100 XP

Introduction

3 minutes

Azure Resource Manager is the deployment and management service for Azure. It provides a management layer that enables you to create, update, and delete resources in your Azure subscription.

After completing this module, you'll be able to:

- Describe what role Azure Resource Manager has in Azure and the benefits of using Azure Resource Manager templates
- Explain what happens when Azure Resource Manager templates are deployed and how to structure them to support your solution
- Create a template with conditional resource deployments
- Choose the correct deployment mode for your solution
- Create and deploy an Azure Resource Manager template by using Visual Studio Code

Next unit: Explore Azure Resource Manager

Continue

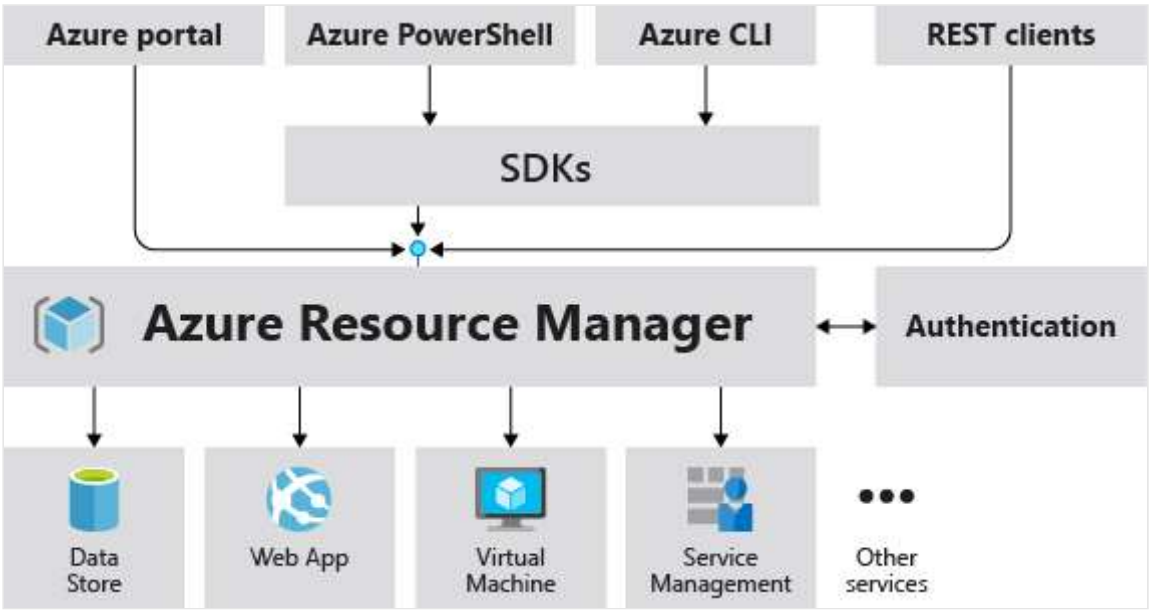
100 XP

Explore Azure Resource Manager

3 minutes

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

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



Why choose Azure Resource Manager templates?

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

- **Declarative syntax**: Azure Resource Manager templates allow you to create and deploy an entire Azure infrastructure declaratively. For example, you can deploy not only virtual machines, but also the network infrastructure, storage systems, and any other resources you may need.

- **Repeatable results**: Repeatedly deploy your infrastructure throughout the development lifecycle and have confidence your resources are deployed in a consistent manner. Templates are idempotent, which means you can deploy the same template many times and get the same resource types in the same state. You can develop one template that represents the desired state, rather than developing lots of separate templates to represent updates.

- **Orchestration**: You don't have to worry about the complexities of ordering operations. Resource Manager orchestrates the deployment of interdependent resources so they're created in the correct order. When possible, Resource Manager deploys resources in parallel so your deployments finish faster than serial deployments. You deploy the template through one command, rather than through multiple imperative commands.

Template file

Within your template, you can write template expressions that extend the capabilities of JSON. These expressions make use of the functions provided by Resource Manager.

The template has the following sections:

- Parameters - Provide values during deployment that allow the same template to be used with different environments.
- Variables - Define values that are reused in your templates. They can be constructed from parameter values.
- User-defined functions - Create customized functions that simplify your template.
- Resources - Specify the resources to deploy.
- Outputs - Return values from the deployed resources.

Next unit: Deploy multi-tiered solutions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Deploy multi-tiered solutions

3 minutes

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

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

JSON Copy

```
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-04-01",
    "name": "mystorageaccount",
    "location": "westus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "StorageV2",
    "properties": {}
  }
]
```

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

HTTP Copy

```
PUT
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceG
roupName}/providers/Microsoft.Storage/storageAccounts/mystorageaccount?api-
version=2019-04-01
REQUEST BODY
{
  "location": "westus",
```

```
  "sku": {
    "name": "Standard_LRS"
  },
  "kind": "StorageV2",
  "properties": {}
}
```

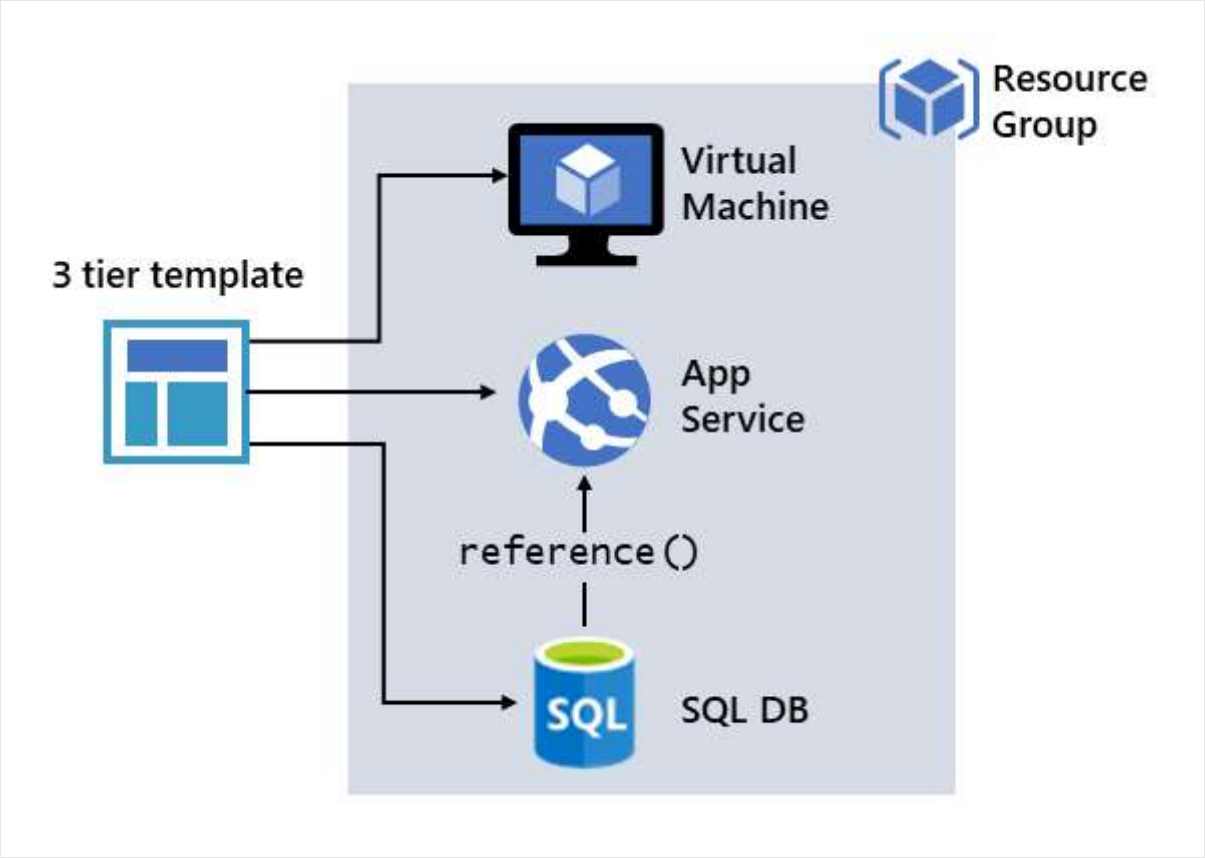
Notice that the **apiVersion** you set in the template for the resource is used as the API version for the REST operation. You can repeatedly deploy the template and have confidence it will continue to work. By using the same API version, you don't have to worry about breaking changes that might be introduced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

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

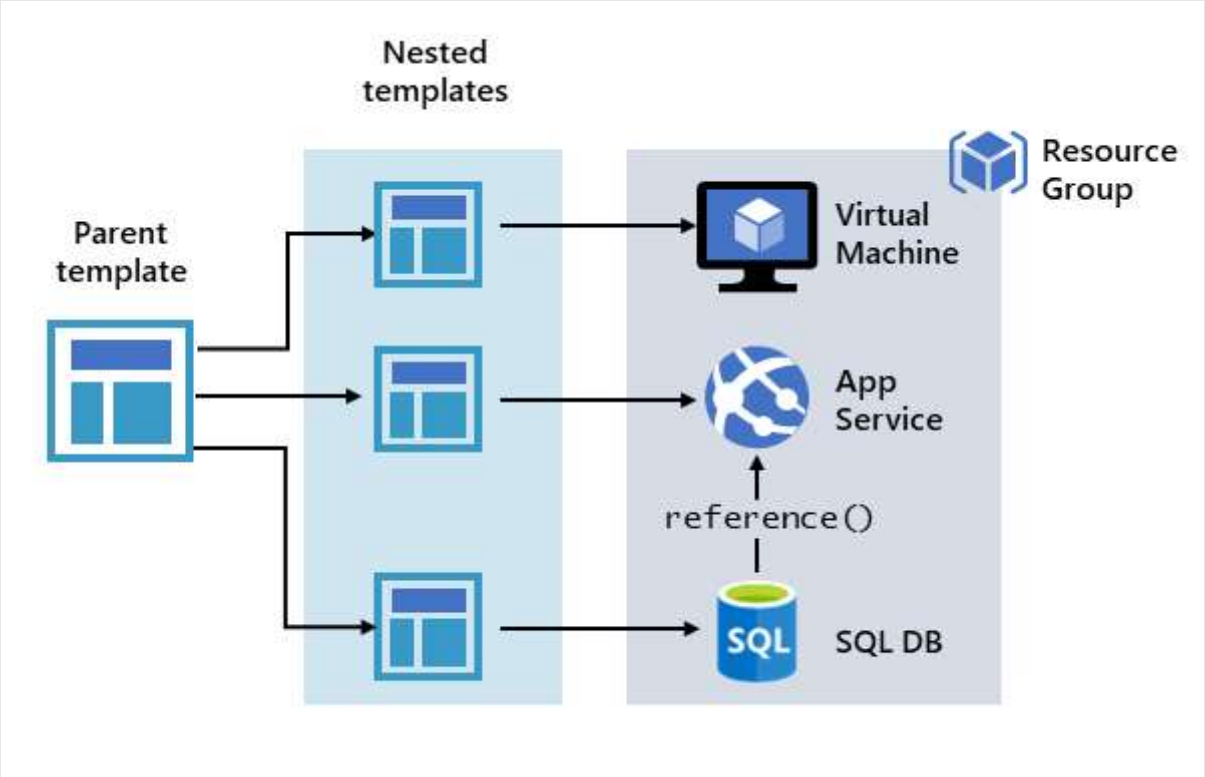
- Azure portal
- Azure CLI
- PowerShell
- REST API
- Button in GitHub repository
- Azure Cloud Shell

Defining multi-tiered templates

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



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



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

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

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

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

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

Share templates

After creating your template, you may wish to share it with other users in your organization. Template specs enable you to store a template as a resource type. You use role-based access control to manage access to the template spec. Users with read access to the template spec can deploy it, but not change the template.

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

Next unit: Explore conditional deployment

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore conditional deployment

3 minutes

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

> **Note**
>
> Conditional deployment doesn't cascade to **child resources**. If you want to conditionally deploy a resource and its child resources, you must apply the same condition to each resource type.

New or existing resource

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

JSON Copy

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-
01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
      "type": "string"
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]"
```

```
    },
    "newOrExisting": {
      "type": "string",
      "defaultValue": "new",
      "allowedValues": [
        "new",
        "existing"
      ]
    }
  },
  "functions": [],
  "resources": [
    {
      "condition": "[equals(parameters('newOrExisting'), 'new')]",
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-06-01",
      "name": "[parameters('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "Standard_LRS",
        "tier": "Standard"
      },
      "kind": "StorageV2",
      "properties": {
        "accessTier": "Hot"
      }
    }
  ]
}
```

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

Runtime functions

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

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

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

removes it from the required dependencies.

Additional resources

- Azure Resource Manager template functions

Next unit: Set the correct deployment mode

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Set the correct deployment mode

3 minutes

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

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

Complete mode

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

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

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

Incremental mode

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

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

Example result

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

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

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

Set deployment mode

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

PowerShell Copy

```
New-AzResourceGroupDeployment `
  -Mode Complete `
  -Name ExampleDeployment `
  -ResourceGroupName ExampleResourceGroup `
  -TemplateFile c:\MyTemplates\storage.json
```

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

Azure CLI Copy

```
az deployment group create \
  --mode Complete \
  --name ExampleDeployment \
```

```
    --resource-group ExampleResourceGroup \
    --template-file storage.json
```

Next unit: Exercise: Create and deploy Azure Resource Manager templates by using Visual Studio Code

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Exercise: Create and deploy Azure Resource Manager templates by using Visual Studio Code

10 minutes

In this exercise you will learn how to use Visual Studio Code, and the Azure Resource Manager Tools extension, to create and edit Azure Resource Manager templates.

- Create an Azure Resource Manager template
- Add an Azure resource to the template
- Add parameters to the template
- Create a parameter file
- Deploy the template
- Clean up resources

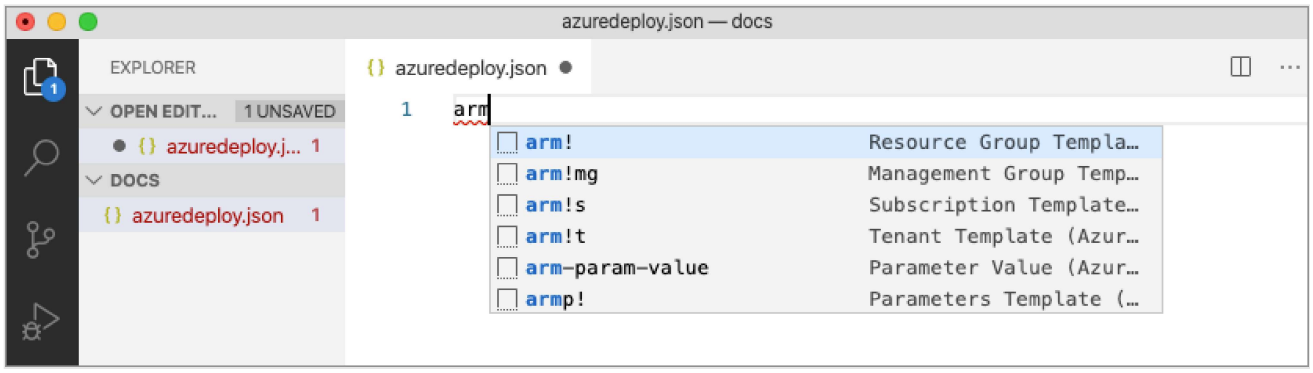
Prerequisites

- An **Azure account** with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free .
- Visual Studio Code with the Azure Resource Manager Tools installed.
- Azure CLI installed locally

Create an Azure Resource Manager template

1. Create and open a new file named *azuredeploy.json* with Visual Studio Code.

2. Enter **arm** in the *azuredeploy.json* file and select **arm!** from the autocomplete options. This will insert a snippet with the basic building blocks for an Azure resource group deployment.



Your file should contain something similar to the example below.

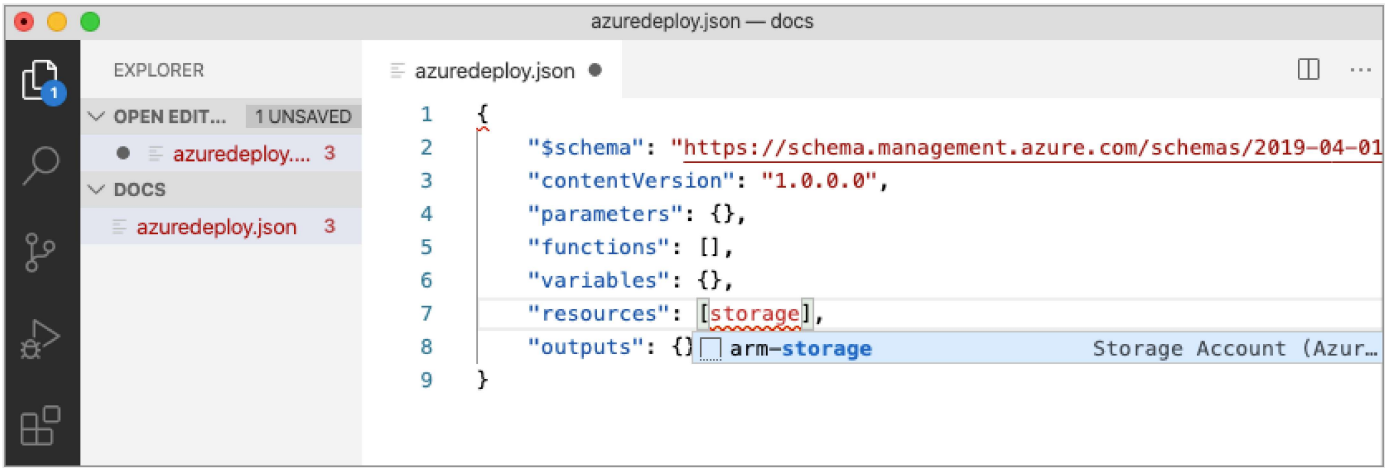
JSON Copy

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

Add an Azure resource to the template

In this section you will add a snippet to support the creation of an Azure storage account to the template.

Place the cursor in the template `resources` block, type in `storage`, and select the **arm-storage** snippet.



The `resources` block should look similar to the example below.

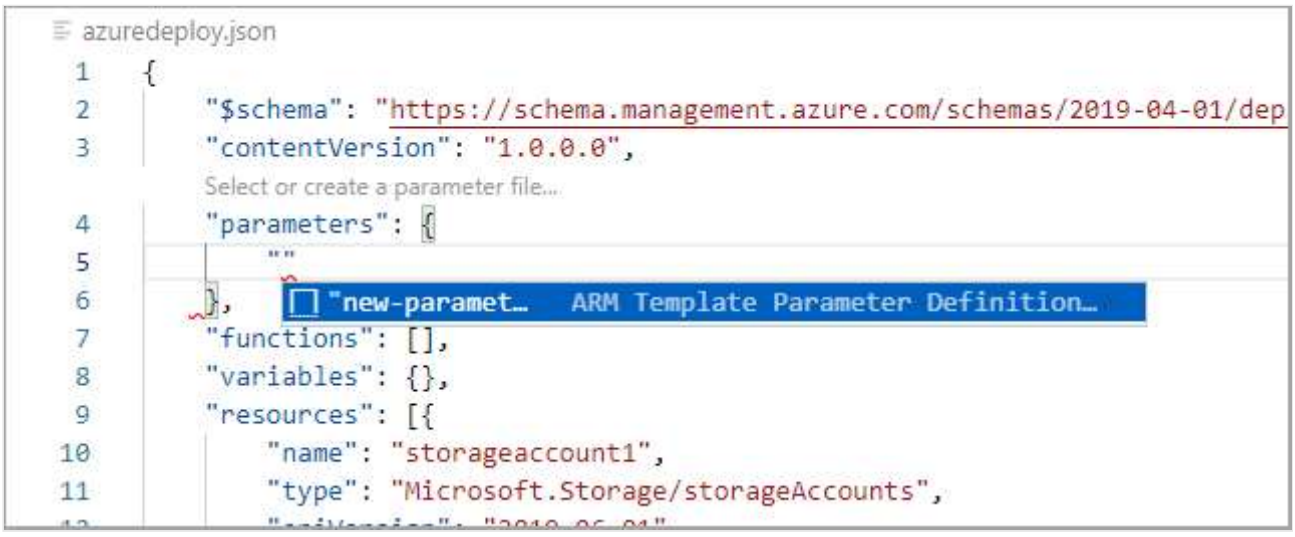
JSON Copy

```
"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"
    }
}],
```

Add parameters to the template

Now you will create and use a parameter to specify the storage account name.

Place your cursor in the parameters block, add a carriage return, type `"`, and then select the `new-parameter` snippet. This action adds a generic parameter to the template.



Make the following changes to the new parameter you just added:

1. Update the name of the parameter to `storageAccountName` and the description to `Storage Account Name.`

2. Azure storage account names have a minimum length of 3 characters and a maximum of 24. Add both `minLength` and `maxLength` to the parameter and provide appropriate values.

The `parameters` block should look similar to the example below.

JSON Copy

```
"parameters": {
    "storageAccountName": {
        "type": "string",
        "metadata": {
            "description": "Storage Account Name"
        },
        "minLength": 3,
        "maxLength": 24
    }
},
```

Follow the steps below to update the name property of the storage resource to use the parameter.

1. In the `resources` block, delete the current default name which is `storageaccount1` in the examples above. Leave the quotes (`""`) around the name in place.

2. Enter a square bracket `[`, which produces a list of Azure Resource Manager template functions. Select **parameters** from the list.

3. Add `()` at the end of **parameters** and select **storageAccountName** from the pop-up. If the list of parameters does not show up automatically you can enter a single quote `'` inside of the round brackets to display the list.

The resources block of the template should now be similar to the example below.

JSON Copy

```
"resources": [{
    "name": "[parameters('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-06-01",
    "tags": {
        "displayName": "storageaccount1"
    },
    "location": "[resourceGroup().location]",
    "kind": "StorageV2",
    "sku": {
```

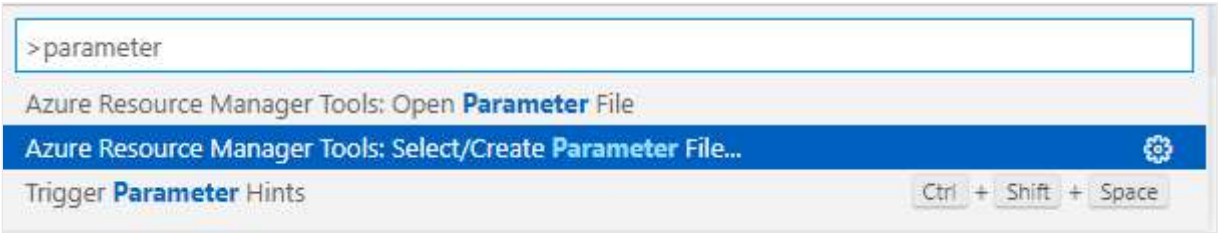
```
        "name": "Premium_LRS",
        "tier": "Premium"
    }
}],
```

Create a parameter file

An Azure Resource Manager template parameter file allows you to store environment-specific parameter values and pass these values in as a group at deployment time. This is useful if you want to have values specific to a test or production environment, for example. The extension makes it easy to create a parameter file that is mapped to your existing template. Follow the steps below to create a parameter file.

1. With the *azuredeploy.json* file in focus open the **Command Palette** by selecting **View** > **Command Palette** from the menu bar.

2. In the **Command Palette** enter "**parameter**" in the search bar and select **Azure Resource Manager Tools:Select/Create Parameter File**.



3. A new dialog box will open at the top of the editor. From those options select **New**, then select **All Parameters**. Accept the default name for the new file.

4. Edit the `value` parameter and type in a name that meets the naming requirements. The *azuredeploy.parameters.json* file should be similar to the example below.

JSON Copy

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-
01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageAccountName": {
            "value": "az204storageacctarm"
        }
    }
}
```

Deploy the template

It's time to deploy the template. Follow the steps below, in the VS Code terminal, to connect to Azure and deploy the new storage account resource.

1. Connect to Azure by using the `az login` command.

```
Bash
az login
```

2. Create a resource group to contain the new resource. Replace `<myLocation>` with a region near you.

```
Bash
az group create --name az204-arm-rg --location <myLocation>
```

3. Use the `az deployment group create` command to deploy your template. The deployment will take a few minutes to complete, progress will be shown in the terminal.

```
Bash
az  deployment group create --resource-group az204-arm-rg --template-file
azuredeploy.json --parameters azuredeploy.parameters.json
```

4. You can verify the deployment by running the command below. Replace `<myStorageAccount>` with the name you used earlier.

```
Bash
az storage account show --resource-group az204-arm-rg --name <myStorageAccount>
```

Clean up resources

When the Azure resources are no longer needed use the Azure CLI command below to delete the resource group.

Bash Copy

```
az group delete --name az204-arm-rg --no-wait
```

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

3 minutes

Check your knowledge

1. What purpose does the **outputs** section of an Azure Resource Manager template serve?

- ○ Specify the resources to deploy.
- ◉ Return values from the deployed resources

  ✔ That's correct. The "outputs" section returns values from the resource(s) that were deployed.

- ○ Define values that are reused in your templates.

2. Which Azure Resource Manager template deployment mode deletes resources in a resource group that aren't specified in the template?

- ◉ Incremental

  ✘ That's incorrect. The incremental mode only adds resources specified in an Azure Resource Manager template deployment.

- ○ Complete

  ✔ That's correct. Complete mode will delete resources not specified in an Azure Resource Manager template deployment.

- ○ Both incremental and complete delete resources

Next unit: Summary

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Summary

3 minutes

In this module, you learned how to:

- Describe what role Azure Resource Manager has in Azure and the benefits of using Azure Resource Manager templates
- Explain what happens when Azure Resource Manager templates are deployed and how to structure them to support your solution
- Create a template with conditional resource deployments
- Choose the correct deployment mode for your solution
- Create and deploy an Azure Resource Manager template by using Visual Studio Code