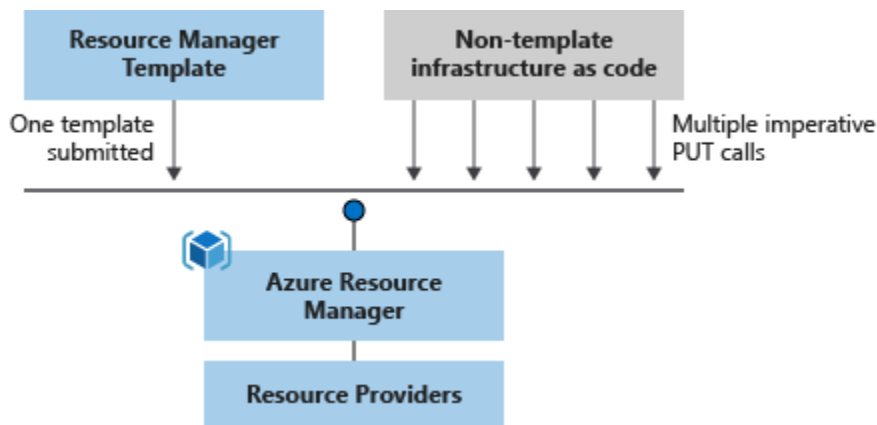
ARM templates allow you to declare what you intend to deploy without having to write the sequence of programming commands to create it. In an ARM template, you specify the resources and the properties for those resources. Then [Azure Resource Manager](#) uses that information to deploy the resources in an organized and consistent manner.

## Benefits of using ARM templates

ARM templates allow you to automate deployments and use the practice of infrastructure as code (IaC). The template code becomes part of your infrastructure and development projects. Just like application code, you can store the IaC files in a source repository and version it.

ARM templates are *idempotent*, which means you can deploy the same template many times and get the same resource types in the same state.

Resource Manager orchestrates the deployment of the resources so they're created in the correct order. When possible, resources will also be created in parallel, so ARM template deployments finish faster than scripted deployments.



Resource Manager also has built-in validation. It checks the template before starting the deployment to make sure the deployment will succeed.

If your deployments become more complex, you can break your ARM templates into smaller, reusable components. You can link these smaller templates together at deployment time. You can also nest templates inside other templates.

In the Azure portal, you can review your deployment history and get information about the state of the deployment. The portal displays values for all parameters and outputs.

You can also integrate your ARM templates into continuous integration and continuous deployment (CI/CD) tools like [Azure Pipelines](#), which can automate your release pipelines for fast and reliable application and infrastructure updates. By using Azure DevOps and ARM template tasks, you can continuously build and deploy your projects.

## ARM template file structure

When writing an ARM template, you need to understand all the parts that make up the template and what they do. ARM template files are made up of the following elements:

Element	Description
<b>schema</b>	A required section that defines the location of the JSON schema file that describes the structure of JSON objects. The schema file number you use depends on the scope of the deployment and your JSON editor.
<b>contentVersion</b>	A required section that defines the version of your template (such as 1.0.0.0). You can use this value to document changes in your template to ensure you're deploying the right template.
<b>apiProfile</b>	An optional section that defines a collection of API versions for resource types. You can use this value to specify API versions for each resource in the template.
<b>parameters</b>	An optional section where you define values that are provided during deployment. These values can be provided in a parameter file, by command-line parameters, or in the Azure portal.
<b>variables</b>	An optional section where you define values that are used to simplify template language expressions.
<b>functions</b>	An optional section where you can define <a href="#">user-defined functions</a> that are available within the template. Using functions can simplify your template when complicated expressions are used repeatedly in your template.
<b>resources</b>	A required section that defines the actual items you want to deploy or update in a resource group or a subscription.
<b>output</b>	An optional section where you specify the values that will be returned at the end of the deployment.

# Deploy an ARM template to Azure

You can deploy an ARM template to Azure in one of the following ways:

- Deploy a local template.
- Deploy a linked template.
- Deploy in a continuous deployment pipeline.

This module focuses on deploying a local ARM template. In future Learn modules, you'll learn how to deploy more complicated infrastructure and how to integrate with Azure Pipelines.

To deploy a local template, you need to have either [Azure PowerShell](#) or the [Azure CLI](#) installed locally.

First, sign in to Azure by using the Azure CLI or Azure PowerShell.

- [Azure CLI](#)
- [PowerShell](#)

Azure CLICopy

```
az login
```

Next, define your resource group. You can use an already-defined resource group or create a new one with the following command. You can obtain available location values from: `az account list-locations` (CLI) or `Get-AzLocation` (PowerShell). You can configure the default location using `az configure --defaults location=<location>`.

- [Azure CLI](#)
- [PowerShell](#)

Azure CLICopy

```
az group create \
  --name {name of your resource group} \
  --location "{location}"
```

To start a template deployment at the resource group, use either the Azure CLI command [az deployment group create](#) or the Azure PowerShell command [New-AzResourceGroupDeployment](#).

## Tip

The difference between `az deployment group create` and `az group deployment create` is that `az group deployment create` is an old command to be deprecated and will be replaced by `az deployment group create`. Therefore, it is recommended to use `az deployment group create` to deploy resources under the resource group scope.

Both commands require the resource group, the region, and the name for the deployment so you can easily identify it in the deployment history. For convenience, the exercises create a variable that stores the path to the template file. This variable makes it easier for you to run deployment commands because you don't have to retype the path every time you deploy. Here's an example:

- [Azure CLI](#)
- [PowerShell](#)

To run this deployment command, you must have the [latest version](#) of Azure CLI.  
Azure CLICopy

```
templateFile="{provide-the-path-to-the-template-file}"
az deployment group create \
  --name blanktemplate \
  --resource-group myResourceGroup \
  --template-file $templateFile
```

Use linked templates to deploy complex solutions. You can break a template into many templates and deploy these templates through a main template. When you deploy the main template, it triggers the deployment of the linked template. You can store and secure the linked template by using a SAS token.

A CI/CD pipeline automates the creation and deployment of development projects, which includes ARM template projects. The two most common pipelines used for template deployment are [Azure Pipelines](#) or [GitHub Actions](#).

More information on these two types of deployment is covered in other modules.


## Add resources to the template

To add a resource to your template, you'll need to know the resource provider and its types of resources. The syntax for this combination is in the form of `{resource-provider}/{resource-type}`. For example, to add a storage account resource to your template, you'll need the *Microsoft.Storage* resource provider. One of the types for this provider is *storageAccount*. So your resource type will be displayed

as `Microsoft.Storage/storageAccounts`. You can use a list of [resource providers for Azure services](#) to find the providers you need.

After you've defined the provider and resource type, you need to understand the properties for each resource type you want to use. For details, see [Define resources in Azure Resource Manager templates](#). View the list in the left column to find the resource. Notice that the properties are sorted by API version.

Azure / Azure Templates

 Filter by title

- > SQL
- > SQL Virtual Machine
- ▼ Storage
  - All resources
  - ▼ 2019-06-01
    - Storage Accounts**
    - > Storage Accounts/
  - > 2019-04-01
  - > 2018-11-01

## Microsoft.Storage storageAccounts template reference

03/05/2020 • 6 minutes to read

API Versions: 2019-06-01 ▼

### Template format

To create a `Microsoft.Storage/storageAccounts` resource, add the following JSON to the resources section of your template.

Here's an example of some of the listed properties from the Storage Accounts page:

## Microsoft.Storage/storageAccounts object

Name	Type	Required	Value
name	string	Yes	The name of the storage account within the specified resource group. Storage account names must be between 3 and 24 characters in length and use numbers and lower-case letters only.
type	enum	Yes	Microsoft.Storage/storageAccounts
apiVersion	enum	Yes	2019-06-01
sku	object	Yes	Required. Gets or sets the SKU name. - <a href="#">Sku object</a>
kind	enum	Yes	Required. Indicates the type of storage account. - Storage, StorageV2, BlobStorage, FileStorage, BlockBlobStorage
location	string	Yes	Required. Gets or sets the location of the resource. This will be one of the supported and registered Azure Geo Regions (e.g. West US, East US, Southeast Asia, etc.). The geo region of a resource cannot be changed once it is created, but if an identical geo region is specified on update, the request will succeed.
tags	object	No	Gets or sets a list of key value pairs that describe the resource. These tags can be used for viewing and grouping this resource (across resource groups). A maximum of 15 tags can be provided for a resource. Each tag must have a key with a length no greater than 128 characters and a value with a length no greater than 256 characters.
identity	object	No	The identity of the resource. - <a href="#">Identity object</a>
properties	object	Yes	The parameters used to create the storage account. - <a href="#">StorageAccountPropertiesCreateParameters object</a>
resources	array	No	<a href="#">encryptionScopes</a> <a href="#">privateEndpointConnections</a> <a href="#">managementPolicies</a>

For our storage example, your template might look like this:

JSONCopy

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.1",
  "apiProfile": "",
  "parameters": {},
  "variables": {},
```



```
"functions": [],
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-06-01",
    "name": "learntemplatestorage123",
    "location": "westus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "StorageV2",
    "properties": {
      "supportsHttpsTrafficOnly": true
    }
  }
],
"outputs": {}
}
```

---

## Next unit: Exercise - Create and deploy an Azure Resource Manager template

# Exercise - Create and deploy an Azure Resource Manager template

Completed 100 XP

- 10 minutes

Sandbox activated! Time remaining:

3 hr 20 min

You have used 3 of 10 sandboxes for today. More sandboxes will be available tomorrow.

Choose your Azure shell

 PowerShell  Azure CLI

### Note

The first time you activate a sandbox and accept the terms, your Microsoft account is associated with a new Azure directory named Microsoft Learn Sandbox. You're also added to a special subscription named Concierge Subscription.

In this exercise, you create an Azure Resource Manager template (ARM template), deploy it to Azure, and then update that ARM template to add parameters and outputs.

This exercise uses [Azure Resource Manager Tools for Visual Studio Code](#). Be sure to install this extension in Visual Studio Code before starting the exercise.

## Create an ARM template

1. Open Visual Studio Code, and create a new file called *azuredeploy.json*.
2. The Visual Studio Code ARM template extension comes configured with snippets to help you develop templates. Let's start by adding a blank template. On the first line of the file, enter *arm*.
3. The VS Code automatically displays several potential choices that start with **arm!**. Select the **Azure Resource Manager (ARM) template**. VS Code automatically processes the schemas and languages for your template.

Your file now looks like this:

JSONCopy

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {},
  "functions": [],
  "variables": {},
  "resources": [],
  "outputs": {}
}
```

Notice that this file has all of the sections of an ARM template that we described in the previous unit.

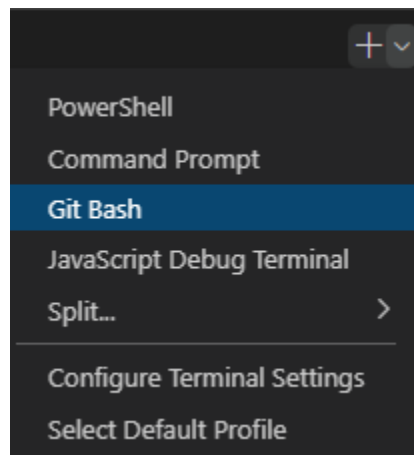
4. Save the changes to the file by pressing **Ctrl+S**.

## Deploy the ARM template to Azure

To deploy this template to Azure, you need to sign in to your Azure account from the Visual Studio Code terminal. Be sure you have installed Azure PowerShell Tools from the VS Code Extensions, and sign in to the same account that activated the sandbox.

1. In the command bar, select **Terminal > New Terminal** to open a PowerShell window.
2. If the command bar of the terminal window shows **PowerShell**, you have the right shell to work from, and you can skip to the next section.

1. If not, select the down arrow and in the dropdown list select PowerShell. If that option is missing, then select **Select Default Profile**.
2. In the input field, scroll down and select **PowerShell**.



3. Select **Terminal > New Terminal** to open a PowerShell terminal window.

Sign in to Azure by using Azure PowerShell

1. From the terminal in Visual Studio Code, run the following command to sign in to Azure. A browser opens so you can sign in to your account.

Azure PowerShellCopy

[Connect-AzAccount](#)

**Tip**

The [Az PowerShell module](#) is the replacement of AzureRM and is the recommended version to use for interacting with Azure.

2. Sign in using the account you used to activate the sandbox. After you've signed in, VS Code lists the subscriptions associated with your account in the terminal window. If you activated the sandbox, you see a code block that contains "name": "Concierge Subscription". This is the subscription to use for the rest of the exercise.

Set the default subscription for all PowerShell commands in this session

1. Run the following command to obtain your subscription(s) and their ID(s). The subscription ID is the second column. Look for *Concierge Subscription*, and copy the value in the second column. It will look something like *cf49fbbc-217c-4eb6-9eb5-a6a6c68295a0*:

Azure PowerShellCopy

```
Get-AzSubscription
```

2. Run the following command, replacing *{Your subscription ID}* with the one you copied in the previous step to change your active subscription to the Concierge Subscription.

Azure PowerShellCopy

```
$context = Get-AzSubscription -SubscriptionId {Your subscription ID}
Set-AzContext $context
```

3. Run the following command to let the default resource group be the resource group created for you in the sandbox environment. This action lets you omit that parameter from the rest of the Azure PowerShell commands in this exercise.

Azure PowerShellCopy

```
Set-AzDefault -ResourceGroupName learn-cbe03d57-3f87-4cab-b6a9-e4d1b1f1329c
```

Deploy the template to Azure

Deploy the template to Azure by running the following commands. The ARM template doesn't have any resources yet, so you won't see resources created.

## Azure PowerShellCopy


```
$templateFile="azuredeploy.json"
$today=Get-Date -Format "MM-dd-yyyy"
$deploymentName="blanktemplate-"+$today
New-AzResourceGroupDeployment `
  -Name $deploymentName `
  -TemplateFile $templateFile
```

The top section of the preceding code sets Azure PowerShell variables, which includes the path to the deployment path and the name of the deployment. Then, the `New-AzResourceGroupDeployment` command deploys the template to Azure. Notice that the deployment name is `blanktemplate` with the date as a suffix.





When you've deployed your ARM template to Azure, go to the [Azure portal](#) and make sure you're in the sandbox subscription. To do that, select your avatar in the upper-right corner of the page. Select **Switch directory**. In the list, choose the **Microsoft Learn Sandbox** directory.

1. In the resource menu, select **Resource groups**.
2. Select the *learn-cbe03d57-3f87-4cab-b6a9-e4d1b1f1329c* resource group.
3. On the **Overview** pane, you see that one deployment succeeded.
4. Select **1 Succeeded** to see the details of the deployment.
5. Select `blanktemplate` to see what resources were deployed. In this case, it will be empty because you didn't specify any resources in the template yet.

Home > learn-83cfc1b4-5153-43f1-8273-36f615aa5d51 | Deployments >

**blanktemplate-05-Jun-2020** | Overview 

Deployment


Search (Cmd+/) <<  Delete  Cancel  Redeploy  Refresh


Overview

Inputs

Outputs

Template

 **Your deployment is complete**

 Deployment name: blanktemplate-05-Jun-2020  
Subscription: [Concierge Subscription](#)  
Resource group: [learn-83cfc1b4-5153-43f1-8273-36f615aa5d51](#)

Start time: 6/5/2020, 12:00:19 PM  
Correlation ID: 5a63dd04-65ef-4e5b-a3b6-efda59dcd02

Deployment details [\(Download\)](#)

Resource	Type	Status	Operation details
No results.			

6. Leave the page open in your browser. You'll check on deployments again.

## Add a resource to the ARM template

In the previous task, you learned how to create a blank template and deploy it. Now, you're ready to deploy an actual resource. In this task, you add an Azure storage account resource to the ARM template by using a snippet from the Azure Resource Manager Tools extension for Visual Studio Code.

1. In the *azuredeploy.json* file in Visual Studio Code, place your cursor inside the brackets in the resources block `"resources": [ ]`.
2. Enter *storage* inside the brackets. A list of related snippets appears. Select **arm-storage**.

Your file will look like this:

JSONCopy

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {},
  "functions": [],
  "variables": {},
  "resources": [
    {
      "name": "storageaccount1",
      "type": "Microsoft.Storage/storageAccounts",
```

```

    "apiVersion": "2019-06-01",
    "tags": {
      "displayName": "storageaccount1"
    },
    "location": "[resourceGroup().location]",
    "kind": "StorageV2",
    "sku": {
      "name": "Premium_LRS",
      "tier": "Premium"
    }
  },
  "outputs": {}
}

```

Values that you should edit are highlighted in the new section of your file and can be navigated by pressing the `Tab` key.

Notice the `tags` and `location` attributes are filled in. The `location` attribute uses a function to set the location of the resource to the location of the resource group. You'll learn about tags and functions in the next module.

3. Change the values of the resource *name* and *displayName* to something unique, (for example, **learnexercise12321**). This name must be unique across all of Azure, so choose something unique to you.
4. Change the value of the sku *name* from **Premium\_LRS** to **Standard\_LRS**. Change the value of *tier* to **Standard**. Notice that Visual Studio Code gives you the proper choices for your attribute values in IntelliSense. Delete the default value including the quotation marks, and enter quotation marks to see this work.
5. The location of the resource is set to the location of the resource group where it will be deployed. Leave the default here.
6. Save the file.

### Deploy the updated ARM template

Here, you change the name of the deployment to better reflect what this deployment does.

Run the following Azure PowerShell commands in the terminal. This snippet is the same code you used previously, but the name of the deployment is changed.

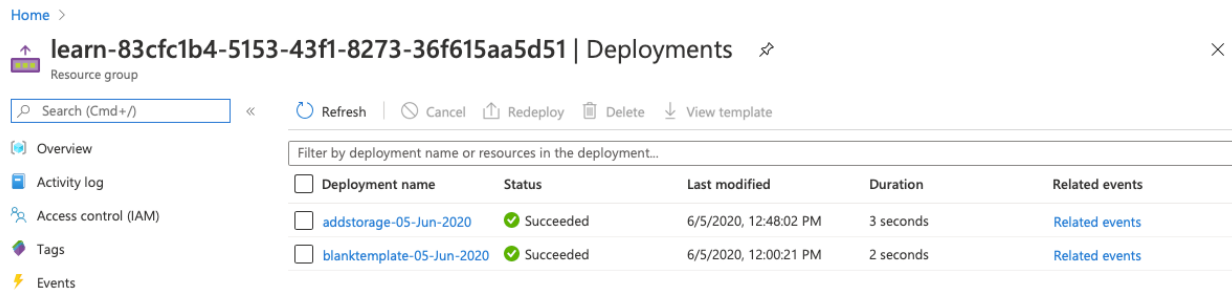
## Azure PowerShellCopy

```
$templateFile="azuredeploy.json"
$today=Get-Date -Format "MM-dd-yyyy"
$deploymentName="addstorage-"+$today
New-AzResourceGroupDeployment `
  -Name $deploymentName `
  -TemplateFile $templateFile
```

## Check your deployment

1. In your browser, go back to the Azure portal. Go to your resource group, and you'll see that there are now **2 Succeeded** deployments. Select this link.

Notice that both deployments are in the list.



The screenshot shows the Azure portal interface for a resource group named 'learn-83cfc1b4-5153-43f1-8273-36f615aa5d51'. The 'Deployments' tab is selected, showing a list of two successful deployments. The table has columns for Deployment name, Status, Last modified, Duration, and Related events.

Deployment name	Status	Last modified	Duration	Related events
addstorage-05-Jun-2020	Succeeded	6/5/2020, 12:48:02 PM	3 seconds	<a href="#">Related events</a>
blanktemplate-05-Jun-2020	Succeeded	6/5/2020, 12:00:21 PM	2 seconds	<a href="#">Related events</a>

2. Select **\*\*addstorage\*\***.

Notice that the storage account has been deployed.

---

## Next unit: Add flexibility to your Azure Resource Manager template by using parameters and outputs

# Add flexibility to your Azure Resource Manager template by using parameters and outputs

Completed 100 XP



- 8 minutes

In the last unit, you created an Azure Resource Manager (ARM) template and added an Azure storage account to the ARM template. You might have noticed that there's a problem with your template. The storage account name is hardcoded. You can only use this template to deploy the same storage account every time. To deploy a storage account with a different name, you would have to create a new template, which isn't a practical way to automate your deployments. The storage account SKU is also hardcoded, which means you can't vary the type of storage account for different environments. Recall that in our scenario each deployment might have a different type of storage account. You can make your template more reusable by adding a parameter for the storage account SKU.

In this unit, you learn about the *parameters* and *outputs* sections of the template.

## ARM template parameters

ARM template parameters enable you to customize the deployment by providing values that are tailored for a particular environment. For example, you pass in different values based on whether you're deploying to an environment for development, test, production, or others. For example, the previous template uses the *Standard\_LRS* storage account SKU. You can reuse this template for other deployments that create a storage account by making the name of the storage account SKU a parameter. Then, you pass in the name of the SKU you want for this particular deployment when the template is deployed. You can do this step either at the command line or by using a parameter file.

In the *parameters* section of the template, you specify which values you can input when you deploy the resources. You're limited to 256 parameters in a template. Parameter definitions can use most template functions.

The available properties for a parameter are:

JSONCopy

```
"parameters": {
  "<parameter-name>": {
    "type": "<type-of-parameter-value>",
    "defaultValue": "<default-value-of-parameter>",
    "allowedValues": [
      "<array-of-allowed-values>"
    ],
    "minValue": <minimum-value-for-int>,
  }
}
```

```

    "maxValue": <maximum-value-for-int>,
    "minLength": <minimum-length-for-string-or-array>,
    "maxLength": <maximum-length-for-string-or-array-parameters>,
    "metadata": {
        "description": "<description-of-the-parameter>"
    }
}
}
}

```

The allowed types of parameters are:

- string
- secureString
- integers
- boolean
- object
- secureObject
- array

## Recommendations for using parameters

Use parameters for settings that vary according to the environment; for example, SKU, size, or capacity. Also use parameters for resource names that you want to specify yourself for easy identification or to comply with internal naming conventions. Provide a description for each parameter, and use default values whenever possible.

For security reasons, never hard code or provide default values for usernames and/or passwords in templates. Always use parameters for usernames and passwords (or secrets). Use *secureString* for all passwords and secrets. If you pass sensitive data in a JSON object, use the *secureObject* type. Template parameters with *secureString* or *secureObject* types can't be read or harvested after the deployment of the resource.

## Use parameters in an ARM template

In the parameters section of the ARM template, specify the parameters that can be input when you deploy the resources. You're limited to 256 parameters in a template.

Here's an example of a template file with a parameter for the storage account SKU defined in the parameters section of the template. You can provide a default for the parameter to be used if no value is specified at execution.

JSONCopy

```

"parameters": {
  "storageAccountType": {
    "type": "string",
    "defaultValue": "Standard_LRS",
    "allowedValues": [
      "Standard_LRS",
      "Standard_GRS",
      "Standard_ZRS",
      "Premium_LRS"
    ],
    "metadata": {
      "description": "Storage Account type"
    }
  }
}
}

```

Then, use the parameter in the resource definition. The syntax is [parameters('name of the parameter')]. You use the parameters function. You learn more about functions in the next module.

JSONCopy

```

"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-04-01",
    "name": "learntemplatestorage123",
    "location": "[resourceGroup().location]",
    "sku": {
      "name": "[parameters('storageAccountType')]"
    },
    "kind": "StorageV2",
    "properties": {
      "supportsHttpsTrafficOnly": true
    }
  }
]

```

When you deploy the template, you can give a value for the parameter. Notice the last line in the following command:

- [Azure CLI](#)
- [PowerShell](#)

Azure CLICopy

```

templateFile="azuredeploy.json"
az deployment group create \
  --name testdeployment1 \
  --template-file $templateFile \
  --parameters storageAccountType=Standard_LRS

```

# ARM template outputs

In the outputs section of your ARM template, you can specify values that will be returned after a successful deployment. Here are the elements that make up the outputs section.

JSONCopy

```
"outputs": {
  "<output-name>": {
    "condition": "<boolean-value-whether-to-output-value>",
    "type": "<type-of-output-value>",
    "value": "<output-value-expression>",
    "copy": {
      "count": <number-of-iterations>,
      "input": <values-for-the-variable>
    }
  }
}
```

Element	Description
---------	-------------

<b>output-name</b>	Must be a valid JavaScript identifier.
--------------------	--

<b>condition</b>	(Optional) A Boolean value that indicates whether this output value is returned. When true, the value is included in the deployment. When false, the output value is skipped for this deployment. When not specified, the default is true.
------------------	--

<b>type</b>	The type of the output value.
-------------	-------------------------------

<b>value</b>	(Optional) A template language expression that's evaluated and returned as an output value.
--------------	---

<b>copy</b>	(Optional) Copy is used to return more than one value for an output.
-------------	--

Use outputs in an ARM template

Here's an example to output the storage account's endpoints.

JSONCopy

```
"outputs": {
  "storageEndpoint": {
    "type": "object",
    "value": "[reference('learntemplatestorage123').primaryEndpoints]"
  }
}
```

Notice the reference part of the expression. This function gets the runtime state of the storage account.

## Deploy an ARM template again

Recall that ARM templates are idempotent, which means you can deploy the template to the same environment again and if nothing was changed in the template, nothing will change in the environment. If a change was made to the template, for example, you changed a parameter value, only that change will be deployed. Your template can contain all of the resources you need for your Azure solution, and you can safely execute a template again. Resources will be created only if they didn't already exist and updated only if there's a change.

---

## Next unit: Exercise - Add parameters and outputs to your Azure Resource Manager template

# Exercise - Add parameters and outputs to your Azure Resource Manager template

Completed 100 XP

- 12 minutes

This module requires a sandbox to complete. You have used 2 of 10 sandboxes for today. More sandboxes will be available tomorrow.

Activate sandbox

Choose your Azure shell



PowerShell



Azure CLI

In this exercise, you add a parameter to define the Azure storage account name during deployment. You then add a parameter to define what storage account SKU is allowed and define which one to use for this deployment. You also add usefulness to the Azure Resource Manager template (ARM template) by adding an output that can be used later in the deployment process.

# Create parameters for the ARM template

Here, you make your ARM template more flexible by adding parameters that can be set at runtime. Create a parameter for the `storageName` value.

1. In the `azuredeploy.json` file in Visual Studio Code, place your cursor inside the braces in the `parameters` attribute. `"parameters":{}`,
2. Select Enter, and then enter **par**. You see a list of related snippets. Choose **new-parameter**. It adds a generic parameter to the template. It will look like this:

JSONCopy

```
"parameters": {  
  "parameter1": {  
    "type": "string",  
    "metadata": {  
      "description": "description"  
    }  
  }  
},
```

3. Change the parameter to be called **storageName**, and leave the type as a string. Add a **minLength** value of **3** and a **maxLength** value of **24**. Add a description value of **The name of the Azure storage resource**.
4. The parameter block should look like this:

JSONCopy

```
"parameters": {  
  "storageName": {  
    "type": "string",  
    "minLength": 3,  
    "maxLength": 24,  
    "metadata": {  
      "description": "The name of the Azure storage resource"  
    }  
  }  
},
```

5. Use the new parameter in the resources block in both the name and `displayName` values. The entire file will look like this:

JSONCopy

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

```

"contentVersion": "1.0.0.0",
"parameters": {
  "storageName": {
    "type": "string",
    "minLength": 3,
    "maxLength": 24,
    "metadata": {
      "description": "The name of the Azure storage resource"
    }
  }
},
"functions": [],
"variables": {},
"resources": [
  {
    "name": "[parameters('storageName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-06-01",
    "tags": {
      "displayName": "[parameters('storageName')]"
    },
    "location": "[resourceGroup().location]",
    "kind": "StorageV2",
    "sku": {
      "name": "Standard_LRS",
      "tier": "Standard"
    }
  }
],
"outputs": {}
}

```

6. Save the file.

Deploy the parameterized ARM template

Here, you change the name of the deployment to better reflect what this deployment does and fill in a value for the new parameter.

Run the following Azure CLI commands in the terminal. This snippet is the same code you used previously, but the name of the deployment is changed. Fill in a unique name for the `storageName` parameter. Remember, this name must be unique across all of Azure. You can use the unique name you created in the last unit. In that case, Azure will update the resource instead of creating a new one.

Azure CLICopy

```

templateFile="azuredeploy.json"
today=$(date +%d-%b-%Y)
DeploymentName="addnameparameter-$today

```

```
az deployment group create \  
  --name $DeploymentName \  
  --template-file $templateFile \  
  --parameters storageName={your-unique-name}
```

Check your deployment

1. In your browser, go back to the Azure portal. Go to your resource group, and see that there are now **3 Succeeded** deployments. Select this link.

Notice that all three deployments are in the list.

2. Explore the *addnameparameter* deployment as you did previously.

Add another parameter to limit allowed values

Here, you use parameters to limit the values allowed for a parameter.

1. Place your cursor after the closing brace for the *storageNameparameter*. Add a comma, and select Enter.
2. Again, enter **par**, and select **new-parameter**.
3. Change the new generic parameter to the following:

JSONCopy

```
"storageSKU": {  
  "type": "string",  
  "defaultValue": "Standard_LRS",  
  "allowedValues": [  
    "Standard_LRS",  
    "Standard_GRS",  
    "Standard_RAGRS",  
    "Standard_ZRS",  
    "Premium_LRS",  
    "Premium_ZRS",  
    "Standard_GZRS",  
    "Standard_RAGZRS"  
  ]  
}
```

Here, you're listing the values that this parameter will allow. If the template runs with a value that isn't allowed, the deployment will fail.

4. Add a comment to this parameter.



```
// This is the allowed values for an Azure storage account
"storageSKU": {
  "type": "string",
  "defaultValue": "Standard_LRS",
  "allowedValues": [
    "Standard_LRS",
    "Standard_GRS",
    "Standard_RAGRS",
    "Standard_ZRS",
    "Premium_LRS",
    "Premium_ZRS",
    "Standard_GZRS",
    "Standard_RAGZRS"
  ]
},
```

ARM templates support `//` and `/* */` comments.

5. Update **resources** to use the `storageSKU` parameter. Take advantage of IntelliSense in Visual Studio Code to make this step easier.

JSONCopy

```
"sku": {
  "name": "[parameters('storageSKU')]"
}
```

The entire file will look like this:

JSONCopy

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageName": {
      "type": "string",
      "minLength": 3,
      "maxLength": 24,
      "metadata": {
        "description": "The name of the Azure storage resource"
      }
    },
    "storageSKU": {
```

```

        "type": "string",
        "defaultValue": "Standard_LRS",
        "allowedValues": [
            "Standard_LRS",
            "Standard_GRS",
            "Standard_RAGRS",
            "Standard_ZRS",
            "Premium_LRS",
            "Premium_ZRS",
            "Standard_GZRS",
            "Standard_RAGZRS"
        ]
    },
    "functions": [],
    "variables": {},
    "resources": [
        {
            "name": "[parameters('storageName')]",
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2019-06-01",
            "tags": {
                "displayName": "[parameters('storageName')]"
            },
            "location": "[resourceGroup().location]",
            "kind": "StorageV2",
            "sku": {
                "name": "[parameters('storageSKU')]",
                "tier": "Standard"
            }
        }
    ],
    "outputs": {}
}

```

6. Save the file.

## Deploy the ARM template

Here, you'll deploy successfully by using a `storageSKU` parameter that's in the allowed list. Then, you'll try to deploy the template by using a `storageSKU` parameter that isn't in the allowed list. The second deployment will fail as expected.

1. Run the following commands to deploy the template. Fill in a unique name for the `storageName` parameter. Remember, this name must be unique across all of Azure. You can use the unique name you created in the last section. In that case, Azure will update the resource instead of creating a new one.

Azure CLICopy

```

templateFile="azuredeploy.json"
today=$(date +%d-%b-%Y)
DeploymentName="addSkuParameter-$today

az deployment group create \
  --name $DeploymentName \
  --template-file $templateFile \
  --parameters storageSKU=Standard_GRS storageName={your-unique-name}

```

Allow this deployment to finish. This deployment succeeds as expected. The allowed values prevent users of your template from passing in parameter values that don't work for the resource. Let's see what happens when you provide an invalid SKU.

2. Run the following commands to deploy the template with a parameter that isn't allowed. Here, you changed the `storageSKU` parameter to **Basic**. Fill in a unique name for the `storageName` parameter. Remember, this name must be unique across all of Azure. You can use the unique name you created in the last section. In that case, Azure will update the resource instead of creating a new one.

Azure CLICopy

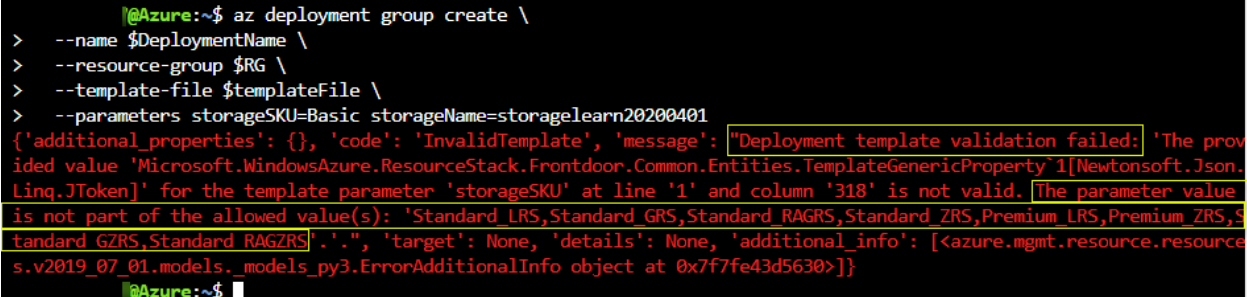
```

templateFile="azuredeploy.json"
today=$(date +%d-%b-%Y)
DeploymentName="addSkuParameter-$today

az deployment group create \
  --name $DeploymentName \
  --template-file $templateFile \
  --parameters storageSKU=Basic storageName={your-unique-name}

```

This deployment fails. Notice the error.



```

@Azure:~$ az deployment group create \
> --name $DeploymentName \
> --resource-group $RG \
> --template-file $templateFile \
> --parameters storageSKU=Basic storageName=storagelearn20200401
{'additional_properties': {}, 'code': 'InvalidTemplate', 'message': "Deployment template validation failed: 'The provided value 'Microsoft.WindowsAzure.ResourceStack.Frontdoor.Common.Entities.TemplateGenericProperty'1[Newtonsoft.Json.Linq.JToken]' for the template parameter 'storageSKU' at line '1' and column '318' is not valid. The parameter value is not part of the allowed value(s): 'Standard_LRS,Standard_GRS,Standard_RAGRS,Standard_ZRS,Premium_LRS,Premium_ZRS,Standard_GZRS,Standard_RAGZRS'.'. ", 'target': None, 'details': None, 'additional_info': [<azure.mgmt.resource.resource.s.v2019_07_01.models.models_py3.ErrorAdditionalInfo object at 0x7f7fe43d5630>]}
@Azure:~$

```

## Add output to the ARM template

Here, you add to the `outputs` section of the ARM template to output the endpoints for the storage account resource.

1. In the `azuredeploy.json` file in Visual Studio Code, place your cursor inside the braces in the `outputs` attribute `"outputs":{}`.
2. Select `Enter`, and then enter `out`. You see a list of related snippets. Select **new-output**. It adds a generic output to the template. It will look like this:

JSONCopy

```
"outputs": {  
  "output1": {  
    "type": "string",  
    "value": "value"  
  }  
}
```

3. Change **"output1"** to **"storageEndpoint"**, then change the value of type to **"object"**, and finally, change the value of value to **"[reference(parameters('storageName')).primaryEndpoints]"**. This expression is the one we described in the previous unit that gets the endpoint data. Because we specified *object* as the type, it will return the object in JSON format.

JSONCopy

```
"outputs": {  
  "storageEndpoint": {  
    "type": "object",  
    "value": "[reference(parameters('storageName')).primaryEndpoints]"  
  }  
}
```

4. Save the file.

Deploy the ARM template with an output

Here, you deploy the template and see the endpoints output as JSON. You need to fill in a unique name for the `storageName` parameter. Remember, this name must be unique across all of Azure. You can use the unique name you created in the last section. In that case, Azure will update the resource instead of creating a new one.

1. Run the following commands to deploy the template. Be sure to replace `{your-unique-name}` with a string unique to you.

Azure CLICopy

```
templateFile="azuredeploy.json"
today=$(date +%d-%b-%Y)
DeploymentName="addoutputs-"$today

az deployment group create \
  --name $DeploymentName \
  --template-file $templateFile \
  --parameters storageSKU=Standard_LRS storageName={your-unique-name}
```

Notice the output.

```
],
  "outputs": {
    "storageEndpoint": {
      "type": "Object",
      "value": {
        "blob": "https://storelearnv1mlemv4t3u7i.blob.core.windows.net/",
        "dfs": "https://storelearnv1mlemv4t3u7i.dfs.core.windows.net/",
        "file": "https://storelearnv1mlemv4t3u7i.file.core.windows.net/",
        "queue": "https://storelearnv1mlemv4t3u7i.queue.core.windows.net/",
        "table": "https://storelearnv1mlemv4t3u7i.table.core.windows.net/",
        "web": "https://storelearnv1mlemv4t3u7i.z22.web.core.windows.net/"
      }
    }
  }
}
```

Check your output deployment

In the Azure portal, go to your *addOutputs* deployment. You can find your output there as well.



## Next unit: Knowledge check

# Summary

Completed 100 XP

- 1 minute

In this module, you were introduced to Azure Resource Manager templates (ARM templates) and used them to deploy a storage account to Azure. You made the template more flexible by adding parameters and got output from the execution of the template.

In summary, you:

- Implemented an ARM template by using Visual Studio Code.
- Declared resources and added flexibility to your template by adding resources, parameters, and outputs.

## Learn more

To learn more, see the following articles:

- [Azure Resource Manager Tools for Visual Studio Code](#)
  - [Understand the structure and syntax of ARM templates](#)
  - [Azure Resource Manager](#)
  - [Azure CLI](#)
  - [Azure PowerShell](#)
  - [Resource providers for Azure services](#)
  - [Define resources in Azure Resource Manager templates](#)
  - [Outputs in Azure Resource Manager templates](#)
  - [Parameters in Azure Resource Manager templates](#)
- 

## Explore other modules

- [Architect compute infrastructure in Azure](#)
- [Deploy and manage resources in Azure by using JSON ARM templates](#)
- [AZ-104: Prerequisites for Azure administrators](#)
- [Deploy and manage compute resources for Azure administrators](#)
- [Administer infrastructure resources in Azure](#)

