

Table of Contents

Overview

[What is Resource Manager?](#)

[Supported services, regions, and API versions](#)

[Understand Resource Manager and Classic deployment](#)

[Prescriptive subscription governance](#)

[Governance examples for enterprises](#)

Get started

[Export template](#)

[Create your first template](#)

[Visual Studio with Resource Manager](#)

How to

[Create templates](#)

[Best practices for templates](#)

[Template sections](#)

[Define dependency between resources](#)

[Set location](#)

[Assign tags](#)

[Set child resource name and type](#)

[Create multiple instances of resource type](#)

[Link to other templates](#)

[Share state between linked templates](#)

[Patterns for designing templates](#)

Deploy

[PowerShell](#)

[Azure CLI](#)

[Portal](#)

[REST API](#)

[Continuous integration with Visual Studio Team Services](#)

[Pass secure values during deployment](#)

Manage

[PowerShell](#)

[Azure CLI](#)

[Portal](#)

[REST API](#)

[Use tags to organize resources](#)

[Move resources to new group or subscription](#)

Control Access

[Create service principal with PowerShell](#)

[Create service principal with Azure CLI 2.0](#)

[Create service principal with Azure CLI 1.0](#)

[Create service principal with portal](#)

[Authentication API to access subscriptions](#)

[Lock resources](#)

[Security considerations](#)

Set resource policies

[What are resource policies?](#)

[Portal policy assignment](#)

[Script policy assignment](#)

[Resource tag policies](#)

[Storage policies](#)

[Linux VM policies](#)

[Windows VM policies](#)

Audit and Troubleshoot

[Troubleshoot common deployment errors](#)

[View activity logs](#)

[View deployment operations](#)

Reference

[Template functions](#)

[PowerShell](#)

[Azure 2.0 CLI](#)

[.NET](#)

[Java](#)

[Python](#)

[Template format](#)

[REST](#)

[Resources](#)

[Throttling requests](#)

[Track asynchronous operations](#)

[Stack Overflow](#)

[Videos](#)

[Service updates](#)

Azure Resource Manager overview

3/30/2017 • 16 min to read • [Edit Online](#)

The infrastructure for your application is typically made up of many components – maybe a virtual machine, storage account, and virtual network, or a web app, database, database server, and 3rd party services. You do not see these components as separate entities, instead you see them as related and interdependent parts of a single entity. You want to deploy, manage, and monitor them as a group. Azure Resource Manager enables you to work with the resources in your solution as a group. You can deploy, update, or delete all the resources for your solution in a single, coordinated operation. You use a template for deployment and that template can work for different environments such as testing, staging, and production. Resource Manager provides security, auditing, and tagging features to help you manage your resources after deployment.

Terminology

If you are new to Azure Resource Manager, there are some terms you might not be familiar with.

- **resource** - A manageable item that is available through Azure. Some common resources are a virtual machine, storage account, web app, database, and virtual network, but there are many more.
- **resource group** - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. See [Resource groups](#).
- **resource provider** - A service that supplies the resources you can deploy and manage through Resource Manager. Each resource provider offers operations for working with the resources that are deployed. Some common resource providers are Microsoft.Compute, which supplies the virtual machine resource, Microsoft.Storage, which supplies the storage account resource, and Microsoft.Web, which supplies resources related to web apps. See [Resource providers](#).
- **Resource Manager template** - A JavaScript Object Notation (JSON) file that defines one or more resources to deploy to a resource group. It also defines the dependencies between the deployed resources. The template can be used to deploy the resources consistently and repeatedly. See [Template deployment](#).
- **declarative syntax** - Syntax that lets you state "Here is what I intend to create" without having to write the sequence of programming commands to create it. The Resource Manager template is an example of declarative syntax. In the file, you define the properties for the infrastructure to deploy to Azure.

The benefits of using Resource Manager

Resource Manager provides several benefits:

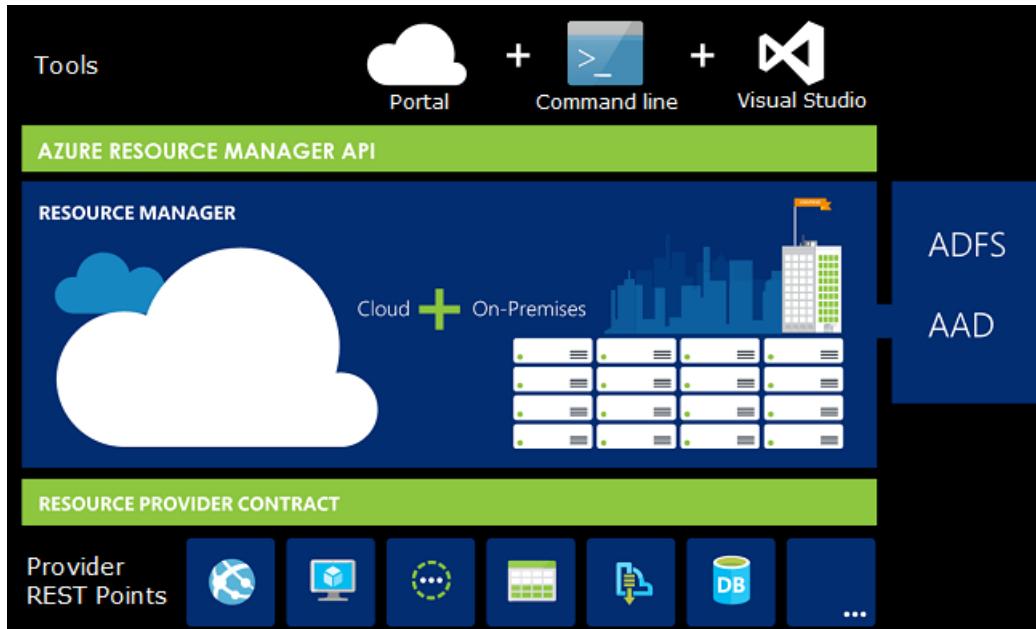
- You can deploy, manage, and monitor all the resources for your solution as a group, rather than handling these resources individually.
- You can repeatedly deploy your solution throughout the development lifecycle and have confidence your resources are deployed in a consistent state.
- You can manage your infrastructure through declarative templates rather than scripts.
- You can define the dependencies between resources so they are deployed in the correct order.
- You can apply access control to all services in your resource group because Role-Based Access Control (RBAC) is natively integrated into the management platform.
- You can apply tags to resources to logically organize all the resources in your subscription.
- You can clarify your organization's billing by viewing costs for a group of resources sharing the same tag.

Resource Manager provides a new way to deploy and manage your solutions. If you used the earlier deployment model and want to learn about the changes, see [Understanding Resource Manager deployment and classic deployment](#).

Consistent management layer

Resource Manager provides a consistent management layer for the tasks you perform through Azure PowerShell, Azure CLI, Azure portal, REST API, and development tools. All the tools use a common set of operations. You use the tools that work best for you, and can use them interchangeably without confusion.

The following image shows how all the tools interact with the same Azure Resource Manager API. The API passes requests to the Resource Manager service, which authenticates and authorizes the requests. Resource Manager then routes the requests to the appropriate resource providers.



Guidance

The following suggestions help you take full advantage of Resource Manager when working with your solutions.

1. Define and deploy your infrastructure through the declarative syntax in Resource Manager templates, rather than through imperative commands.
2. Define all deployment and configuration steps in the template. You should have no manual steps for setting up your solution.
3. Run imperative commands to manage your resources, such as to start or stop an app or machine.
4. Arrange resources with the same lifecycle in a resource group. Use tags for all other organizing of resources.

For recommendations about templates, see [Best practices for creating Azure Resource Manager templates](#).

For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Resource groups

There are some important factors to consider when defining your resource group:

1. All the resources in your group should share the same lifecycle. You deploy, update, and delete them together. If one resource, such as a database server, needs to exist on a different deployment cycle it should be in another resource group.
2. Each resource can only exist in one resource group.

3. You can add or remove a resource to a resource group at any time.
4. You can move a resource from one resource group to another group. For more information, see [Move resources to new resource group or subscription](#).
5. A resource group can contain resources that reside in different regions.
6. A resource group can be used to scope access control for administrative actions.
7. A resource can interact with resources in other resource groups. This interaction is common when the two resources are related but do not share the same lifecycle (for example, web apps connecting to a database).

When creating a resource group, you need to provide a location for that resource group. You may be wondering, "Why does a resource group need a location? And, if the resources can have different locations than the resource group, why does the resource group location matter at all?" The resource group stores metadata about the resources. Therefore, when you specify a location for the resource group, you are specifying where that metadata is stored. For compliance reasons, you may need to ensure that your data is stored in a particular region.

Resource providers

Each resource provider offers a set of resources and operations for working with an Azure service. For example, if you want to store keys and secrets, you work with the **Microsoft.KeyVault** resource provider. This resource provider offers a resource type called **vaults** for creating the key vault.

Before getting started with deploying your resources, you should gain an understanding of the available resource providers. Knowing the names of resource providers and resources helps you define resources you want to deploy to Azure.

You can see all resource providers through the portal. In the blade for your subscription, select **Resource providers**:

PROVIDER	STATUS	
Microsoft.ADHybridHealthService	Registered	Unregister
Microsoft.Authorization	Registered	Unregister
Microsoft.ClassicStorage	Registered	Unregister
Microsoft.Features	Registered	Unregister
Microsoft.OperationalInsights	Registered	Unregister
Microsoft.Resources	Registered	Unregister
Microsoft.Storage	Registered	Unregister
microsoft.support	Registered	Unregister
84codes.CloudAMQP	NotRegistered	Register
AppDynamics.APM	NotRegistered	Register

You retrieve all resource providers with the following PowerShell cmdlet:

```
Get-AzureRmResourceProvider -ListAvailable
```

Or, with Azure CLI 2.0, you retrieve all resource providers with the following command:

```
az provider list
```

You can look through the returned list for the resource providers that you need to use.

To get details about a resource provider, add the provider namespace to your command. The command returns the supported resource types for the resource provider, and the supported locations and API versions for each resource type. The following PowerShell cmdlet gets details about Microsoft.Compute:

```
(Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Compute).ResourceTypes
```

Or, with Azure CLI 2.0, retrieve the supported resource types, locations, and API versions for Microsoft.Compute, with the following command:

```
az provider show --namespace Microsoft.Compute
```

For more information, see [Resource Manager providers, regions, API versions, and schemas](#).

Template deployment

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 create a solution from the portal, the solution automatically includes a deployment template. You do not have to create your template from scratch because you can start with the template for your solution and customize it to meet your specific needs. You can retrieve a template for an existing resource group by either exporting the current state of the resource group, or viewing the template used for a particular deployment. Viewing the [exported template](#) is a helpful way to learn about the template syntax.

To learn about the format of the template and how you construct it, see [Create your first Azure Resource Manager template](#). To view the JSON syntax for resources types, see [Define resources in Azure Resource Manager templates](#).

Resource Manager processes the template like any other request (see the image for [Consistent management layer](#)). It parses the template and converts its syntax into REST API operations for the appropriate resource providers. For example, when Resource Manager receives a template with the following resource definition:

```
"resources": [
  {
    "apiVersion": "2016-01-01",
    "type": "Microsoft.Storage/storageAccounts",
    "name": "mystorageaccount",
    "location": "westus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "Storage",
    "properties": {}
  }
]
```

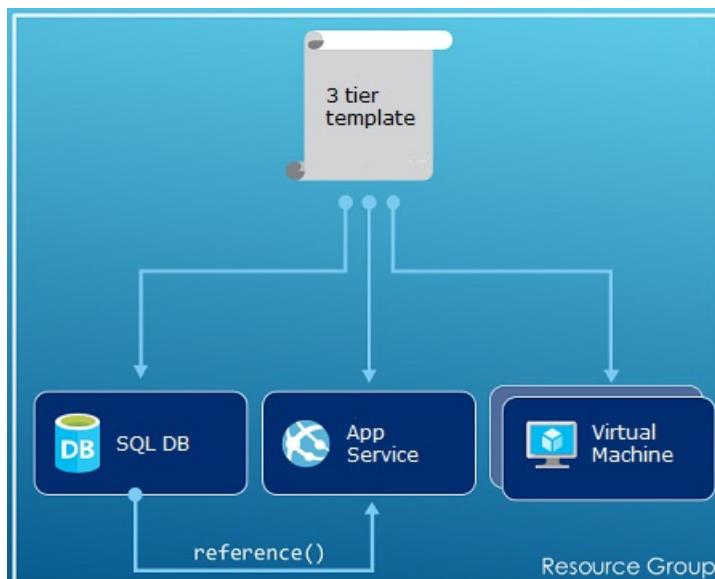
It converts the definition to the following REST API operation, which is sent to the Microsoft.Storage resource provider:

```

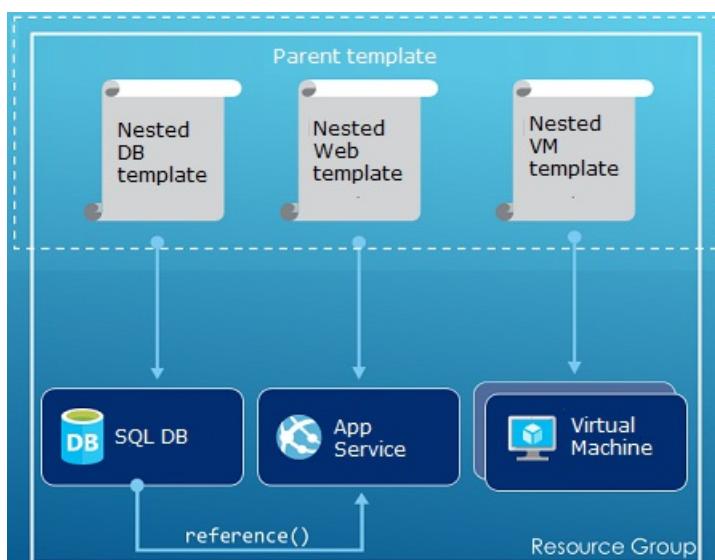
PUT
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Storage/storageAccounts/mystorageaccount?api-version=2016-01-01
REQUEST BODY
{
  "location": "westus",
  "properties": {
  },
  "sku": {
    "name": "Standard_LRS"
  },
  "kind": "Storage"
}

```

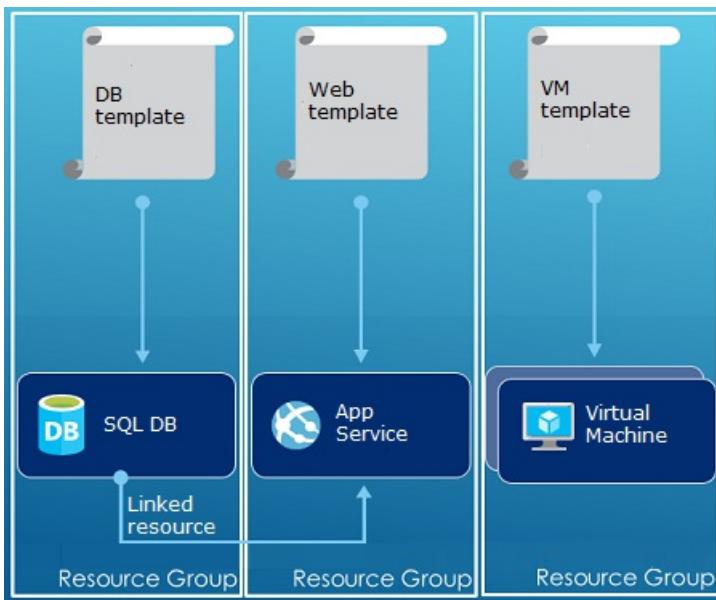
How you define templates and resource groups is entirely up to you and how you want to manage your solution. For example, you can deploy your three tier application through a single template to a single resource group.



But, you do not 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. Notice the resources can still be linked to resources in other resource groups.



For more suggestions about designing your templates, see [Patterns for designing Azure Resource Manager templates](#). For information about nested templates, see [Using linked templates with Azure Resource Manager](#).

For a four part series about automating deployment, see [Automating application deployments to Azure Virtual Machines](#). This series covers application architecture, access and security, availability and scale, and application deployment.

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 is not included in the setup. If you are already using a configuration management service, like DSC, Chef or Puppet, you can continue working with that service by using extensions. For information about virtual machine extensions, see [About virtual machine extensions and features](#).

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.

After defining your template, you are ready to deploy the resources to Azure. For the commands to deploy the resources, see:

- [Deploy resources with Resource Manager templates and Azure PowerShell](#)
- [Deploy resources with Resource Manager templates and Azure CLI](#)
- [Deploy resources with Resource Manager templates and Azure portal](#)
- [Deploy resources with Resource Manager templates and Resource Manager REST API](#)

Tags

Resource Manager provides a tagging feature that enables you to categorize resources according to your requirements for managing or billing. Use tags when you have a complex collection of resource groups and resources, and need to visualize those assets in the way that makes the most sense to you. For example, you could tag resources that serve a similar role in your organization or belong to the same department. Without tags, users in your organization can create multiple resources that may be difficult to later identify and manage. For example,

you may wish to delete all the resources for a particular project. If those resources are not tagged for the project, you have to manually find them. Tagging can be an important way for you to reduce unnecessary costs in your subscription.

Resources do not need to reside in the same resource group to share a tag. You can create your own tag taxonomy to ensure that all users in your organization use common tags rather than users inadvertently applying slightly different tags (such as "dept" instead of "department").

The following example shows a tag applied to a virtual machine.

```
"resources": [
  {
    "type": "Microsoft.Compute/virtualMachines",
    "apiVersion": "2015-06-15",
    "name": "SimpleWindowsVM",
    "location": "[resourceGroup().location]",
    "tags": {
      "costCenter": "Finance"
    },
    ...
  }
]
```

To retrieve all the resources with a tag value, use the following PowerShell cmdlet:

```
Find-AzureRmResource -TagName costCenter -TagValue Finance
```

Or, the following Azure CLI 2.0 command:

```
az resource list --tag costCenter=Finance
```

You can also view tagged resources through the Azure portal.

The [usage report](#) for your subscription includes tag names and values, which enables you to break out costs by tags. For more information about tags, see [Using tags to organize your Azure resources](#).

Access control

Resource Manager enables you to control who has access to specific actions for your organization. It natively integrates role-based access control (RBAC) into the management platform and applies that access control to all services in your resource group.

There are two main concepts to understand when working with role-based access control:

- Role definitions - describe a set of permissions and can be used in many assignments.
- Role assignments - associate a definition with an identity (user or group) for a particular scope (subscription, resource group, or resource). The assignment is inherited by lower scopes.

You can add users to pre-defined platform and resource-specific roles. For example, you can take advantage of the pre-defined role called Reader that permits users to view resources but not change them. You add users in your organization that need this type of access to the Reader role and apply the role to the subscription, resource group, or resource.

Azure provides the following four platform roles:

1. Owner - can manage everything, including access
2. Contributor - can manage everything except access

3. Reader - can view everything, but can't make changes
4. User Access Administrator - can manage user access to Azure resources

Azure also provides several resource-specific roles. Some common ones are:

1. Virtual Machine Contributor - can manage virtual machines but not grant access to them, and cannot manage the virtual network or storage account to which they are connected
2. Network Contributor - can manage all network resources, but not grant access to them
3. Storage Account Contributor - Can manage storage accounts, but not grant access to them
4. SQL Server Contributor - Can manage SQL servers and databases, but not their security-related policies
5. Website Contributor - Can manage websites, but not the web plans to which they are connected

For the full list of roles and permitted actions, see [RBAC: Built in Roles](#). For more information about role-based access control, see [Azure Role-based Access Control](#).

In some cases, you want to run code or script that accesses resources, but you do not want to run it under a user's credentials. Instead, you want to create an identity called a service principal for the application and assign the appropriate role for the service principal. Resource Manager enables you to create credentials for the application and programmatically authenticate the application. To learn about creating service principals, see one of following topics:

- [Use Azure PowerShell to create a service principal to access resources](#)
- [Use Azure CLI to create a service principal to access resources](#)
- [Use portal to create Active Directory application and service principal that can access resources](#)

You can also explicitly lock critical resources to prevent users from deleting or modifying them. For more information, see [Lock resources with Azure Resource Manager](#).

Activity logs

Resource Manager logs all operations that create, modify, or delete a resource. You can use the activity logs to find an error when troubleshooting or to monitor how a user in your organization modified a resource. To see the logs, select **Activity logs** in the **Settings** blade for a resource group. You can filter the logs by many different values including which user initiated the operation. For information about working with the activity logs, see [View activity logs to manage Azure resources](#).

Customized policies

Resource Manager enables you to create customized policies for managing your resources. The types of policies you create can include diverse scenarios. You can enforce a naming convention on resources, limit which types and instances of resources can be deployed, or limit which regions can host a type of resource. You can require a tag value on resources to organize billing by departments. You create policies to help reduce costs and maintain consistency in your subscription.

You define policies with JSON and then apply those policies either across your subscription or within a resource group. Policies are different than role-based access control because they are applied to resource types.

The following example shows a policy that ensures tag consistency by specifying that all resources include a costCenter tag.

```
{  
  "if": {  
    "not" : {  
      "field" : "tags",  
      "containsKey" : "costCenter"  
    }  
  },  
  "then" : {  
    "effect" : "deny"  
  }  
}
```

There are many more types of policies you can create. For more information, see [Use Policy to manage resources and control access](#).

SDKs

Azure SDKs are available for multiple languages and platforms. Each of these language implementations is available through its ecosystem package manager and GitHub.

The code in each of these SDKs is generated from Azure RESTful API specifications. These specifications are open source and based on the Swagger 2.0 specification. The SDK code is generated via an open-source project called AutoRest. AutoRest transforms these RESTful API specifications into client libraries in multiple languages. If you want to improve any aspects of the generated code in the SDKs, the entire set of tools to create the SDKs are open, freely available, and based on a widely adopted API specification format.

Here are our Open Source SDK repositories. We welcome feedback, issues, and pull requests.

[.NET](#) | [Java](#) | [Node.js](#) | [PHP](#) | [Python](#) | [Ruby](#)

NOTE

If the SDK doesn't provide the required functionality, you can also call to the [Azure REST API](#) directly.

Samples

.NET

- [Manage Azure resources and resource groups](#)
- [Deploy an SSH enabled VM with a template](#)

Java

- [Manage Azure resources](#)
- [Manage Azure resource groups](#)
- [Deploy an SSH enabled VM with a template](#)

Node.js

- [Manage Azure resources and resource groups](#)
- [Deploy an SSH enabled VM with a template](#)

Python

- [Manage Azure resources and resource groups](#)
- [Deploy an SSH enabled VM with a template](#)

Ruby

- [Manage Azure resources and resource groups](#)

- [Deploy an SSH enabled VM with a template](#)

In addition to these samples, you can search through the gallery samples.

[.NET](#) | [Java](#) | [Node.js](#) | [Python](#) | [Ruby](#)

Next steps

- For a simple introduction to working with templates, see [Export an Azure Resource Manager template from existing resources](#).
- For a more thorough walkthrough of creating a template, see [Create your first Azure Resource Manager template](#).
- To understand the functions you can use in a template, see [Template functions](#)
- For information about using Visual Studio with Resource Manager, see [Creating and deploying Azure resource groups through Visual Studio](#).

Here's a video demonstration of this overview:

Resource Manager providers, regions, API versions and schemas

3/30/2017 • 8 min to read • [Edit Online](#)

This topic provides a list of resource providers that support Azure Resource Manager.

When deploying your resources, you also need to know which regions support those resources and which API versions are available for the resources. The section [Supported regions](#) shows you how to find out which regions work for your subscription and resources. The section [Supported API versions](#) shows you how to determine which API versions you can use.

To see which services are supported in the Azure portal and classic portal, see [Azure portal availability chart](#). To see which services support moving resources, see [Move resources to new resource group or subscription](#).

The following tables list which Microsoft services support deployment and management through Resource Manager and which do not. There are also many third-party resource providers that support Resource Manager. You learn how to see all the resource providers in the [Resource providers and types](#) section.

Compute

Service	Resource Manager Enabled	REST API	Template Format
Batch	Yes	Batch REST	Batch resources
Container Registry	Yes	Container Registry REST	Container Registry resources
Container Service	Yes	Container Service REST	Container Service resources
Dynamics Lifecycle Services	Yes		
Scale Sets	Yes	Scale Set REST	Scale Set resources
Service Fabric	Yes	Service Fabric Rest	Service Fabric resources
Virtual Machines	Yes	VM REST	VM resources
Virtual Machines (classic)	Limited	-	-
Remote App	No	-	-
Cloud Services (classic)	Limited (see below)	-	-

Virtual Machines (classic) refers to resources that were deployed through the classic deployment model, instead of through the Resource Manager deployment model. In general, these resources do not support Resource Manager operations, but there are some operations that have been enabled. For more information about these deployment models, see [Understanding Resource Manager deployment and classic deployment](#).

Cloud Services (classic) can be used with other classic resources. However, classic resources do not take advantage of all Resource Manager features and are not a good option for future solutions. Instead, consider changing your application infrastructure to use resources from the Microsoft.Compute, Microsoft.Storage, and Microsoft.Network

namespaces.

Networking

SERVICE	RESOURCE MANAGER ENABLED	REST API	TEMPLATE FORMAT
Application Gateway	Yes	Application Gateway REST	Application Gateway resources
DNS	Yes	DNS REST	DNS resources
ExpressRoute	Yes	ExpressRoute REST	ExpressRoute resources
Load Balancer	Yes	Load Balancer REST	Load Balancer resources
Traffic Manager	Yes	Traffic Manager REST	Traffic Manager resources
Virtual Networks	Yes	Virtual Network REST	Virtual Network resources
Network Gateway	Yes	Network Gateway REST	Connection resources Local Network Gateway resources Virtual Network Gateway resources

Storage

SERVICE	RESOURCE MANAGER ENABLED	REST API	TEMPLATE FORMAT
Import Export	Yes	Import Export REST	Import Export resources
Storage	Yes	Storage REST	Storage resources
StorSimple	Yes		

Databases

SERVICE	RESOURCE MANAGER ENABLED	REST API	TEMPLATE FORMAT
DocumentDB	Yes	DocumentDB REST	DocumentDB resources
Redis Cache	Yes	Redis Cache REST	Redis resources
SQL Database	Yes	SQL Database REST	SQL Database resources
SQL Data Warehouse	Yes		

Web & Mobile

Service	Resource Manager Enabled	REST API	Template Format
API Apps	Yes	App Service REST	Web resources
API Management	Yes	API Management REST	API Management resources
Certificate Registration	Yes	Certificate Registration REST	Certificate Registration resources
Content Moderator	Yes		
Domain Registration	Yes	Domain Registration	Domain Registration resources
Function App	Yes	Function App REST	Web resources
Logic Apps	Yes	Logic Apps REST	Logic App resources
Mobile Apps	Yes	App Service REST	Web resources
Mobile Engagements	Yes	Mobile Engagement REST	
Search	Yes	Search REST	Search resources
Web Apps	Yes	Web Apps REST	Web resources

Intelligence + Analytics

Service	Resource Manager Enabled	REST API	Template Format
Analysis Services	Yes	Analysis Service REST	Analysis Services resources
Cognitive Services	Yes	Cognitive Services REST	Cognitive Services resources
Data Catalog	Yes	Data Catalog REST	Data Catalog Schema
Data Factory	Yes	Data Factory REST	
Data Lake Analytics	Yes	Data Lake REST	Data Lake Analytics resources
Data Lake Store	Yes	Data Lake Store REST	Data Lake Store resources
HDInsights	Yes	HDInsights REST	
Machine Learning	Yes	Machine Learning REST	Machine Learning resources
Stream Analytics	Yes	Steam Analytics REST	
Power BI	Yes	Power BI Embedded REST	Power BI resources

Internet of Things

Service	Resource Manager Enabled	REST API	Template Format
Event Hub	Yes	Event Hub REST	Event Hub resources
IoTHubs	Yes	IoT Hub REST	IoT Hub resources
Notification Hubs	Yes	Notification Hub REST	Notification Hub resources

Media & CDN

Service	Resource Manager Enabled	REST API	Template Format
CDN	Yes	CDN REST	CDN resources
Media Service	Yes	Media Services REST	Media resources

Enterprise Integration

Service	Resource Manager Enabled	REST API	Template Format
BizTalk Services	Yes		BizTalk Schema
Relay	Yes		Relay resources
Service Bus	Yes	Service Bus REST	Service Bus resources

Identity & Access Management

Azure Active Directory works with Resource Manager to enable role-based access control for your subscription. To learn about using role-based access control and Active Directory, see [Azure Role-based Access Control](#).

Developer Services

Service	Resource Manager Enabled	REST API	Template Format
Monitor	Yes	Monitor REST	Insights resources
Bing Maps	Yes		
DevTest Labs	Yes	DevTest Labs REST	DevTest Labs resources
Visual Studio account	Yes		Visual Studio Schema

Management and Security

Service	Resource Manager Enabled	REST API	Template Format
Advisor	Yes	Advisor REST	-
Automation	Yes	Automation REST	Automation resources

Service	Resource Manager Enabled	REST API	Template Format
Billing	Yes	Billing REST	-
Key Vault	Yes	Key Vault REST	Key vault resources
Operational Insights	Yes		
Recovery Service	Yes	Recovery Services REST	Recovery Services resources
Scheduler	Yes	Scheduler REST	Scheduler resources
Security	Yes	Security REST	
Server Management	Yes	Server Management REST	Server Management resources

Resource Manager

Feature	Resource Manager Enabled	REST API	Template Format
Authorization	Yes	Authorization REST	Authorization resources
Resources	Yes	Resources REST	Deployment resources

Resource providers and types

When deploying resources, you frequently need to retrieve information about the resource providers and types. You can retrieve this information through REST API, Azure PowerShell, or Azure CLI.

To work with a resource provider, that resource provider must be registered with your account. By default, many resource providers are automatically registered; however, you may need to manually register some resource providers. The examples in this section show how to get the registration status of a resource provider, and register the resource provider.

Portal

You can easily see a list of supported resources providers by selecting **Resource providers** from the subscription blade. To register your subscription with a resource provider, select the **Register** link.

 Visual Studio Enterprise - Resource providers

Subscription

Refresh

Search (Ctrl+)

Diagnose and solve problems

SETTINGS

Programmatic deployment

Resource groups

Resources

Usage + quotas

Management certificates

My permissions

Resource providers

Properties

Search to filter resource providers...

PROVIDER	STATUS	
Microsoft.ADHybridHealthService	Registered	Unregister
Microsoft.Authorization	Registered	Unregister
Microsoft.ClassicStorage	Registered	Unregister
Microsoft.Features	Registered	Unregister
Microsoft.OperationalInsights	Registered	Unregister
Microsoft.Resources	Registered	Unregister
Microsoft.Storage	Registered	Unregister
microsoft.support	Registered	Unregister
84codes.CloudAMQP	NotRegistered	Register
AppDynamics.APM	NotRegistered	Register

REST API

To get all the available resource providers, including their types, locations, API versions, and registration status, use the [List all resource providers](#) operation. If you need to register a resource provider, see [Register a subscription with a resource provider](#).

PowerShell

The following example shows how to get all the available resource providers.

```
Get-AzureRmResourceProvider -ListAvailable
```

The next example shows how to get the resource types for a particular resource provider.

```
(Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes
```

To register a resource provider, provide the namespace:

```
Register-AzureRmResourceProvider -ProviderNamespace Microsoft.ApiManagement
```

Azure CLI

The following example shows how to get all the available resource providers.

```
az provider list
```

You can view the information for a particular resource provider with the following command:

```
az provider show --namespace Microsoft.Web
```

To register a resource provider, provide the namespace:

```
az provider register --namespace Microsoft.ServiceBus
```

Supported regions

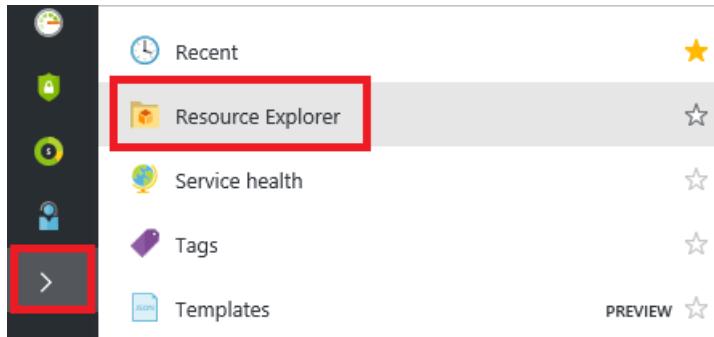
When deploying resources, you typically need to specify a region for the resources. Resource Manager is supported in all regions, but the resources you deploy might not be supported in all regions. In addition, there may be limitations on your subscription that prevent you from using some regions that support the resource. These limitations may be related to tax issues for your home country, or the result of a policy placed by your subscription administrator to use only certain regions.

For a complete list of all supported regions for all Azure services, see [Services by region](#). However, this list may include regions that your subscription does not support. You can determine the regions for a particular resource type that your subscription supports through the portal, REST API, PowerShell, or Azure CLI.

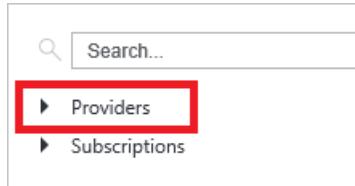
Portal

You can see the supported regions for a resource type through the following steps:

1. Select **More services > Resource Explorer**.



2. Open the **Providers** node.



3. Select a resource provider, and view the supported regions and API versions.

The screenshot shows the Azure REST API Explorer interface. On the left, there is a search bar and a sidebar with a tree view of resource providers. The 'Microsoft.Cdn' provider is selected, indicated by a blue border around its node. The main pane displays the API endpoint `/providers/Microsoft.Cdn?api-version=2014-04-01-preview` and its corresponding JSON response. The JSON response shows the provider's namespace, supported resource types (profiles), and locations across various regions.

```

1 {
2   "namespace": "Microsoft.Cdn",
3   "resourceTypes": [
4     {
5       "resourceType": "profiles",
6       "locations": [
7         "Australia East",
8         "Australia Southeast",
9         "Brazil South",
10        "Canada Central",
11        "Canada East",
12        "Central India",
13        "Central US",
14        "East Asia",
15        "East US",
16        "East US 2",
17        "Japan East",
18        "Japan West",
19        "North Central US",
20        "North Europe",
21        "South Central US",
22        "South India",
23        "Southeast Asia",
24        "West Europe",
25        "West India",
26        "West US"
27      ],
28      "apiVersions": [
29        "2016-10-02",
30        "2016-04-02",
31        "2015-06-01"
32    ],
33  },

```

REST API

To discover which regions are available for a particular resource type in your subscription, use the [List all resource providers](#) operation.

PowerShell

The following example shows how to get the supported regions for web sites.

```
((Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | Where-Object ResourceTypeName -eq sites).Locations
```

Azure CLI

The following example show how to get the supported locations for web sites.

```
az provider show --namespace Microsoft.Web --query "resourceTypes[?resourceType=='sites'].locations"
```

Supported API versions

When you deploy a template, you must specify an API version to use for creating each resource. The API version corresponds to a version of REST API operations that are released by the resource provider. As a resource provider enables new features, it releases a new version of the REST API. Therefore, the version of the API you specify in your template affects which properties you can specify in the template. In general, you want to select the most recent API

version when creating templates. For existing templates, you can decide whether you want to continue using an earlier API version, or update your template for the latest version to take advantage of new features.

Portal

You determine the supported API versions in the same way you determined supported regions (shown previously).

REST API

To discover which API versions are available for resource types, use the [List all resource providers](#) operation.

PowerShell

The following example shows how to get the available API versions for a particular resource type.

```
((Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | Where-Object ResourceType -eq sites).ApiVersions
```

The output is similar to:

```
2015-08-01  
2015-07-01  
2015-06-01  
2015-05-01  
2015-04-01  
2015-02-01  
2014-11-01  
2014-06-01  
2014-04-01-preview  
2014-04-01
```

Azure CLI

You get the available API versions for a resource provider with the following command:

```
az provider show --namespace Microsoft.Web --query "resourceTypes[?resourceType=='sites'].apiVersions"
```

Next steps

- To learn about creating Resource Manager templates, see [Authoring Azure Resource Manager templates](#).
- To learn about deploying resources, see [Deploy an application with Azure Resource Manager template](#).

Azure Resource Manager vs. classic deployment: Understand deployment models and the state of your resources

4/3/2017 • 12 min to read • [Edit Online](#)

In this topic, you learn about Azure Resource Manager and classic deployment models, the state of your resources, and why your resources were deployed with one or the other. The Resource Manager and classic deployment models represent two different ways of deploying and managing your Azure solutions. You work with them through two different API sets, and the deployed resources can contain important differences. The two models are not completely compatible with each other. This topic describes those differences.

To simplify the deployment and management of resources, Microsoft recommends that you use Resource Manager for all new resources. If possible, Microsoft recommends that you redeploy existing resources through Resource Manager.

If you are new to Resource Manager, you may want to first review the terminology defined in the [Azure Resource Manager overview](#).

History of the deployment models

Azure originally provided only the classic deployment model. In this model, each resource existed independently; there was no way to group related resources together. Instead, you had to manually track which resources made up your solution or application, and remember to manage them in a coordinated approach. To deploy a solution, you had to either create each resource individually through the classic portal or create a script that deployed all the resources in the correct order. To delete a solution, you had to delete each resource individually. You could not easily apply and update access control policies for related resources. Finally, you could not apply tags to resources to label them with terms that help you monitor your resources and manage billing.

In 2014, Azure introduced Resource Manager, which added the concept of a resource group. A resource group is a container for resources that share a common lifecycle. The Resource Manager deployment model provides several benefits:

- You can deploy, manage, and monitor all the services for your solution as a group, rather than handling these services individually.
- You can repeatedly deploy your solution throughout its lifecycle and have confidence your resources are deployed in a consistent state.
- You can apply access control to all resources in your resource group, and those policies are automatically applied when new resources are added to the resource group.
- You can apply tags to resources to logically organize all the resources in your subscription.
- You can use JavaScript Object Notation (JSON) to define the infrastructure for your solution. The JSON file is known as a Resource Manager template.
- You can define the dependencies between resources so they are deployed in the correct order.

When Resource Manager was added, all resources were retroactively added to default resource groups. If you create a resource through classic deployment now, the resource is automatically created within a default resource group for that service, even though you did not specify that resource group at deployment. However, just existing within a resource group does not mean that the resource has been converted to the Resource Manager model. We'll look at how each service handles the two deployment models in the next section.

Understand support for the models

When deciding which deployment model to use for your resources, there are three scenarios to be aware of:

1. The service supports Resource Manager and provides only a single type.
2. The service supports Resource Manager but provides two types - one for Resource Manager and one for classic.
This scenario applies only to virtual machines, storage accounts, and virtual networks.
3. The service does not support Resource Manager.

To discover whether a service supports Resource Manager, see [Resource Manager supported providers](#).

If the service you wish to use does not support Resource Manager, you must continue using classic deployment.

If the service supports Resource Manager and **is not** a virtual machine, storage account or virtual network, you can use Resource Manager without any complications.

For virtual machines, storage accounts, and virtual networks, if the resource was created through classic deployment, you must continue to operate on it through classic operations. If the virtual machine, storage account, or virtual network was created through Resource Manager deployment, you must continue using Resource Manager operations. This distinction can get confusing when your subscription contains a mix of resources created through Resource Manager and classic deployment. This combination of resources can create unexpected results because the resources do not support the same operations.

In some cases, a Resource Manager command can retrieve information about a resource created through classic deployment, or can perform an administrative task such as moving a classic resource to another resource group. But, these cases should not give the impression that the type supports Resource Manager operations. For example, suppose you have a resource group that contains a virtual machine that was created with classic deployment. If you run the following Resource Manager PowerShell command:

```
Get-AzureRmResource -ResourceGroupName ExampleGroup -ResourceType Microsoft.ClassicCompute/virtualMachines
```

It returns the virtual machine:

```
Name          : ExampleClassicVM
ResourceId    :
/subscriptions/{guid}/resourceGroups/ExampleGroup/providers/Microsoft.ClassicCompute/virtualMachines/ExampleClassicVM
ResourceName   : ExampleClassicVM
ResourceType   : Microsoft.ClassicCompute/virtualMachines
ResourceGroupName : ExampleGroup
Location       : westus
SubscriptionId : {guid}
```

However, the Resource Manager cmdlet **Get-AzureRmVM** only returns virtual machines deployed through Resource Manager. The following command does not return the virtual machine created through classic deployment.

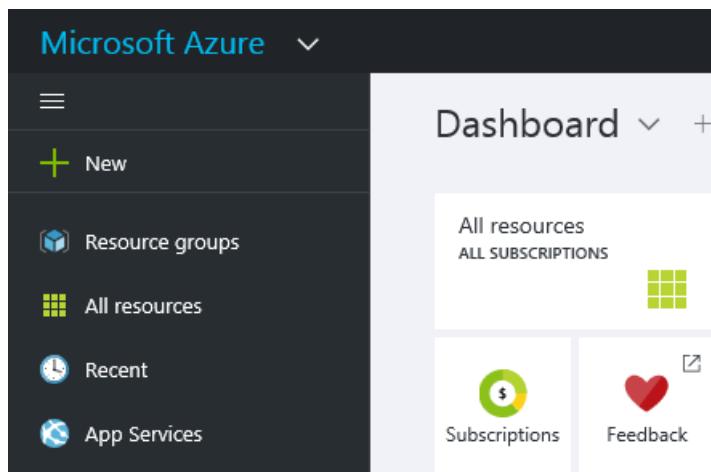
```
Get-AzureRmVM -ResourceGroupName ExampleGroup
```

Only resources created through Resource Manager support tags. You cannot apply tags to classic resources.

Resource Manager characteristics

To help you understand the two models, let's review the characteristics of Resource Manager types:

- Created through the [Azure portal](#).



For Compute, Storage, and Networking resources, you have the option of using either Resource Manager or Classic deployment. Select **Resource Manager**.

Select a deployment model ⓘ

Resource Manager

Create

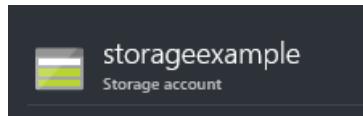
- Created with the Resource Manager version of the Azure PowerShell cmdlets. These commands have the format *Verb-AzureRmNoun*.

```
New-AzureRmResourceGroupDeployment
```

- Created through the [Azure Resource Manager REST API](#) for REST operations.
- Created through Azure CLI commands run in the **arm** mode.

```
azure config mode arm  
azure group deployment create
```

- The resource type does not include (**classic**) in the name. The following image shows the type as **Storage account**.



Classic deployment characteristics

You may also know the classic deployment model as the Service Management model.

Resources created in the classic deployment model share the following characteristics:

- Created through the [classic portal](#)

The screenshot shows the Microsoft Azure portal interface. The top navigation bar includes the Microsoft Azure logo and a dropdown menu. A blue banner on the right says "Check out the new por...". The main content area is titled "all items". On the left, there's a sidebar with icons for "ALL ITEMS", "WEB APPS" (0), "VIRTUAL MACHINES" (0), and "MOBILE SERVICES" (0). The main pane displays a table with one row: "NAME" and "Tom FitzMacken".

Or, the Azure portal and you specify **Classic** deployment (for Compute, Storage, and Networking).

The screenshot shows the Azure portal's "Create" blade. It has a dropdown menu labeled "Select a deployment model" with "Classic" selected. At the bottom is a large blue "Create" button.

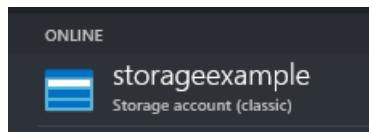
- Created through the Service Management version of the Azure PowerShell cmdlets. These command names have the format *Verb-AzureNoun*.

```
New-AzureVM
```

- Created through the [Service Management REST API](#) for REST operations.
- Created through Azure CLI commands run in **asm** mode.

```
azure config mode asm  
azure vm create
```

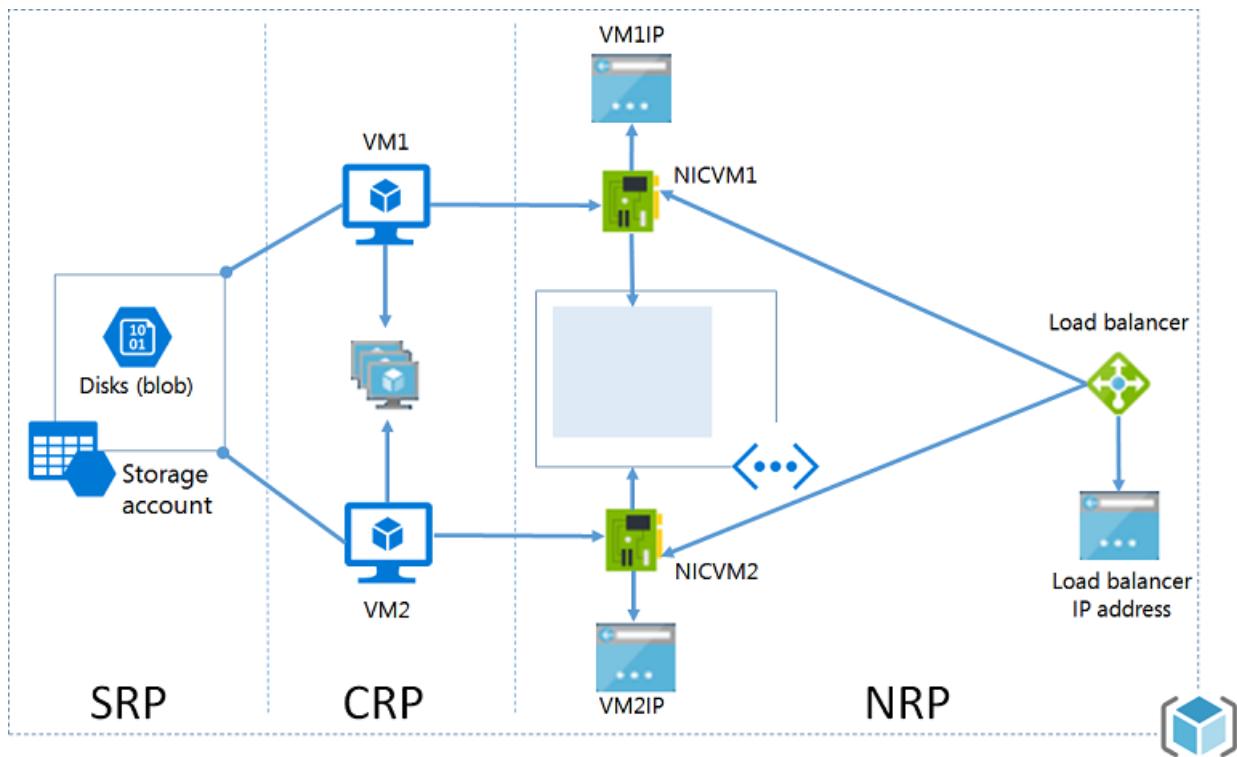
- The resource type includes (**classic**) in the name. The following image shows the type as **Storage account (classic)**.



You can use the Azure portal to manage resources that were created through classic deployment.

Changes for compute, network, and storage

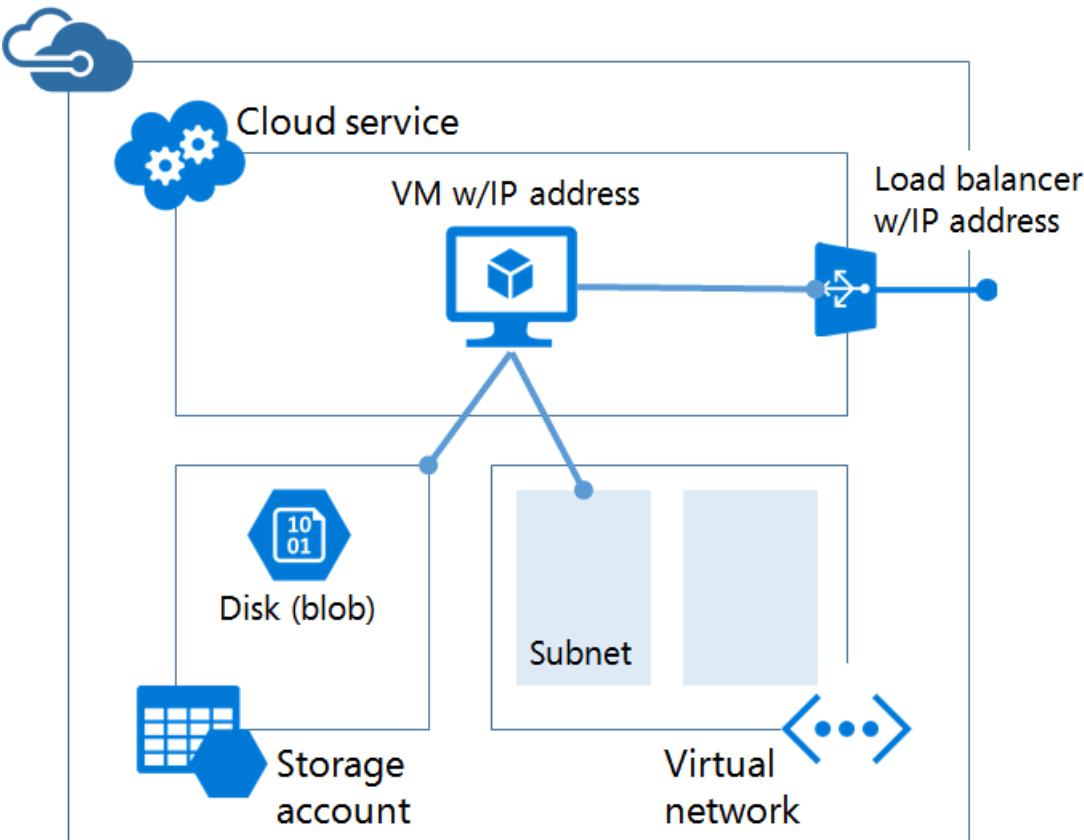
The following diagram displays compute, network, and storage resources deployed through Resource Manager.



Note the following relationships between the resources:

- All the resources exist within a resource group.
- The virtual machine depends on a specific storage account defined in the Storage resource provider to store its disks in blob storage (required).
- The virtual machine references a specific NIC defined in the Network resource provider (required) and an availability set defined in the Compute resource provider (optional).
- The NIC references the virtual machine's assigned IP address (required), the subnet of the virtual network for the virtual machine (required), and to a Network Security Group (optional).
- The subnet within a virtual network references a Network Security Group (optional).
- The load balancer instance references the backend pool of IP addresses that include the NIC of a virtual machine (optional) and references a load balancer public or private IP address (optional).

Here are the components and their relationships for classic deployment:



The classic solution for hosting a virtual machine includes:

- A required cloud service that acts as a container for hosting virtual machines (compute). Virtual machines are automatically provided with a network interface card (NIC) and an IP address assigned by Azure. Additionally, the cloud service contains an external load balancer instance, a public IP address, and default endpoints to allow remote desktop and remote PowerShell traffic for Windows-based virtual machines and Secure Shell (SSH) traffic for Linux-based virtual machines.
- A required storage account that stores the VHDs for a virtual machine, including the operating system, temporary, and additional data disks (storage).
- An optional virtual network that acts as an additional container, in which you can create a subnetted structure and designate the subnet on which the virtual machine is located (network).

The following table describes changes in how Compute, Network, and Storage resource providers interact:

ITEM	CLASSIC	RESOURCE MANAGER
Cloud Service for Virtual Machines	Cloud Service was a container for holding the virtual machines that required Availability from the platform and Load Balancing.	Cloud Service is no longer an object required for creating a Virtual Machine using the new model.
Virtual Networks	A virtual network is optional for the virtual machine. If included, the virtual network cannot be deployed with Resource Manager.	Virtual machine requires a virtual network that has been deployed with Resource Manager.
Storage Accounts	The virtual machine requires a storage account that stores the VHDs for the operating system, temporary, and additional data disks.	The virtual machine requires a storage account to store its disks in blob storage.

ITEM	CLASSIC	RESOURCE MANAGER
Availability Sets	Availability to the platform was indicated by configuring the same "AvailabilitySetName" on the Virtual Machines. The maximum count of fault domains was 2.	Availability Set is a resource exposed by Microsoft.Compute Provider. Virtual Machines that require high availability must be included in the Availability Set. The maximum count of fault domains is now 3.
Affinity Groups	Affinity Groups were required for creating Virtual Networks. However, with the introduction of Regional Virtual Networks, that was not required anymore.	To simplify, the Affinity Groups concept doesn't exist in the APIs exposed through Azure Resource Manager.
Load Balancing	Creation of a Cloud Service provides an implicit load balancer for the Virtual Machines deployed.	The Load Balancer is a resource exposed by the Microsoft.Network provider. The primary network interface of the Virtual Machines that needs to be load balanced should be referencing the load balancer. Load Balancers can be internal or external. A load balancer instance references the backend pool of IP addresses that include the NIC of a virtual machine (optional) and references a load balancer public or private IP address (optional). Read more .
Virtual IP Address	Cloud Services get a default VIP (Virtual IP Address) when a VM is added to a cloud service. The Virtual IP Address is the address associated with the implicit load balancer.	Public IP address is a resource exposed by the Microsoft.Network provider. Public IP Address can be Static (Reserved) or Dynamic. Dynamic Public IPs can be assigned to a Load Balancer. Public IPs can be secured using Security Groups.
Reserved IP Address	You can reserve an IP Address in Azure and associate it with a Cloud Service to ensure that the IP Address is sticky.	Public IP Address can be created in "Static" mode and it offers the same capability as a "Reserved IP Address". Static Public IPs can only be assigned to a Load balancer right now.
Public IP Address (PIP) per VM	Public IP Addresses can also be associated to a VM directly.	Public IP address is a resource exposed by the Microsoft.Network provider. Public IP Address can be Static (Reserved) or Dynamic. However, only dynamic Public IPs can be assigned to a Network Interface to get a Public IP per VM right now.
Endpoints	Input Endpoints needed to be configured on a Virtual Machine to be open up connectivity for certain ports. One of the common modes of connecting to virtual machines done by setting up input endpoints.	Inbound NAT Rules can be configured on Load Balancers to achieve the same capability of enabling endpoints on specific ports for connecting to the VMs.

ITEM	CLASSIC	RESOURCE MANAGER
DNS Name	A cloud service would get an implicit globally unique DNS Name. For example: <code>mycoffeeshop.cloudapp.net</code>	DNS Names are optional parameters that can be specified on a Public IP Address resource. The FQDN is in the following format - <code><domainlabel>. <region>.cloudapp.azure.com</code>
Network Interfaces	Primary and Secondary Network Interface and its properties were defined as network configuration of a Virtual machine.	Network Interface is a resource exposed by Microsoft.Network Provider. The lifecycle of the Network Interface is not tied to a Virtual Machine. It references the virtual machine's assigned IP address (required), the subnet of the virtual network for the virtual machine (required), and to a Network Security Group (optional).

To learn about connecting virtual networks from different deployment models, see [Connect virtual networks from different deployment models in the portal](#).

Migrate from classic to Resource Manager

If you are ready to migrate your resources from classic deployment to Resource Manager deployment, see:

1. [Technical deep dive on platform-supported migration from classic to Azure Resource Manager](#)
2. [Platform supported migration of IaaS resources from Classic to Azure Resource Manager](#)
3. [Migrate IaaS resources from classic to Azure Resource Manager by using Azure PowerShell](#)
4. [Migrate IaaS resources from classic to Azure Resource Manager by using Azure CLI](#)

Frequently Asked Questions

Can I create a virtual machine using Azure Resource Manager to deploy in a virtual network created using classic deployment?

This is not supported. You cannot use Azure Resource Manager to deploy a virtual machine into a virtual network that was created using classic deployment.

Can I create a virtual machine using the Azure Resource Manager from a user image that was created using the Azure Service Management APIs?

This is not supported. However, you can copy the VHD files from a storage account that was created using the Service Management APIs, and add them to a new account created through Azure Resource Manager.

What is the impact on the quota for my subscription?

The quotas for the virtual machines, virtual networks, and storage accounts created through the Azure Resource Manager are separate from other quotas. Each subscription gets quotas to create the resources using the new APIs. You can read more about the additional quotas [here](#).

Can I continue to use my automated scripts for provisioning virtual machines, virtual networks, and storage accounts through the Resource Manager APIs?

All the automation and scripts that you've built continue to work for the existing virtual machines, virtual networks created under the Azure Service Management mode. However, the scripts have to be updated to use the new schema for creating the same resources through the Resource Manager mode.

Where can I find examples of Azure Resource Manager templates?

A comprehensive set of starter templates can be found on [Azure Resource Manager Quickstart Templates](#).

Next steps

- To walk through the creation of template that defines a virtual machine, storage account, and virtual network, see [Resource Manager template walkthrough](#).
- To see the commands for deploying a template, see [Deploy an application with Azure Resource Manager template](#).

Azure enterprise scaffold - prescriptive subscription governance

3/31/2017 • 15 min to read • [Edit Online](#)

Enterprises are increasingly adopting the public cloud for its agility and flexibility. They are utilizing the cloud's strengths to generate revenue or optimize resources for the business. Microsoft Azure provides a multitude of services that enterprises can assemble like building blocks to address a wide array of workloads and applications.

But, knowing where to begin is often difficult. After deciding to use Azure, a few questions commonly arise:

- "How do I meet our legal requirements for data sovereignty in certain countries?"
- "How do I ensure that someone does not inadvertently change a critical system?"
- "How do I know what every resource is supporting so I can account for it and bill it back accurately?"

The prospect of an empty subscription with no guard rails is daunting. This blank space can hamper your move to Azure.

This article provides a starting point for technical professionals to address the need for governance, and balance it with the need for agility. It introduces the concept of an enterprise scaffold that guides organizations in implementing and managing their Azure subscriptions.

Need for governance

When moving to Azure, you must address the topic of governance early to ensure the successful use of the cloud within the enterprise. Unfortunately, the time and bureaucracy of creating a comprehensive governance system means some business groups go directly to vendors without involving enterprise IT. This approach can leave the enterprise open to vulnerabilities if the resources are not properly managed. The characteristics of the public cloud - agility, flexibility, and consumption-based pricing - are important to business groups that need to quickly meet the demands of customers (both internal and external). But, enterprise IT needs to ensure that data and systems are effectively protected.

In real life, scaffolding is used to create the basis of the structure. The scaffold guides the general outline, and provides anchor points for more permanent systems to be mounted. An enterprise scaffold is the same: a set of flexible controls and Azure capabilities that provide structure to the environment, and anchors for services built on the public cloud. It provides the builders (IT and business groups) a foundation to create and attach new services.

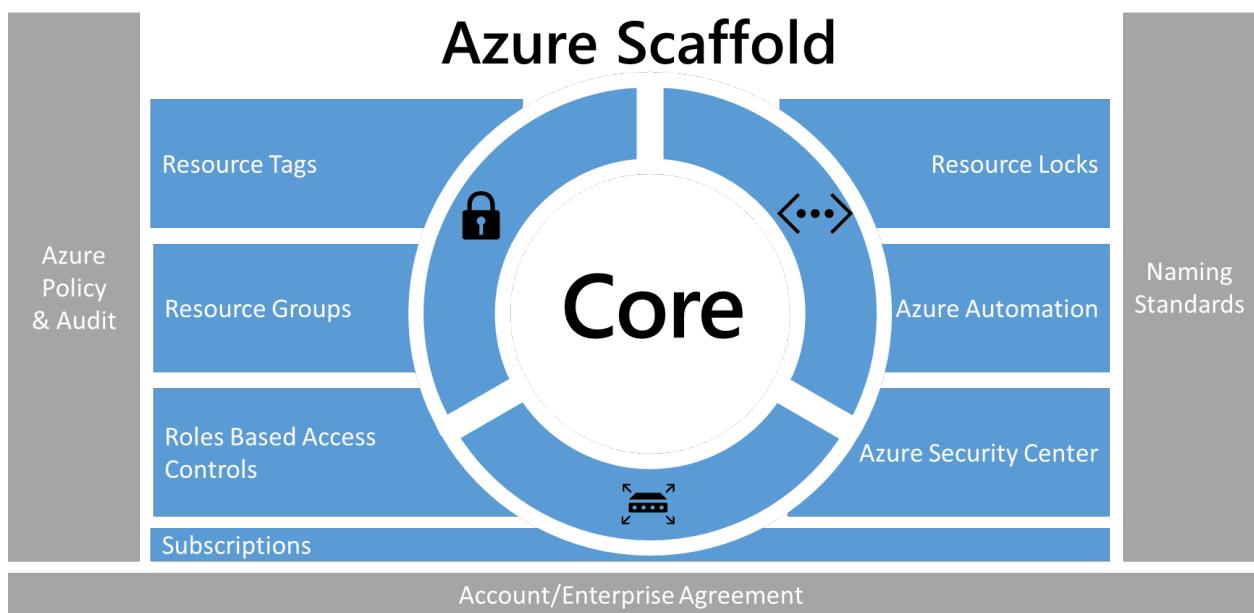
The scaffold is based on practices we have gathered from many engagements with clients of various sizes. Those clients range from small organizations developing solutions in the cloud to Fortune 500 enterprises and independent software vendors who are migrating and developing solutions in the cloud. The enterprise scaffold is "purpose-built" to be flexible to support both traditional IT workloads and agile workloads; such as, developers creating software-as-a-service (SaaS) applications based on Azure capabilities.

The enterprise scaffold is intended to be the foundation of each new subscription within Azure. It enables administrators to ensure workloads meet the minimum governance requirements of an organization without preventing business groups and developers from quickly meeting their own goals.

IMPORTANT

Governance is crucial to the success of Azure. This article targets the technical implementation of an enterprise scaffold but only touches on the broader process and relationships between the components. Policy governance flows from the top down and is determined by what the business wants to achieve. Naturally, the creation of a governance model for Azure includes representatives from IT, but more importantly it should have strong representation from business group leaders, and security and risk management. In the end, an enterprise scaffold is about mitigating business risk to facilitate an organization's mission and objectives.

The following image describes the components of the scaffold. The foundation relies on a solid plan for departments, accounts, and subscriptions. The pillars consist of Resource Manager policies and strong naming standards. The rest of scaffold comes from core Azure capabilities and features that enable a secure and manageable environment.

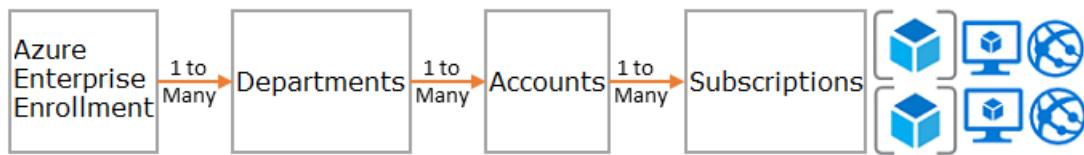


NOTE

Azure has grown rapidly since its introduction in 2008. This growth required Microsoft engineering teams to rethink their approach for managing and deploying services. The Azure Resource Manager model was introduced in 2014 and replaces the classic deployment model. Resource Manager enables organizations to more easily deploy, organize, and control Azure resources. Resource Manager includes parallelization when creating resources for faster deployment of complex, interdependent solutions. It also includes granular access control, and the ability to tag resources with metadata. Microsoft recommends that you create all resources through the Resource Manager model. The enterprise scaffold is explicitly designed for the Resource Manager model.

Define your hierarchy

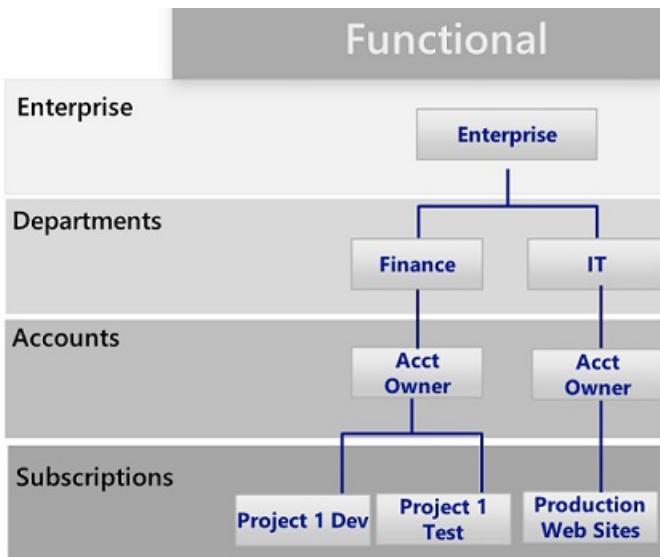
The foundation of the scaffold is the Azure Enterprise Enrollment (and the Enterprise Portal). The enterprise enrollment defines the shape and use of Azure services within a company and is the core governance structure. Within the enterprise agreement, customers are able to further subdivide the environment into departments, accounts, and finally, subscriptions. An Azure subscription is the basic unit where all resources are contained. It also defines several limits within Azure, such as number of cores, resources, etc.



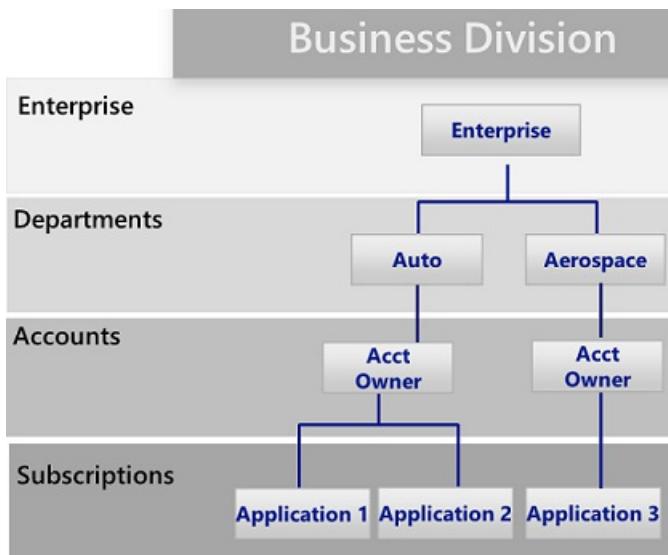
Every enterprise is different and the hierarchy in the previous image allows for significant flexibility in how Azure is organized within the company. Before implementing the guidance contained in this document, you should model your hierarchy and understand the impact on billing, resource access, and complexity.

The three common patterns for Azure Enrollments are:

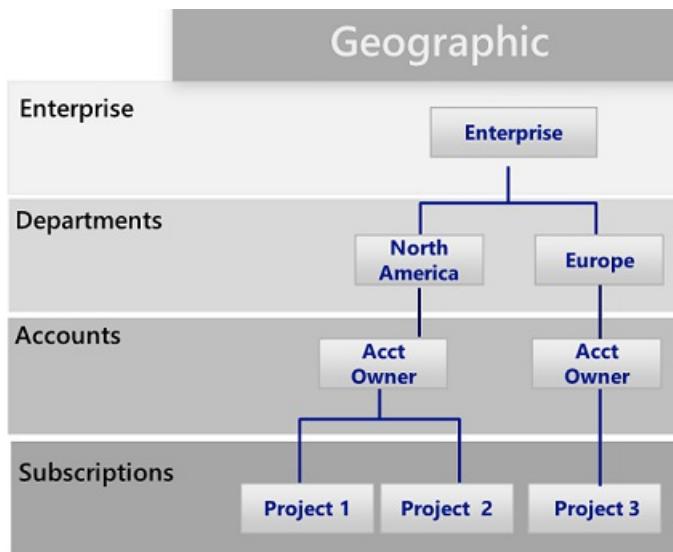
- The **functional** pattern



- The **business unit** pattern



- The **geographic** pattern



You apply the scaffold at the subscription level to extend the governance requirements of the enterprise into the subscription.

Naming standards

The first pillar of the scaffold is naming standards. Well-designed naming standards enable you to identify resources in the portal, on a bill, and within scripts. Most likely, you already have naming standards for on-premise infrastructure. When adding Azure to your environment, you should extend those naming standards to your Azure resources. Naming standard facilitate more efficient management of the environment at all levels.

TIP

For naming conventions:

- Review and adopt where possible the [Patterns and Practices guidance](#). This guidance helps you decide on a meaningful naming standard.
- Use camelCasing for names of resources (such as myResourceGroup and vnetNetworkName). Note: There are certain resources, such as storage accounts, where the only option is to use lower case (and no other special characters).
- Consider using Azure Resource Manager policies (described in the next section) to enforce naming standards.

The preceding tips help you implement a consistent naming convention.

Policies and auditing

The second pillar of the scaffold involves creating [Azure Resource Manager policies](#) and [auditing the activity log](#). Resource Manager policies provide you with the ability to manage risk in Azure. You can define policies that ensure data sovereignty by restricting, enforcing, or auditing certain actions.

- Policy is a default **allow** system. You control actions by defining and assigning policies to resources that deny or audit actions on resources.
- Policies are described by policy definitions in a policy definition language (if-then conditions).
- You create policies with JSON (Javascript Object Notation) formatted files. After defining a policy, you assign it to a particular scope: subscription, resource group, or resource.

Policies have multiple actions that allow for a fine-grained approach to your scenarios. The actions are:

- **Deny:** Blocks the resource request
- **Audit:** Allows the request but adds a line to the activity log (which can be used to provide alerts or to trigger runbooks)

- **Append:** Adds specified information to the resource. For example, if there is not a "CostCenter" tag on a resource, add that tag with a default value.

Common uses of Resource Manager policies

Azure Resource Manager policies are a powerful tool in the Azure toolkit. They enable you to avoid unexpected costs, to identify a cost center for resources through tagging, and to ensure that compliancy requirements are met. When policies are combined with the built-in auditing features, you can fashion complex and flexible solutions. Policies allow companies to provide controls for "Traditional IT" workloads and "Agile" workloads; such as, developing customer applications. The most common patterns we see for policies are:

- **Geo-compliance/data sovereignty** - Azure provides regions across the world. Enterprises often wish to control where resources are created (whether to ensure data sovereignty or just to ensure resources are created close to the end consumers of the resources).
- **Cost management** - An Azure subscription can contain resources of many types and scale. Corporations often wish to ensure that standard subscriptions avoid using unnecessarily large resources, which can cost hundreds of dollars a month or more.
- **Default governance through required tags** - Requiring tags is one of the most common and highly desired features. Using Azure Resource Manager Policies enterprises are able to ensure that a resource is appropriately tagged. The most common tags are: Department, Resource Owner, and Environment type (for example - production, test, development)

Examples

"Traditional IT" subscription for line-of-business applications

- Enforce Department and Owner tags on all resources
- Restrict resource creation to the North American Region
- Restrict the ability to create G-Series VMs and HDInsight Clusters

"Agile" Environment for a business unit creating cloud applications

- To meet data sovereignty requirements, allow the creation of resources ONLY in a specific region.
- Enforce Environment tag on all resources. If a resource is created without a tag, append the **Environment: Unknown** tag to the resource.
- Audit when resources are created outside of North America but do not prevent.
- Audit when high-cost resources are created.

TIP

The most common use of Resource Manager policies across organizations is to control *where* resources can be created and *what* types of resources can be created. In addition to providing controls on *where* and *what*, many enterprises use policies to ensure resources have the appropriate metadata to bill back for consumption. We recommend applying policies at the subscription level for:

- Geo-compliance/data sovereignty
- Cost management
- Required tags (Determined by business need, such as BillTo, Application Owner)

You can apply additional policies at lower levels of scope.

Audit - what happened?

To view how your environment is functioning, you need to audit user activity. Most resource types within Azure create diagnostic logs that you can analyze through a log tool or in Azure Operations Management Suite. You can gather activity logs across multiple subscriptions to provide a departmental or enterprise view. Audit records are both an important diagnostic tool and a crucial mechanism to trigger events in the Azure environment.

Activity logs from Resource Manager deployments enable you to determine the **operations** that took place and who performed them. Activity logs can be collected and aggregated using tools like Log Analytics.

Resource tags

As users in your organization add resources to the subscription, it becomes increasingly important to associate resources with the appropriate department, customer, and environment. You can attach metadata to resources through [tags](#). You use tags to provide information about the resource or the owner. Tags enable you to not only aggregate and group resources in various ways, but use that data for the purposes of chargeback. You can tag resources with up to 15 key:value pairs.

Resource tags are flexible and should be attached to most resources. Examples of common resource tags are:

- BillTo
- Department (or Business Unit)
- Environment (Production, Stage, Development)
- Tier (Web Tier, Application Tier)
- Application Owner
- ProjectName



For more examples of tags, see [Recommended naming conventions for Azure resources](#).

TIP

Consider making a policy that mandates tagging for:

- Resource groups
- Storage
- Virtual Machines
- Application Service Environments/web servers

This tagging strategy identifies across your subscriptions what metadata is needed for the business, finance, security, risk management, and overall management of the environment.

Resource group

Resource Manager enables you to put resources into meaningful groups for management, billing, or natural affinity. As mentioned earlier, Azure has two deployment models. In the earlier Classic model, the basic unit of management was the subscription. It was difficult to break down resources within a subscription, which led to the creation of large numbers of subscriptions. With the Resource Manager model, we saw the introduction of resource groups. Resource groups are containers of resources that have a common lifecycle or share an attribute such as "all SQL servers" or "Application A".

Resource groups cannot be contained within each other and resources can only belong to one resource group.

You can apply certain actions on all resources in a resource group. For example, deleting a resource group removes all resources within the resource group. Typically, you place an entire application or related system in the same resource group. For example, a three tier application called Contoso Web Application would contain the web server, application server and SQL server in the same resource group.

TIP

How you organize your resource groups may vary from "Traditional IT" workloads to "Agile IT" workloads:

- "Traditional IT" workloads are most commonly grouped by items within the same lifecycle, such as an application. Grouping by application allows for individual application management.
- "Agile IT" workloads tend to focus on external customer-facing cloud applications. The resource groups should reflect the layers of deployment (such as Web Tier, App Tier) and management.

Understanding your workload helps you develop a resource group strategy.

Role-based access control

You probably are asking yourself "who should have access to resources?" and "how do I control this access?" Allowing or disallowing access to the Azure portal, and controlling access to resources in the portal is crucial.

When Azure was initially released, access controls to a subscription were basic: Administrator or Co-Administrator. Access to a subscription in the Classic model implied access to all the resources in the portal. This lack of fine-grained control led to the proliferation of subscriptions to provide a level of reasonable access control for an Azure Enrollment.

This proliferation of subscriptions is no longer needed. With role-based access control, you can assign users to standard roles (such as common "reader" and "writer" types of roles). You can also define custom roles.

TIP

To implement role-based access control:

- Connect your corporate identity store (most commonly Active Directory) to Azure Active Directory using the AD Connect tool.
- Control the Admin/Co-Admin of a subscription using a managed identity. **Don't** assign Admin/Co-admin to a new subscription owner. Instead, use RBAC roles to provide **Owner** rights to a group or individual.
- Add Azure users to a group (for example, Application X Owners) in Active Directory. Use the synced group to provide group members the appropriate rights to manage the resource group containing the application.
- Follow the principle of granting the **least privilege** required to do the expected work. For example:
 - Deployment Group: A group that is only able to deploy resources.
 - Virtual Machine Management: A group that is able to restart VMs (for operations)

These tips help you manage user access across your subscription.

Azure resource locks

As your organization adds core services to the subscription, it becomes increasingly important to ensure that those services are available to avoid business disruption. **Resource locks** enable you to restrict operations on high-value resources where modifying or deleting them would have a significant impact on your applications or cloud infrastructure. You can apply locks to a subscription, resource group, or resource. Typically, you apply locks to foundational resources such as virtual networks, gateways, and storage accounts.

Resource locks currently support two values: CanNotDelete and ReadOnly. CanNotDelete means that users (with the appropriate rights) can still read or modify a resource but cannot delete it. ReadOnly means that authorized

users can't delete or modify a resource.

To create or delete management locks, you must have access to `Microsoft.Authorization/*` or `Microsoft.Authorization/locks/*` actions. Of the built-in roles, only Owner and User Access Administrator are granted those actions.

TIP

Core network options should be protected with locks. Accidental deletion of a gateway, site-to-site VPN would be disastrous to an Azure subscription. Azure doesn't allow you to delete a virtual network that is in use, but applying more restrictions is a helpful precaution.

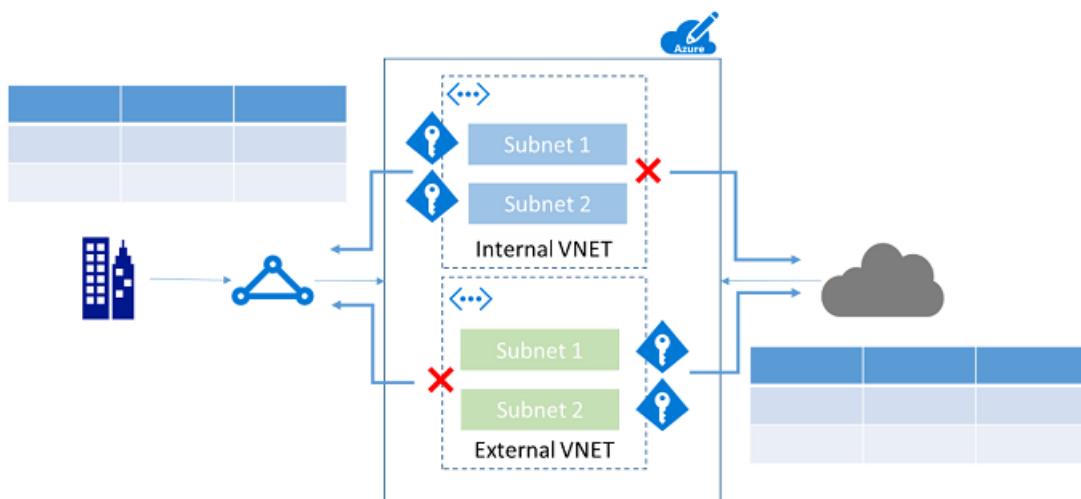
- Virtual Network: CanNotDelete
- Network Security Group: CanNotDelete
- Policies: CanNotDelete

Policies are also crucial to the maintenance of appropriate controls. We recommend that you apply a **CanNotDelete** lock to policies that are in use.

Core networking resources

Access to resources can be either internal (within the corporation's network) or external (through the internet). It is easy for users in your organization to inadvertently put resources in the wrong spot, and potentially open them to malicious access. As with on-premise devices, enterprises must add appropriate controls to ensure that Azure users make the right decisions. For subscription governance, we identify core resources that provide basic control of access. The core resources consist of:

- **Virtual networks** are container objects for subnets. Though not strictly necessary, it is often used when connecting applications to internal corporate resources.
- **Network security groups** are similar to a firewall and provide rules for how a resource can "talk" over the network. They provide granular control over how/if a subnet (or virtual machine) can connect to the Internet or other subnets in the same virtual network.



TIP

For networking:

- Create virtual networks dedicated to external-facing workloads and internal-facing workloads. This approach reduces the chance of inadvertently placing virtual machines that are intended for internal workloads in an external facing space.
- Configure network security groups to limit access. At a minimum, block access to the internet from internal virtual networks, and block access to the corporate network from external virtual networks.

These tips help you implement secure networking resources.

Automation

Managing resources individually is both time-consuming and potentially error prone for certain operations.

Azure provides various automation capabilities including Azure Automation, Logic Apps, and Azure Functions.

[Azure Automation](#) enables administrators to create and define runbooks to handle common tasks in managing resources. You create runbooks by using either a PowerShell code editor or a graphical editor. You can produce complex multi-stage workflows. Azure Automation is often used to handle common tasks such as shutting down unused resources, or creating resources in response to a specific trigger without needing human intervention.

TIP

For automation:

- Create an Azure Automation account and review the available runbooks (both graphical and command line) available in the [Runbook Gallery](#).
- Import and customize key runbooks for your own use.

A common scenario is the ability to Start/Shutdown virtual machines on a schedule. There are example runbooks that are available in the Gallery that both handle this scenario and teach you how to expand it.

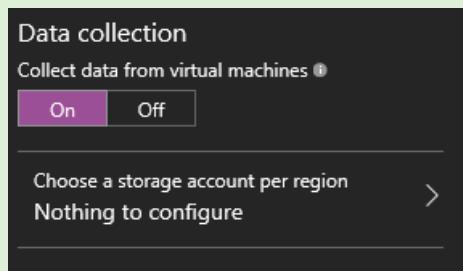
Azure Security Center

Perhaps one of the biggest blockers to cloud adoption has been the concerns over security. IT risk managers and security departments need to ensure that resources in Azure are secure.

The [Azure Security Center](#) provides a central view of the security status of resources in the subscriptions, and provides recommendations that help prevent compromised resources. It can enable more granular policies (for example, applying policies to specific resource groups that allow the enterprise to tailor their posture to the risk they are addressing). Finally, Azure Security Center is an open platform that enables Microsoft partners and independent software vendors to create software that plugs into Azure Security Center to enhance its capabilities.

TIP

Azure Security Center is enabled by default in each subscription. However, you must enable data collection from virtual machines to allow Azure Security Center to install its agent and begin gathering data.



Next steps

- Now that you have learned about subscription governance, it's time to see these recommendations in practice. See [Examples of implementing Azure subscription governance](#).

Examples of implementing Azure enterprise scaffold

1/25/2017 • 9 min to read • [Edit Online](#)

This topic provides examples of how an enterprise can implement the recommendations for an [Azure enterprise scaffold](#). It uses a fictional company named Contoso to illustrate best practices for common scenarios.

Background

Contoso is a worldwide company that provides supply chain solutions for customers in everything from a "Software as a Service" model to a packaged model deployed on-premises. They develop software across the globe with significant development centers in India, the United States, and Canada.

The ISV portion of the company is divided into several independent business units that manage products in a significant business. Each business unit has its own developers, product managers, and architects.

The Enterprise Technology Services (ETS) business unit provides centralized IT capability, and manages several data centers where business units host their applications. Along with managing the data centers, the ETS organization provides and manages centralized collaboration (such as email and websites) and network/telephony services. They also manage customer-facing workloads for smaller business units who don't have operational staff.

The following personas are used in this topic:

- Dave is the ETS Azure administrator.
- Alice is Contoso's Director of Development in the supply chain business unit.

Contoso needs to build a line-of-business app and a customer-facing app. It has decided to run the apps on Azure. Dave reads the [prescriptive subscription governance](#) topic, and is now ready to implement the recommendations.

Scenario 1: line-of-business application

Contoso is building a source code management system (BitBucket) to be used by developers across the world. The application uses Infrastructure as a Service (IaaS) for hosting, and consists of web servers and a database server. Developers access servers in their development environments, but they don't need access to the servers in Azure. Contoso ETS wishes to allow the application owner and team to manage the application. The application is only available while on Contoso's corporate network. Dave needs to set up the subscription for this application. The subscription will also host other developer-related software in the future.

Naming standards & resource groups

Dave creates a subscription to support developer tools that are common across all the business units. He needs to create meaningful names for the subscription and resource groups (for the application and the networks). He creates the following subscription and resource groups:

ITEM	NAME	DESCRIPTION
Subscription	Contoso ETS DeveloperTools Production	Supports common developer tools
Resource Group	rgBitBucket	Contains the application web server and database server

ITEM	NAME	DESCRIPTION
Resource Group	rgCoreNetworks	Contains the virtual networks and site-to-site gateway connection

Role-based access control

After creating his subscription, Dave wants to ensure that the appropriate teams and application owners can access their resources. Dave recognizes that each team has different requirements. He utilizes the groups that have been synced from Contoso's on-premises Active Directory (AD) to Azure Active Directory, and provides the right level of access to the teams.

Dave assigns the following roles for the subscription:

ROLE	ASSIGNED TO	DESCRIPTION
Owner	Managed ID from Contoso's AD	This ID is controlled with Just in Time (JIT) access through Contoso's Identity Management tool and ensures that subscription owner access is fully audited.
Security Manager	Security and risk management department	This role allows users to look at the Azure Security Center and the status of the resources.
Network Contributor	Network team	This role allows Contoso's network team to manage the Site to Site VPN and the Virtual Networks.
Custom role	Application owner	Dave creates a role that grants the ability to modify resources within the resource group. For more information, see Custom Roles in Azure RBAC

Policies

Dave has the following requirements for managing resources in the subscription:

- Because the development tools support developers across the world, he doesn't want to block users from creating resources in any region. However, he needs to know where resources are created.
- He is concerned with costs. Therefore, he wants to prevent application owners from creating unnecessarily expensive virtual machines.
- Because this application serves developers in many business units, he wants to tag each resource with the business unit and application owner. By using these tags, ETS can bill the appropriate teams.

He creates the following [Resource Manager policies](#):

FIELD	EFFECT	DESCRIPTION
location	audit	Audit the creation of the resources in any region
type	deny	Deny creation of G-Series virtual machines
tags	deny	Require application owner tag

FIELD	EFFECT	DESCRIPTION
tags	deny	Require cost center tag
tags	append	Append tag name BusinessUnit and tag value ETS to all resources

Resource tags

Dave understands that he needs to have specific information on the bill to identify the cost center for the BitBucket implementation. Additionally, Dave wants to know all the resources that ETS owns.

He adds the following [tags](#) to the resource groups and resources.

TAG NAME	TAG VALUE
ApplicationOwner	The name of the person who manages this application.
CostCenter	The cost center of the group that is paying for the Azure consumption.
BusinessUnit	ETS (the business unit associated with the subscription)

Core network

The Contoso ETS information security and risk management team reviews Dave's proposed plan to move the application to Azure. They want to ensure that the application is not exposed to the internet. Dave also has developer apps that in the future will be moved to Azure. These apps require public interfaces. To meet these requirements, he provides both internal and external virtual networks, and a network security group to restrict access.

He creates the following resources:

RESOURCE TYPE	NAME	DESCRIPTION
Virtual Network	vnInternal	Used with the BitBucket application and is connected via ExpressRoute to Contoso's corporate network. A subnet (sbBitBucket) provides the application with a specific IP address space.
Virtual Network	vnExternal	Available for future applications that require public-facing endpoints.
Network Security Group	nsgBitBucket	Ensures that the attack surface of this workload is minimized by allowing connections only on port 443 for the subnet where the application lives (sbBitBucket).

Resource locks

Dave recognizes that the connectivity from Contoso's corporate network to the internal virtual network must be protected from any wayward script or accidental deletion.

He creates the following [resource lock](#):

LOCK TYPE	RESOURCE	DESCRIPTION
CanNotDelete	vnlInternal	Prevents users from deleting the virtual network or subnets, but does not prevent the addition of new subnets.

Azure Automation

Dave has nothing to automate for this application. Although he created an Azure Automation account, he won't initially use it.

Azure Security Center

Contoso IT service management needs to quickly identify and handle threats. They also want to understand what problems may exist.

To fulfill these requirements, Dave enables the [Azure Security Center](#), and provides access to the Security Manager role.

Scenario 2: customer-facing app

The business leadership in the supply chain business unit has identified various opportunities to increase engagement with Contoso's customers by using a loyalty card. Alice's team must create this application and decides that Azure increases their ability to meet the business need. Alice works with Dave from ETS to configure two subscriptions for developing and operating this application.

Azure subscriptions

Dave logs in to the Azure Enterprise Portal and sees that the supply chain department already exists. However, as this project is the first development project for the supply chain team in Azure, Dave recognizes the need for a new account for Alice's development team. He creates the "R&D" account for her team and assigns access to Alice. Alice logs in via the Azure portal and creates two subscriptions: one to hold the development servers and one to hold the production servers. She follows the previously established naming standards when creating the following subscriptions:

SUBSCRIPTION USE	NAME
Development	SupplyChain ResearchDevelopment LoyaltyCard Development
Production	SupplyChain Operations LoyaltyCard Production

Policies

Dave and Alice discuss the application and identify that this application only serves customers in the North American region. Alice and her team plan to use Azure's Application Service Environment and Azure SQL to create the application. They may need to create virtual machines during development. Alice wants to ensure that her developers have the resources they need to explore and examine problems without pulling in ETS.

For the **development subscription**, they create the following policy:

FIELD	EFFECT	DESCRIPTION
location	audit	Audit the creation of the resources in any region.

They do not limit the type of sku a user can create in development, and they do not require tags for any resource groups or resources.

For the **production subscription**, they create the following policies:

FIELD	EFFECT	DESCRIPTION
location	deny	Deny the creation of any resources outside of the US data centers.
tags	deny	Require application owner tag
tags	deny	Require department tag.
tags	append	Append tag to each resource group that indicates production environment.

They do not limit the type of sku a user can create in production.

Resource tags

Dave understands that he needs to have specific information to identify the correct business groups for billing and ownership. He defines resource tags for resource groups and resources.

TAG NAME	TAG VALUE
ApplicationOwner	The name of the person who manages this application.
Department	The cost center of the group that is paying for the Azure consumption.
EnvironmentType	Production (Even though the subscription includes Production in the name, including this tag enables easy identification when looking at resources in the portal or on the bill.)

Core networks

The Contoso ETS information security and risk management team reviews Dave's proposed plan to move the application to Azure. They want to ensure that the Loyalty Card application is properly isolated and protected in a DMZ network. To fulfill this requirement, Dave and Alice create an external virtual network and a network security group to isolate the Loyalty Card application from the Contoso corporate network.

For the **development subscription**, they create:

RESOURCE TYPE	NAME	DESCRIPTION
Virtual Network	vnlInternal	Serves the Contoso Loyalty Card development environment and is connected via ExpressRoute to Contoso's corporate network.

For the **production subscription**, they create:

RESOURCE TYPE	NAME	DESCRIPTION

RESOURCE TYPE	NAME	DESCRIPTION
Virtual Network	vnExternal	Hosts the Loyalty Card application and is not connected directly to Contoso's ExpressRoute. Code is pushed via their Source Code system directly to the PaaS services.
Network Security Group	nsgBitBucket	Ensures that the attack surface of this workload is minimized by only allowing in-bound communication on TCP 443. Contoso is also investigating using a Web Application Firewall for additional protection.

Resource locks

Dave and Alice confer and decide to add resource locks on some of the key resources in the environment to prevent accidental deletion during an errant code push.

They create the following lock:

LOCK TYPE	RESOURCE	DESCRIPTION
CanNotDelete	vnExternal	To prevent people from deleting the virtual network or subnets. The lock does not prevent the addition of new subnets.

Azure Automation

Alice and her development team have extensive runbooks to manage the environment for this application. The runbooks allow for the addition/deletion of nodes for the application and other DevOps tasks.

To use these runbooks, they enable [Automation](#).

Azure Security Center

Contoso IT service management needs to quickly identify and handle threats. They also want to understand what problems may exist.

To fulfill these requirements, Dave enables Azure Security Center. He ensures that the Azure Security Center is monitoring the resources, and provides access to the DevOps and security teams.

Next steps

- To learn about creating Resource Manager templates, see [Best practices for creating Azure Resource Manager templates](#).

Export an Azure Resource Manager template from existing resources

3/31/2017 • 12 min to read • [Edit Online](#)

Resource Manager enables you to export a Resource Manager template from existing resources in your subscription. You can use that generated template to learn about the template syntax or to automate the redeployment of your solution as needed.

It is important to note that there are two different ways to export a template:

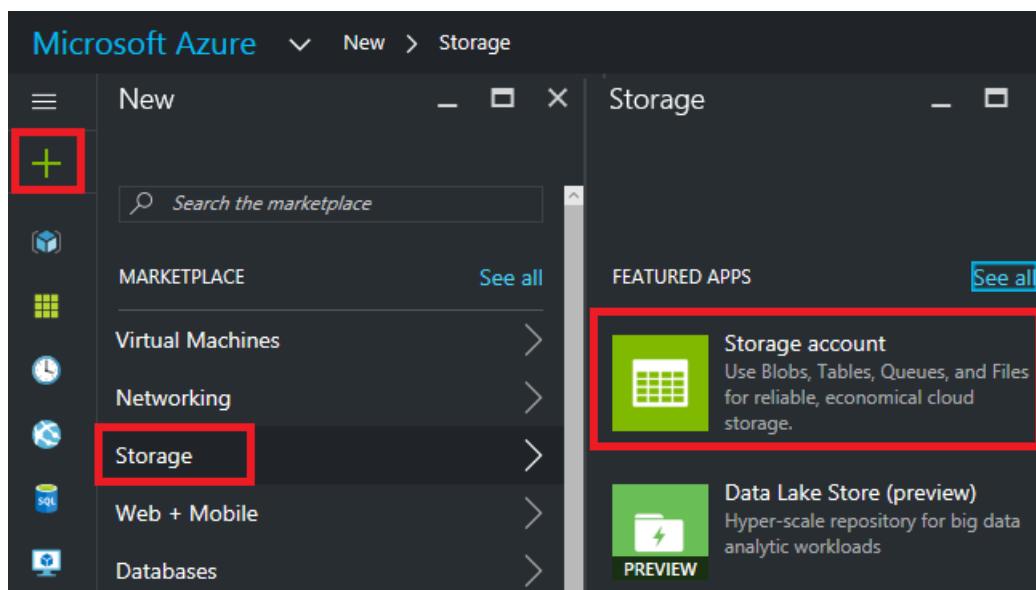
- You can export the actual template that you used for a deployment. The exported template includes all the parameters and variables exactly as they appeared in the original template. This approach is helpful when you have deployed resources through the portal. Now, you want to see how to construct the template to create those resources.
- You can export a template that represents the current state of the resource group. The exported template is not based on any template that you used for deployment. Instead, it creates a template that is a snapshot of the resource group. The exported template has many hard-coded values and probably not as many parameters as you would typically define. This approach is useful when you have modified the resource group through the portal or scripts. Now, you need to capture the resource group as a template.

This topic shows both approaches.

In this tutorial, you sign in to the Azure portal, create a storage account, and export the template for that storage account. You add a virtual network to modify the resource group. Finally, you export a new template that represents its current state. Although this article focuses on a simplified infrastructure, you could use these same steps to export a template for a more complicated solution.

Create a storage account

1. In the [Azure portal](#), select **New > Storage > Storage account**.



2. Create a storage account with the name **storage**, your initials, and the date. The storage account name must be unique across Azure. If the name is already in use, you see an error message indicating the name is in use. Try a variation. For resource group, select **Create new** and name it **ExportGroup**. You can use the

default values for the other properties. Select **Create**.

The screenshot shows the 'Create storage account' dialog box. It includes fields for Name (storagetc03032017.core.windows.net), Deployment model (Resource manager), Account kind (General purpose), Performance (Standard), Replication (Read-access geo-redundant storage (RA...)), Storage service encryption (Disabled), Subscription (Visual Studio Enterprise), Resource group (Create new - ExportGroup), Location (South Central US), and a 'Pin to dashboard' checkbox. At the bottom are 'Create' and 'Automation options' buttons.

The deployment may take a minute. After the deployment finishes, your subscription contains the storage account.

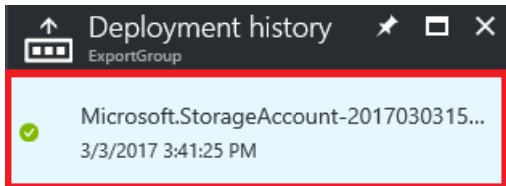
View a template from deployment history

1. Go to the resource group blade for your new resource group. Notice that the blade shows the result of the last deployment. Select this link.

The screenshot shows the 'ExportGroup' Resource group blade. On the left, there's a navigation menu with 'Overview' (selected), 'Activity log', 'Access control (IAM)', and 'Tags'. On the right, there's a table header with 'Add', 'Columns', and 'Delete' buttons. Below it, the 'Essentials' section shows 'Subscription name (change) Visual Studio Enterprise'. Under 'Deployments', it says '1 Succeeded'. A red box highlights the '1 Succeeded' text. At the bottom is a 'Filter by name...' input field.

2. You see a history of deployments for the group. In your case, the blade probably lists only one deployment.

Select this deployment.



3. The blade displays a summary of the deployment. The summary includes the status of the deployment and its operations and the values that you provided for parameters. To see the template that you used for the deployment, select **View template**.

A screenshot of the 'Microsoft.StorageAccount-20170303154053' deployment details blade. The 'View template' button is highlighted with a red box. The blade contains sections for Summary, Outputs, Inputs, and Operation details. In the 'Inputs' section, the 'NAME' field is set to 'storagetc03032017'. The 'Operation details' table shows one operation: 'storagetc03032017' of type 'Microsoft.Storage/s...' with status 'OK' and timestamp '2017-03-03T23:41:2...'.

RESOURCE	TYPE	STATUS	TIMESTAMP
storagetc03032017	Microsoft.Storage/s...	OK	2017-03-03T23:41:2...

4. Resource Manager retrieves the following seven files for you:

- a. **Template** - The template that defines the infrastructure for your solution. When you created the storage account through the portal, Resource Manager used a template to deploy it and saved that template for future reference.
- b. **Parameters** - A parameter file that you can use to pass in values during deployment. It contains the values that you provided during the first deployment, but you can change any of these values when you redeploy the template.
- c. **CLI** - An Azure command-line-interface (CLI) script file that you can use to deploy the template.

- d. **CLI 2.0** - An Azure command-line-interface (CLI) script file that you can use to deploy the template.
- e. **PowerShell** - An Azure PowerShell script file that you can use to deploy the template.
- f. **.NET** - A .NET class that you can use to deploy the template.
- g. **Ruby** - A Ruby class that you can use to deploy the template.

The files are available through links across the blade. By default, the blade displays the template.

The screenshot shows the Azure Resource Manager template blade for a storage account. At the top, there are buttons for 'Download', 'Add to library', and 'Deploy'. Below these are tabs: 'Template' (which is selected and highlighted with a red box), 'Parameters', 'CLI', 'CLI 2.0 (Preview)', 'PowerShell', '.NET', and 'Ruby'. On the left, there's a sidebar with icons for 'Parameters (4)', 'Variables (0)', and 'Resources (1)'. Under 'Resources (1)', it lists '[parameters('name')] (Microsoft.St...'. The main area on the right displays the JSON template code:

```
1 {
2   "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "name": {
6       "type": "String"
7     },
8     "accountType": {
9       "type": "String"
10    }
11 }
```

Let's pay particular attention to the template. Your template should look similar to:

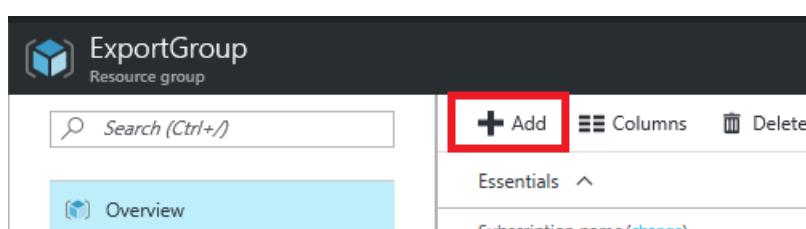
```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "name": {
            "type": "String"
        },
        "accountType": {
            "type": "String"
        },
        "location": {
            "type": "String"
        },
        "encryptionEnabled": {
            "defaultValue": false,
            "type": "Bool"
        }
    },
    "resources": [
        {
            "type": "Microsoft.Storage/storageAccounts",
            "sku": {
                "name": "[parameters('accountType')]"
            },
            "kind": "Storage",
            "name": "[parameters('name')]",
            "apiVersion": "2016-01-01",
            "location": "[parameters('location')]",
            "properties": {
                "encryption": {
                    "services": {
                        "blob": {
                            "enabled": "[parameters('encryptionEnabled')]"
                        }
                    },
                    "keySource": "Microsoft.Storage"
                }
            }
        }
    ]
}
```

This template is the actual template used to create your storage account. Notice it contains parameters that enable you to deploy different types of storage accounts. To learn more about the structure of a template, see [Authoring Azure Resource Manager templates](#). For the complete list of the functions you can use in a template, see [Azure Resource Manager template functions](#).

Add a virtual network

The template that you downloaded in the previous section represented the infrastructure for that original deployment. However, it will not account for any changes you make after the deployment. To illustrate this issue, let's modify the resource group by adding a virtual network through the portal.

1. In the resource group blade, select **Add**.



2. Select **Virtual network** from the available resources.

The screenshot shows the Azure portal's search interface with the title "Everything". A search bar at the top contains the placeholder "Search Everything". Below the search bar, there are four items in a row, each with a small icon and the text "PREVIEW" below it: "Web App" (Microsoft), "API App" (Microsoft), "Windows Server 2012 R2" (Microsoft), and "Team Project (preview)" (Microsoft). Under the heading "Recommended for you", there are four more items: "Automation" (Microsoft), "Logic App (preview)" (Microsoft), "Web App" (Microsoft), and "Virtual network" (Microsoft). The "Virtual network" item is highlighted with a red rectangular border around its icon and text.

3. Name your virtual network **VNET**, and use the default values for the other properties. Select **Create**.

Create virtual network

* Name
VNET

* Address space ⓘ
10.0.0.0/16
10.0.0.0 - 10.0.255.255 (65536 addresses)

* Subnet name
default

* Subnet address range ⓘ
10.0.0.0/24
10.0.0.0 - 10.0.0.255 (256 addresses)

* Subscription
Windows Azure MSDN - Visual Studio Ultir

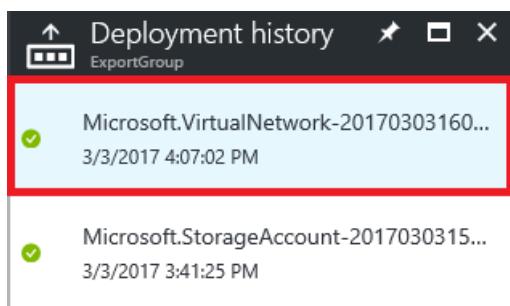
* Resource group
ExportGroup

* Location
North Europe

Pin to dashboard

Create

4. After the virtual network has successfully deployed to your resource group, look again at the deployment history. You now see two deployments. If you do not see the second deployment, you may need to close your resource group blade and reopen it. Select the more recent deployment.



5. View the template for that deployment. Notice that it defines only the virtual network. It does not include the storage account you deployed earlier. You no longer have a template that represents all the resources in your resource group.

Export the template from resource group

To get the current state of your resource group, export a template that shows a snapshot of the resource group.

NOTE

You cannot export a template for a resource group that has more than 200 resources.

- To view the template for a resource group, select **Automation script**.

The screenshot shows the Azure portal interface for a resource group named 'ExportGroup'. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Quickstart, Resource costs, Deployments, Properties, Locks, and Automation script. The 'Automation script' option is highlighted with a red box. The main content area shows deployment details: 'Subscription name (change) Visual Studio Enterprise', 'Deployments 2 Succeeded', and a list of two items: 'storagetcf03032017' and 'VNET'. A 'Filter by name...' search bar is also present.

Not all resource types support the export template function. If your resource group only contains the storage account and virtual network shown in this article, you do not see an error. However, if you have created other resource types, you may see an error stating that there is a problem with the export. You learn how to handle those issues in the [Fix export issues](#) section.

- You again see the six files that you can use to redeploy the solution, but this time the template is a little different. This template has only two parameters: one for the storage account name, and one for the virtual network name.

```
"parameters": {
    "virtualNetworks_VNET_name": {
        "defaultValue": "VNET",
        "type": "String"
    },
    "storageAccounts_storagetcf05092016_name": {
        "defaultValue": "storagetcf05092016",
        "type": "String"
    }
},
```

Resource Manager did not retrieve the templates that you used during deployment. Instead, it generated a new template that's based on the current configuration of the resources. For example, the template sets the storage account location and replication value to:

```
"location": "northeurope",
"tags": {},
"properties": {
    "accountType": "Standard_RAGRS"
},
```

- You have a couple of options for continuing to work with this template. You can either download the template and work on it locally with a JSON editor. Or, you can save the template to your library and work

on it through the portal.

If you are comfortable using a JSON editor like [VS Code](#) or [Visual Studio](#), you might prefer downloading the template locally and using that editor. If you are not set up with a JSON editor, you might prefer editing the template through the portal. The remainder of this topic assumes you have saved the template to your library in the portal. However, you make the same syntax changes to the template whether working locally with a JSON editor or through the portal.

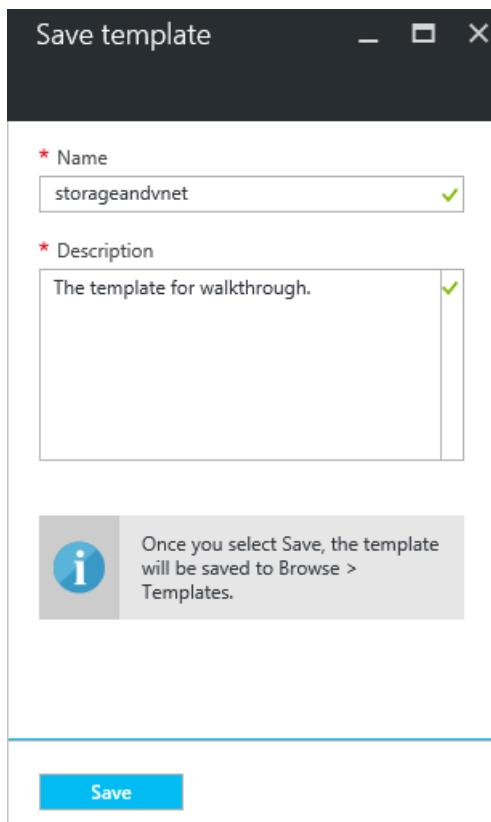
To work locally, select **Download**.



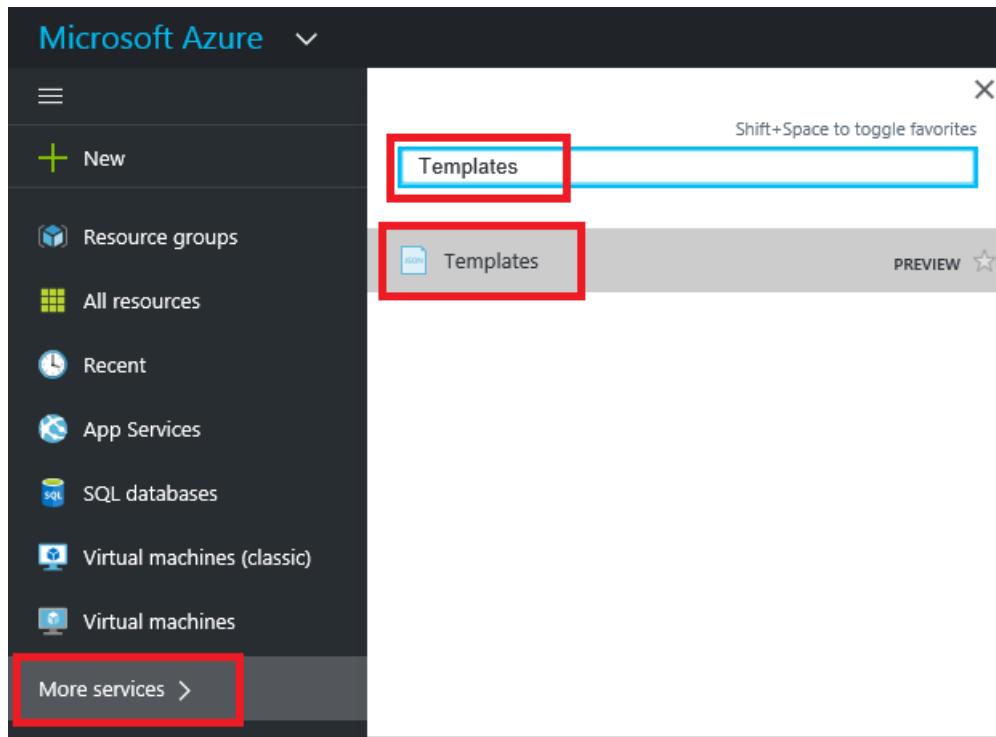
To work through the portal, select **Add to library**.



When adding a template to the library, give the template a name and description. Then, select **Save**.



4. To view a template saved in your library, select **More services**, type **Templates** to filter results, select **Templates**.



5. Select the template with the name you saved.

A screenshot of the 'Templates' blade in the Azure portal. The title bar says 'Templates'. Below it are buttons for '+ Add', 'Columns', and 'Refresh'. The 'Directory:' section shows a dropdown labeled 'Switch directories' with a 'Filter items...' input field. The main area is titled 'NAME' and lists three templates:

- myvmttemplate (JSON)
- simpleweb (JSON)
- storageandvnet (JSON)

Customize the template

The exported template works fine if you want to create the same storage account and virtual network for every deployment. However, Resource Manager provides options so that you can deploy templates with a lot more flexibility. For example, during deployment, you might want to specify the type of storage account to create or the values to use for the virtual network address prefix and subnet prefix.

In this section, you add parameters to the exported template so that you can reuse the template when you deploy these resources to other environments. You also add some features to your template to decrease the likelihood of encountering an error when you deploy your template. You no longer have to guess a unique name for your storage account. Instead, the template creates a unique name. You restrict the values that can be specified for the storage account type to only valid options.

1. To customize the template, select **Edit**.

The screenshot shows the 'View Template' page for a storage account template. The template details are as follows:

- DESCRIPTION:** for demo
- PUBLISHER:** (empty)
- MODIFIED:** 10/20/2016

The JSON code for the template is displayed on the right:

```
1 {
2   "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "virtualNetworks_VNET_name": {
6       "defaultValue": "VNET",
7       "type": "String"
8     },
9     "storageAccounts_storagetc1020_name": {
10      "defaultValue": "storagetc1020",
11      "type": "String"
12    }
}
```

2. Select the template.

The screenshot shows the 'Edit Template' preview window. It contains two main sections:

- General:** storageandvnet
- ARM Template:** Template added

3. To be able to pass the values that you might want to specify during deployment, replace the **parameters** section with new parameter definitions. Notice the values of **allowedValues** for **storageAccount_accountType**. If you accidentally provide an invalid value, that error is recognized before the deployment starts. Also, notice that you are providing only a prefix for the storage account name, and the prefix is limited to 11 characters. When you limit the prefix to 11 characters, you ensure that the complete name does not exceed the maximum number of characters for a storage account. The prefix enables you to apply a naming convention to your storage accounts. You will see how to create a unique name in the next step.

```

"parameters": {
    "storageAccount_prefix": {
        "type": "string",
        "maxLength": 11
    },
    "storageAccount_accountType": {
        "defaultValue": "Standard_RAGRS",
        "type": "string",
        "allowedValues": [
            "Standard_LRS",
            "Standard_ZRS",
            "Standard_GRS",
            "Standard_RAGRS",
            "Premium_LRS"
        ]
    },
    "virtualNetwork_name": {
        "type": "string"
    },
    "addressPrefix": {
        "defaultValue": "10.0.0.0/16",
        "type": "string"
    },
    "subnetName": {
        "defaultValue": "subnet-1",
        "type": "string"
    },
    "subnetAddressPrefix": {
        "defaultValue": "10.0.0.0/24",
        "type": "string"
    }
},

```

- The **variables** section of your template is currently empty. In the **variables** section, you create values that simplify the syntax for the rest of your template. Replace this section with a new variable definition. The **storageAccount_name** variable concatenates the prefix from the parameter to a unique string that is generated based on the identifier of the resource group. You no longer have to guess a unique name when providing a parameter value.

```

"variables": {
    "storageAccount_name": "[concat(parameters('storageAccount_prefix'),
uniqueString(resourceGroup().id))]"
},

```

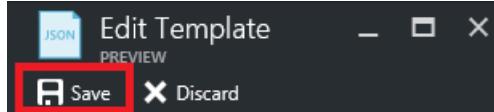
- To use the parameters and variable in the resource definitions, replace the **resources** section with new resource definitions. Notice that little has changed in the resource definitions other than the value that's assigned to the resource property. The properties are the same as the properties from the exported template. You are simply assigning properties to parameter values instead of hard-coded values. The location of the resources is set to use the same location as the resource group through the **resourceGroup().location** expression. The variable that you created for the storage account name is referenced through the **variables** expression.

```

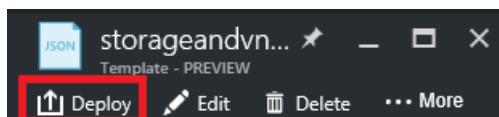
"resources": [
    {
        "type": "Microsoft.Network/virtualNetworks",
        "name": "[parameters('virtualNetwork_name')]",
        "apiVersion": "2015-06-15",
        "location": "[resourceGroup().location]",
        "properties": {
            "addressSpace": {
                "addressPrefixes": [
                    "[parameters('addressPrefix')]"
                ]
            },
            "subnets": [
                {
                    "name": "[parameters('subnetName')]",
                    "properties": {
                        "addressPrefix": "[parameters('subnetAddressPrefix')]"
                    }
                }
            ]
        },
        "dependsOn": []
    },
    {
        "type": "Microsoft.Storage/storageAccounts",
        "name": "[variables('storageAccount_name')]",
        "apiVersion": "2015-06-15",
        "location": "[resourceGroup().location]",
        "tags": {},
        "properties": {
            "accountType": "[parameters('storageAccount_accountType')]"
        },
        "dependsOn": []
    }
]

```

6. Select **OK** when you are done editing the template.
7. Select **Save** to save the changes to the template.



8. To deploy the updated template, select **Deploy**.



9. Provide parameter values, and select a new resource group to deploy the resources to.

Update the downloaded parameters file

If you are working with the downloaded files (rather than the portal library), you need to update the downloaded parameter file. It no longer matches the parameters in your template. You do not have to use a parameter file, but it can simplify the process when you redeploy an environment. You use the default values that are defined in the template for many of the parameters so that your parameter file only needs two values.

Replace the contents of the parameters.json file with:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccount_prefix": {
      "value": "storage"
    },
    "virtualNetwork_name": {
      "value": "VNET"
    }
  }
}
```

The updated parameter file provides values only for parameters that do not have a default value. You can provide values for the other parameters when you want a value that is different from the default value.

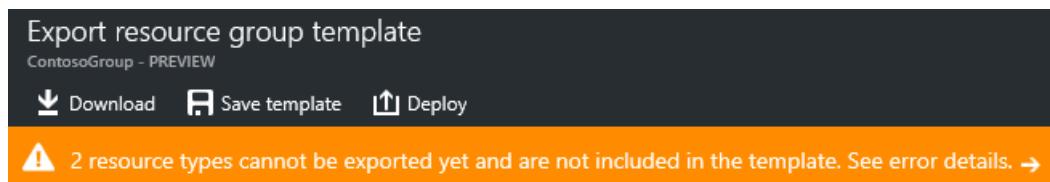
Fix export issues

Not all resource types support the export template function. Resource Manager specifically does not export some resource types to prevent exposing sensitive data. For example, if you have a connection string in your site config, you probably do not want it explicitly displayed in an exported template. To resolve this issue, manually add the missing resources back into your template.

NOTE

You only encounter export issues when exporting from a resource group rather than from your deployment history. If your last deployment accurately represents the current state of the resource group, you should export the template from the deployment history rather than from the resource group. Only export from a resource group when you have made changes to the resource group that are not defined in a single template.

For example, if you export a template for a resource group that contains a web app, SQL Database, and connection string in the site config, you see the following message:



Selecting the message shows you exactly which resource types were not exported.

Error details

Export template operation completed with errors.
Some resources were not exported. Please see details for more information. (Code: ExportTemplateCompletedWithErrors)

- Could not get resources of the type 'Microsoft.Web/sites/config'. Resources of this type will not be exported. (Code: ExportTemplateProviderError)
- Could not get resources of the type 'Microsoft.Web/sites/extensions'. Resources of this type will not be exported. (Code: ExportTemplateProviderError)

This topic shows common fixes.

Connection string

In the web sites resource, add a definition for the connection string to the database:

```
{
  "type": "Microsoft.Web/sites",
  ...
  "resources": [
    {
      "apiVersion": "2015-08-01",
      "type": "config",
      "name": "connectionstrings",
      "dependsOn": [
        "[concat('Microsoft.Web/Sites/', parameters('site-name'))]"
      ],
      "properties": {
        "DefaultConnectionString": {
          "value": "[concat('Data Source=tcp:', reference(concat('Microsoft.Sql/servers/',
parameters('<database-server-name>')).fullyQualifiedDomainName, ',1433;Initial Catalog=',
parameters('<database-name>'), ';User Id=', parameters('<admin-login>'), '@', parameters('<database-server-name>'), ';Password=',
parameters('<admin-password>'), ';')])",
          "type": "SQLServer"
        }
      }
    }
  ]
}
```

Web site extension

In the web site resource, add a definition for the code to install:

```
{
  "type": "Microsoft.Web/sites",
  ...
  "resources": [
    {
      "name": "MSDeploy",
      "type": "extensions",
      "location": "[resourceGroup().location]",
      "apiVersion": "2015-08-01",
      "dependsOn": [
        "[concat('Microsoft.Web/sites/', parameters('site-name'))]"
      ],
      "properties": {
        "packageUri": "[concat(parameters('<artifacts-location>'), '/', parameters('<package-folder>'), '/',
parameters('<package-file-name>'), parameters('<sas-token>'))]",
        "dbType": "None",
        "connectionString": "",
        "setParameters": {
          "IIS Web Application Name": "[parameters('site-name')]"
        }
      }
    }
  ]
}
```

Virtual machine extension

For examples of virtual machine extensions, see [Azure Windows VM Extension Configuration Samples](#).

Virtual network gateway

Add a virtual network gateway resource type.

```
{
  "type": "Microsoft.Network/virtualNetworkGateways",
  "name": "[parameters('<gateway-name>')]",
  "apiVersion": "2015-06-15",
  "location": "[resourceGroup().location]",
  "properties": {
    "gatewayType": "[parameters('<gateway-type>')]",
    "ipConfigurations": [
      {
        "name": "default",
        "properties": {
          "privateIPAllocationMethod": "Dynamic",
          "subnet": {
            "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('<vnet-name>'), parameters('<new-subnet-name>'))]"
          },
          "publicIpAddress": {
            "id": "[resourceId('Microsoft.Network/publicIPAddresses', parameters('<new-public-ip-address-Name>'))]"
          }
        }
      }
    ],
    "enableBgp": false,
    "vpnType": "[parameters('<vpn-type>')]"
  },
  "dependsOn": [
    "Microsoft.Network/virtualNetworks/codegroup4/subnets/GatewaySubnet",
    "[concat('Microsoft.Network/publicIPAddresses/', parameters('<new-public-ip-address-Name>'))]"
  ]
},
```

Local network gateway

Add a local network gateway resource type.

```
{
  "type": "Microsoft.Network/localNetworkGateways",
  "name": "[parameters('<local-network-gateway-name>')]",
  "apiVersion": "2015-06-15",
  "location": "[resourceGroup().location]",
  "properties": {
    "localNetworkAddressSpace": {
      "addressPrefixes": "[parameters('<address-prefixes>')]"
    }
  }
}
```

Connection

Add a connection resource type.

```
{  
    "apiVersion": "2015-06-15",  
    "name": "[parameters('<connection-name>')]",  
    "type": "Microsoft.Network/connections",  
    "location": "[resourceGroup().location]",  
    "properties": {  
        "virtualNetworkGateway1": {  
            "id": "[resourceId('Microsoft.Network/virtualNetworkGateways', parameters('<gateway-name>'))]"  
        },  
        "localNetworkGateway2": {  
            "id": "[resourceId('Microsoft.Network/localNetworkGateways', parameters('<local-gateway-name>'))]"  
        },  
        "connectionType": "IPsec",  
        "routingWeight": 10,  
        "sharedKey": "[parameters('<shared-key>')]"  
    }  
,
```

Next steps

Congratulations! You have learned how to export a template from resources that you created in the portal.

- You can deploy a template through [PowerShell](#), [Azure CLI](#), or [REST API](#).
- To see how to export a template through PowerShell, see [Using Azure PowerShell with Azure Resource Manager](#).
- To see how to export a template through Azure CLI, see [Use the Azure CLI for Mac, Linux, and Windows with Azure Resource Manager](#).

Create your first Azure Resource Manager template

3/31/2017 • 6 min to read • [Edit Online](#)

This topic walks you through the steps of creating your first Azure Resource Manager template. Resource Manager templates are JSON files that define the resources you need to deploy for your solution. To understand the concepts associated with deploying and managing your Azure solutions, see [Azure Resource Manager overview](#). If you have existing resources and want to get a template for those resources, see [Export an Azure Resource Manager template from existing resources](#).

To create and revise templates, you need a JSON editor. [Visual Studio Code](#) is a lightweight, open-source, cross-platform code editor. It supports creating and editing Resource Manager templates through an extension. This topic assumes you are using VS Code; however, if you have another JSON editor (like Visual Studio), you can use that editor.

Get VS Code and extension

1. If needed, install VS Code from <https://code.visualstudio.com/>.
2. Install the [Azure Resource Manager Tools](#) extension by accessing Quick Open (Ctrl+P) and running:

```
ext install msazurermtools.azurerm-vscode-tools
```

3. Restart VS Code when prompted to enable the extension.

Create blank template

Let's start with a blank template that includes only the basic sections of a template.

1. Create a file.
2. Copy and paste the following JSON syntax into your file:

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": { },
  "variables": { },
  "resources": [ ],
  "outputs": { }
}
```

3. Save this file as **azuredeploy.json**.

Add storage account

1. To define a storage account for deployment, you add that storage account to the **resources** section of your template. To find the values that are available for the storage account, look at the [storage accounts template reference](#). Copy the JSON that is shown for the storage account.
2. Paste that JSON into the **resources** section of your template, as shown in the following example:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": { },
    "variables": { },
    "resources": [
        {
            "name": "string",
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2016-05-01",
            "sku": {
                "name": "string"
            },
            "kind": "string",
            "location": "string",
            "tags": {},
            "properties": {
                "customDomain": {
                    "name": "string",
                    "useSubDomain": boolean
                },
                "encryption": {
                    "services": {
                        "blob": {
                            "enabled": boolean
                        }
                    },
                    "keySource": "Microsoft.Storage"
                },
                "accessTier": "string"
            }
        }
    ],
    "outputs": { }
}
```

The preceding example includes many placeholder values and some properties that you might not need in your storage account.

Set values for storage account

Now, you are ready to set values for your storage account.

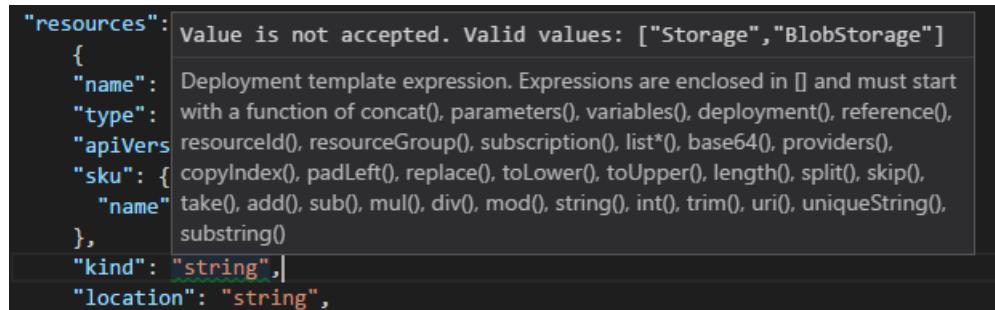
1. Look again at the [storage accounts template reference](#) where you copied the JSON. There are several tables that describe the properties and provide available values.
2. Notice that within the **properties** element, **customDomain**, **encryption**, and **accessTier** are all listed as not required. These values may be important for your scenarios, but to keep this example simple, let's remove them.

```

"resources": [
  {
    "name": "string",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2016-05-01",
    "sku": {
      "name": "string"
    },
    "kind": "string",
    "location": "string",
    "tags": {},
    "properties": {}
  }
],

```

3. Currently, the **kind** element is set to a placeholder value ("string"). VS Code includes many features that help you understand the values to use in your template. Notice that VS Code indicates this value is not valid. If you hover over "string", VS Code suggests that the valid values for **kind** are `Storage` or `BlobStorage`.

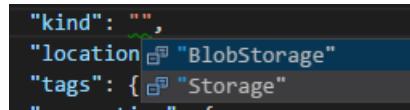


```

"resources": [
  {
    "name": "string",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2016-05-01",
    "sku": {
      "name": "string"
    },
    "kind": "string", Value is not accepted. Valid values: ["Storage", "BlobStorage"]
    "location": "string",
  }
]

```

To see the available values, delete the characters between the double-quotes and select **Ctrl+Space**. Select **Storage** from the available options.



```

"kind": "", Value is not accepted. Valid values: ["Storage", "BlobStorage"]
"location": "BlobStorage"
"tags": { Value is not accepted. Valid values: ["Storage", "BlobStorage"]
  "string": "string"
}

```

If you are not using VS Code, look at the storage accounts template reference page. Notice that the description lists the same valid values. Set the element to **Storage**.

```

"kind": "Storage",

```

Your template now looks like:

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": { },
  "variables": { },
  "resources": [
    {
      "name": "string",
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2016-05-01",
      "sku": {
        "name": "string"
      },
      "kind": "Storage",
      "location": "string",
      "tags": {},
      "properties": {
      }
    }
  ],
  "outputs": { }
}
```

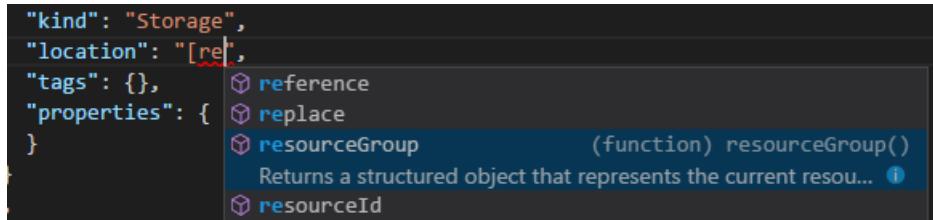
Add template function

You use functions within your template to simplify the syntax of the template, and to retrieve values that are only available when the template is being deployed. For the full set of template functions, see [Azure Resource Manager template functions](#).

To specify that the storage account is deployed to the same location as the resource group, set the **location** property to:

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

Again, VS Code helps you by suggesting available functions.



Notice that the function is surrounded by square brackets. The `resourceGroup` function returns an object with a property called `location`. The resource group holds all related resources for your solution. You could hardcode the location property to a value like "Central US" but you would have to manually change the template to redeploy to a different location. Using the `resourceGroup` function, makes it easy to redeploy this template to a different resource group in a different location.

Your template now looks like:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": { },
    "variables": { },
    "resources": [
        {
            "name": "string",
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2016-05-01",
            "sku": {
                "name": "string"
            },
            "kind": "Storage",
            "location": "[resourceGroup().location]",
            "tags": {},
            "properties": {}
        }
    ],
    "outputs": { }
}
```

Add parameters and variables

There are only two values left to set in your template - **name** and **sku.name**. For these properties, you add parameters that enable you to customize these values during deployment.

Storage account names have several restrictions that make them difficult to set. The name must be between 3 and 24 characters in length, use only numbers and lower-case letters, and be unique. Rather than trying to guess a unique value that matches the restrictions, use the [uniqueString](#) function to generate a hash value. To give this hash value more meaning, add a prefix that helps you identify it as a storage account after deployment.

1. To pass in a prefix for the name that matches your naming conventions, go to the **parameters** section of your template. Add a parameter to the template that accepts a prefix for the storage account name:

```
"parameters": {
    "storageNamePrefix": {
        "type": "string",
        "maxLength": 11,
        "defaultValue": "storage",
        "metadata": {
            "description": "The value to use for starting the storage account name."
        }
    }
},
```

The prefix is limited to a maximum of 11 characters because [uniqueString](#) returns 13 characters, and the name cannot exceed 24 characters. If you do not pass in a value for the parameter during deployment, the default value is used.

2. Go to the **variables** section of the template. To construct the name from the prefix and unique string, add the following variable:

```
"variables": {
    "storageName": "[concat(parameters('storageNamePrefix'), uniqueString(resourceGroup().id))]"
},
```

3. In the **resources** section, set the storage account name to that variable.

```
"name": "[variables('storageName')]",
```

4. To enable passing in different SKUs for the storage account, go to the **parameters** section. After the parameter for storage name prefix, add a parameter that specifies the allowed SKU values and a default value. You can find the allowed values from either the template reference page or VS Code. In the following example, you include all valid values for SKU. However, you could limit the allowed values to only those types of SKUs that you want to deploy through this template.

```
"parameters": {  
    "storageNamePrefix": {  
        "type": "string",  
        "maxLength": 11,  
        "defaultValue": "storage",  
        "metadata": {  
            "description": "The value to use for starting the storage account name."  
        }  
    },  
    "storageSKU": {  
        "type": "string",  
        "allowedValues": [  
            "Standard_LRS",  
            "Standard_ZRS",  
            "Standard_GRS",  
            "Standard_RAGRS",  
            "Premium_LRS"  
        ],  
        "defaultValue": "Standard_LRS",  
        "metadata": {  
            "description": "The type of replication to use for the storage account."  
        }  
    }  
},
```

5. Change the SKU property to use the value from the parameter:

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

6. Save your file.

Your template now looks like:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageNamePrefix": {
            "type": "string",
            "maxLength": 11,
            "defaultValue": "storage",
            "metadata": {
                "description": "The value to use for starting the storage account name."
            }
        },
        "storageSKU": {
            "type": "string",
            "allowedValues": [
                "Standard_LRS",
                "Standard_ZRS",
                "Standard_GRS",
                "Standard_RAGRS",
                "Premium_LRS"
            ],
            "defaultValue": "Standard_LRS",
            "metadata": {
                "description": "The type of replication to use for the storage account."
            }
        }
    },
    "variables": {
        "storageName": "[concat(parameters('storageNamePrefix'), uniqueString(resourceGroup().id))]"
    },
    "resources": [
        {
            "name": "[variables('storageName')]",
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2016-05-01",
            "sku": {
                "name": "[parameters('storageSKU')]"
            },
            "kind": "Storage",
            "location": "[resourceGroup().location]",
            "tags": {},
            "properties": {}
        }
    ],
    "outputs": { }
}
```

Next steps

- Your template is complete, and you are ready to deploy it to your subscription. To deploy, see [Deploy resources to Azure](#).
- To learn more about the structure of a template, see [Authoring Azure Resource Manager templates](#).
- To view complete templates for many different types of solutions, see the [Azure Quickstart Templates](#).

Creating and deploying Azure resource groups through Visual Studio

3/13/2017 • 9 min to read • [Edit Online](#)

With Visual Studio and the [Azure SDK](#), you can create a project that deploys your infrastructure and code to Azure. For example, you can define the web host, web site, and database for your app, and deploy that infrastructure along with the code. Or, you can define a Virtual Machine, Virtual Network and Storage Account, and deploy that infrastructure along with a script that is executed on Virtual Machine. The **Azure Resource Group** deployment project enables you to deploy all the needed resources in a single, repeatable operation. For more information about deploying and managing your resources, see [Azure Resource Manager overview](#).

Azure Resource Group projects contain Azure Resource Manager JSON templates, which define the resources that you deploy to Azure. To learn about the elements of the Resource Manager template, see [Authoring Azure Resource Manager templates](#). Visual Studio enables you to edit these templates, and provides tools that simplify working with templates.

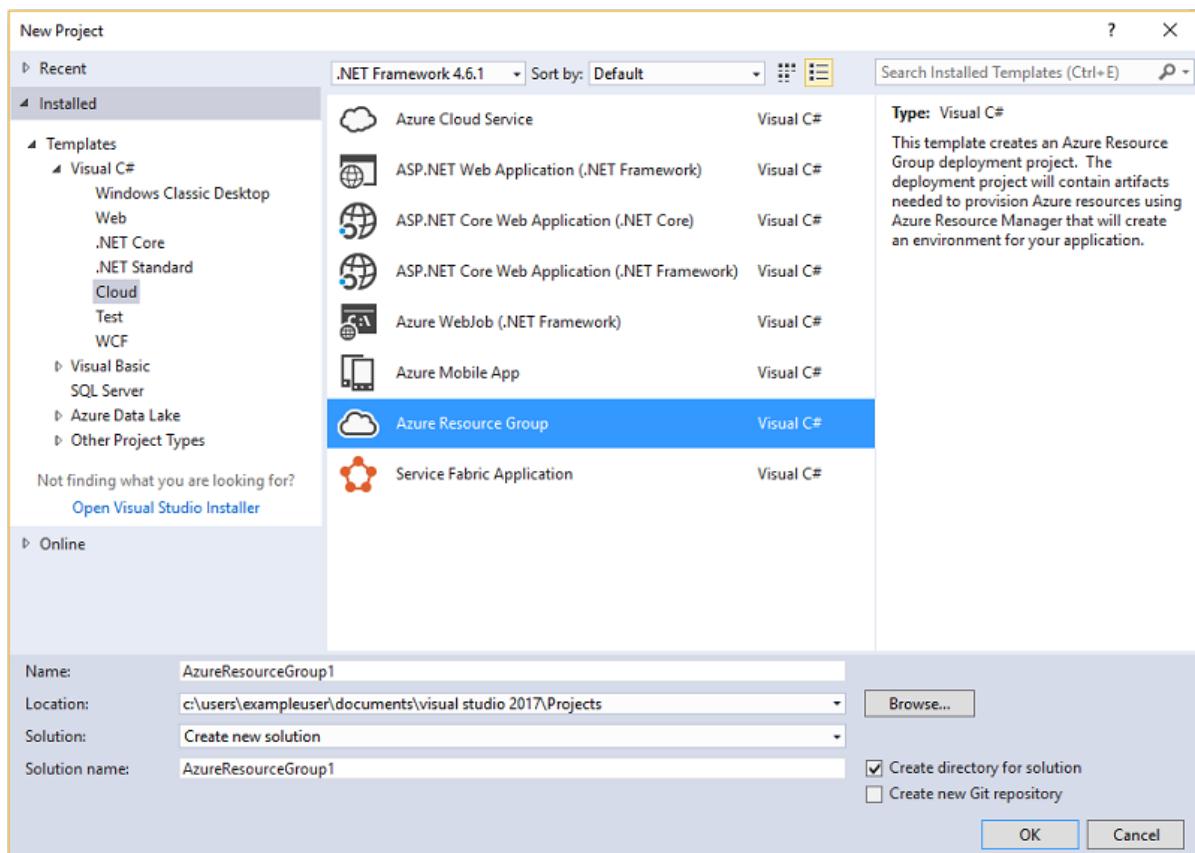
In this article, you deploy a web app and SQL Database. However, the steps are almost the same for any type resource. You can as easily deploy a Virtual Machine and its related resources. Visual Studio provides many different starter templates for deploying common scenarios.

This article shows Visual Studio 2017. If you use Visual Studio 2015 Update 2 and Microsoft Azure SDK for .NET 2.9, or Visual Studio 2013 with Azure SDK 2.9, your experience is largely the same. You can use versions of the Azure SDK from 2.6 or later; however, your experience of the user interface may be different than the user interface shown in this article. We strongly recommend that you install the latest version of the [Azure SDK](#) before starting the steps.

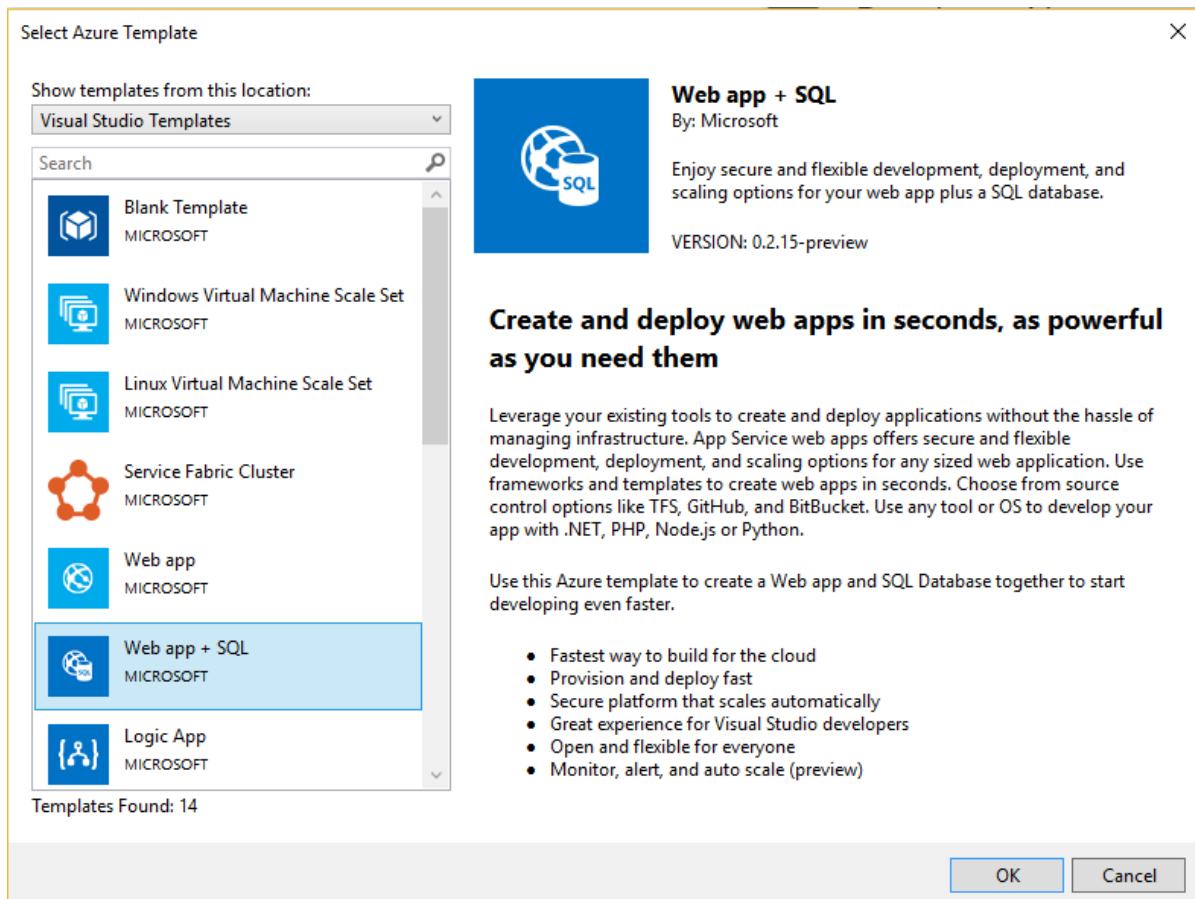
Create Azure Resource Group project

In this procedure, you create an Azure Resource Group project with a **Web app + SQL** template.

1. In Visual Studio, choose **File, New Project**, choose **C#** or **Visual Basic**. Then choose **Cloud**, and **Azure Resource Group** project.



- Choose the template that you want to deploy to Azure Resource Manager. Notice there are many different options based on the type of project you wish to deploy. For this article, choose the **Web app + SQL** template.



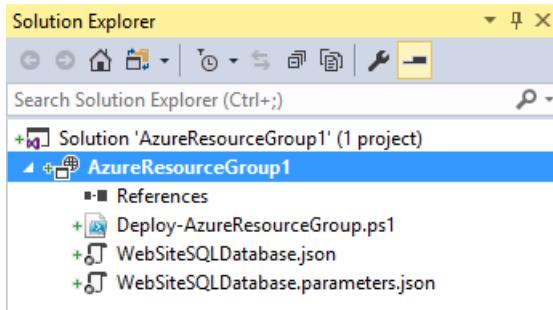
The template you pick is just a starting point; you can add and remove resources to fulfill your scenario.

NOTE

Visual Studio retrieves a list of available templates online. The list may change.

Visual Studio creates a resource group deployment project for the web app and SQL database.

3. To see what you created, look at the node in the deployment project.



Since we chose the Web app + SQL template for this example, you see the following files:

FILE NAME	DESCRIPTION
Deploy-AzureResourceGroup.ps1	A PowerShell script that invokes PowerShell commands to deploy to Azure Resource Manager. Note Visual Studio uses this PowerShell script to deploy your template. Any changes you make to this script affect deployment in Visual Studio, so be careful.
WebSiteSQLDatabase.json	The Resource Manager template that defines the infrastructure you want to deploy to Azure, and the parameters you can provide during deployment. It also defines the dependencies between the resources so Resource Manager deploys the resources in the correct order.
WebSiteSQLDatabase.parameters.json	A parameters file that contains values needed by the template. You pass in parameter values to customize each deployment.

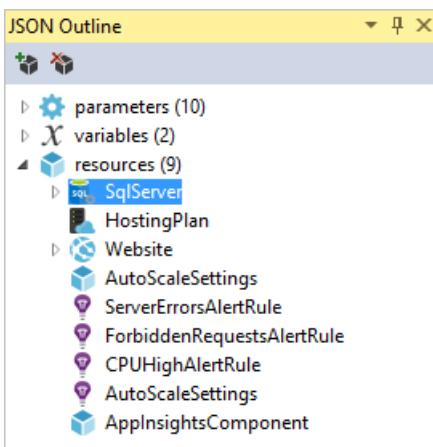
All resource group deployment projects contain these basic files. Other projects may contain additional files to support other functionality.

Customize the Resource Manager template

You can customize a deployment project by modifying the JSON templates that describe the resources you want to deploy. JSON stands for JavaScript Object Notation, and is a serialized data format that is easy to work with. The JSON files use a schema that you reference at the top of each file. If you want to understand the schema, you can download and analyze it. The schema defines what elements are valid, the types and formats of fields, the possible values of enumerated values, and so on. To learn about the elements of the Resource Manager template, see [Authoring Azure Resource Manager templates](#).

To work on your template, open **WebSiteSQLDatabase.json**.

The Visual Studio editor provides tools to assist you with editing the Resource Manager template. The **JSON Outline** window makes it easy to see the elements defined in your template.

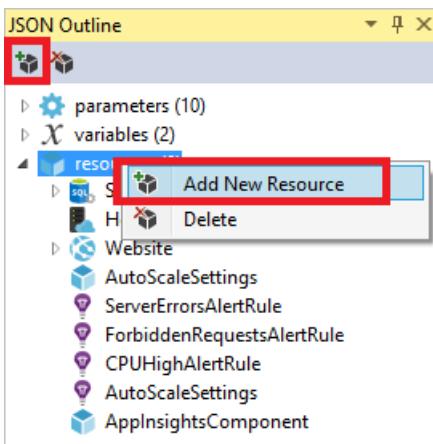


Selecting any of the elements in the outline takes you to that part of the template and highlights the corresponding JSON.

The screenshot shows the 'JSON Outline' window with the 'Website' node selected. The main pane displays the JSON code for a 'Website' resource:

```
Schema: http://schema.management.azure.com/schemas/2015-01-01/deployment.json# properties . . .
  "name": "[parameters('hostingPlanName')]"
},
{
  "apiVersion": "2015-08-01",
  "name": "[variables('webSiteName')]",
  "type": "Microsoft.Web/sites",
  "location": "[resourceGroup().location]",
  "dependsOn": [
    "[concat('Microsoft.Web/serverFarms/',"
],
  "tags": {
    "[concat('hidden-related:', resourceGroup().name, '/', 'displayName': 'Website"
  },
  "properties": {
    "name": "[variables('webSiteName')]",
    "serverFarmId": "[resourceId('Microsoft.Web/serverFarms', "
  }
}
```

You can add a resource by either selecting the **Add Resource** button at the top of the JSON Outline window, or by right-clicking **resources** and selecting **Add New Resource**.



For this tutorial, select **Storage Account** and give it a name. Provide a name that is no more than 11 characters, and only contains numbers and lower-case letters.

Add Resource

Notice that not only was the resource added, but also a parameter for the type storage account, and a variable for the name of the storage account.

JSON Outline

```

parameters (11)
  hostingPlanName
  skuName
  skuCapacity
  administratorLogin
  administratorLoginPassword
  databaseName
  collation
  edition
  maxSizeBytes
  requestedServiceObjectiveName
  storageType

variables (3)
  webSiteName
  sqlserverName
  storageName

resources (10)
  SqlServer
  HostingPlan
  Website
    connectionstrings
    AutoScaleSettings
    ServerErrorsAlertRule
    ForbiddenRequestsAlertRule
    CPUHighAlertRule
    AutoScaleSettings
    AppInsightsComponent
  storage

```

The **storageType** parameter is pre-defined with allowed types and a default type. You can leave these values or edit them for your scenario. If you do not want anyone to deploy a **Premium_LRS** storage account through this template, remove it from the allowed types.

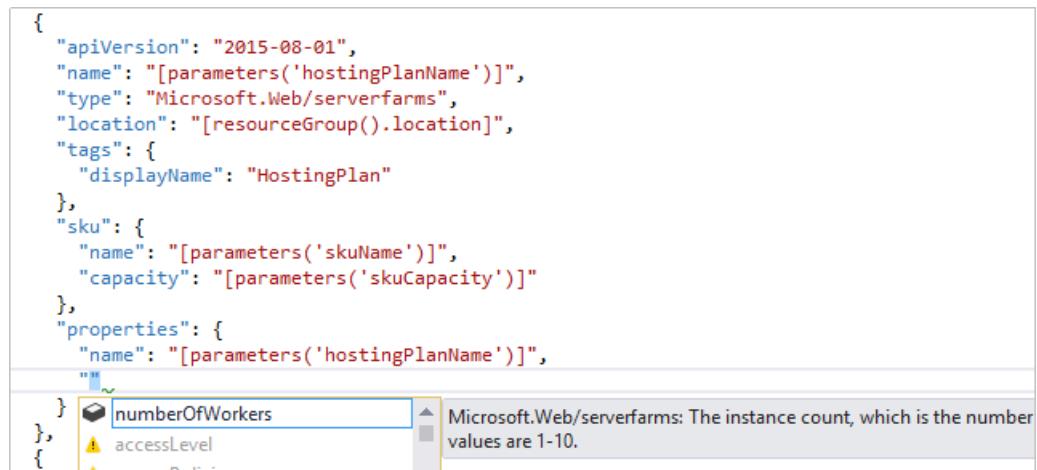
```

"storageType": {
  "type": "string",
  "defaultValue": "Standard_LRS",
  "allowedValues": [
    "Standard_LRS",
    "Standard_ZRS",
    "Standard_GRS",
    "Standard_RAGRS"
  ]
}

```

Visual Studio also provides intellisense to help you understand what properties are available when editing the

template. For example, to edit the properties for your App Service plan, navigate to the **HostingPlan** resource, and add a value for the **properties**. Notice that intellisense shows the available values and provides a description of that value.



```
{  
  "apiVersion": "2015-08-01",  
  "name": "[parameters('hostingPlanName')]",  
  "type": "Microsoft.Web/serverfarms",  
  "location": "[resourceGroup().location]",  
  "tags": {  
    "displayName": "HostingPlan"  
  },  
  "sku": {  
    "name": "[parameters('skuName')]",  
    "capacity": "[parameters('skuCapacity')]"  
  },  
  "properties": {  
    "name": "[parameters('hostingPlanName')]",  
    "numberOfWorkers": 1,  
    "accessLevel": "Owner"  
  }  
}
```

The screenshot shows the 'HostingPlan' resource definition in JSON. The 'sku' field has a tooltip: 'Microsoft.Web/serverfarms: The instance count, which is the number of workers. Values are 1-10.' The 'numberOfWorkers' field is highlighted with a red box.

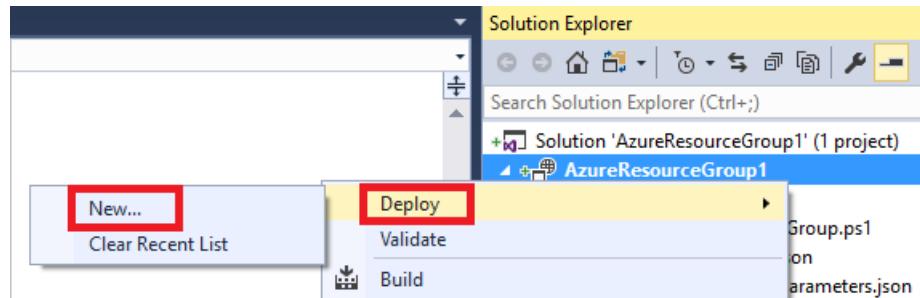
You can set **numberOfWorkers** to 1.

```
"properties": {  
  "name": "[parameters('hostingPlanName')]",  
  "numberOfWorkers": 1  
}
```

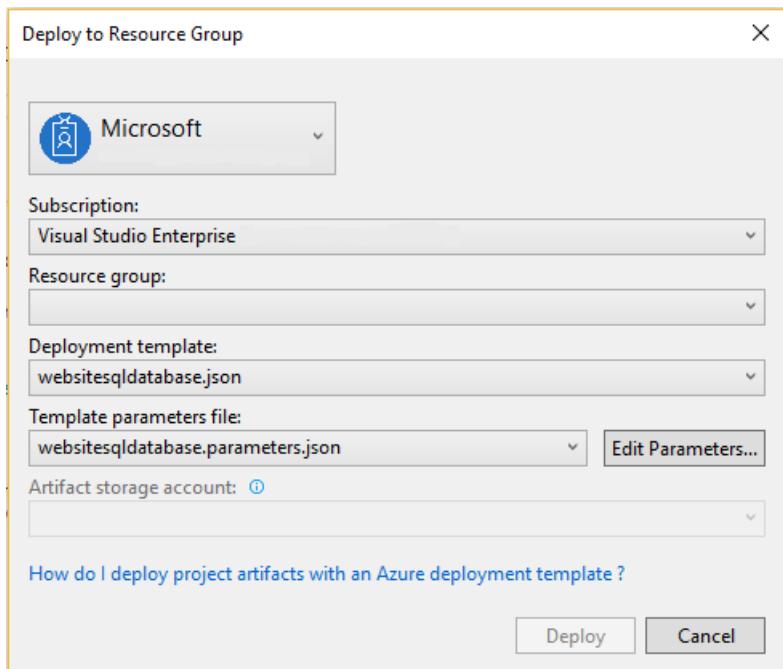
Deploy the Resource Group project to Azure

You are now ready to deploy your project. When you deploy an Azure Resource Group project, you deploy it to an Azure resource group. The resource group is a logical grouping of resources that share a common lifecycle.

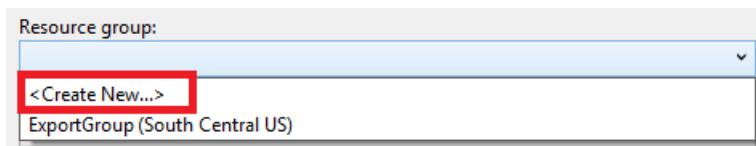
1. On the shortcut menu of the deployment project node, choose **Deploy > New**.



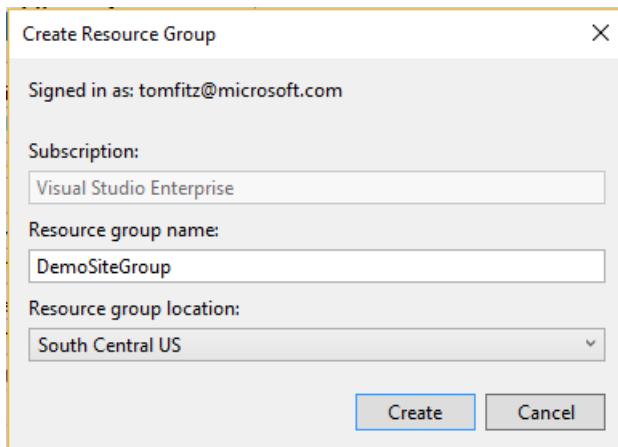
The **Deploy to Resource Group** dialog box appears.



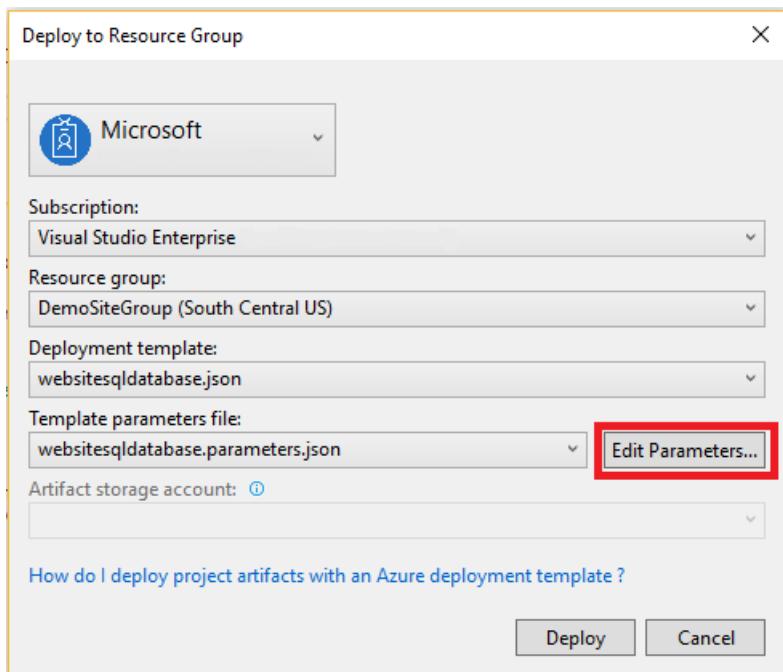
2. In the **Resource group** dropdown box, choose an existing resource group or create a new one. To create a resource group, open the **Resource Group** dropdown box and choose **Create New**.



The **Create Resource Group** dialog box appears. Give your group a name and location, and select the **Create** button.



3. Edit the parameters for the deployment by selecting the **Edit Parameters** button.



4. Provide values for the empty parameters and select the **Save** button. The empty parameters are **hostingPlanName**, **administratorLogin**, **administratorLoginPassword**, and **databaseName**.

hostingPlanName specifies a name for the [App Service plan](#) to create.

administratorLogin specifies the user name for the SQL Server administrator. Do not use common admin names like **sa** or **admin**.

The **administratorLoginPassword** specifies a password for SQL Server administrator. The **Save passwords as plain text in the parameters file** option is not secure; therefore, do not select this option. Since the password is not saved as plain text, you need to provide this password again during deployment.

databaseName specifies a name for the database to create.

The screenshot shows the 'Edit Parameters' dialog box. It displays a table of parameter values:

Parameter Name	Value
hostingPlanName	DemoSitePlan
skuName	F1
skuCapacity	1
administratorLogin	DemoAdmin
administratorLoginPassword	*****
databaseName	DemoDatabase
collation	SQL_Latin1_General_CI_AS
edition	Basic
maxSizeBytes	1073741824
requestedServiceObjectiveName	Basic
storageType	Standard_LRS

Below the table is a checkbox labeled 'Save passwords as plain text in the parameters file'. At the bottom are 'Save' and 'Cancel' buttons, with 'Save' highlighted by a blue box.

5. Choose the **Deploy** button to deploy the project to Azure. A PowerShell console opens outside of the Visual Studio instance. Enter the SQL Server administrator password in the PowerShell console when prompted. **Your PowerShell console may be hidden behind other items or minimized in the task bar.** Look for this console and select it to provide the password.

NOTE

Visual Studio may ask you to install the Azure PowerShell cmdlets. You need the Azure PowerShell cmdlets to successfully deploy resource groups. If prompted, install them.

6. The deployment may take a few minutes. In the **Output** windows, you see the status of the deployment. When the deployment has finished, the last message indicates a successful deployment with something similar to:

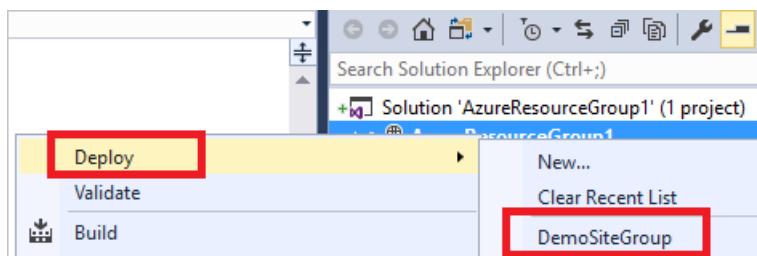
```
...
18:00:58 - Successfully deployed template 'websitesqldatabase.json' to resource group 'DemoSiteGroup'.
```

7. In a browser, open the [Azure portal](#) and sign in to your account. To see the resource group, select **Resource groups** and the resource group you deployed to.

The screenshot shows the Microsoft Azure Resource Groups blade. On the left is a sidebar with icons for Storage, SQL, Compute, and other services. The main area is titled "Resource groups" and shows a list of items under "Subscriptions: All 6 selected – Don't see". A filter bar with "Filter by name..." is present. The list contains three items: "MyGroup", "DemoSiteGroup" (which is highlighted with a red box), and "ExportGroup".

8. You see all the deployed resources. Notice that the name of the storage account is not exactly what you specified when adding that resource. The storage account must be unique. The template automatically adds a string of characters to the name you provided to provide a unique name.

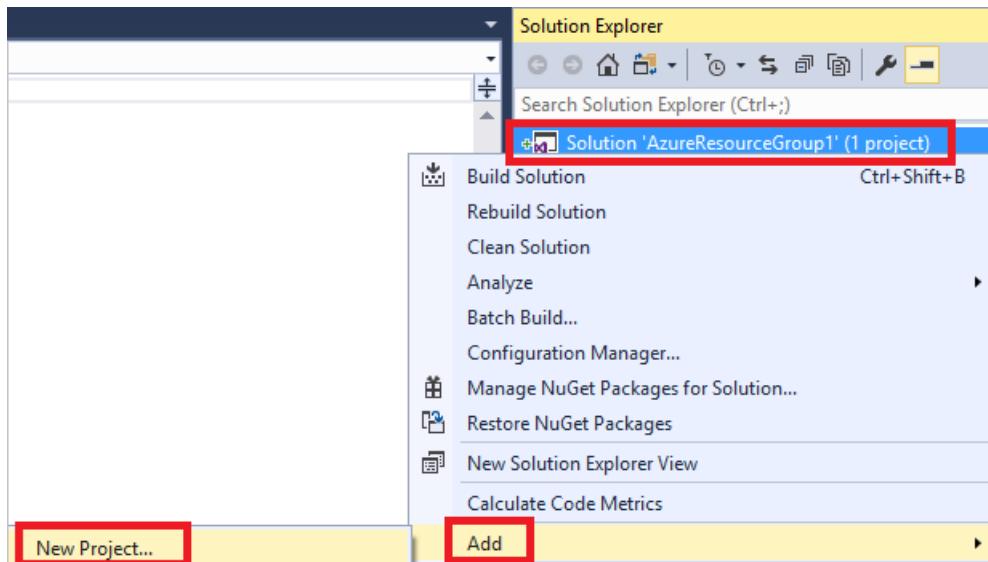
9. If you make changes and want to redeploy your project, choose the existing resource group from the shortcut menu of Azure resource group project. On the shortcut menu, choose **Deploy**, and then choose the resource group you deployed.



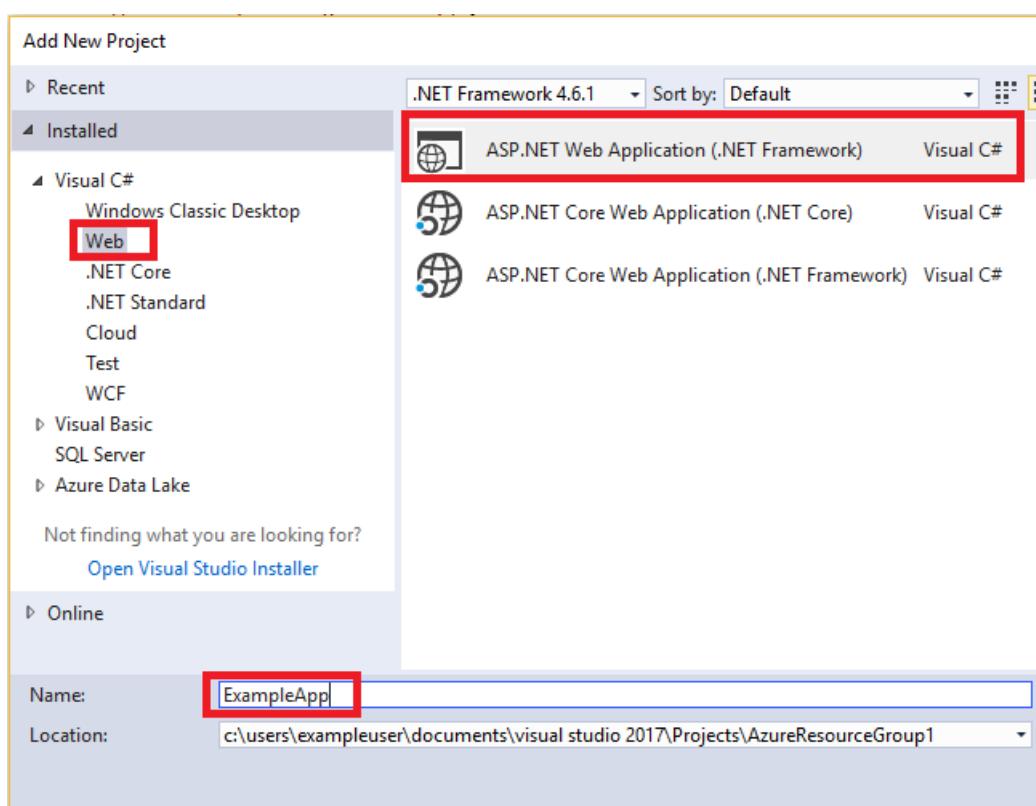
Deploy code with your infrastructure

At this point, you have deployed the infrastructure for your app, but there is no actual code deployed with the project. This article shows how to deploy a web app and SQL Database tables during deployment. If you are deploying a Virtual Machine instead of a web app, you want to run some code on the machine as part of deployment. The process for deploying code for a web app or for setting up a Virtual Machine is almost the same.

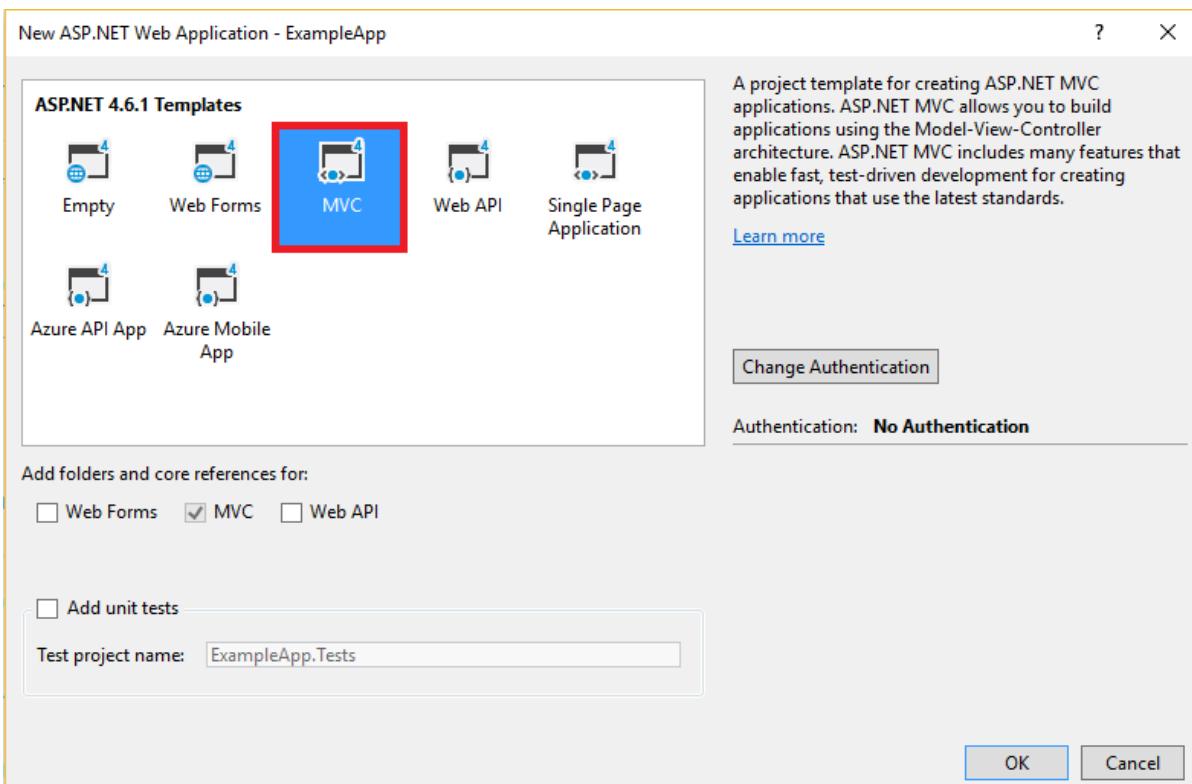
1. Add a project to your Visual Studio solution. Right-click the solution, and select **Add > New Project**.



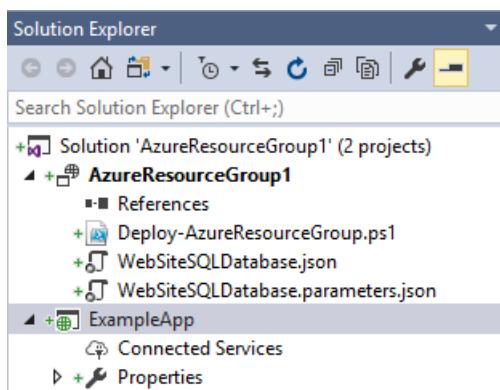
2. Add an **ASP.NET Web Application**.



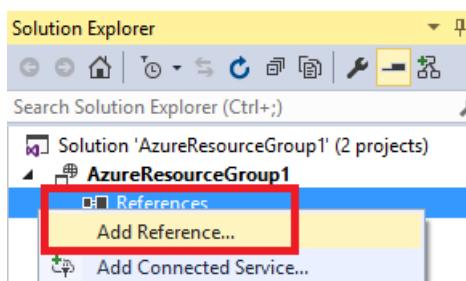
3. Select **MVC**.



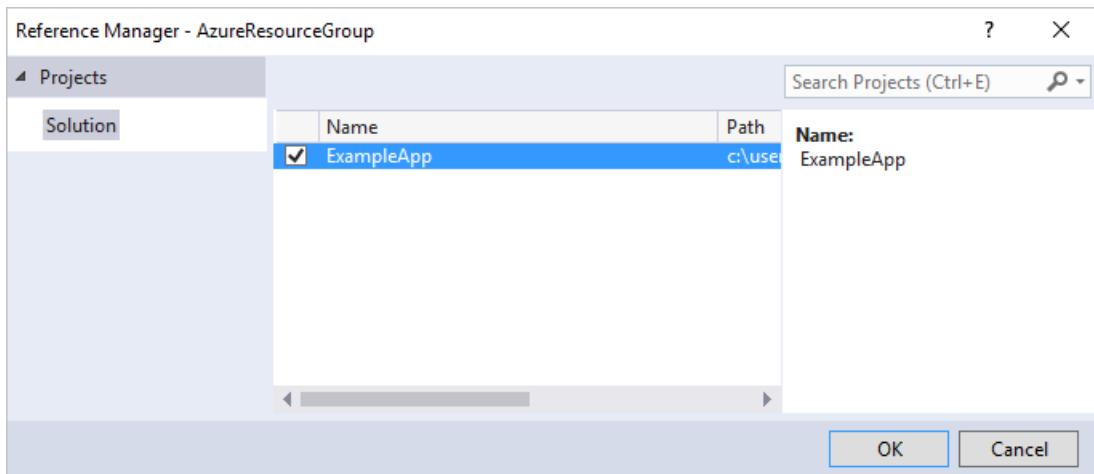
4. After Visual Studio creates your web app, you see both projects in the solution.



5. Now, you need to make sure your resource group project is aware of the new project. Go back to your resource group project (AzureResourceGroup1). Right-click **References** and select **Add Reference...**.



6. Select the web app project that you created.



By adding a reference, you link the web app project to the resource group project, and automatically set three key properties. You see these properties in the **Properties** window for the reference.

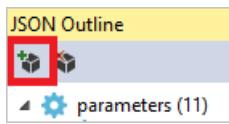
Additional Properties	PackageLocation=..\AzureResourceGroup1\obj\Debug\ProjectReferences\\ExampleApp\package.zip
Include File Path	obj\Debug\ProjectReferences\\ExampleApp\package.zip
Include Targets	Build;Package

The properties are:

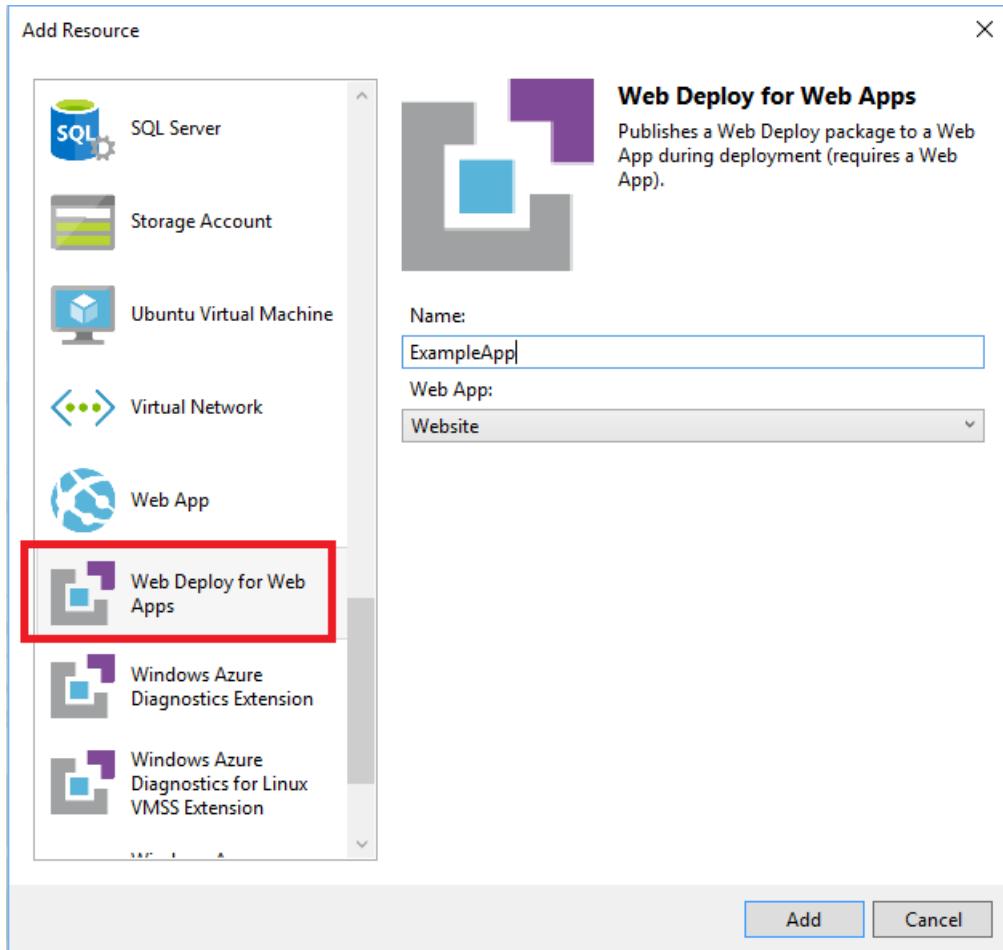
- The **Additional Properties** contains the web deployment package staging location that is pushed to the Azure Storage. Note the folder (ExampleApp) and file (package.zip). You need to know these values because you provide them as parameters when deploying the app.
- The **Include File Path** contains the path where the package is created. The **Include Targets** contains the command that deployment executes.
- The default value of **Build;Package** enables the deployment to build and create a web deployment package (package.zip).

You do not need a publish profile as the deployment gets the necessary information from the properties to create the package.

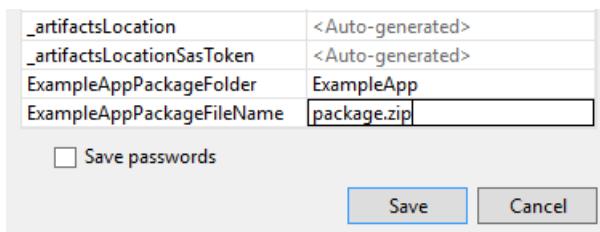
7. Go back to WebSiteSQLDatabase.json and add a resource to the template.



8. This time select **Web Deploy for Web Apps**.



9. Redeploy your resource group project to the resource group. This time there are some new parameters. You do not need to provide values for `_artifactsLocation` or `_artifactsLocationSasToken` because Visual Studio automatically generates those values. However, you have to set the folder and file name to the path that contains the deployment package (shown as **ExampleAppPackageFolder** and **ExampleAppPackageName** in the following image). Provide the values you saw earlier in the reference properties (**ExampleApp** and **package.zip**).



For the **Artifact storage account**, select the one deployed with this resource group.

10. After the deployment has finished, select your web app in the portal. Select the URL to browse to the site.

Resource group	URL
DemoSiteGroup	http://websitelg52msdchkveo.azurewebsite...
Status	App Service plan/pricing tier
Running	DemoSitePlan (Free)
Location	FTP/Deployment username
West US	webSitelg52msdchkveo\tomfitz
Subscription name	FTP hostname
Windows Azure MSDN - Visual Studio Ulti...	ftp://waws-prod-bay-065.ftp.azurewebsites...
Subscription ID	FTPS hostname
	ftps://waws-prod-bay-065.ftp.azurewebsite...

11. Notice that you have successfully deployed the default ASP.NET app.

The screenshot shows a web browser displaying the deployed ASP.NET application. The top navigation bar includes 'Application name' (highlighted in red), 'Home', 'About', 'Contact', 'Register', and 'Log in'. The main content area features a large 'ASP.NET' logo, followed by the text: 'ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.' Below this text is a blue button labeled 'Learn more »'.

Next steps

- To learn about managing your resources through the portal, see [Using the Azure portal to manage your Azure resources](#).
- To learn more about templates, see [Authoring Azure Resource Manager templates](#).

Best practices for creating Azure Resource Manager templates

4/3/2017 • 10 min to read • [Edit Online](#)

These guidelines can help you create Azure Resource Manager templates that are reliable and easy to use. The guidelines are only suggestions. They are not requirements, except where noted. Your scenario might require a variation of one of the following approaches or examples.

Resource names

Generally, you work with three types of resource names in Resource Manager:

- Resource names that must be unique.
- Resource names that are not required to be unique, but you choose to provide a name that can help you identify a resource based on context.
- Resource names that can be generic.

For help establishing a naming convention, see the [Azure infrastructure naming guidelines](#). For information about resource name restrictions, see [Recommended naming conventions for Azure resources](#).

Unique resource names

You must provide a unique resource name for any resource type that has a data access endpoint. Some common resource types that require a unique name include:

- Azure Storage¹
- Web Apps feature of Azure App Service
- SQL Server
- Azure Key Vault
- Azure Redis Cache
- Azure Batch
- Azure Traffic Manager
- Azure Search
- Azure HDInsight

¹ Storage account names also must be lowercase, 24 characters or less, and not have any hyphens.

If you provide a parameter for a resource name, you must provide a unique name when you deploy the resource. Optionally, you can create a variable that uses the [uniqueString\(\)](#) function to generate a name.

You also might want to add a prefix or suffix to the **uniqueString** result. Modifying the unique name can help you more easily identify the resource type from the name. For example, you can generate a unique name for a storage account by using the following variable:

```
"variables": {  
    "storageAccountName": "[concat(uniqueString(resourceGroup().id), 'storage')]"  
}
```

Resource names for identification

Some resource types you might want to name, but their names do not have to be unique. For these resource types,

you can provide a name that identifies both the resource context and the resource type. Provide a descriptive name that helps you identify the resource in a list of resources. If you need to use a different resource name for different deployments, you can use a parameter for the name:

```
"parameters": {  
    "vmName": {  
        "type": "string",  
        "defaultValue": "demoLinuxVM",  
        "metadata": {  
            "description": "The name of the VM to create."  
        }  
    }  
}
```

If you do not need to pass in a name during deployment, you can use a variable:

```
"variables": {  
    "vmName": "demoLinuxVM"  
}
```

You also can use a hard-coded value:

```
{  
    "type": "Microsoft.Compute/virtualMachines",  
    "name": "demoLinuxVM",  
    ...  
}
```

Generic resource names

For resource types that you mostly access through a different resource, you can use a generic name that is hard-coded in the template. For example, you can set a standard, generic name for firewall rules on a SQL server:

```
{  
    "type": "firewallrules",  
    "name": "AllowAllWindowsAzureIps",  
    ...  
}
```

Parameters

The following information can be helpful when you work with parameters:

- Minimize your use of parameters. Whenever possible, use a variable or a literal value. Use parameters only for these scenarios:
 - Settings that you want to use variations of according to environment (SKU, size, capacity).
 - Resource names that you want to specify for easy identification.
 - Values that you use frequently to complete other tasks (such as an admin user name).
 - Secrets (such as passwords).
 - The number or array of values to use when you create multiple instances of a resource type.
- Use camel case for parameter names.
- Provide a description of every parameter in the metadata:

```

"parameters": {
    "storageAccountType": {
        "type": "string",
        "metadata": {
            "description": "The type of the new storage account created to store the VM disks."
        }
    }
}

```

- Define default values for parameters (except for passwords and SSH keys):

```

"parameters": {
    "storageAccountType": {
        "type": "string",
        "defaultValue": "Standard_GRS",
        "metadata": {
            "description": "The type of the new storage account created to store the VM disks."
        }
    }
}

```

- Use **SecureString** for all passwords and secrets:

```

"parameters": {
    "secretValue": {
        "type": "securestring",
        "metadata": {
            "description": "The value of the secret to store in the vault."
        }
    }
}

```

- Whenever possible, don't use a parameter to specify location. Instead, use the **location** property of the resource group. By using the **resourceGroup().location** expression for all your resources, resources in the template are deployed in the same location as the resource group:

```

"resources": [
{
    "name": "[variables('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2016-01-01",
    "location": "[resourceGroup().location]",
    ...
}
]

```

If a resource type is supported in only a limited number of locations, you might want to specify a valid location directly in the template. If you must use a **location** parameter, share that parameter value as much as possible with resources that are likely to be in the same location. This minimizes the number of times users are asked to provide location information.

- Avoid using a parameter or variable for the API version for a resource type. Resource properties and values can vary by version number. IntelliSense in a code editor cannot determine the correct schema when the API version is set to a parameter or variable. Instead, hard-code the API version in the template.

Variables

The following information can be helpful when you work with variables:

- Use variables for values that you need to use more than once in a template. If a value is used only once, a hard-coded value makes your template easier to read.
- You cannot use the **reference** function in the **variables** section of the template. The **reference** function derives its value from the resource's runtime state. However, variables are resolved during the initial parsing of the template. Construct values that need the **reference** function directly in the **resources** or **outputs** section of the template.
- Include variables for resource names that must be unique, as described in [Resource names](#).
- You can group variables into complex objects. Use the **variable.subentry** format to reference a value from a complex object. Grouping variables can help you track related variables. It also improves readability of the template. Here's an example:

```

"variables": {
  "storage": {
    "name": "[concat(uniqueString(resourceGroup().id), 'storage')]",
    "type": "Standard_LRS"
  }
},
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "name": "[variables('storage').name]",
    "apiVersion": "2016-01-01",
    "location": "[resourceGroup().location]",
    "sku": {
      "name": "[variables('storage').type]"
    },
    ...
  }
]

```

NOTE

A complex object cannot contain an expression that references a value from a complex object. Define a separate variable for this purpose.

For advanced examples of using complex objects as variables, see [Share state in Azure Resource Manager templates](#).

Resources

The following information can be helpful when you work with resources:

- To help other contributors understand the purpose of the resource, specify **comments** for each resource in the template:

```

"resources": [
  {
    "name": "[variables('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2016-01-01",
    "location": "[resourceGroup().location]",
    "comments": "This storage account is used to store the VM disks.",
    ...
  }
]

```

- You can use tags to add metadata to resources. Use metadata to add information about your resources. For

example, you can add metadata to record billing details for a resource. For more information, see [Using tags to organize your Azure resources](#).

- If you use a *public endpoint* in your template (such as an Azure Blob storage public endpoint), *do not hard-code* the namespace. Use the **reference** function to dynamically retrieve the namespace. You can use this approach to deploy the template to different public namespace environments without manually changing the endpoint in the template. Set the API version to the same version that you are using for the storage account in your template:

```
"osDisk": {  
    "name": "osdisk",  
    "vhd": {  
        "uri": "[concat(reference(concat('Microsoft.Storage/storageAccounts/'),  
variables('storageAccountName')), '2016-01-01').primaryEndpoints.blob,  
variables('vmStorageAccountContainerName'), '/',variables('OSDiskName'),'.vhd')]"  
    }  
}
```

If the storage account is deployed in the same template that you are creating, you do not need to specify the provider namespace when you reference the resource. This is the simplified syntax:

```
"osDisk": {  
    "name": "osdisk",  
    "vhd": {  
        "uri": "[concat(reference(variables('storageAccountName'), '2016-01-  
01').primaryEndpoints.blob, variables('vmStorageAccountContainerName'),  
'/',variables('OSDiskName'),'.vhd')]"  
    }  
}
```

If you have other values in your template that are configured to use a public namespace, change these values to reflect the same **reference** function. For example, you can set the **storageUri** property of the virtual machine diagnostics profile:

```
"diagnosticsProfile": {  
    "bootDiagnostics": {  
        "enabled": "true",  
        "storageUri": "[reference(concat('Microsoft.Storage/storageAccounts/'),  
variables('storageAccountName')), '2016-01-01').primaryEndpoints.blob]"  
    }  
}
```

You also can reference an existing storage account that is in a different resource group:

```
"osDisk": {  
    "name": "osdisk",  
    "vhd": {  
        "uri": "[concat(reference(resourceId(parameters('existingResourceGroup'),  
'Microsoft.Storage/storageAccounts/'), parameters('existingStorageAccountName')), '2016-01-  
01').primaryEndpoints.blob, variables('vmStorageAccountContainerName'), '/',  
variables('OSDiskName'),'.vhd')]"  
    }  
}
```

- Assign public IP addresses to a virtual machine only when an application requires it. To connect to a virtual machine (VM) for debugging, or for management or administrative purposes, use inbound NAT rules, a virtual network gateway, or a jumpbox.

For more information about connecting to virtual machines, see:

- [Run VMs for an N-tier architecture in Azure](#)
 - [Set up WinRM access for VMs in Azure Resource Manager](#)
 - [Allow external access to your VM by using the Azure portal](#)
 - [Allow external access to your VM by using PowerShell](#)
 - [Allow external access to your Linux VM by using Azure CLI](#)
- The **domainNameLabel** property for public IP addresses must be unique. The **domainNameLabel** value must be between 3 and 63 characters long, and follow the rules specified by this regular expression:
[a-z][a-z0-9-]{1,61}[a-z0-9]\$. Because the **uniqueString** function generates a string that is 13 characters long, the **dnsPrefixString** parameter is limited to 50 characters:

```
"parameters": {  
    "dnsPrefixString": {  
        "type": "string",  
        "maxLength": 50,  
        "metadata": {  
            "description": "The DNS label for the public IP address. It must be lowercase. It should  
match the following regular expression, or it will raise an error: [a-z][a-z0-9-]{1,61}[a-z0-9]$"  
        }  
    },  
    "variables": {  
        "dnsPrefix": "[concat(parameters('dnsPrefixString'),uniquestring(resourceGroup().id))]"  
    }  
}
```

- When you add a password to a custom script extension, use the **commandToExecute** property in the **protectedSettings** property:

```
"properties": {  
    "publisher": "Microsoft.Azure.Extensions",  
    "type": "CustomScript",  
    "typeHandlerVersion": "2.0",  
    "autoUpgradeMinorVersion": true,  
    "settings": {  
        "fileUris": [  
            "[concat(variables('template').assets, '/lamp-app/install_lamp.sh')]"  
        ]  
    },  
    "protectedSettings": {  
        "commandToExecute": "[concat('sh install_lamp.sh ', parameters('mySqlPassword'))]"  
    }  
}
```

NOTE

To ensure that secrets are encrypted when they are passed as parameters to VMs and extensions, use the **protectedSettings** property of the relevant extensions.

Outputs

If you use a template to create public IP addresses, include an **outputs** section that returns details of the IP address and the fully qualified domain name (FQDN). You can use output values to easily retrieve details about public IP addresses and FQDNs after deployment. When you reference the resource, use the API version that you used to create it:

```

"outputs": {
    "fqdn": {
        "value": "[reference(resourceId('Microsoft.Network/publicIPAddresses',parameters('publicIPAddressName')), '2016-07-01').dnsSettings.fqdn]",
        "type": "string"
    },
    "ipaddress": {
        "value": "[reference(resourceId('Microsoft.Network/publicIPAddresses',parameters('publicIPAddressName')), '2016-07-01').ipAddress]",
        "type": "string"
    }
}

```

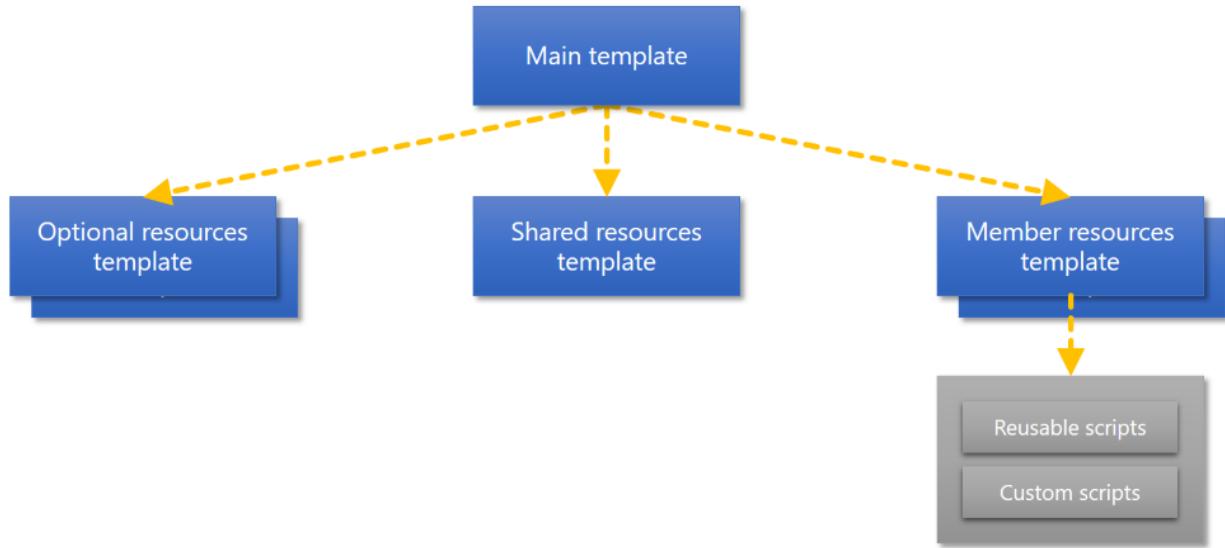
Single template vs. nested templates

To deploy your solution, you can use either a single template or a main template with multiple nested templates. Nested templates are common for more advanced scenarios. Using a nested template gives you the following advantages:

- You can break down a solution into targeted components.
- You can reuse nested templates with different main templates.

If you choose to use nested templates, the following guidelines can help you standardize your template design. These guidelines are based on [patterns for designing Azure Resource Manager templates](#). We recommend a design that has the following templates:

- **Main template** (azuredeploy.json). Use for the input parameters.
- **Shared resources template**. Use to deploy shared resources that all other resources use (for example, virtual network and availability sets). Use the **dependsOn** expression to ensure that this template is deployed before other templates.
- **Optional resources template**. Use to conditionally deploy resources based on a parameter (for example, a jumpbox).
- **Member resources template**. Each instance type within an application tier has its own configuration. Within a tier, you can define different instance types. (For example, the first instance creates a cluster, and additional instances are added to the existing cluster.) Each instance type has its own deployment template.
- **Scripts**. Widely reusable scripts are applicable for each instance type (for example, initialize and format additional disks). Custom scripts that you create for a specific customization purpose are different, based on the instance type.



For more information, see [Use linked templates with Azure Resource Manager](#).

Conditionally link to nested templates

You can use a parameter to conditionally link to nested templates. The parameter becomes part of the URI for the template:

```

"parameters": {
  "newOrExisting": {
    "type": "String",
    "allowedValues": [
      "new",
      "existing"
    ]
  }
},
"variables": {
  "templatelink": "[concat('https://raw.githubusercontent.com/Contoso/Templates/master/', parameters('newOrExisting'), 'StorageAccount.json')]"
},
"resources": [
  {
    "apiVersion": "2015-01-01",
    "name": "nestedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
      "mode": "incremental",
      "templateLink": {
        "uri": "[variables('templatelink')]",
        "contentVersion": "1.0.0.0"
      },
      "parameters": {}
    }
  }
]

```

Template format

It's a good practice to pass your template through a JSON validator. A validator can help you remove extraneous commas, parentheses, and brackets that might cause an error during deployment. Try [JSONLint](#) or a linter package for your favorite editing environment (Visual Studio Code, Atom, Sublime Text, Visual Studio).

It's also a good idea to format your JSON for better readability. You can use a JSON formatter package for your local editor. In Visual Studio, to format the document, press **Ctrl+K, Ctrl+D**. In Visual Studio Code, press **Alt+Shift+F**. If your local editor doesn't format the document, you can use an [online formatter](#).

Next steps

- For guidance on architecting your solution for virtual machines, see [Run a Windows VM in Azure](#) and [Run a Linux VM in Azure](#).
- For guidance on setting up a storage account, see [Azure Storage performance and scalability checklist](#).
- For help with virtual networks, see the [networking infrastructure guidelines](#).
- To learn about how an enterprise can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold: Prescriptive subscription governance](#).

Understand the structure and syntax of Azure Resource Manager templates

4/13/2017 • 9 min to read • [Edit Online](#)

This topic describes the structure of an Azure Resource Manager template. It presents the different sections of a template and the properties that are available in those sections. The template consists of JSON and expressions that you can use to construct values for your deployment. For a step-by-step tutorial on creating a template, see [Create your first Azure Resource Manager template](#).

Limit the size of your template to 1 MB, and each parameter file to 64 KB. The 1-MB limit applies to the final state of the template after it has been expanded with iterative resource definitions, and values for variables and parameters.

Template format

In its simplest structure, a template contains the following elements:

```
{  
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
  "contentVersion": "",  
  "parameters": { },  
  "variables": { },  
  "resources": [ ],  
  "outputs": { }  
}
```

ELEMENT NAME	REQUIRED	DESCRIPTION
\$schema	Yes	Location of the JSON schema file that describes the version of the template language. Use the URL shown in the preceding example.
contentVersion	Yes	Version of the template (such as 1.0.0.0). You can provide any value for this element. When deploying resources using the template, this value can be used to make sure that the right template is being used.
parameters	No	Values that are provided when deployment is executed to customize resource deployment.
variables	No	Values that are used as JSON fragments in the template to simplify template language expressions.
resources	Yes	Resource types that are deployed or updated in a resource group.

ELEMENT NAME	REQUIRED	DESCRIPTION
outputs	No	Values that are returned after deployment.

Each element contains properties you can set. The following example contains the full syntax for a template:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "",
    "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>"
            }
        }
    },
    "variables": {
        "<variable-name>": "<variable-value>",
        "<variable-name>": {
            <variable-complex-type-value>
        }
    },
    "resources": [
        {
            "apiVersion": "<api-version-of-resource>",
            "type": "<resource-provider-namespace/resource-type-name>",
            "name": "<name-of-the-resource>",
            "location": "<location-of-resource>",
            "tags": "<name-value-pairs-for-resource-tagging>",
            "comments": "<your-reference-notes>",
            "dependsOn": [
                "<array-of-related-resource-names>"
            ],
            "properties": "<settings-for-the-resource>",
            "copy": {
                "name": "<name-of-copy-loop>",
                "count": "<number-of-iterations>"
            },
            "resources": [
                "<array-of-child-resources>"
            ]
        }
    ],
    "outputs": {
        "<outputName>": {
            "type" : "<type-of-output-value>",
            "value": "<output-value-expression>"
        }
    }
}
```

We examine the sections of the template in greater detail later in this topic.

Expressions and functions

The basic syntax of the template is JSON. However, expressions and functions extend the JSON values available

within the template. Expressions are written within JSON string literals whose first and last characters are the brackets: [and], respectively. The value of the expression is evaluated when the template is deployed. While written as a string literal, the result of evaluating the expression can be of a different JSON type, such as an array or integer, depending on the actual expression. To have a literal string start with a bracket [, but not have it interpreted as an expression, add an extra bracket to start the string with [[].

Typically, you use expressions with functions to perform operations for configuring the deployment. Just like in JavaScript, function calls are formatted as `functionName(arg1,arg2,arg3)`. You reference properties by using the dot and [index] operators.

The following example shows how to use several functions when constructing values:

```
"variables": {  
    "location": "[resourceGroup().location]",  
    "usernameAndPassword": "[concat(parameters('username'), ':', parameters('password'))]",  
    "authorizationHeader": "[concat('Basic ', base64(variables('usernameAndPassword')))]"  
}
```

For the full list of template functions, see [Azure Resource Manager template functions](#).

Parameters

In the parameters section of the template, you specify which values you can input when deploying the resources. These parameter values enable you to customize the deployment by providing values that are tailored for a particular environment (such as dev, test, and production). You do not have to provide parameters in your template, but without parameters your template would always deploy the same resources with the same names, locations, and properties.

You define parameters with the following structure:

```
"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>"  
        }  
    }  
}
```

ELEMENT NAME	REQUIRED	DESCRIPTION
parameterName	Yes	Name of the parameter. Must be a valid JavaScript identifier.
type	Yes	Type of the parameter value. See the list of allowed types after this table.
defaultValue	No	Default value for the parameter, if no value is provided for the parameter.

ELEMENT NAME	REQUIRED	DESCRIPTION
allowedValues	No	Array of allowed values for the parameter to make sure that the right value is provided.
minValue	No	The minimum value for int type parameters, this value is inclusive.
maxValue	No	The maximum value for int type parameters, this value is inclusive.
minLength	No	The minimum length for string, secureString, and array type parameters, this value is inclusive.
maxLength	No	The maximum length for string, secureString, and array type parameters, this value is inclusive.
description	No	Description of the parameter that is displayed to users through the portal.

The allowed types and values are:

- **string**
- **secureString**
- **int**
- **bool**
- **object**
- **secureObject**
- **array**

To specify a parameter as optional, provide a defaultValue (can be an empty string).

If you specify a parameter name in your template that matches a parameter in the command to deploy the template, there is potential ambiguity about the values you provide. Resource Manager resolves this confusion by adding the postfix **FromTemplate** to the template parameter. For example, if you include a parameter named **ResourceGroupName** in your template, it conflicts with the **ResourceGroupName** parameter in the [New-AzureRmResourceGroupDeployment](#) cmdlet. During deployment, you are prompted to provide a value for **ResourceGroupNameFromTemplate**. In general, you should avoid this confusion by not naming parameters with the same name as parameters used for deployment operations.

NOTE

All passwords, keys, and other secrets should use the **secureString** type. If you pass sensitive data in a JSON object, use the **secureObject** type. Template parameters with secureString or secureObject types cannot be read after resource deployment.

For example, the following entry in the deployment history shows the value for a string and object but not for secureString and secureObject.

Inputs	
USERNAME	Example Person
PASSWORD	
REGULAROBJECT	{"test": "a"}
SECUREOBJECT	

The following example shows how to define parameters:

```
"parameters": {  
    "siteName": {  
        "type": "string",  
        "defaultValue": "[concat('site', uniqueString(resourceGroup().id))]"  
    },  
    "hostingPlanName": {  
        "type": "string",  
        "defaultValue": "[concat(parameters('siteName'), '-plan')]"  
    },  
    "skuName": {  
        "type": "string",  
        "defaultValue": "F1",  
        "allowedValues": [  
            "F1",  
            "D1",  
            "B1",  
            "B2",  
            "B3",  
            "S1",  
            "S2",  
            "S3",  
            "P1",  
            "P2",  
            "P3",  
            "P4"  
        ]  
    },  
    "skuCapacity": {  
        "type": "int",  
        "defaultValue": 1,  
        "minValue": 1  
    }  
}
```

For how to input the parameter values during deployment, see [Deploy an application with Azure Resource Manager template](#).

Variables

In the variables section, you construct values that can be used throughout your template. You do not need to

define variables, but they often simplify your template by reducing complex expressions.

You define variables with the following structure:

```
"variables": {  
    "<variable-name>": "<variable-value>",  
    "<variable-name>": {  
        <variable-complex-type-value>  
    }  
}
```

The following example shows how to define a variable that is constructed from two parameter values:

```
"variables": {  
    "connectionString": "[concat('Name=', parameters('username'), ';Password=', parameters('password'))]"  
}
```

The next example shows a variable that is a complex JSON type, and variables that are constructed from other variables:

```
"parameters": {  
    "environmentName": {  
        "type": "string",  
        "allowedValues": [  
            "test",  
            "prod"  
        ]  
    }  
},  
"variables": {  
    "environmentSettings": {  
        "test": {  
            "instancesSize": "Small",  
            "instancesCount": 1  
        },  
        "prod": {  
            "instancesSize": "Large",  
            "instancesCount": 4  
        }  
    },  
    "currentEnvironmentSettings": "[variables('environmentSettings')[parameters('environmentName')]]",  
    "instancesSize": "[variables('currentEnvironmentSettings').instancesSize]",  
    "instancesCount": "[variables('currentEnvironmentSettings').instancesCount]"  
}
```

Resources

In the resources section, you define the resources that are deployed or updated. This section can get complicated because you must understand the types you are deploying to provide the right values. For the resource-specific values (apiVersion, type, and properties) that you need to set, see [Define resources in Azure Resource Manager templates](#).

You define resources with the following structure:

```

"resources": [
    {
        "apiVersion": "<api-version-of-resource>",
        "type": "<resource-provider-namespace/resource-type-name>",
        "name": "<name-of-the-resource>",
        "location": "<location-of-resource>",
        "tags": "<name-value-pairs-for-resource-tagging>",
        "comments": "<your-reference-notes>",
        "dependsOn": [
            "<array-of-related-resource-names>"
        ],
        "properties": "<settings-for-the-resource>",
        "copy": {
            "name": "<name-of-copy-loop>",
            "count": "<number-of-iterations>"
        },
        "resources": [
            "<array-of-child-resources>"
        ]
    }
]

```

ELEMENT NAME	REQUIRED	DESCRIPTION
apiVersion	Yes	Version of the REST API to use for creating the resource.
type	Yes	Type of the resource. This value is a combination of the namespace of the resource provider and the resource type (such as Microsoft.Storage/storageAccounts).
name	Yes	Name of the resource. The name must follow URI component restrictions defined in RFC3986. In addition, Azure services that expose the resource name to outside parties validate the name to make sure it is not an attempt to spoof another identity.
location	Varies	Supported geo-locations of the provided resource. You can select any of the available locations, but typically it makes sense to pick one that is close to your users. Usually, it also makes sense to place resources that interact with each other in the same region. Most resource types require a location, but some types (such as a role assignment) do not require a location. See Set resource location in Azure Resource Manager templates .
tags	No	Tags that are associated with the resource. See Tag resources in Azure Resource Manager templates .

ELEMENT NAME	REQUIRED	DESCRIPTION
comments	No	Your notes for documenting the resources in your template
dependsOn	No	Resources that must be deployed before this resource is deployed. Resource Manager evaluates the dependencies between resources and deploys them in the correct order. When resources are not dependent on each other, they are deployed in parallel. The value can be a comma-separated list of a resource names or resource unique identifiers. Only list resources that are deployed in this template. Resources that are not defined in this template must already exist. Avoid adding unnecessary dependencies as they can slow your deployment and create circular dependencies. For guidance on setting dependencies, see Defining dependencies in Azure Resource Manager templates .
properties	No	Resource-specific configuration settings. The values for the properties are the same as the values you provide in the request body for the REST API operation (PUT method) to create the resource.
copy	No	If more than one instance is needed, the number of resources to create. For more information, see Create multiple instances of resources in Azure Resource Manager .
resources	No	Child resources that depend on the resource being defined. Only provide resource types that are permitted by the schema of the parent resource. The fully qualified type of the child resource includes the parent resource type, such as Microsoft.Web/sites/extensions . Dependency on the parent resource is not implied. You must explicitly define that dependency.

The resources section contains an array of the resources to deploy. Within each resource, you can also define an array of child resources. Therefore, your resources section could have a structure like:

```

"resources": [
  {
    "name": "resourceA",
  },
  {
    "name": "resourceB",
    "resources": [
      {
        "name": "firstChildResourceB",
      },
      {
        "name": "secondChildResourceB",
      }
    ]
  },
  {
    "name": "resourceC",
  }
]

```

For more information about defining child resources, see [Set name and type for child resource in Resource Manager template](#).

Outputs

In the Outputs section, you specify values that are returned from deployment. For example, you could return the URI to access a deployed resource.

The following example shows the structure of an output definition:

```

"outputs": {
  "<outputName>" : {
    "type" : "<type-of-output-value>",
    "value": "<output-value-expression>"
  }
}

```

ELEMENT NAME	REQUIRED	DESCRIPTION
outputName	Yes	Name of the output value. Must be a valid JavaScript identifier.
type	Yes	Type of the output value. Output values support the same types as template input parameters.
value	Yes	Template language expression that is evaluated and returned as output value.

The following example shows a value that is returned in the Outputs section.

```
"outputs": {
    "siteUri" : {
        "type" : "string",
        "value": "[concat('http://',reference(resourceId('Microsoft.Web/sites',
parameters('siteName'))).hostNames[0])]"
    }
}
```

For more information about working with output, see [Sharing state in Azure Resource Manager templates](#).

Next Steps

- To view complete templates for many different types of solutions, see the [Azure Quickstart Templates](#).
- For details about the functions you can use from within a template, see [Azure Resource Manager Template Functions](#).
- To combine multiple templates during deployment, see [Using linked templates with Azure Resource Manager](#).
- You may need to use resources that exist within a different resource group. This scenario is common when working with storage accounts or virtual networks that are shared across multiple resource groups. For more information, see the [resourceId function](#).

Define the order for deploying resources in Azure Resource Manager templates

3/6/2017 • 5 min to read • [Edit Online](#)

For a given resource, there can be other resources that must exist before the resource is deployed. For example, a SQL server must exist before attempting to deploy a SQL database. You define this relationship by marking one resource as dependent on the other resource. You define a dependency with the **dependsOn** element, or by using the **reference** function.

Resource Manager evaluates the dependencies between resources, and deploys them in their dependent order. When resources are not dependent on each other, Resource Manager deploys them in parallel. You only need to define dependencies for resources that are deployed in the same template.

dependsOn

Within your template, the dependsOn element enables you to define one resource as a dependent on one or more resources. Its value can be a comma-separated list of resource names.

The following example shows a virtual machine scale set that depends on a load balancer, virtual network, and a loop that creates multiple storage accounts. These other resources are not shown in the following example, but they would need to exist elsewhere in the template.

```
{  
  "type": "Microsoft.Compute/virtualMachineScaleSets",  
  "name": "[variables('namingInfix')]",  
  "location": "[variables('location')]",  
  "apiVersion": "2016-03-30",  
  "tags": {  
    "displayName": "VMScaleSet"  
  },  
  "dependsOn": [  
    "[variables('loadBalancerName')]",  
    "[variables('virtualNetworkName')]",  
    "storageLoop",  
  ],  
  ...  
}
```

In the preceding example, a dependency is included on the resources that are created through a copy loop named **storageLoop**. For an example, see [Create multiple instances of resources in Azure Resource Manager](#).

When defining dependencies, you can include the resource provider namespace and resource type to avoid ambiguity. For example, to clarify a load balancer and virtual network that may have the same names as other resources, use the following format:

```
"dependsOn": [  
  "[concat('Microsoft.Network/loadBalancers/', variables('loadBalancerName'))]",  
  "[concat('Microsoft.Network/virtualNetworks/', variables('virtualNetworkName'))]"  
]
```

While you may be inclined to use dependsOn to map relationships between your resources, it's important to understand why you're doing it. For example, to document how resources are interconnected, dependsOn is not

the right approach. You cannot query which resources were defined in the dependsOn element after deployment. By using dependsOn, you potentially impact deployment time because Resource Manager does not deploy in parallel two resources that have a dependency. To document relationships between resources, instead use [resource linking](#).

Child resources

The resources property allows you to specify child resources that are related to the resource being defined. Child resources can only be defined five levels deep. It is important to note that an implicit dependency is not created between a child resource and the parent resource. If you need the child resource to be deployed after the parent resource, you must explicitly state that dependency with the dependsOn property.

Each parent resource accepts only certain resource types as child resources. The accepted resource types are specified in the [template schema](#) of the parent resource. The name of child resource type includes the name of the parent resource type, such as **Microsoft.Web/sites/config** and **Microsoft.Web/sites/extensions** are both child resources of the **Microsoft.Web/sites**.

The following example shows a SQL server and SQL database. Notice that an explicit dependency is defined between the SQL database and SQL server, even though the database is a child of the server.

```
"resources": [
  {
    "name": "[variables('sqlserverName')]",
    "type": "Microsoft.Sql/servers",
    "location": "[resourceGroup().location]",
    "tags": {
      "displayName": "SqlServer"
    },
    "apiVersion": "2014-04-01-preview",
    "properties": {
      "administratorLogin": "[parameters('administratorLogin')]",
      "administratorLoginPassword": "[parameters('administratorLoginPassword')]"
    },
    "resources": [
      {
        "name": "[parameters('databaseName')]",
        "type": "databases",
        "location": "[resourceGroup().location]",
        "tags": {
          "displayName": "Database"
        },
        "apiVersion": "2014-04-01-preview",
        "dependsOn": [
          "[variables('sqlserverName')]"
        ],
        "properties": {
          "edition": "[parameters('edition')]",
          "collation": "[parameters('collation')]",
          "maxSizeBytes": "[parameters('maxSizeBytes')]",
          "requestedServiceObjectiveName": "[parameters('requestedServiceObjectiveName')]"
        }
      }
    ]
  }
]
```

reference function

The [reference function](#) enables an expression to derive its value from other JSON name and value pairs or runtime resources. Reference expressions implicitly declare that one resource depends on another. The general format is:

```
reference('resourceName').propertyPath
```

In the following example, a CDN endpoint explicitly depends on the CDN profile, and implicitly depends on a web app.

```
{
    "name": "[variables('endpointName')]",
    "type": "endpoints",
    "location": "[resourceGroup().location]",
    "apiVersion": "2016-04-02",
    "dependsOn": [
        "[variables('profileName')]"
    ],
    "properties": {
        "originHostHeader": "[reference(variables('webAppName')).hostNames[0]]",
        ...
    }
}
```

You can use either this element or the `dependsOn` element to specify dependencies, but you do not need to use both for the same dependent resource. Whenever possible, use an implicit reference to avoid adding an unnecessary dependency.

To learn more, see [reference function](#).

Recommendations for setting dependencies

When deciding what dependencies to set, use the following guidelines:

- Set as few dependencies as possible.
- Set a child resource as dependent on its parent resource.
- Use the **reference** function to set implicit dependencies between resources that need to share a property. Do not add an explicit dependency (**dependsOn**) when you have already defined an implicit dependency. This approach reduces the risk of having unnecessary dependencies.
- Set a dependency when a resource cannot be **created** without functionality from another resource. Do not set a dependency if the resources only interact after deployment.
- Let dependencies cascade without setting them explicitly. For example, your virtual machine depends on a virtual network interface, and the virtual network interface depends on a virtual network and public IP addresses. Therefore, the virtual machine is deployed after all three resources, but do not explicitly set the virtual machine as dependent on all three resources. This approach clarifies the dependency order and makes it easier to change the template later.
- If a value can be determined before deployment, try deploying the resource without a dependency. For example, if a configuration value needs the name of another resource, you might not need a dependency. This guidance does not always work because some resources verify the existence of the other resource. If you receive an error, add a dependency.

Resource Manager identifies circular dependencies during template validation. If you receive an error stating that a circular dependency exists, evaluate your template to see if any dependencies are not needed and can be removed. If removing dependencies does not work, you can avoid circular dependencies by moving some deployment operations into child resources that are deployed after the resources that have the circular dependency. For example, suppose you are deploying two virtual machines but you must set properties on each one that refer to the other. You can deploy them in the following order:

1. vm1
2. vm2

3. Extension on vm1 depends on vm1 and vm2. The extension sets values on vm1 that it gets from vm2.
4. Extension on vm2 depends on vm1 and vm2. The extension sets values on vm2 that it gets from vm1.

For information about assessing the deployment order and resolving dependency errors, see [Check deployment sequence](#).

Next steps

- To learn about troubleshooting dependencies during deployment, see [Troubleshoot common Azure deployment errors with Azure Resource Manager](#).
- To learn about creating Azure Resource Manager templates, see [Authoring templates](#).
- For a list of the available functions in a template, see [Template functions](#).

Set resource location in Azure Resource Manager templates

2/21/2017 • 1 min to read • [Edit Online](#)

When deploying a template, you must provide a location for each resource. This topic shows how to determine the locations that are available to your subscription for each resource type.

Determine supported locations

For a complete list of supported locations for each resource type, see [Products available by region](#). However, your subscription might not have access to all the locations in that list. To see a customized list of locations that are available to your subscription, use Azure PowerShell or Azure CLI.

The following example uses PowerShell to get the locations for the `Microsoft.Web\sites` resource type:

```
((Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | Where-Object ResourceTypeName -eq sites).Locations
```

The following example uses Azure CLI 2.0 to get the locations for the `Microsoft.Web\sites` resource type:

```
az provider show -n Microsoft.Web --query "resourceTypes[?resourceType=='sites'].locations"
```

Set location in template

After determining the supported locations for your resources, you need to set that location in your template. The easiest way to set this value is to create a resource group in a location that supports the resource types, and set each location to `[resourceGroup().location]`. You can redeploy the template to resource groups in different locations, and not change any values in the template or parameters.

The following example shows a storage account that is deployed to the same location as the resource group:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "variables": {
        "storageName": "[concat('storage', uniqueString(resourceGroup().id))]"
    },
    "resources": [
    {
        "apiVersion": "2016-01-01",
        "type": "Microsoft.Storage/storageAccounts",
        "name": "[variables('storageName')]",
        "location": "[resourceGroup().location]",
        "tags": {
            "Dept": "Finance",
            "Environment": "Production"
        },
        "sku": {
            "name": "Standard_LRS"
        },
        "kind": "Storage",
        "properties": { }
    }
]
}
```

If you need to hardcode the location in your template, provide the name of one of the supported regions. The following example shows a storage account that is always deployed to North Central US:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "resources": [
    {
        "apiVersion": "2016-01-01",
        "type": "Microsoft.Storage/storageAccounts",
        "name": "[concat('storageloc', uniqueString(resourceGroup().id))]",
        "location": "North Central US",
        "tags": {
            "Dept": "Finance",
            "Environment": "Production"
        },
        "sku": {
            "name": "Standard_LRS"
        },
        "kind": "Storage",
        "properties": { }
    }
]
```

Next Steps

- For recommendations about how to create templates, see [Best practices for creating Azure Resource Manager templates](#).

Tag resources in Azure Resource Manager templates

2/6/2017 • 2 min to read • [Edit Online](#)

You apply tags to your Azure resources to logically organize them by categories. Each tag consists of a key and a value. For example, you can apply the key "Environment" and the value "Production" to all the resources in production. Without this tag, you may have difficulty identifying whether a resource is intended for development, test, or production. However, "Environment" and "Production" are just examples. You define the keys and values that make the most sense for organizing your subscription.

After applying tags, you can retrieve all the resources in your subscription with that tag key and value. Tags enable you to retrieve related resources that reside in different resource groups. This approach is helpful when you need to organize resources for billing or management.

The following limitations apply to tags:

- Each resource or resource group can have a maximum of 15 tags.
- The tag name is limited to 512 characters.
- The tag value is limited to 256 characters.
- Tags applied to the resource group are not inherited by the resources in that resource group.

Add tags to your template

To tag a resource during deployment, add the `tags` element to the resource you are deploying. Provide the tag name and value.

Apply literal value to tag name

The following example shows a storage account with two tags (`Dept` and `Environment`) that are set to literal values:

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "resources": [  
    {  
      "apiVersion": "2016-01-01",  
      "type": "Microsoft.Storage/storageAccounts",  
      "name": "[concat('storage', uniqueString(resourceGroup().id))]",  
      "location": "[resourceGroup().location]",  
      "tags": {  
        "Dept": "Finance",  
        "Environment": "Production"  
      },  
      "sku": {  
        "name": "Standard_LRS"  
      },  
      "kind": "Storage",  
      "properties": { }  
    }  
  ]  
}
```

Apply object to tag element

You can define an object parameter that stores several tags, and apply that object to the tag element. Each property in the object becomes a separate tag for the resource. The following example has a parameter named `tagValues` that is applied to the tag element.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "tagValues": {
            "type": "object",
            "defaultValue": {
                "Dept": "Finance",
                "Environment": "Production"
            }
        }
    },
    "resources": [
        {
            "apiVersion": "2016-01-01",
            "type": "Microsoft.Storage/storageAccounts",
            "name": "[concat('storage', uniqueString(resourceGroup().id))]",
            "location": "[resourceGroup().location]",
            "tags": "[parameters('tagValues')]",
            "sku": {
                "name": "Standard_LRS"
            },
            "kind": "Storage",
            "properties": {}
        }
    ]
}
```

Apply JSON string to tag name

To store many values in a single tag, apply a JSON string that represents the values. The entire JSON string is stored as one tag that cannot exceed 256 characters. The following example has a single tag named `CostCenter` that contains several values from a JSON string:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "resources": [
        {
            "apiVersion": "2016-01-01",
            "type": "Microsoft.Storage/storageAccounts",
            "name": "[concat('storage', uniqueString(resourceGroup().id))]",
            "location": "[resourceGroup().location]",
            "tags": {
                "CostCenter": "{\"Dept\":\"Finance\", \"Environment\":\"Production\"}"
            },
            "sku": {
                "name": "Standard_LRS"
            },
            "kind": "Storage",
            "properties": { }
        }
    ]
}
```

Next Steps

- For information about managing tags, see [Use tags to organize your Azure resources](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Set name and type for child resource in Resource Manager template

3/2/2017 • 1 min to read • [Edit Online](#)

When creating a template, you frequently need to include a child resource that is related to a parent resource. For example, your template may include a SQL server and a database. The SQL server is the parent resource, and the database is the child resource.

The format of the child resource type is:

```
{resource-provider-namespace}/{parent-resource-type}/{child-resource-type}
```

The format of the child resource name is: `{parent-resource-name}/{child-resource-name}`

However, you specify the type and name in a template differently based on whether it is nested within the parent resource, or on its own at the top level. This topic shows how to handle both approaches.

Nested child resource

The easiest way to define a child resource is to nest it within the parent resource. The following example shows a SQL database nested within in a SQL Server.

```
{
  "name": "exampleserver",
  "type": "Microsoft.Sql/servers",
  "apiVersion": "2014-04-01",
  ...
  "resources": [
    {
      "name": "exampledatabase",
      "type": "databases",
      "apiVersion": "2014-04-01",
      ...
    }
  ]
}
```

For the child resource, the type is set to `databases` but its full resource type is `Microsoft.Sql/servers/databases`. You do not provide `Microsoft.Sql/servers/` because it is assumed from the parent resource type. The child resource name is set to `exampledatabase` but the full name includes the parent name. You do not provide `exampleserver` because it is assumed from the parent resource.

Top-level child resource

You can define the child resource at the top level. You might use this approach if the parent resource is not deployed in the same template, or if want to use `copy` to create multiple child resources. With this approach, you must provide the full resource type, and include the parent resource name in the child resource name.

```
{  
  "name": "exampleserver",  
  "type": "Microsoft.Sql/servers",  
  "apiVersion": "2014-04-01",  
  "resources": [  
    ],  
    ...  
  },  
  {  
    "name": "exampleserver/exampledatabase",  
    "type": "Microsoft.Sql/servers/databases",  
    "apiVersion": "2014-04-01",  
    ...  
  }  
}
```

The database is a child resource to the server even though they are defined on the same level in the template.

Next steps

- For recommendations about how to create templates, see [Best practices for creating Azure Resource Manager templates](#).
- For an example of creating multiple child resources, see [Deploy multiple instances of resources in Azure Resource Manager templates](#).

Deploy multiple instances of resources in Azure Resource Manager templates

2/27/2017 • 10 min to read • [Edit Online](#)

This topic shows you how to iterate in your Azure Resource Manager template to create multiple instances of a resource.

copy, copyIndex, and length

Within the resource to create multiple times, you can define a **copy** object that specifies the number of times to iterate. The copy takes the following format:

```
"copy": {  
    "name": "websitescopy",  
    "count": "[parameters('count')]"  
}
```

You can access the current iteration value with the **copyIndex()** function. The following example uses **copyIndex** with the concat function to construct a name.

```
[concat('examplecopy-', copyIndex())]
```

When creating multiple resources from an array of values, you can use the **length** function to specify the count. You provide the array as the parameter to the length function.

```
"copy": {  
    "name": "websitescopy",  
    "count": "[length(parameters('siteNames'))]"  
}
```

You can only apply the copy object to a top-level resource. You cannot apply it to a property on a resource type, or to a child resource. However, this topic shows how to specify multiple items for a property, and create multiple instances of a child resource. The following pseudo-code example shows where copy can be applied:

```

"resources": [
  {
    "type": "{provider-namespace-and-type}",
    "name": "parentResource",
    "copy": {
      /* yes, copy can be applied here */
    },
    "properties": {
      "exampleProperty": {
        /* no, copy cannot be applied here */
      }
    },
    "resources": [
      {
        "type": "{provider-type}",
        "name": "childResource",
        /* copy can be applied if resource is promoted to top level */
      }
    ]
  }
]

```

Although you cannot apply **copy** to a property, that property is still part of the iterations of the resource that contains the property. Therefore, you can use **copyIndex()** within the property to specify values.

There are several scenarios where you might want to iterate on a property in a resource. For example, you may want to specify multiple data disks for a virtual machine. To see how to iterate on a property, see [Create multiple instances when copy won't work](#).

To work with child resources, see [Create multiple instances of a child resource](#).

Use index value in name

You can use the copy operation create multiple instances of a resource that are uniquely named based on the incrementing index. For example, you might want to add a unique number to the end of each resource name that is deployed. To deploy three web sites named:

- examplecopy-0
- examplecopy-1
- examplecopy-2.

Use the following template:

```

"parameters": {
  "count": {
    "type": "int",
    "defaultValue": 3
  }
},
"resources": [
  {
    "name": "[concat('examplecopy-', copyIndex())]",
    "type": "Microsoft.Web/sites",
    "location": "East US",
    "apiVersion": "2015-08-01",
    "copy": {
      "name": "websitetscopy",
      "count": "[parameters('count')]"
    },
    "properties": {
      "serverFarmId": "hostingPlanName"
    }
  }
]

```

Offset index value

In the preceding example, the index value goes from zero to 2. To offset the index value, you can pass a value in the **copyIndex()** function, such as **copyIndex(1)**. The number of iterations to perform is still specified in the copy element, but the value of copyIndex is offset by the specified value. So, using the same template as the previous example, but specifying **copyIndex(1)** would deploy three web sites named:

- examplecopy-1
- examplecopy-2
- examplecopy-3

Use copy with array

The copy operation is helpful when working with arrays because you can iterate through each element in the array. To deploy three web sites named:

- examplecopy-Contoso
- examplecopy-Fabrikam
- examplecopy-Coho

Use the following template:

```

"parameters": {
    "org": {
        "type": "array",
        "defaultValue": [
            "Contoso",
            "Fabrikam",
            "Coho"
        ]
    }
},
"resources": [
{
    "name": "[concat('examplecopy-', parameters('org')[copyIndex()])]",
    "type": "Microsoft.Web/sites",
    "location": "East US",
    "apiVersion": "2015-08-01",
    "copy": {
        "name": "websitetescopy",
        "count": "[length(parameters('org'))]"
    },
    "properties": {
        "serverFarmId": "hostingPlanName"
    }
}
]

```

Of course, you can set the copy count to a value other than the length of the array. For example, you could create an array with many values, and then pass in a parameter value that specifies how many of the array elements to deploy. In that case, you set the copy count as shown in the first example.

Depend on resources in a loop

You can specify that a resource is deployed after another resource by using the **dependsOn** element. To deploy a resource that depends on the collection of resources in a loop, provide the name of the copy loop in the **dependsOn** element. The following example shows how to deploy three storage accounts before deploying the Virtual Machine. The full Virtual Machine definition is not shown. Notice that the copy element has **name** set to **storagecopy** and the **dependsOn** element for the Virtual Machines is also set to **storagecopy**.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {},
    "resources": [
        {
            "apiVersion": "2015-06-15",
            "type": "Microsoft.Storage/storageAccounts",
            "name": "[concat('storage', uniqueString(resourceGroup().id), copyIndex())]",
            "location": "[resourceGroup().location]",
            "properties": {
                "accountType": "Standard_LRS"
            },
            "copy": {
                "name": "storagecopy",
                "count": 3
            }
        },
        {
            "apiVersion": "2015-06-15",
            "type": "Microsoft.Compute/virtualMachines",
            "name": "[concat('VM', uniqueString(resourceGroup().id))]",
            "dependsOn": ["storagecopy"],
            ...
        },
        ],
        "outputs": {}
    }
}
```

Create multiple instances of a child resource

You cannot use a copy loop for a child resource. To create multiple instances of a resource that you typically define as nested within another resource, you must instead create that resource as a top-level resource. You define the relationship with the parent resource through the **type** and **name** properties.

For example, suppose you typically define a dataset as a child resource within a data factory.

```
"resources": [
{
    "type": "Microsoft.DataFactory/datafactories",
    "name": "exampleDataFactory",
    ...
    "resources": [
        {
            "type": "datasets",
            "name": "exampleDataSet",
            "dependsOn": [
                "exampleDataFactory"
            ],
            ...
        }
    ]
}]
```

To create multiple instances of data sets, move it outside of the data factory. The dataset must be at the same level as the data factory, but it is still a child resource of the data factory. You preserve the relationship between data set and data factory through the **type** and **name** properties. Since type can no longer be inferred from its position in the template, you must provide the fully qualified type in the format:

`{resource-provider-namespace}/{parent-resource-type}/{child-resource-type} .`

To establish a parent/child relationship with an instance of the data factory, provide a name for the data set that includes the parent resource name. Use the format: `{parent-resource-name}/{child-resource-name} .`

The following example shows the implementation:

```
"resources": [
{
  "type": "Microsoft.DataFactory/datafactories",
  "name": "exampleDataFactory",
  ...
},
{
  "type": "Microsoft.DataFactory/datafactories/datasets",
  "name": "[concat('exampleDataFactory', '/', 'exampleDataSet', copyIndex())]",
  "dependsOn": [
    "exampleDataFactory"
  ],
  "copy": {
    "name": "datasetcopy",
    "count": "3"
  }
  ...
}
]
```

Create multiple instances when copy won't work

You can only use **copy** on resource types, not on properties within a resource type. This requirement may create problems for you when you want to create multiple instances of something that is part of a resource. A common scenario is to create multiple data disks for a Virtual Machine. You cannot use **copy** with the data disks because **dataDisks** is a property on the Virtual Machine, not its own resource type. Instead, you create an array with as many data disks as you need, and pass in the actual number of data disks to create. In the virtual machine definition, you use the **take** function to get only the number of elements that you actually want from the array.

A full example of this pattern is show in the [Create a VM with a dynamic selection of data disks](#) template.

The relevant sections of the deployment template are shown in the following example. Much of the template has been removed to highlight the sections involved in dynamically creating a number of data disks. Notice the parameter **numDataDisks** that enables you to pass in the number of disks to create.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    ...
    "numDataDisks": {
      "type": "int",
      "maxValue": 64,
      "metadata": {
        "description": "This parameter allows you to select the number of disks you want"
      }
    }
  },
  "variables": {
    "storageAccountName": "[concat(uniquestring(resourceGroup().id), 'dynamicdisk')]",
    "sizeOfDataDisksInGB": 100,
    "diskCaching": "ReadWrite",
    "diskArray": [
      {
        "name": "datadisk1",
        "lun": 0,
        "vhd": {
          "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk1.vhd')]"
        },
        "createOption": "Empty",
        "caching": "[variables('diskCaching')]"
      }
    ]
  }
}
```

```

    "caching": "[variables('diskCaching')]" ,
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
},
{
    "name": "datadisk2",
    "lun": 1,
    "vhd": {
        "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk2.vhd')]"
    },
    "createOption": "Empty",
    "caching": "[variables('diskCaching')]",
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
},
{
    "name": "datadisk3",
    "lun": 2,
    "vhd": {
        "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk3.vhd')]"
    },
    "createOption": "Empty",
    "caching": "[variables('diskCaching')]",
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
},
{
    "name": "datadisk4",
    "lun": 3,
    "vhd": {
        "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk4.vhd')]"
    },
    "createOption": "Empty",
    "caching": "[variables('diskCaching')]",
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
},
...
{
    "name": "datadisk63",
    "lun": 62,
    "vhd": {
        "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk63.vhd')]"
    },
    "createOption": "Empty",
    "caching": "[variables('diskCaching')]",
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
},
{
    "name": "datadisk64",
    "lun": 63,
    "vhd": {
        "uri": "[concat('http://', variables('storageAccountName'), '.blob.core.windows.net/vhds/',
'datadisk64.vhd')]"
    },
    "createOption": "Empty",
    "caching": "[variables('diskCaching')]",
    "diskSizeGB": "[variables('sizeOfDataDisksInGB')]"
}
]
},
"resources": [
...
{
    "type": "Microsoft.Compute/virtualMachines",
    "properties": {
        ...
        "storageProfile": {
            ...

```

```

        "dataDisks": "[take(variables('diskArray'),parameters('numDataDisks'))]"
    },
    ...
}
...
]
}

```

You can use the **take** function and the **copy** element together when you need to create multiple instances of a resource with a variable number of items for a property. For example, suppose you need to create multiple virtual machines, but each virtual machine has a different number of data disks. To give each data disk a name that identifies the associated virtual machine, put your array of data disks into a separate template. Include parameters for the virtual machine name, and the number of data disks to return. In the outputs section, return the number of specified items.

```

{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "vmName": {
      "type": "string"
    },
    "storageAccountName": {
      "type": "string"
    },
    "numDataDisks": {
      "type": "int",
      "maxLength": 16,
      "metadata": {
        "description": "This parameter allows the user to select the number of disks they want"
      }
    }
  },
  "variables": {
    "diskArray": [
      {
        "name": "[concat(parameters('vmName'), '-datadisk1')]",
        "vhd": {
          "uri": "[concat('http://', parameters('storageAccountName'), '.blob.core.windows.net/vhds/', parameters('vmName'), '-datadisk1.vhd')]"
        },
        ...
      },
      ...
    ],
    "resources": [
    ],
    "outputs": {
      "result": {
        "type": "array",
        "value": "[take(variables('diskArray'),parameters('numDataDisks'))]"
      }
    }
  }
}

```

In the parent template, you include parameters for the number of virtual machines, and an array for the number of data disks for each virtual machine.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    ...
    "numberOfInstances": {
      "type": "int",
      "defaultValue": 2,
      "metadata": {
        "description": "Number of VMs to deploy"
      }
    },
    "numberOfDataDisksPerVM": {
      "type": "array",
      "defaultValue": [1,2]
    }
  },
}
```

In the resources section, deploy multiple instances of the template that defines the data disks.

```
{
  "apiVersion": "2016-09-01",
  "name": "[concat('nested-', copyIndex())]",
  "type": "Microsoft.Resources/deployments",
  "copy": {
    "name": "deploycopy",
    "count": "[parameters('numberOfInstances')]"
  },
  "properties": {
    "mode": "incremental",
    "templateLink": {
      "uri": "{data-disk-template-uri}",
      "contentVersion": "1.0.0.0"
    },
    "parameters": {
      "vmName": { "value": "[concat('myvm', copyIndex())]" },
      "storageAccountName": { "value": "[variables('storageAccountName')]" },
      "numDataDisks": { "value": "[parameters('numberOfDataDisksPerVM')[copyIndex()]]" }
    }
  }
},
```

In the resources section, deploy multiple instances of the virtual machine. For the data disks, reference the nested deployment that contains the correct number of data disks and the correct names for data disks.

```
{
  "type": "Microsoft.Compute/virtualMachines",
  "name": "[concat('myvm', copyIndex())]",
  "copy": {
    "name": "virtualMachineLoop",
    "count": "[parameters('numberOfInstances')]"
  },
  "properties": {
    "storageProfile": {
      ...
      "dataDisks": "[reference(concat('nested-', copyIndex())).outputs.result.value]"
    },
    ...
  },
  ...
}
```

Return values from a loop

While creating multiple instances of a resource type is convenient, returning values from that loop can be difficult. One way to retain and return values is to use **copy** with a nested template and round trip an array that contains all the values to return. For example, suppose you want to create multiple storage accounts, and return the primary endpoint for each one.

First, create the nested template that creates the storage account. Notice that it accepts an array parameter for the blob URIs. You use this parameter to round trip all the values from previous deployments. The output of the template is an array that concatenates the new blob URI to the previous URIs.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "indexValue": {
      "type": "int"
    },
    "blobURIs": {
      "type": "array",
      "defaultValue": []
    }
  },
  "variables": {
    "storageName": "[concat('storage', uniqueString(resourceGroup().id), parameters('indexValue'))]"
  },
  "resources": [
    {
      "apiVersion": "2016-01-01",
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[variables('storageName')]",
      "location": "[resourceGroup().location]",
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "Storage",
      "properties": {}
    }
  ],
  "outputs": {
    "result": {
      "type": "array",
      "value": "[concat(parameters('blobURIs'),split(reference(variables('storageName')).primaryEndpoints.blob, ','))]"
    }
  }
}
```

Now, create the parent template that has one static instance of the nested template, and loops over the remaining instances of the nested template. For each instance of the looped deployment, pass an array that is the output of the previous deployment.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "numberofStorage": { "type": "int", "minValue": 2 }
  },
  "resources": [
    {
      "apiVersion": "2016-09-01",
      "name": "nestedTemplate0",
      "type": "Microsoft.Resources/deployments",
      "properties": {
        "mode": "incremental",
        "templateLink": {
          "uri": "{storage-template-uri}",
          "contentVersion": "1.0.0.0"
        },
        "parameters": {
          "indexValue": {"value": 0}
        }
      }
    },
    {
      "apiVersion": "2016-09-01",
      "name": "[concat('nestedTemplate', copyIndex(1))]",
      "type": "Microsoft.Resources/deployments",
      "copy": {
        "name": "storagecopy",
        "count": "[sub(parameters('numberofStorage'), 1)]"
      },
      "properties": {
        "mode": "incremental",
        "templateLink": {
          "uri": "{storage-template-uri}",
          "contentVersion": "1.0.0.0"
        },
        "parameters": {
          "indexValue": {"value": "[copyIndex(1)]"},
          "blobURIs": {"value": "[reference(concat('nestedTemplate', copyIndex())).outputs.result.value]"}
        }
      }
    }
  ],
  "outputs": {
    "result": {
      "type": "object",
      "value": "[reference(concat('nestedTemplate', sub(parameters('numberofStorage'), 1))).outputs.result]"
    }
  }
}
```

Next steps

- If you want to learn about the sections of a template, see [Authoring Azure Resource Manager Templates](#).
- For all the functions you can use in a template, see [Azure Resource Manager Template Functions](#).
- To learn how to deploy your template, see [Deploy an application with Azure Resource Manager Template](#).

Using linked templates when deploying Azure resources

3/14/2017 • 7 min to read • [Edit Online](#)

From within one Azure Resource Manager template, you can link to another template, which enables you to decompose your deployment into a set of targeted, purpose-specific templates. As with decomposing an application into several code classes, decomposition provides benefits in terms of testing, reuse, and readability.

You can pass parameters from a main template to a linked template, and those parameters can directly map to parameters or variables exposed by the calling template. The linked template can also pass an output variable back to the source template, enabling a two-way data exchange between templates.

Linking to a template

You create a link between two templates by adding a deployment resource within the main template that points to the linked template. You set the **templateLink** property to the URI of the linked template. You can provide parameter values for the linked template directly in your template or in a parameter file. The following example uses the **parameters** property to specify a parameter value directly.

```
"resources": [
  {
    "apiVersion": "2015-01-01",
    "name": "linkedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
      "mode": "incremental",
      "templateLink": {
        "uri": "https://www.contoso.com/AzureTemplates/newStorageAccount.json",
        "contentVersion": "1.0.0.0"
      },
      "parameters": {
        "StorageAccountName": {"value": "[parameters('StorageAccountName')]"}
      }
    }
  }
]
```

Like other resource types, you can set dependencies between the linked template and other resources. Therefore, when other resources require an output value from the linked template, you can make sure the linked template is deployed before them. Or, when the linked template relies on other resources, you can make sure other resources are deployed before the linked template. You can retrieve a value from a linked template with the following syntax:

```
[reference('linkedTemplate').outputs.exampleProperty]
```

The Resource Manager service must be able to access the linked template. You cannot specify a local file or a file that is only available on your local network for the linked template. You can only provide a URI value that includes either **http** or **https**. One option is to place your linked template in a storage account, and use the URI for that item, such as shown in the following example:

```

"templateLink": {
  "uri": "http://mystorageaccount.blob.core.windows.net/templates/template.json",
  "contentVersion": "1.0.0.0",
}

```

Although the linked template must be externally available, it does not need to be generally available to the public. You can add your template to a private storage account that is accessible to only the storage account owner. Then, you create a shared access signature (SAS) token to enable access during deployment. You add that SAS token to the URI for the linked template. For steps on setting up a template in a storage account and generating a SAS token, see [Deploy resources with Resource Manager templates and Azure PowerShell](#) or [Deploy resources with Resource Manager templates and Azure CLI](#).

The following example shows a parent template that links to another template. The linked template is accessed with a SAS token that is passed in as a parameter.

```

"parameters": {
  "sasToken": { "type": "securestring" }
},
"resources": [
  {
    "apiVersion": "2015-01-01",
    "name": "linkedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
      "mode": "incremental",
      "templateLink": {
        "uri": "[concat('https://storagecontosotemplates.blob.core.windows.net/templates/helloworld.json',
parameters('sasToken'))]",
        "contentVersion": "1.0.0.0"
      }
    }
  }
],

```

Even though the token is passed in as a secure string, the URI of the linked template, including the SAS token, is logged in the deployment operations. To limit exposure, set an expiration for the token.

Resource Manager handles each linked template as a separate deployment. In the deployment history for the resource group, you see separate deployments for the parent and nested templates.

The screenshot shows the Azure portal's deployment history interface. On the left, there's a navigation bar with 'Add', 'Columns', 'Delete', 'Refresh', and 'Move' buttons. Below that is a section titled 'Essentials' with 'Subscription name: Third Internal Consumption', 'Deployment ID: 2 Succeeded', and a 'Filter by name...' search bar. On the right, a vertical sidebar shows deployment history for a resource group named 'ExampleGroup'. It lists two entries: 'parentTemplate' (status: succeeded, timestamp: 11/28/2016 8:11:34 AM) and 'nestedTemplate' (status: succeeded, timestamp: 11/28/2016 8:11:30 AM). Both entries have green checkmarks next to them.

Linking to a parameter file

The next example uses the **parametersLink** property to link to a parameter file.

```

"resources": [
    {
        "apiVersion": "2015-01-01",
        "name": "linkedTemplate",
        "type": "Microsoft.Resources/deployments",
        "properties": {
            "mode": "incremental",
            "templateLink": {
                "uri": "https://www.contoso.com/AzureTemplates/newStorageAccount.json",
                "contentVersion": "1.0.0.0"
            },
            "parametersLink": {
                "uri": "https://www.contoso.com/AzureTemplates/parameters.json",
                "contentVersion": "1.0.0.0"
            }
        }
    }
]

```

The URI value for the linked parameter file cannot be a local file, and must include either **http** or **https**. The parameter file can also be limited to access through a SAS token.

Using variables to link templates

The previous examples showed hard-coded URL values for the template links. This approach might work for a simple template but it does not work well when working with a large set of modular templates. Instead, you can create a static variable that stores a base URL for the main template and then dynamically create URLs for the linked templates from that base URL. The benefit of this approach is you can easily move or fork the template because you only need to change the static variable in the main template. The main template passes the correct URIs throughout the decomposed template.

The following example shows how to use a base URL to create two URLs for linked templates (**sharedTemplateUrl** and **vmTemplate**).

```

"variables": {
    "templateBaseUrl": "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/postgresql-on-ubuntu/",
    "sharedTemplateUrl": "[concat(variables('templateBaseUrl'), 'shared-resources.json')]",
    "vmTemplateUrl": "[concat(variables('templateBaseUrl'), 'database-2disk-resources.json')]"
}

```

You can also use [deployment\(\)](#) to get the base URL for the current template, and use that to get the URL for other templates in the same location. This approach is useful if your template location changes (maybe due to versioning) or you want to avoid hard coding URLs in the template file.

```

"variables": {
    "sharedTemplateUrl": "[uri(deployment().properties.templateLink.uri, 'shared-resources.json')]"
}

```

Conditionally linking to templates

You can link to different templates by passing in a parameter value that is used to construct the URI of the linked template. This approach works well when you need to specify during deployment the linked template to use. For example, you can specify one template to use for an existing storage account, and another template to use for a new storage account.

The following example shows a parameter for a storage account name, and a parameter to specify whether the

storage account is new or existing.

```
"parameters": {  
    "storageAccountName": {  
        "type": "String"  
    },  
    "newOrExisting": {  
        "type": "String",  
        "allowedValues": [  
            "new",  
            "existing"  
        ]  
    }  
},
```

You create a variable for the template URI that includes the value of the new or existing parameter.

```
"variables": {  
    "templatelink": "  
[concat('https://raw.githubusercontent.com/exampleuser/templates/master/','parameters('newOrExisting'),'Storage  
Account.json')]"  
},
```

You provide that variable value for the deployment resource.

```
"resources": [  
    {  
        "apiVersion": "2015-01-01",  
        "name": "linkedTemplate",  
        "type": "Microsoft.Resources/deployments",  
        "properties": {  
            "mode": "incremental",  
            "templateLink": {  
                "uri": "[variables('templatelink')]",  
                "contentVersion": "1.0.0.0"  
            },  
            "parameters": {  
                "StorageAccountName": {  
                    "value": "[parameters('storageAccountName')]"  
                }  
            }  
        }  
    }  
],
```

The URI resolves to a template named either **existingStorageAccount.json** or **newStorageAccount.json**. Create templates for those URIs.

The following example shows the **existingStorageAccount.json** template.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
      "type": "String"
    }
  },
  "variables": {},
  "resources": [],
  "outputs": {
    "storageAccountInfo": {
      "value": "[reference(concat('Microsoft.Storage/storageAccounts/', parameters('storageAccountName')), providers('Microsoft.Storage', 'storageAccounts').apiVersions[0]))]",
      "type": "object"
    }
  }
}
```

The next example shows the **newStorageAccount.json** template. Notice that like the existing storage account template the storage account object is returned in the outputs. The master template works with either linked template.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
      "type": "string"
    }
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[parameters('StorageAccountName')]",
      "apiVersion": "2016-01-01",
      "location": "[resourceGroup().location]",
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "Storage",
      "properties": {}
    }
  ],
  "outputs": {
    "storageAccountInfo": {
      "value": "[reference(concat('Microsoft.Storage/storageAccounts/', parameters('StorageAccountName')), providers('Microsoft.Storage', 'storageAccounts').apiVersions[0]))]",
      "type": "object"
    }
  }
}
```

Complete example

The following example templates show a simplified arrangement of linked templates to illustrate several of the concepts in this article. It assumes the templates have been added to the same container in a storage account with public access turned off. The linked template passes a value back to the main template in the **outputs** section.

The **parent.json** file consists of:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "containerSasToken": { "type": "string" }
  },
  "resources": [
    {
      "apiVersion": "2015-01-01",
      "name": "linkedTemplate",
      "type": "Microsoft.Resources/deployments",
      "properties": {
        "mode": "incremental",
        "templateLink": {
          "uri": "[concat(uri(deployment().properties.templateLink.uri, 'helloworld.json'), parameters('containerSasToken'))]",
          "contentVersion": "1.0.0.0"
        }
      }
    }
  ],
  "outputs": {
    "result": {
      "type": "object",
      "value": "[reference('linkedTemplate').outputs.result]"
    }
  }
}
```

The **helloworld.json** file consists of:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {},
  "variables": {},
  "resources": [],
  "outputs": {
    "result": {
      "value": "Hello World",
      "type" : "string"
    }
  }
}
```

In PowerShell, you get a token for the container and deploy the templates with:

```
Set-AzureRmCurrentStorageAccount -ResourceGroupName ManageGroup -Name storagecontosotemplates
$token = New-AzureStorageContainerSASToken -Name templates -Permission r -ExpiryTime (Get-Date).AddMinutes(30.0)
$url = (Get-AzureStorageBlob -Container templates -Blob parent.json).ICloudBlob.uri.AbsoluteUri
New-AzureRmResourceGroupDeployment -ResourceGroupName ExampleGroup -TemplateUri ($url + $token) -
containerSasToken $token
```

In Azure CLI 2.0, you get a token for the container and deploy the templates with the following code:

```
seconds='@$(( $(date +%s) + 1800 ))'
expiretime=$(date +%Y-%m-%dT%H:%MZ --date=$seconds)
connection=$(az storage account show-connection-string \
--resource-group ManageGroup \
--name storagecontosotemplates \
--query connectionString)
token=$(az storage container generate-sas \
--name templates \
--expiry $expiretime \
--permissions r \
--output tsv \
--connection-string $connection)
url=$(az storage blob url \
--container-name templates \
--name parent.json \
--output tsv \
--connection-string $connection)
parameter='{"containerSasToken":{"value":"'?"$token'"}}'
az group deployment create --resource-group ExampleGroup --template-uri $url?$token --parameters $parameter
```

Next steps

- To learn about the defining the deployment order for your resources, see [Defining dependencies in Azure Resource Manager templates](#)
- To learn how to define one resource but create many instances of it, see [Create multiple instances of resources in Azure Resource Manager](#)

Share state to and from Azure Resource Manager templates

1/24/2017 • 9 min to read • [Edit Online](#)

This topic shows best practices for managing and sharing state within templates. The parameters and variables shown in this topic are examples of the type of objects you can define to conveniently organize your deployment requirements. From these examples, you can implement your own objects with property values that make sense for your environment.

This topic is part of a larger whitepaper. To read the full paper, download [World Class Resource Manager Templates Considerations and Proven Practices](#).

Provide standard configuration settings

Rather than offer a template that provides total flexibility and countless variations, a common pattern is to provide a selection of known configurations. In effect, users can select standard t-shirt sizes such as sandbox, small, medium, and large. Other examples of t-shirt sizes are product offerings, such as community edition or enterprise edition. In other cases, it may be workload-specific configurations of a technology – such as map reduce or no sql.

With complex objects, you can create variables that contain collections of data, sometimes known as "property bags" and use that data to drive the resource declaration in your template. This approach provides good, known configurations of varying sizes that are preconfigured for customers. Without known configurations, users of the template must determine cluster sizing on their own, factor in platform resource constraints, and do math to identify the resulting partitioning of storage accounts and other resources (due to cluster size and resource constraints). In addition to making a better experience for the customer, a few known configurations are easier to support and can help you deliver a higher level of density.

The following example shows how to define variables that contain complex objects for representing collections of data. The collections define values that are used for virtual machine size, network settings, operating system settings and availability settings.

```
"variables": {  
    "tshirtSize": "[variables(concat('tshirtSize', parameters('tshirtSize')))]",  
    "tshirtSizeSmall": {  
        "vmSize": "Standard_A1",  
        "diskSize": 1023,  
        "vmTemplate": "[concat(variables('templateBaseUrl'), 'database-2disk-resources.json')]",  
        "vmCount": 2,  
        "storage": {  
            "name": "[parameters('storageAccountNamePrefix')]",  
            "count": 1,  
            "pool": "db",  
            "map": [0,0],  
            "jumpbox": 0  
        }  
    },  
    "tshirtSizeMedium": {  
        "vmSize": "Standard_A3",  
        "diskSize": 1023,  
        "vmTemplate": "[concat(variables('templateBaseUrl'), 'database-8disk-resources.json')]",  
        "vmCount": 2,  
        "storage": {  
            "name": "[parameters('storageAccountNamePrefix')]",  
            "count": 2,  
            "pool": "db",  
            "map": [0,0]  
        }  
    }  
},  
"outputs": {}  
}
```

```

        "map": [0,1],
        "jumpbox": 0
    }
},
"tshirtSizeLarge": {
    "vmSize": "Standard_A4",
    "diskSize": 1023,
    "vmTemplate": "[concat(variables('templateBaseUrl'), 'database-16disk-resources.json')]",
    "vmCount": 3,
    "slaveCount": 2,
    "storage": {
        "name": "[parameters('storageAccountNamePrefix')]",
        "count": 2,
        "pool": "db",
        "map": [0,1,1],
        "jumpbox": 0
    }
},
"osSettings": {
    "scripts": [
        "[concat(variables('templateBaseUrl'), 'install_postgresql.sh')]",
        "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/shared_scripts/ubuntu/vm-disk-utils-0.1.sh"
    ],
    "imageReference": {
        "publisher": "Canonical",
        "offer": "UbuntuServer",
        "sku": "14.04.2-LTS",
        "version": "latest"
    }
},
"networkSettings": {
    "vnetName": "[parameters('virtualNetworkName')]",
    "addressPrefix": "10.0.0.0/16",
    "subnets": {
        "dmz": {
            "name": "dmz",
            "prefix": "10.0.0.0/24",
            "vnet": "[parameters('virtualNetworkName')]"
        },
        "data": {
            "name": "data",
            "prefix": "10.0.1.0/24",
            "vnet": "[parameters('virtualNetworkName')]"
        }
    }
},
"availabilitySetSettings": {
    "name": "pgsqlAvailabilitySet",
    "fdCount": 3,
    "udCount": 5
}
}
}

```

Notice that the **tshirtSize** variable concatenates the t-shirt size you provided through a parameter (**Small**, **Medium**, **Large**) to the text **tshirtSize**. You use this variable to retrieve the associated complex object variable for that t-shirt size.

You can then reference these variables later in the template. The ability to reference named-variables and their properties simplifies the template syntax, and makes it easy to understand context. The following example defines a resource to deploy by using the objects shown previously to set values. For example, the VM size is set by retrieving the value for `variables('tshirtSize').vmSize` while the value for the disk size is retrieved from `variables('tshirtSize').diskSize`. In addition, the URI for a linked template is set with the value for `variables('tshirtSize').vmTemplate`.

```

"name": "master-node",
"type": "Microsoft.Resources/deployments",
"apiVersion": "2015-01-01",
"dependsOn": [
    "[concat('Microsoft.Resources/deployments/', 'shared')]"
],
"properties": {
    "mode": "Incremental",
    "templateLink": {
        "uri": "[variables('tshirtSize').vmTemplate]",
        "contentVersion": "1.0.0.0"
    },
    "parameters": {
        "adminPassword": {
            "value": "[parameters('adminPassword')]"
        },
        "replicatorPassword": {
            "value": "[parameters('replicatorPassword')]"
        },
        "osSettings": {
            "value": "[variables('osSettings')]"
        },
        "subnet": {
            "value": "[variables('networkSettings').subnets.data]"
        },
        "commonSettings": {
            "value": {
                "region": "[parameters('region')]",
                "adminUsername": "[parameters('adminUsername')]",
                "namespace": "ms"
            }
        },
        "storageSettings": {
            "value": "[variables('tshirtSize').storage]"
        },
        "machineSettings": {
            "value": {
                "vmSize": "[variables('tshirtSize').vmSize]",
                "diskSize": "[variables('tshirtSize').diskSize]",
                "vmCount": 1,
                "availabilitySet": "[variables('availabilitySetSettings').name]"
            }
        },
        "masterIpAddress": {
            "value": "0"
        },
        "dbType": {
            "value": "MASTER"
        }
    }
}
}

```

Pass state to a template

You share state into a template through parameters that you provide directly during deployment.

The following table lists commonly used parameters in templates.

NAME	VALUE	DESCRIPTION
location	String from a constrained list of Azure regions	The location where the resources are deployed.

NAME	VALUE	DESCRIPTION
storageAccountNamePrefix	String	Unique DNS name for the Storage Account where the VM's disks are placed
domainName	String	Domain name of the publicly accessible jumpbox VM in the format: {domainName}. {location}.cloudapp.com For example: mydomainname.westus.cloudapp.azure.com
adminUsername	String	Username for the VMs
adminPassword	String	Password for the VMs
tshirtSize	String from a constrained list of offered t-shirt sizes	The named scale unit size to provision. For example, "Small", "Medium", "Large"
virtualNetworkName	String	Name of the virtual network that the consumer wants to use.
enableJumpbox	String from a constrained list (enabled/disabled)	Parameter that identifies whether to enable a jumpbox for the environment. Values: "enabled", "disabled"

The **tshirtSize** parameter used in the previous section is defined as:

```
"parameters": {
  "tshirtSize": {
    "type": "string",
    "defaultValue": "Small",
    "allowedValues": [
      "Small",
      "Medium",
      "Large"
    ],
    "metadata": {
      "Description": "T-shirt size of the MongoDB deployment"
    }
  }
}
```

Pass state to linked templates

When connecting to linked templates, you often use a mix of static and generated variables.

Static variables

Static variables are often used to provide base values, such as URLs, that are used throughout a template.

In the following template excerpt, `templateBaseUrl` specifies the root location for the template in GitHub. The next line builds a new variable `sharedTemplateUrl` that concatenates the base URL with the known name of the shared resources template. Below that line, a complex object variable is used to store a t-shirt size, where the base URL is concatenated to the known configuration template location and stored in the `vmTemplate` property.

The benefit of this approach is that if the template location changes, you only need to change the static variable in one place, which passes it throughout the linked templates.

```

"variables": {
    "templateBaseUrl": "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/postgresql-on-ubuntu/",
    "sharedTemplateUrl": "[concat(variables('templateBaseUrl'), 'shared-resources.json')]",
    "tshirtSizeSmall": {
        "vmSize": "Standard_A1",
        "diskSize": 1023,
        "vmTemplate": "[concat(variables('templateBaseUrl'), 'database-2disk-resources.json')]",
        "vmCount": 2,
        "slaveCount": 1,
        "storage": {
            "name": "[parameters('storageAccountNamePrefix')]",
            "count": 1,
            "pool": "db",
            "map": [0,0],
            "jumpbox": 0
        }
    }
}

```

Generated variables

In addition to static variables, several variables are generated dynamically. This section identifies some of the common types of generated variables.

tshirtSize

You are familiar with this generated variable from the examples above.

networkSettings

In a capacity, capability, or end-to-end scoped solution template, the linked templates typically create resources that exist on a network. One straightforward approach is to use a complex object to store network settings and pass them to linked templates.

An example of communicating network settings can be seen below.

```

"networkSettings": {
    "vnetName": "[parameters('virtualNetworkName')]",
    "addressPrefix": "10.0.0.0/16",
    "subnets": {
        "dmz": {
            "name": "dmz",
            "prefix": "10.0.0.0/24",
            "vnet": "[parameters('virtualNetworkName')]"
        },
        "data": {
            "name": "data",
            "prefix": "10.0.1.0/24",
            "vnet": "[parameters('virtualNetworkName')]"
        }
    }
}

```

availabilitySettings

Resources created in linked templates are often placed in an availability set. In the following example, the availability set name is specified and also the fault domain and update domain count to use.

```
"availabilitySetSettings": {
    "name": "pgsqlAvailabilitySet",
    "fdCount": 3,
    "udCount": 5
}
```

If you need multiple availability sets (for example, one for master nodes and another for data nodes), you can use a name as a prefix, specify multiple availability sets, or follow the model shown earlier for creating a variable for a specific t-shirt size.

storageSettings

Storage details are often shared with linked templates. In the example below, a *storageSettings* object provides details about the storage account and container names.

```
"storageSettings": {
    "vhdStorageAccountName": "[parameters('storageAccountName')]",
    "vhdContainerName": "[variables('vmStorageAccountContainerName')]",
    "destinationVhdsContainer": "[concat('https://', parameters('storageAccountName'),
variables('vmStorageAccountDomain'), '/', variables('vmStorageAccountContainerName'), '/')]"
}
```

osSettings

With linked templates, you may need to pass operating system settings to various nodes types across different known configuration types. A complex object is an easy way to store and share operating system information and also makes it easier to support multiple operating system choices for deployment.

The following example shows an object for *osSettings*:

```
"osSettings": {
    "imageReference": {
        "publisher": "Canonical",
        "offer": "UbuntuServer",
        "sku": "14.04.2-LTS",
        "version": "latest"
    }
}
```

machineSettings

A generated variable, *machineSettings* is a complex object containing a mix of core variables for creating a VM. The variables include administrator user name and password, a prefix for the VM names, and an operating system image reference.

```
"machineSettings": {
    "adminUsername": "[parameters('adminUsername')]",
    "adminPassword": "[parameters('adminPassword')]",
    "machineNamePrefix": "mongodb-",
    "osImageReference": {
        "publisher": "[variables('osFamilySpec').imagePublisher]",
        "offer": "[variables('osFamilySpec').imageOffer]",
        "sku": "[variables('osFamilySpec').imageSKU]",
        "version": "latest"
    }
},
```

Note that *osImageReference* retrieves the values from the *osSettings* variable defined in the main template. That means you can easily change the operating system for a VM—entirely or based on the preference of a template consumer.

vmScripts

The `vmScripts` object contains details about the scripts to download and execute on a VM instance, including outside and inside references. Outside references include the infrastructure. Inside references include the installed software installed and configuration.

You use the `scriptsToDelete` property to list the scripts to download to the VM. This object also contains references to command-line arguments for different types of actions. These actions include executing the default installation for each individual node, an installation that runs after all nodes are deployed, and any additional scripts that may be specific to a given template.

This example is from a template used to deploy MongoDB, which requires an arbiter to deliver high availability. The `arbiterNodeInstallCommand` has been added to `vmScripts` to install the arbiter.

The variables section is where you find the variables that define the specific text to execute the script with the proper values.

```
"vmScripts": {  
    "scriptsToDelete": [  
        "[concat(variables('scriptUrl'), 'mongodb-', variables('osFamilySpec').osName, '-install.sh')]",  
        "[concat(variables('sharedScriptUrl'), 'vm-disk-utils-0.1.sh')]"  
    ],  
    "regularNodeInstallCommand": "[variables('installCommand')]",  
    "lastNodeInstallCommand": "[concat(variables('installCommand'), ' -l')]",  
    "arbiterNodeInstallCommand": "[concat(variables('installCommand'), ' -a')]"  
},
```

Return state from a template

Not only can you pass data into a template, you can also share data back to the calling template. In the **outputs** section of a linked template, you can provide key/value pairs that can be consumed by the source template.

The following example shows how to pass the private IP address generated in a linked template.

```
"outputs": {  
    "masterip": {  
        "value": "  
[reference(concat(variables('nicName'),0)).ipConfigurations[0].properties.privateIPAddress]",  
        "type": "string"  
    }  
}
```

Within the main template, you can use that data with the following syntax:

```
"[reference('master-node').outputs.masterip.value]"
```

You can use this expression in either the outputs section or the resources section of the main template. You cannot use the expression in the variables section because it relies on the runtime state. To return this value from the main template, use:

```
"outputs": {  
    "masterIpAddress": {  
        "value": "[reference('master-node').outputs.masterip.value]",  
        "type": "string"  
    }  
}
```

For an example of using the outputs section of a linked template to return data disks for a virtual machine, see

Define authentication settings for virtual machine

You can use the same pattern shown previously for configuration settings to specify the authentication settings for a virtual machine. You create a parameter for passing in the type of authentication.

```
"parameters": {  
    "authenticationType": {  
        "allowedValues": [  
            "password",  
            "sshPublicKey"  
        ],  
        "defaultValue": "password",  
        "metadata": {  
            "description": "Authentication type"  
        },  
        "type": "string"  
    }  
}
```

You add variables for the different authentication types, and a variable to store which type is used for this deployment based on the value of the parameter.

```
"variables": {  
    "osProfile": "[variables(concat('osProfile', parameters('authenticationType')))]",  
    "osProfilepassword": {  
        "adminPassword": "[parameters('adminPassword')]",  
        "adminUsername": "notused",  
        "computerName": "[parameters('vmName')]",  
        "customData": "[base64(variables('customData'))]"  
    },  
    "osProfilesshPublicKey": {  
        "adminUsername": "notused",  
        "computerName": "[parameters('vmName')]",  
        "customData": "[base64(variables('customData'))]",  
        "linuxConfiguration": {  
            "disablePasswordAuthentication": "true",  
            "ssh": {  
                "publicKeys": [  
                    {  
                        "keyData": "[parameters('sshPublicKey')]",  
                        "path": "/home/notused/.ssh/authorized_keys"  
                    }  
                ]  
            }  
        }  
    }  
}
```

When defining the virtual machine, you set the **osProfile** to the variable you created.

```
{  
    "type": "Microsoft.Compute/virtualMachines",  
    ...  
    "osProfile": "[variables('osProfile')]"  
}
```

Next steps

- To learn about the sections of the template, see [Authoring Azure Resource Manager Templates](#)
- To see the functions that are available within a template, see [Azure Resource Manager Template Functions](#)

Design patterns for Azure Resource Manager templates when deploying complex solutions

3/30/2017 • 23 min to read • [Edit Online](#)

Using a flexible approach based on Azure Resource Manager templates, you can deploy complex topologies quickly and consistently. You can adapt these deployments easily as core offerings evolve or to accommodate variants for outlier scenarios or customers.

This topic is part of a larger whitepaper. To read the full paper, download [World Class Azure Resource Manager Templates Considerations and Proven Practices](#).

Templates combine the benefits of the underlying Azure Resource Manager with the adaptability and readability of JavaScript Object Notation (JSON). Using templates, you can:

- Deploy topologies and their workloads consistently.
- Manage all your resources in an application together using resource groups.
- Apply role-based access control (RBAC) to grant appropriate access to users, groups, and services.
- Use tagging associations to streamline tasks such as billing rollups.

This article provides details on consumption scenarios, architecture, and implementation patterns identified during our design sessions and real-world template implementations with Azure Customer Advisory Team (AzureCAT) customers. Far from academic, these approaches are proven practices informed by the development of templates for 12 of the top Linux-based OSS technologies, including: Apache Kafka, Apache Spark, Cloudera, Couchbase, Hortonworks HDP, DataStax Enterprise powered by Apache Cassandra, Elasticsearch, Jenkins, MongoDB, Nagios, PostgreSQL, Redis, and Nagios.

This article shares these proven practices to help you architect world class Azure Resource Manager templates.

In our work with customers, we have identified several Resource Manager template consumption experiences across enterprises, System Integrators (SIs), and CSVs. The following sections provide a high-level overview of common scenarios and patterns for different customer types.

Enterprises and system integrators

Within large organizations, we commonly see two consumers of Resource Manager templates: internal software development teams and corporate IT. We've found that the scenarios for the SIs map to the scenarios for Enterprises, so the same considerations apply.

Internal software development teams

If your team develops software to support your business, templates provide an easy way to quickly deploy technologies for use in business-specific solutions. You can also use templates to rapidly create training environments that enable team members to gain necessary skills.

You can use templates as-is or extend or compose them to accommodate your needs. Using tagging within templates, you can provide a billing summary with various views such as team, project, individual, and education.

Businesses often want software development teams to create a template for consistent deployment of a solution. The template facilitates constraints so certain items within that environment remain fixed and can't be overridden. For example, a bank might require a template to include RBAC so a programmer can't revise a banking solution to send data to a personal storage account.

Corporate IT

Corporate IT organizations typically use templates for delivering cloud capacity and cloud-hosted capabilities.

Cloud capacity

A common way for corporate IT groups to provide cloud capacity for teams is with "t-shirt sizes", which are standard offering sizes such as small, medium, and large. The t-shirt sized offerings can mix different resource types and quantities while providing a level of standardization that makes it possible to use templates. The templates deliver capacity in a consistent way that enforces corporate policies and uses tagging to provide chargeback to consuming organizations.

For example, you may need to provide development, test, or production environments within which the software development teams can deploy their solutions. The environment has a predefined network topology and elements that the software development teams cannot change, such as rules governing access to the public internet and packet inspection. You may also have organization-specific roles for these environments with distinct access rights for the environment.

Cloud-hosted capabilities

You can use templates to support cloud-hosted capabilities, including individual software packages or composite offerings that are offered to internal lines of business. An example of a composite offering would be analytics-as-a-service—analytics, visualization, and other technologies—delivered in an optimized, connected configuration on a predefined network topology.

Cloud-hosted capabilities are affected by the security and role considerations established by the cloud capacity offering on which they're built. These capabilities are offered as is or as a managed service. For the latter, access-constrained roles are required to enable access into the environment for management purposes.

Cloud service vendors

After talking to many CSVs, we identified multiple approaches you can take to deploy services for your customers and associated requirements.

CSV-hosted offering

If you host your offering in your own Azure subscription, two hosting approaches are common: deploying a distinct deployment for every customer or deploying scale units that underpin a shared infrastructure used for all customers.

- **Distinct deployments for each customer.** Distinct deployments per customer require fixed topologies of different known configurations. These deployments may have different virtual machine (VM) sizes, varying numbers of nodes, and different amounts of associated storage. Tagging of deployments is used for roll-up billing of each customer. RBAC may be enabled to allow customers access to aspects of their cloud environment.
- **Scale units in shared multi-tenant environments.** A template can represent a scale unit for multi-tenant environments. In this case, the same infrastructure is used to support all customers. The deployments represent a group of resources that deliver a level of capacity for the hosted offering, such as number of users and number of transactions. These scale units are increased or decreased as demand requires.

CSV offering injected into customer subscription

You may want to deploy your software into subscriptions owned by end customers. You can use templates to deploy distinct deployments into a customer's Azure account.

These deployments use RBAC so you can update and manage the deployment within the customer's account.

Azure Marketplace

To advertise and sell your offerings through a marketplace, such as Azure Marketplace, you can develop templates to deliver distinct types of deployments that run in a customer's Azure account. These distinct deployments can be typically described as a t-shirt size (small, medium, large), product/audience type (community, developer, enterprise), or feature type (basic, high availability). In some cases, these types allow you to specify certain

attributes of the deployment, such as VM type or number of disks.

OSS projects

Within open source projects, Resource Manager templates enable a community to deploy a solution quickly using proven practices. You can store templates in a GitHub repository so the community can revise them over time. Users deploy these templates in their own Azure subscriptions.

The following sections identify the things you need to consider before designing your solution.

Identifying what is outside and inside a VM

As you design your template, it's helpful to look at the requirements in terms of what's outside and inside the virtual machines (VMs):

- Outside means the VMs and other resources of your deployment, such as the network topology, tagging, references to the certs/secrets, and role-based access control. All these resources are part of your template.
- Inside means the installed software and overall desired state configuration. Other mechanisms, such as VM extensions or scripts, are used in whole or in part. These mechanisms may be identified and executed by the template but aren't in it.

Common examples of activities you would do "inside the box" include -

- Install or remove server roles and features
- Install and configure software at the node or cluster level
- Deploy websites on a web server
- Deploy database schemas
- Manage registry or other types of configuration settings
- Manage files and directories
- Start, stop, and manage processes and services
- Manage local groups and user accounts
- Install and manage packages (.msi, .exe, yum, etc.)
- Manage environment variables
- Run native scripts (Windows PowerShell, bash, etc.)

Desired state configuration (DSC)

Thinking about the internal state of your VMs beyond deployment, you want to make sure this deployment doesn't "drift" from the configuration that you have defined and checked into source control. This approach ensures your developers or operations staff don't make ad-hoc changes to an environment that are not vetted, tested, or recorded in source control. This control is important, because the manual changes are not in source control. They are also not part of the standard deployment and will impact future automated deployments of the software.

Beyond your internal employees, desired state configuration is also important from a security perspective. Hackers are regularly trying to compromise and exploit software systems. When successful, it's common to install files and otherwise change the state of a compromised system. Using desired state configuration, you can identify deltas between the desired and actual state and restore a known configuration.

There are resource extensions for the most popular mechanisms for DSC - PowerShell DSC, Chef, and Puppet. Each of these extensions can deploy the initial state of your VM and also be used to make sure the desired state is maintained.

Common template scopes

In our experience, we've seen three key solution templates scopes emerge. These three scopes – capacity,

capability, and end-to-end solution – are described in the following sections.

Capacity scope

A capacity scope delivers a set of resources in a standard topology that is pre-configured to be in compliance with regulations and policies. The most common example is deploying a standard development environment in an Enterprise IT or SI scenario.

Capability scope

A capability scope is focused on deploying and configuring a topology for a given technology. Common scenarios including technologies such as SQL Server, Cassandra, Hadoop.

End-to-end solution scope

An End-to-End Solution Scope is targeted beyond a single capability, and instead focused on delivering an end to end solution comprised of multiple capabilities.

A solution-scoped template scope manifests itself as a set of one or more capability-scoped templates with solution-specific resources, logic, and desired state. An example of a solution-scoped template is an end to end data pipeline solution template. The template might mix solution-specific topology and state with multiple capability-scoped solution templates such as Kafka, Storm, and Hadoop.

Choosing free-form vs. known configurations

You might initially think a template should give consumers the utmost flexibility, but many considerations affect the choice of whether to use free-form configurations vs. known configurations. This section identifies the key customer requirements and technical considerations that shaped the approach shared in this document.

Free-form configurations

On the surface, free-form configurations sound ideal. They allow you to select a VM type and provide an arbitrary number of nodes and attached disks for those nodes — and do so as parameters to a template. However, this approach is not ideal for some scenarios.

In [Sizes for virtual machines](#), the different VM types and available sizes are identified, and each of the number of durable disks (2, 4, 8, 16, or 32) that can be attached. Each attached disk provides 500 IOPS and multiples of these disks can be pooled for a multiplier of that number of IOPS. For example, 16 disks can be pooled to provide 8,000 IOPS. Pooling is done with configuration in the operating system, using Microsoft Windows Storage Spaces or redundant array of inexpensive disks (RAID) in Linux.

A free-form configuration enables the selection several VM instances, various VM types and sizes for those instances, various disks for the VM type, and one or more scripts to configure the VM contents.

It is common that a deployment may have multiple types of nodes, such as master and data nodes, so this flexibility is often provided for every node type.

As you start to deploy clusters of any significance, you begin to work with these complex scenarios. If you were deploying a Hadoop cluster, for example, with 8 master nodes and 200 data nodes, and pooled 4 attached disks on each master node and pooled 16 attached disks per data node, you would have 208 VMs and 3,232 disks to manage.

A storage account will throttle requests above its identified 20,000 transactions/second limit, so you should look at storage account partitioning and use calculations to determine the appropriate number of storage accounts to accommodate this topology. Given the multitude of combinations supported by the free-form approach, dynamic calculations are required to determine the appropriate partitioning. The Azure Resource Manager Template Language does not presently provide mathematical functions, so you must perform these calculations in code, generating a unique, hard-coded template with the appropriate details.

In enterprise IT and SI scenarios, someone must maintain the templates and support the deployed topologies for

one or more organizations. This additional overhead — different configurations and templates for each customer — is far from desirable.

You can use these templates to deploy environments in your customer's Azure subscription, but both corporate IT teams and CSVs typically deploy them into their own subscriptions, using a chargeback function to bill their customers. In these scenarios, the goal is to deploy capacity for multiple customers across a pool of subscriptions and keep deployments densely populated into the subscriptions to minimize subscription sprawl—that is, more subscriptions to manage. With truly dynamic deployment sizes, achieving this type of density requires careful planning and additional development for scaffolding work on behalf of the organization.

In addition, you can't create subscriptions via an API call but must do so manually through the portal. As the number of subscriptions increases, any resulting subscription sprawl requires human intervention—it can't be automated. With so much variability in the sizes of deployments, you would have to pre-provision a number of subscriptions manually to ensure subscriptions are available.

Considering all these factors, a truly free-form configuration is less appealing than at first blush.

Known configurations — the t-shirt sizing approach

Rather than offer a template that provides total flexibility and countless variations, in our experience a common pattern is to provide the ability to select known configurations — in effect, standard t-shirt sizes such as sandbox, small, medium, and large. Other examples of t-shirt sizes are product offerings, such as community edition or enterprise edition. In other cases, it may be workload-specific configurations of a technology – such as map reduce or no sql.

Many enterprise IT organizations, OSS vendors, and SIs make their offerings available today in this way in on-premises, virtualized environments (enterprises) or as software-as-a-service (SaaS) offerings (CSVs and OSVs).

This approach provides good, known configurations of varying sizes that are preconfigured for customers. Without known configurations, end customers must determine cluster sizing on their own, factor in platform resource constraints, and do math to identify the resulting partitioning of storage accounts and other resources (due to cluster size and resource constraints). Known configurations enable customers to easily select the right t-shirt size —that is, a given deployment. In addition to making a better experience for the customer, a small number of known configurations is easier to support and can help you deliver a higher level of density.

A known configuration approach focused on t-shirt sizes may also have varying number of nodes within a size. For example, a small t-shirt size may be between 3 and 10 nodes. The t-shirt size would be designed to accommodate up to 10 nodes and provide the consumer the ability to make free form selections up to the maximum size identified.

A t-shirt size based on workload type, may be more free form in nature in terms of the number of nodes that can be deployed but will have workload distinct node size and configuration of the software on the node.

T-shirt sizes based on product offerings, such as community or Enterprise, may have distinct resource types and maximum number of nodes that can be deployed, typically tied to licensing considerations or feature availability across the different offerings.

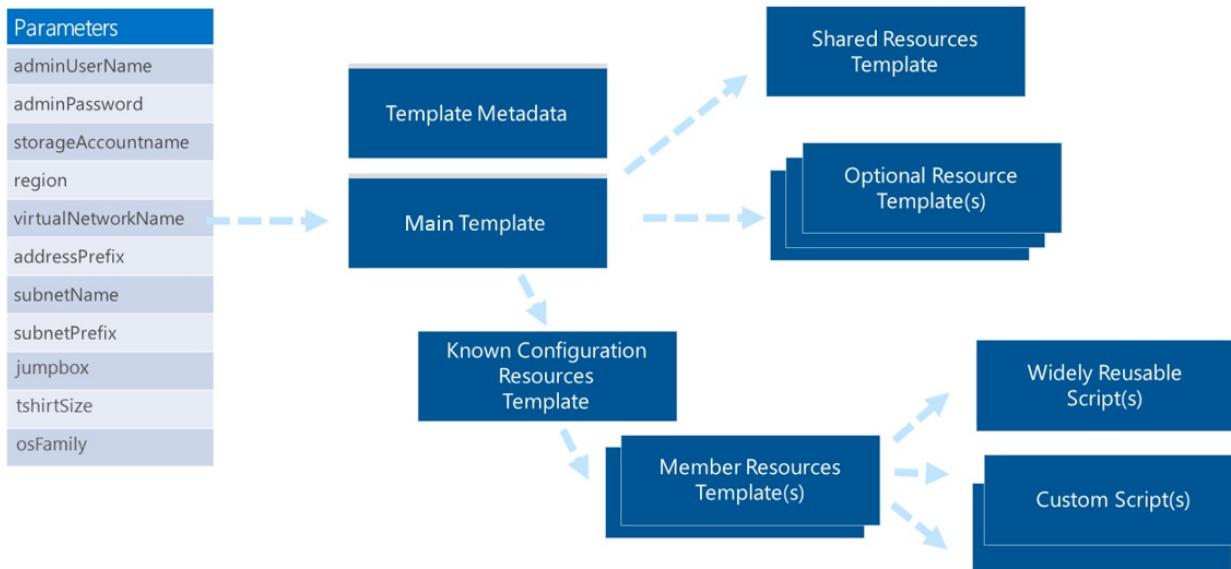
You can also accommodate customers with unique variants using the JSON-based templates. When dealing with outliers, you can incorporate the appropriate planning and considerations for development, support, and costing.

Based on the customer template consumption scenarios, and requirements identified at the start of this document, we identified a pattern for template decomposition.

Capacity and capability-scoped solution templates

Decomposition provides a modular approach to template development that supports reuse, extensibility, testing, and tooling. This section provides detail on how a decomposition approach can be applied to templates with a Capacity or Capability scope.

In this approach, a main template receives parameter values from a template consumer, then links to several types of templates and scripts downstream as shown below. Parameters, static variables, and generated variables are used to provide values in and out of the linked templates.

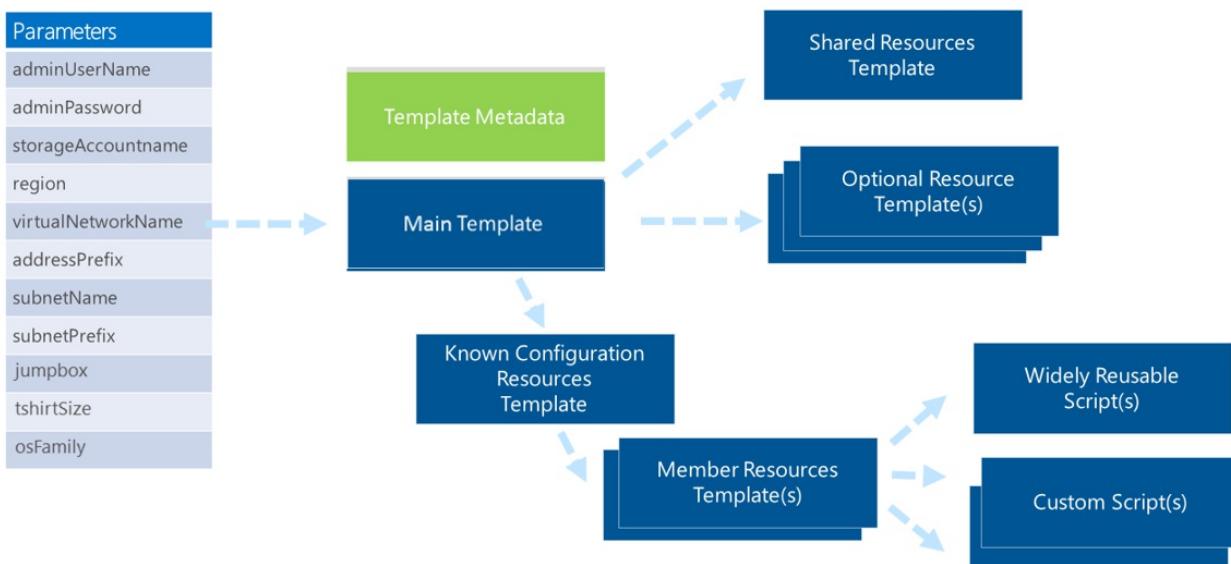


Parameters are passed to a main template then to linked templates

The following sections focus on the types of templates and scripts that a single template is decomposed into. The sections present approaches for passing state information among the templates. Each template and the script types in the image are described along with examples. For a contextual example, see "Putting it together: a sample implementation" later in this document.

Template metadata

Template metadata (the `metadata.json` file) contains key/value pairs that describe a template in JSON, which can be read by humans and software systems.



Template metadata is described in the `metadata.json` file

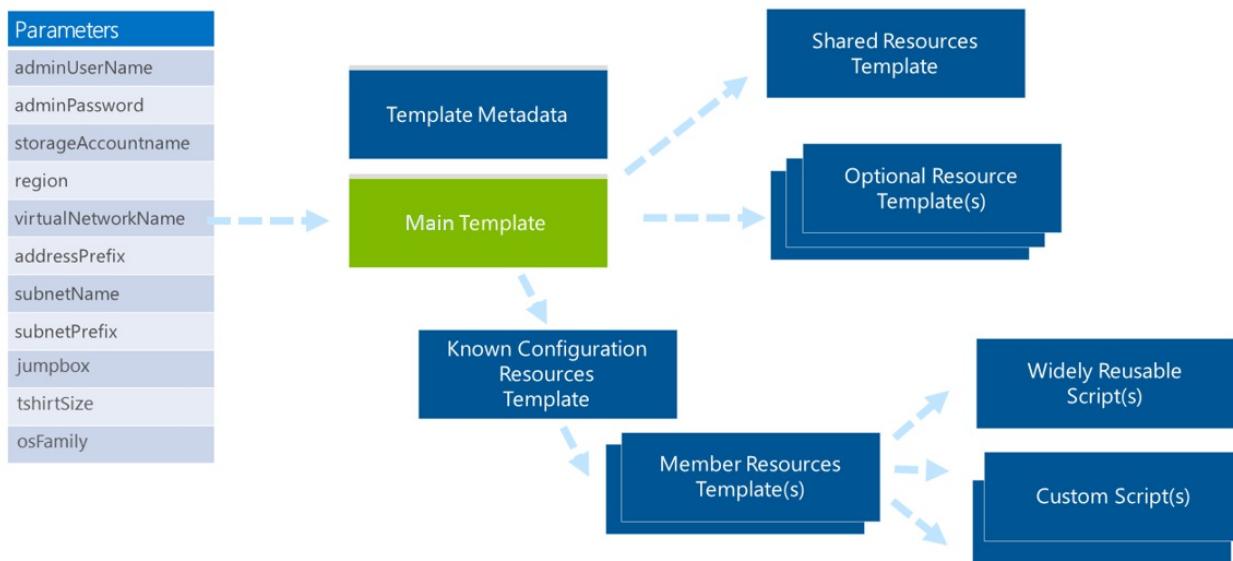
Software agents can retrieve the metadata.json file and publish the information and a link to the template in a web page or directory. Elements include *itemDisplayName*, *description*, *summary*, *githubUsername*, and *dateUpdated*.

An example file is shown below in its entirety.

```
{  
    "itemDisplayName": "PostgreSQL 9.3 on Ubuntu VMs",  
    "description": "This template creates a PostgreSQL streaming-replication between a master and one or more slave servers each with 2 striped data disks. The database servers are deployed into a private-only subnet with one publicly accessible jumpbox VM in a DMZ subnet with public IP.",  
    "summary": "PostgreSQL stream-replication with multiple slave servers and a publicly accessible jumpbox VM",  
    "githubUsername": "arsenvlad",  
    "dateUpdated": "2015-04-24"  
}
```

Main template

The main template receives parameters from a user, uses that information to populate complex object variables, and executes the linked templates.



The main template receives parameters from a user

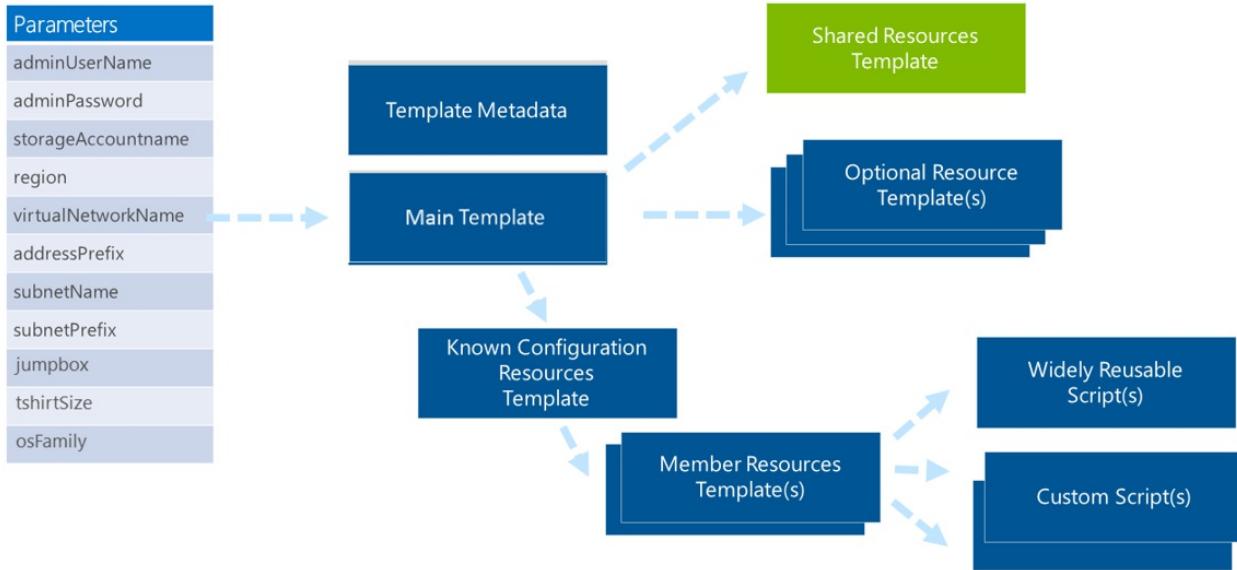
One parameter that is provided is a known configuration type also known as the t-shirt size parameter because of its standardized values such as small, medium, or large. In practice, you can use this parameter in multiple ways. For details, see "Known configuration resources template" later in this document.

Some resources are deployed regardless of the known configuration specified by a user parameter. These resources are provisioned using a single shared resource template and are shared by other templates, so the shared resource template is run first.

Some resources are deployed optionally regardless of the specified known configuration.

Shared resources template

This template delivers resources that are common across all known configurations. It contains the virtual network, availability sets, and other resources that are required regardless of the known configuration template that is deployed.

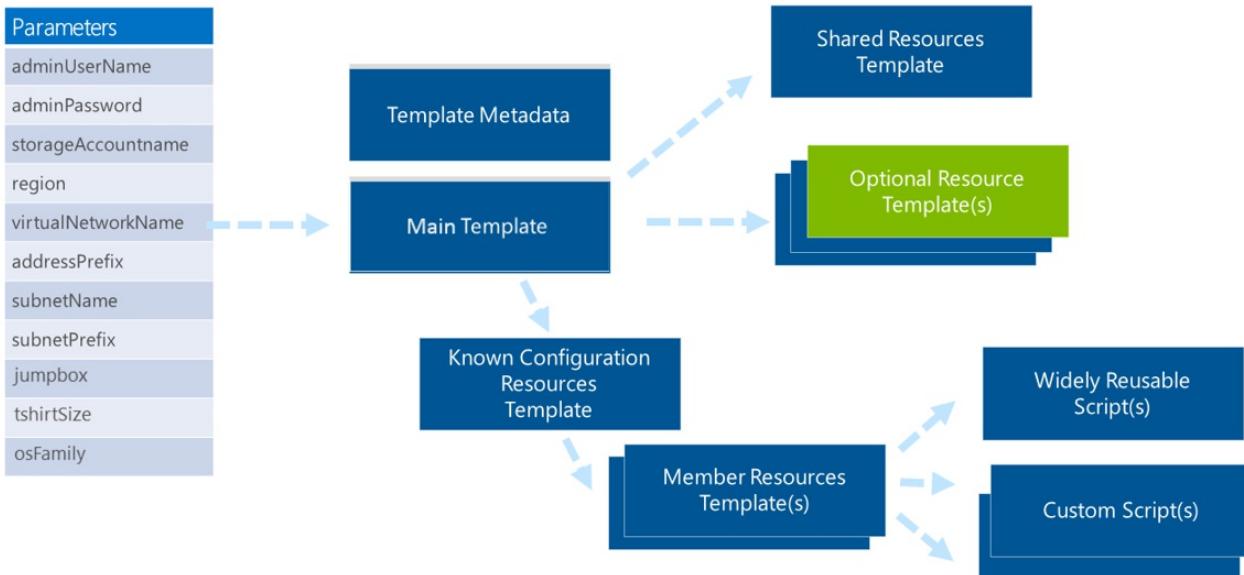


Shared resources template

Resource names, such as the virtual network name, are based on the main template. You can specify them as a variable within that template or receive them as a parameter from the user, as required by your organization.

Optional resources template

The optional resources template contains resources that are programmatically deployed based on the value of a parameter or variable.



Optional resources template

For example, you can use an optional resources template to configure a jumpbox that enables indirect access to a deployed environment from the public Internet. You would use a parameter or variable to identify whether the jumpbox should be enabled and the `concat` function to build the target name for the template, such as `jumpbox_enabled.json`. Template linking would use the resulting variable to install the jumpbox.

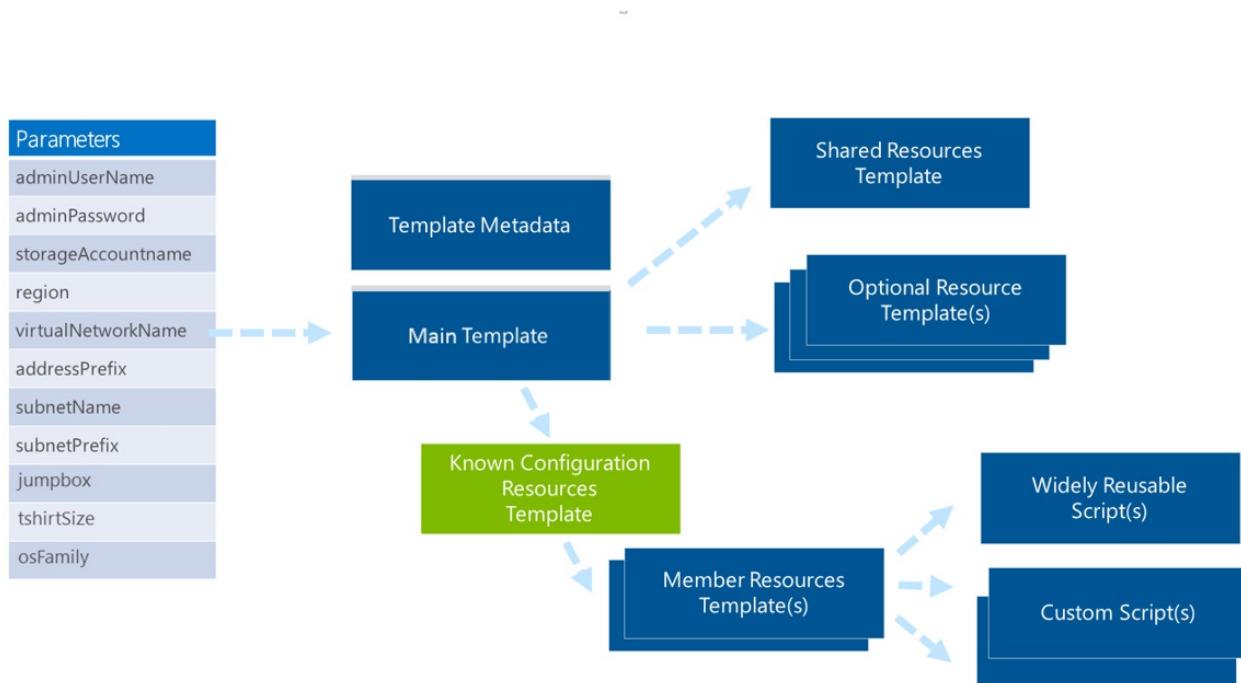
You can link the optional resources template from multiple places:

- When applicable to every deployment, create a parameter-driven link from the shared resources template.
- When applicable to select known configurations—for example, only install on large deployments—create a parameter-driven or variable-driven link from the known configuration template.

Whether a given resource is optional may not be driven by the template consumer but instead by the template provider. For example, you may need to satisfy a particular product requirement or product add-on (common for CSVs) or to enforce policies (common for SIs and enterprise IT groups). In these cases, you can use a variable to identify whether the resource should be deployed.

Known configuration resources template

In the main template, a parameter can be exposed to allow the template consumer to specify a desired known configuration to deploy. Often, this known configuration uses a t-shirt size approach with a set of fixed configuration sizes such as sandbox, small, medium, and large.



Known configuration resources template

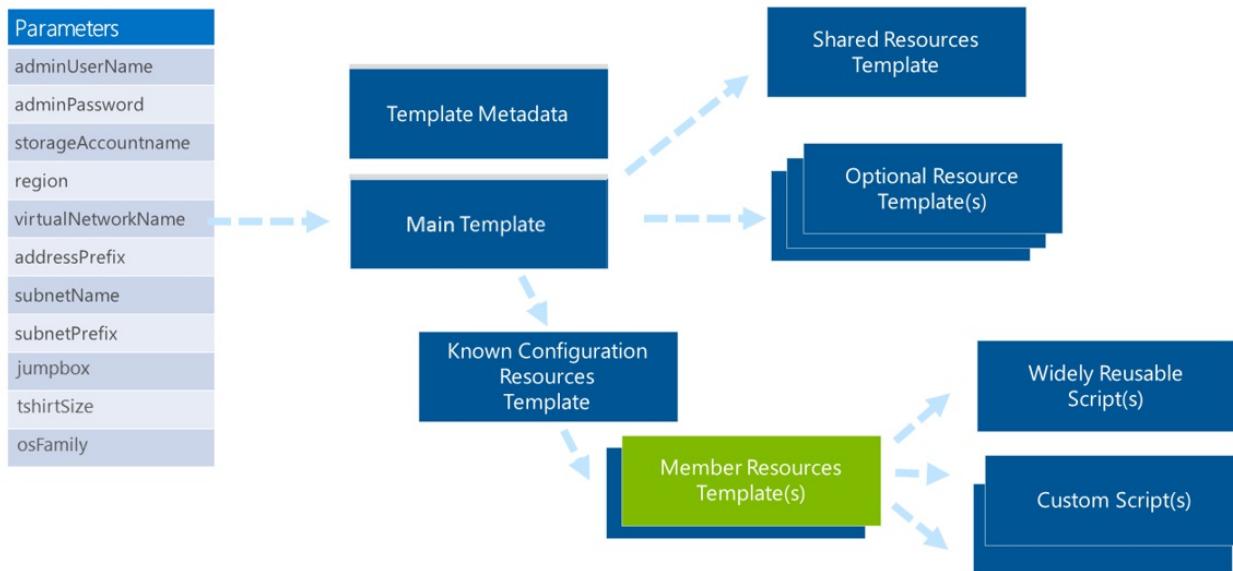
The t-shirt size approach is commonly used, but the parameters can represent any set of known configurations. For example, you can specify a set of environments for an enterprise application such as Development, Test, and Product. Or you could use it for a cloud service to represent different scale units, product versions, or product configurations such as Community, Developer, or Enterprise.

As with the shared resource template, variables are passed to the known configurations template from either:

- An end user—that is, the parameters sent to the main template.
- An organization—that is, the variables in the main template that represent internal requirements or policies.

Member resources template

Within a known configuration, one or more member node types are often included. For example, with Hadoop you have master nodes and data nodes. If you are installing MongoDB, you have data nodes and an arbiter. If you are deploying DataStax, you have data nodes and a VM with OpsCenter installed.



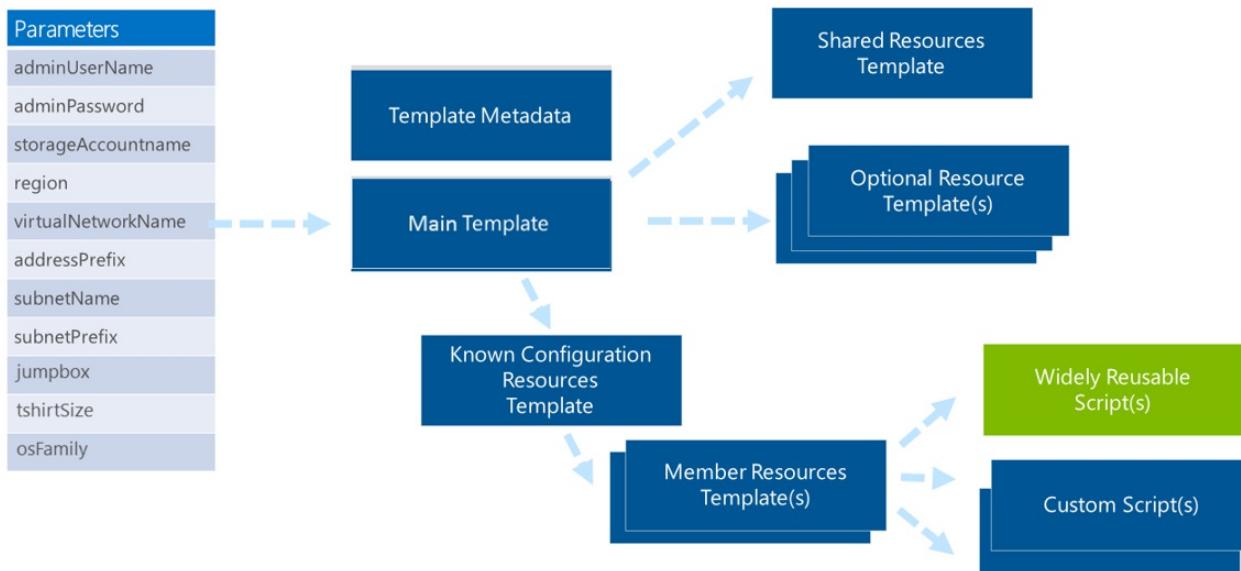
Member resources template

Each type of nodes can have different sizes of VMs, numbers of attached disks, scripts to install and set up the nodes, port configurations for the VMs, number of instances, and other details. So each node type gets its own member resource template, which contains the details for deploying and configuring an infrastructure as well as executing scripts to deploy and configure software within the VM.

For VMs, typically two types of scripts are used, widely reusable and custom scripts.

Widely reusable scripts

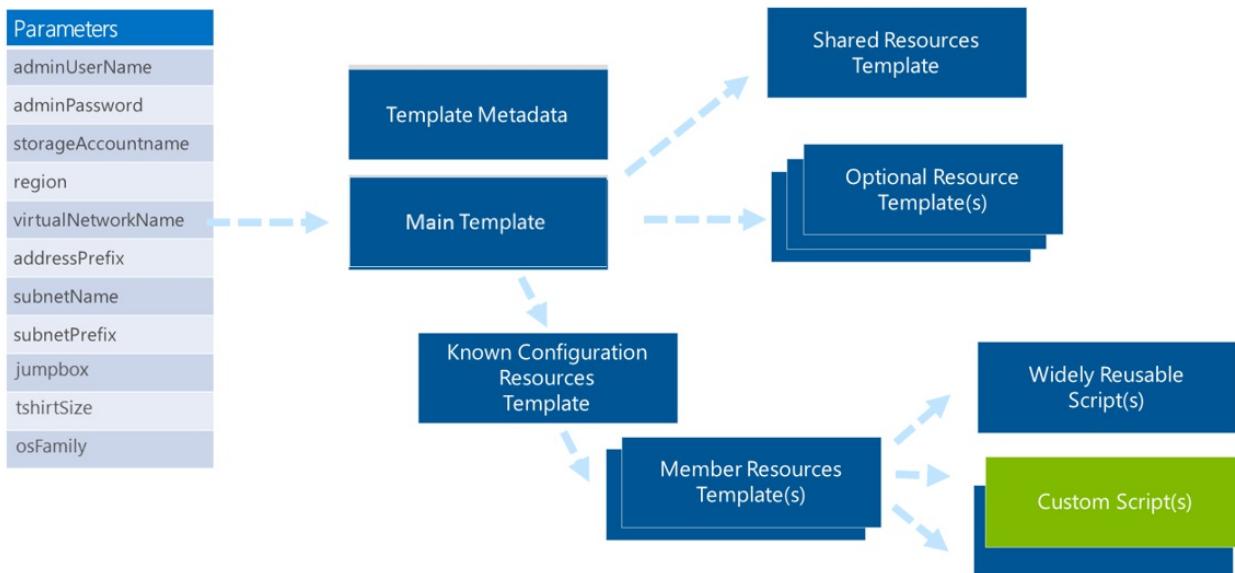
Widely reusable scripts can be used across multiple types of templates. One of the better examples of these widely reusable scripts sets up RAID on Linux to pool disks and gain a greater number of IOPS. Regardless of the software being installed in the VM, this script provides reuse of proven practices for common scenarios.



Member resources templates can call widely reusable scripts

Custom scripts

Templates commonly call one or more scripts that install and configure software within VMs. A common pattern is seen with large topologies where multiple instances of one or more member types are deployed. An installation script is initiated for every VM that can be run in parallel, followed by a setup script that is called after all VMs (or all VMs of a given member type) are deployed.



Member resources templates can call scripts for a specific purpose such as VM configuration

Capability-scoped solution template example - Redis

To show how an implementation might work, let's look at a practical example of building a template that facilitates the deployment and configuration of Redis in standard t-shirt sizes.

For the deployment, there are a set of shared resources (virtual network, storage account, availability sets) and an optional resource (jumpbox). There are multiple known configurations represented as t-shirt sizes (small, medium, large) but each with a single node type. There are also two purpose-specific scripts (installation, configuration).

Creating the template files

You would create a Main Template named `azuredeploy.json`.

You create Shared Resources Template named `shared-resources.json`

You create an Optional Resource Template to enable the deployment of a jumpbox, named `jumpbox_enabled.json`

Redis uses just a single node type, so you create a single Member Resource Template named `node-resources.json`.

With Redis, you want to install each individual node, and then set up the cluster. You have scripts to accommodate the installation and set up, `redis-cluster-install.sh` and `redis-cluster-setup.sh`.

Linking the templates

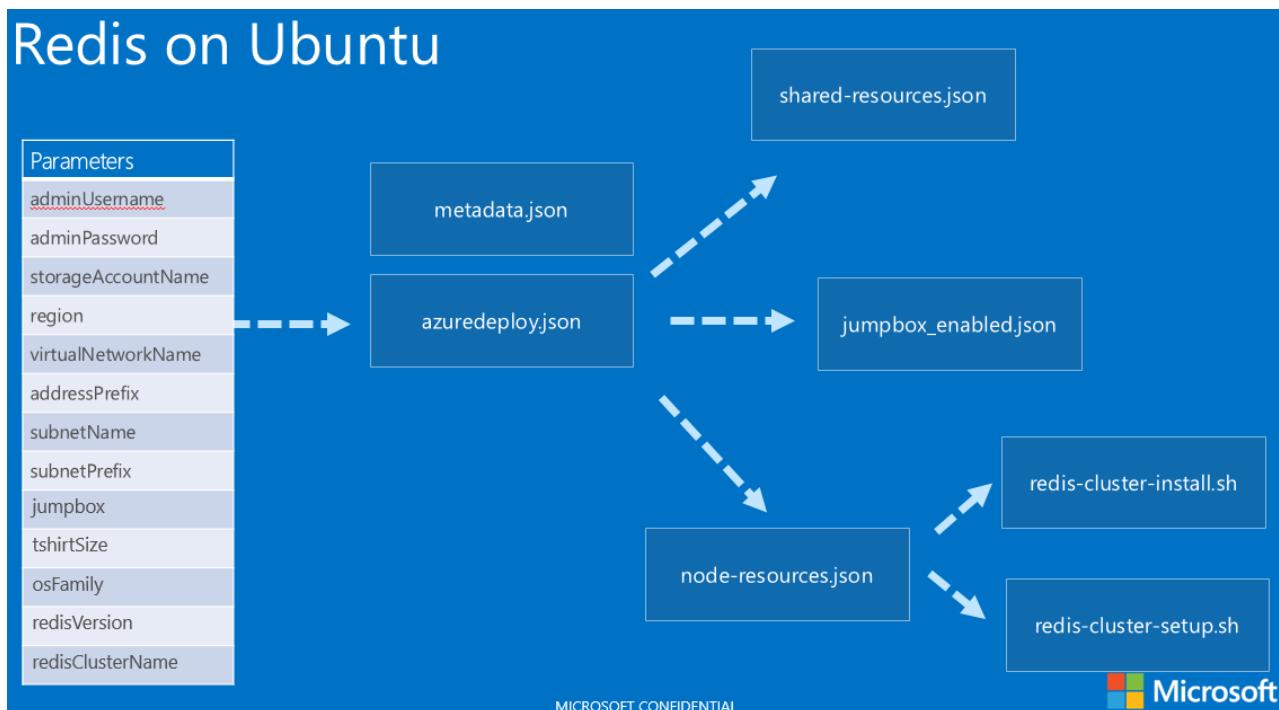
Using template linking, the main template links out to the shared resources template, which establishes the virtual network.

Logic is added within the main template to enable consumers of the template to specify whether a jumpbox should be deployed. An `enabled` value for the `EnableJumpbox` parameter indicates that the customer wants to deploy a jumpbox. When this value is provided, the template concatenates `_enabled` as a suffix to a base template name for the jumpbox capability.

The main template applies the `large` parameter value as a suffix to a base template name for t-shirt sizes, and then

uses that value in a template link out to `technology_on_os_large.json`.

The topology would resemble this illustration.



Template structure for a Redis template

Configuring state

For the nodes in the cluster, there are two steps to configuring the state, both represented by Purpose Specific Scripts. "`redis-cluster-install.sh`" installs Redis and "`redis-cluster-setup.sh`" sets up the cluster.

Supporting Different Size Deployments

Inside variables, the t-shirt size template specifies the number of nodes of each type to deploy for the specified size (*large*). It then deploys that number of VM instances using resource loops, providing unique names to resources by appending a node name with a numeric sequence number from `copyIndex()`. It does these steps for both hot and warm zone VMs, as defined in the t-shirt name template

Decomposition and end-to-end solution scoped templates

A solution template with an end-to-end solution scope is focused on delivering an end-to-end solution. This approach is typically a composition of multiple capability-scoped templates with additional resources, logic, and state.

As highlighted in the image below, the same model used for capability scoped templates is extended for templates with an End-to-End Solution Scope.

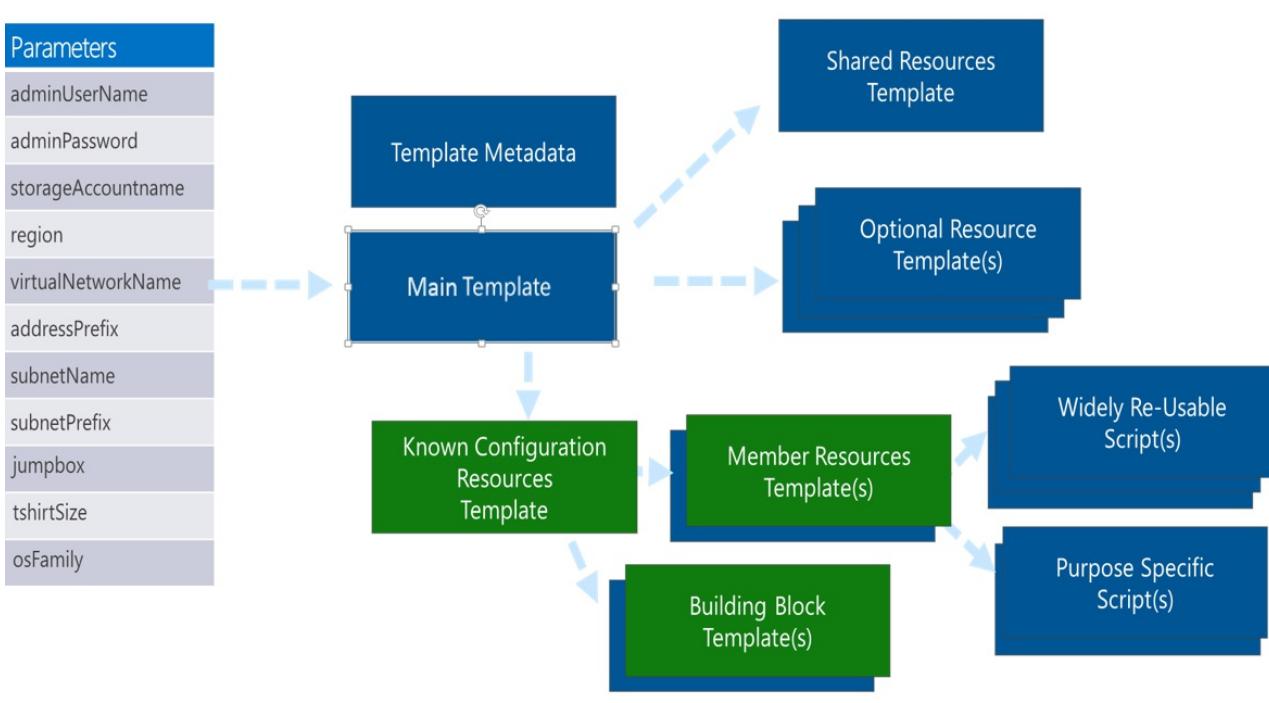
A Shared Resources Template and Optional Resources Templates serve the same function as in the capacity and capability scoped template approaches, but are scoped for the end to end solution.

As end to end solution scoped templates also can typically have t-shirt sizes, the Known Configuration Resources template reflects what is required for a given known configuration of the solution.

The Known Configuration Resources Template links to one or more capability scoped solution templates that are relevant to the end to end solution as well as the Member Resource Templates that are required for the end to end solution.

As the t-shirt size of the solution may be different than the individual capability-scoped template, variables within the Known Configuration Resources Template are used to provide the appropriate values for downstream

capability scoped solution templates to deploy the appropriate t-shirt size.



The model used for capacity or capability scoped solution templates can be readily extended for end to end solution template scopes

Preparing templates for the Marketplace

The preceding approach readily accommodates scenarios where Enterprises, SIs, and CSVs want to either deploy the templates themselves or enable their customers to deploy on their own.

Another desired scenario is deploying a template via the marketplace. This decomposition approach works for the marketplace as well, with some minor changes.

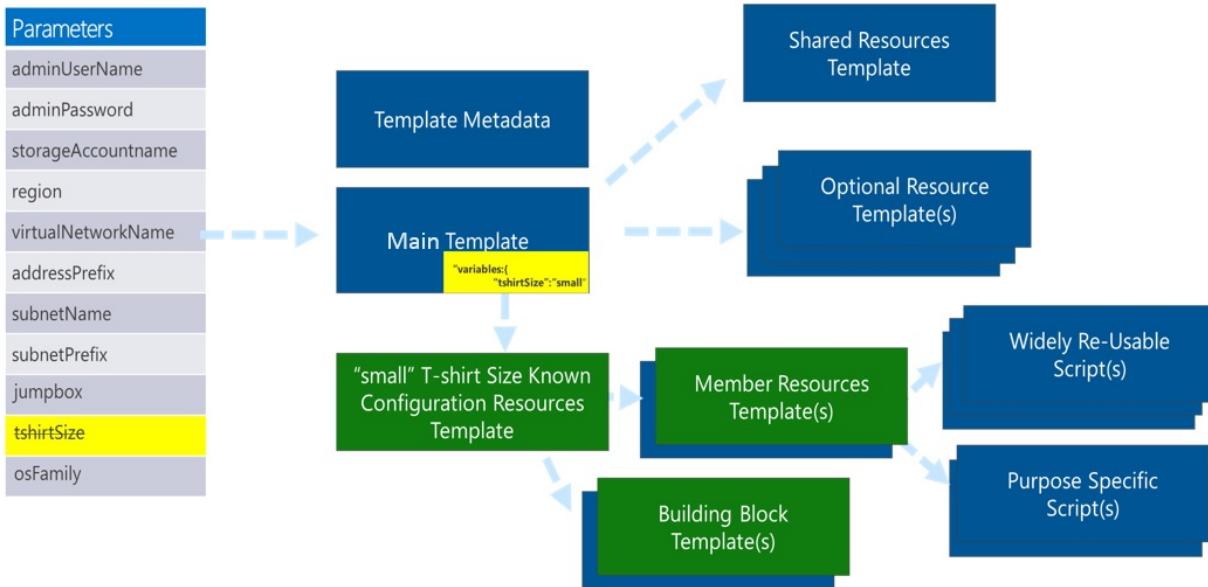
As mentioned previously, templates can be used to offer distinct deployment types for sale in the marketplace. Distinct deployment types may be t-shirt sizes (small, medium, large), product/audience type (community, developer, enterprise), or feature type (basic, high availability).

As shown below, the existing end to end solution or capability scoped templates can be readily utilized to list the different known configurations in the marketplace.

The parameters to the main template are first modified to remove the inbound parameter named `tshirtSize`.

While the distinct deployment types map to the Known Configuration Resources Template, they also need the common resources and configuration found in the Shared Resources Template and potentially those in Optional Resource Templates.

If you want to publish your template to the marketplace, you establish distinct copies of your Main template that replaces the previously available inbound parameter of `tshirtSize` to a variable embedded within the template.



Adapting a solution scoped template for the marketplace

Next steps

- For recommendations about how to handle security in Azure Resource Manager, see [Security considerations for Azure Resource Manager](#)
- To learn about sharing state into and out of templates, see [Sharing state in Azure Resource Manager templates](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Deploy resources with Resource Manager templates and Azure PowerShell

3/30/2017 • 6 min to read • [Edit Online](#)

This topic explains how to use Azure PowerShell with Resource Manager templates to deploy your resources to Azure. Your template can be either a local file or an external file that is available through a URI. When your template resides in a storage account, you can restrict access to the template and provide a shared access signature (SAS) token during deployment.

Deploy

- To quickly get started with deployment, use the following commands to deploy a local template with inline parameters:

```
Login-AzureRmAccount  
Set-AzureRmContext -SubscriptionID {your-subscription-ID}  
New-AzureRmResourceGroup -Name ExampleGroup -Location "South Central US"  
New-AzureRmResourceGroupDeployment -Name ExampleDeployment -ResourceGroupName ExampleResourceGroup -  
TemplateFile c:\MyTemplates\storage.json -storageNamePrefix contoso -storageSKU Standard_GRS
```

The deployment can take a few minutes to complete. When it finishes, you see a message that includes the result:

```
ProvisioningState : Succeeded
```

- The `Set-AzureRmContext` cmdlet is only needed if you want to use a subscription other than your default subscription. To see all your subscriptions and their IDs, use:

```
Get-AzureRmSubscription
```

- To deploy an external template, use the **TemplateUri** parameter:

```
New-AzureRmResourceGroupDeployment -Name ExampleDeployment -ResourceGroupName ExampleResourceGroup -  
TemplateUri https://raw.githubusercontent.com/exampleuser/MyTemplates/master/storage.json -  
storageNamePrefix contoso -storageSKU Standard_GRS
```

- To pass the parameter values in a file, use the **TemplateParameterFile** parameter:

```
New-AzureRmResourceGroupDeployment -Name ExampleDeployment -ResourceGroupName ExampleResourceGroup -  
TemplateFile c:\MyTemplates\storage.json -TemplateParameterFile c:\MyTemplates\storage.parameters.json
```

The parameter file must be in the following format:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageNamePrefix": {
            "value": "contoso"
        },
        "storageSKU": {
            "value": "Standard_GRS"
        }
    }
}
```

Incremental and complete deployments

When deploying your resources, you specify that the deployment is either an incremental update or a complete update. The primary difference between these two modes is how Resource Manager handles existing resources in the resource group that are not in the template:

- In complete mode, Resource Manager **deletes** resources that exist in the resource group but are not specified in the template.
- In incremental mode, Resource Manager **leaves unchanged** resources that exist in the resource group but are not specified in the template.

For both modes, Resource Manager attempts to provision all resources specified in the template. If the resource already exists in the resource group and its settings are unchanged, the operation results in no change. If you change the settings for a resource, the resource is provisioned with those new settings. However, you cannot update the location or type of an existing resource. Instead, deploy a new resource with the location or type that you need.

By default, Resource Manager uses the incremental mode.

To use complete mode, use the **Mode** parameter:

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

Deploy private template with SAS token

You can add your templates to a storage account and link to them during deployment with a SAS token.

IMPORTANT

By following the steps below, the blob containing the template is accessible to only the account owner. However, when you create a SAS token for the blob, the blob is accessible to anyone with that URI. If another user intercepts the URL, that user is able to access the template. Using a SAS token is a good way of limiting access to your templates, but you should not include sensitive data like passwords directly in the template.

Add private template to storage account

The following example sets up a private storage account container and uploads a template:

```
New-AzureRmResourceGroup -Name ManageGroup -Location "South Central US"
New-AzureRmStorageAccount -ResourceGroupName ManageGroup -Name {your-unique-name} -Type Standard_LRS -
Location "West US"
Set-AzureRmCurrentStorageAccount -ResourceGroupName ManageGroup -Name {your-unique-name}
New-AzureStorageContainer -Name templates -Permission Off
Set-AzureStorageBlobContent -Container templates -File c:\MyTemplates\storage.json
```

Provide SAS token during deployment

To deploy a private template in a storage account, generate a SAS token and include it in the URI for the template. Set the expiry time to allow enough time to complete the deployment.

```
Set-AzureRmCurrentStorageAccount -ResourceGroupName ManageGroup -Name {your-unique-name}
$templateuri = New-AzureStorageBlobSASToken -Container templates -Blob storage.json -Permission r -ExpiryTime
(Get-Date).AddHours(2.0) -FullUri
New-AzureRmResourceGroupDeployment -ResourceGroupName ExampleGroup -TemplateUri $templateuri
```

For an example of using a SAS token with linked templates, see [Using linked templates with Azure Resource Manager](#).

Parameters

If your template includes a parameter with the same name as one of the parameters in the PowerShell command, you are prompted to provide a value for that parameter. Azure PowerShell presents the parameter from your template with the postfix **FromTemplate**. For example, a parameter named **ResourceGroupName** in your template conflicts with the **ResourceGroupName** parameter in the [New-AzureRmResourceGroupDeployment](#) cmdlet. You are prompted to provide a value for **ResourceGroupNameFromTemplate**. In general, you should avoid this confusion by not naming parameters with the same name as parameters used for deployment operations.

You can use inline parameters and a local parameter file in the same deployment operation. For example, you can specify some values in the local parameter file and add other values inline during deployment. If you provide values for a parameter in both the local parameter file and inline, the inline value takes precedence.

However, when you use an external parameter file, you cannot pass other values either inline or from a local file. When you specify a parameter file in the **TemplateParameterUri** parameter, all inline parameters are ignored. Provide all parameter values in the external file. If your template includes a sensitive value that you cannot include in the parameter file, either add that value to a key vault, or dynamically provide all parameter values inline.

Debug

If you want to log additional information about the deployment that may help you troubleshoot any deployment errors, use the **DeploymentLogLevel** parameter. You can specify that request content, response content, or both be logged with the deployment operation.

```
New-AzureRmResourceGroupDeployment -Name ExampleDeployment -DeploymentLogLevel All -ResourceGroupName
ExampleGroup -TemplateFile storage.json
```

To get details about a failed deployment operation, use:

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName ExampleDeployment -ResourceGroupName
ExampleGroup).Properties | Where-Object ProvisioningState -eq Failed
```

For tips on resolving common deployment errors, see [Troubleshoot common Azure deployment errors with Azure Resource Manager](#).

Complete deployment script

The following example shows the PowerShell script for deploying a template that is generated by the [export template](#) feature:

```
<#
.SYNOPSIS
    Deploys a template to Azure

.DESCRIPTION
    Deploys an Azure Resource Manager template

.PARAMETER subscriptionId
    The subscription id where the template will be deployed.

.PARAMETER resourceGroupName
    The resource group where the template will be deployed. Can be the name of an existing or a new resource group.

.PARAMETER resourceGroupLocation
    Optional, a resource group location. If specified, will try to create a new resource group in this location. If not specified, assumes resource group is existing.

.PARAMETER deploymentName
    The deployment name.

.PARAMETER templateFilePath
    Optional, path to the template file. Defaults to template.json.

.PARAMETER parametersFilePath
    Optional, path to the parameters file. Defaults to parameters.json. If file is not found, will prompt for parameter values based on template.
#>

param(
    [Parameter(Mandatory=$True)]
    [string]
    $subscriptionId,

    [Parameter(Mandatory=$True)]
    [string]
    $resourceGroupName,

    [string]
    $resourceGroupLocation,

    [Parameter(Mandatory=$True)]
    [string]
    $deploymentName,

    [string]
    $templateFilePath = "template.json",

    [string]
    $parametersFilePath = "parameters.json"
)

<#
.SYNOPSIS
    Registers RPs
#>
Function RegisterRP {
    Param(
```

```

    [string]$ResourceProviderNamespace
)

Write-Host "Registering resource provider '$ResourceProviderNamespace'";
Register-AzureRmResourceProvider -ProviderNamespace $ResourceProviderNamespace;
}

*****
# Script body
# Execution begins here
*****
$ErrorActionPreference = "Stop"

# sign in
Write-Host "Logging in...";
Login-AzureRmAccount;

# select subscription
Write-Host "Selecting subscription '$subscriptionId'";
Select-AzureRmSubscription -SubscriptionID $subscriptionId;

# Register RPs
$resourceProviders = @();
if($resourceProviders.length) {
    Write-Host "Registering resource providers"
    foreach($resourceProvider in $resourceProviders) {
        RegisterRP($resourceProvider);
    }
}

#Create or check for existing resource group
$resourceGroup = Get-AzureRmResourceGroup -Name $resourceGroupName -ErrorAction SilentlyContinue
if(!$resourceGroup)
{
    Write-Host "Resource group '$resourceGroupName' does not exist. To create a new resource group, please enter a location.";
    if(!$resourceGroupLocation) {
        $resourceGroupLocation = Read-Host "resourceGroupLocation";
    }
    Write-Host "Creating resource group '$resourceGroupName' in location '$resourceGroupLocation'";
    New-AzureRmResourceGroup -Name $resourceGroupName -Location $resourceGroupLocation
}
else{
    Write-Host "Using existing resource group '$resourceGroupName'";
}

# Start the deployment
Write-Host "Starting deployment...";
if(Test-Path $parametersFilePath) {
    New-AzureRmResourceGroupDeployment -ResourceGroupName $resourceGroupName -TemplateFile $templateFilePath
    -TemplateParameterFile $parametersFilePath;
} else {
    New-AzureRmResourceGroupDeployment -ResourceGroupName $resourceGroupName -TemplateFile $templateFilePath;
}

```

Next steps

- For an example of deploying resources through the .NET client library, see [Deploy resources using .NET libraries and a template](#).
- To define parameters in template, see [Authoring templates](#).
- For guidance on deploying your solution to different environments, see [Development and test environments in Microsoft Azure](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

- For a four part series about automating deployment, see [Automating application deployments to Azure Virtual Machines](#). This series covers application architecture, access and security, availability and scale, and application deployment.

Deploy resources with Resource Manager templates and Azure CLI

3/31/2017 • 6 min to read • [Edit Online](#)

This topic explains how to use [Azure CLI 2.0](#) with Resource Manager templates to deploy your resources to Azure. Your template can be either a local file or an external file that is available through a URI. When your template resides in a storage account, you can restrict access to the template and provide a shared access signature (SAS) token during deployment.

Deploy

- To quickly get started with deployment, use the following commands to deploy a local template with inline parameters:

```
az login
az account set --subscription {subscription-id}

az group create --name ExampleGroup --location "Central US"
az group deployment create \
    --name ExampleDeployment \
    --resource-group ExampleGroup \
    --template-file storage.json \
    --parameters '{"storageNamePrefix":{"value":"contoso"},"storageSKU":{"value":"Standard_GRS"}}'
```

The deployment can take a few minutes to complete. When it finishes, you see a message that includes the result:

```
"provisioningState": "Succeeded",
```

- The `az account set` command is only needed if you want to use a subscription other than your default subscription. To see all your subscriptions and their IDs, use:

```
az account list
```

- To deploy an external template, use the **template-uri** parameter:

```
az group deployment create \
    --name ExampleDeployment \
    --resource-group ExampleGroup \
    --template-uri "https://raw.githubusercontent.com/exampleuser/MyTemplates/master/storage.json" \
    --parameters '{"storageNamePrefix":{"value":"contoso"},"storageSKU":{"value":"Standard_GRS"}}'
```

- To pass the parameter values in a file, use:

```
az group deployment create \
    --name ExampleDeployment \
    --resource-group ExampleGroup \
    --template-file storage.json \
    --parameters @storage.parameters.json
```

The parameter file must be in the following format:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "storageNamePrefix": {  
            "value": "contoso"  
        },  
        "storageSKU": {  
            "value": "Standard_GRS"  
        }  
    }  
}
```

Incremental and complete deployments

When deploying your resources, you specify that the deployment is either an incremental update or a complete update. The primary difference between these two modes is how Resource Manager handles existing resources in the resource group that are not in the template:

- In complete mode, Resource Manager **deletes** resources that exist in the resource group but are not specified in the template.
- In incremental mode, Resource Manager **leaves unchanged** resources that exist in the resource group but are not specified in the template.

For both modes, Resource Manager attempts to provision all resources specified in the template. If the resource already exists in the resource group and its settings are unchanged, the operation results in no change. If you change the settings for a resource, the resource is provisioned with those new settings. However, you cannot update the location or type of an existing resource. Instead, deploy a new resource with the location or type that you need.

By default, Resource Manager uses the incremental mode.

To use complete mode, use the mode parameter:

```
az group deployment create \  
    --name ExampleDeployment \  
    --mode Complete \  
    --resource-group ExampleGroup \  
    --template-file storage.json \  
    --parameters '{"storageNamePrefix":{"value":"contoso"}, "storageSKU":{"value":"Standard_GRS"}}'
```

Deploy template from storage with SAS token

You can add your templates to a storage account and link to them during deployment with a SAS token.

IMPORTANT

By following the steps below, the blob containing the template is accessible to only the account owner. However, when you create a SAS token for the blob, the blob is accessible to anyone with that URL. If another user intercepts the URL, that user is able to access the template. Using a SAS token is a good way of limiting access to your templates, but you should not include sensitive data like passwords directly in the template.

Add private template to storage account

The following example sets up a private storage account container and uploads a template:

```

az group create --name "ManageGroup" --location "South Central US"
az storage account create \
    --resource-group ManageGroup \
    --location "South Central US" \
    --sku Standard_LRS \
    --kind Storage \
    --name {your-unique-name}
connection=$(az storage account show-connection-string \
    --resource-group ManageGroup \
    --name {your-unique-name} \
    --query connectionString)
az storage container create \
    --name templates \
    --public-access Off \
    --connection-string $connection
az storage blob upload \
    --container-name templates \
    --file vmlinu.json \
    --name vmlinu.json \
    --connection-string $connection

```

Provide SAS token during deployment

To deploy a private template in a storage account, generate a SAS token and include it in the URI for the template. Set the expiry time to allow enough time to complete the deployment.

```

seconds='@'$(($date +%s) + 1800 ))
expiretime=$(date +%Y-%m-%dT%H:%MZ --date=$seconds)
connection=$(az storage account show-connection-string \
    --resource-group ManageGroup \
    --name {your-unique-name} \
    --query connectionString)
token=$(az storage blob generate-sas \
    --container-name templates \
    --name vmlinu.json \
    --expiry $expiretime \
    --permissions r \
    --output tsv \
    --connection-string $connection)
url=$(az storage blob url \
    --container-name templates \
    --name vmlinu.json \
    --output tsv \
    --connection-string $connection)
az group deployment create --resource-group ExampleGroup --template-uri $url?$token

```

For an example of using a SAS token with linked templates, see [Using linked templates with Azure Resource Manager](#).

Debug

To see information about the operations for a failed deployment, use:

```

az group deployment operation list --resource-group ExampleGroup --name vmlinu --query "[*].[properties.statusMessage]"

```

For tips on resolving common deployment errors, see [Troubleshoot common Azure deployment errors with Azure Resource Manager](#).

Complete deployment script

The following example shows the Azure CLI 2.0 script for deploying a template that is generated by the [export template](#) feature:

```
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'

# -e: immediately exit if any command has a non-zero exit status
# -o: prevents errors in a pipeline from being masked
# IFS new value is less likely to cause confusing bugs when looping arrays or arguments (e.g. $@)

usage() { echo "Usage: $0 -i <subscriptionId> -g <resourceGroupName> -n <deploymentName> -l <resourceGroupLocation>" 1>&2; exit 1; }

declare subscriptionId=""
declare resourceGroupName=""
declare deploymentName=""
declare resourceGroupLocation=""

# Initialize parameters specified from command line
while getopts ":i:g:n:l:" arg; do
    case "${arg}" in
        i)
            subscriptionId=${OPTARG}
            ;;
        g)
            resourceGroupName=${OPTARG}
            ;;
        n)
            deploymentName=${OPTARG}
            ;;
        l)
            resourceGroupLocation=${OPTARG}
            ;;
    esac
done
shift $((OPTIND-1))

#Prompt for parameters is some required parameters are missing
if [[ -z "$subscriptionId" ]]; then
    echo "Subscription Id:"
    read subscriptionId
    [[ "${subscriptionId:?}" ]]
fi

if [[ -z "$resourceGroupName" ]]; then
    echo "ResourceGroupName:"
    read resourceGroupName
    [[ "${resourceGroupName:?}" ]]
fi

if [[ -z "$deploymentName" ]]; then
    echo "DeploymentName:"
    read deploymentName
fi

if [[ -z "$resourceGroupLocation" ]]; then
    echo "Enter a location below to create a new resource group else skip this"
    echo "ResourceGroupLocation:"
    read resourceGroupLocation
fi

#templateFile Path - template file to be used
templateFilePath="template.json"

if [ ! -f "$templateFilePath" ]; then
    echo "$templateFilePath not found"
....
```

```

    exit 1
fi

#parameter file path
parametersFilePath="parameters.json"

if [ ! -f "$parametersFilePath" ]; then
    echo "$parametersFilePath not found"
    exit 1
fi

if [ -z "$subscriptionId" ] || [ -z "$resourceGroupName" ] || [ -z "$deploymentName" ]; then
    echo "Either one of subscriptionId, resourceName, deploymentName is empty"
    usage
fi

#login to azure using your credentials
az account show 1> /dev/null

if [ $? != 0 ];
then
    az login
fi

#set the default subscription id
az account set --name $subscriptionId

set +e

#Check for existing RG
az group show $resourceGroupName 1> /dev/null

if [ $? != 0 ]; then
    echo "Resource group with name" $resourceGroupName "could not be found. Creating new resource group.."
    set -e
    (
        set -x
        az resource group create --name $resourceGroupName --location $resourceGroupLocation 1> /dev/null
    )
    else
        echo "Using existing resource group..."
fi

#Start deployment
echo "Starting deployment..."
(
    set -x
    az resource group deployment create --name $deploymentName --resource-group $resourceGroupName --template-file $templateFilePath --parameters $parametersFilePath
)

if [ $? == 0 ];
then
    echo "Template has been successfully deployed"
fi

```

Next steps

- For an example of deploying resources through the .NET client library, see [Deploy resources using .NET libraries and a template](#).
- To define parameters in template, see [Authoring templates](#).
- For guidance on deploying your solution to different environments, see [Development and test environments in Microsoft Azure](#).
- For details about using a KeyVault reference to pass secure values, see [Pass secure values during deployment](#).

- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).
- For a four part series about automating deployment, see [Automating application deployments to Azure Virtual Machines](#). This series covers application architecture, access and security, availability and scale, and application deployment.

Deploy resources with Resource Manager templates and Azure portal

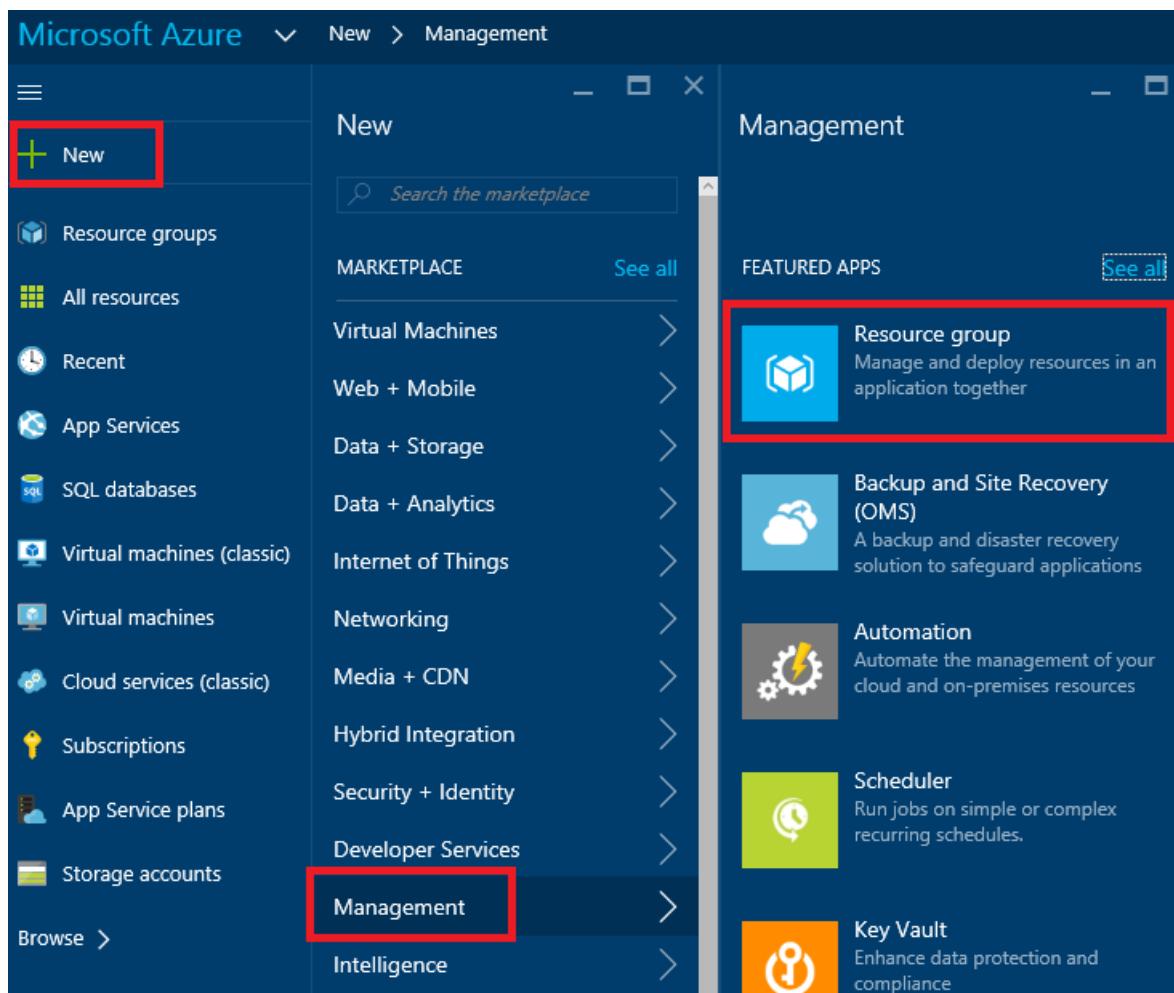
3/30/2017 • 3 min to read • [Edit Online](#)

This topic shows how to use the [Azure portal](#) with [Azure Resource Manager](#) to deploy your Azure resources. To learn about managing your resources, see [Manage Azure resources through portal](#).

Currently, not every service supports the portal or Resource Manager. For those services, you need to use the [classic portal](#). For the status of each service, see [Azure portal availability chart](#).

Create resource group

1. To create an empty resource group, select **New > Management > Resource Group**.



2. Give it a name and location, and, if necessary, select a subscription. You need to provide a location for the resource group because the resource group stores metadata about the resources. For compliance reasons, you may want to specify where that metadata is stored. In general, we recommend that you specify a location where most of your resources will reside. Using the same location can simplify your template.

* Resource group name
ExampleGroup

* Subscription
Windows Azure MSDN - Visual Studio Ultir

* Resource group location
West US

Deploy resources from Marketplace

After you create a resource group, you can deploy resources to it from the Marketplace. The Marketplace provides pre-defined solutions for common scenarios.

1. To start a deployment, select **New** and the type of resource you would like to deploy. Then, look for the particular version of the resource you would like to deploy.

Microsoft Azure New > Virtual Machines

New

Virtual Machines

Marketplace

- Virtual Machines
- Web + Mobile
- Data + Storage
- Data + Analytics
- Internet of Things

Ubuntu Server 14.04 LTS

Ubuntu Server delivers the best value scale-out performance available.

2. If you do not see the particular solution you would like to deploy, you can search the Marketplace for it.

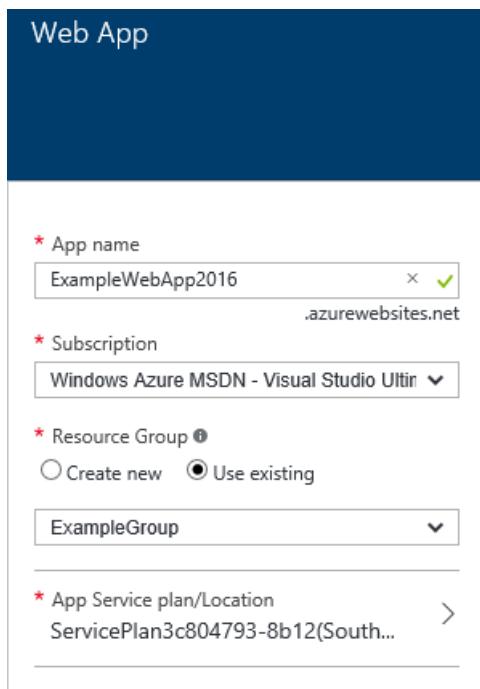
New

Search: wordp

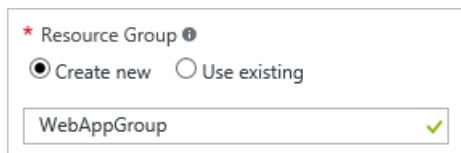
- WordPress
- Scalable WordPress
- WordPress Japanese Package
- wordpress + mysql
- WordPress using MySQL Replication Cluster

3. Depending on the type of selected resource, you have a collection of relevant properties to set before deployment. Those options are not shown here, as they vary based on resource type. For all types, you must

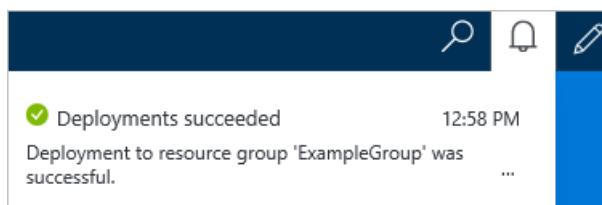
select a destination resource group. The following image shows how to create a web app and deploy it to the resource group you created.



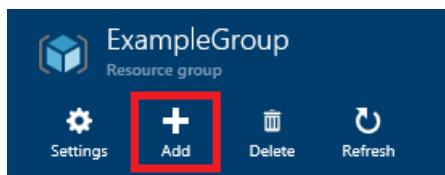
Alternatively, you can decide to create a resource group when deploying your resources. Select **Create new** and give the resource group a name.



4. Your deployment begins. The deployment could take a few minutes. When the deployment has finished, you see a notification.



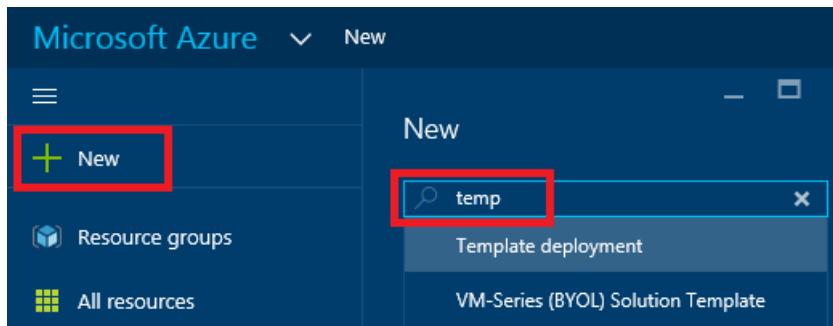
5. After deploying your resources, you can add more resources to the resource group by using the **Add** command on the resource group blade.



Deploy resources from custom template

If you want to execute a deployment but not use any of the templates in the Marketplace, you can create a customized template that defines the infrastructure for your solution. To learn about creating templates, see [Authoring Azure Resource Manager templates](#).

1. To deploy a customized template through the portal, select **New**, and start searching for **Template Deployment** until you can select it from the options.



2. Select **Template Deployment** from the available resources.

This screenshot shows the Azure search results for 'Template deployment'. The search bar at the top contains 'Template deployment'. Below it, the results section is titled 'Results'. It shows a single item: 'Template deployment' by Microsoft. The item is highlighted with a red box.

3. After launching the template deployment, open the blank template that is available for customizing.

This screenshot shows the 'Edit template' blade. On the left, under 'Custom deployment', there's a list with 'Template deployment' highlighted with a red box. On the right, the 'Edit template' section shows the JSON template code. The code starts with a schema reference and basic parameters.

In the editor, add the JSON syntax that defines the resources you want to deploy. Select **Save** when done.
For guidance on writing the JSON syntax, see [Resource Manager template walkthrough](#).

Edit template

Edit your Azure Resource Manager template

Quickstart Download

Parameters (0)

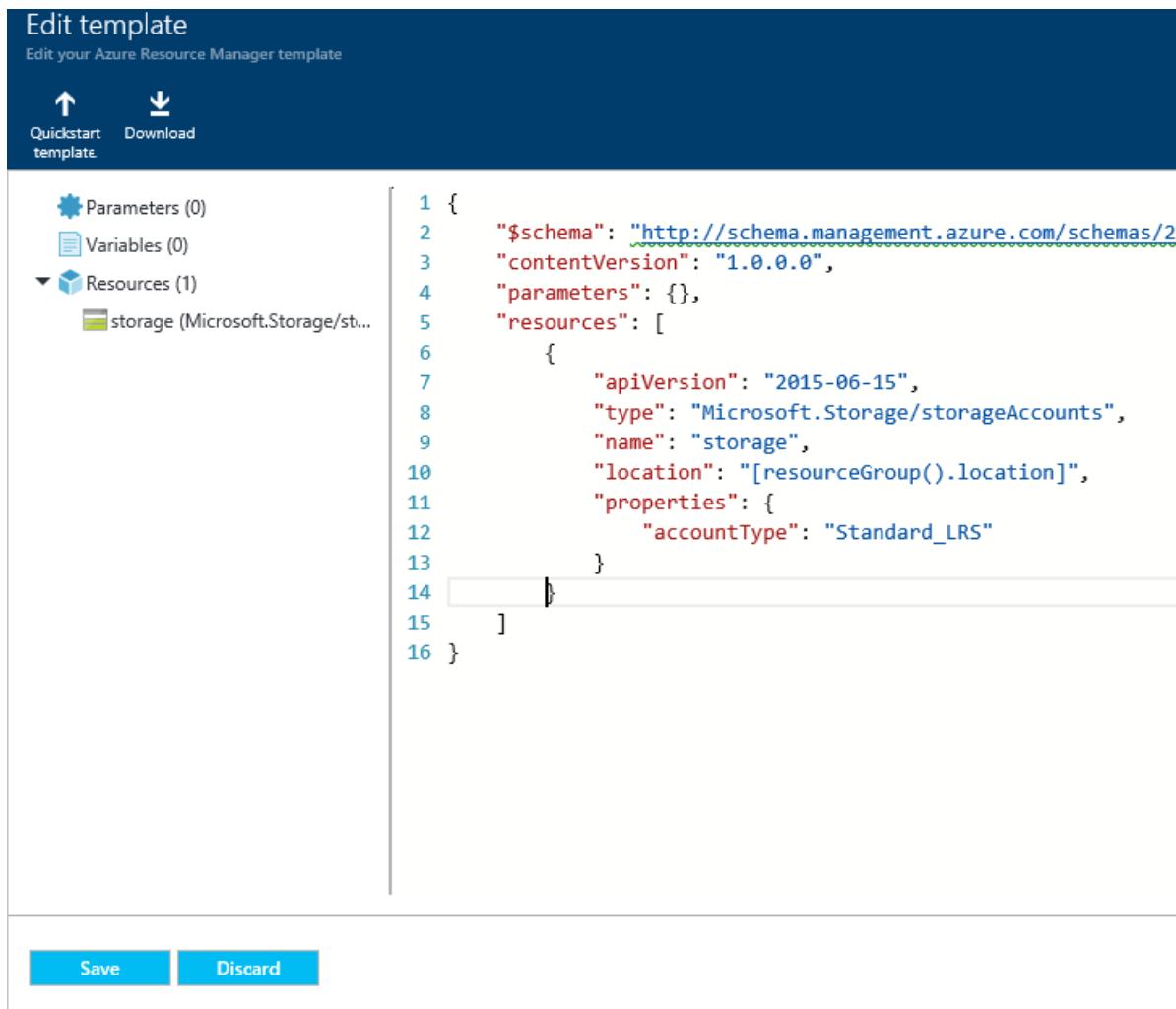
Variables (0)

Resources (1)

storage (Microsoft.Storage/st...)

```
1 {
2   "$schema": "http://schema.management.azure.com/schemas/2015-01-01-deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {},
5   "resources": [
6     {
7       "apiVersion": "2015-06-15",
8       "type": "Microsoft.Storage/storageAccounts",
9       "name": "storage",
10      "location": "[resourceGroup().location]",
11      "properties": {
12        "accountType": "Standard_LRS"
13      }
14    }
15  ]
16 }
```

Save Discard



4. Or, you can select a pre-existing template from the [Azure quickstart templates](#). These templates are contributed by the community. They cover many common scenarios, and someone may have added a template that is similar to what you are trying to deploy. You can search the templates to find something that matches your scenario.

Edit template

Edit your Azure Resource Manager template

Quickstart template Download

Load a quickstart template

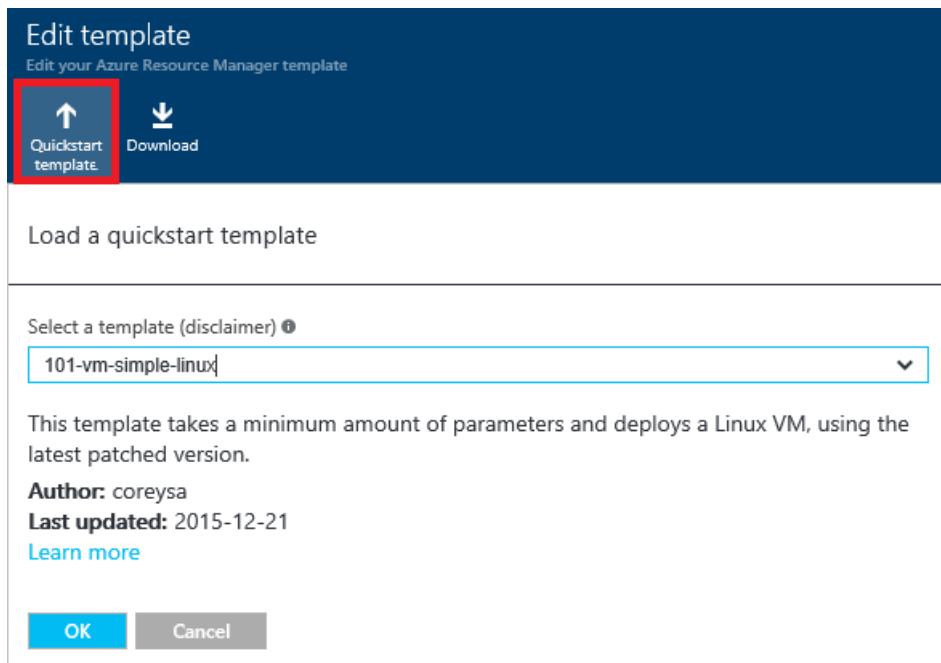
Select a template (disclaimer) ⓘ

101-vm-simple-linux

This template takes a minimum amount of parameters and deploys a Linux VM, using the latest patched version.

Author: coreysa
Last updated: 2015-12-21
[Learn more](#)

OK Cancel



You can view the selected template in the editor.

5. After providing all the other values, select **Create** to deploy the template.

Custom deployment

Deploy from a custom template

* Template >
Edit template

* Parameters >
Edit parameters

* Subscription
Windows Azure MSDN - Visual Studio Ultir ▾

* Resource group ⓘ
○ Create new ○ Use existing
ExampleGroup ▾

* Resource group location
West US ▾

* Legal terms >
Legal terms accepted

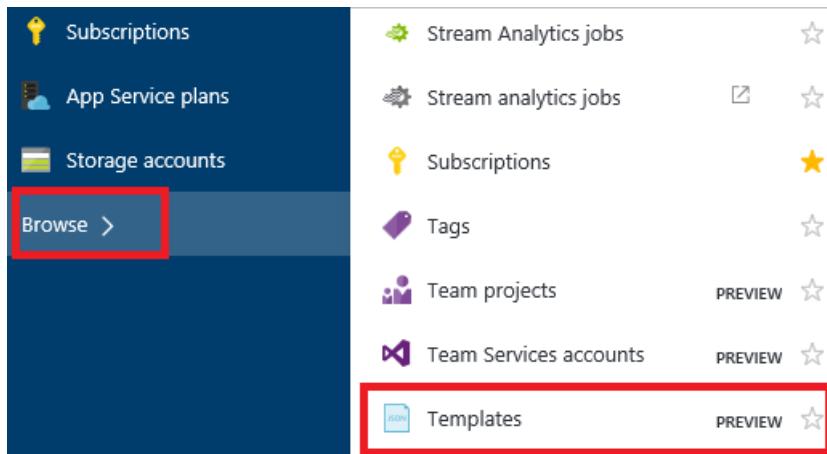
Pin to dashboard

Create

Deploy resources from a template saved to your account

The portal enables you to save a template to your Azure account, and redeploy it later. For more information about working with these saved templates, [Get started with private templates on the Azure portal](#).

1. To find your saved templates, select **Browse > Templates**.



2. From the list of templates saved to your account, select the one you wish to work on.

Templates

Tom FitzMacken - PREVIEW

Add Columns Refresh

Directory: Tom FitzMacken – [Switch directories](#)

Filter items...

NAME	DESCRIPTION	MODIFIED	SHARED WITH
myvmtemplate	An example template	4/8/2016	Only me
...			

3. Select **Deploy** to redeploy this saved template.



Next Steps

- To view audit logs, see [Audit operations with Resource Manager](#).
- To troubleshoot deployment errors, see [View deployment operations](#).
- To retrieve a template from a deployment or resource group, see [Export Azure Resource Manager template from existing resources](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).
- For a four part series about automating deployment, see [Automating application deployments to Azure Virtual Machines](#). This series covers application architecture, access and security, availability and scale, and application deployment.

Deploy resources with Resource Manager templates and Resource Manager REST API

3/30/2017 • 3 min to read • [Edit Online](#)

This article explains how to use the Resource Manager REST API with Resource Manager templates to deploy your resources to Azure.

TIP

For help with debugging an error during deployment, see:

- [View deployment operations](#) to learn about getting information that helps you troubleshoot your error
- [Troubleshoot common errors when deploying resources to Azure with Azure Resource Manager](#) to learn how to resolve common deployment errors

Your template can be either a local file or an external file that is available through a URI. When your template resides in a storage account, you can restrict access to the template and provide a shared access signature (SAS) token during deployment.

Incremental and complete deployments

When deploying your resources, you specify that the deployment is either an incremental update or a complete update. The primary difference between these two modes is how Resource Manager handles existing resources in the resource group that are not in the template:

- In complete mode, Resource Manager **deletes** resources that exist in the resource group but are not specified in the template.
- In incremental mode, Resource Manager **leaves unchanged** resources that exist in the resource group but are not specified in the template.

For both modes, Resource Manager attempts to provision all resources specified in the template. If the resource already exists in the resource group and its settings are unchanged, the operation results in no change. If you change the settings for a resource, the resource is provisioned with those new settings. However, you cannot update the location or type of an existing resource. Instead, deploy a new resource with the location or type that you need.

By default, Resource Manager uses the incremental mode.

Deploy with the REST API

1. Set [common parameters and headers](#), including authentication tokens.
2. If you do not have an existing resource group, create a resource group. Provide your subscription ID, the name of the new resource group, and location that you need for your solution. For more information, see [Create a resource group](#).

```

PUT
https://management.azure.com/subscriptions/<YourSubscriptionId>/resourcegroups/<YourResourceGroupName>?
api-version=2015-01-01
<common headers>
{
    "location": "West US",
    "tags": {
        "tagname1": "tagvalue1"
    }
}

```

3. Validate your deployment before executing it by running the [Validate a template deployment](#) operation. When testing the deployment, provide parameters exactly as you would when executing the deployment (shown in the next step).
4. Create a deployment. Provide your subscription ID, the name of the resource group, the name of the deployment, and a link to your template. For information about the template file, see [Parameter file](#). For more information about the REST API to create a resource group, see [Create a template deployment](#). Notice the **mode** is set to **Incremental**. To run a complete deployment, set **mode** to **Complete**. Be careful when using the complete mode as you can inadvertently delete resources that are not in your template.

```

PUT
https://management.azure.com/subscriptions/<YourSubscriptionId>/resourcegroups/<YourResourceGroupName>/
providers/Microsoft.Resources/deployments/<YourDeploymentName>?api-version=2015-01-01
<common headers>
{
    "properties": {
        "templateLink": {
            "uri": "http://mystorageaccount.blob.core.windows.net/templates/template.json",
            "contentVersion": "1.0.0.0"
        },
        "mode": "Incremental",
        "parametersLink": {
            "uri": "http://mystorageaccount.blob.core.windows.net/templates/parameters.json",
            "contentVersion": "1.0.0.0"
        }
    }
}

```

If you want to log response content, request content, or both, include **debugSetting** in the request.

```

"debugSetting": {
    "detailLevel": "requestContent, responseContent"
}

```

You can set up your storage account to use a shared access signature (SAS) token. For more information, see [Delegating Access with a Shared Access Signature](#).

5. Get the status of the template deployment. For more information, see [Get information about a template deployment](#).

```

GET
https://management.azure.com/subscriptions/<YourSubscriptionId>/resourcegroups/<YourResourceGroupName>/
providers/Microsoft.Resources/deployments/<YourDeploymentName>?api-version=2015-01-01
<common headers>

```

Parameter file

If you use a parameter file to pass parameter values during deployment, you need to create a JSON file with a format similar to the following example:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "webSiteName": {  
            "value": "ExampleSite"  
        },  
        "webSiteHostingPlanName": {  
            "value": "DefaultPlan"  
        },  
        "webSiteLocation": {  
            "value": "West US"  
        },  
        "adminPassword": {  
            "reference": {  
                "keyVault": {  
                    "id": "/subscriptions/{guid}/resourceGroups/{group-name}/providers/Microsoft.KeyVault/vaults/{vault-name}"  
                },  
                "secretName": "sqlAdminPassword"  
            }  
        }  
    }  
}
```

The size of the parameter file cannot be more than 64 KB.

If you need to provide a sensitive value for a parameter (such as a password), add that value to a key vault. Retrieve the key vault during deployment as shown in the previous example. For more information, see [Pass secure values during deployment](#).

Next steps

- To learn about handling asynchronous REST operations, see [Track asynchronous Azure operations](#).
- For an example of deploying resources through the .NET client library, see [Deploy resources using .NET libraries and a template](#).
- To define parameters in template, see [Authoring templates](#).
- For guidance on deploying your solution to different environments, see [Development and test environments in Microsoft Azure](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).
- For a four part series about automating deployment, see [Automating application deployments to Azure Virtual Machines](#). This series covers application architecture, access and security, availability and scale, and application deployment.

Continuous integration in Visual Studio Team Services using Azure Resource Group deployment projects

1/17/2017 • 8 min to read • [Edit Online](#)

To deploy an Azure template, you need to perform tasks to go through the various stages: Build, Test, Copy to Azure (also called "Staging"), and Deploy Template. There are two different ways to deploy templates Visual Studio Team Services (VS Team Services). Both methods provide the same results, so choose the one that best fits your workflow.

1. Add a single step to your build definition that runs the PowerShell script that's included in the Azure Resource Group deployment project (Deploy-AzureResourceGroup.ps1). The script copies artifacts and then deploys the template.
2. Add multiple VS Team Services build steps, each one performing a stage task.

This article demonstrates both options. The first option has the advantage of using the same script used by developers in Visual Studio providing consistency throughout the lifecycle. The second option offers a convenient alternative to the built in script. Both procedures assume you already have a Visual Studio deployment project checked into VS Team Services.

Copy artifacts to Azure

Regardless of the scenario, if you have any artifacts that are needed for template deployment, you need to give Azure Resource Manager access to them. These artifacts can include files such as:

- Nested templates
- Configuration scripts and DSC scripts
- Application binaries

Nested Templates and Configuration Scripts

When you use the templates provided by Visual Studio (or built with Visual Studio snippets), the PowerShell script not only stages the artifacts, it also parameterizes the URI for the resources for different deployments. The script then copies the artifacts to a secure container in Azure, creates a SaaS token for that container, and then passes that information on to the template deployment. See [Create a template deployment](#) to learn more about nested templates. When using tasks in VS Team Services, you need to select the appropriate tasks for your template deployment and if necessary pass parameter values from the staging step to the template deployment.

Set up continuous deployment in VS Team Services

To call the PowerShell script in VS Team Services, you need to update your build definition. In brief, the steps are:

1. Edit the build definition.
2. Set up Azure authorization in VS Team Services.
3. Add an Azure PowerShell build step that references the PowerShell script in the Azure Resource Group deployment project.
4. Set the value of the `-ArtifactsStagingDirectory` parameter to work with a project built in VS Team Services.

Detailed walkthrough for Option 1

The following steps walk you through the steps necessary to configure continuous deployment in VS Team Services using a single task that runs the PowerShell script in your project.

1. Edit your VS Team Services build definition and add an Azure PowerShell build step. Choose the build definition under the **Build definitions** category and then choose the **Edit** link.

The screenshot shows the 'Build definitions' list in Visual Studio Online. On the left, there's a sidebar with sections like 'My favorites', 'Team favorites', and 'Build definitions'. Under 'Build definitions', 'DSC-CI' is listed. On the right, the details for 'DSC-CI' are shown, with the 'Edit' button highlighted by a red box.

2. Add a new **Azure PowerShell** build step to the build definition and then choose the **Add build step...** button.

The screenshot shows the 'Builds' screen for the 'DSC-CI' build definition. It includes tabs for 'Build', 'Options', 'Repository', 'Variables', and 'Triggers'. Below these are buttons for 'Save', 'Queue build...', and 'Undo'. A prominent red box highlights the 'Add build step...' button.

3. Choose the **Deploy task** category, select the **Azure PowerShell** task, and then choose its **Add** button.

The screenshot shows the 'ADD TASKS' dialog box. The 'Deploy' category is selected on the left. In the main area, several tasks are listed with their icons and descriptions. The 'Azure PowerShell' task is highlighted with a red box, and its 'Add' button is also highlighted.

4. Choose the **Azure PowerShell** build step and then fill in its values.

- a. If you already have an Azure service endpoint added to VS Team Services, choose the subscription in the **Azure Subscription** drop down list box and then skip to the next section.

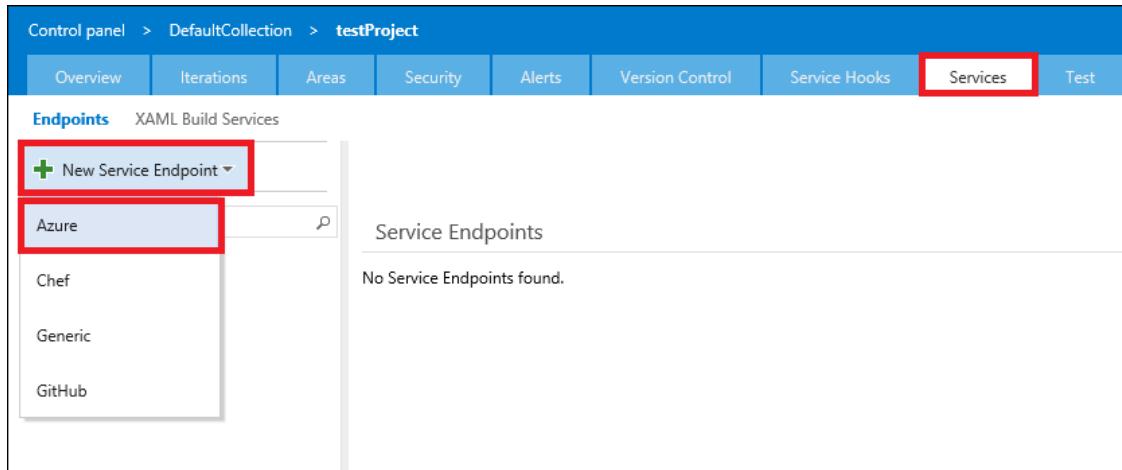
If you don't have an Azure service endpoint in VS Team Services, you need to add one. This subsection takes you through the process. If your Azure account uses a Microsoft account (such as

Hotmail), you need to take the following steps to get a Service Principal authentication.

- b. Choose the **Manage** link next to the **Azure Subscription** drop down list box.



- c. Choose **Azure** in the **New Service Endpoint** drop down list box.



- d. In the **Add Azure Subscription** dialog box, select the **Service Principal** option.



- e. Add your Azure subscription information to the **Add Azure Subscription** dialog box. You need to provide the following items:

- Subscription Id
- Subscription Name
- Service Principal Id
- Service Principal Key
- Tenant Id

- f. Add a name of your choice to the **Subscription** name box. This value appears later in the **Azure Subscription** drop down list in VS Team Services.

- g. If you don't know your Azure subscription ID, you can use one of the following commands to retrieve it.

For PowerShell scripts, use:

```
Get-AzureRmSubscription
```

For Azure CLI, use:

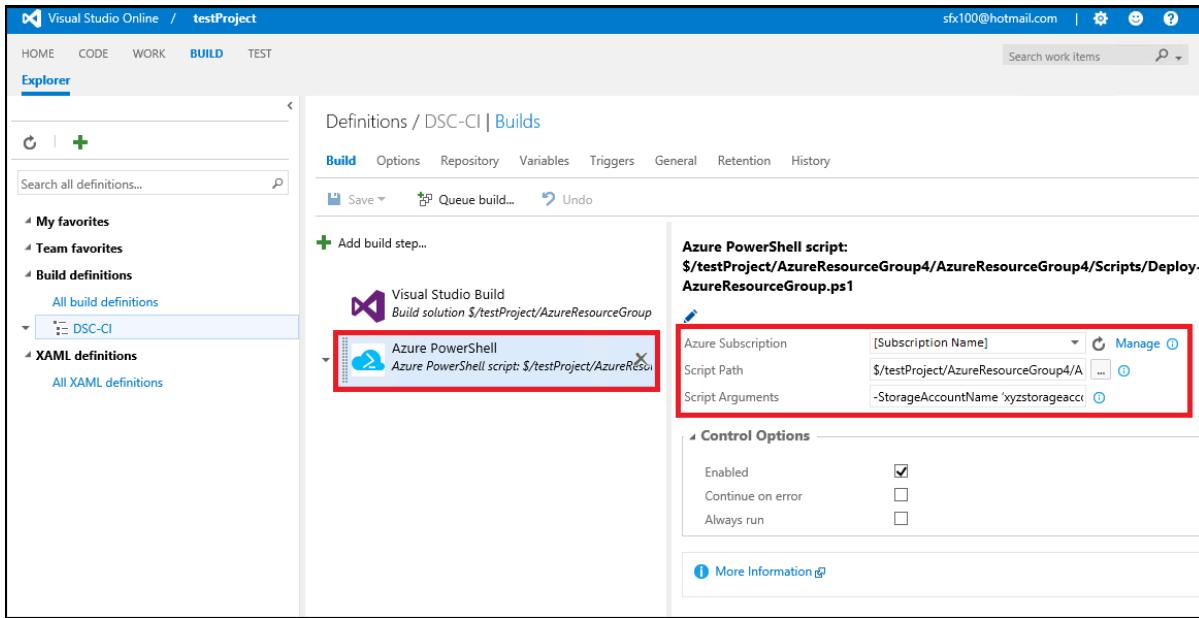
```
az account show
```

- h. To get a Service Principal ID, Service Principal Key, and Tenant ID, follow the procedure in [Create Active Directory application and service principal using portal](#) or [Authenticating a service principal with Azure Resource Manager](#).

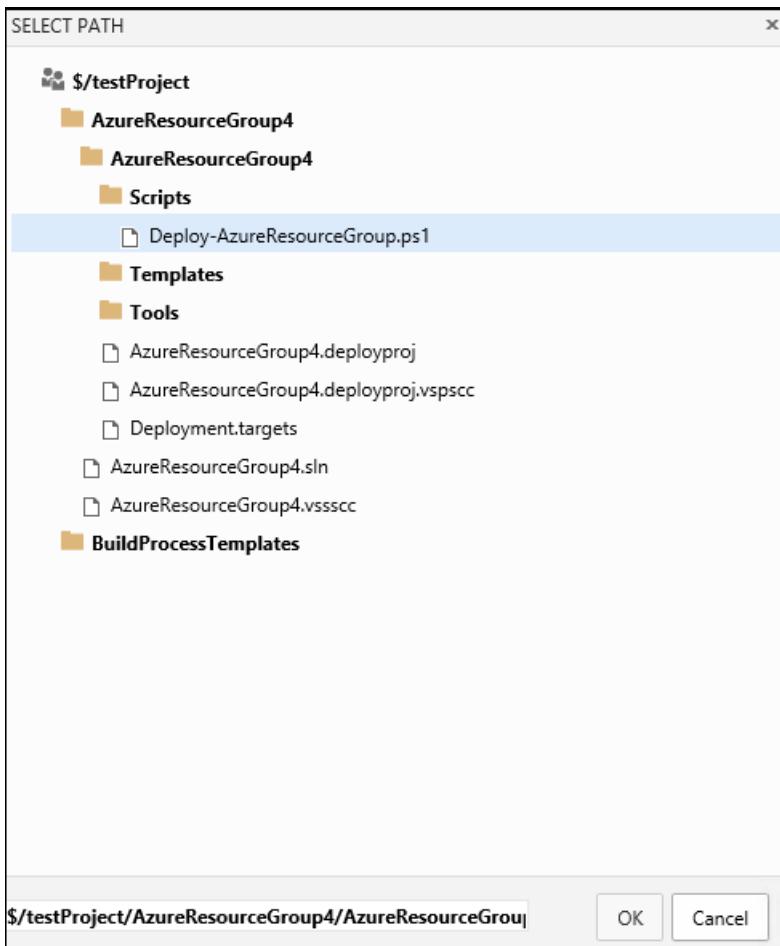
- i. Add the Service Principal ID, Service Principal Key, and Tenant ID values to the **Add Azure Subscription** dialog box and then choose the **OK** button.

You now have a valid Service Principal to use to run the Azure PowerShell script.

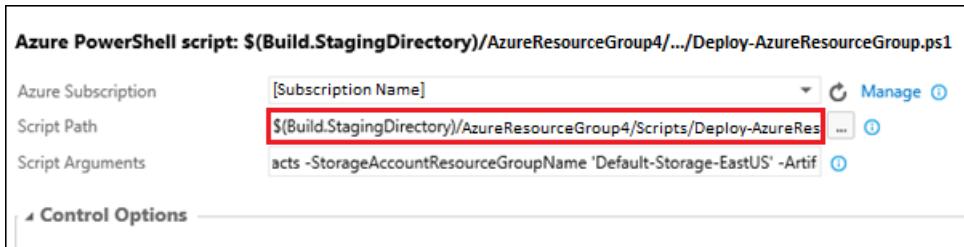
5. Edit the build definition and choose the **Azure PowerShell** build step. Select the subscription in the **Azure Subscription** drop down list box. (If the subscription doesn't appear, choose the **Refresh** button next the **Manage** link.)



6. Provide a path to the Deploy-AzureResourceGroup.ps1 PowerShell script. To do this, choose the ellipsis (...) button next to the **Script Path** box, navigate to the Deploy-AzureResourceGroup.ps1 PowerShell script in the **Scripts** folder of your project, select it, and then choose the **OK** button.



7. After you select the script, update the path to the script so that it's run from the Build.StagingDirectory (the same directory that *ArtifactsLocation* is set to). You can do this by adding "\$(Build.StagingDirectory)/" to the beginning of the script path.



8. In the **Script Arguments** box, enter the following parameters (in a single line). When you run the script in Visual Studio, you can see how VS uses the parameters in the **Output** window. You can use this as a starting point for setting the parameter values in your build step.

PARAMETER	DESCRIPTION
-ResourceGroupLocation	The geo-location value where the resource group is located, such as eastus or ' East US '. (Add single quotes if there's a space in the name.) See Azure Regions for more information.
-ResourceGroupName	The name of the resource group used for this deployment.
-UploadArtifacts	This parameter, when present, specifies that artifacts need to be uploaded to Azure from the local system. You only need to set this switch if your template deployment requires extra artifacts that you want to stage using the PowerShell script (such as configuration scripts or nested templates).
-StorageAccountName	The name of the storage account used to stage artifacts for this deployment. This parameter is only used if you are staging artifacts for deployment. If this parameter is supplied, a new storage account is created if the script has not created one during a previous deployment. If the parameter is specified, the storage account must already exist.
-StorageAccountResourceGroupName	The name of the resource group associated with the storage account. This parameter is required only if you provide a value for the StorageAccountName parameter.
-TemplateFile	The path to the template file in the Azure Resource Group deployment project. To enhance flexibility, use a path for this parameter that is relative to the location of the PowerShell script instead of an absolute path.
-TemplateParametersFile	The path to the parameters file in the Azure Resource Group deployment project. To enhance flexibility, use a path for this parameter that is relative to the location of the PowerShell script instead of an absolute path.
-ArtifactStagingDirectory	This parameter lets the PowerShell script know the folder from where the project's binary files should be copied. This value overrides the default value used by the PowerShell script. For VS Team Services use, set the value to: -ArtifactStagingDirectory \$(Build.StagingDirectory)

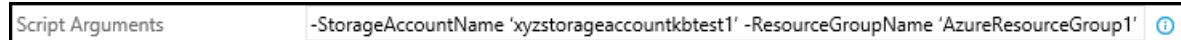
Here's a script arguments example (line broken for readability):

```

-ResourceGroupName 'MyGroup' -ResourceGroupLocation 'eastus' -TemplateFile
'..\templates\azuredeploy.json'
-TemplateParametersFile '..\templates\azuredeploy.parameters.json' -UploadArtifacts -StorageAccountName
'mystorageacct'
-StorageAccountResourceGroupName 'Default-Storage-EastUS' -ArtifactStagingDirectory
'$(Build.StagingDirectory)'

```

When you're finished, the **Script Arguments** box should resemble the following list:

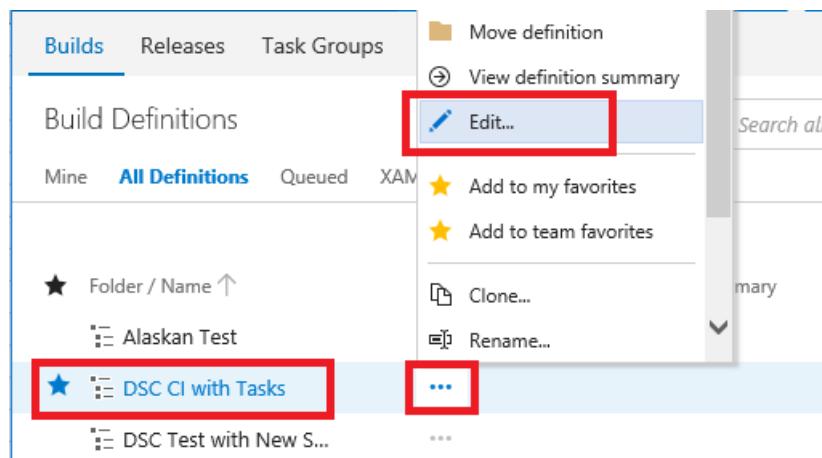


- After you've added all the required items to the Azure PowerShell build step, choose the **Queue** build button to build the project. The **Build** screen shows the output from the PowerShell script.

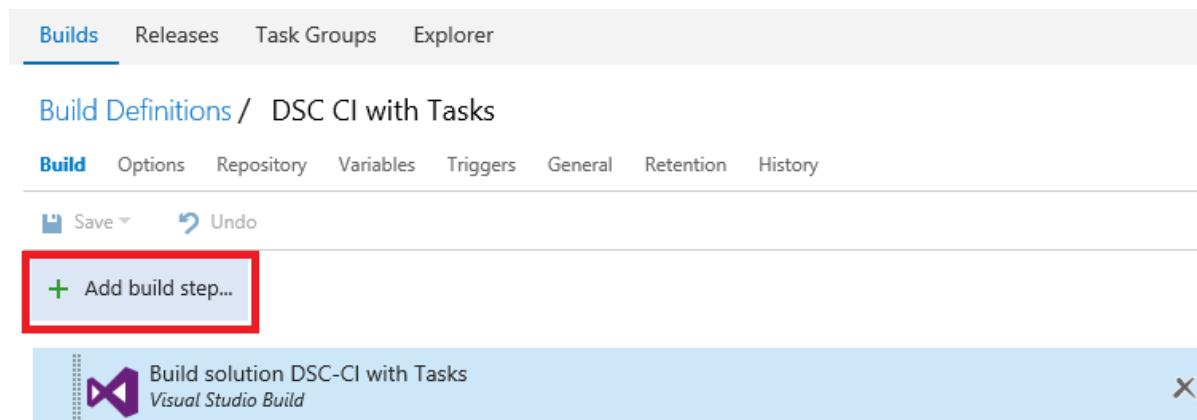
Detailed walkthrough for Option 2

The following steps walk you through the steps necessary to configure continuous deployment in VS Team Services using the built-in tasks.

- Edit your VS Team Services build definition to add two new build steps. Choose the build definition under the **Build definitions** category and then choose the **Edit** link.



- Add the new build steps to the build definition using the **Add build step...** button.



- Choose the **Deploy** task category, select the **Azure File Copy** task, and then choose its **Add** button.

Task catalog

All Build Utility Test Package Deploy

Azure File Copy

Azure PowerShell

Azure Resource Group Deployment

Azure SQL Database Deployment

Chef

Don't see what you need? Check out our Marketplace.

Add Add Add Add Add Add Add Add Add

Close

The screenshot shows the 'Task catalog' window with the 'Deploy' category selected. The 'Azure Resource Group Deployment' task is highlighted with a red box, and its 'Add' button is also highlighted with a red box. Other tasks listed include 'Azure File Copy', 'Azure PowerShell', 'Azure SQL Database Deployment', and 'Chef'. There is also a link to the 'Marketplace'.

4. Choose the **Azure Resource Group Deployment** task, then choose its **Add** button and then **Close** the **Task Catalog**.

The screenshot shows the Task catalog interface in VS Team Services. On the left, there's a sidebar with categories: All, Build, Utility, Test, Package, Deploy (which is highlighted with a red box), and a link to the Marketplace. The main area lists various tasks with their descriptions and an 'Add' button. One task, 'Azure Resource Group Deployment', is also highlighted with a red box. At the bottom right of the catalog window is a 'Close' button.

5. Choose the **Azure File Copy** task and fill in its values.

If you already have an Azure service endpoint added to VS Team Services, choose the subscription in the **Azure Subscription** drop down list box. If you do not have a subscription see [Option 1](#) for instructions on setting one up in VS Team Services.

- Source - enter **`$(Build.StagingDirectory)`**
- Azure Connection Type - select **Azure Resource Manager**
- Azure RM Subscription - select the subscription for the storage account you want to use in the **Azure Subscription** drop down list box. If the subscription doesn't appear, choose the **Refresh** button next the **Manage** link.
- Destination Type - select **Azure Blob**
- RM Storage Account - select the storage account you would like to use for staging artifacts
- Container Name - enter the name of the container you would like to use for staging, it can be any valid container name but use one dedicated to this build definition

For the output values:

- Storage Container URI - enter **`artifactsLocation`**
- Storage Container SAS Token - enter **`artifactsLocationSasToken`**

Build Definitions / DSC CI with Tasks

not built Summary Queue new build S

Build Options Repository Variables Triggers General Retention History

Save Undo

Add build step...

Build solution DSC-CI with Visual Studio Build

AzureBlob File Copy

Azure Deployment

AzureBlob File Copy

Source

`$(Build.StagingDirectory)`

Azure Connection Type

Azure Resource Manager

Azure RM Subscription

Azure Build Principal

Destination Type

Azure Blob

RM Storage Account

stageec0f79d316f04892816

Container Name

nestedsamplevsts

Blob Prefix

Additional Arguments

Output

Storage Container URI

artifactsLocation

Storage Container SAS Token

artifactsLocationSasToken

Control Options

6. Choose the **Azure Resource Group Deployment** build step and then fill in its values.

- Azure Connection Type - select **Azure Resource Manager**
- Azure RM Subscription - select the subscription for deployment in the **Azure Subscription** drop down list box. This will usually be the same subscription used in the previous step.
- Action - select **Create or Update Resource Group**
- Resource Group - select a resource group or enter the name of a new resource group for the deployment
- Location - select the location for the resource group
- Template - enter the path and name of the template to be deployed prepending **\$(Build.StagingDirectory)**, for example: **\$(Build.StagingDirectory)/DSC-CI/azuredeploy.json**
- Template Parameters - enter the path and name of the parameters to be used, prepending **\$(Build.StagingDirectory)**, for example: **\$(Build.StagingDirectory)/DSC-CI/azuredeploy.parameters.json**
- Override Template Parameters - enter or copy and paste the following code:

```
_artifactsLocation $(artifactsLocation) _artifactsLocationSasToken (ConvertTo-SecureString -  
String "$(artifactsLocationSasToken)" -AsPlainText -Force)
```

Save Undo

Add build step...

Build solution DSC-CI with Visual Studio Build

 AzureBlob File Copy
Azure File Copy Azure Deployment
Azure Resource Group Deploy**Azure Deployment**

Azure Connection Type	Azure Resource Manager
Azure RM Subscription	Azure Build Principal
Action	Create Or Update Resource Group
Resource Group	DSC-CI
Location	Central US
Template	\$(Build.StagingDirectory)/DSC-CI/azuredeploy.json
Template Parameters	\$(Build.StagingDirectory)/DSC-CI/azuredeploy.parameters.json
Override Template Parameters	-_artifactsLocation \$(artifactsLocation) -_artifactsLocationSasToken (ConvertTo-SecureString)
Deployment Mode	Incremental
Enable Deployment Prerequisites	<input type="checkbox"/>

7. After you've added all the required items, save the build definition and choose **Queue new build** at the top.

Next steps

Read [Azure Resource Manager overview](#) to learn more about Azure Resource Manager and Azure resource groups.

Use Key Vault to pass secure parameter value during deployment

1/17/2017 • 4 min to read • [Edit Online](#)

When you need to pass a secure value (like a password) as a parameter during deployment, you can retrieve the value from an [Azure Key Vault](#). You retrieve the value by referencing the key vault and secret in your parameter file. The value is never exposed because you only reference its key vault ID. You do not need to manually enter the value for the secret each time you deploy the resources. The key vault can exist in a different subscription than the resource group you are deploying to. When referencing the key vault, you include the subscription ID.

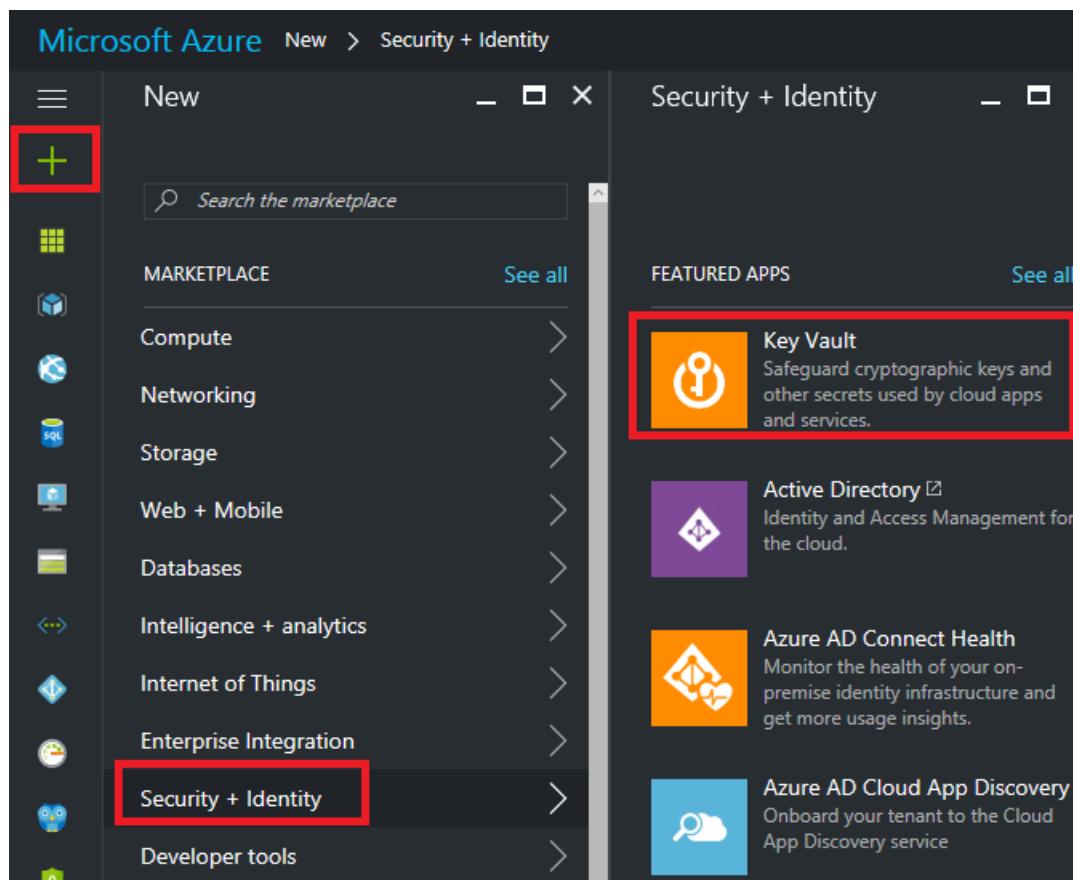
This topic shows you how to create a key vault and secret, configure access to the secret for a Resource Manager template, and pass the secret as a parameter. If you already have a key vault and secret, but need to check template and user access, go to the [Enable access to the secret](#) section. If you already have the key vault and secret, and are sure it is configured for template and user access, go to the [Reference a secret with static id](#) section.

Deploy a key vault and secret

You can deploy a key vault and secret through a Resource Manager template. For an example, see [Key vault template](#) and [Key vault secret template](#). When creating the key vault, set the **enabledForTemplateDeployment** property to **true** so it can be referenced from other Resource Manager templates.

Or, you can create the key vault and secret through the Azure portal.

1. Select **New** -> **Security + Identity** -> **Key Vault**.



2. Provide values for the key vault. You can ignore the **Access policies** and **Advanced access policy** settings for now. Those settings are covered in the section. Select **Create**.

Create Key Vault

* Name
examplevault

* Subscription
Azure Consumption

* Resource Group ⓘ
 Create new Use existing
vault-group

* Location
South Central US

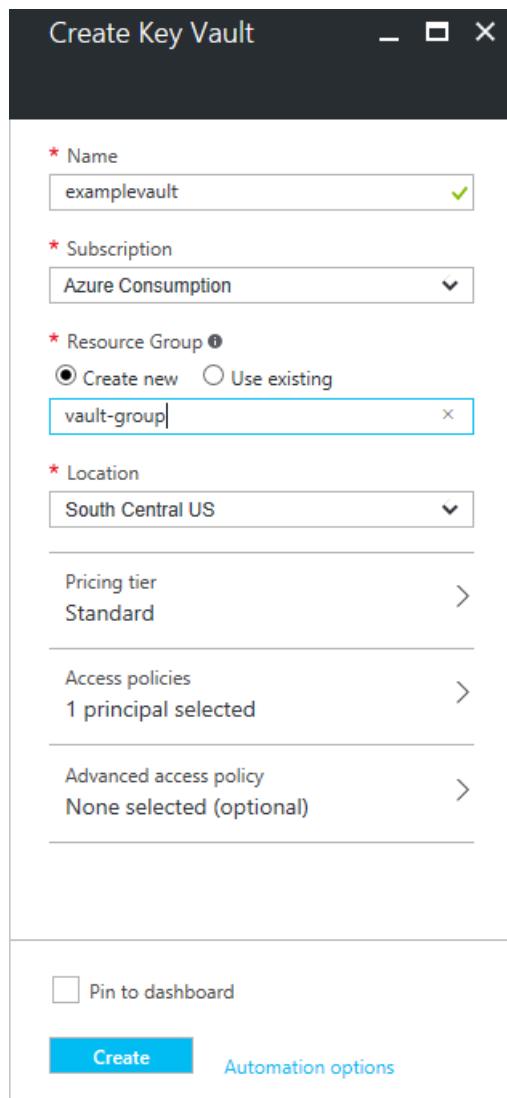
Pricing tier
Standard

Access policies
1 principal selected

Advanced access policy
None selected (optional)

Pin to dashboard

Create [Automation options](#)



3. You now have a key vault. Select that key vault.

4. In the key vault blade, select **Secrets**.

examplevault
Key vault

Delete Refresh Move

Search (Ctrl+/)

Overview

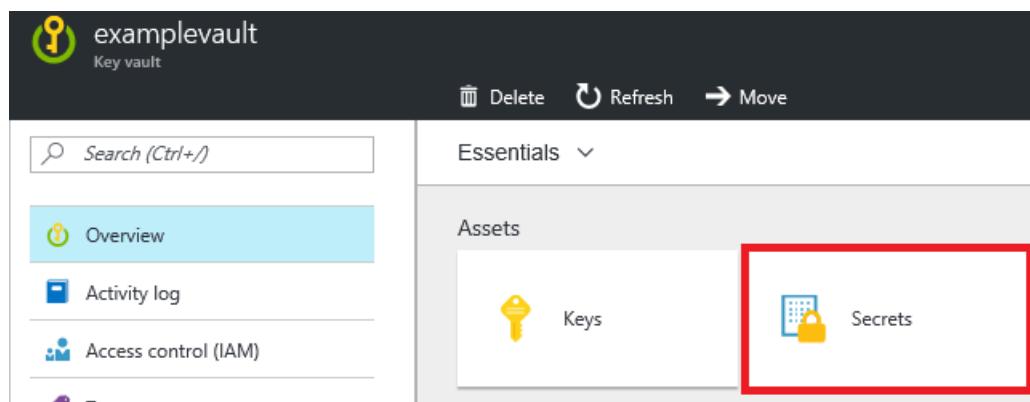
Activity log

Access control (IAM)

Assets

Keys

Secrets

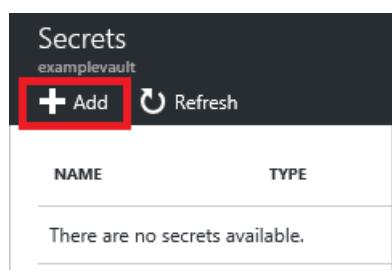


5. Select **Add**.

Secrets
examplevault

+ Add Refresh

NAME	TYPE
There are no secrets available.	



6. Select **Manual** for upload options. Provide a name and value for secret. Select **Create**.

Create a secret

Upload options

Manual

* Name

examplesecret

* Value

Content type (optional)

Set activation date.

Set expiration date.

Enabled Yes No

Pin to dashboard

Create

You have created your key vault and secret.

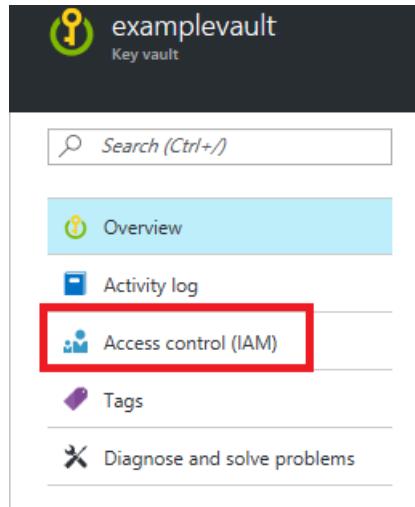
Enable access to the secret

Whether you are using a new key vault or an existing one, ensure that the user deploying the template can access the secret. The user deploying a template that references a secret must have the

`Microsoft.KeyVault/vaults/deploy/action` permission for the key vault. The **Owner** and **Contributor** roles both grant this access. You can also create a **custom role** that grants this permission and add the user to that role. Also, you must grant Resource Manager the ability to access the key vault during deployment.

You can check or perform these steps through the portal.

1. Select **Access control (IAM)**.



2. If the account you intend to use for deploying templates is not already an Owner or Contributor (or added to a custom role that grants `Microsoft.KeyVault/vaults/deploy/action` permission), select **Add**

The screenshot shows the 'Access control (IAM)' section of the Azure Key Vault 'examplevault'. A red box highlights the '+ Add' button. To its right, a callout box provides information about custom roles and groups. Below is a table mapping users to roles:

USER	ROLE
Subscription admins	Owner

3. Select the Contributor or Owner role, and search for the identity to assign to that role. Select **Okay** to complete adding the identity to the role.

The image displays two windows side-by-side. The left window is titled 'Add access' and shows step 1 ('Select a role: Contributor') with a red box around it, and step 2 ('Add users: None selected'). The right window is titled 'Add users' and shows a search result for 'exampleapp' with its details.

4. To enable access from a template during deployment, select **Advanced access control**. Select the option **Enable access to Azure Resource Manager for template deployment**.

The screenshot shows the 'Advanced access policies' section of the Azure Key Vault 'examplevaulttf'. A red box highlights the 'Enable access to Azure Resource Manager for template deployment' checkbox, which is checked. Other options shown are 'Enable access to Azure Virtual Machines for deployment' (unchecked) and 'Enable access to Azure Disk Encryption for volume encryption' (unchecked).

Reference a secret with static id

You reference the secret from within a **parameters file (not the template)** that passes values to your template.

You reference the secret by passing the resource identifier of the key vault and the name of the secret. In the following example, the key vault secret must already exist, and you provide a static value for its resource ID.

```
{  
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "sqlsvrAdminLoginPassword": {  
            "reference": {  
                "keyVault": {  
                    "id": "/subscriptions/{guid}/resourceGroups/{group-name}/providers/Microsoft.KeyVault/vaults/{vault-name}"  
                },  
                "secretName": "adminPassword"  
            }  
        },  
        "sqlsvrAdminLogin": {  
            "value": "exampleadmin"  
        }  
    }  
}
```

In the template, the parameter that accepts the secret should be a **securestring**. The following example shows the relevant sections of a template that deploys a SQL server that requires an administrator password.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "sqlsvrAdminLogin": {  
            "type": "string",  
            "minLength": 4  
        },  
        "sqlsvrAdminLoginPassword": {  
            "type": "securestring"  
        },  
        ...  
    },  
    "resources": [  
        {  
            "name": "[variables('sqlsvrName')]",  
            "type": "Microsoft.Sql/servers",  
            "location": "[resourceGroup().location]",  
            "apiVersion": "2014-04-01-preview",  
            "properties": {  
                "administratorLogin": "[parameters('sqlsvrAdminLogin')]",  
                "administratorLoginPassword": "[parameters('sqlsvrAdminLoginPassword')]"  
            },  
            ...  
        }  
    ],  
    "variables": {  
        "sqlsvrName": "[concat('sqlsvr', uniqueString(resourceGroup().id))]"  
    },  
    "outputs": { }  
}
```

Reference a secret with dynamic id

The previous section showed how to pass a static resource ID for the key vault secret. However, in some scenarios, you need to reference a key vault secret that varies based on the current deployment. In that case, you cannot

hard-code the resource ID in the parameters file. Unfortunately, you cannot dynamically generate the resource ID in the parameters file because template expressions are not permitted in the parameters file.

To dynamically generate the resource ID for a key vault secret, you must move the resource that needs the secret into a nested template. In your master template, you add the nested template and pass in a parameter that contains the dynamically generated resource ID.

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "vaultName": {  
            "type": "string"  
        },  
        "secretName": {  
            "type": "string"  
        }  
    },  
    "resources": [  
        {  
            "apiVersion": "2015-01-01",  
            "name": "nestedTemplate",  
            "type": "Microsoft.Resources/deployments",  
            "properties": {  
                "mode": "incremental",  
                "templateLink": {  
                    "uri": "https://www.contoso.com/AzureTemplates/newVM.json",  
                    "contentVersion": "1.0.0.0"  
                },  
                "parameters": {  
                    "adminPassword": {  
                        "reference": {  
                            "keyVault": {  
                                "id": "[concat(resourceGroup().id, '/providers/Microsoft.KeyVault/vaults/',  
parameters('vaultName'))]"  
                            },  
                            "secretName": "[parameters('secretName')]"  
                        }  
                    }  
                }  
            }  
        }  
    ],  
    "outputs": {}  
}
```

Next steps

- For general information about key vaults, see [Get started with Azure Key Vault](#).
- For information about using a key vault with a Virtual Machine, see [Security considerations for Azure Resource Manager](#).
- For complete examples of referencing key secrets, see [Key Vault examples](#).

Manage resources with Azure PowerShell and Resource Manager

3/3/2017 • 8 min to read • [Edit Online](#)

In this topic, you learn how to manage your solutions with Azure PowerShell and Azure Resource Manager. If you are not familiar with Resource Manager, see [Resource Manager Overview](#). This topic focuses on management tasks. You will:

1. Create a resource group
2. Add a resource to the resource group
3. Add a tag to the resource
4. Query resources based on names or tag values
5. Apply and remove a lock on the resource
6. Create a Resource Manager template from your resource group
7. Delete a resource group

Get started with Azure PowerShell

If you have not installed Azure PowerShell, see [How to install and configure Azure PowerShell](#).

If you have installed Azure PowerShell in the past but have not updated it recently, consider installing the latest version. You can update the version through the same method you used to install it. For example, if you used the Web Platform Installer, launch it again and look for an update.

To check your version of the Azure Resources module, use the following cmdlet:

```
Get-Module -ListAvailable -Name AzureRm.Resources | Select Version
```

This topic was updated for version 3.3.0. If you have an earlier version, your experience might not match the steps shown in this topic. For documentation about the cmdlets in this version, see [AzureRM.Resources Module](#).

Log in to your Azure account

Before working on your solution, you must log in to your account.

To log in to your Azure account, use the **Login-AzureRmAccount** cmdlet.

```
Login-AzureRmAccount
```

The cmdlet prompts you for the login credentials for your Azure account. After logging in, it downloads your account settings so they are available to Azure PowerShell.

The cmdlet returns information about your account and the subscription to use for the tasks.

```
Environment      : AzureCloud
Account         : example@contoso.com
TenantId        : {guid}
SubscriptionId  : {guid}
SubscriptionName : Example Subscription One
CurrentStorageAccount :
```

If you have more than one subscription, you can switch to a different subscription. First, let's see all the subscriptions for your account.

```
Get-AzureRmSubscription
```

It returns enabled and disabled subscriptions.

```
SubscriptionName : Example Subscription One
SubscriptionId   : {guid}
TenantId        : {guid}
State           : Enabled

SubscriptionName : Example Subscription Two
SubscriptionId   : {guid}
TenantId        : {guid}
State           : Enabled

SubscriptionName : Example Subscription Three
SubscriptionId   : {guid}
TenantId        : {guid}
State           : Disabled
```

To switch to a different subscription, provide the subscription name with the **Set-AzureRmContext** cmdlet.

```
Set-AzureRmContext -SubscriptionName "Example Subscription Two"
```

Create a resource group

Before deploying any resources to your subscription, you must create a resource group that will contain the resources.

To create a resource group, use the **New-AzureRmResourceGroup** cmdlet. The command uses the **Name** parameter to specify a name for the resource group and the **Location** parameter to specify its location.

```
New-AzureRmResourceGroup -Name TestRG1 -Location "South Central US"
```

The output is in the following format:

```
ResourceGroupName : TestRG1
Location         : southcentralus
ProvisioningState : Succeeded
Tags             :
ResourceId       : /subscriptions/{guid}/resourceGroups/TestRG1
```

If you need to retrieve the resource group later, use the following cmdlet:

```
Get-AzureRmResourceGroup -ResourceGroupName TestRG1
```

To get all the resource groups in your subscription, do not specify a name:

```
Get-AzureRmResourceGroup
```

Add resources to a resource group

To add a resource to the resource group, you can use the **New-AzureRmResource** cmdlet or a cmdlet that is specific to the type of resource you are creating (like **New-AzureRmStorageAccount**). You might find it easier to use a cmdlet that is specific to a resource type because it includes parameters for the properties that are needed for the new resource. To use **New-AzureRmResource**, you must know all the properties to set without being prompted for them.

However, adding a resource through cmdlets might cause future confusion because the new resource does not exist in a Resource Manager template. Microsoft recommends defining the infrastructure for your Azure solution in a Resource Manager template. Templates enable you to reliably and repeatedly deploy your solution. This topic does not show how to deploy a Resource Manager template to your subscription. For that information, see [Deploy resources with Resource Manager templates and Azure PowerShell](#). For this topic, you create a storage account with a PowerShell cmdlet, but later you generate a template from your resource group.

The following cmdlet creates a storage account. Instead of using the name shown in the example, provide a unique name for the storage account. The name must be between 3 and 24 characters in length, and use only numbers and lower-case letters. If you use the name shown in the example, you receive an error because that name is already in use.

```
New-AzureRmStorageAccount -ResourceGroupName TestRG1 -AccountName mystoragename -Type "Standard_LRS" -Location "South Central US"
```

If you need to retrieve this resource later, use the following cmdlet:

```
Get-AzureRmResource -ResourceName mystoragename -ResourceGroupName TestRG1
```

Add a tag

Tags enable you to organize your resources according to different properties. For example, you may have several resources in different resource groups that belong to the same department. You can apply a department tag and value to those resources to mark them as belonging to the same category. Or, you can mark whether a resource is used in a production or test environment. In this topic, you apply tags to only one resource, but in your environment it most likely makes sense to apply tags to all your resources.

The following cmdlet applies two tags to your storage account:

```
Set-AzureRmResource -Tag @{ Dept="IT"; Environment="Test" } -ResourceName mystoragename -ResourceGroupName TestRG1 -ResourceType Microsoft.Storage/storageAccounts
```

Tags are updated as a single object. To add a tag to a resource that already includes tags, first retrieve the existing tags. Add the new tag to the object that contains the existing tags, and reapply all the tags to the resource.

```
$tags = (Get-AzureRmResource -ResourceName mystoragename -ResourceGroupName TestRG1).Tags  
$tags += @{$Status="Approved"}  
Set-AzureRmResource -Tag $tags -ResourceName mystoragename -ResourceGroupName TestRG1 -ResourceType  
Microsoft.Storage/storageAccounts
```

Search for resources

Use the **Find-AzureRmResource** cmdlet to retrieve resources for different search conditions.

- To get a resource by name, provide the **ResourceNameContains** parameter:

```
Find-AzureRmResource -ResourceNameContains mystoragename
```

- To get all the resources in a resource group, provide the **ResourceGroupNameContains** parameter:

```
Find-AzureRmResource -ResourceGroupNameContains TestRG1
```

- To get all the resources with a tag name and value, provide the **TagName** and **TagValue** parameters:

```
Find-AzureRmResource -TagName Dept -TagValue IT
```

- To get all the resources with a particular resource type, provide the **ResourceType** parameter:

```
Find-AzureRmResource -ResourceType Microsoft.Storage/storageAccounts
```

Lock a resource

When you need to make sure a critical resource is not accidentally deleted or modified, apply a lock to the resource. You can specify either a **CanNotDelete** or **ReadOnly**.

To create or delete management locks, you must have access to `Microsoft.Authorization/*` or `Microsoft.Authorization/locks/*` actions. Of the built-in roles, only Owner and User Access Administrator are granted those actions.

To apply a lock, use the following cmdlet:

```
New-AzureRmResourceLock -LockLevel CanNotDelete -LockName LockStorage -ResourceName mystoragename -  
ResourceType Microsoft.Storage/storageAccounts -ResourceGroupName TestRG1
```

The locked resource in the preceding example cannot be deleted until the lock is removed. To remove a lock, use:

```
Remove-AzureRmResourceLock -LockName LockStorage -ResourceName mystoragename -ResourceType  
Microsoft.Storage/storageAccounts -ResourceGroupName TestRG1
```

For more information about setting locks, see [Lock resources with Azure Resource Manager](#).

Export Resource Manager template

For an existing resource group (deployed through PowerShell or one of the other methods like the portal), you can view the Resource Manager template for the resource group. Exporting the template offers two benefits:

1. You can easily automate future deployments of the solution because all the infrastructure is defined in the template.
2. You can become familiar with template syntax by looking at the JavaScript Object Notation (JSON) that represents your solution.

NOTE

The export template feature is in preview, and not all resource types currently support exporting a template. When attempting to export a template, you may see an error that states some resources were not exported. If needed, you can manually define these resources in your template after downloading it.

To view the template for a resource group, run the **Export-AzureRmResourceGroup** cmdlet.

```
Export-AzureRmResourceGroup -ResourceGroupName TestRG1 -Path c:\Azure\Templates\Downloads\TestRG1.json
```

There are many options and scenarios for exporting a Resource Manager template. For more information, see [Export an Azure Resource Manager template from existing resources](#).

Remove resources or resource group

You can remove a resource or resource group. When you remove a resource group, you also remove all the resources within that resource group.

- To delete a resource from the resource group, use the **Remove-AzureRmResource** cmdlet. This cmdlet deletes the resource, but does not delete the resource group.

```
Remove-AzureRmResource -ResourceName mystoragename -ResourceType Microsoft.Storage/storageAccounts -  
ResourceGroupName TestRG1
```

- To delete a resource group and all its resources, use the **Remove-AzureRmResourceGroup** cmdlet.

```
Remove-AzureRmResourceGroup -Name TestRG1
```

For both cmdlets, you are asked to confirm that you wish to remove the resource or resource group. If the operation successfully deletes the resource or resource group, it returns **True**.

Run Resource Manager scripts with Azure Automation

This topic shows you how to perform basic operations on your resources with Azure PowerShell. For more advanced management scenarios, you typically want to create a script, and reuse that script as needed or on a schedule. [Azure Automation](#) provides a way for you to automate frequently used scripts that manage your Azure solutions.

The following topics show you how to use Azure Automation, Resource Manager, and PowerShell to effectively perform management tasks:

- For information about creating a runbook, see [My first PowerShell runbook](#).
- For information about working with galleries of scripts, see [Runbook and module galleries for Azure Automation](#).
- For runbooks that start and stop virtual machines, see [Azure Automation scenario: Using JSON-formatted tags to create a schedule for Azure VM startup and shutdown](#).
- For runbooks that start and stop virtual machines off-hours, see [Start/Stop VMs during off-hours solution in](#)

Next Steps

- To learn about creating Resource Manager templates, see [Authoring Azure Resource Manager Templates](#).
- To learn about deploying templates, see [Deploy an application with Azure Resource Manager Template](#).
- You can move existing resources to a new resource group. For examples, see [Move Resources to New Resource Group or Subscription](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Use the Azure CLI to manage Azure resources and resource groups

3/20/2017 • 7 min to read • [Edit Online](#)

The Azure Command-Line Interface (Azure CLI) is one of several tools you can use to deploy and manage resources with Resource Manager. This article introduces common ways to manage Azure resources and resource groups by using the Azure CLI in Resource Manager mode. For information about using the CLI to deploy resources, see [Deploy resources with Resource Manager templates and Azure CLI](#). For background about Azure resources and Resource Manager, visit the [Azure Resource Manager Overview](#).

NOTE

To manage Azure resources with the Azure CLI, you need to [install the Azure CLI](#), and [log in to Azure](#) by using the `azure login` command. Make sure the CLI is in Resource Manager mode (run `azure config mode arm`). If you've done these things, you're ready to go.

Get resource groups and resources

Resource groups

To get a list of all resource groups in your subscription and their locations, run this command.

```
azure group list
```

Resources

To list all resources in a group, such as one with name `testRG`, use the following command.

```
azure resource list testRG
```

To view an individual resource within the group, such as a VM named `MyUbuntuVM`, use a command like the following.

```
azure resource show testRG MyUbuntuVM Microsoft.Compute/virtualMachines -o "2015-06-15"
```

Notice the **Microsoft.Compute/virtualMachines** parameter. This parameter indicates the type of the resource you are requesting information on.

NOTE

When using the **azure resource** commands other than the **list** command, you must specify the API version of the resource with the **-o** parameter. If you're unsure about the API version, consult the template file and find the `apiVersion` field for the resource. For more about API versions in Resource Manager, see [Resource Manager providers, regions, API versions, and schemas](#).

When viewing details on a resource, it is often useful to use the `--json` parameter. This parameter makes the output more readable, because some values are nested structures, or collections. The following example demonstrates returning the results of the **show** command as a JSON document.

```
azure resource show testRG MyUbuntuVM Microsoft.Compute/virtualMachines -o "2015-06-15" --json
```

NOTE

If you want, save the JSON data to file by using the > character to direct the output to a file. For example:

```
azure resource show testRG MyUbuntuVM Microsoft.Compute/virtualMachines -o "2015-06-15" --json > myfile.json
```

Tags

To add a tag to a resource group, use **azure group set**. If the resource group does not have any existing tags, pass in the tag.

```
azure group set -n tag-demo-group -t Dept=Finance
```

Tags are updated as a whole. If you want to add a tag to a resource group that has existing tags, pass all the tags.

```
azure group set -n tag-demo-group -t Dept=Finance;Environment=Production;Project=Upgrade
```

Tags are not inherited by resources in a resource group. To add a tag to a resource, use **azure resource set**. Pass the API version number for the resource type that you are adding the tag to. If you need to retrieve the API version, use the following command with the resource provider for the type you are setting:

```
azure provider show -n Microsoft.Storage --json
```

In the results, look for the resource type you want.

```
"resourceTypes": [
{
  "resourceType": "storageAccounts",
  ...
  "apiVersions": [
    "2016-01-01",
    "2015-06-15",
    "2015-05-01-preview"
  ]
}
...
...
```

Now, provide that API version, resource group name, resource name, resource type, and tag value as parameters.

```
azure resource set -g tag-demo-group -n storagetagdemo -r Microsoft.Storage/storageAccounts -t Dept=Finance -o 2016-01-01
```

Tags exist directly on resources and resource groups. To see the existing tags, get a resource group and its resources with **azure group show**.

```
azure group show -n tag-demo-group --json
```

Which returns metadata about the resource group, including any tags applied to it.

```
{  
  "id": "/subscriptions/4705409c-9372-42f0-914c-64a504530837/resourceGroups/tag-demo-group",  
  "name": "tag-demo-group",  
  "properties": {  
    "provisioningState": "Succeeded"  
  },  
  "location": "southcentralus",  
  "tags": {  
    "Dept": "Finance",  
    "Environment": "Production",  
    "Project": "Upgrade"  
  },  
  ...  
}
```

You view the tags for a particular resource by using **azure resource show**.

```
azure resource show -g tag-demo-group -n storagetagdemo -r Microsoft.Storage/storageAccounts -o 2016-01-01 --json
```

To retrieve all the resources with a tag value, use:

```
azure resource list -t Dept=Finance --json
```

To retrieve all the resource groups with a tag value, use:

```
azure group list -t Dept=Finance
```

You can view the existing tags in your subscription with the following command:

```
azure tag list
```

Manage resources

To add a resource such as a storage account to a resource group, run a command similar to:

```
azure resource create testRG MyStorageAccount "Microsoft.Storage/storageAccounts" "westus" -o "2015-06-15" -p  
"{"accountType": "Standard_LRS"}"
```

In addition to specifying the API version of the resource with the **-o** parameter, use the **-p** parameter to pass a JSON-formatted string with any required or additional properties.

To delete an existing resource such as a virtual machine resource, use a command like the following.

```
azure resource delete testRG MyUbuntuVM Microsoft.Compute/virtualMachines -o "2015-06-15"
```

To move existing resources to another resource group or subscription, use the **azure resource move** command. The following example shows how to move a Redis Cache to a new resource group. In the **-i** parameter, provide a comma-separated list of the resource id's to move.

```
azure resource move -i  
"/subscriptions/{guid}/resourceGroups/OldRG/providers/Microsoft.Cache/Redis/examplecache" -d "NewRG"
```

Control access to resources

You can use the Azure CLI to create and manage policies to control access to Azure resources. For background about policy definitions and assigning policies to resources, see [Use policy to manage resources and control access](#).

For example, define the following policy to deny all requests where location is not West US or North Central US, and save it to the policy definition file policy.json:

```
{  
  "if" : {  
    "not" : {  
      "field" : "location",  
      "in" : ["westus", "northcentralus"]  
    }  
  },  
  "then" : {  
    "effect" : "deny"  
  }  
}
```

Then run the **policy definition create** command:

```
azure policy definition create MyPolicy -p c:\temp\policy.json
```

This command shows output similar to the following.

```
+ Creating policy definition MyPolicy  
data: PolicyName: MyPolicy  
data: PolicyDefinitionId: /subscriptions/#####-###-####-####-#####  
#####/providers/Microsoft.Authorization/policyDefinitions/MyPolicy  
  
data: PolicyType: Custom  
data: DisplayName: undefined  
data: Description: undefined  
data: PolicyRule: field=location, in=[westus, northcentralus], effect=deny
```

To assign a policy at the scope you want, use the **PolicyDefinitionId** returned from the previous command. In the following example, this scope is the subscription, but you can scope to resource groups or individual resources:

```
azure policy assignment create MyPolicyAssignment -p /subscriptions/#####-###-####-####-#####  
#####/providers/Microsoft.Authorization/policyDefinitions/MyPolicy -s /subscriptions/#####-###-  
###-####-#####
```

You can get, change, or remove policy definitions by using the **policy definition show**, **policy definition set**, and **policy definition delete** commands.

Similarly, you can get, change, or remove policy assignments by using the **policy assignment show**, **policy assignment set**, and **policy assignment delete** commands.

Export a resource group as a template

For an existing resource group, you can view the Resource Manager template for the resource group. Exporting the template offers two benefits:

1. You can easily automate future deployments of the solution because all the infrastructure is defined in the template.

2. You can become familiar with template syntax by looking at the JSON that represents your solution.

Using the Azure CLI, you can either export a template that represents the current state of your resource group, or download the template that was used for a particular deployment.

- **Export the template for a resource group** - This is helpful when you made changes to a resource group, and need to retrieve the JSON representation of its current state. However, the generated template contains only a minimal number of parameters and no variables. Most of the values in the template are hard-coded. Before deploying the generated template, you may wish to convert more of the values into parameters so you can customize the deployment for different environments.

To export the template for a resource group to a local directory, run the `azure group export` command as shown in the following example. (Substitute a local directory appropriate for your operating system environment.)

```
azure group export testRG ~/azure/templates/
```

- **Download the template for a particular deployment** -- This is helpful when you need to view the actual template that was used to deploy resources. The template includes all parameters and variables defined for the original deployment. However, if someone in your organization made changes to the resource group outside of the definition in the template, this template doesn't represent the current state of the resource group.

To download the template used for a particular deployment to a local directory, run the

```
azure group deployment template download
```

 command. For example:

```
azure group deployment template download TestRG testRGDeploy ~/azure/templates/downloads/
```

NOTE

Template export is in preview, and not all resource types currently support exporting a template. When attempting to export a template, you may see an error that states some resources were not exported. If needed, manually define these resources in your template after downloading it.

Next steps

- To get details of deployment operations and troubleshoot deployment errors with the Azure CLI, see [View deployment operations](#).
- If you want to use the CLI to set up an application or script to access resources, see [Use Azure CLI to create a service principal to access resources](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Manage Azure resources through portal

1/17/2017 • 5 min to read • [Edit Online](#)

This topic shows how to use the [Azure portal](#) with [Azure Resource Manager](#) to manage your Azure resources. To learn about deploying resources through the portal, see [Deploy resources with Resource Manager templates and Azure portal](#).

Currently, not every service supports the portal or Resource Manager. For those services, you need to use the [classic portal](#). For the status of each service, see [Azure portal availability chart](#).

Manage resource groups

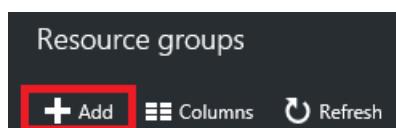
A resource group is a container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. Generally, add resources that share the same lifecycle to the same resource group so you can easily deploy, update, and delete them as a group.

The resource group stores metadata about the resources. Therefore, when you specify a location for the resource group, you are specifying where that metadata is stored. For compliance reasons, you may need to ensure that your data is stored in a particular region.

1. To see all the resource groups in your subscription, select **Resource groups**.

The screenshot shows the Microsoft Azure portal interface. The top navigation bar has 'Microsoft Azure' and a dropdown arrow, followed by 'Resource groups'. On the left, there's a sidebar with icons for 'New', 'Resource groups' (which is highlighted with a red box), 'All resources', 'Recent', 'App Services', and 'SQL databases'. The main content area is titled 'Resource groups' with 'Subscriptions: Windows Azure MSDN - Vis'. It includes a 'Filter items...' input field and a table with two rows. The first row has a blue cube icon and the name 'dashboards'. The second row has a blue cube icon and the name 'ExampleGroup'. At the bottom of the content area are 'Add', 'Columns', and 'Refresh' buttons.

2. To create an empty resource group, select **Add**.



3. Provide a name and location for the new resource group. Select **Create**.

Resource group

Create an empty resource group

* Resource group name
ContosoGroup

* Subscription
Windows Azure MSDN - Visual Studio Ultir ▾

* Resource group location
West US ▾

4. You may need to select **Refresh** to see the recently created resource group.

Resource groups

+ Add Columns Refresh

Subscriptions: Windows Azure MSDN - Visual Studio Ultima

Filter items...

NAME
ContosoGroup
dashboards
ExampleGroup

5. To customize the information displayed for your resource groups, select **Columns**.

Resource groups

+ Add Columns Refresh

6. Select the columns to add, and then select **Update**.

Choose columns

Resource groups

COLUMN

SUBSCRIPTION

LOCATION ⓘ

LOCATION ID ⓘ

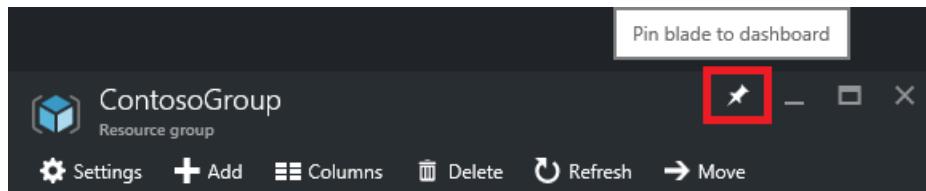
RESOURCE GROUP ID ⓘ

STATUS ⓘ

SUBSCRIPTION ID

7. To learn about deploying resources to your new resource group, see [Deploy resources with Resource Manager templates and Azure portal](#).

8. For quick access to a resource group, you can pin the blade to your dashboard.



9. The dashboard displays the resource group and its resources. You can select either the resource groups or any of its resources to navigate to the item.

The screenshot shows the Azure Dashboard interface. On the left, there's a sidebar with links for 'All resources' (ALL SUBSCRIPTIONS), 'Subscriptions', 'Feedback', 'Marketplace', and 'How it works'. The main area is titled 'Resources' under 'CONTOSOGROUP'. A red box highlights this list, which contains five items:

- ContosoWebAppExample
- contososerverexample
- ContosoData
- ServicePlanf6ec7013-8b9e
- ContosoWebAppExample

Tag resources

You can apply tags to resource groups and resources to logically organize your assets. For information about working with tags, see [Using tags to organize your Azure resources](#).

1. To view the tags for a resource or resource group, select the **Tags** icon.

The screenshot shows the Azure Storage account blade for 'storagew42yqtm7kpjuq'. The left sidebar includes links for 'Overview', 'Activity log', 'Access control (IAM)', and 'Tags'. The 'Tags' link is highlighted with a red box. The main content area is currently empty, showing a message: 'There are no tags applied to this resource yet.'

2. You see the existing tags for the resource. If you have not previously applied tags, the list is empty.

Save

 Tags are key/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. [Learn more](#)

* Key : * Value :

Dept : Finance ...
Environment : Production ...

3. To add a tag, type a key and value, or select an existing one from the dropdown menu. Select **Save**.

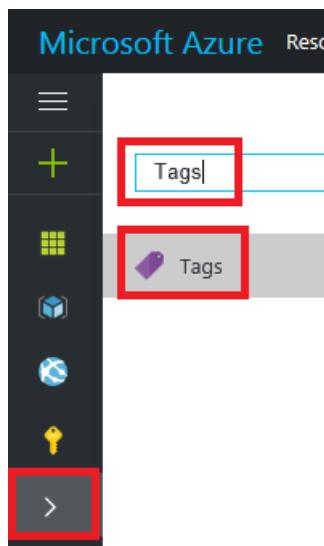
Save

 Tags are key/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. [Learn more](#)

* Key : * Value :

Dept : Finance ...
Environment : Production ...

4. To view all the resources with a tag value, select > (More services), and enter the word **Tags** into the filter text box. Select **Tags** from the available options.



5. You see a summary of the tags in your subscriptions.

The screenshot shows the Microsoft Tags blade. At the top, it says "Subscriptions: All 5 selected". Below this is a dropdown menu set to "All subscriptions". A callout box provides information about tags: "Tags are key/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups." There are three entries listed under the tag "Environment : Production": "CostCenter : IT", "Dept : Finance", and "Environment : Production". Each entry has a three-dot ellipsis icon to its right.

6. Select any of the tags to display the resources and resource groups with that tag.

The screenshot shows the details for the "Environment : Production" tag. The title bar says "Environment : Production" and "Tag". Below this is a "Subscriptions: All 5 selected" section with a "Filter items..." input field and a "All subscriptions" dropdown. The main area is a table with columns "NAME" and "SUBSCRIPTION". It lists five resources: "TagTestGroup" (Subscription: storage2w42yqtm7kpjuq), and four storage accounts ("storage3w42yqtm7kpjuq", "storage4w42yqtm7kpjuq", "storage5w42yqtm7kpjuq", "storage6w42yqtm7kpjuq") all under the same subscription.

7. Select **Pin blade to dashboard** for quick access.

The screenshot shows the pinned tag blade titled "Environment : Production" with a red star icon indicating it is pinned. This blade is displayed on the Microsoft Azure dashboard.

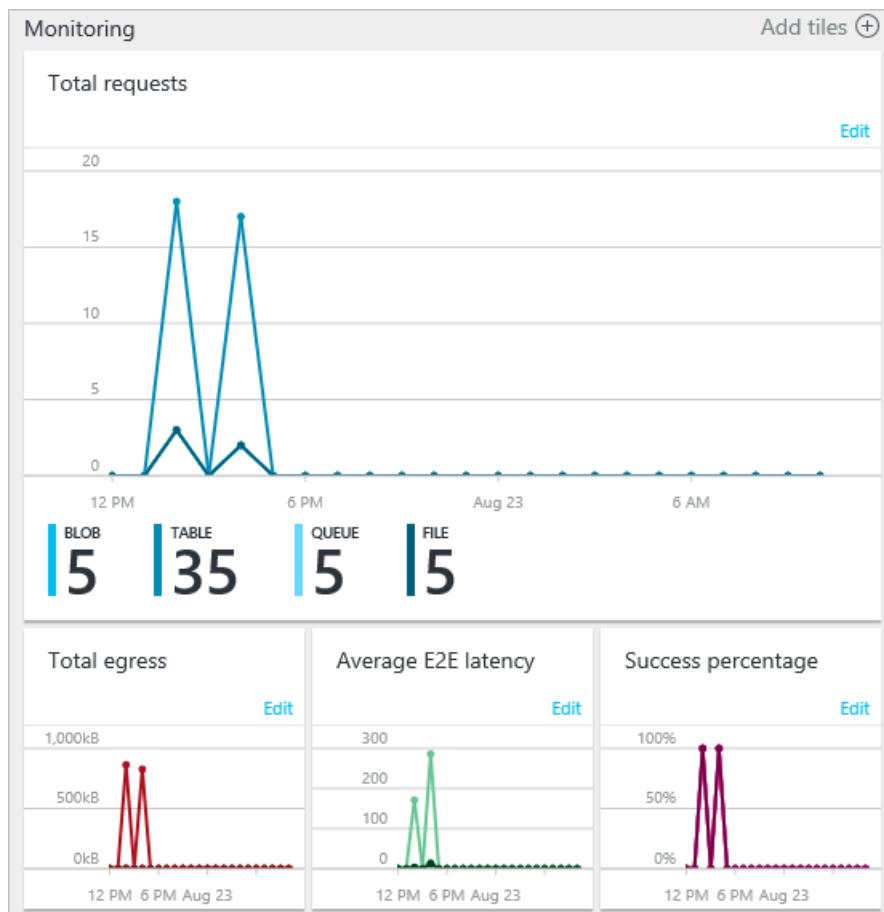
8. You can select the pinned tag from the dashboard to see the resources with that tag.

The screenshot shows the Microsoft Azure dashboard. On the left is a navigation menu with options like "New", "All resources", "Resource groups", "App Services", and "SQL databases". The main dashboard area features a pinned tag blade for "Environment : Production" which is highlighted with a red border. To the right of the blade is a "Service health" section with a map of the Americas showing green dots and a "MY RESOURCES" section with a globe icon.

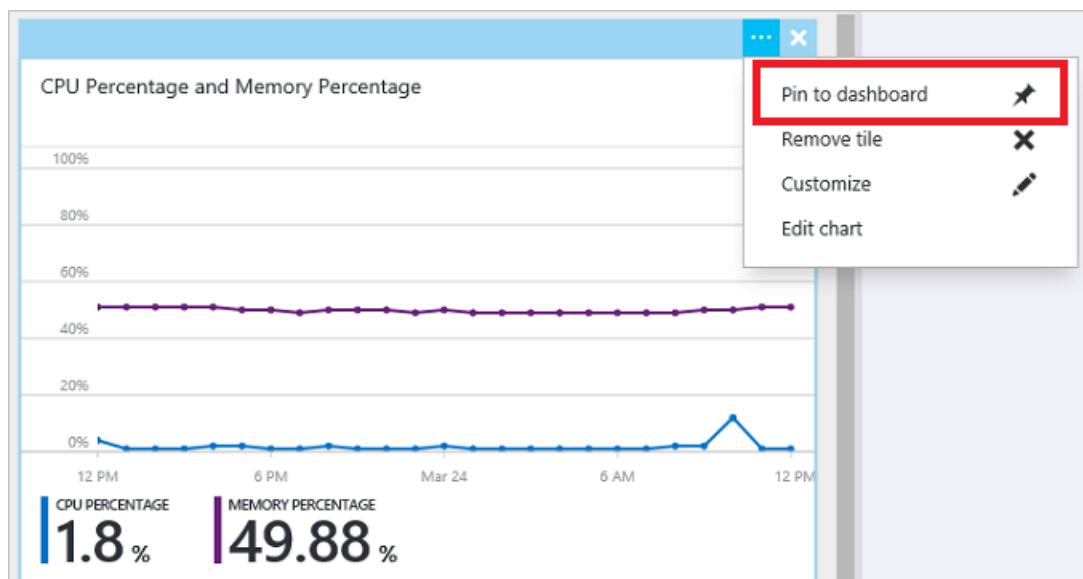
Monitor resources

When you select a resource, the resource blade presents default graphs and tables for monitoring that resource type.

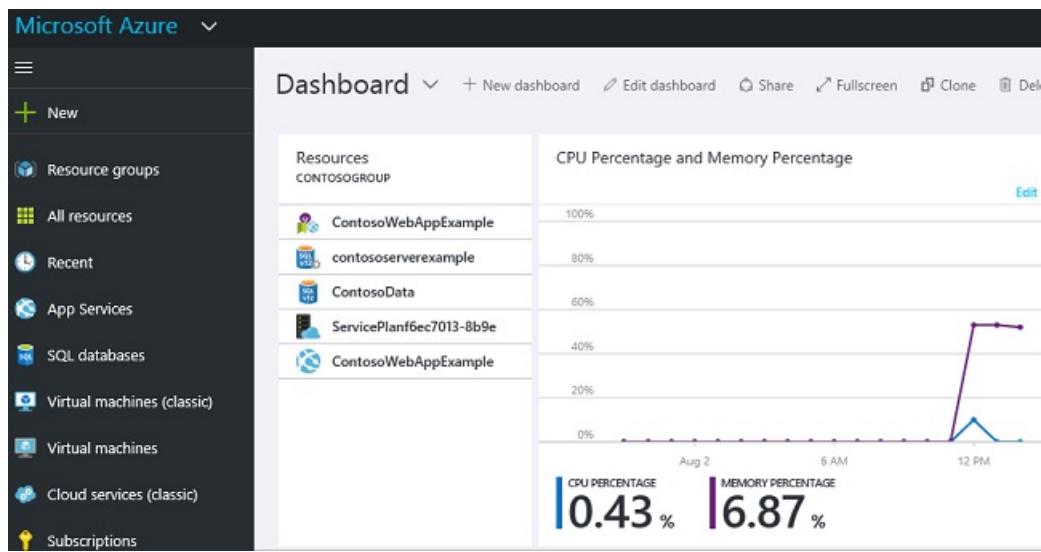
1. Select a resource and notice the **Monitoring** section. It includes graphs that are relevant to the resource type. The following image shows the default monitoring data for a storage account.



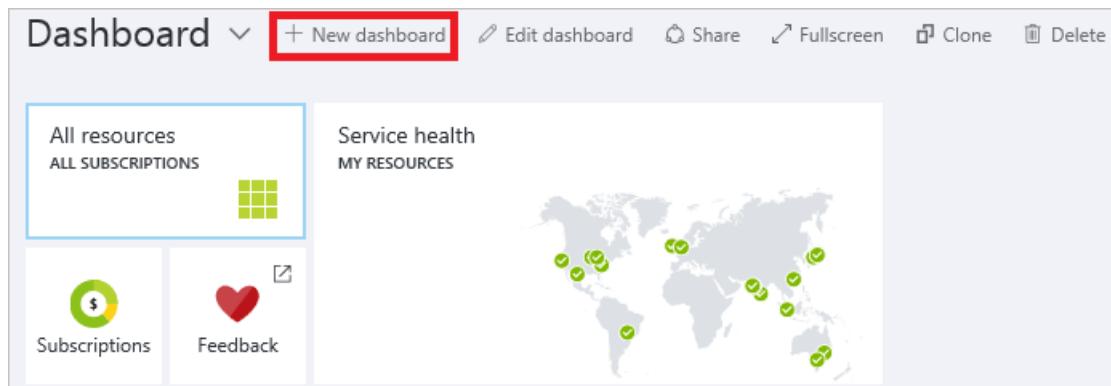
2. You can pin a section of the blade to your dashboard by selecting the ellipsis (...) above the section. You can also customize the size the section in the blade or remove it completely. The following image shows how to pin, customize, or remove the CPU and Memory section.



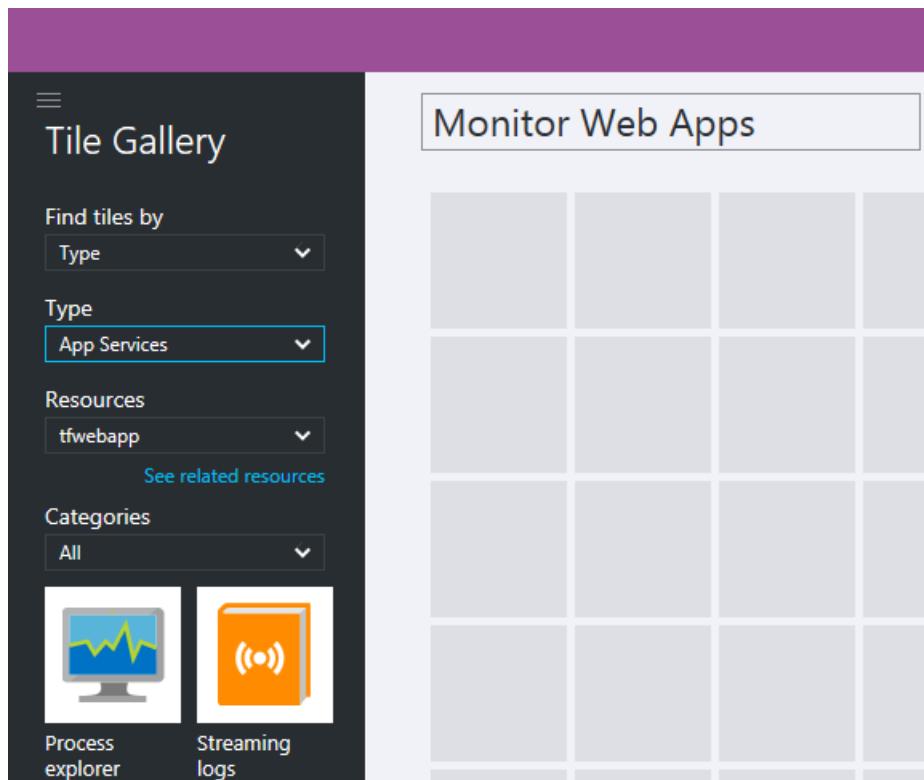
3. After pinning the section to the dashboard, you will see the summary on the dashboard. And, selecting it immediately takes you to more details about the data.



- To completely customize the data you monitor through the portal, navigate to your default dashboard, and select **New dashboard**.



- Give your new dashboard a name and drag tiles onto the dashboard. The tiles are filtered by different options.



To learn about working with dashboards, see [Creating and sharing dashboards in the Azure portal](#).

Manage resources

In the blade for a resource, you see the options for managing the resource. The portal presents management options for that particular resource type. You see the management commands across the top of the resource blade and on the left side.

The screenshot shows the Azure portal interface for managing a virtual machine named 'tfvm'. The top navigation bar includes 'Connect', 'Start', 'Restart', 'Stop', and 'Delete' buttons, all highlighted with a red box. On the left, a sidebar lists management options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Availability set, Disks, Extensions, Network interfaces, Size, and Properties, all of which are also highlighted with a red box. The main content area is divided into sections: 'Essentials' (Resource group: testrg1, Status: Running, Location: West US, Subscription name: Windows Azure MSDN - Visual Studio Ul...), 'Monitoring' (CPU percentage chart showing a single spike at 100%), and 'SETTINGS' (Availability set, Disks, Extensions, Network interfaces, Size, Properties).

From these options, you can perform operations such as starting and stopping a virtual machine, or reconfiguring the properties of the virtual machine.

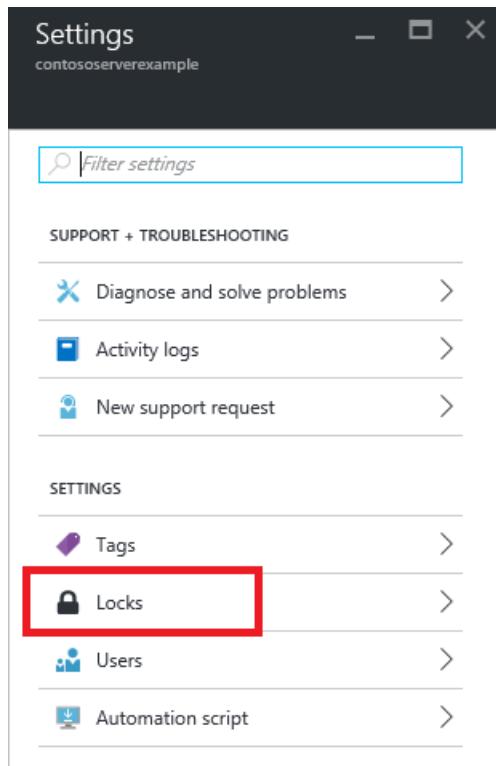
Move resources

If you need to move resources to another resource group or another subscription, see [Move resources to new resource group or subscription](#).

Lock resources

You can lock a subscription, resource group, or resource to prevent other users in your organization from accidentally deleting or modifying critical resources. For more information, see [Lock resources with Azure Resource Manager](#).

1. In the Settings blade for the resource, resource group, or subscription that you wish to lock, select **Locks**.



2. To add a lock, select **Add**. If you want to create a lock at a parent level, select the parent. The currently selected resource inherits the lock from the parent. For example, you could lock the resource group to apply a lock to all its resources.

A screenshot of the 'Management locks' page for the same resource group. The top navigation bar includes 'Add', 'Resource group', 'Subscription', and 'Refresh' buttons. The 'Add' button is highlighted with a red box. The main content area shows a table with columns: LOCK NAME, LOCK TYPE, SCOPE, and NOTES. A message below the table states 'This resource has no locks.'

3. Give the lock a name and lock level. Optionally, you can add notes that describe the lock.

A screenshot of the 'Add lock' dialog box. It contains fields for 'Lock name' (set to 'DatabaseServerLock'), 'Lock type' (set to 'Delete'), and 'Notes' (containing the text 'Prevent deleting the database server'). At the bottom are 'OK' and 'Cancel' buttons.

4. To delete the lock, select the ellipsis and **Delete** from the available options.

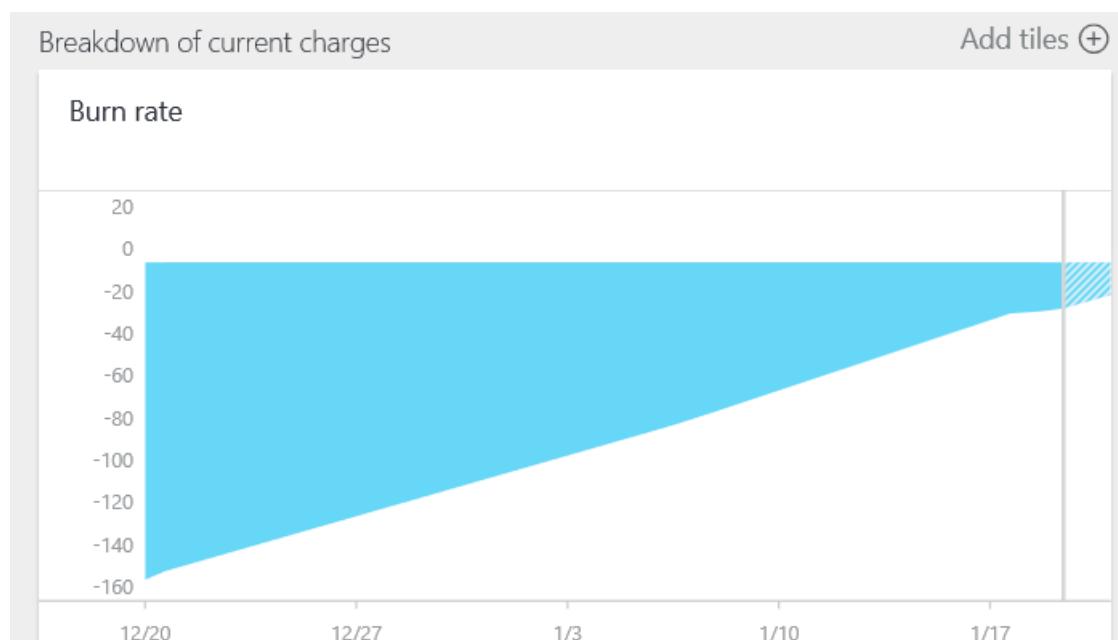
The screenshot shows the 'Management locks' blade in the Azure portal. At the top, there's a header with the title 'Management locks' and a subtitle 'contososerverexample'. Below the header are buttons for '+ Add', 'Resource group', 'Subscription', and 'Refresh'. The main area has a table with columns: 'LOCK NAME', 'LOCK TYPE', 'SCOPE', and 'NOTES'. A single row is shown: 'DatabaseS...', 'Delete', 'This resource', and 'Prevent deleting the database server'. The 'NOTES' column ends with a blue '...' button, which is highlighted with a red box.

View your subscription and costs

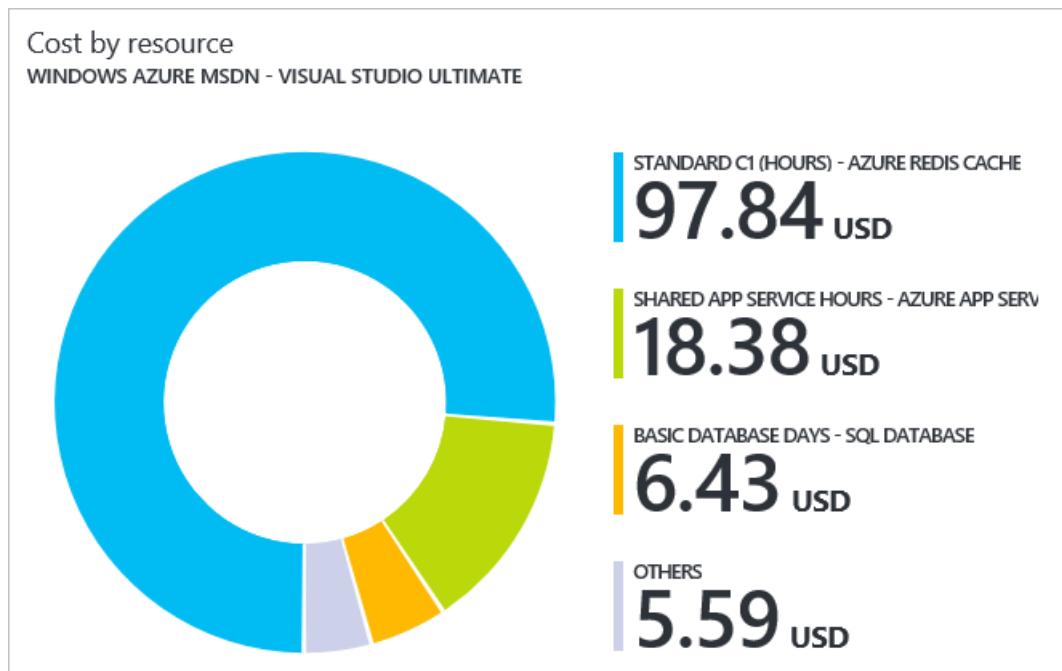
You can view information about your subscription and the rolled-up costs for all your resources. Select **Subscriptions** and the subscription you want to see. You might only have one subscription to select.

The screenshot shows the 'Subscriptions' blade in the Azure portal. On the left, there's a sidebar with various links: 'New', 'Resource groups', 'All resources', 'Recent', 'App Services', 'SQL databases', 'Virtual machines (classic)', 'Virtual machines', 'Cloud services (classic)', 'Subscriptions' (which is highlighted with a red box), and 'App Service plans'. The main area is titled 'Subscriptions' and shows a list of subscriptions. One subscription, 'Windows Azure MSDN - Visual Studio Ultim...', is highlighted with a red box. The list includes columns for 'SUBSCRIPTION' and 'SUBSCRIPTION ID'.

Within the subscription blade, you see a burn rate.



And, a breakdown of costs by resource type.



Export template

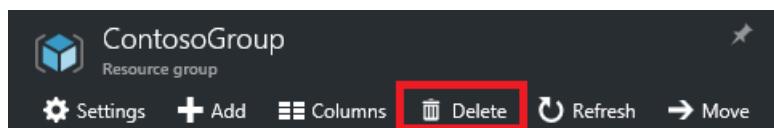
After setting up your resource group, you may want to view the Resource Manager template for the resource group. Exporting the template offers two benefits:

1. You can easily automate future deployments of the solution because the template contains all the complete infrastructure.
2. You can become familiar with template syntax by looking at the JavaScript Object Notation (JSON) that represents your solution.

For step-by-step guidance, see [Export Azure Resource Manager template from existing resources](#).

Delete resource group or resources

Deleting a resource group deletes all the resources contained within it. You can also delete individual resources within a resource group. You want to exercise caution when you delete a resource group because there might be resources in other resource groups that are linked to it. Resource Manager does not delete linked resources, but they may not operate correctly without the expected resources.



Next Steps

- To view activity logs, see [Audit operations with Resource Manager](#).
- To view details about a deployment, see [View deployment operations](#).
- To deploy resources through the portal, see [Deploy resources with Resource Manager templates and Azure portal](#).
- To manage access to resources, see [Use role assignments to manage access to your Azure subscription resources](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Resource Manager REST APIs

1/17/2017 • 6 min to read • [Edit Online](#)

Behind every call to Azure Resource Manager, behind every deployed template, behind every configured storage account there are one or more calls to the Azure Resource Manager's RESTful API. This topic is devoted to those APIs and how you can call them without using any SDK at all. This approach is useful if you want full control of requests to Azure, or if the SDK for your preferred language is not available or doesn't support the operations you need.

This article does not go through every API that is exposed in Azure, but rather uses some operations as examples of how you connect to them. After you understand the basics, you can read the [Azure Resource Manager REST API Reference](#) to find detailed information on how to use the rest of the APIs.

Authentication

Authentication for Resource Manager is handled by Azure Active Directory (AD). To connect to any API, you first need to authenticate with Azure AD to receive an authentication token that you can pass on to every request. As we are describing a pure call directly to the REST APIs, we assume that you don't want to authenticate by being prompted for a username and password. We also assume you are not using two factor authentication mechanisms. Therefore, we create what is called an Azure AD Application and a service principal that are used to log in. But remember that Azure AD support several authentication procedures and all of them could be used to retrieve that authentication token that we need for subsequent API requests. Follow [Create Azure AD Application and Service Principle](#) for step by step instructions.

Generating an Access Token

Authentication against Azure AD is done by calling out to Azure AD, located at login.microsoftonline.com. To authenticate, you need to have the following information:

- Azure AD Tenant ID (the name of that Azure AD you are using to log in, often the same as your company but not necessary)
- Application ID (taken during the Azure AD application creation step)
- Password (that you selected while creating the Azure AD Application)

In the following HTTP request, make sure to replace "Azure AD Tenant ID", "Application ID" and "Password" with the correct values.

Generic HTTP Request:

```
POST /<Azure AD Tenant ID>/oauth2/token?api-version=1.0 HTTP/1.1 HTTP/1.1
Host: login.microsoftonline.com
Cache-Control: no-cache
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&resource=https%3A%2F%2Fmanagement.core.windows.net%2F&client_id=<Application
ID>&client_secret=<Password>
```

... will (if authentication succeeds) result in a response similar to the following response:

```
{  
  "token_type": "Bearer",  
  "expires_in": "3600",  
  "expires_on": "1448199959",  
  "not_before": "1448196059",  
  "resource": "https://management.core.windows.net/",  
  "access_token": "eyJ0eXAiOiJKV1QiLCJhb...86U3JI_0InPUk_1ZqWvKiEWsayA"  
}
```

(The access_token in the preceding response have been shortened to increase readability)

Generating access token using Bash:

```
curl -X POST -H "Content-Type: application/x-www-form-urlencoded" -d  
"grant_type=client_credentials&resource=https://management.core.windows.net&client_id=<application  
id>&client_secret=<password you selected for authentication>" https://login.microsoftonline.com/<Azure AD  
Tenant ID>/oauth2/token?api-version=1.0
```

Generating access token using PowerShell:

```
Invoke-RestMethod -Uri https://login.microsoftonline.com/<Azure AD Tenant ID>/oauth2/token?api-version=1.0 -  
Method Post  
-Body @{"grant_type" = "client_credentials"; "resource" = "https://management.core.windows.net/"; "client_id"  
= "<application id>"; "client_secret" = "<password you selected for authentication>" }
```

The response contains an access token, information about how long that token is valid, and information about what resource you can use that token for. The access token you received in the previous HTTP call must be passed in for all request to the Resource Manager API. You pass it as a header value named "Authorization" with the value "Bearer YOUR_ACCESS_TOKEN". Notice the space between "Bearer" and your access token.

As you can see from the above HTTP Result, the token is valid for a specific period of time during which you should cache and reuse that same token. Even if it is possible to authenticate against Azure AD for each API call, it would be highly inefficient.

Calling Resource Manager REST APIs

This topic only uses a few APIs to explain the basic usage of the REST operations. For information about all the operations, see [Azure Resource Manager REST APIs](#).

List all subscriptions

One of the simplest operations you can do is to list the available subscriptions that you can access. In the following request, you see how the access token is passed in as a header:

(Replace YOUR_ACCESS_TOKEN with your actual Access Token.)

```
GET /subscriptions?api-version=2015-01-01 HTTP/1.1  
Host: management.azure.com  
Authorization: Bearer YOUR_ACCESS_TOKEN  
Content-Type: application/json
```

... and as a result, you get a list of subscriptions that this service principal is allowed to access

(Subscription IDs have been shortened for readability)

```
{
  "value": [
    {
      "id": "/subscriptions/3a8555...555995",
      "subscriptionId": "3a8555...555995",
      "displayName": "Azure Subscription",
      "state": "Enabled",
      "subscriptionPolicies": {
        "locationPlacementId": "Internal_2014-09-01",
        "quotaId": "Internal_2014-09-01"
      }
    }
  ]
}
```

List all resource groups in a specific subscription

All resources available with the Resource Manager APIs are nested inside a resource group. You can query Resource Manager for existing resource groups in your subscription using the following HTTP GET request. Notice how the subscription ID is passed in as part of the URL this time.

(Replace YOUR_ACCESS_TOKEN and SUBSCRIPTION_ID with your actual access token and subscription ID)

```
GET /subscriptions/SUBSCRIPTION_ID/resourcegroups?api-version=2015-01-01 HTTP/1.1
Host: management.azure.com
Authorization: Bearer YOUR_ACCESS_TOKEN
Content-Type: application/json
```

The response you get depends on whether you have any resource groups defined and if so, how many.

(Subscription IDs have been shortened for readability)

```
{
  "value": [
    {
      "id": "/subscriptions/3a8555...555995/resourceGroups/myfirstresourcegroup",
      "name": "myfirstresourcegroup",
      "location": "eastus",
      "properties": {
        "provisioningState": "Succeeded"
      }
    },
    {
      "id": "/subscriptions/3a8555...555995/resourceGroups/mysecondresourcegroup",
      "name": "mysecondresourcegroup",
      "location": "northeurope",
      "tags": {
        "tagname1": "My first tag"
      },
      "properties": {
        "provisioningState": "Succeeded"
      }
    }
  ]
}
```

Create a resource group

So far, we've only been querying the Resource Manager APIs for information. It's time we create some resources, and let's start by the simplest of them all, a resource group. The following HTTP request creates a resource group in a region/location of your choice, and adds a tag to it.

(Replace YOUR_ACCESS_TOKEN, SUBSCRIPTION_ID, RESOURCE_GROUP_NAME with your actual Access Token, Subscription ID and name of the Resource Group you want to create)

```
PUT /subscriptions/SUBSCRIPTION_ID/resourcegroups/RESOURCE_GROUP_NAME?api-version=2015-01-01 HTTP/1.1
Host: management.azure.com
Authorization: Bearer YOUR_ACCESS_TOKEN
Content-Type: application/json

{
  "location": "northeurope",
  "tags": {
    "tagname1": "test-tag"
  }
}
```

If successful, you get a response that is similar to the following response:

```
{
  "id": "/subscriptions/3a8555...555995/resourceGroups/RESOURCE_GROUP_NAME",
  "name": "RESOURCE_GROUP_NAME",
  "location": "northeurope",
  "tags": {
    "tagname1": "test-tag"
  },
  "properties": {
    "provisioningState": "Succeeded"
  }
}
```

You've successfully created a resource group in Azure. Congratulations!

Deploy resources to a resource group using a Resource Manager Template

With Resource Manager, you can deploy your resources using templates. A template defines several resources and their dependencies. For this section, we assume you are familiar with Resource Manager templates, and we just show you how to make the API call to start deployment. For more information about constructing templates, see [Authoring Azure Resource Manager templates](#).

Deployment of a template doesn't differ much to how you call other APIs. One important aspect is that deployment of a template can take quite a long time. The API call just returns, and it's up to you as developer to query for status of the deployment to find out when the deployment is done. For more information, see [Track asynchronous Azure operations](#).

For this example, we use a publicly exposed template available on [GitHub](#). The template we use deploys a Linux VM to the West US region. Even though this example uses a template available in a public repository like GitHub, you can instead pass the full template as part of the request. Note that we provide parameter values in the request that are used inside the deployed template.

(Replace SUBSCRIPTION_ID, RESOURCE_GROUP_NAME, DEPLOYMENT_NAME, YOUR_ACCESS_TOKEN, GLOBALY_UNIQUE_STORAGE_ACCOUNT_NAME, ADMIN_USER_NAME, ADMIN_PASSWORD and DNS_NAME_FOR_PUBLIC_IP to values appropriate for your request)

```
PUT  
/subscriptions/SUBSCRIPTION_ID/resourcegroups/RESOURCE_GROUP_NAME/providers/microsoft.resources/deployments/DE  
PLOYMENT_NAME?api-version=2015-01-01 HTTP/1.1  
Host: management.azure.com  
Authorization: Bearer YOUR_ACCESS_TOKEN  
Content-Type: application/json  
  
{  
  "properties": {  
    "templateLink": {  
      "uri": "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-simple-linux-  
vm/azuredeploy.json",  
      "contentVersion": "1.0.0.0",  
    },  
    "mode": "Incremental",  
    "parameters": {  
      "newStorageAccountName": {  
        "value": "GLOBALY_UNIQUE_STORAGE_ACCOUNT_NAME"  
      },  
      "adminUsername": {  
        "value": "ADMIN_USER_NAME"  
      },  
      "adminPassword": {  
        "value": "ADMIN_PASSWORD"  
      },  
      "dnsNameForPublicIP": {  
        "value": "DNS_NAME_FOR_PUBLIC_IP"  
      },  
      "ubuntuOSVersion": {  
        "value": "15.04"  
      }  
    }  
  }  
}
```

The long JSON response for this request has been omitted to improve readability of this documentation. The response contains information about the templated deployment that you created.

Next steps

- To learn about handling asynchronous REST operations, see [Track asynchronous Azure operations](#).

Use tags to organize your Azure resources

2/21/2017 • 9 min to read • [Edit Online](#)

You apply tags to your Azure resources to logically organize them by categories. Each tag consists of a key and a value. For example, you can apply the key "Environment" and the value "Production" to all the resources in production. Without this tag, you may have difficulty identifying whether a resource is intended for development, test, or production. However, "Environment" and "Production" are just examples. You define the keys and values that make the most sense for organizing your subscription.

After applying tags, you can retrieve all the resources in your subscription with that tag key and value. Tags enable you to retrieve related resources that reside in different resource groups. This approach is helpful when you need to organize resources for billing or management.

The following limitations apply to tags:

- Each resource or resource group can have a maximum of 15 tags.
- The tag name is limited to 512 characters.
- The tag value is limited to 256 characters.
- Tags applied to the resource group are not inherited by the resources in that resource group.

NOTE

You can only apply tags to resources that support Resource Manager operations. If you created a Virtual Machine, Virtual Network, or Storage through the classic deployment model (such as through the classic portal), you cannot apply a tag to that resource. To support tagging, redeploy these resources through Resource Manager. All other resources support tagging.

Ensure tag consistency with policies

Resource policies enable you to create standard rules for your organization. You can create policies that ensure resources are tagged with the appropriate values. For more information, see [Apply resource policies for tags](#).

Templates

To tag a resource during deployment, add the `tags` element to the resource you are deploying. Provide the tag name and value.

Apply literal value to tag name

The following example shows a storage account with two tags (`Dept` and `Environment`) that are set to literal values:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "resources": [
    {
      "apiVersion": "2016-01-01",
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[concat('storage', uniqueString(resourceGroup().id))]",
      "location": "[resourceGroup().location]",
      "tags": {
        "Dept": "Finance",
        "Environment": "Production"
      },
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "Storage",
      "properties": { }
    }
  ]
}
```

Apply object to tag element

You can define an object parameter that stores several tags, and apply that object to the tag element. Each property in the object becomes a separate tag for the resource. The following example has a parameter named `tagValues` that is applied to the tag element.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "tagValues": {
      "type": "object",
      "defaultValue": {
        "Dept": "Finance",
        "Environment": "Production"
      }
    }
  },
  "resources": [
    {
      "apiVersion": "2016-01-01",
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[concat('storage', uniqueString(resourceGroup().id))]",
      "location": "[resourceGroup().location]",
      "tags": "[parameters('tagValues')]",
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "Storage",
      "properties": { }
    }
  ]
}
```

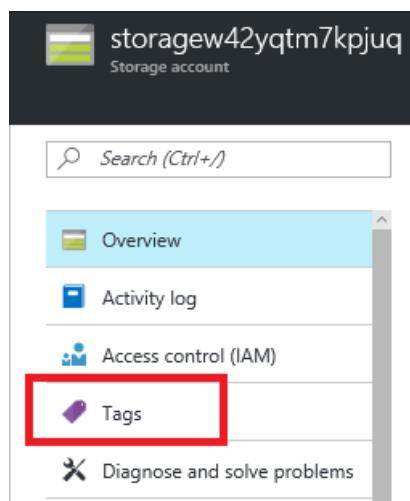
Apply JSON string to tag name

To store many values in a single tag, apply a JSON string that represents the values. The entire JSON string is stored as one tag that cannot exceed 256 characters. The following example has a single tag named `CostCenter` that contains several values from a JSON string:

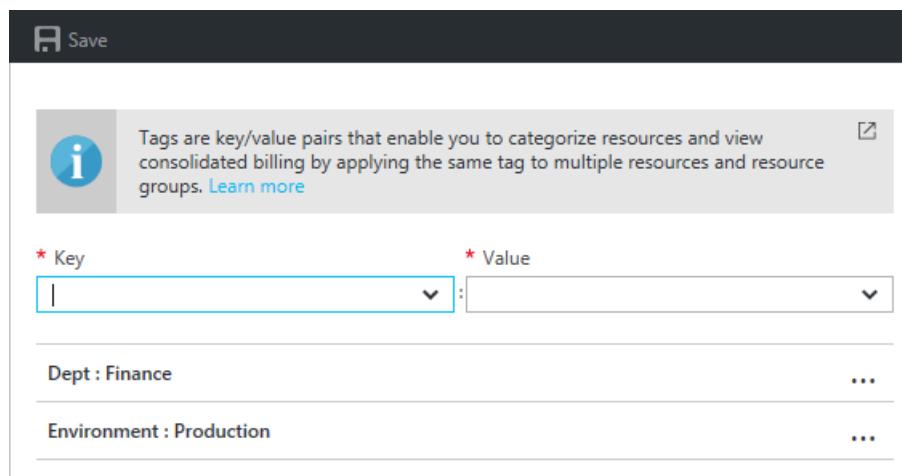
```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "resources": [
    {
      "apiVersion": "2016-01-01",
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[concat('storage', uniqueString(resourceGroup().id))]",
      "location": "[resourceGroup().location]",
      "tags": {
        "CostCenter": "{\"Dept\":\"Finance\", \"Environment\":\"Production\"}"
      },
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "Storage",
      "properties": { }
    }
  ]
}
```

Portal

- To view the tags for a resource or resource group, select the **Tags** icon.



- You see the existing tags for the resource. If you have not previously applied tags, the list is empty.



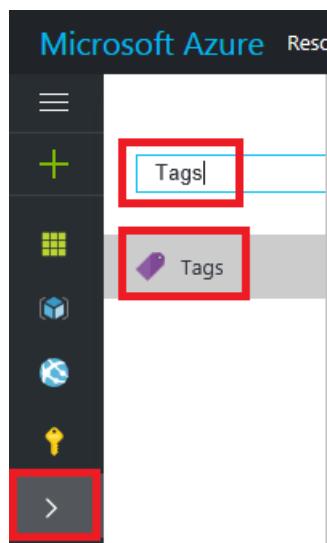
- To add a tag, type a key and value, or select an existing one from the dropdown menu. Select **Save**.

Save

Tags are key/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. [Learn more](#)

* Key	* Value
CostCenter	: IT
Dept : Finance	...
Environment : Production	...

- To view all the resources with a tag value, select > (More services), and enter the word **Tags** into the filter text box. Select **Tags** from the available options.



- You see a summary of the tags in your subscriptions.

Tags

Subscriptions: All 5 selected

All subscriptions

Tags are key/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. [Learn more](#)

CostCenter : IT	...
Dept : Finance	...
Environment : Production	...

- Select any of the tags to display the resources and resource groups with that tag.

Environment : Production

Tag

Subscriptions: All 5 selected

Filter items... All subscriptions

NAME	SUBSCRIPTION
TagTestGroup	
storage2w42yqtm7kpjuq	
storage3w42yqtm7kpjuq	
storage4w42yqtm7kpjuq	

7. Select **Pin blade to dashboard** for quick access.

Environment : Production

Tag

8. You can select the pinned tag from the dashboard to see the resources with that tag.

Microsoft Azure

Dashboard + New dashboard Edit dashboard

All resources

Resource groups

App Services

SQL databases

Service health MY RESOURCES

PowerShell

Version 3.0 of the AzureRm.Resources module included significant changes in how you work with tags. Before proceeding, check your version:

```
Get-Module -ListAvailable -Name AzureRm.Resources | Select Version
```

If your results show version 3.0 or later, the examples in this topic work with your environment. If you do not have version 3.0 or later, [update your version](#) by using PowerShell Gallery or Web Platform Installer before proceeding with this topic.

Version

3.5.0

Every time you apply tags to a resource or resource group, you overwrite the existing tags on that resource or resource group. Therefore, you must use a different approach based on whether the resource or resource group has existing tags that you want to preserve. To add tags to a:

- resource group without existing tags.

```
Set-AzureRmResourceGroup -Name TagTestGroup -Tag @{ Dept="IT"; Environment="Test" }
```

- resource group with existing tags.

```
$tags = (Get-AzureRmResourceGroup -Name TagTestGroup).Tags
/tags += @{Status="Approved"}
Set-AzureRmResourceGroup -Tag $tags -Name TagTestGroup
```

- resource without existing tags.

```
Set-AzureRmResource -Tag @{ Dept="IT"; Environment="Test" } -ResourceName storageexample -
ResourceGroupName TagTestGroup -ResourceType Microsoft.Storage/storageAccounts
```

- resource with existing tags.

```
$tags = (Get-AzureRmResource -ResourceName storageexample -ResourceGroupName TagTestGroup).Tags
$tags += @{Status="Approved"}
Set-AzureRmResource -Tag $tags -ResourceName storageexample -ResourceGroupName TagTestGroup -
ResourceType Microsoft.Storage/storageAccounts
```

To apply all tags from a resource group to its resources, and **not retain existing tags on the resources**, use the following script:

```
$groups = Get-AzureRmResourceGroup
foreach ($g in $groups)
{
    Find-AzureRmResource -ResourceGroupNameEquals $g.ResourceGroupName | ForEach-Object {Set-AzureRmResource
-ResourceId $_.ResourceId -Tag $g.Tags -Force }
}
```

To apply all tags from a resource group to its resources, and **retain existing tags on resources that are not duplicates**, use the following script:

```
$groups = Get-AzureRmResourceGroup
foreach ($g in $groups)
{
    if ($g.Tags -ne $null) {
        $resources = Find-AzureRmResource -ResourceGroupNameEquals $g.ResourceGroupName
        foreach ($r in $resources)
        {
            $resourcetags = (Get-AzureRmResource -ResourceId $r.ResourceId).Tags
            foreach ($key in $g.Tags.Keys)
            {
                if ($resourcetags.ContainsKey($key)) { $resourcetags.Remove($key) }
            }
            $resourcetags += $g.Tags
            Set-AzureRmResource -Tag $resourcetags -ResourceId $r.ResourceId -Force
        }
    }
}
```

To remove all tags, pass an empty hash table.

```
Set-AzureRmResourceGroup -Tag @{} -Name TagTestGgroup
```

To get resource groups with a specific tag, use `Find-AzureRmResourceGroup` cmdlet.

```
(Find-AzureRmResourceGroup -Tag @{ Dept="Finance" }).Name
```

To get all the resources with a particular tag and value, use the `Find-AzureRmResource` cmdlet.

```
(Find-AzureRmResource -TagName Dept -TagValue Finance).Name
```

Azure CLI 2.0

With Azure CLI 2.0, you can add tags to resources and resource group, and query resources by tag values.

Every time you apply tags to a resource or resource group, you overwrite the existing tags on that resource or resource group. Therefore, you must use a different approach based on whether the resource or resource group has existing tags that you want to preserve. To add tags to a:

- resource group without existing tags.

```
az group update -n TagTestGroup --set tags.Environment=Test tags.Dept=IT
```

- resource without existing tags.

```
az resource tag --tags Dept=IT Environment=Test -g TagTestGroup -n storageexample --resource-type "Microsoft.Storage/storageAccounts"
```

To add tags to a resource that already has tags, first retrieve the existing tags:

```
az resource show --query tags --output list -g TagTestGroup -n storageexample --resource-type "Microsoft.Storage/storageAccounts"
```

Which returns the following format:

```
Dept      : Finance
Environment : Test
```

Reapply the existing tags to the resource, and add the new tags.

```
az resource tag --tags Dept=Finance Environment=Test CostCenter=IT -g TagTestGroup -n storageexample --resource-type "Microsoft.Storage/storageAccounts"
```

To get resource groups with a specific tag, use `az group list`.

```
az group list --tag Dept=IT
```

To get all the resources with a particular tag and value, use `az resource list`.

```
az resource list --tag Dept=Finance
```

Azure CLI 1.0

To add a tag to a resource group, use **azure group set**. If the resource group does not have any existing tags, pass in the tag.

```
azure group set -n tag-demo-group -t Dept=Finance
```

Tags are updated as a whole. If you want to add a tag to a resource group that has existing tags, pass all the tags.

```
azure group set -n tag-demo-group -t Dept=Finance;Environment=Production;Project=Upgrade
```

Tags are not inherited by resources in a resource group. To add a tag to a resource, use **azure resource set**. Pass the API version number for the resource type that you are adding the tag to. If you need to retrieve the API version, use the following command with the resource provider for the type you are setting:

```
azure provider show -n Microsoft.Storage --json
```

In the results, look for the resource type you want.

```
"resourceTypes": [
{
  "resourceType": "storageAccounts",
  ...
  "apiVersions": [
    "2016-01-01",
    "2015-06-15",
    "2015-05-01-preview"
  ]
}
...
...
```

Now, provide that API version, resource group name, resource name, resource type, and tag value as parameters.

```
azure resource set -g tag-demo-group -n storagetagdemo -r Microsoft.Storage/storageAccounts -t Dept=Finance -o 2016-01-01
```

Tags exist directly on resources and resource groups. To see the existing tags, get a resource group and its resources with **azure group show**.

```
azure group show -n tag-demo-group --json
```

Which returns metadata about the resource group, including any tags applied to it.

```
{  
  "id": "/subscriptions/4705409c-9372-42f0-914c-64a504530837/resourceGroups/tag-demo-group",  
  "name": "tag-demo-group",  
  "properties": {  
    "provisioningState": "Succeeded"  
  },  
  "location": "southcentralus",  
  "tags": {  
    "Dept": "Finance",  
    "Environment": "Production",  
    "Project": "Upgrade"  
  },  
  ...  
}
```

You view the tags for a particular resource by using **azure resource show**.

```
azure resource show -g tag-demo-group -n storagetagdemo -r Microsoft.Storage/storageAccounts -o 2016-01-01 --json
```

To retrieve all the resources with a tag value, use:

```
azure resource list -t Dept=Finance --json
```

To retrieve all the resource groups with a tag value, use:

```
azure group list -t Dept=Finance
```

You can view the existing tags in your subscription with the following command:

```
azure tag list
```

REST API

The portal and PowerShell both use the [Resource Manager REST API](#) behind the scenes. If you need to integrate tagging into another environment, you can get tags with a GET on the resource id and update the set of tags with a PATCH call.

Tags and billing

Tags enable you to group your billing data. For example, if you are running multiple VMs for different organizations, use the tags to group usage by cost center. You can also use tags to categorize costs by runtime environment; such as, the billing usage for VMs running in production environment.

You can retrieve information about tags through the [Azure Resource Usage and RateCard APIs](#) or the usage comma-separated values (CSV) file. You download the usage file from the [Azure accounts portal](#) or [EA portal](#). For more information about programmatic access to billing information, see [Gain insights into your Microsoft Azure resource consumption](#). For REST API operations, see [Azure Billing REST API Reference](#).

When you download the usage CSV for services that support tags with billing, the tags appear in the **Tags** column. For more information, see [Understand your bill for Microsoft Azure](#).

Daily Usage							
Usage Date	Meter Category	Unit	Consume	Resource Group	Instance Id	Tags	
5/14/2015	"Virtual Machines"	"Hours"	3.999984	"computeRG"	"virtualMachines/catalogVM"	<pre>{"costCenter":"finance", "env":"prod"}"</pre>	
5/14/2015	"Virtual Machines"	"Hours"	3.999984	"businessRG"	"virtualMachines/dataVM"	<pre>{"costCenter":"hr", "env":"test"}"</pre>	

Next Steps

- You can apply restrictions and conventions across your subscription with customized policies. The policy you define could require that all resources have a value for a particular tag. For more information, see [Use Policy to manage resources and control access](#).
- For an introduction to using Azure PowerShell when deploying resources, see [Using Azure PowerShell with Azure Resource Manager](#).
- For an introduction to using Azure CLI when deploying resources, see [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management](#).
- For an introduction to using the portal, see [Using the Azure portal to manage your Azure resources](#)
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Move resources to new resource group or subscription

4/12/2017 • 12 min to read • [Edit Online](#)

This topic shows you how to move resources to either a new subscription or a new resource group in the same subscription. You can use the portal, PowerShell, Azure CLI, or the REST API to move resource. The move operations in this topic are available to you without any assistance from Azure support.

When moving resources, both the source group and the target group are locked during the operation. Write and delete operations are blocked on the resource groups until the move completes. This lock means you cannot add, update, or delete resources in the resource groups, but it does not mean the resources are frozen. For example, if you move a SQL Server and its database to a new resource group, an application that uses the database experiences no downtime. It can still read and write to the database.

You cannot change the location of the resource. Moving a resource only moves it to a new resource group. The new resource group may have a different location, but that does not change the location of the resource.

NOTE

This article describes how to move resources within an existing Azure account offering. If you actually want to change your Azure account offering (such as upgrading from pay-as-you-go to pre-pay) while continuing to work with your existing resources, see [Switch your Azure subscription to another offer](#).

Checklist before moving resources

There are some important steps to perform before moving a resource. By verifying these conditions, you can avoid errors.

1. The source and destination subscriptions must exist within the same [Active Directory tenant](#). To check that both subscriptions have the same tenant ID, use Azure PowerShell or Azure CLI.

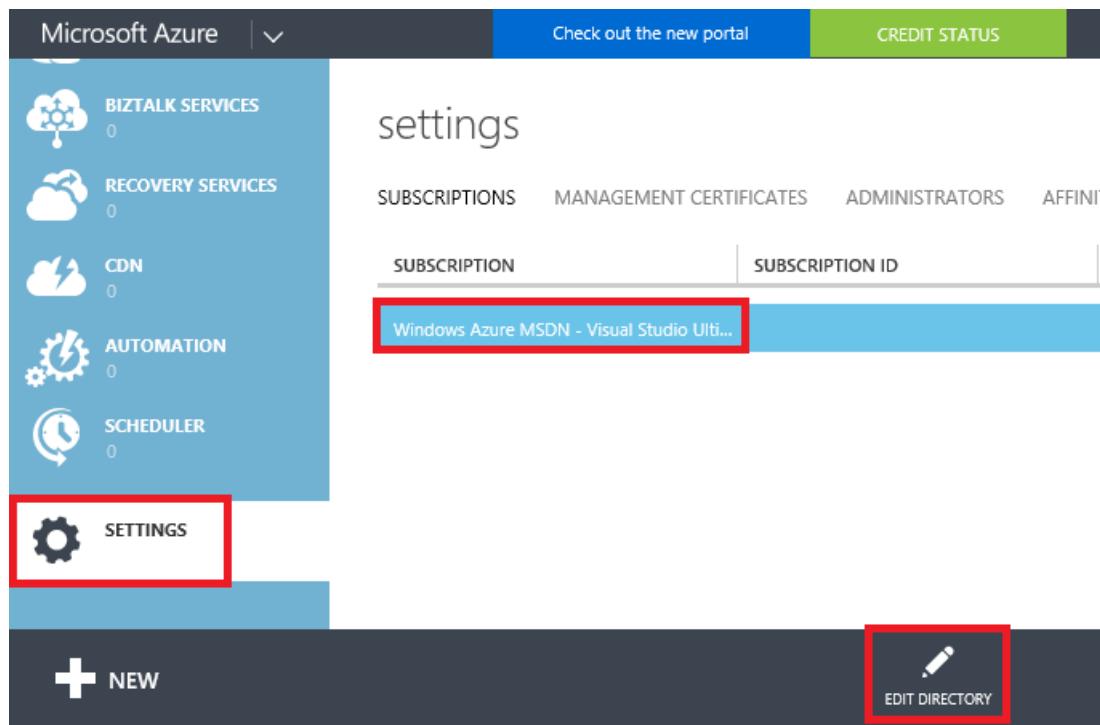
For Azure PowerShell, use:

```
(Get-AzureRmSubscription -SubscriptionName "Example Subscription").TenantId
```

For Azure CLI 2.0, use:

```
az account show --subscription "Example Subscription" --query tenantId
```

If the tenant IDs for the source and destination subscriptions are not the same, you can attempt to change the directory for the subscription. However, this option is only available to Service Administrators who are signed in with a Microsoft account (not an organizational account). To attempt changing the directory, log in to the [classic portal](#), and select **Settings**, and select the subscription. If the **Edit Directory** icon is available, select it to change the associated Active Directory.



If that icon is not available, you must contact support to move the resources to a new tenant.

2. The service must enable the ability to move resources. This topic lists which services enable moving resources and which services do not enable moving resources.
3. The destination subscription must be registered for the resource provider of the resource being moved. If not, you receive an error stating that the **subscription is not registered for a resource type**. You might encounter this problem when moving a resource to a new subscription, but that subscription has never been used with that resource type. To learn how to check the registration status and register resource providers, see [Resource providers and types](#).

When to call support

You can move most resources through the self-service operations shown in this topic. Use the self-service operations to:

- Move Resource Manager resources.
- Move classic resources according to the [classic deployment limitations](#).

Call support when you need to:

- Move your resources to a new Azure account (and Active Directory tenant).
- Move classic resources but are having trouble with the limitations.

Services that enable move

For now, the services that enable moving to both a new resource group and subscription are:

- API Management
- App Service apps (web apps) - see [App Service limitations](#)
- Automation
- Batch
- Bing Maps
- CDN
- Cloud Services - see [Classic deployment limitations](#)

- Cognitive Services
- Content Moderator
- Data Catalog
- Data Factory
- Data Lake Analytics
- Data Lake Store
- DNS
- DocumentDB
- Event Hubs
- HDInsight clusters - see [HDInsight limitations](#)
- IoT Hubs
- Key Vault
- Load Balancers
- Logic Apps
- Machine Learning
- Media Services
- Mobile Engagement
- Notification Hubs
- Operational Insights
- Operations Management
- Power BI
- Redis Cache
- Scheduler
- Search
- Server Management
- Service Bus
- Service Fabric
- Storage
- Storage (classic) - see [Classic deployment limitations](#)
- Stream Analytics
- SQL Database server - The database and server must reside in the same resource group. When you move a SQL server, all its databases are also moved.
- Traffic Manager
- Virtual Machines - Does not support move to a new subscription when its certificates are stored in a Key Vault
- Virtual Machines (classic) - see [Classic deployment limitations](#)
- Virtual Machine Scale Sets
- Virtual Networks - Currently, a peered Virtual Network cannot be moved until VNet peering has been disabled. Once disabled, the Virtual Network can be moved successfully and the VNet peering can be enabled.
- VPN Gateway

Services that do not enable move

The services that currently do not enable moving a resource are:

- AD Hybrid Health Service
- Application Gateway
- Application Insights
- BizTalk Services

- Container Service
- Express Route
- DevTest Labs - Move to new resource group in same subscription is enabled, but cross subscription move is not enabled.
- Dynamics LCS
- Recovery Services vault - also do not move the Compute, Network, and Storage resources associated with the Recovery Services vault, see [Recovery Services limitations](#).
- Security
- Virtual Machines with certificate stored in Key Vault
- Virtual Machines with Managed Disks
- Availability sets with Virtual Machines with Managed Disks
- Managed Disks
- Images created from Managed Disks
- Snapshots created from Managed Disks
- Virtual Networks (classic) - see [Classic deployment limitations](#)
- Virtual Machines created from Marketplace resources - cannot be moved across subscriptions. Resource needs to be deprovisioned in the current subscription and deployed again in the new subscription

App Service limitations

When working with App Service apps, you cannot move only an App Service plan. To move App Service apps, your options are:

- Move the App Service plan and all other App Service resources in that resource group to a new resource group that does not already have App Service resources. This requirement means you must move even the App Service resources that are not associated with the App Service plan.
- Move the apps to a different resource group, but keep all App Service plans in the original resource group.

If your original resource group also includes an Application Insights resource, you cannot move that resource because Application Insights does not currently enable the move operation. If you include the Application Insights resource when moving App Service apps, the entire move operation fails. However, the Application Insights and App Service plan do not need to reside in the same resource group as the app for the app to function correctly.

For example, if your resource group contains:

- **web-a** which is associated with **plan-a** and **app-insights-a**
- **web-b** which is associated with **plan-b** and **app-insights-b**

Your options are:

- Move **web-a**, **plan-a**, **web-b**, and **plan-b**
- Move **web-a** and **web-b**
- Move **web-a**
- Move **web-b**

All other combinations involve either moving a resource type that can't move (Application Insights) or leaving behind a resource type that can't be left behind when moving an App Service plan (any type of App Service resource).

If your web app resides in a different resource group than its App Service plan but you want to move both to a new resource group, you must perform the move in two steps. For example:

- **web-a** resides in **web-group**
- **plan-a** resides in **plan-group**

- You want **web-a** and **plan-a** to reside in **combined-group**

To accomplish this move, perform two separate move operations in the following sequence:

1. Move the **web-a** to **plan-group**
2. Move **web-a** and **plan-a** to **combined-group**.

You can move an App Service Certificate to a new resource group or subscription without any issues. However, if your web app includes an SSL certificate that you purchased externally and uploaded to the app, you must delete the certificate before moving the web app. For example, you can perform the following steps:

1. Delete the uploaded certificate from the web app
2. Move the web app
3. Upload the certificate to the web app

Recovery Services limitations

Move is not enabled for Storage, Network, or Compute resources used to set up disaster recovery with Azure Site Recovery.

For example, suppose you have set up replication of your on-premises machines to a storage account (Storage1) and want the protected machine to come up after failover to Azure as a virtual machine (VM1) attached to a virtual network (Network1). You cannot move any of these Azure resources - Storage1, VM1, and Network1 - across resource groups within the same subscription or across subscriptions.

HDInsight limitations

You can move HDInsight clusters to a new subscription or resource group. However, you cannot move across subscriptions the networking resources linked to the HDInsight cluster (such as the virtual network, NIC, or load balancer). In addition, you cannot move to a new resource group a NIC that is attached to a virtual machine for the cluster.

When moving an HDInsight cluster to a new subscription, first move other resources (like the storage account). Then, move the HDInsight cluster by itself.

Classic deployment limitations

The options for moving resources deployed through the classic model differ based on whether you are moving the resources within a subscription or to a new subscription.

Same subscription

When moving resources from one resource group to another resource group within the same subscription, the following restrictions apply:

- Virtual networks (classic) cannot be moved.
- Virtual machines (classic) must be moved with the cloud service.
- Cloud service can only be moved when the move includes all its virtual machines.
- Only one cloud service can be moved at a time.
- Only one storage account (classic) can be moved at a time.
- Storage account (classic) cannot be moved in the same operation with a virtual machine or a cloud service.

To move classic resources to a new resource group within the same subscription, use the standard move operations through the [portal](#), [Azure PowerShell](#), [Azure CLI](#), or [REST API](#). You use the same operations as you use for moving Resource Manager resources.

New subscription

When moving resources to a new subscription, the following restrictions apply:

- All classic resources in the subscription must be moved in the same operation.
- The target subscription must not contain any other classic resources.
- The move can only be requested through a separate REST API for classic moves. The standard Resource Manager move commands do not work when moving classic resources to a new subscription.

To move classic resources to a new subscription, use either the portal or REST operations that are specific to classic resources. For information about moving classic resources through the portal, see [Use portal](#). To use REST, perform the following steps:

1. Check if the source subscription can participate in a cross-subscription move. Use the following operation:

```
POST  
https://management.azure.com/subscriptions/{sourceSubscriptionId}/providers/Microsoft.ClassicCompute/va  
lidateSubscriptionMoveAvailability?api-version=2016-04-01
```

In the request body, include:

```
{  
  "role": "source"  
}
```

The response for the validation operation is in the following format:

```
{  
  "status": "{status}",  
  "reasons": [  
    "reason1",  
    "reason2"  
  ]  
}
```

2. Check if the destination subscription can participate in a cross-subscription move. Use the following operation:

```
POST  
https://management.azure.com/subscriptions/{destinationSubscriptionId}/providers/Microsoft.ClassicCompu  
te/validateSubscriptionMoveAvailability?api-version=2016-04-01
```

In the request body, include:

```
{  
  "role": "target"  
}
```

The response is in the same format as the source subscription validation.

3. If both subscriptions pass validation, move all classic resources from one subscription to another subscription with the following operation:

```
POST https://management.azure.com/subscriptions/{subscription-  
id}/providers/Microsoft.ClassicCompute/moveSubscriptionResources?api-version=2016-04-01
```

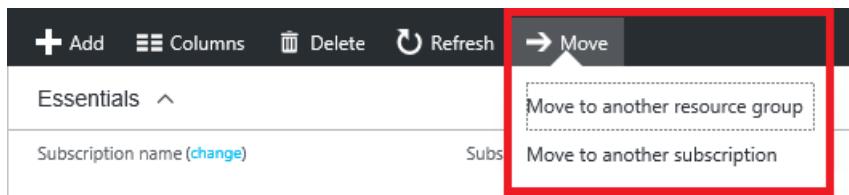
In the request body, include:

```
{  
  "target": "/subscriptions/{target-subscription-id}"  
}
```

The operation may run for several minutes.

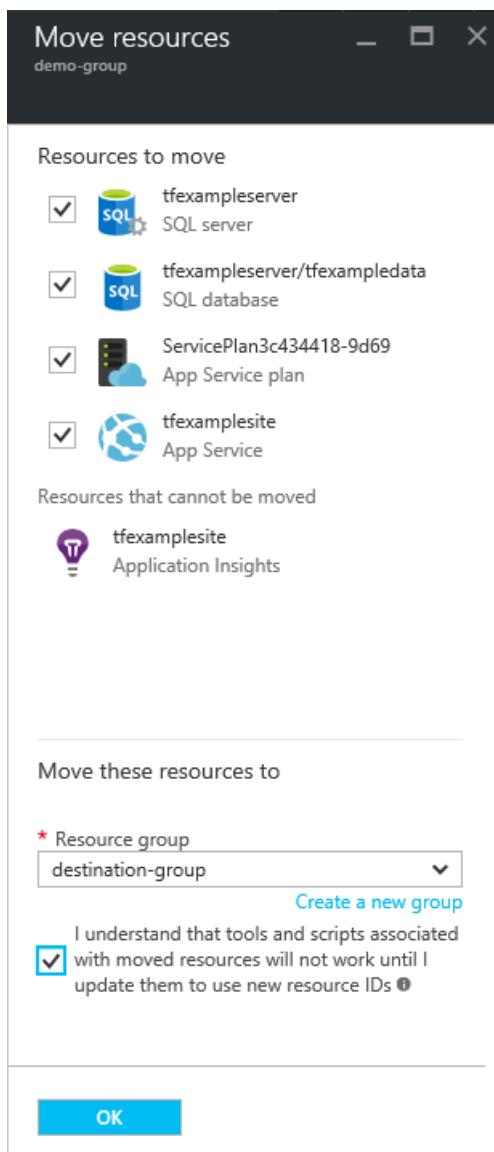
Use portal

To move resources, select the resource group containing those resources, and then select the **Move** button.



Select whether you are moving the resources to a new resource group or a new subscription.

Select the resources to move and the destination resource group. Acknowledge that you need to update scripts for these resources and select **OK**. If you selected the edit subscription icon in the previous step, you must also select the destination subscription.



In **Notifications**, you see that the move operation is running.



When it has completed, you are notified of the result.



Use PowerShell

To move existing resources to another resource group or subscription, use the `Move-AzureRmResource` command.

The first example shows how to move one resource to a new resource group.

```
$resource = Get-AzureRmResource -ResourceName ExampleApp -ResourceGroupName OldRG  
Move-AzureRmResource -DestinationResourceGroupName NewRG -ResourceId $resource.ResourceId
```

The second example shows how to move multiple resources to a new resource group.

```
$webapp = Get-AzureRmResource -ResourceGroupName OldRG -ResourceName ExampleSite  
$plan = Get-AzureRmResource -ResourceGroupName OldRG -ResourceName ExamplePlan  
Move-AzureRmResource -DestinationResourceGroupName NewRG -ResourceId $webapp.ResourceId, $plan.ResourceId
```

To move to a new subscription, include a value for the `DestinationSubscriptionId` parameter.

You are asked to confirm that you want to move the specified resources.

```
Confirm  
Are you sure you want to move these resources to the resource group  
'/subscriptions/{guid}/resourceGroups/newRG' the resources:  
  
'/subscriptions/{guid}/resourceGroups/destinationgroup/providers/Microsoft.Web/serverFarms/exampleplan'  
'/subscriptions/{guid}/resourceGroups/destinationgroup/providers/Microsoft.Web/sites/examplesite'  
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y
```

Use Azure CLI 2.0

To move existing resources to another resource group or subscription, use the `az resource move` command.

Provide the resource IDs of the resources to move. You can get resource IDs with the following command:

```
az resource show -g sourceGroup -n storagedemo --resource-type "Microsoft.Storage/storageAccounts" --query id
```

The following example shows how to move a storage account to a new resource group. In the `--ids` parameter, provide a space-separated list of the resource IDs to move.

```
az resource move --destination-group newgroup --ids  
"/subscriptions/{guid}/resourceGroups/sourceGroup/providers/Microsoft.Storage/storageAccounts/storagedemo"
```

To move to a new subscription, provide the `--destination-subscription-id` parameter.

Use Azure CLI 1.0

To move existing resources to another resource group or subscription, use the `azure resource move` command.

Provide the resource IDs of the resources to move. You can get resource IDs with the following command:

```
azure resource list -g sourceGroup --json
```

Which returns the following format:

```
[  
 {  
   "id":  
     "/subscriptions/{guid}/resourceGroups/sourceGroup/providers/Microsoft.Storage/storageAccounts/storagedemo",  
     "name": "storagedemo",  
     "type": "Microsoft.Storage/storageAccounts",  
     "location": "southcentralus",  
     "tags": {},  
     "kind": "Storage",  
     "sku": {  
       "name": "Standard_RAGRS",  
       "tier": "Standard"  
     }  
 }  
]
```

The following example shows how to move a storage account to a new resource group. In the `-i` parameter, provide a comma-separated list of the resource IDs to move.

```
azure resource move -i  
"/subscriptions/{guid}/resourceGroups/sourceGroup/providers/Microsoft.Storage/storageAccounts/storagedemo" -d  
"destinationGroup"
```

You are asked to confirm that you want to move the specified resource.

Use REST API

To move existing resources to another resource group or subscription, run:

```
POST https://management.azure.com/subscriptions/{source-subscription-id}/resourcegroups/{source-resource-group-name}/moveResources?api-version={api-version}
```

In the request body, you specify the target resource group and the resources to move. For more information about the move REST operation, see [Move resources](#).

Next steps

- To learn about PowerShell cmdlets for managing your subscription, see [Using Azure PowerShell with Resource Manager](#).
- To learn about Azure CLI commands for managing your subscription, see [Using the Azure CLI with Resource](#)

[Manager](#).

- To learn about portal features for managing your subscription, see [Using the Azure portal to manage resources](#).
- To learn about applying a logical organization to your resources, see [Using tags to organize your resources](#).

Use Azure PowerShell to create a service principal to access resources

4/13/2017 • 10 min to read • [Edit Online](#)

When you have an app or script that needs to access resources, you can set up an identity for the app and authenticate the app with its own credentials. This identity is known as a service principal. This approach enables you to:

- Assign permissions to the app identity that are different than your own permissions. Typically, these permissions are restricted to exactly what the app needs to do.
- Use a certificate for authentication when executing an unattended script.

This topic shows you how to use [Azure PowerShell](#) to set up everything you need for an application to run under its own credentials and identity.

Required permissions

To complete this topic, you must have sufficient permissions in both your Azure Active Directory and your Azure subscription. Specifically, you must be able to create an app in the Active Directory, and assign the service principal to a role.

The easiest way to check whether your account has adequate permissions is through the portal. See [Check required permission](#).

Now, proceed to a section for either [password](#) or [certificate](#) authentication.

Create service principal with password

The following script creates an identity for your application, and assigns it to the Contributor role for the specified scope:

```

Param (
    # Use to set scope to resource group. If no value is provided, scope is set to subscription.
    [Parameter(Mandatory=$false)]
    [String] $ResourceGroup,
    # Use to set subscription. If no value is provided, default subscription is used.
    [Parameter(Mandatory=$false)]
    [String] $SubscriptionId,
    [Parameter(Mandatory=$true)]
    [String] $ApplicationDisplayName,
    [Parameter(Mandatory=$true)]
    [String] $Password
)

Login-AzureRmAccount
Import-Module AzureRM.Resources

if ($SubscriptionId -eq "")
{
    $SubscriptionId = (Get-AzureRmContext).Subscription.SubscriptionId
}
else
{
    Set-AzureRmContext -SubscriptionId $SubscriptionId
}

if ($ResourceGroup -eq "")
{
    $Scope = "/subscriptions/" + $SubscriptionId
}
else
{
    $Scope = (Get-AzureRmResourceGroup -Name $ResourceGroup -ErrorAction Stop).ResourceId
}

# Create Active Directory application with password
$Application = New-AzureRmADApplication -DisplayName $ApplicationDisplayName -HomePage ("http://" + $ApplicationDisplayName) -IdentifierUris ("http://" + $ApplicationDisplayName) -Password $Password

# Create Service Principal for the AD app
$ServicePrincipal = New-AzureRMServerPrincipal -ApplicationId $Application.ApplicationId
Get-AzureRmServerPrincipal -ObjectId $ServicePrincipal.Id

$NewRole = $null
$Retries = 0;
While ($NewRole -eq $null -and $Retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active (should only
    # take a couple of seconds normally)
    Sleep 15
    New-AzureRMRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName $Application.ApplicationId
    -Scope $Scope | Write-Verbose -ErrorAction SilentlyContinue
    $NewRole = Get-AzureRMRoleAssignment -ServicePrincipalName $Application.ApplicationId -ErrorAction
    SilentlyContinue
    $Retries++;
}

```

A few items to note about the script:

- To grant the identity access to the default subscription, you do not need to provide either ResourceGroup or SubscriptionId parameters.
- Specify the ResourceGroup parameter only when you want to limit the scope of the role assignment to a

resource group.

- For single-tenant applications, the home page and identifier URIs are not validated.
- In this example, you add the service principal to the Contributor role. For other roles, see [RBAC: Built-in roles](#).
- The script sleeps for 15 seconds to allow some time for the new service principal to propagate throughout Active Directory. If your script does not wait long enough, you see an error stating: "PrincipalNotFound: Principal {id} does not exist in the directory."
- If you need to grant the service principal access to more subscriptions or resource groups, run the `New-AzureRMRoleAssignment` cmdlet again with different scopes.

Provide credentials through PowerShell

Now, you need to log in as the application to perform operations. For the user name, use the `ApplicationId` that you created for the application. For the password, use the one you specified when creating the account.

```
$creds = Get-Credential  
Login-AzureRmAccount -Credential $creds -ServicePrincipal -TenantId {tenant-id}
```

The tenant ID is not sensitive, so you can embed it directly in your script. If you need to retrieve the tenant ID, use:

```
(Get-AzureRmSubscription -SubscriptionName "Contoso Default").TenantId
```

Create service principal with self-signed certificate

To generate a self-signed certificate and service principal with Azure PowerShell 2.0 on Windows 10 or Windows Server 2016 Technical Preview, use the following script:

```

Param (
    # Use to set scope to resource group. If no value is provided, scope is set to subscription.
    [Parameter(Mandatory=$false)]
    [String] $ResourceGroup,
    # Use to set subscription. If no value is provided, default subscription is used.
    [Parameter(Mandatory=$false)]
    [String] $SubscriptionId,
    [Parameter(Mandatory=$true)]
    [String] $ApplicationDisplayName
)

Login-AzureRmAccount
Import-Module AzureRM.Resources

if ($SubscriptionId -eq "")
{
    $SubscriptionId = (Get-AzureRmContext).Subscription.SubscriptionId
}
else
{
    Set-AzureRmContext -SubscriptionId $SubscriptionId
}

if ($ResourceGroup -eq "")
{
    $Scope = "/subscriptions/" + $SubscriptionId
}
else
{
    $Scope = (Get-AzureRmResourceGroup -Name $ResourceGroup -ErrorAction Stop).ResourceId
}

$cert = New-SelfSignedCertificate -CertStoreLocation "cert:\CurrentUser\My" -Subject
"CN=exampleappScriptCert" -KeySpec KeyExchange
$keyValue = [System.Convert]::ToBase64String($cert.GetRawCertData())

# Use Key credentials
$Application = New-AzureRmADApplication -DisplayName $ApplicationDisplayName -HomePage ("http://" +
$ApplicationDisplayName) -IdentifierUris ("http://" + $ApplicationDisplayName) -CertValue $keyValue -EndDate
$cert.NotAfter -StartDate $cert.NotBefore

$ServicePrincipal = New-AzureRMADServicePrincipal -ApplicationId $Application.ApplicationId
Get-AzureRmADServicePrincipal -ObjectId $ServicePrincipal.Id

$NewRole = $null
$Retries = 0;
While ($NewRole -eq $null -and $Retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active (should only
    # take a couple of seconds normally)
    Sleep 15
    New-AzureRMRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName $Application.ApplicationId
    -Scope $Scope | Write-Verbose -ErrorAction SilentlyContinue
    $NewRole = Get-AzureRMRoleAssignment -ServicePrincipalName $Application.ApplicationId -ErrorAction
    SilentlyContinue
    $Retries++;
}

```

A few items to note about the script:

- To grant the identity access to the default subscription, you do not need to provide either ResourceGroup or SubscriptionId parameters.
- Specify the ResourceGroup parameter only when you want to limit the scope of the role assignment to a

resource group.

- For single-tenant applications, the home page and identifier URIs are not validated.
- In this example, you add the service principal to the Contributor role. For other roles, see [RBAC: Built-in roles](#).
- The script sleeps for 15 seconds to allow some time for the new service principal to propagate throughout Active Directory. If your script does not wait long enough, you see an error stating: "PrincipalNotFound: Principal {id} does not exist in the directory."
- If you need to grant the service principal access to more subscriptions or resource groups, run the `New-AzureRMRoleAssignment` cmdlet again with different scopes.

If you **do not have Windows 10 or Windows Server 2016 Technical Preview**, you need to download the [Self-signed certificate generator](#) from Microsoft Script Center. Extract its contents and import the cmdlet you need.

```
# Only run if you could not use New-SelfSignedCertificate
Import-Module -Name c:\ExtractedModule\New-SelfSignedCertificateEx.ps1
```

In the script, substitute the following two lines to generate the certificate.

```
New-SelfSignedCertificateEx -StoreLocation CurrentUser -StoreName My -Subject "CN=exampleapp" -KeySpec "Exchange" -FriendlyName "exampleapp"
$cert = Get-ChildItem -path Cert:\CurrentUser\my | where {$PSitem.Subject -eq 'CN=exampleapp' }
```

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant id of the directory for your AD app. A tenant is an instance of Active Directory. If you only have one subscription, you can use:

```
Param (

    [Parameter(Mandatory=$true)]
    [String] $CertSubject,

    [Parameter(Mandatory=$true)]
    [String] $ApplicationId,

    [Parameter(Mandatory=$true)]
    [String] $TenantId
)

$Thumbprint = (Get-ChildItem cert:\CurrentUser\My\ | Where-Object {$_.Subject -match $CertSubject }).Thumbprint
Login-AzureRmAccount -ServicePrincipal -CertificateThumbprint $Thumbprint -ApplicationId $ApplicationId - TenantId $TenantId
```

The application ID and tenant ID are not sensitive, so you can embed them directly in your script. If you need to retrieve the tenant ID, use:

```
(Get-AzureRmSubscription -SubscriptionName "Contoso Default").TenantId
```

If you need to retrieve the application ID, use:

```
(Get-AzureRmADApplication -DisplayNameStartWith {display-name}).ApplicationId
```

Create service principal with certificate from Certificate Authority

To use a certificate issued from a Certificate Authority to create service principal, use the following script:

```
Param (
    [Parameter(Mandatory=$true)]
    [String] $ApplicationDisplayName,
    [Parameter(Mandatory=$true)]
    [String] $SubscriptionId,
    [Parameter(Mandatory=$true)]
    [String] $CertPath,
    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword
)

Login-AzureRmAccount
Import-Module AzureRM.Resources
Set-AzureRmContext -SubscriptionId $SubscriptionId

$keyId = (New-Guid).Guid
$certPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force

$PFXCert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList
@($CertPath, $certPassword)
$keyValue = [System.Convert]::ToBase64String($PFXCert.GetRawCertData())

$keyCredential = New-Object Microsoft.Azure.Commands.Resources.Models.ActiveDirectory.PSADKeyCredential
$keyCredential.StartDate = $PFXCert.NotBefore
$keyCredential.EndDate= $PFXCert.NotAfter
$keyCredential.KeyId = $keyId
$keyCredential.CertValue = $keyValue

# Use Key credentials
$application = New-AzureRmADApplication -DisplayName $ApplicationDisplayName -HomePage ("http://" +
$ApplicationDisplayName) -IdentifierUris ("http://" + $keyId) -KeyCredentials $keyCredential

$servicePrincipal = New-AzureRMADServicePrincipal -ApplicationId $application.ApplicationId
Get-AzureRmADServicePrincipal -ObjectId $servicePrincipal.Id

$newRole = $null
$retries = 0;
While ($newRole -eq $null -and $retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active (should only
    take a couple of seconds normally)
    Sleep 15
    New-AzureRMRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName $application.ApplicationId
    | Write-Verbose -ErrorAction SilentlyContinue
    $newRole = Get-AzureRMRoleAssignment -ServicePrincipalName $application.ApplicationId -ErrorAction
    SilentlyContinue
    $retries++;
}

$newRole
```

A few items to note about the script:

- Access is scoped to the subscription.
- For single-tenant applications, the home page and identifier URIs are not validated.
- In this example, you add the service principal to the Contributor role. For other roles, see [RBAC: Built-in roles](#).
- The script sleeps for 15 seconds to allow some time for the new service principal to propagate throughout Active Directory. If your script does not wait long enough, you see an error stating: "PrincipalNotFound: Principal {id} does not exist in the directory."

- If you need to grant the service principal access to more subscriptions or resource groups, run the `New-AzureRMRoleAssignment` cmdlet again with different scopes.

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant id of the directory for your AD app. A tenant is an instance of Active Directory.

```
Param (
    [Parameter(Mandatory=$true)]
    [String] $CertPath,
    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword,
    [Parameter(Mandatory=$true)]
    [String] $ApplicationId,
    [Parameter(Mandatory=$true)]
    [String] $TenantId
)

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force
$PFXCert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList
@($CertPath, $CertPassword)
$Thumbprint = $PFXCert.Thumbprint

Login-AzureRmAccount -ServicePrincipal -CertificateThumbprint $Thumbprint -ApplicationId $ApplicationId -
TenantId $TenantId
```

The application ID and tenant ID are not sensitive, so you can embed them directly in your script. If you need to retrieve the tenant ID, use:

```
(Get-AzureRmSubscription -SubscriptionName "Contoso Default").TenantId
```

If you need to retrieve the application ID, use:

```
(Get-AzureRmADApplication -DisplayNameStartWith {display-name}).ApplicationId
```

Change credentials

To change the credentials for an AD app, either because of a security compromise or a credential expiration, use the [Remove-AzureRmADAppCredential](#) and [New-AzureRmADAppCredential](#) cmdlets.

To remove all the credentials for an application, use:

```
Remove-AzureRmADAppCredential -ApplicationId 8bc80782-a916-47c8-a47e-4d76ed755275 -All
```

To add a password, use:

```
New-AzureRmADAppCredential -ApplicationId 8bc80782-a916-47c8-a47e-4d76ed755275 -Password p@ssword!
```

To add a certificate value, create a self-signed certificate as shown in this topic. Then, use:

```
New-AzureRmADAppCredential -ApplicationId 8bc80782-a916-47c8-a47e-4d76ed755275 -CertValue $keyValue -EndDate  
$cert.NotAfter -StartDate $cert.NotBefore
```

Save access token to simplify log in

To avoid providing the service principal credentials every time it needs to log in, you can save the access token.

To use the current access token in a later session, save the profile.

```
Save-AzureRmProfile -Path c:\Users\exampleuser\profile\exampleSP.json
```

Open the profile and examine its contents. Notice that it contains an access token. Instead of manually logging in again, simply load the profile.

```
Select-AzureRmProfile -Path c:\Users\exampleuser\profile\exampleSP.json
```

NOTE

The access token expires, so using a saved profile only works for as long as the token is valid.

Alternatively, you can invoke REST operations from PowerShell to log in. From the authentication response, you can retrieve the access token for use with other operations. For an example of retrieving the access token by invoking REST operations, see [Generating an Access Token](#).

Debug

You may encounter the following errors when creating a service principal:

- **"Authentication_Unauthorized"** or **"No subscription found in the context."** - You see this error when your account does not have the [required permissions](#) on the Active Directory to register an app. Typically, you see this error when only admin users in your Active Directory can register apps, and your account is not an admin. Ask your administrator to either assign you to an administrator role, or to enable users to register apps.
- Your account "**does not have authorization to perform action 'Microsoft.Authorization/roleAssignments/write' over scope '/subscriptions/{guid}'.**" - You see this error when your account does not have sufficient permissions to assign a role to an identity. Ask your subscription administrator to add you to User Access Administrator role.

Sample applications

The following sample applications show how to log in as the service principal.

.NET

- [Deploy an SSH Enabled VM with a Template with .NET](#)
- [Manage Azure resources and resource groups with .NET](#)

Java

- [Getting Started with Resources - Deploy Using Azure Resource Manager Template - in Java](#)
- [Getting Started with Resources - Manage Resource Group - in Java](#)

Python

- [Deploy an SSH Enabled VM with a Template in Python](#)
- [Managing Azure Resource and Resource Groups with Python](#)

Node.js

- [Deploy an SSH Enabled VM with a Template in Nodejs](#)
- [Manage Azure resources and resource groups with Node.js](#)

Ruby

- [Deploy an SSH Enabled VM with a Template in Ruby](#)
- [Managing Azure Resource and Resource Groups with Ruby](#)

Next Steps

- For detailed steps on integrating an application into Azure for managing resources, see [Developer's guide to authorization with the Azure Resource Manager API](#).
- For a more detailed explanation of applications and service principals, see [Application Objects and Service Principal Objects](#).
- For more information about Active Directory authentication, see [Authentication Scenarios for Azure AD](#).

Use Azure CLI to create a service principal to access resources

4/3/2017 • 8 min to read • [Edit Online](#)

When you have an app or script that needs to access resources, you can set up an identity for the app and authenticate the app with its own credentials. This identity is known as a service principal. This approach enables you to:

- Assign permissions to the app identity that are different than your own permissions. Typically, these permissions are restricted to exactly what the app needs to do.
- Use a certificate for authentication when executing an unattended script.

This article shows you how to use [Azure CLI 1.0](#) to set up an application to run under its own credentials and identity. Install the latest version of [Azure CLI 1.0](#) to make sure your environment matches the examples in this article.

Required permissions

To complete this topic, you must have sufficient permissions in both your Azure Active Directory and your Azure subscription. Specifically, you must be able to create an app in the Active Directory, and assign the service principal to a role.

The easiest way to check whether your account has adequate permissions is through the portal. See [Check required permission in portal](#).

Now, proceed to a section for either [password](#) or [certificate](#) authentication.

Create service principal with password

In this section, you perform the steps to create the AD application with a password, and assign the Reader role to the service principal.

1. Sign in to your account.

```
azure login
```

2. To create an app identity, provide the name of the app and a password, as shown in the following command:

```
azure ad sp create -n exampleapp -p {your-password}
```

The new service principal is returned. The Object Id is needed when granting permissions. The guid listed with the Service Principal Names is needed when logging in. This guid is the same value as the app id. In the sample applications, this value is referred to as the `client ID`.

```
info: Executing command ad sp create

Creating application exampleapp
 / Creating service principal for application 7132aca4-1bdb-4238-ad81-996ff91d8db+
data: Object Id: ff863613-e5e2-4a6b-af07-fff6f2de3f4e
data: Display Name: exampleapp
data: Service Principal Names:
data: 7132aca4-1bdb-4238-ad81-996ff91d8db4
data: https://www.contoso.org/example
info: ad sp create command OK
```

3. Grant the service principal permissions on your subscription. In this example, you add the service principal to the Reader role, which grants permission to read all resources in the subscription. For other roles, see [RBAC: Built-in roles](#). For the objectid parameter, provide the Object Id that you used when creating the application. Before running this command, you must allow some time for the new service principal to propagate throughout Active Directory. When you run these commands manually, usually enough time has elapsed between tasks. In a script, you should add a step to sleep between the commands (like `sleep 15`). If you see an error stating the principal does not exist in the directory, rerun the command.

```
azure role assignment create --objectId ff863613-e5e2-4a6b-af07-fff6f2de3f4e -o Reader -c
/subscriptions/{subscriptionId}/
```

That's it! Your AD application and service principal are set up. The next section shows you how to log in with the credential through Azure CLI. If you want to use the credential in your code application, you do not need to continue with this topic. You can jump to the [Sample applications](#) for examples of logging in with your application id and password.

Provide credentials through Azure CLI

Now, you need to log in as the application to perform operations.

1. Whenever you sign in as a service principal, you need to provide the tenant id of the directory for your AD app. A tenant is an instance of Active Directory. To retrieve the tenant id for your currently authenticated subscription, use:

```
azure account show
```

Which returns:

```
info: Executing command account show
data: Name : Windows Azure MSDN - Visual Studio Ultimate
data: ID : {guid}
data: State : Enabled
data: Tenant ID : {guid}
data: Is Default : true
...
```

If you need to get the tenant id of another subscription, use the following command:

```
azure account show -s {subscription-id}
```

2. If you need to retrieve the client id to use for logging in, use:

```
azure ad sp show -c exampleapp --json
```

The value to use for logging in is the guid listed in the service principal names.

```
[  
  {  
    "objectId": "ff863613-e5e2-4a6b-af07-fff6f2de3f4e",  
    "objectType": "ServicePrincipal",  
    "displayName": "exampleapp",  
    "appId": "7132aca4-1bdb-4238-ad81-996ff91d8db4",  
    "servicePrincipalNames": [  
      "https://www.contoso.org/example",  
      "7132aca4-1bdb-4238-ad81-996ff91d8db4"  
    ]  
  }  
]
```

3. Log in as the service principal.

```
azure login -u 7132aca4-1bdb-4238-ad81-996ff91d8db4 --service-principal --tenant {tenant-id}
```

You are prompted for the password. Provide the password you specified when creating the AD application.

```
info: Executing command login  
Password: *****
```

You are now authenticated as the service principal for the service principal that you created.

Alternatively, you can invoke REST operations from the command line to log in. From the authentication response, you can retrieve the access token for use with other operations. For an example of retrieving the access token by invoking REST operations, see [Generating an Access Token](#).

Create service principal with certificate

In this section, you perform the steps to:

- create a self-signed certificate
- create the AD application with the certificate, and the service principal
- assign the Reader role to the service principal

To complete these steps, you must have [OpenSSL](#) installed.

1. Create a self-signed certificate.

```
openssl req -x509 -days 3650 -newkey rsa:2048 -out cert.pem -nodes -subj '/CN=exampleapp'
```

2. The preceding step created two files - privkey.pem and cert.pem. Combine the public and private keys into a single file.

```
cat privkey.pem cert.pem > examplecert.pem
```

3. Open the **examplecert.pem** file and look for the long sequence of characters between **-----BEGIN CERTIFICATE-----** and **-----END CERTIFICATE-----**. Copy the certificate data. You pass this data as a parameter when creating the service principal.

4. Sign in to your account.

```
azure login
```

- To create the service principal, provide the name of the app and the certificate data, as shown in the following command:

```
azure ad sp create -n exampleapp --cert-value {certificate data}
```

The new service principal is returned. The Object Id is needed when granting permissions. The guid listed with the Service Principal Names is needed when logging in. This guid is the same value as the app id. In the sample applications, this value is referred to as the Client ID.

```
info: Executing command ad sp create

Creating service principal for application 4fd39843-c338-417d-b549-a545f584a74+
data: Object Id: 7dbc8265-51ed-4038-8e13-31948c7f4ce7
data: Display Name: exampleapp
data: Service Principal Names:
data: 4fd39843-c338-417d-b549-a545f584a745
data: https://www.contoso.org/example
info: ad sp create command OK
```

- Grant the service principal permissions on your subscription. In this example, you add the service principal to the Reader role, which grants permission to read all resources in the subscription. For other roles, see [RBAC: Built-in roles](#). For the objectid parameter, provide the Object Id that you used when creating the application. Before running this command, you must allow some time for the new service principal to propagate throughout Active Directory. When you run these commands manually, usually enough time has elapsed between tasks. In a script, you should add a step to sleep between the commands (like `sleep 15`). If you see an error stating the principal does not exist in the directory, rerun the command.

```
azure role assignment create --objectId 7dbc8265-51ed-4038-8e13-31948c7f4ce7 -o Reader -c /subscriptions/{subscriptionId}/
```

Provide certificate through automated Azure CLI script

Now, you need to log in as the application to perform operations.

- Whenever you sign in as a service principal, you need to provide the tenant id of the directory for your AD app. A tenant is an instance of Active Directory. To retrieve the tenant id for your currently authenticated subscription, use:

```
azure account show
```

Which returns:

```
info: Executing command account show
data: Name : Windows Azure MSDN - Visual Studio Ultimate
data: ID : {guid}
data: State : Enabled
data: Tenant ID : {guid}
data: Is Default : true
...
```

If you need to get the tenant id of another subscription, use the following command:

```
azure account show -s {subscription-id}
```

2. To retrieve the certificate thumbprint and remove unneeded characters, use:

```
openssl x509 -in "C:\certificates\examplecert.pem" -fingerprint -noout | sed 's/SHA1 Fingerprint=//g'  
| sed 's/://g'
```

Which returns a thumbprint value similar to:

```
30996D9CE48A0B6E0CD49DBB9A48059BF9355851
```

3. If you need to retrieve the client id to use for logging in, use:

```
azure ad sp show -c exampleapp
```

The value to use for logging in is the guid listed in the service principal names.

```
[  
 {  
   "objectId": "7dbc8265-51ed-4038-8e13-31948c7f4ce7",  
   "objectType": "ServicePrincipal",  
   "displayName": "exampleapp",  
   "appId": "4fd39843-c338-417d-b549-a545f584a745",  
   "servicePrincipalNames": [  
     "https://www.contoso.org/example",  
     "4fd39843-c338-417d-b549-a545f584a745"  
   ]  
 }  
]
```

4. Log in as the service principal.

```
azure login --service-principal --tenant {tenant-id} -u 4fd39843-c338-417d-b549-a545f584a745 --  
certificate-file C:\certificates\examplecert.pem --thumbprint {thumbprint}
```

You are now authenticated as the service principal for the Active Directory application that you created.

Change credentials

To change the credentials for an AD app, either because of a security compromise or a credential expiration, use

```
azure ad app set .
```

To change a password, use:

```
azure ad app set --applicationId 4fd39843-c338-417d-b549-a545f584a745 --password p@ssword
```

To change a certificate value, use:

```
azure ad app set --applicationId 4fd39843-c338-417d-b549-a545f584a745 --cert-value {certificate data}
```

Debug

You may encounter the following errors when creating a service principal:

- **"Authentication_Unauthorized"** or **"No subscription found in the context."** - You see this error when your account does not have the [required permissions](#) on the Active Directory to register an app. Typically, you see this error when only admin users in your Active Directory can register apps, and your account is not an admin. Ask your administrator to either assign you to an administrator role, or to enable users to register apps.
- Your account **"does not have authorization to perform action 'Microsoft.Authorization/roleAssignments/write' over scope '/subscriptions/{guid}'."** - You see this error when your account does not have sufficient permissions to assign a role to an identity. Ask your subscription administrator to add you to User Access Administrator role.

Sample applications

The following sample applications show how to log in as the service principal.

.NET

- [Deploy an SSH Enabled VM with a Template with .NET](#)
- [Manage Azure resources and resource groups with .NET](#)

Java

- [Getting Started with Resources - Deploy Using Azure Resource Manager Template - in Java](#)
- [Getting Started with Resources - Manage Resource Group - in Java](#)

Python

- [Deploy an SSH Enabled VM with a Template in Python](#)
- [Managing Azure Resource and Resource Groups with Python](#)

Node.js

- [Deploy an SSH Enabled VM with a Template in Node.js](#)
- [Manage Azure resources and resource groups with Nodejs](#)

Ruby

- [Deploy an SSH Enabled VM with a Template in Ruby](#)
- [Managing Azure Resource and Resource Groups with Ruby](#)

Next Steps

- For detailed steps on integrating an application into Azure for managing resources, see [Developer's guide to authorization with the Azure Resource Manager API](#).
- To get more information about using certificates and Azure CLI, see [Certificate-based authentication with Azure Service Principals from Linux command line](#).

Use portal to create Active Directory application and service principal that can access resources

1/24/2017 • 5 min to read • [Edit Online](#)

When you have an application that needs to access or modify resources, you must set up an Active Directory (AD) application and assign the required permissions to it. This approach is preferable to running the app under your own credentials because:

- You can assign permissions to the app identity that are different than your own permissions. Typically, these permissions are restricted to exactly what the app needs to do.
- You do not have to change the app's credentials if your responsibilities change.
- You can use a certificate to automate authentication when executing an unattended script.

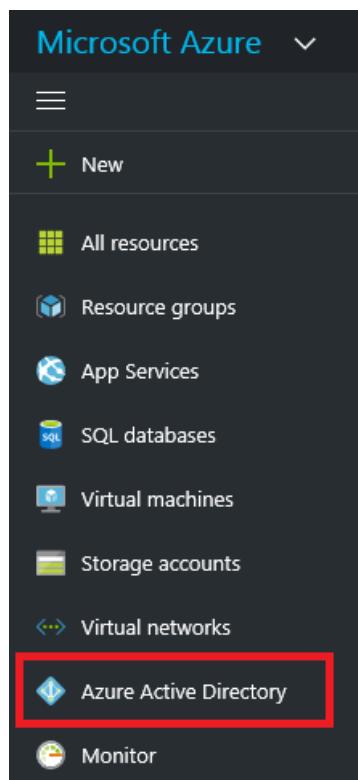
This topic shows you how to perform those steps through the portal. It focuses on a single-tenant application where the application is intended to run within only one organization. You typically use single-tenant applications for line-of-business applications that run within your organization.

Required permissions

To complete this topic, you must have sufficient permissions to register an application with your Active Directory, and assign the application to a role in your Azure subscription. Let's make sure you have the right permissions to perform those steps.

Check Active Directory permissions

1. Log in to your Azure Account through the [Azure portal](#).
2. Select **Azure Active Directory**.



3. In your Active Directory, select **User settings**.

The screenshot shows the Microsoft Azure Active Directory - PREVIEW interface. The left sidebar has a search bar at the top, followed by 'Overview' and 'Quick start' buttons. Below these are sections for 'MANAGE' with links to 'Users and groups', 'Enterprise applications', 'App registrations', 'Azure AD Connect', 'Domain names', 'User settings' (which is highlighted with a red box), and 'Properties'.

4. Check the **App registrations** setting. If set to **Yes**, non-admin users can register AD apps. This setting means any user in the Active Directory can register an app. You can proceed to [Check Azure subscription permissions](#).

The screenshot shows the 'Microsoft - User settings' page. The left sidebar includes 'Overview', 'Quick start', 'MANAGE' (with 'User settings' selected and highlighted with a blue box), and other options like 'Enterprise applications', 'App registrations', 'Azure AD Connect', 'Domain names', 'Properties'. The main content area shows 'Enterprise applications' settings: 'Users can allow apps to access their data' (Yes) and 'Users can add gallery apps to their Access Panel' (Yes). A red box highlights the 'App registrations' section, which contains 'Users can register applications' (Yes). The 'External Users' section includes 'Guest users permissions are limited' (Yes), 'Admins and users in the guest inviter role can invite' (Yes), and 'Members can invite' (Yes).

5. If the app registrations setting is set to **No**, only admin users can register apps. You need to check whether your account is an admin for the Active Directory. Select **Overview** and **Find a user** from Quick tasks.

The screenshot shows the Microsoft Azure Active Directory - PREVIEW portal. At the top, there's a navigation bar with 'Classic portal', 'Switch directory', and 'Delete' buttons. A message banner says 'The Azure AD management experience is in preview. Learn more →'. On the left, a sidebar has a 'Search (Ctrl+/' input field, an 'Overview' button (which is highlighted with a red box), a 'Quick start' link, and sections for 'MANAGE' (with 'Users and groups' and 'Enterprise applications' links). In the center, there's a 'Users and groups' section with a grid of icons representing users and groups, and a 'Quick tasks' sidebar with links like 'Add a user', 'Add a group', 'Find a user' (which is highlighted with a red box), 'Find a group', and 'Find an enterprise app'.

6. Search for your account, and select it when you find it.

The screenshot shows a list of users. At the top, there are buttons for 'Add', 'Columns', 'Multi-Factor A...', and 'Filter'. Below is a search bar containing 'Example Person'. The main area shows two columns: 'NAME' and 'USER NAME'. A user named 'Example Person' is listed, with their profile picture and email address ('example@contoso.org'). The 'NAME' column for this user is highlighted with a red box.

7. For your account, select **Directory role**.

The screenshot shows the 'Profile' section of the user's account. The sidebar on the left includes 'Overview' (highlighted with a red box), 'MANAGE' (with 'Profile' and 'Directory role' (highlighted with a red box) under it), and 'Groups'.

8. View your assigned role for the Active Directory. If your account is assigned to the User role, but the app registration setting (from the preceding steps) is limited to admin users, ask your administrator to either assign you to an administrator role, or to enable users to register apps.

The screenshot shows the 'Directory role' configuration page. It has a 'Save' and 'Discard' button at the top. The 'User' radio button is selected under 'Directory role'. A note below says 'Users can access assigned resources but cannot manage most directory resources.' with a 'Learn More' link.

Check Azure subscription permissions

In your Azure subscription, your account must have `Microsoft.Authorization/*/Write` access to assign an AD app to a role. This action is granted through the **Owner** role or **User Access Administrator** role. If your account is assigned to the **Contributor** role, you do not have adequate permission. You will receive an error when

attempting to assign the service principal to a role.

To check your subscription permissions:

1. If you are not already looking at your Active Directory account from the preceding steps, select **Azure Active Directory** from the left pane.
2. Find your Active Directory account. Select **Overview** and **Find a user** from Quick tasks.

The screenshot shows the Azure Active Directory - PREVIEW portal. At the top, there's a search bar labeled "Search (Ctrl+ /)" and navigation links for "Classic portal", "Switch directory", and "Delete". A message says "The Azure AD management experience is in preview. Learn more →". On the left, a sidebar has "Overview" (highlighted with a red box), "Quick start", and sections for "MANAGE" (Users and groups, Enterprise applications). The main area shows "Users and groups" with a grid of icons representing users (e.g., DS, DD, SM, SC, _D, CL) and groups (e.g., D1, GA, CO, TE, RO, RS, GP, RC). To the right, under "Quick tasks", there are links for "Add a user", "Add a group", "Find a user" (highlighted with a red box), "Find a group", and "Find an enterprise app".

3. Search for your account, and select it when you find it.

The screenshot shows a search results page for users. At the top, there are buttons for "Add", "Columns", "Multi-Factor A...", and "Filter". A search bar contains "Example Person". Below, a table lists users with columns for "NAME" and "USER NAME". One row is highlighted with a red box, showing a profile picture of "Example Person" and the email "example@contoso.org".

4. Select **Azure resources**.

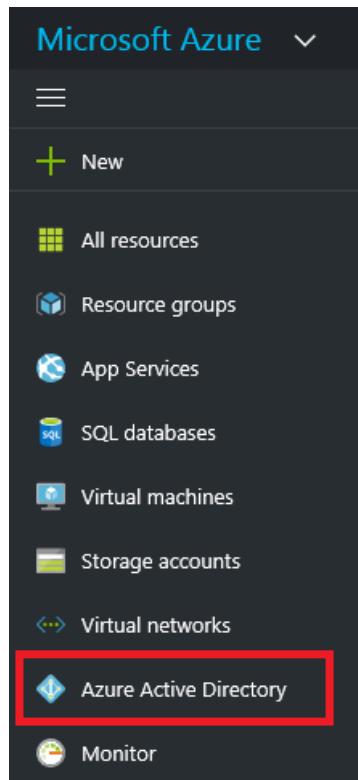
The screenshot shows the user profile page for "Example Person". The left sidebar includes "Overview" (highlighted with a red box), "MANAGE" (Profile, Directory role, Groups), and "Azure resources" (highlighted with a red box).

5. View your assigned roles, and determine if you have adequate permissions to assign an AD app to a role. If not, ask your subscription administrator to add you to User Access Administrator role. In the following image, the user is assigned to the Owner role for two subscriptions, which means that user has adequate permissions.

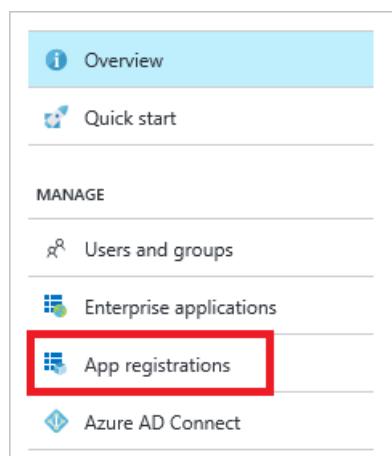
RESOURCE NAME	RESOURCE TYPE	ROLE	ASSIGNED TO
Microsoft Azure Internal Cons...	Subscription	Owner	Subscription ad...
Visual Studio Enterprise	Subscription	Owner	Subscription ad...

Create an Active Directory application

1. Log in to your Azure Account through the [Azure portal](#).
2. Select **Azure Active Directory**.



3. Select **App registrations**.



4. Select **Add**.



5. Provide a name and URL for the application. Select either **Web app / API** or **Native** for the type of application you want to create. After setting the values, select **Create**.

Create

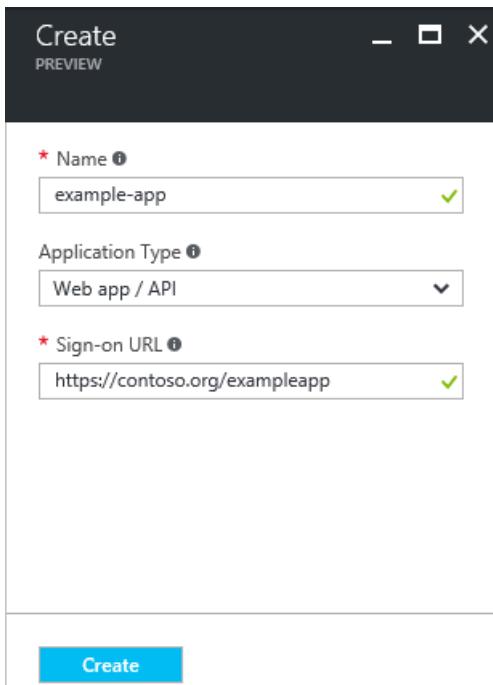
PREVIEW

* Name ⓘ
example-app ✓

Application Type ⓘ
Web app / API ▼

* Sign-on URL ⓘ
https://contoso.org/exampleapp ✓

Create



You have created your application.

Get application ID and authentication key

When programmatically logging in, you need the ID for your application and an authentication key. To get those values, use the following steps:

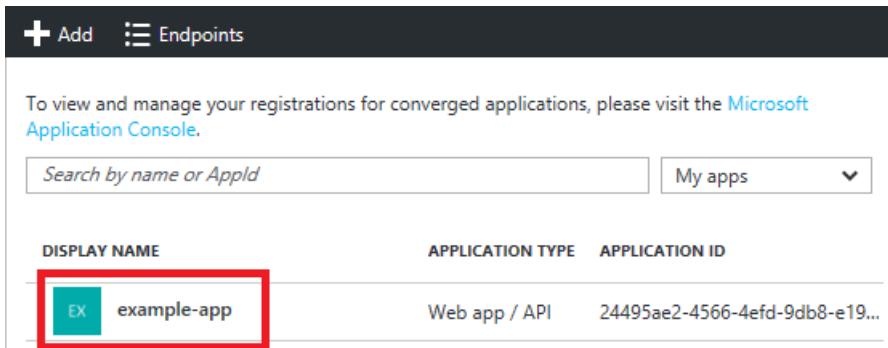
1. From **App registrations** in Active Directory, select your application.

+ Add Endpoints

To view and manage your registrations for converged applications, please visit the [Microsoft Application Console](#).

Search by name or AppId My apps ▼

DISPLAY NAME	APPLICATION TYPE	APPLICATION ID
EX example-app	Web app / API	2449ae2-4566-4efd-9db8-e19...



2. Copy the **Application ID** and store it in your application code. The applications in the [sample applications](#) section refer to this value as the client id.

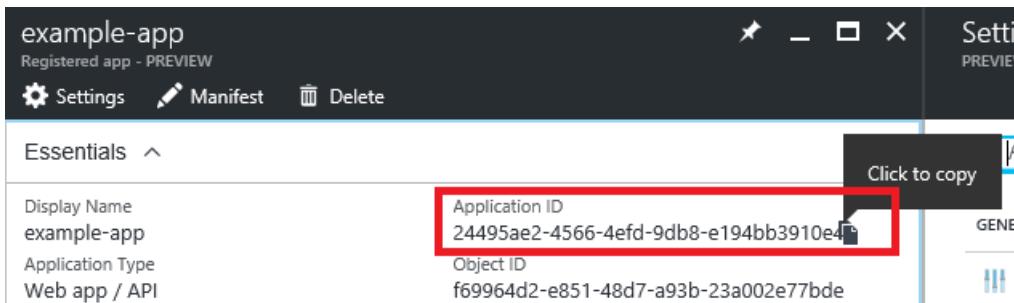
example-app
Registered app - PREVIEW

Settings Manifest Delete

Essentials ^

Display Name example-app	Application ID 2449ae2-4566-4efd-9db8-e194bb3910e4
Application Type Web app / API	Object ID f69964d2-e851-48d7-a93b-23a002e77bde

Click to copy



3. To generate an authentication key, select **Keys**.

The screenshot shows the 'Settings' blade in the Azure portal. Under the 'API ACCESS' section, there is a list of items: 'Required permissions' and 'Keys'. The 'Keys' item is highlighted with a red box.

4. Provide a description of the key, and a duration for the key. When done, select **Save**.

The screenshot shows the 'Keys' blade. A new key entry is being created with the following details:
DESCRIPTION: first key
EXPIRES: In 1 year
VALUE: Value will be displayed on save

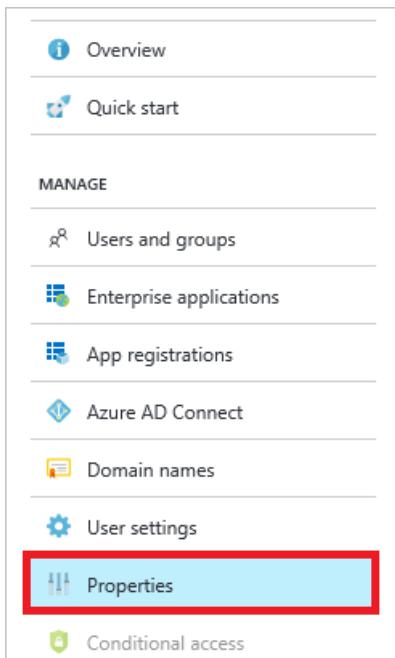
After saving the key, the value of the key is displayed. Copy this value because you are not able to retrieve the key later. You provide the key value with the application ID to log in as the application. Store the key value where your application can retrieve it.

The screenshot shows the 'Keys' blade after saving the key. A warning message is displayed: '⚠️ Copy the key value. You won't be able to retrieve after you leave this blade.' The key value 'jH9ALPPUICi0d1QTE1uiK9k09IY4Sr4hl/PdbI2Yr0=' is highlighted with a red box.

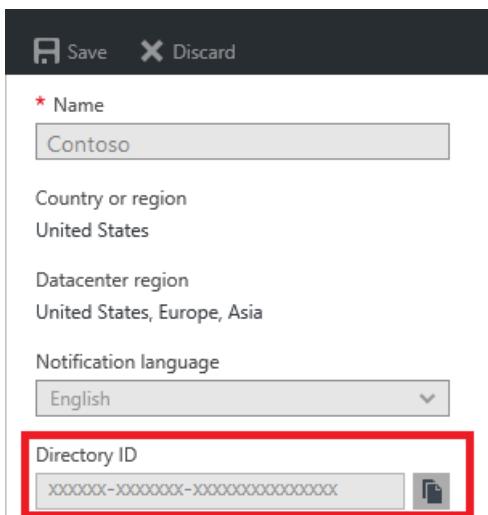
Get tenant ID

When programmatically logging in, you need to pass the tenant ID with your authentication request.

1. To get the tenant ID, select **Properties** for your Active Directory.



2. Copy the **Directory ID**. This value is your tenant ID.

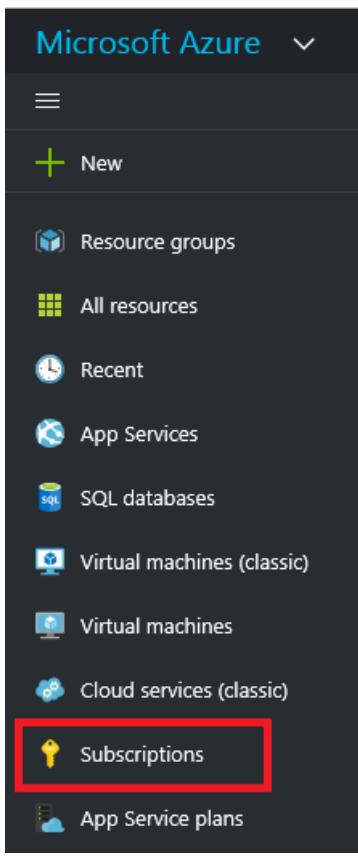


Assign application to role

To access resources in your subscription, you must assign the application to a role. Decide which role represents the right permissions for the application. To learn about the available roles, see [RBAC: Built in Roles](#).

You can set the scope at the level of the subscription, resource group, or resource. Permissions are inherited to lower levels of scope. For example, adding an application to the Reader role for a resource group means it can read the resource group and any resources it contains.

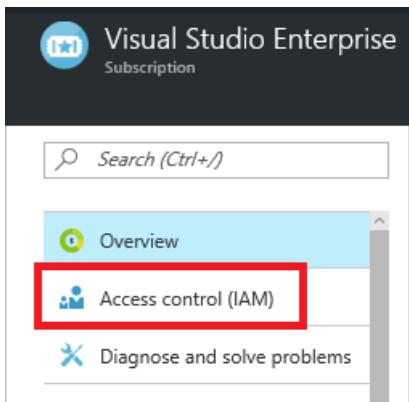
1. Navigate to the level of scope you wish to assign the application to. For example, to assign a role at the subscription scope, select **Subscriptions**. You could instead select a resource group or resource.



2. Select the particular subscription (resource group or resource) to assign the application to.

Subscriptions	
Microsoft	
+ Add	
Role <small>i</small>	Status <small>i</small>
<input type="button" value="All"/> <input type="button" value="All"/>	
<input type="button" value="Apply"/>	
<input type="text" value="Search to filter items..."/>	
SUBSCRIPTION	
SUBSCRIPTION ID	MY ROLE
 Visual Studio Enterprise	Account admin

3. Select **Access Control (IAM)**.



4. Select **Add**.

The screenshot shows the 'Access control (IAM)' section of the Visual Studio Enterprise dashboard. At the top right, there is a red box highlighting the '+ Add' button. Below it, there is a search bar and a sidebar with 'Overview' and 'Access control (IAM)' options.

5. Select the role you wish to assign to the application. The following image shows the **Reader** role.

This screenshot shows the 'Select a role' dialog. It lists three roles: 'Owner', 'Contributor', and 'Reader'. The 'Reader' option is highlighted with a red box.

6. Search for your application, and select it.

This screenshot shows the 'Add users' dialog. In the search bar, 'example-app' is typed. Below the search bar, the application 'example-app' is listed in the results, with a red box highlighting it.

7. Select **OK** to finish assigning the role. You see your application in the list of users assigned to a role for that scope.

Log in as the application

Your application is now set up in Active Directory. You have an ID and key to use for signing in as the application. The application is assigned to a role that gives it certain actions it can perform.

To log in through PowerShell, see [Provide credentials through PowerShell](#).

To log in through Azure CLI, see [Provide credentials through Azure CLI](#).

To get the access token for REST operations, see [Create the request](#).

Look at the following sample applications to learn about logging in through application code.

Sample applications

The following sample applications show how to log in as the AD application.

[.NET](#)

- [Deploy an SSH Enabled VM with a Template with .NET](#)
- [Manage Azure resources and resource groups with .NET](#)

Java

- [Getting Started with Resources - Deploy Using Azure Resource Manager Template - in Java](#)
- [Getting Started with Resources - Manage Resource Group - in Java](#)

Python

- [Deploy an SSH Enabled VM with a Template in Python](#)
- [Managing Azure Resource and Resource Groups with Python](#)

Node.js

- [Deploy an SSH Enabled VM with a Template in Node.js](#)
- [Manage Azure resources and resource groups with Node.js](#)

Ruby

- [Deploy an SSH Enabled VM with a Template in Ruby](#)
- [Managing Azure Resource and Resource Groups with Ruby](#)

Next Steps

- To set up a multi-tenant application, see [Developer's guide to authorization with the Azure Resource Manager API](#).
- To learn about specifying security policies, see [Azure Role-based Access Control](#).

Use Resource Manager authentication API to access subscriptions

4/5/2017 • 14 min to read • [Edit Online](#)

Introduction

If you are a software developer who needs to create an app that manages customer's Azure resources, this topic shows you how to authenticate with the Azure Resource Manager APIs and gain access to resources in other subscriptions.

Your app can access the Resource Manager APIs in couple of ways:

1. **User + app access:** for apps that access resources on behalf of a signed-in user. This approach works for apps, such as web apps and command-line tools, that deal with only "interactive management" of Azure resources.
2. **App-only access:** for apps that run daemon services and scheduled jobs. The app's identity is granted direct access to the resources. This approach works for apps that need long-term headless (unattended) access to Azure.

This topic provides step-by-step instructions to create an app that employs both these authorization methods. It shows how to perform each step with REST API or C#. The complete ASP.NET MVC application is available at <https://github.com/dushyantgill/VipSwapper/tree/master/CloudSense>.

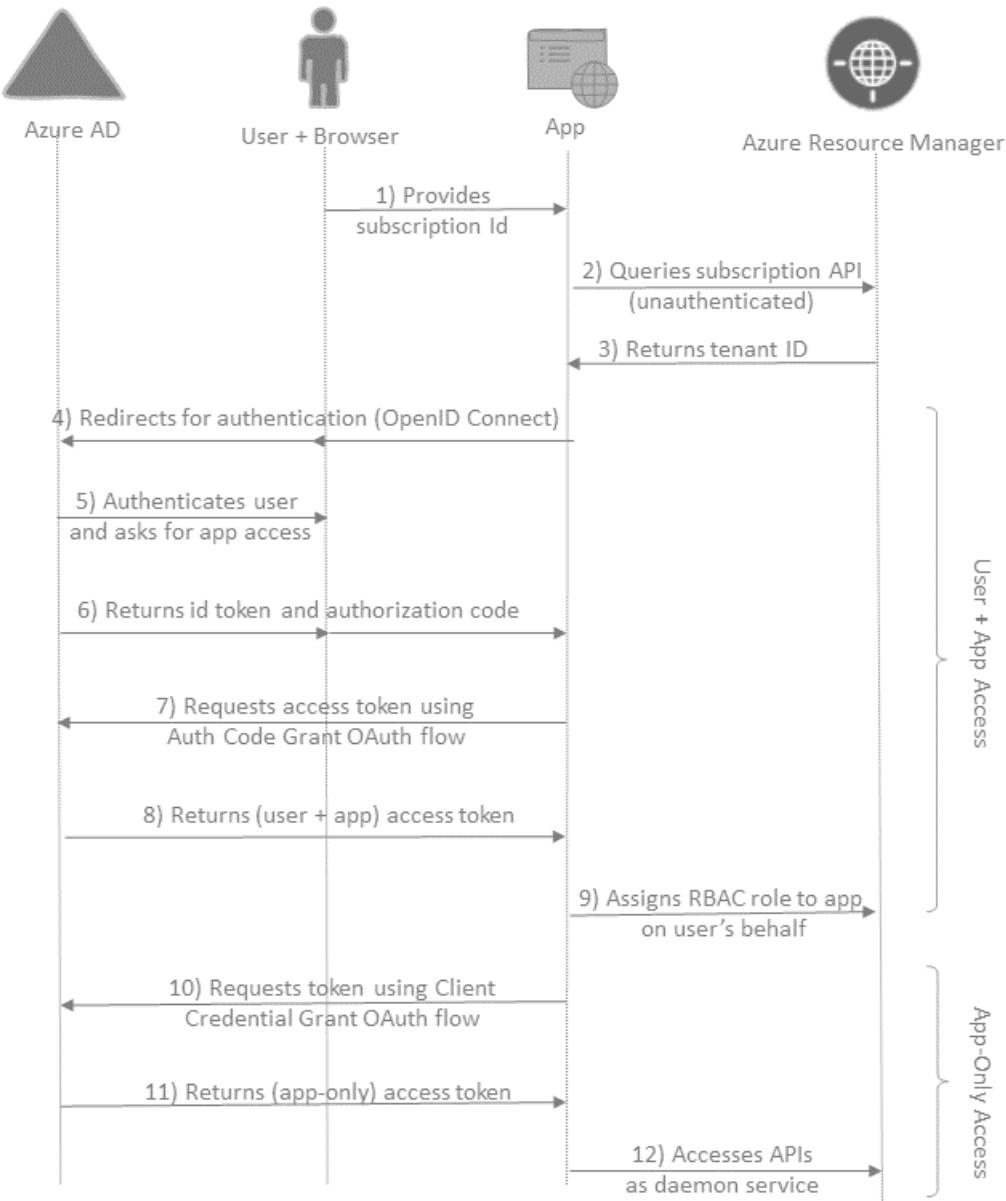
All the code for this topic is running as a web app that you can try at <http://vipswapper.azurewebsites.net/cloudsense>.

What the web app does

The web app:

1. Signs-in an Azure user.
2. Asks user to grant the web app access to Resource Manager.
3. Gets user + app access token for accessing Resource Manager.
4. Uses token (from step 3) to call Resource Manager and assign the app's service principal to a role in the subscription, which gives the app long-term access to the subscription.
5. Gets app-only access token.
6. Uses token (from step 5) to manage resources in the subscription through Resource Manager.

Here's the end-to-end flow of the web application.



As a user, you provide the subscription id for the subscription you want to use:

CloudSense

CloudSense

We monitor your Azure resources. You ... sleep well.

Enter your Azure subscription id Connect

Select the account to use for logging in.

CloudSense



Example User
example@contoso.org

...



Use another account

Provide your credentials.

Sign in

Microsoft account [What's this?](#)

example@contoso.org

Password

Keep me signed in

Sign in

[Can't access your account?](#)

[Sign in with a single-use code](#)

Grant the app access to your Azure subscriptions:

CloudSense

App publisher website: localhost

CloudSense needs permission to:

- Sign you in and read your profile [?](#)
- Access Azure Service Management as you (preview) [?](#)

You're signed in as: example@contoso.org

[Show details](#)

Accept

Cancel

Manage your connected subscriptions:

The screenshot shows the CloudSense web application. At the top, it says "CloudSense". Below that is a large "CloudSense" logo with the tagline "We monitor your Azure resources. You ... sleep well.". There is a text input field labeled "Enter your Azure subscription id" and a blue "Connect" button. Underneath, there's a section titled "Your connected subscriptions" with a table. The table has columns for "Subscription Id", "Connected On", and "Action". It contains one row with a GUID value for the Subscription Id, connected on 8/26/2016, and a red "Disconnect" button.

Subscription Id	Connected On	Action
{guid}	Connected On: 8/26/2016	<button>Disconnect</button>

Register application

Before you start coding, register your web app with Azure Active Directory (AD). The app registration creates a central identity for your app in Azure AD. It holds basic information about your application like OAuth Client ID, Reply URLs, and credentials that your application uses to authenticate and access Azure Resource Manager APIs. The app registration also records the various delegated permissions that your application needs when accessing Microsoft APIs on behalf of the user.

Because your app accesses other subscription, you must configure it as a multi-tenant application. To pass validation, provide a domain associated with your Active Directory. To see the domains associated with your Active Directory, log in to the [classic portal](#). Select your Active Directory and then select **Domains**.

The following example shows how to register the app by using Azure PowerShell. You must have the latest version (August 2016) of Azure PowerShell for this command to work.

```
$app = New-AzureRmADApplication -DisplayName "{app name}" -HomePage "https://{{your domain}}/{app name}" -IdentifierUris "https://{{your domain}}/{app name}" -Password "{your password}" -AvailableToOtherTenants $true
```

To log in as the AD application, you need the application id and password. To see the application id that is returned from the previous command, use:

```
$app.ApplicationId
```

The following example shows how to register the app by using Azure CLI.

```
azure ad app create --name {app name} --home-page https://{{your domain}}/{app name} --identifier-uris https://{{your domain}}/{app name} --password {your password} --available true
```

The results include the AppId, which you need when authenticating as the application.

Optional configuration - certificate credential

Azure AD also supports certificate credentials for applications: you create a self-signed cert, keep the private key,

and add the public key to your Azure AD application registration. For authentication, your application sends a small payload to Azure AD signed using your private key, and Azure AD validates the signature using the public key that you registered.

For information about creating an AD app with a certificate, see [Use Azure PowerShell to create a service principal to access resources](#) or [Use Azure CLI to create a service principal to access resources](#).

Get tenant id from subscription id

To request a token that can be used to call Resource Manager, your application needs to know the tenant ID of the Azure AD tenant that hosts the Azure subscription. Most likely, your users know their subscription ids, but they might not know their tenant ids for Active Directory. To get the user's tenant id, ask the user for the subscription id. Provide that subscription id when sending a request about the subscription:

```
https://management.azure.com/subscriptions/{subscription-id}?api-version=2015-01-01
```

The request fails because the user has not logged in yet, but you can retrieve the tenant id from the response. In that exception, retrieve the tenant id from the response header value for **WWW-Authenticate**. You see this implementation in the [GetDirectoryForSubscription](#) method.

Get user + app access token

Your application redirects the user to Azure AD with an OAuth 2.0 Authorize Request - to authenticate the user's credentials and get back an authorization code. Your application uses the authorization code to get an access token for Resource Manager. The [ConnectSubscription](#) method creates the authorization request.

This topic shows the REST API requests to authenticate the user. You can also use helper libraries to perform authentication in your code. For more information about these libraries, see [Azure Active Directory Authentication Libraries](#). For guidance on integrating identity management in an application, see [Azure Active Directory developer's guide](#).

Auth request (OAuth 2.0)

Issue an Open ID Connect/OAuth2.0 Authorize Request to the Azure AD Authorize endpoint:

```
https://login.microsoftonline.com/{tenant-id}/OAuth2/Authorize
```

The query string parameters that are available for this request are described in the [request an authorization code](#) topic.

The following example shows how to request OAuth2.0 authorization:

```
https://login.microsoftonline.com/{tenant-id}/OAuth2/Authorize?client_id=a0448380-c346-4f9f-b897-c18733de9394&response_mode=query&response_type=code&redirect_uri=http%3a%2fwww.vipswapper.com%2fccloudsense%2fAccount%2fSignIn&resource=https%3a%2f%2fgraph.windows.net%2f&domain_hint=live.com
```

Azure AD authenticates the user, and, if necessary, asks the user to grant permission to the app. It returns the authorization code to the Reply URL of your application. Depending on the requested response_mode, Azure AD either sends back the data in query string or as post data.

```
code=AAABAAAAiL****FDMZBUwZ8eCAA&session_state=2d16bbce-d5d1-443f-acdf-75f6b0ce8850
```

Auth request (Open ID Connect)

If you not only wish to access Azure Resource Manager on behalf of the user, but also allow the user to sign in to your application using their Azure AD account, issue an Open ID Connect Authorize Request. With Open ID Connect, your application also receives an id_token from Azure AD that your app can use to sign in the user.

The query string parameters that are available for this request are described in the [Send the sign-in request](#) topic.

An example Open ID Connect request is:

```
https://login.microsoftonline.com/{tenant-id}/OAuth2/Authorize?client_id=a0448380-c346-4f9f-b897-c18733de9394&response_mode=form_post&response_type=code+id_token&redirect_uri=http%3a%2f%2fwww.vipswapper.com%2fccloudsense%2fAccount%2fSignIn&resource=https%3a%2f%2fgraph.windows.net%2f&scope=openid+profile&nonce=63567Dc4MDAw&domain_hint=live.com&state=M_12tMyKaM8
```

Azure AD authenticates the user, and, if necessary, asks the user to grant permission to the app. It returns the authorization code to the Reply URL of your application. Depending on the requested response_mode, Azure AD either sends back the data in query string or as post data.

An example Open ID Connect response is:

```
code=AAABAAAAiL*****I4rDwD7zXsH6WUjlkIEQxIAA&id_token=eyJ0eXAiOiJKV1Q*****T3GrzzSFxg&state=M_12tMyKaM8&session_state=2d16bbce-d5d1-443f-acdf-75f6b0ce8850
```

Token request (OAuth2.0 Code Grant Flow)

Now that your application has received the authorization code from Azure AD, it is time to get the access token for Azure Resource Manager. Post an OAuth2.0 Code Grant Token Request to the Azure AD Token endpoint:

```
https://login.microsoftonline.com/{tenant-id}/OAuth2/Token
```

The query string parameters that are available for this request are described in the [use the authorization code](#) topic.

The following example shows a request for code grant token with password credential:

```
POST https://login.microsoftonline.com/7fe877e6-a150-4992-bbfe-f517e304dfa0/oauth2/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 1012

grant_type=authorization_code&code=AAABAAAAiL9Kn2Z*****L1nVMH3Z5ESiAA&redirect_uri=http%3A%2F%2Flocalhost%3A62080%2FAccount%2FSignIn&client_id=a0448380-c346-4f9f-b897-c18733de9394&client_secret=olna84E8*****goSc0g%3D
```

When working with certificate credentials, create a JSON Web Token (JWT) and sign (RSA SHA256) using the private key of your application's certificate credential. The claim types for the token are shown in [JWT token claims](#). For reference, see the [Active Directory Auth Library \(.NET\) code](#) to sign Client Assertion JWT tokens.

See the [Open ID Connect spec](#) for details on client authentication.

The following example shows a request for code grant token with certificate credential:

```
POST https://login.microsoftonline.com/7fe877e6-a150-4992-bbfe-f517e304dfa0/oauth2/token HTTP/1.1

Content-Type: application/x-www-form-urlencoded
Content-Length: 1012

grant_type=authorization_code&code=AAABAAAAiL9Kn2Z****L1nVMH3Z5ESiAA&redirect_uri=http%3A%2Flocalhost%3A62080%2FAccount%2FSignIn&client_id=a0448380-c346-4f9f-b897-c18733de9394&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&client_assertion=eyJhbG****Y9cYo8nEjMyA
```

An example response for code grant token:

```
HTTP/1.1 200 OK
```

```
{"token_type": "Bearer", "expires_in": "3599", "expires_on": "1432039858", "not_before": "1432035958", "resource": "https://management.core.windows.net/", "access_token": "eyJ0eXAiOiJKV1Q****M7Cw6JWtfY2lGc5A", "refresh_token": "AAABAAAiL9Kn2Z****55j-sjnyYgAA", "scope": "user_impersonation", "id_token": "eyJ0eXAiOiJKV1Q****-drP1J3P-HnHi9Rr46kGZnukEBH4dsg"}
```

Handle code grant token response

A successful token response contains the (user + app) access token for Azure Resource Manager. Your application uses this access token to access Resource Manager on behalf of the user. The lifetime of access tokens issued by Azure AD is one hour. It is unlikely that your web application needs to renew the (user + app) access token. If it needs to renew the access token, use the refresh token that your application receives in the token response. Post an OAuth2.0 Token Request to the Azure AD Token endpoint:

```
https://login.microsoftonline.com/{tenant-id}/OAuth2/Token
```

The parameters to use with the refresh request are described in [refreshing the access token](#).

The following example shows how to use the refresh token:

```
POST https://login.microsoftonline.com/7fe877e6-a150-4992-bbfe-f517e304dfa0/oauth2/token HTTP/1.1

Content-Type: application/x-www-form-urlencoded
Content-Length: 1012

grant_type=refresh_token&refresh_token=AAABAAAAiL9Kn2Z****55j-sjnyYgAA&client_id=a0448380-c346-4f9f-b897-c18733de9394&client_secret=olna84E8****goSc0g%3D
```

Although refresh tokens can be used to get new access tokens for Azure Resource Manager, they are not suitable for offline access by your application. The refresh tokens lifetime is limited, and refresh tokens are bound to the user. If the user leaves the organization, the application using the refresh token loses access. This approach isn't suitable for applications that are used by teams to manage their Azure resources.

Check if user can assign access to subscription

Your application now has a token to access Azure Resource Manager on behalf of the user. The next step is to connect your app to the subscription. After connecting, your app can manage those subscriptions even when the user isn't present (long-term offline access).

For each subscription to connect, call the [Resource Manager list permissions](#) API to determine whether the user has access management rights for the subscription.

The [UserCanManagerAccessForSubscription](#) method of the ASP.NET MVC sample app implements this call.

The following example shows how to request a user's permissions on a subscription. 83cfe939-2402-4581-b761-

4f59b0a041e4 is the id of the subscription.

```
GET https://management.azure.com/subscriptions/83cfe939-2402-4581-b761-  
4f59b0a041e4/providers/microsoft.authorization/permissions?api-version=2015-07-01 HTTP/1.1  
  
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGwiOiJlbW1tMm7Cw6JWtfY2lGc5A
```

An example of the response to get user's permissions on subscription is:

```
HTTP/1.1 200 OK  
  
{"value": [{"actions": ["*"], "notActions": ["Microsoft.Authorization/*/Write", "Microsoft.Authorization/*/Delete"]}, {"actions": ["/read"], "notActions": []}]}  
  
The permissions API returns multiple permissions. Each permission consists of allowed actions (actions) and disallowed actions (notactions). If an action is present in the allowed actions list of any permission and not present in the notactions list of that permission, the user is allowed to perform that action.  
microsoft.authorization/roleassignments/write is the action that grants access management rights. Your application must parse the permissions result to look for a regex match on this action string in the actions and notactions of each permission.
```

Get app-only access token

Now, you know if the user can assign access to the Azure subscription. The next steps are:

1. Assign the appropriate RBAC role to your application's identity on the subscription.
2. Validate the access assignment by querying for the Application's permission on the subscription or by accessing Resource Manager using app-only token.
3. Record the connection in your applications "connected subscriptions" data structure - persisting the id of the subscription.

Let's look closer at the first step. To assign the appropriate RBAC role to the application's identity, you must determine:

- The object id of your application's identity in the user's Azure Active Directory
- The identifier of the RBAC role that your application requires on the subscription

When your application authenticates a user from an Azure AD, it creates a service principal object for your application in that Azure AD. Azure allows RBAC roles to be assigned to service principals to grant direct access to corresponding applications on Azure resources. This action is exactly what we wish to do. Query the Azure AD Graph API to determine the identifier of the service principal of your application in the signed-in user's Azure AD.

You only have an access token for Azure Resource Manager - you need a new access token to call the Azure AD Graph API. Every application in Azure AD has permission to query its own service principal object, so an app-only access token is sufficient.

Get app-only access token for Azure AD Graph API

To authenticate your app and get a token to Azure AD Graph API, issue a Client Credential Grant OAuth2.0 flow token request to Azure AD token endpoint

(https://login.microsoftonline.com/{directory_domain_name}/OAuth2/Token).

The [GetObjectOfServicePrincipalInOrganization](#) method of the ASP.net MVC sample application gets an app-only access token for Graph API using the Active Directory Authentication Library for .NET.

The query string parameters that are available for this request are described in the [Request an Access Token](#) topic.

An example request for client credential grant token:

```
POST https://login.microsoftonline.com/62e173e9-301e-423e-bcd4-29121ec1aa24/oauth2/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 187</pre>
<pre>grant_type=client_credentials&client_id=a0448380-c346-4f9f-b897-
c18733de9394&resource=https%3A%2F%2Fgraph.windows.net%2F &client_secret=olna8C*****0g%3D
```

An example response for client credential grant token:

```
HTTP/1.1 200 OK
```

```
{"token_type": "Bearer", "expires_in": "3599", "expires_on": "1432039862", "not_before": "1432035962", "resource": "https://graph.windows.net/", "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNBVGZNNXBPWW1KS E1iYTlnb0VLWSISImtpZCI6Ik1uQ19WVmNBVGZNNXBPWW1KSE1iYTlnb0VLWSJ9.eyJhdWQiOiJodHRwczovL2dyYXBoLndpbmRv****G5gUTV -kKorR-pg"}
```

Get ObjectId of application service principal in user Azure AD

Now, use the app-only access token to query the [Azure AD Graph Service Principals](#) API to determine the Object Id of the application's service principal in the directory.

The [GetObjectIdOfServicePrincipalInOrganization](#) method of the ASP.net MVC sample application implements this call.

The following example shows how to request an application's service principal. a0448380-c346-4f9f-b897-c18733de9394 is the client id of the application.

```
GET https://graph.windows.net/62e173e9-301e-423e-bcd4-29121ec1aa24/servicePrincipals?api-version=1.5&$filter=appId%20eq%20'a0448380-c346-4f9f-b897-c18733de9394' HTTP/1.1
Authorization: Bearer eyJ0eXAiOiJK*****-kKorR-pg
```

The following example shows a response to the request for an application's service principal

```
HTTP/1.1 200 OK
```

```
{"odata.metadata": "https://graph.windows.net/62e173e9-301e-423e-bcd4-29121ec1aa24/$metadata#directoryObjects/Microsoft.DirectoryServices.ServicePrincipal", "value": [
  {
    "odata.type": "Microsoft.DirectoryServices.ServicePrincipal", "objectType": "ServicePrincipal", "objectId": "9b5018d4-6951-42ed-8a92-f11ec283ccce", "deletionTimestamp": null, "accountEnabled": true, "appDisplayName": "CloudSense", "appId": "a0448380-c346-4f9f-b897-c18733de9394", "appOwnerTenantId": "62e173e9-301e-423e-bcd4-29121ec1aa24", "appRoleAssignmentRequired": false, "appRoles": [
      {"displayName": "CloudSense", "errorUrl": null, "homepage": "http://www.vipswapper.com/cloudsense", "keyCredential": [{"s": []}], "logoutUrl": null, "oauth2Permissions": [{"adminConsentDescription": "Allow the application to access CloudSense on behalf of the signed-in user.", "adminConsentDisplayName": "Access CloudSense", "id": "b7b7338e-683a-4796-b95e-60c10380de1c", "isEnabled": true, "type": "User", "userConsentDescription": "Allow the application to access CloudSense on your behalf.", "userConsentDisplayName": "Access CloudSense", "value": "user_impersonation"}]}, {"passwordCredentials": [], "preferredTokenSigningKeyThumbprint": null, "publisherName": "vipswapper"quot;, "replyUrls": ["http://www.vipswapper.com/cloudsense", "http://www.vipswapper.com", "http://vipswapper.com", "http://vipswapper.azurewebsites.net", "http://localhost:62080"], "samlMetadataUrl": null, "servicePrincipalNames": ["http://www.vipswapper.com/cloudsense", "a0448380-c346-4f9f-b897-c18733de9394"], "tags": ["WindowsAzureActiveDirectoryIntegratedApp"]}]}}
```

Get Azure RBAC role identifier

To assign the appropriate RBAC role to your service principal, you must determine the identifier of the Azure RBAC role.

The right RBAC role for your application:

- If your application only monitors the subscription, without making any changes, it requires only reader permissions on the subscription. Assign the **Reader** role.
- If your application manages Azure the subscription, creating/modifying/deleting entities, it requires one of the contributor permissions.
 - To manage a particular type of resource, assign the resource-specific contributor roles (Virtual Machine Contributor, Virtual Network Contributor, Storage Account Contributor, etc.)
 - To manage any resource type, assign the **Contributor** role.

The role assignment for your application is visible to users, so select the least-required privilege.

Call the [Resource Manager role definition API](#) to list all Azure RBAC roles and search then iterate over the result to find the desired role definition by name.

The [GetRoleID](#) method of the ASP.net MVC sample app implements this call.

The following request example shows how to get Azure RBAC role identifier. 09cbd307-aa71-4aca-b346-5f253e6e3ebb is the id of the subscription.

```
GET https://management.azure.com/subscriptions/09cbd307-aa71-4aca-b346-  
5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions?api-version=2015-07-01 HTTP/1.1  
  
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaXNzIjoi  
https://management.azure.com/subscriptions/09cbd307-aa71-4aca-b346-5f253e6e3ebb/providers/  
Microsoft.Authorization/roleDefinitions/312a565d-c81f-4fd8-895a-4e21e48d571c&lt;br&ampgt  
.type: "Microsoft.Authorization/roleDefinitions", "name": "312a565d-c81f-4fd8-895a-4e21e48d571c"},  
{ "properties": { "roleName": "Application Insights Component  
Contributor", "type": "BuiltInRole", "description": "Lets you manage Application Insights components, but not  
access to them.", "scope": "/", "permissions": [ { "actions":  
[ "Microsoft.Insights/components/*", "Microsoft.Insights/webtests/*", "Microsoft.Authorization/*/read", "Microsoft.  
Resources/subscriptions/resources/read", "Microsoft.Resources/subscriptions/resourceGroups/read", "Microsoft.Re  
sources/subscriptions/resourceGroups/resources/read", "Microsoft.Resources/subscriptions/resourceGroups/deploy  
ments/*", "Microsoft.Insights/alertRules/*", "Microsoft.Support/*"], "notActions": [] } ], "id": "/subscriptions/09cbd307-aa71-4aca-b346-  
5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions/ae349356-3a1b-4a5e-921d-  
050484c6347e", "type": "Microsoft.Authorization/roleDefinitions", "name": "ae349356-3a1b-4a5e-921d-  
050484c6347e"} }
```

The response is in the following format:

```
HTTP/1.1 200 OK  
  
{"value": [{"properties": {"roleName": "API Management Service  
Contributor", "type": "BuiltInRole", "description": "Lets you manage API Management services, but not access to  
them.", "scope": "/", "permissions": [{"actions":  
[ "Microsoft.ApiManagement/Services/*", "Microsoft.Authorization/*/read", "Microsoft.Resources/subscriptions/reso  
urces/read", "Microsoft.Resources/subscriptions/resourceGroups/read", "Microsoft.Resources/subscriptions/resourc  
eGroups/resources/read", "Microsoft.Resources/subscriptions/resourceGroups/deployments/*", "Microsoft.Insights/a  
lertRules/*", "Microsoft.Support/*"], "notActions": [] } ], "id": "/subscriptions/09cbd307-aa71-4aca-b346-  
5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions/312a565d-c81f-4fd8-895a-  
4e21e48d571c", "type": "Microsoft.Authorization/roleDefinitions", "name": "312a565d-c81f-4fd8-895a-4e21e48d571c"},  
, {"properties": {"roleName": "Application Insights Component  
Contributor", "type": "BuiltInRole", "description": "Lets you manage Application Insights components, but not  
access to them.", "scope": "/", "permissions": [{"actions":  
[ "Microsoft.Insights/components/*", "Microsoft.Insights/webtests/*", "Microsoft.Authorization/*/read", "Microsoft.  
Resources/subscriptions/resources/read", "Microsoft.Resources/subscriptions/resourceGroups/read", "Microsoft.Re  
sources/subscriptions/resourceGroups/resources/read", "Microsoft.Resources/subscriptions/resourceGroups/deploy  
ments/*", "Microsoft.Insights/alertRules/*", "Microsoft.Support/*"], "notActions": [] } ], "id": "/subscriptions/09cbd307-aa71-4aca-b346-  
5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions/ae349356-3a1b-4a5e-921d-  
050484c6347e", "type": "Microsoft.Authorization/roleDefinitions", "name": "ae349356-3a1b-4a5e-921d-  
050484c6347e"} ]}
```

You do not need to call this API on an ongoing basis. Once you've determined the well-known GUID of the role definition, you can construct the role definition id as:

```
/subscriptions/{subscription_id}/providers/Microsoft.Authorization/roleDefinitions/{well-known-role-guid}
```

Here are the well-known guids of commonly used built-in roles:

ROLE	GUID
Reader	acdd72a7-3385-48ef-bd42-f606fba81ae7
Contributor	b24988ac-6180-42a0-ab88-20f7382dd24c
Virtual Machine Contributor	d73bb868-a0df-4d4d-bd69-98a00b01fccb
Virtual Network Contributor	b34d265f-36f7-4a0d-a4d4-e158ca92e90f
Storage Account Contributor	86e8f5dc-a6e9-4c67-9d15-de283e8eac25
Website Contributor	de139f84-1756-47ae-9be6-808fbe84772
Web Plan Contributor	2cc479cb-7b4d-49a8-b449-8c00fd0f0a4b
SQL Server Contributor	6d8ee4ec-f05a-4a1d-8b00-a9b17e38b437
SQL DB Contributor	9b7fa17d-e63e-47b0-bb0a-15c516ac86ec

Assign RBAC role to application

You have everything you need to assign the appropriate RBAC role to your service principal by using the [Resource Manager create role assignment](#) API.

The [GrantRoleToServicePrincipalOnSubscription](#) method of the ASP.net MVC sample app implements this call.

An example request to assign RBAC role to application:

```
PUT https://management.azure.com/subscriptions/09cbd307-aa71-4aca-b346-
5f253e6e3ebb/providers/microsoft.authorization/roleassignments/4f87261d-2816-465d-8311-70a27558df4c?api-
version=2015-07-01 HTTP/1.1

Authorization: Bearer eyJ0eXAiOiJKV1QiL*****FlwO1mM7Cw6JWtfY2lGc5
Content-Type: application/json
Content-Length: 230

{"properties": {"roleDefinitionId":"/subscriptions/09cbd307-aa71-4aca-b346-
5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions/acdd72a7-3385-48ef-bd42-
f606fba81ae7","principalId":"c3097b31-7309-4c59-b4e3-770f8406bad2"}}
```

In the request, the following values are used:

GUID	DESCRIPTION
09cbd307-aa71-4aca-b346-5f253e6e3ebb	the id of the subscription
c3097b31-7309-4c59-b4e3-770f8406bad2	the object id of the service principal of the application
acdd72a7-3385-48ef-bd42-f606fba81ae7	the id of the reader role
4f87261d-2816-465d-8311-70a27558df4c	a new guid created for the new role assignment

The response is in the following format:

```
HTTP/1.1 201 Created
```

```
{"properties": {"roleDefinitionId": "/subscriptions/09cbd307-aa71-4aca-b346-5f253e6e3ebb/providers/Microsoft.Authorization/roleDefinitions/acdd72a7-3385-48ef-bd42-f606fba81ae7", "principalId": "c3097b31-7309-4c59-b4e3-770f8406bad2", "scope": "/subscriptions/09cbd307-aa71-4aca-b346-5f253e6e3ebb"}, "id": "/subscriptions/09cbd307-aa71-4aca-b346-5f253e6e3ebb/providers/Microsoft.Authorization/roleAssignments/4f87261d-2816-465d-8311-70a27558df4c", "type": "Microsoft.Authorization/roleAssignments", "name": "4f87261d-2816-465d-8311-70a27558df4c"}
```

Get app-only access token for Azure Resource Manager

To validate that app has the desired access on the subscription, perform a test task on the subscription using an app-only token.

To get an app-only access token, follow instructions from section [Get app-only access token for Azure AD Graph API](#), with a different value for the resource parameter:

```
https://management.core.windows.net/
```

The [ServicePrincipalHasReadAccessToSubscription](#) method of the ASP.NET MVC sample application gets an app-only access token for Azure Resource Manager using the Active Directory Authentication Library for .net.

Get Application's Permissions on Subscription

To check that your application has the desired access on an Azure subscription, you may also call the [Resource Manager Permissions](#) API. This approach is similar to how you determined whether the user has Access Management rights for the subscription. However, this time, call the permissions API with the app-only access token that you received in the previous step.

The [ServicePrincipalHasReadAccessToSubscription](#) method of the ASP.NET MVC sample app implements this call.

Manage connected subscriptions

When the appropriate RBAC role is assigned to your application's service principal on the subscription, your application can keep monitoring/managing it using app-only access tokens for Azure Resource Manager.

If a subscription owner removes your application's role assignment using the classic portal or command-line tools, your application is no longer able to access that subscription. In that case, you should notify the user that the connection with the subscription was severed from outside the application and give them an option to "repair" the connection. "Repair" would simply re-create the role assignment that was deleted offline.

Just as you enabled the user to connect subscriptions to your application, you must allow the user to disconnect subscriptions too. From an access management point of view, disconnect means removing the role assignment that the application's service principal has on the subscription. Optionally, any state in the application for the subscription might be removed too. Only users with access management permission on the subscription are able to disconnect the subscription.

The [RevokeRoleFromServicePrincipalOnSubscription](#) method of the ASP.net MVC sample app implements this call.

That's it - users can now easily connect and manage their Azure subscriptions with your application.

Lock resources to prevent unexpected changes

1/24/2017 • 4 min to read • [Edit Online](#)

As an administrator, you may need to lock a subscription, resource group, or resource to prevent other users in your organization from accidentally deleting or modifying critical resources. You can set the lock level to **CanNotDelete** or **ReadOnly**.

- **CanNotDelete** means authorized users can still read and modify a resource, but they can't delete the resource.
- **ReadOnly** means authorized users can read a resource, but they can't delete or update the resource. Applying this lock is similar to restricting all authorized users to the permissions granted by the **Reader** role.

Resource Manager locks apply only to operations that happen in the management plane, which consists of operations sent to <https://management.azure.com>. The locks do not restrict how resources perform their own functions. Resource changes are restricted, but resource operations are not restricted. For example, a **ReadOnly** lock on a SQL Database prevents you from deleting or modifying the database, but it does not prevent you from creating, updating, or deleting data in the database. Data transactions are permitted because those operations are not sent to <https://management.azure.com>.

Applying **ReadOnly** can lead to unexpected results because some operations that seem like read operations actually require additional actions. For example, placing a **ReadOnly** lock on a storage account prevents all users from listing the keys. The list keys operation is handled through a POST request because the returned keys are available for write operations. For another example, placing a **ReadOnly** lock on an App Service resource prevents Visual Studio Server Explorer from displaying files for the resource because that interaction requires write access.

Unlike role-based access control, you use management locks to apply a restriction across all users and roles. To learn about setting permissions for users and roles, see [Azure Role-based Access Control](#).

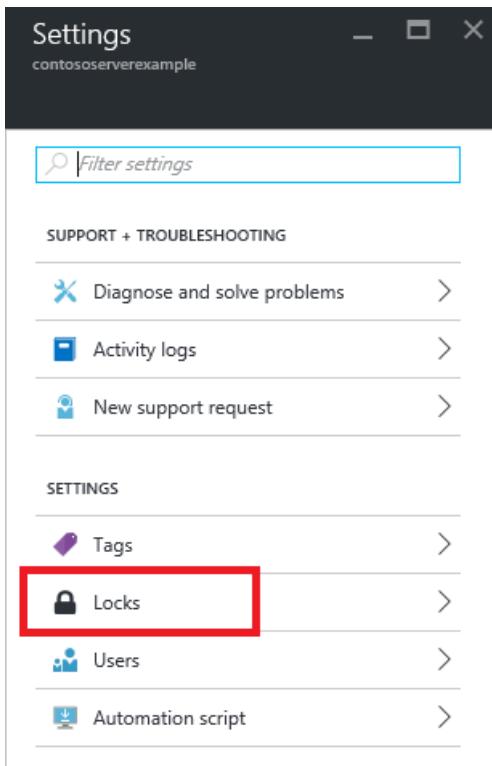
When you apply a lock at a parent scope, all resources within that scope inherit the same lock. Even resources you add later inherit the lock from the parent. The most restrictive lock in the inheritance takes precedence.

Who can create or delete locks in your organization

To create or delete management locks, you must have access to `Microsoft.Authorization/*` or `Microsoft.Authorization/locks/*` actions. Of the built-in roles, only **Owner** and **User Access Administrator** are granted those actions.

Creating a lock through the portal

1. In the Settings blade for the resource, resource group, or subscription that you wish to lock, select **Locks**.



2. To add a lock, select **Add**. If you want to create a lock at a parent level, select the parent. The currently selected resource inherits the lock from the parent. For example, you could lock the resource group to apply a lock to all its resources.

A screenshot of the 'Management locks' page. At the top, it says 'Management locks' and 'contososerverexample'. There are four buttons: '+ Add' (highlighted with a red box), 'Resource group', 'Subscription', and 'Refresh'. Below them is a table with columns 'LOCK NAME', 'LOCK TYPE', 'SCOPE', and 'NOTES'. A message at the bottom says 'This resource has no locks.'

3. Give the lock a name and lock level. Optionally, you can add notes that describe the lock.

A screenshot of the 'Add lock' dialog box. It has a header 'Management locks' and 'contososerverexample' with buttons '+ Add', 'Resource group', 'Subscription', and 'Refresh'. The main area is titled 'Add lock'. It has two input fields: 'Lock name' containing 'DatabaseServerLock' with a green checkmark, and 'Lock type' with a dropdown menu showing 'Delete'. Below that is a 'Notes' section with a text input field containing 'Prevent deleting the database server'. At the bottom are 'OK' and 'Cancel' buttons.

4. To delete the lock, select the ellipsis and **Delete** from the available options.

Creating a lock in a template

The following example shows a template that creates a lock on a storage account. The storage account on which to apply the lock is provided as a parameter. The name of the lock is created by concatenating the resource name with **/Microsoft.Authorization/** and the name of the lock, in this case **myLock**.

The type provided is specific to the resource type. For storage, set the type to "Microsoft.Storage/storageaccounts/providers/locks".

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "lockedResource": {
      "type": "string"
    }
  },
  "resources": [
    {
      "name": "[concat(parameters('lockedResource'), '/Microsoft.Authorization/myLock')]",
      "type": "Microsoft.Storage/storageAccounts/providers/locks",
      "apiVersion": "2015-01-01",
      "properties": {
        "level": "CannotDelete"
      }
    }
  ]
}
```

Creating a lock with REST API

You can lock deployed resources with the [REST API for management locks](#). The REST API enables you to create and delete locks, and retrieve information about existing locks.

To create a lock, run:

```
PUT https://management.azure.com/{scope}/providers/Microsoft.Authorization/locks/{lock-name}?api-version={api-version}
```

The scope could be a subscription, resource group, or resource. The lock-name is whatever you want to call the lock. For api-version, use **2015-01-01**.

In the request, include a JSON object that specifies the properties for the lock.

```
{  
  "properties": {  
    "level": "CanNotDelete",  
    "notes": "Optional text notes."  
  }  
}
```

Creating a lock with Azure PowerShell

You can lock deployed resources with Azure PowerShell by using the **New-AzureRmResourceLock** as shown in the following example.

```
New-AzureRmResourceLock -LockLevel CanNotDelete -LockName LockSite -ResourceName examplesite -ResourceType  
Microsoft.Web/sites -ResourceGroupName exempleresourcegroup
```

Azure PowerShell provides other commands for working locks, such as **Set-AzureRmResourceLock** to update a lock and **Remove-AzureRmResourceLock** to delete a lock.

Next steps

- For more information about working with resource locks, see [Lock Down Your Azure Resources](#)
- To learn about logically organizing your resources, see [Using tags to organize your resources](#)
- To change which resource group a resource resides in, see [Move resources to new resource group](#)
- You can apply restrictions and conventions across your subscription with customized policies. For more information, see [Use Policy to manage resources and control access](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Security considerations for Azure Resource Manager

1/17/2017 • 19 min to read • [Edit Online](#)

When looking at aspects of security for your Azure Resource Manager templates, there are several areas to consider – keys and secrets, role-based access control, and network security groups.

This topic assumes you are familiar with Role-Based Access Control (RBAC) in Azure Resource Manager. For more information, see [Azure Role-based Access Control](#).

This topic is part of a larger whitepaper. To read the full paper, download [World Class ARM Templates Considerations and Proven Practices](#).

Secrets and certificates

Azure Virtual Machines, Azure Resource Manager and Azure Key Vault are fully integrated to provide support for the secure handling of certificates which are to be deployed in the VM. Utilizing Azure Key Vault with Resource Manager to orchestrate and store VM secrets and certificates is a best practice and provides the following advantages:

- The templates only contain URI references to the secrets, which means the actual secrets are not in code, configuration or source code repositories. This prevents key phishing attacks on internal or external repos, such as harvest-bots in GitHub.
- Secrets stored in the Key Vault are under full RBAC control of a trusted operator. If the trusted operator leaves the company or transfers within the company to a new group, they no longer have access to the keys they created in the Vault.
- Full compartmentalization of all assets:
 - the templates to deploy the keys
 - the templates to deploy a VM with references to the keys
 - the actual key materials in the Vault.

Each template (and action) can be under different RBAC roles for full separation of duties.
- The loading of secrets into a VM at deployment time occurs via direct channel between the Azure Fabric and the Key Vault within the confines of the Microsoft datacenter. Once the keys are in the Key Vault, they never see 'daylight' over an untrusted channel outside of the datacenter.
- Key Vaults are always regional, so the secrets always have locality (and sovereignty) with the VMs. There are no global Key Vaults.

Separation of keys from deployments

A best practice is to maintain separate templates for:

1. Creation of vaults (which will contain the key material)
2. Deployment of the VMs (with URI references to the keys contained in the vaults)

A typical enterprise scenario is to have a small group of Trusted Operators who have access to critical secrets within the deployed workloads, with a broader group of dev/ops personnel who can create or update VM deployments. Below is an example ARM template which creates and configures a new vault in the context of the currently authenticated user's identity in Azure Active Directory. This user would have the default permission to create, delete, list, update, backup, restore, and get the public half of keys in this new key vault.

While most of the fields in this template should be self-explanatory, the **enableVaultForDeployment** setting deserves more background: vaults do not have any default standing access by any other Azure infrastructure

component. By setting this value, it allows the Azure Compute infrastructure components read-only access to this specific named vault. Therefore, a further best practice is to not comingle corporate sensitive data in the same vault as virtual machine secrets.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "keyVaultName": {
            "type": "string",
            "metadata": {
                "description": "Name of the Vault"
            }
        },
        "location": {
            "type": "string",
            "allowedValues": ["East US", "West US", "West Europe", "East Asia", "South East Asia"],
            "metadata": {
                "description": "Location of the Vault"
            }
        },
        "tenantId": {
            "type": "string",
            "metadata": {
                "description": "Tenant Id of the subscription. Get using Get-AzureSubscription cmdlet or Get Subscription API"
            }
        },
        "objectId": {
            "type": "string",
            "metadata": {
                "description": "Object Id of the AD user. Get using Get-AzureADUser cmdlet"
            }
        },
        "skuName": {
            "type": "string",
            "allowedValues": ["Standard", "Premium"],
            "metadata": {
                "description": "SKU for the vault"
            }
        },
        "enableVaultForDeployment": {
            "type": "bool",
            "allowedValues": [true, false],
            "metadata": {
                "description": "Specifies if the vault is enabled for a VM deployment"
            }
        }
    },
    "resources": [
        {
            "type": "Microsoft.KeyVault/vaults",
            "name": "[parameters('keyVaultName')]",
            "apiVersion": "2014-12-19-preview",
            "location": "[parameters('location')]",
            "properties": {
                "enabledForDeployment": "[parameters('enableVaultForDeployment')]",
                "tenantid": "[parameters('tenantId')]",
                "accessPolicies": [
                    {
                        "tenantId": "[parameters('tenantId')]",
                        "objectId": "[parameters('objectId')]",
                        "permissions": {
                            "secrets": ["all"],
                            "keys": ["all"]
                        }
                    }
                ],
                "sku": {
                    "name": "[parameters('skuName')]"
                }
            }
        }
    ]
}
```

```

        "family": "A"
    }
}
}]
}

```

Once the vault is created, the next step is to reference that vault in the deployment template of a new VM. As mentioned above, a best practice is to have a different dev/ops group manage VM deployments, with that group having no direct access to the keys as stored in the vault.

The below template fragment would be composed into higher order deployment constructs, each safely and securely referencing highly-sensitive secrets which are not under the direct control of the operator.

```

"vaultName": {
    "type": "string",
    "metadata": {
        "description": "Name of Key Vault that has a secret"
    }
},
{
    "apiVersion": "2015-05-01-preview",
    "type": "Microsoft.Compute/virtualMachines",
    "name": "[parameters('vmName')]",
    "location": "[parameters('location')]",
    "properties": {
        "osProfile": {
            "secrets": [
                {
                    "sourceVault": {
                        "id": "[resourceId('vaultrg', 'Microsoft.KeyVault/vaults', 'kayvault')]"
                    },
                    "vaultCertificates": [
                        {
                            "certificateUrl": "[parameters('secretUrlWithVersion')]",
                            "certificateStore": "My"
                        }
                    ]
                }
            ]
        }
    }
}

```

To pass a value from a key vault as a parameter during deployment of a template, see [Pass secure values during deployment](#).

Service principals for cross-subscription interactions

Service identities are represented by service principals in Active Directory. Service principals will be at the center of enabling key scenarios for Enterprise IT organizations, System Integrators (SI), and Cloud Service Vendors (CSV). Specifically, there will be use cases where one of these organizations will need to interact with the subscription of one of their customers.

Your organization could provide an offering that will monitor a solution deployed in your customers environment and subscription. In this case, you will need to get access to logs and other data within a customers account so that you can utilize it in your monitoring solution. If you're a corporate IT organization, a systems integrator, you may provide an offering to a customer where you will deploy and manage a capability for them, such as a data analytics platform, where the offering resides in the customers own subscription.

In these use cases, your organization would require an identity that could be given access to perform these actions within the context of a customer subscription.

These scenarios bring with them a certain set of considerations for your customer:

- For security reasons, access may need to be scoped to certain types of actions, e.g. read only access.

- As deployed resources are provided at a cost, there may be similar constraints on access required for financial reasons.
- For security reasons, access may need to be scoped only to a specific resource (storage accounts) or resources (resource group containing an environment or solution)
- As a relationship with a vendor may change, the customer will want to have the ability to enable/disable access to SI or CSV
- As actions against this account having billing implications, the customer desires support for auditability and accountability for billing.
- From a compliance perspective, the customer will want to be able to audit your behavior within their environment

A combination of a service principal and RBAC can be used to address these requirements.

Network security groups

Many scenarios will have requirements that specify how traffic to one or more VM instances in your virtual network is controlled. You can use a Network Security Group (NSG) to do this as part of an ARM template deployment.

A network security group is a top-level object that is associated with your subscription. An NSG contains access control rules that allow or deny traffic to VM instances. The rules of an NSG can be changed at any time, and changes are applied to all associated instances. To use an NSG, you must have a virtual network that is associated with a region (location). NSGs are not compatible with virtual networks that are associated with an affinity group. If you don't have a regional virtual network and you want to control traffic to your endpoints, please see [About Network Access Control Lists \(ACLs\)](#).

You can associate an NSG with a VM, or to a subnet within a virtual network. When associated with a VM, the NSG applies to all the traffic that is sent and received by the VM instance. When applied to a subnet within your virtual network, it applies to all the traffic that is sent and received by all the VM instances in the subnet. A VM or subnet can be associated with only 1 NSG, but each NSG can contain up to 200 rules. You can have 100 NSGs per subscription.

NOTE

Endpoint-based ACLs and network security groups are not supported on the same VM instance. If you want to use an NSG and have an endpoint ACL already in place, first remove the endpoint ACL. For information about how to do this, see [Managing Access Control Lists \(ACLs\) for Endpoints by using PowerShell](#).

How network security groups work

Network security groups are different than endpoint-based ACLs. Endpoint ACLs work only on the public port that is exposed through the Input endpoint. An NSG works on one or more VM instances and controls all the traffic that is inbound and outbound on the VM.

A network security group has a *Name*, is associated with a *Region* (one of the supported Azure locations), and has a descriptive label. It contains two types of rules, Inbound and Outbound. The Inbound rules are applied on the incoming packets to a VM and the Outbound rules are applied to the outgoing packets from the VM. The rules are applied at the server machine where the VM is located. An incoming or outgoing packet must match an Allow rule to be permitted; otherwise, it's dropped.

Rules are processed in the order of priority. For example, a rule with a lower priority number such as 100 is processed before rules with a higher priority numbers such as 200. Once a match is found, no more rules are processed.

A rule specifies the following:

- Name: A unique identifier for the rule
- Type: Inbound/Outbound
- Priority: An integer between 100 and 4096 (rules processed from low to high)
- Source IP Address: CIDR of source IP range
- Source Port Range: An integer or range between 0 and 65536
- Destination IP Range: CIDR of the destination IP Range
- Destination Port Range: An integer or range between 0 and 65536
- Protocol: TCP, UDP or '*'
- Access: Allow/Deny

Default rules

An NSG contains default rules. The default rules can't be deleted, but because they are assigned the lowest priority, they can be overridden by the rules that you create. The default rules describe the default settings recommended by the platform. As illustrated by the default rules below, traffic originating and ending in a virtual network is allowed both in Inbound and Outbound directions.

While connectivity to the Internet is allowed for outbound direction, it is by default blocked for inbound direction. A default rule allows the Azure load balancer to probe the health of a VM. You can override this rule if the VM or set of VMs under the NSG does not participate in the load balanced set.

The default rules are shown in the tables below.

Inbound default rules

NAME	PRIORITY	SOURCE IP	SOURCE PORT	DESTINATION IP	DESTINATION PORT	PROTOCOL	ACCESS
ALLOW VNET INBOUND	65000	VIRTUAL_NETWORK	*	VIRTUAL_NETWORK	*	*	ALLOW
ALLOW AZURE LOAD BALANCER INBOUND	65001	AZURE_LOADBALANCER	*	*	*	*	ALLOW
DENY ALL INBOUND	65500	*	*	*	*	*	DENY

Outbound default rules

NAME	PRIORITY	SOURCE IP	SOURCE PORT	DESTINATION IP	DESTINATION PORT	PROTOCOL	ACCESS
ALLOW VNET OUTBOUND	65000	VIRTUAL_NETWORK	*	VIRTUAL_NETWORK	*	*	ALLOW
ALLOW INTERNET OUTBOUND	65001	*	*	INTERNET	*	*	ALLOW

Name	Priority	Source IP	Source Port	Destination IP	Destination Port	Protocol	Access
DENY ALL OUTBOUND	65500	*	*	*	*	*	DENY

Special infrastructure rules

NSG rules are explicit. No traffic is allowed or denied beyond what is specified in the NSG rules. However, two types of traffic are always allowed regardless of the Network Security group specification. These provisions are made to support the infrastructure:

- **Virtual IP of the Host Node:** Basic infrastructure services such as DHCP, DNS, and Health monitoring are provided through the virtualized host IP address 168.63.129.16. This public IP address belongs to Microsoft and will be the only virtualized IP address used in all regions for this purpose. This IP address maps to the physical IP address of the server machine (host node) hosting the VM. The host node acts as the DHCP relay, the DNS recursive resolver, and the probe source for the load balancer health probe and the machine health probe. Communication to this IP address should not be considered as an attack.
- **Licensing (Key Management Service):** Windows images running in the VMs should be licensed. To do this, a licensing request is sent to the Key Management Service host servers that handle such queries. This will always be on outbound port 1688.

Default tags

Default tags are system-provided identifiers to address a category of IP addresses. Default tags can be specified in user-defined rules.

Default tags for NSGs

TAG	DESCRIPTION
VIRTUAL_NETWORK	Denotes all of your network address space. It includes the virtual network address space (IP CIDR in Azure) as well as all connected on-premises address space (Local Networks). This also includes virtual network-to-virtual network address spaces.
AZURE_LOADBALANCER	Denotes the Azure Infrastructure load balancer and will translate to an Azure datacenter IP where Azure's health probes will originate. This is needed only if the VM or set of VMs associated with the NSG is participating in a load balanced set.
INTERNET	Denotes the IP address space that is outside the virtual network and can be reached by public Internet. This range includes Azure-owned public IP space as well.

Ports and port ranges

NSG rules can be specified on a single source or destination port, or on a port range. This approach is particularly useful when you want to open a wide range of ports for an application, such as FTP. The range must be sequential and can't be mixed with individual port specifications. To specify a range of ports, use the hyphen (-) character. For example, **100-500**.

ICMP traffic

With the current NSG rules, you can specify TCP or UDP as protocols but not ICMP. However, ICMP traffic is allowed within a virtual network by default through the Inbound rules that support traffic from and to any port and

protocol (*) within the virtual network.

Associating an NSG with a VM

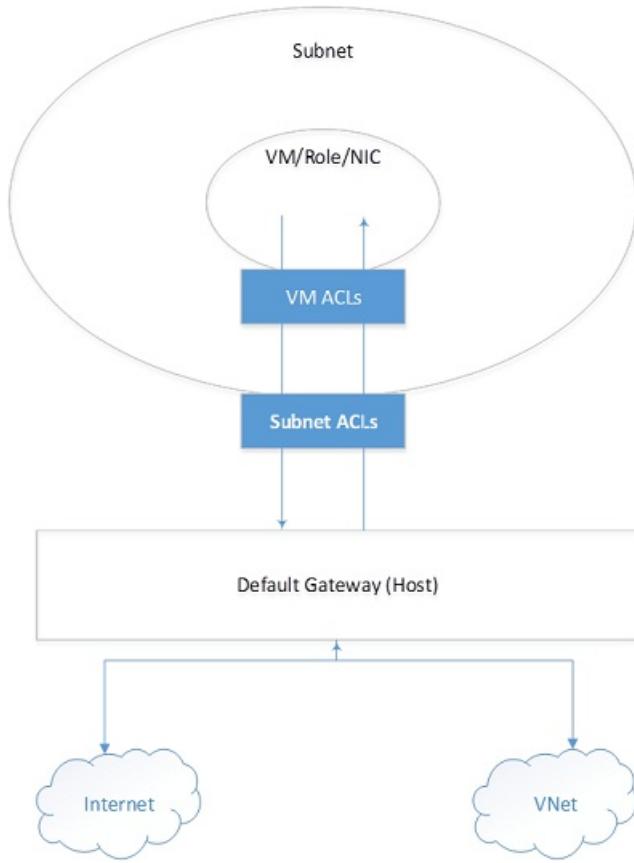
When an NSG is directly associated with a VM, the network access rules in the NSG are directly applied to all traffic that is destined to the VM. Whenever the NSG is updated for rule changes, the traffic handling reflects the updates within minutes. When the NSG is disassociated from the VM, the state reverts to its pre-NSG condition—that is, to the system defaults before the NSG was introduced.

Associating an NSG with a subnet

When an NSG is associated with a subnet, the network access rules in the NSG are applied to all the VMs in the subnet. Whenever the access rules in the NSG are updated, the changes are applied to all VMs in the subnet within minutes.

Associating an NSG with a subnet and a VM

You can associate one NSG with a VM and another NSG with the subnet where the VM resides. This scenario is supported to provide the VM with two layers of protection. On the inbound traffic, the packet follows the access rules specified in the subnet, followed by rules in the VM. When outbound, the packet follows the rules specified in the VM first, then follows the rules specified in the subnet as shown below.



When an NSG is associated with a VM or subnet, the network access control rules become very explicit. The platform will not insert any implicit rule to allow traffic to a particular port. In this case, if you create an endpoint in the VM, you must also create a rule to allow traffic from the Internet. If you don't do this, the `VIP:{Port}` can't be accessed from outside.

For example, you can create a new VM and a new NSG. You associate the NSG with the VM. The VM can communicate with other VMs in the virtual network through the ALLOW VNET INBOUND rule. The VM can also make outbound connections to the Internet using the ALLOW INTERNET OUTBOUND rule. Later, you create an endpoint on port 80 to receive traffic to your website running in the VM. Packets destined to port 80 on the VIP (public Virtual IP address) from the Internet will not reach the VM until you add a rule similar to the following table to the NSG.

Explicit rule allowing traffic to a particular port

NAME	PRIORITY	SOURCE IP	SOURCE PORT	DESTINATION IP	DESTINATION PORT	PROTOCOL	ACCESS
WEB	100	INTERNET	*	*	80	TCP	ALLOW

User-defined routes

Azure uses a route table to decide how to forward IP traffic based on the destination of each packet. Although Azure provides a default route table based on your virtual network settings, you may need to add custom routes to that table.

The most common need for a custom entry in the route table is the use of a virtual appliance in your Azure environment. Take into account the scenario shown in the Figure below. Suppose you want to ensure that all traffic directed to the mid-tier and backed subnets initiated from the front end subnet go through a virtual firewall appliance. Simply adding the appliance to your virtual network and connecting it to the different subnets will not provide this functionality. You must also change the routing table applied to your subnet to ensure packets are forwarded to the virtual firewall appliance.

The same would be true if you implemented a virtual NAT appliance to control traffic between your Azure virtual network and the Internet. To ensure the virtual appliance is used you have to create a route specifying that all traffic destined to the Internet must be forwarded to the virtual appliance.

Routing

Packets are routed over a TCP/IP network based on a route table defined at each node on the physical network. A route table is a collection of individual routes used to decide where to forward packets based on the destination IP address. A route consists of the following:

- Address Prefix. The destination CIDR to which the route applies, such as 10.1.0.0/16.
- Next hop type. The type of Azure hop the packet should be sent to. Possible values are:
 - Local. Represents the local virtual network. For instance, if you have two subnets, 10.1.0.0/16 and 10.2.0.0/16 in the same virtual network, the route for each subnet in the route table will have a next hop value of Local.
 - VPN Gateway. Represents an Azure S2S VPN Gateway.
 - Internet. Represents the default Internet gateway provided by the Azure Infrastructure
 - Virtual Appliance. Represents a virtual appliance you added to your Azure virtual network.
 - NULL. Represents a black hole. Packets forwarded to a black hole will not be forwarded at all.
- Nexthop Value. The next hop value contains the IP address packets should be forwarded to. Next hop values are only allowed in routes where the next hop type is *Virtual Appliance*. The next hop needs to be on the subnet (the local interface of the virtual appliance according to the network ID), not a remote subnet.

Default routes

Every subnet created in a virtual network is automatically associated with a route table that contains the following default route rules:

- Local VNet Rule: This rule is automatically created for every subnet in a virtual network. It specifies that there is a direct link between the VMs in the VNet and there is no intermediate next hop. This enables the VMs on the same subnet, regardless of the network ID that the VMs exist in, to communicate with each other without requiring a default gateway address.
- On-premises Rule: This rule applies to all traffic destined to the on-premises address range and uses VPN gateway as the next hop destination.
- Internet Rule: This rule handles all traffic destined to the public Internet and uses the infrastructure internet gateway as the next hop for all traffic destined to the Internet.

BGP routes

At the time of this writing, [ExpressRoute](#) is not yet supported in the [Network Resource Provider](#) for Azure Resource Manager. If you have an ExpressRoute connection between your on-premises network and Azure, you can enable BGP to propagate routes from your on-premises network to Azure once ExpressRoute is supported in the NRP. These BGP routes are used in the same way as default routes and user defined routes in each Azure subnet. For more information see [ExpressRoute Introduction](#).

NOTE

When ExpressRoute on NRP is supported, you will be able to configure your Azure environment to use forced tunneling through your on-premises network by creating a user defined route for subnet 0.0.0.0/0 that uses the VPN gateway as the next hop. However, this only works if you are using a VPN gateway, not ExpressRoute. For ExpressRoute, forced tunneling is configured through BGP.

User-defined routes

You cannot view the default routes specified above in your Azure environment, and for most environments, those are the only routes you will need. However, you may need to create a route table and add one or more routes in specific cases, such as:

- Forced tunneling to the Internet via your on-premises network.
- Use of virtual appliances in your Azure environment.

In the scenarios above, you will have to create a route table and add user defined routes to it. You can have multiple route tables, and the same route table can be associated to one or more subnets. And each subnet can only be associated to a single route table. All VMs and cloud services in a subnet use the route table associated to that subnet.

Subnets rely on default routes until a route table is associated to the subnet. Once an association exists, routing is done based on [Longest Prefix Match \(LPM\)](#) among both user defined routes and default routes. If there is more than one route with the same LPM match then a route is selected based on its origin in the following order:

1. User defined route
2. BGP route (when ExpressRoute is used)
3. Default route

NOTE

User defined routes are only applied to Azure VMs and cloud services. For instance, if you want to add a firewall virtual appliance between your on-premises network and Azure, you will have to create a user defined route for your Azure route tables that forwards all traffic going to the on-premises address space to the virtual appliance. However, incoming traffic from the on-premises address space will flow through your VPN gateway or ExpressRoute circuit straight to the Azure environment, bypassing the virtual appliance.

IP forwarding

As described above, one of the main reasons to create a user defined route is to forward traffic to a virtual appliance. A virtual appliance is nothing more than a VM that runs an application used to handle network traffic in some way, such as a firewall or a NAT device.

This virtual appliance VM must be able to receive incoming traffic that is not addressed to itself. To allow a VM to receive traffic addressed to other destinations, you must enable IP Forwarding in the VM.

Next steps

- To understand how to set up security principals with the correct access to work with resources in your organization, see [Authenticating a Service Principal with Azure Resource Manager](#)
- If you need to lock access to a resource, you can use management locks. See [Lock Resources with Azure Resource Manager](#)
- To configure routing and IP forwarding, see [Create User Defined Routes \(UDR\) in Resource Manager by using a template](#)
- For an overview of role-based access control, see [Role-based access control in the Microsoft Azure portal](#)

Resource policy overview

4/3/2017 • 7 min to read • [Edit Online](#)

Resource policies enable you to establish conventions for resources in your organization. By defining conventions, you can control costs and more easily manage your resources. For example, you can specify that only certain types of virtual machines are allowed, or you can require that all resources have a particular tag. Policies are inherited by all child resources. So, if a policy is applied to a resource group, it is applicable to all the resources in that resource group.

There are two concepts to understand about policies:

- policy definition - you describe when the policy is enforced and what action to take
- policy assignment - you apply the policy definition to a scope (subscription or resource group)

This topic focuses on policy definition. For information about policy assignment, see [Use Azure portal to assign and manage resource policies](#) or [Assign and manage policies through script](#).

Azure provides some built-in policy definitions that may reduce the number of policies you have to define. If a built-in policy definition works for your scenario, use that definition when assigning to a scope.

Policies are evaluated when creating and updating resources (PUT and PATCH operations).

NOTE

Currently, policy does not evaluate resource types that do not support tags, kind, and location, such as the Microsoft.Resources/deployments resource type. This support will be added at a future time. To avoid backward compatibility issues, you should explicitly specify type when authoring policies. For example, a tag policy that does not specify types is applied for all types. In that case, a template deployment may fail if there is a nested resource that doesn't support tags, and the deployment resource type has been added to policy evaluation.

How is it different from RBAC?

There are a few key differences between policy and role-based access control (RBAC). RBAC focuses on **user** actions at different scopes. For example, you are added to the contributor role for a resource group at the desired scope, so you can make changes to that resource group. Policy focuses on **resource** properties during deployment. For example, through policies, you can control the types of resources that can be provisioned or restrict the locations in which the resources can be provisioned. Unlike RBAC, policy is a default allow and explicit deny system.

To use policies, you must be authenticated through RBAC. Specifically, your account needs the:

- `Microsoft.Authorization/policydefinitions/write` permission to define a policy
- `Microsoft.Authorization/policyassignments/write` permission to assign a policy

These permissions are not included in the **Contributor** role.

Policy definition structure

You use JSON to create a policy definition. The policy definition contains elements for:

- parameters
- display name

- description
- policy rule
 - logical evaluation
 - effect

The following example shows a policy that limits where resources are deployed:

```
{
  "properties": {
    "parameters": {
      "allowedLocations": {
        "type": "array",
        "metadata": {
          "description": "The list of locations that can be specified when deploying resources",
          "strongType": "location",
          "displayName": "Allowed locations"
        }
      }
    },
    "displayName": "Allowed locations",
    "description": "This policy enables you to restrict the locations your organization can specify when deploying resources.",
    "policyRule": {
      "if": {
        "not": {
          "field": "location",
          "in": "[parameters('allowedLocations')]"
        }
      },
      "then": {
        "effect": "deny"
      }
    }
  }
}
```

Parameters

Using parameters helps simplify your policy management by reducing the number of policy definitions. You define a policy for a resource property (such as limiting the locations where resources can be deployed), and include parameters in the definition. Then, you reuse that policy definition for different scenarios by passing in different values (such as specifying one set of locations for a subscription) when assigning the policy.

You declare parameters when you create policy definitions.

```
"parameters": {
  "allowedLocations": {
    "type": "array",
    "metadata": {
      "description": "The list of allowed locations for resources.",
      "displayName": "Allowed locations"
    }
  }
}
```

The type of a parameter can be either string or array. The metadata property is used for tools like Azure portal to display user-friendly information.

In the policy rule, you reference parameters with the following syntax:

```
{  
    "field": "location",  
    "in": "[parameters('allowedLocations')]"  
}
```

Display name and description

You use the **displayName** and **description** to identify the policy definition, and provide context for when it is used.

Policy rule

The policy rule consists of **If** and **Then** blocks. In the **If** block, you define one or more conditions that specify when the policy is enforced. You can apply logical operators to these conditions to precisely define the scenario for a policy. In the **Then** block, you define the effect that happens when the **If** conditions are fulfilled.

```
{  
    "if": {  
        <condition> | <logical operator>  
    },  
    "then": {  
        "effect": "deny | audit | append"  
    }  
}
```

Logical operators

The supported logical operators are:

- "not": {condition or operator}
- "allOf": [{condition or operator},{condition or operator}]
- "anyOf": [{condition or operator},{condition or operator}]

The **not** syntax inverts the result of the condition. The **allOf** syntax (similar to the logical **And** operation) requires all conditions to be true. The **anyOf** syntax (similar to the logical **Or** operation) requires one or more conditions to be true.

You can nest logical operators. The following example shows a **Not** operation that is nested within an **And** operation.

```
"if": {  
    "allOf": [  
        {  
            "not": {  
                "field": "tags",  
                "containsKey": "application"  
            }  
        },  
        {  
            "field": "type",  
            "equals": "Microsoft.Storage/storageAccounts"  
        }  
    ]  
},
```

Conditions

The condition evaluates whether a **field** meets certain criteria. The supported conditions are:

- `"equals": "value"`
- `"like": "value"`
- `"contains": "value"`
- `"in": ["value1", "value2"]`
- `"containsKey": "keyName"`
- `"exists": "bool"`

When using the **like** condition, you can provide a wildcard (*) in the value.

Fields

Conditions are formed by using fields. A field represents properties in the resource request payload that is used to describe the state of the resource.

The following fields are supported:

- `name`
- `kind`
- `type`
- `location`
- `tags`
- `tags.*`
- property aliases

You use property aliases to access specific properties for a resource type. The supported aliases are:

- Microsoft.CDN/profiles/sku.name
- Microsoft.Compute/virtualMachines/imageOffer
- Microsoft.Compute/virtualMachines/imagePublisher
- Microsoft.Compute/virtualMachines/sku.name
- Microsoft.Compute/virtualMachines/imageSku
- Microsoft.Compute/virtualMachines/imageVersion
- Microsoft.SQL/servers/databases/edition
- Microsoft.SQL/servers/databases/elasticPoolName
- Microsoft.SQL/servers/databases/requestedServiceObjectiveId
- Microsoft.SQL/servers/databases/requestedServiceObjectiveName
- Microsoft.SQL/servers/elasticPools/dtu
- Microsoft.SQL/servers/elasticPools/edition
- Microsoft.SQL/servers/version
- Microsoft.Storage/storageAccounts/accessTier
- Microsoft.Storage/storageAccounts/enableBlobEncryption
- Microsoft.Storage/storageAccounts/sku.name
- Microsoft.Web/serverFarms/sku.name

Effect

Policy supports three types of effect - `deny`, `audit`, and `append`.

- **Deny** generates an event in the audit log and fails the request
- **Audit** generates a warning event in audit log but does not fail the request
- **Append** adds the defined set of fields to the request

For **append**, you must provide the following details:

```
"effect": "append",
"details": [
  {
    "field": "field name",
    "value": "value of the field"
  }
]
```

The value can be either a string or a JSON format object.

Policy examples

The following topics contain policy examples:

- For examples of tag policies, see [Apply resource policies for tags](#).
- For examples of storage policies, see [Apply resource policies to storage accounts](#).
- For examples of virtual machine policies, see [Apply resource policies to Linux VMs](#) and [Apply resource policies to Windows VMs](#)

Allowed resource locations

To specify which locations are allowed, see the example in the [Policy definition structure](#) section. To assign this policy definition, use the built-in policy with the resource ID

```
/providers/Microsoft.Authorization/policyDefinitions/e56962a6-4747-49cd-b67b-bf8b01975c4c .
```

Not allowed resource locations

To specify which locations are not allowed, use the following policy definition:

```
{
  "properties": {
    "parameters": {
      "notAllowedLocations": {
        "type": "array",
        "metadata": {
          "description": "The list of locations that are not allowed when deploying resources",
          "strongType": "location",
          "displayName": "Not allowed locations"
        }
      }
    },
    "displayName": "Not allowed locations",
    "description": "This policy enables you to block locations that your organization can specify when deploying resources.",
    "policyRule": {
      "if": {
        "field": "location",
        "in": "[parameters('notAllowedLocations')]"
      },
      "then": {
        "effect": "deny"
      }
    }
  }
}
```

Allowed resource types

The following example shows a policy that permits deployments for only the Microsoft.Resources, Microsoft.Compute, Microsoft.Storage, Microsoft.Network resource types. All others are denied:

```
{
  "if": {
    "not": {
      "anyOf": [
        {
          "field": "type",
          "like": "Microsoft.Resources/*"
        },
        {
          "field": "type",
          "like": "Microsoft.Compute/*"
        },
        {
          "field": "type",
          "like": "Microsoft.Storage/*"
        },
        {
          "field": "type",
          "like": "Microsoft.Network/*"
        }
      ]
    },
    "then": {
      "effect": "deny"
    }
  }
}
```

Set naming convention

The following example shows the use of wildcard, which is supported by the **like** condition. The condition states that if the name does match the mentioned pattern (`namePrefix*nameSuffix`) then deny the request:

```
{
  "if": {
    "not": {
      "field": "name",
      "like": "namePrefix*nameSuffix"
    }
  },
  "then": {
    "effect": "deny"
  }
}
```

Next steps

- After defining a policy rule, assign it to a scope. To assign policies through the portal, see [Use Azure portal to assign and manage resource policies](#). To assign policies through REST API, PowerShell or Azure CLI, see [Assign and manage policies through script](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).
- The policy schema is published at <http://schema.management.azure.com/schemas/2015-10-01-preview/policyDefinition.json>.

Use Azure portal to assign and manage resource policies

3/30/2017 • 2 min to read • [Edit Online](#)

The Azure portal enables you to assign resource policies to resource groups and subscriptions. The user interface makes it easy to select the policy that you want to assign, and specify parameter values for that policy to customize the policy settings.

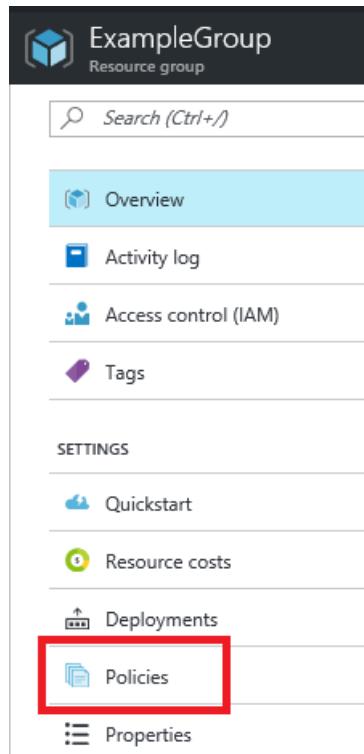
To assign a policy through the portal, the policy definition must already exist in your subscription. Your subscription has several built-in policy definitions that are ready for you to assign to resource groups or subscriptions. You see these built-in policies and any custom policies you have defined when using the portal to assign policies. For an introduction to policies and how to define customized policy, see [Resource policy overview](#).

Policies are inherited by all child resources. So, if a policy is applied to a resource group, it is applicable to all the resources in that resource group. In this article, the term **scope** refers to the resource group or subscription that is assigned the policy.

Policies are evaluated when creating and updating resources (PUT and PATCH operations).

Assign a policy

1. To assign a policy to either a resource group or subscription, select that resource group or subscription. In the settings, select **Policies**.



2. To create a policy assignment for this scope, select **Add assignment**.

The screenshot shows the Azure portal interface for a resource group named 'ExampleGroup'. On the left, there's a sidebar with links for 'Overview', 'Activity log', 'Access control (IAM)', and 'Tags'. The main area is titled 'Policies' and contains a search bar and a button labeled '+ Add assignment' which is highlighted with a red box. Below it is a section for 'ASSIGNMENT NAME' with a note stating 'No policy assignments to display within the given scope'.

3. Select the policy you want to assign. Even if you have not added any policy definitions to your subscription, you see the built-in policies that are available for assignment. These built-in policies cover many common scenarios.

The screenshot shows the 'Add assignment' dialog box. It includes an information icon and a note about assigning policies to multiple subscriptions or resource groups. A red box highlights the 'Policy definition' dropdown menu, which lists several options: 'Allowed locations', 'Allowed resource types', 'Allowed storage account SKUs', 'Allowed virtual machine SKUs', 'Not allowed resource types', and 'Require SQL Server version 12.0'.

4. After selecting a policy, you see a description of the policy, and any parameters for that policy. For example, the following image shows the **Allowed locations** parameter, which is required for the policy that restricts the available locations.

Add assignment

Policies can be assigned to multiple subscriptions and resource groups. Each subscription or resource group can be assigned multiple policies (N:N). Fill in the below fields to create a new assignment for a policy. [Learn more](#)

*** Policy definition ⓘ**

Allowed locations

This policy enables you to restrict the locations your organization can specify when deploying resources. Use to enforce your geo-compliance requirements.

*** Allowed locations ⓘ**

0 selected

*** Assignment name ⓘ**

*** Assignment id ⓘ**

Description

*** Scope ⓘ**

ExampleGroup

OK

- Through the user interface, select the values to specify for the policy parameters (such as the locations that can be used for deployment).

*** Allowed locations ⓘ**

2 selected

Filter

- Canada East
- Central India
- Central US
- East Asia
- East US
- East US 2
- Japan East
- Japan West
- Korea Central
- Vietnam Central

- Provide values for the other parameters. The scope is automatically assigned based on the blade you selected when starting the policy assignment. Select **OK** when done.

Add assignment

Policies can be assigned to multiple subscriptions and resource groups. Each subscription or resource group can be assigned multiple policies (N:N). Fill in the below fields to create a new assignment for a policy. [Learn more](#)

* Policy definition ⓘ

Allowed locations

This policy enables you to restrict the locations your organization can specify when deploying resources. Use to enforce your geo-compliance requirements.

* Allowed locations ⓘ

2 selected

* Assignment name ⓘ

Restrict regions for ExampleGroup ✓

* Assignment id ⓘ

example-group-region-policy ✓

Description

Limit resources to East regions for this resource group.

* Scope ⓘ

ExampleGroup

OK

You have assigned the policy to the specified scope.

View policy assignments

After assigning a policy, you see it in the list of policies for that scope. The **Details** tab shows a summary of the policy assignment.

Add assignment Edit assignment Delete

Search policy assignments.

ASSIGNMENT NAME	POLICY DEFINITION	SCOPE
Restrict regions for ExampleGroup	Allowed locations	Third Internal Consumption\ExampleGroup

Restrict regions for ExampleGroup

Details **Assignment rule**

Assignment id
`/subscriptions/resourceGroups/ExampleGroup/providers/Microsoft.Authorization/policyAssignments/example-group-region-policy`

Definition
 Allowed locations

Assignment description
 Limit resources to East regions for this resource group.

Definition description
 This policy enables you to restrict the locations your organization can specify when deploying resources. Use to enforce your geo-compliance requirements.

Scope
 Third Internal Consumption\ExampleGroup

Parameters

- **Allowed locations:** eastus, eastus2

The **Assignment rule** tab shows the JSON for the policy definition.

Restrict regions for ExampleGroup

Details **Assignment rule**

```

1 {
2   "if": {
3     "not": {
4       "field": "location",
5       "in": "[parameters('listOfAllowedLocations')]"
6     }
7   },
8   "then": {
9     "effect": "Deny"
10 }
11 }
```

Change an existing policy assignment

To change a policy, select **Edit assignment** or **Delete**

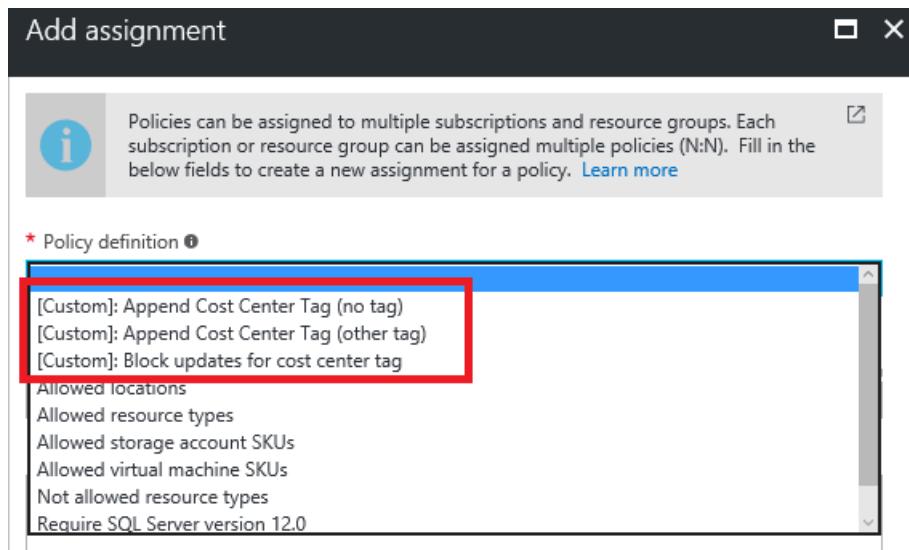
Add assignment **Edit assignment** Delete

Search policy assignments.

ASSIGNMENT NAME	POLICY DEFINITION
Restrict regions for ExampleGroup	Allowed locations

Assign custom policies

If you have defined custom policies in your subscription, those policies are available for assignment through the portal. Those policies are prefaced with **[Custom]**



The screenshot shows the 'Add assignment' dialog box. At the top, there's a message: 'Policies can be assigned to multiple subscriptions and resource groups. Each subscription or resource group can be assigned multiple policies (N:N). Fill in the below fields to create a new assignment for a policy.' Below this is a section titled 'Policy definition' with a red box highlighting three items: '[Custom]: Append Cost Center Tag (no tag)', '[Custom]: Append Cost Center Tag (other tag)', and '[Custom]: Block updates for cost center tag'. Further down, there are sections for 'Allowed locations', 'Allowed resource types', 'Allowed storage account SKUs', 'Allowed virtual machine SKUs', 'Not allowed resource types', and 'Require SQL Server version 12.0'.

Next steps

- To learn about the JSON syntax for defining policies, see [Resource policy overview](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).
- The policy schema is published at <http://schema.management.azure.com/schemas/2015-10-01-preview/policyDefinition.json>.

Assign and manage resource policies

3/30/2017 • 5 min to read • [Edit Online](#)

To implement a policy, you must perform three steps:

1. Define the policy rule with JSON.
2. Create a policy definition in your subscription from the JSON you created in the preceding step. This step makes the policy available for assignment but does not apply the rules to your subscription.
3. Assign the policy to a scope (such as a subscription or resource group). The rules of the policy are now enforced.

Azure provides some pre-defined policies that may reduce the number of policies you have to define. If a pre-defined policy works for your scenario, skip the first two steps and assign the pre-defined policy to a scope.

This article focuses on the steps to create a policy definition and assign that definition to a scope through REST API, PowerShell, or Azure CLI. If you prefer to use the portal to assign policies, see [Use Azure portal to assign and manage resource policies](#). This article does not focus on the syntax for creating the policy definition. For information about policy syntax, see [Resource policy overview](#).

REST API

Create policy definition

You can create a policy with the [REST API for Policy Definitions](#). The REST API enables you to create and delete policy definitions, and get information about existing definitions.

To create a policy definition, run:

```
PUT https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.authorization/policydefinitions/{policyDefinitionName}?api-version={api-version}
```

Include a request body similar to the following example:

```
{
  "properties": {
    "parameters": {
      "allowedLocations": {
        "type": "array",
        "metadata": {
          "description": "The list of locations that can be specified when deploying resources",
          "strongType": "location",
          "displayName": "Allowed locations"
        }
      }
    },
    "displayName": "Allowed locations",
    "description": "This policy enables you to restrict the locations your organization can specify when deploying resources.",
    "policyRule": {
      "if": {
        "not": {
          "field": "location",
          "in": "[parameters('allowedLocations')]"
        }
      },
      "then": {
        "effect": "deny"
      }
    }
  }
}
```

Assign policy

You can apply the policy definition at the desired scope through the [REST API for policy assignments](#). The REST API enables you to create and delete policy assignments, and get information about existing assignments.

To create a policy assignment, run:

```
PUT https://management.azure.com /subscriptions/{subscription-id}/providers/Microsoft.authorization/policyassignments/{policyAssignmentName}?api-version={api-version}
```

The {policy-assignment} is the name of the policy assignment.

Include a request body similar to the following example:

```
{
  "properties": {
    "displayName": "West US only policy assignment on the subscription",
    "description": "Resources can only be provisioned in West US regions",
    "parameters": {
      "allowedLocations": { "value": [ "northeurope", "westus" ] }
    },
    "policyDefinitionId": "/subscriptions/{subscription-id}/providers/Microsoft.Authorization/policyDefinitions/{definition-name}",
    "scope": "/subscriptions/{subscription-id}"
  }
}
```

View policy

To get a policy, use the [Get policy definition](#) operation.

Get aliases

You can retrieve aliases through the REST API:

```
GET /subscriptions/{id}/providers?$expand=resourceTypes/aliases&api-version=2015-11-01
```

The following example shows a definition of an alias. As you can see, an alias defines paths in different API versions, even when there is a property name change.

```
"aliases": [
  {
    "name": "Microsoft.Storage/storageAccounts/sku.name",
    "paths": [
      {
        "path": "properties.accountType",
        "apiVersions": [
          "2015-06-15",
          "2015-05-01-preview"
        ]
      },
      {
        "path": "sku.name",
        "apiVersions": [
          "2016-01-01"
        ]
      }
    ]
  }
]
```

PowerShell

Create policy definition

You can create a policy definition using the `New-AzureRmPolicyDefinition` cmdlet. The following example creates a policy definition for allowing resources only in North Europe and West Europe.

```
$policy = New-AzureRmPolicyDefinition -Name regionPolicyDefinition -Description "Policy to allow resource creation only in certain regions" -Policy '{
  "if": {
    "not": {
      "field": "location",
      "in": "[parameters(''allowedLocations'')]"
    }
  },
  "then": {
    "effect": "deny"
  }
}' -Parameter '{
  "allowedLocations": {
    "type": "array",
    "metadata": {
      "description": "An array of permitted locations for resources.",
      "strongType": "location",
      "displayName": "List of locations"
    }
  }
}'
```

The output is stored in a `$policy` object, which is used during policy assignment.

Rather than specifying the JSON as a parameter, you can provide the path to a json file containing the policy rule.

```
$policy = New-AzureRmPolicyDefinition -Name regionPolicyDefinition -Description "Policy to allow resource creation only in certain regions" -Policy "c:\policies\storageskupolicy.json"
```

Assign policy

You apply the policy at the desired scope by using the `New-AzureRmPolicyAssignment` cmdlet:

```
$rg = Get-AzureRmResourceGroup -Name "ExampleGroup"
$array = @("West US", "West US 2")
$param = @{"allowedLocations"=$array}
New-AzureRMPolicyAssignment -Name regionPolicyAssignment -Scope $rg.ResourceId -PolicyDefinition $policy -PolicyParameterObject $param
```

View policies

To get all policy assignments, use:

```
Get-AzureRmPolicyAssignment
```

To get a specific policy, use:

```
$rg = Get-AzureRmResourceGroup -Name "ExampleGroup"
(Get-AzureRmPolicyAssignment -Name regionPolicyAssignment -Scope $rg.ResourceId
```

To view the policy rule for a policy definition, use:

```
(Get-AzureRmPolicyDefinition -Name regionPolicyDefinition).Properties.policyRule | ConvertTo-Json
```

Remove policy assignment

To remove a policy assignment, use:

```
Remove-AzureRmPolicyAssignment -Name regionPolicyAssignment -Scope /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}
```

Azure CLI 2.0

Create policy definition

You can create a policy definition using Azure CLI 2.0 with the policy definition command. The following example creates a policy for allowing resources only in North Europe and West Europe.

```
az policy definition create --name regionPolicyDefinition --description "Policy to allow resource creation only in certain regions" --rules '{
  "if" : {
    "not" : {
      "field" : "location",
      "in" : ["northeurope", "westeurope"]
    }
  },
  "then" : {
    "effect" : "deny"
  }
}'
```

Assign policy

You can apply the policy to the desired scope by using the policy assignment command:

```
az policy assignment create --name regionPolicyAssignment --policy regionPolicyDefinition --scope /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}
```

View policy definition

To get a policy definition, use the following command:

```
az policy definition show --name regionPolicyAssignment
```

Remove policy assignment

To remove a policy assignment, use:

```
az policy assignment delete --name regionPolicyAssignment --scope /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}
```

Azure CLI 1.0

Create policy definition

You can create a policy definition using the Azure CLI with the policy definition command. The following example creates a policy for allowing resources only in North Europe and West Europe.

```
azure policy definition create --name regionPolicyDefinition --description "Policy to allow resource creation only in certain regions" --policy-string '{
  "if" : {
    "not" : {
      "field" : "location",
      "in" : ["northeurope" , "westeurope"]
    }
  },
  "then" : {
    "effect" : "deny"
  }
}'
```

It is possible to specify the path to a json file containing the policy instead of specifying the policy inline.

```
azure policy definition create --name regionPolicyDefinition --description "Policy to allow resource creation only in certain regions" --policy "path-to-policy-json-on-disk"
```

Assign policy

You can apply the policy to the desired scope by using the policy assignment command:

```
azure policy assignment create --name regionPolicyAssignment --policy-definition-id /subscriptions/{subscription-id}/providers/Microsoft.Authorization/policyDefinitions/{policy-name} --scope /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}
```

The scope here is the name of the resource group you specify. If the value of the parameter policy-definition-id is unknown, it is possible to obtain it through the Azure CLI.

```
azure policy definition show {policy-name}
```

View policy

To get a policy, use the following command:

```
azure policy definition show {definition-name} --json
```

Remove policy assignment

To remove a policy assignment, use:

```
azure policy assignment delete --name regionPolicyAssignment --scope /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}
```

Next steps

- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Apply resource policies for tags

3/30/2017 • 2 min to read • [Edit Online](#)

This topic provides common policy rules you can apply to ensure consistent use of tags on resources.

Applying a tag policy to a resource group or subscription with existing resources does not retroactively apply the policy to those resources. To enforce the policies on those resources, trigger an update to the existing resources, as shown in [Trigger updates to existing resources](#).

Ensure all resources in a resource group have a tag/value

A common requirement is that all resources in a resource group have a particular tag and value. This requirement is often needed to track costs by department. The following conditions must be met:

- The required tag and value are appended to new and updated resources that do not have any existing tags.
- The required tag and value are appended to new and updated resources that have other tags, but not the required tag and value.
- The required tag and value cannot be removed from any existing resources.

You accomplish this requirement by applying to a resource group the following three policies:

- [Append tag](#)
- [Append tag with other tags](#)
- [Require tag and value](#)

Append tag

The following policy rule appends costCenter tag with a predefined value when no tags are present:

```
{  
  "if": {  
    "field": "tags",  
    "exists": "false"  
  },  
  "then": {  
    "effect": "append",  
    "details": [  
      {  
        "field": "tags",  
        "value": {"costCenter": "myDepartment"}  
      }  
    ]  
  }  
}
```

Append tag with other tags

The following policy rule appends costCenter tag with a predefined value when tags are present, but the costCenter tag is not defined:

```
{
  "if": {
    "allOf": [
      {
        "field": "tags",
        "exists": "true"
      },
      {
        "field": "tags.costCenter",
        "exists": "false"
      }
    ]
  },
  "then": {
    "effect": "append",
    "details": [
      {
        "field": "tags.costCenter",
        "value": "myDepartment"
      }
    ]
  }
}
```

Require tag and value

The following policy rule denies update or creation of resources that do not have the costCenter tag assigned to the predefined value.

```
{
  "if": {
    "not": {
      "field": "tags.costCenter",
      "equals": "myDepartment"
    }
  },
  "then": {
    "effect": "deny"
  }
}
```

Require tags for a resource type

The following example shows how to nest logical operators to require an application tag for only a specified resource type (in this case, storage accounts).

```
{
  "if": {
    "allOf": [
      {
        "not": {
          "field": "tags",
          "containsKey": "application"
        }
      },
      {
        "field": "type",
        "equals": "Microsoft.Storage/storageAccounts"
      }
    ],
    "then": {
      "effect": "audit"
    }
  }
}
```

Require tag

The following policy denies requests that don't have a tag containing "costCenter" key (any value can be applied):

```
{
  "if": {
    "not" : {
      "field" : "tags",
      "containsKey" : "costCenter"
    }
  },
  "then" : {
    "effect" : "deny"
  }
}
```

Trigger updates to existing resources

The following PowerShell script triggers an update to existing resources to enforce tag policies you have added.

```
$group = Get-AzureRmResourceGroup -Name "ExampleGroup"

$resources = Find-AzureRmResource -ResourceGroupName $group.ResourceGroupName

foreach($r in $resources)
{
  try{
    $r | Set-AzureRmResource -Tags ($a;if($_.Tags -eq $NULL) { @{} } else { $_.Tags }) -Force -UsePatchSemantics
  }
  catch{
    Write-Host $r.ResourceId + "can't be updated"
  }
}
```

Next steps

- After defining a policy rule (as shown in the preceding examples), you need to create the policy definition and assign it to a scope. The scope can be a subscription, resource group, or resource. To assign policies through the

portal, see [Use Azure portal to assign and manage resource policies](#). To assign policies through REST API, PowerShell or Azure CLI, see [Assign and manage policies through script](#).

- For an introduction to resource policies, see [Resource policy overview](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Apply resource policies to storage accounts

3/30/2017 • 1 min to read • [Edit Online](#)

This topic shows several [resource policies](#) you can apply to Azure storage accounts. These policies ensure consistency for the storage accounts deployed in your organization.

Define permitted storage account types

The following policy restricts which [storage account types](#) can be deployed:

```
{  
  "if": {  
    "allOf": [  
      {  
        "field": "type",  
        "equals": "Microsoft.Storage/storageAccounts"  
      },  
      {  
        "not": {  
          "field": "Microsoft.Storage/storageAccounts/sku.name",  
          "in": [  
            "Standard_LRS",  
            "Standard_GRS"  
          ]  
        }  
      }  
    ],  
    "then": {  
      "effect": "deny"  
    }  
  }  
}
```

A similar policy rule with a parameter for accepting the allowed SKUs is available as a built-in policy definition. The built-in policy has the resource ID of

`/providers/Microsoft.Authorization/policyDefinitions/7433c107-6db4-4ad1-b57a-a76dce0154a1`.

Define permitted access tier

The following policy specifies the type of [access tier](#) that can be specified for storage accounts:

```
{
  "if": {
    "allOf": [
      {
        "field": "type",
        "equals": "Microsoft.Storage/storageAccounts"
      },
      {
        "field": "kind",
        "equals": "BlobStorage"
      },
      {
        "not": {
          "field": "Microsoft.Storage/storageAccounts/accessTier",
          "equals": "cool"
        }
      }
    ]
  },
  "then": {
    "effect": "deny"
  }
}
```

Ensure encryption is enabled

The following policy requires all storage accounts to enable [Storage service encryption](#):

```
{
  "if": {
    "allOf": [
      {
        "field": "type",
        "equals": "Microsoft.Storage/storageAccounts"
      },
      {
        "not": {
          "field": "Microsoft.Storage/storageAccounts/enableBlobEncryption",
          "equals": "true"
        }
      }
    ]
  },
  "then": {
    "effect": "deny"
  }
}
```

This policy rule is also available as a built-in policy definition with the resource ID of

`/providers/Microsoft.Authorization/policyDefinitions/7c5a74bf-ae94-4a74-8fcf-644d1e0e6e6f`.

Next steps

- After defining a policy rule (as shown in the preceding examples), you need to create the policy definition and assign it to a scope. The scope can be a subscription, resource group, or resource. To assign policies through the portal, see [Use Azure portal to assign and manage resource policies](#). To assign policies through REST API, PowerShell or Azure CLI, see [Assign and manage policies through script](#).
- For guidance on how enterprises can use Resource Manager to effectively manage subscriptions, see [Azure enterprise scaffold - prescriptive subscription governance](#).

Apply security and policies to Linux VMs with Azure Resource Manager

4/4/2017 • 2 min to read • [Edit Online](#)

By using policies, an organization can enforce various conventions and rules throughout the enterprise. Enforcement of the desired behavior can help mitigate risk while contributing to the success of the organization. In this article, we will describe how you can use Azure Resource Manager policies to define the desired behavior for your organization's Virtual Machines.

The outline for the steps to accomplish this is as below

1. Azure Resource Manager Policy 101
2. Define a policy for your Virtual Machine
3. Create the policy
4. Apply the policy

Azure Resource Manager Policy 101

For getting started with Azure Resource Manager policies, we recommend reading the article below and then continuing with the steps in this article. The article below describes the basic definition and structure of a policy, how policies get evaluated and gives various examples of policy definitions.

- [Use Policy to manage resources and control access](#)

Define a policy for your Virtual Machine

One of the common scenarios for an enterprise might be to only allow their users to create Virtual Machines from specific operating systems that have been tested to be compatible with a LOB application. Using an Azure Resource Manager policy this task can be accomplished in a few steps. In this policy example, we are going to allow only Ubuntu 14.04.2-LTS Virtual Machines to be created. The policy definition looks like below

```

"if": {
  "allOf": [
    {
      "field": "type",
      "equals": "Microsoft.Compute/virtualMachines"
    },
    {
      "not": {
        "allOf": [
          {
            "field": "Microsoft.Compute/virtualMachines/imagePublisher",
            "equals": "Canonical"
          },
          {
            "field": "Microsoft.Compute/virtualMachines/imageOffer",
            "equals": "UbuntuServer"
          },
          {
            "field": "Microsoft.Compute/virtualMachines/imageSku",
            "equals": "14.04.2-LTS"
          }
        ]
      }
    }
  ],
  "then": {
    "effect": "deny"
  }
}

```

The above policy can easily be modified to a scenario where you might want to allow any Ubuntu LTS image to be used for a Virtual Machine deployment with the below change

```
{
  "field": "Microsoft.Compute/virtualMachines/imageSku",
  "like": "*LTS"
}
```

Virtual Machine Property Fields

The table below describes the Virtual Machine properties that can be used as fields in your policy definition. For more information about policy fields, see [Use policy to manage resources and control access](#).

FIELD NAME	DESCRIPTION
imagePublisher	Specifies the publisher of the image
imageOffer	Specifies the offer for the chosen image publisher
imageSku	Specifies the SKU for the chosen offer
imageVersion	Specifies the image version for the chosen SKU

Create the Policy

A policy can easily be created using the REST API directly or the PowerShell cmdlets. You can read more about [creating and assigning a policy](#).

Apply the Policy

After creating the policy you'll need to apply it on a defined scope. The scope can be a subscription, resource group or even the resource. You can read more about [creating and assigning a policy](#).

Apply security and policies to Windows VMs with Azure Resource Manager

3/30/2017 • 2 min to read • [Edit Online](#)

By using policies, an organization can enforce various conventions and rules throughout the enterprise. Enforcement of the desired behavior can help mitigate risk while contributing to the success of the organization. In this article, we will describe how you can use Azure Resource Manager policies to define the desired behavior for your organization's Virtual Machines.

The outline for the steps to accomplish this is as below

1. Azure Resource Manager Policy 101
2. Define a policy for your Virtual Machine
3. Create the policy
4. Apply the policy

Azure Resource Manager Policy 101

For getting started with Azure Resource Manager policies, we recommend reading the article below and then continuing with the steps in this article. The article below describes the basic definition and structure of a policy, how policies get evaluated and gives various examples of policy definitions.

- [Use Policy to manage resources and control access](#)

Define a policy for your Virtual Machine

One of the common scenarios for an enterprise might be to only allow their users to create Virtual Machines from specific operating systems that have been tested to be compatible with a LOB application. Using an Azure Resource Manager policy this task can be accomplished in a few steps. In this policy example, we are going to allow only Windows Server 2012 R2 Datacenter Virtual Machines to be created. The policy definition looks like below

```

"if": {
  "allOf": [
    {
      "field": "type",
      "equals": "Microsoft.Compute/virtualMachines"
    },
    {
      "not": {
        "allOf": [
          {
            "field": "Microsoft.Compute/virtualMachines/imagePublisher",
            "equals": "MicrosoftWindowsServer"
          },
          {
            "field": "Microsoft.Compute/virtualMachines/imageOffer",
            "equals": "WindowsServer"
          },
          {
            "field": "Microsoft.Compute/virtualMachines/imageSku",
            "equals": "2012-R2-Datacenter"
          }
        ]
      }
    }
  ],
  "then": {
    "effect": "deny"
  }
}

```

The above policy can easily be modified to a scenario where you might want to allow any Windows Server Datacenter image to be used for a Virtual Machine deployment with the below change

```
{
  "field": "Microsoft.Compute/virtualMachines/imageSku",
  "like": "*Datacenter"
}
```

Virtual Machine Property Fields

The table below describes the Virtual Machine properties that can be used as fields in your policy definition. For more on policy fields, see the article below:

- [Fields and Sources](#)

FIELD NAME	DESCRIPTION
imagePublisher	Specifies the publisher of the image
imageOffer	Specifies the offer for the chosen image publisher
imageSku	Specifies the SKU for the chosen offer
imageVersion	Specifies the image version for the chosen SKU

Create the Policy

A policy can easily be created using the REST API directly or the PowerShell cmdlets. For creating the policy, see the article below:

- [Creating a Policy](#)

Apply the Policy

After creating the policy you'll need to apply it on a defined scope. The scope can be a subscription, resource group or even the resource. For applying the policy, see the article below:

- [Creating a Policy](#)

Troubleshoot common Azure deployment errors with Azure Resource Manager

4/3/2017 • 20 min to read • [Edit Online](#)

This topic describes how you can resolve some common Azure deployment errors you may encounter.

Two types of errors

There are two types of errors you can receive:

- validation errors
- deployment errors

The following image shows the activity log for a subscription. There are three operations that occurred in two deployments. In the first deployment, the template passed validation but failed when creating the resources (**Write Deployments**). In the second deployment, the template failed validation and did not proceed to the **Write Deployments**.

The screenshot shows the Azure Activity Log interface. At the top, there are buttons for 'Columns', 'Export', and 'Log search'. Below that is a banner with the text 'Gain insights into Azure activities using log search and visualization for' and a 'Log search' button. The main area has several filter options: 'Select query ...', 'Subscription' (set to 'Windows Azure MSDN - Visual Studio Ultimate'), 'Resource group' (set to 'All resource groups'), 'Timespan' (set to 'Last 6 hours'), 'Event category' (set to 'All categories'), and 'Category' (set to 'All categories'). Below the filters are 'Apply' and 'Reset' buttons. A message at the bottom says 'Query returned 5 items. Click here to download all the items as csv.' The table below lists the operations:

OPERATION NAME	STATUS
! Validate	Failed
▶ ! Write Deployments	Failed
! Validate	Succeeded

Validation errors arise from scenarios that can be pre-determined to cause a problem. Validation errors include syntax errors in your template, or trying to deploy resources that would exceed your subscription quotas. Deployment errors arise from conditions that occur during the deployment process. For example, a deployment error might arise from trying to access a resource that is being deployed in parallel.

Both types of errors return an error code that you use to troubleshoot the deployment. Both types of errors appear in the [activity log](#). However, validation errors do not appear in your deployment history because the deployment never started.

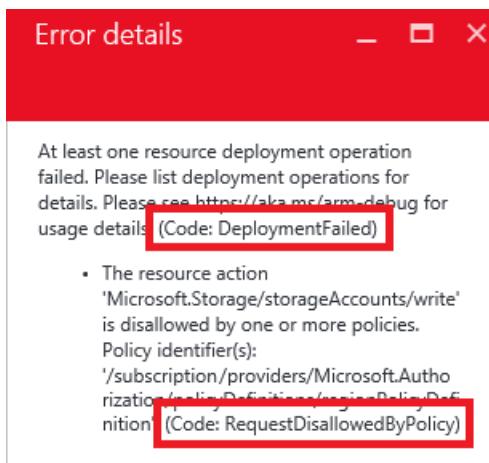
Error codes

The following error codes are described in this topic:

- [AccountNameInvalid](#)
- [Authorization failed](#)
- [BadRequest](#)
- [DeploymentFailed](#)
- [DisallowedOperation](#)
- [InvalidContentLink](#)
- [InvalidTemplate](#)
- [MissingSubscriptionRegistration](#)
- [NotFound](#)
- [NoRegisteredProviderFound](#)
- [OperationNotAllowed](#)
- [ParentResourceNotFound](#)
- [QuotaExceeded](#)
- [RequestDisallowedByPolicy](#)
- [ResourceNotFound](#)
- [SkuNotAvailable](#)
- [StorageAccountAlreadyExists](#)
- [StorageAccountAlreadyTaken](#)

DeploymentFailed

This error code indicates a general deployment error, but it is not the error code you need to start troubleshooting. The error code that actually helps you resolve the issue is usually one level below this error. For example, the following image shows the **RequestDisallowedByPolicy** error code that is under the deployment error.



SkuNotAvailable

When deploying a resource (typically a virtual machine), you may receive the following error code and error message:

```
Code: SkuNotAvailable
Message: The requested tier for resource '<resource>' is currently not available in location '<location>' for subscription '<subscriptionID>'. Please try another tier or deploy to a different location.
```

You receive this error when the resource SKU you have selected (such as VM size) is not available for the location you have selected. To resolve this issue, you need to determine which SKUs are available in a region. You can use either the portal or a REST operation to find available SKUs.

- To use the [portal](#), log in to the portal and add a resource through the interface. As you set the values, you see the available SKUs for that resource. You do not need to complete the deployment.

Choose a size

Browse the available sizes and their features

Prices presented below are estimates in your local currency that include only Azure infrastructure costs and any discounts for the subscription and location. The prices don't include any applicable software costs. Recommended sizes are determined by the publisher of the selected image based on hardware and software requirements.

 Recommended | [View all](#)

DS1_V2 Standard	DS2_V2 Standard	DS3_V2 Standard
1 Core	2 Cores	4 Cores
3.5 GB	7 GB	14 GB
 2 Data disks	 4 Data disks	 8 Data disks
 3200 Max IOPS	 6400 Max IOPS	 12800 Max IOPS
 7 GB Local SSD	 14 GB Local SSD	 28 GB Local SSD
 Load balancing	 Load balancing	 Load balancing
 Auto scale	 Auto scale	 Auto scale
 Premium disk supp...	 Premium disk supp...	 Premium disk supp...
54.31 USD/MONTH (ESTIMATED)	108.62 USD/MONTH (ESTIMATED)	217.99 USD/MONTH (ESTIMATED)

- To use the REST API for virtual machines, send the following request:

```
GET  
https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.Compute/skus?api-version=2016-03-30
```

It returns available SKUs and regions in the following format:

```
{  
  "value": [  
    {  
      "resourceType": "virtualMachines",  
      "name": "Standard_A0",  
      "tier": "Standard",  
      "size": "A0",  
      "locations": [  
        "eastus"  
      ],  
      "restrictions": []  
    },  
    {  
      "resourceType": "virtualMachines",  
      "name": "Standard_A1",  
      "tier": "Standard",  
      "size": "A1",  
      "locations": [  
        "eastus"  
      ],  
      "restrictions": []  
    },  
    ...  
  ]  
}
```

If you are unable to find a suitable SKU in that region or an alternative region that meets your business needs, contact [Azure Support](#).

DisallowedOperation

```
Code: DisallowedOperation
Message: The current subscription type is not permitted to perform operations on any provider
namespace. Please use a different subscription.
```

If you receive this error, you are using a subscription that is not permitted to access any Azure services other than Azure Active Directory. You might have this type of subscription when you need to access the classic portal but are not permitted to deploy resources. To resolve this issue, you must use a subscription that has permission to deploy resources.

To view your available subscriptions with PowerShell, use:

```
Get-AzureRmSubscription
```

And, to set the current subscription, use:

```
Set-AzureRmContext -SubscriptionName {subscription-name}
```

To view your available subscriptions with Azure CLI 2.0, use:

```
az account list
```

And, to set the current subscription, use:

```
az account set --subscription {subscription-name}
```

InvalidTemplate

This error can result from several different types of errors.

- Syntax error

If you receive an error message that indicates the template failed validation, you may have a syntax problem in your template.

```
Code=InvalidTemplate
Message=Deployment template validation failed
```

This error is easy to make because template expressions can be intricate. For example, the following name assignment for a storage account contains one set of brackets, three functions, three sets of parentheses, one set of single quotes, and one property:

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

If you do not provide the matching syntax, the template produces a value that is different than your intention.

When you receive this type of error, carefully review the expression syntax. Consider using a JSON editor like [Visual Studio](#) or [Visual Studio Code](#), which can warn you about syntax errors.

- Incorrect segment lengths

Another invalid template error occurs when the resource name is not in the correct format.

```
Code=InvalidTemplate
Message=Deployment template validation failed: 'The template resource {resource-name}' for type {resource-type} has incorrect segment lengths.
```

A root level resource must have one less segment in the name than in the resource type. Each segment is differentiated by a slash. In the following example, the type has two segments and the name has one segment, so it is a **valid name**.

```
{
  "type": "Microsoft.Web/serverfarms",
  "name": "myHostingPlanName",
  ...
}
```

But the next example is **not a valid name** because it has the same number of segments as the type.

```
{
  "type": "Microsoft.Web/serverfarms",
  "name": "appPlan/myHostingPlanName",
  ...
}
```

For child resources, the type and name have the same number of segments. This number of segments makes sense because the full name and type for the child includes the parent name and type. Therefore, the full name still has one less segment than the full type.

```
"resources": [
  {
    "type": "Microsoft.KeyVault/vaults",
    "name": "contosokeyvault",
    ...
    "resources": [
      {
        "type": "secrets",
        "name": "appPassword",
        ...
      }
    ]
  }
]
```

Getting the segments right can be tricky with Resource Manager types that are applied across resource providers. For example, applying a resource lock to a web site requires a type with four segments. Therefore, the name is three segments:

```
{
  "type": "Microsoft.Web/sites/providers/locks",
  "name": "[concat(variables('siteName'), '/Microsoft.Authorization/MySiteLock')]",
  ...
}
```

- Copy index is not expected

You encounter this **InvalidTemplate** error when you have applied the **copy** element to a part of the template that does not support this element. You can only apply the copy element to a resource type. You cannot apply copy to a property within a resource type. For example, you apply copy to a virtual machine, but you cannot apply it to the OS disks for a virtual machine. In some cases, you can convert a child resource to a parent resource to create a copy loop. For more information about using copy, see [Create multiple instances of resources in Azure Resource Manager](#).

- Parameter is not valid

If the template specifies permitted values for a parameter, and you provide a value that is not one of those values, you receive a message similar to the following error:

```
Code=InvalidTemplate;
Message=Deployment template validation failed: 'The provided value {parameter value}
for the template parameter {parameter name} is not valid. The parameter value is not
part of the allowed values'
```

Double check the allowed values in the template, and provide one during deployment.

- Circular dependency detected

You receive this error when resources depend on each other in a way that prevents the deployment from starting. A combination of interdependencies makes two or more resources wait for other resources that are also waiting. For example, resource1 depends on resource3, resource2 depends on resource1, and resource3 depends on resource2. You can usually solve this problem by removing unnecessary dependencies. For suggestions on troubleshooting dependency errors, see [Check deployment sequence](#).

NotFound and ResourceNotFound

When your template includes the name of a resource that cannot be resolved, you receive an error similar to:

```
Code=NotFound;
Message=Cannot find ServerFarm with name exampleplan.
```

If you are attempting to deploy the missing resource in the template, check whether you need to add a dependency. Resource Manager optimizes deployment by creating resources in parallel, when possible. If one resource must be deployed after another resource, you need to use the **dependsOn** element in your template to create a dependency on the other resource. For example, when deploying a web app, the App Service plan must exist. If you have not specified that the web app depends on the App Service plan, Resource Manager creates both resources at the same time. You receive an error stating that the App Service plan resource cannot be found, because it does not exist yet when attempting to set a property on the web app. You prevent this error by setting the dependency in the web app.

```
{
  "apiVersion": "2015-08-01",
  "type": "Microsoft.Web/sites",
  "dependsOn": [
    "[variables('hostingPlanName')]"
  ],
  ...
}
```

For suggestions on troubleshooting dependency errors, see [Check deployment sequence](#).

You also see this error when the resource exists in a different resource group than the one being deployed to. In that case, use the **resourceId** function to get the fully qualified name of the resource.

```
"properties": {
    "name": "[parameters('siteName')]",
    "serverFarmId": "[resourceId('plangroup', 'Microsoft.Web/serverfarms', parameters('hostingPlanName'))]"
}
```

If you attempt to use the [reference](#) or [listKeys](#) functions with a resource that cannot be resolved, you receive the following error:

```
Code=ResourceNotFound;
Message=The Resource 'Microsoft.Storage/storageAccounts/{storage name}' under resource
group {resource group name} was not found.
```

Look for an expression that includes the **reference** function. Double check that the parameter values are correct.

ParentResourceNotFound

When one resource is a parent to another resource, the parent resource must exist before creating the child resource. If it does not yet exist, you receive the following error:

```
Code=ParentResourceNotFound;
Message=Can not perform requested operation on nested resource. Parent resource 'exampleserver' not found."
```

The name of the child resource includes the parent name. For example, a SQL Database might be defined as:

```
{
    "type": "Microsoft.Sql/servers/databases",
    "name": "[concat(variables('databaseServerName'), '/', parameters('databaseName'))]",
    ...
}
```

But, if you do not specify a dependency on the parent resource, the child resource may get deployed before the parent. To resolve this error, include a dependency.

```
"dependsOn": [
    "[variables('databaseServerName')]"
]
```

StorageAccountAlreadyExists and StorageAccountAlreadyTaken

For storage accounts, you must provide a name for the resource that is unique across Azure. If you do not provide a unique name, you receive an error like:

```
Code=StorageAccountAlreadyTaken
Message=The storage account named mystorage is already taken.
```

You can create a unique name by concatenating your naming convention with the result of the [uniqueString](#) function.

```
"name": "[concat('storage', uniqueString(resourceGroup().id))]",
"type": "Microsoft.Storage/storageAccounts",
```

If you deploy a storage account with the same name as an existing storage account in your subscription, but provide a different location, you receive an error indicating the storage account already exists in a different location. Either delete the existing storage account, or provide the same location as the existing storage account.

AccountNameInvalid

You see the **AccountNameInvalid** error when attempting to give a storage account a name that includes prohibited characters. Storage account names must be between 3 and 24 characters in length and use numbers and lower-case letters only. The [uniqueString](#) function returns 13 characters. If you concatenate a prefix to the **uniqueString** result, provide a prefix that is 11 characters or less.

BadRequest

You may encounter a BadRequest status when you provide an invalid value for a property. For example, if you provide an incorrect SKU value for a storage account, the deployment fails. To determine valid values for property, look at the [REST API](#) for the resource type you are deploying.

NoRegisteredProviderFound and MissingSubscriptionRegistration

When deploying resource, you may receive the following error code and message:

```
Code: NoRegisteredProviderFound  
Message: No registered resource provider found for location {location}  
and API version {api-version} for type {resource-type}.
```

Or, you may receive a similar message that states:

```
Code: MissingSubscriptionRegistration  
Message: The subscription is not registered to use namespace {resource-provider-namespace}
```

You receive these errors for one of three reasons:

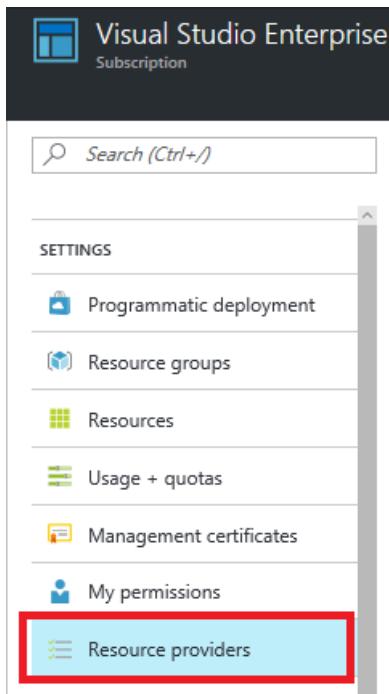
1. The resource provider has not been registered for your subscription
2. API version not supported for the resource type
3. Location not supported for the resource type

The error message should give you suggestions for the supported locations and API versions. You can change your template to one of the suggested values. Most providers are registered automatically by the Azure portal or the command-line interface you are using, but not all. If you have not used a particular resource provider before, you may need to register that provider. You can discover more about resource providers through PowerShell or Azure CLI.

Portal

You can see the registration status and register a resource provider namespace through the portal.

1. For your subscription, select **Resource providers**.



2. Look at the list of resource providers, and if necessary, select the **Register** link to register the resource provider of the type you are trying to deploy.

Provider	Status	Action
Microsoft.ClassicStorage	Registered	Unregister
Microsoft.OperationalInsights	Registered	Unregister
Microsoft.Storage	Registered	Unregister
84codes.CloudAMQP	NotRegistered	Register
AppDynamics.APM	NotRegistered	Register

PowerShell

To see your registration status, use **Get-AzureRmResourceProvider**.

```
Get-AzureRmResourceProvider -ListAvailable
```

To register a provider, use **Register-AzureRmResourceProvider** and provide the name of the resource provider you wish to register.

```
Register-AzureRmResourceProvider -ProviderNamespace Microsoft.Cdn
```

To get the supported locations for a particular type of resource, use:

```
((Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | Where-Object ResourceTypeName -eq sites).Locations
```

To get the supported API versions for a particular type of resource, use:

```
((Get-AzureRmResourceProvider -ProviderNamespace Microsoft.Web).ResourceTypes | Where-Object ResourceTypeName -eq sites).ApiVersions
```

Azure CLI

To see whether the provider is registered, use the `azure provider list` command.

```
az provider list
```

To register a resource provider, use the `azure provider register` command, and specify the *namespace* to register.

```
az provider register --namespace Microsoft.Cdn
```

To see the supported locations and API versions for a resource type, use:

```
az provider show -n Microsoft.Web --query "resourceTypes[?resourceType=='sites'].locations"
```

QuotaExceeded and OperationNotAllowed

You might have issues when deployment exceeds a quota, which could be per resource group, subscriptions, accounts, and other scopes. For example, your subscription may be configured to limit the number of cores for a region. If you attempt to deploy a virtual machine with more cores than the permitted amount, you receive an error stating the quota has been exceeded. For complete quota information, see [Azure subscription and service limits, quotas, and constraints](#).

To examine your subscription's quotas for cores, you can use the `azure vm list-usage` command in the Azure CLI. The following example illustrates that the core quota for a free trial account is 4:

```
az vm list-usage --location "South Central US"
```

Which returns:

```
[  
 {  
   "currentValue": 0,  
   "limit": 2000,  
   "name": {  
     "localizedValue": "Availability Sets",  
     "value": "availabilitySets"  
   }  
 },  
 ...  
 ]
```

If you deploy a template that creates more than four cores in the West US region, you get a deployment error that looks like:

```
Code=OperationNotAllowed  
Message=Operation results in exceeding quota limits of Core.  
Maximum allowed: 4, Current in use: 4, Additional requested: 2.
```

Or in PowerShell, you can use the `Get-AzureRmVMUsage` cmdlet.

```
Get-AzureRmVMUsage
```

Which returns:

```
...
CurrentValue : 0
Limit       : 4
Name        : {
    "value": "cores",
    "localizedValue": "Total Regional Cores"
}
Unit        : null
...
```

In these cases, you should go to the portal and file a support issue to raise your quota for the region into which you want to deploy.

NOTE

Remember that for resource groups, the quota is for each individual region, not for the entire subscription. If you need to deploy 30 cores in West US, you have to ask for 30 Resource Manager cores in West US. If you need to deploy 30 cores in any of the regions to which you have access, you should ask for 30 Resource Manager cores in all regions.

InvalidContentLink

When you receive the error message:

```
Code=InvalidContentLink
Message=Unable to download deployment content from ...
```

You have most likely attempted to link to a nested template that is not available. Double check the URI you provided for the nested template. If the template exists in a storage account, make sure the URI is accessible. You may need to pass a SAS token. For more information, see [Using linked templates with Azure Resource Manager](#).

RequestDisallowedByPolicy

You receive this error when your subscription includes a resource policy that prevents an action you are trying to perform during deployment. In the error message, look for the policy identifier.

```
Policy identifier(s):
'/subscriptions/{guid}/providers/Microsoft.Authorization/policyDefinitions/regionPolicyDefinition'
```

In **PowerShell**, provide that policy identifier as the **Id** parameter to retrieve details about the policy that blocked your deployment.

```
(Get-AzureRmPolicyDefinition -Id
"/subscriptions/{guid}/providers/Microsoft.Authorization/policyDefinitions/regionPolicyDefinition").Properties
.policyRule | ConvertTo-Json
```

In **Azure CLI 2.0**, provide the name of the policy definition:

```
az policy definition show --name regionPolicyAssignment
```

For more information about policies, see [Use Policy to manage resources and control access](#).

Authorization failed

You may receive an error during deployment because the account or service principal attempting to deploy the resources does not have access to perform those actions. Azure Active Directory enables you or your administrator to control which identities can access what resources with a great degree of precision. For example, if your account is assigned to the Reader role, you are not able to create resources. In that case, you see an error message indicating that authorization failed.

For more information about role-based access control, see [Azure Role-Based Access Control](#).

Troubleshooting tricks and tips

Enable debug logging

You can discover valuable information about how your deployment is processed by logging the request, response, or both.

- PowerShell

In PowerShell, set the **DeploymentDebugLogLevel** parameter to All, ResponseContent, or RequestContent.

```
New-AzureRmResourceGroupDeployment -ResourceGroupName examplegroup -TemplateFile c:\Azure\Templates\storage.json -DeploymentDebugLogLevel All
```

Examine the request content with the following cmdlet:

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName storageonly -ResourceGroupName startgroup).Properties.request | ConvertTo-Json
```

Or, the response content with:

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName storageonly -ResourceGroupName startgroup).Properties.response | ConvertTo-Json
```

This information can help you determine whether a value in the template is being incorrectly set.

- Azure CLI 2.0

Examine the deployment operations with the following command:

```
az group deployment operation list --resource-group ExampleGroup --name vmlinu
```

- Nested template

To log debug information for a nested template, use the **debugSetting** element.

```
{
    "apiVersion": "2016-09-01",
    "name": "nestedTemplate",
    "type": "Microsoft.Resources/deployments",
    "properties": {
        "mode": "Incremental",
        "templateLink": {
            "uri": "{template-uri}",
            "contentVersion": "1.0.0.0"
        },
        "debugSetting": {
            "detailLevel": "requestContent, responseContent"
        }
    }
}
```

Create a troubleshooting template

In some cases, the easiest way to troubleshoot your template is to test parts of it. You can create a simplified template that enables you to focus on the part that you believe is causing the error. For example, suppose you are receiving an error when referencing a resource. Rather than dealing with an entire template, create a template that returns the part that may be causing your problem. It can help you determine whether you are passing in the right parameters, using template functions correctly, and getting the resource you expect.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageName": {
            "type": "string"
        },
        "storageResourceGroup": {
            "type": "string"
        }
    },
    "variables": {},
    "resources": [],
    "outputs": {
        "exampleOutput": {
            "value": "[reference(resourceId(parameters('storageResourceGroup'),
'Microsoft.Storage/storageAccounts', parameters('storageName')), '2016-05-01')]",
            "type" : "object"
        }
    }
}
```

Or, suppose you are encountering deployment errors that you believe are related to incorrectly set dependencies. Test your template by breaking it into simplified templates. First, create a template that deploys only a single resource (like a SQL Server). When you are sure you have that resource correctly defined, add a resource that depends on it (like a SQL Database). When you have those two resources correctly defined, add other dependent resources (like auditing policies). In between each test deployment, delete the resource group to make sure you adequately testing the dependencies.

Check deployment sequence

Many deployment errors happen when resources are deployed in an unexpected sequence. These errors arise when dependencies are not correctly set. When you are missing a needed dependency, one resource attempts to use a value for another resource but the other does not yet exist. You get an error stating that a resource is not found. You may encounter this type of error intermittently because the deployment time for each resource can vary. For example, your first attempt to deploy your resources succeeds because a required resource randomly completes in time. However, your second attempt fails because the required resource did not complete in time.

But, you want to avoid setting dependencies that are not needed. When you have unnecessary dependencies, you prolong the duration of the deployment by preventing resources that are not dependent on each other from being deployed in parallel. In addition, you may create circular dependencies that block the deployment. The [reference](#) function creates an implicit dependency on the resource you specify as a parameter in the function, if that resource is deployed in the same template. Therefore, you may have more dependencies than the dependencies specified in the **dependsOn** property. The [resourceId](#) function does not create an implicit dependency or validate that the resource exists.

When you encounter dependency problems, you need to gain insight into the order of resource deployment. To view the order of deployment operations:

1. Select the deployment history for your resource group.

The screenshot shows the Azure portal interface for a resource group named 'examplegroup'. At the top, there's a search bar and navigation buttons for 'Add', 'Columns', and 'Delete'. Below that, under the heading 'Essentials', it displays the 'Subscription name (change) Azure Internal Consumption' and 'Deployments' section. The 'Deployments' section indicates '1 Succeeded'. A red box highlights the '1 Succeeded' text.

2. Select a deployment from the history, and select **Events**.

The screenshot shows a detailed view of a deployment history item. The title bar says 'Deployment hi... examplegroup' and the main title is 'Microsoft.StorageAccount-201612121139 Deployment'. The deployment status is 'Succeeded'. On the left, a list of events is shown, with the first event 'Microsoft.StorageAccount-2016121211...' and its timestamp '12/12/2016 11:40:09 AM' highlighted with a red box. On the right, a summary table provides details like Deployment Date, Status, Duration, Resource Group, and a 'Events' button, which is also highlighted with a red box.

3. Examine the sequence of events for each resource. Pay attention to the status of each operation. For example, the following image shows three storage accounts that deployed in parallel. Notice that the three storage accounts are started at the same time.

EVENT	L...	STATUS	TIME
Microsoft.Resources/deployments/write		Succeeded	4 min ago
Microsoft.Storage/storageAccounts/write		Succeeded	4 min ago
Microsoft.Storage/storageAccounts/write		Succeeded	4 min ago
Microsoft.Storage/storageAccounts/write		Succeeded	4 min ago
Microsoft.Storage/storageAccounts/write		Accepted	4 min ago
Microsoft.Storage/storageAccounts/write		Accepted	4 min ago
Microsoft.Storage/storageAccounts/write		Accepted	4 min ago
Microsoft.Storage/storageAccounts/write		Started	4 min ago
Microsoft.Storage/storageAccounts/write		Started	4 min ago
Microsoft.Storage/storageAccounts/write		Started	4 min ago
Microsoft.Resources/deployments/write		Succeeded	4 min ago
Microsoft.Resources/deployments/write		Started	4 min ago

The next image shows three storage accounts that are not deployed in parallel. The second storage account depends on the first storage account, and the third storage account depends on the second storage account. Therefore, the first storage account is started, accepted, and completed before the next is started.

EVENT	L...	STATUS	TIME
Microsoft.Resources/deployments/write		Succeeded	Just now
Microsoft.Storage/storageAccounts/write		Succeeded	Just now
Microsoft.Storage/storageAccounts/write		Accepted	1 min ago
Microsoft.Storage/storageAccounts/write		Started	1 min ago
Microsoft.Storage/storageAccounts/write		Succeeded	1 min ago
Microsoft.Storage/storageAccounts/write		Accepted	2 min ago
Microsoft.Storage/storageAccounts/write		Started	2 min ago
Microsoft.Storage/storageAccounts/write		Succeeded	2 min ago
Microsoft.Storage/storageAccounts/write		Accepted	2 min ago
Microsoft.Storage/storageAccounts/write		Started	2 min ago
Microsoft.Resources/deployments/write		Succeeded	2 min ago
Microsoft.Resources/deployments/write		Started	2 min ago

Real world scenarios can be considerably more complicated, but you can use the same technique to discover when deployment is started and completed for each resource. Look through your deployment events to see if the sequence is different than you would expect. If so, reevaluate the dependencies for this resource.

Resource Manager identifies circular dependencies during template validation. It returns an error message that

specifically states a circular dependency exists. To solve a circular dependency:

1. In your template, find the resource identified in the circular dependency.
2. For that resource, examine the **dependsOn** property and any uses of the **reference** function to see which resources it depends on.
3. Examine those resources to see which resources they depend on. Follow the dependencies until you notice a resource that depends on the original resource.
4. For the resources involved in the circular dependency, carefully examine all uses of the **dependsOn** property to identify any dependencies that are not needed. Remove those dependencies. If you are unsure that a dependency is needed, try removing it.
5. Redeploy the template.

Removing values from the **dependsOn** property can cause errors when you deploy the template. If you encounter an error, add the dependency back into the template.

If that approach does not solve the circular dependency, consider moving part of your deployment logic into child resources (such as extensions or configuration settings). Configure those child resources to deploy after the resources involved in the circular dependency. For example, suppose you are deploying two virtual machines but you must set properties on each one that refer to the other. You can deploy them in the following order:

1. vm1
2. vm2
3. Extension on vm1 depends on vm1 and vm2. The extension sets values on vm1 that it gets from vm2.
4. Extension on vm2 depends on vm1 and vm2. The extension sets values on vm2 that it gets from vm1.

The same approach works for App Service apps. Consider moving configuration values into a child resource of the app resource. You can deploy two web apps in the following order:

1. webapp1
2. webapp2
3. config for webapp1 depends on webapp1 and webapp2. It contains app settings with values from webapp2.
4. config for webapp2 depends on webapp1 and webapp2. It contains app settings with values from webapp1.

Troubleshooting other services

If the preceding deployment error codes did not help you troubleshoot your issue, you can find more detailed troubleshooting guidance for each Azure service.

The following table lists troubleshooting topics for Virtual Machines.

ERROR	ARTICLES
Custom script extension errors	Windows VM extension failures or Linux VM extension failures
OS image provisioning errors	New Windows VM errors or New Linux VM errors
Allocation failures	Windows VM allocation failures or Linux VM allocation failures
Secure Shell (SSH) errors when attempting to connect	Secure Shell connections to Linux VM

ERROR	ARTICLES
Errors connecting to application running on VM	Application running on Windows VM or Application running on a Linux VM
Remote Desktop connection errors	Remote Desktop connections to Windows VM
Connection errors resolved by redeploying	Redeploy Virtual Machine to new Azure node
Cloud service errors	Cloud service deployment problems

The following table lists troubleshooting topics for other Azure services. It focuses on issues related to deploying or configuring resources. If you need help troubleshooting run-time issues with a resource, see the documentation for that Azure service.

SERVICE	ARTICLE
Automation	Troubleshooting tips for common errors in Azure Automation
Azure Stack	Microsoft Azure Stack troubleshooting
Data Factory	Troubleshoot Data Factory issues
Service Fabric	Monitor and diagnose Azure Service Fabric applications
Site Recovery	Monitor and troubleshoot protection for virtual machines and physical servers
Storage	Monitor, diagnose, and troubleshoot Microsoft Azure Storage
StorSimple	Troubleshoot StorSimple device deployment issues
SQL Database	Troubleshoot connection issues to Azure SQL Database
SQL Data Warehouse	Troubleshooting Azure SQL Data Warehouse

Next steps

- To learn about auditing actions, see [Audit operations with Resource Manager](#).
- To learn about actions to determine the errors during deployment, see [View deployment operations](#).

View activity logs to audit actions on resources

1/24/2017 • 3 min to read • [Edit Online](#)

Through activity logs, you can determine:

- what operations were taken on the resources in your subscription
- who initiated the operation (although operations initiated by a backend service do not return a user as the caller)
- when the operation occurred
- the status of the operation
- the values of other properties that might help you research the operation

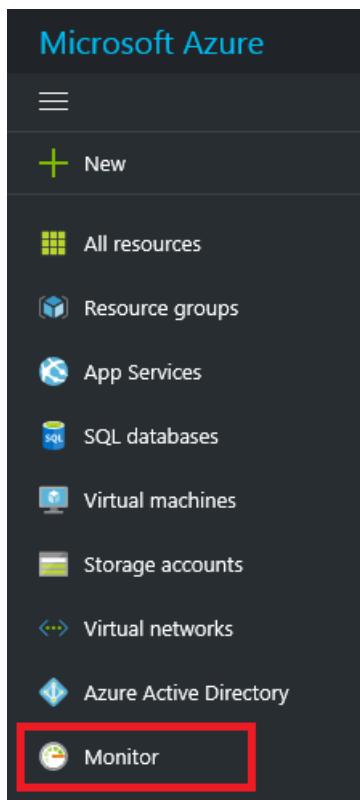
The activity log contains all write operations (PUT, POST, DELETE) performed on your resources. It does not include read operations (GET). You can use the audit logs to find an error when troubleshooting or to monitor how a user in your organization modified a resource.

Activity logs are retained for 90 days. You can query for any range of dates, as long as the starting date is not more than 90 days in the past.

You can retrieve information from the activity logs through the portal, PowerShell, Azure CLI, Insights REST API, or [Insights .NET Library](#).

Portal

1. To view the activity logs through the portal, select **Monitor**.



Or, to automatically filter the activity log for a particular resource or resource group, select **Activity log** from that resource blade. Notice that the activity log is automatically filtered by the selected resource.

The screenshot shows the 'Activity log' blade for a storage account named 'storagedemo'. The 'Activity log' tab is selected. In the top navigation bar, there are 'Columns', 'Export', and a search bar. Below the search bar are filter options: 'Subscription' (Windows Azure MSDN), 'Resource group' (demo-group), 'Resource' (storagedemo), 'Timespan' (Last 1 hour), 'Event category' (All categories), and 'Event severity' (4 selected). The 'Apply' button is highlighted in blue.

2. In the **Activity Log** blade, you see a summary of recent operations.

The screenshot shows the 'Activity log' blade with the following details:

- Filter Options:** Subscription: Windows Azure MSDN - Visual..., Resource group: All resource groups, Resource: All resources, Timespan: Last 1 hour, Event category: All categories, Event severity: 4 selected.
- Message:** Query returned 5 items. [Click here to download all the items as csv.](#)
- Table Headers:** OPERATION NAME, STATUS, TIME, TIME STAMP.
- Table Data:**

OPERATION NAME	STATUS	TIME	TIME STAMP
► ❌ Write Deployments	Failed	Just now	Mon Aug 22...
ℹ️ Validate	Succeeded	Just now	Mon Aug 22...
ℹ️ Register	Succeeded	3 min ago	Mon Aug 22...
► ℹ️ Delete resource group	Succeeded	3 min ago	Mon Aug 22...
ℹ️ Register	Succeeded	3 min ago	Mon Aug 22...

3. To restrict the number of operations displayed, select different conditions. For example, the following image shows the **Timespan** and **Event initiated by** fields changed to view the actions taken by a particular user or application for the past month. Select **Apply** to view the results of your query.

The screenshot shows the 'Activity log' blade with the following details:

- Filter Options:** Subscription: Windows Azure MSDN, Resource group: All resource groups, Resource type: All resource types, Timespan: Last month, Event category: All categories, Event severity: 4 selected, Event initiated by: CloudSense.
- Message:** Insights (Last 24 hours): 1 failed dep

4. If you need to run the query again later, select **Save** and give the query a name.

5. To quickly run a query, you can select one of the built-in queries, such as failed deployments.

The selected query automatically sets the required filter values.

6. Select one of the operations to see a summary of the event.

OPERATION NAME	STATUS	TIME	TIME STAMP	SUBSCRIBER
! Validate	Failed	4 min ago	Mon Jan 09 2...	Third Line

OPERATION NAME
! Validate

Validate

Summary JSON

Operation name
Validate

Time stamp
Mon Jan 09 2017 14:14:39 GMT-0800 (Pacific Standard Time)

Event initiated by

Error code
KeyVaultParameterReferenceNotFound

Message
The specified KeyVault '/subscriptions/ ... could not be found.

PowerShell

1. To retrieve log entries, run the **Get-AzureRmLog** command. You provide additional parameters to filter the list of entries. If you do not specify a start and end time, entries for the last hour are returned. For example, to retrieve the operations for a resource group during the past hour run:

```
Get-AzureRmLog -ResourceGroup ExampleGroup
```

The following example shows how to use the activity log to research operations taken during a specified time. The start and end dates are specified in a date format.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime 2015-08-28T06:00 -EndTime 2015-09-10T06:00
```

Or, you can use date functions to specify the date range, such as the last 14 days.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime (Get-Date).AddDays(-14)
```

2. Depending on the start time you specify, the previous commands can return a long list of operations for the resource group. You can filter the results for what you are looking for by providing search criteria. For example, if you are trying to research how a web app was stopped, you could run the following command:

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime (Get-Date).AddDays(-14) | Where-Object  
OperationName -eq Microsoft.Web/sites/stop/action
```

Which for this example shows that a stop action was performed by someone@contoso.com.

```
Authorization      :  
Scope      : /subscriptions/xxxxx/resourcegroups/ExampleGroup/providers/Microsoft.Web/sites/ExampleSite  
Action     : Microsoft.Web/sites/stop/action  
Role       : Subscription Admin  
Condition   :  
Caller      : someone@contoso.com  
CorrelationId : 84beae59-92aa-4662-a6fc-b6fecc0ff8da  
EventSource   : Administrative  
EventTimestamp : 8/28/2015 4:08:18 PM  
OperationName : Microsoft.Web/sites/stop/action  
ResourceGroupName : ExampleGroup  
ResourceId    :  
/subscriptions/xxxxx/resourcegroups/ExampleGroup/providers/Microsoft.Web/sites/ExampleSite  
Status       : Succeeded  
SubscriptionId : xxxxx  
SubStatus    : OK
```

3. You can look up the actions taken by a particular user, even for a resource group that no longer exists.

```
Get-AzureRmLog -ResourceGroup deletedgroup -StartTime (Get-Date).AddDays(-14) -Caller  
someone@contoso.com
```

4. You can filter for failed operations.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -Status Failed
```

5. You can focus on one error by looking at the status message for that entry.

```
((Get-AzureRmLog -Status Failed -ResourceGroup ExampleGroup -  
DetailedOutput).Properties[1].Content["statusMessage"] | ConvertFrom-Json).error
```

Which returns:

```
code          message
-----
DnsRecordInUse DNS record dns.westus.cloudapp.azure.com is already used by another public IP.
```

Azure CLI

- To retrieve log entries, you run the **azure group log show** command.

```
azure group log show ExampleGroup --json
```

REST API

The REST operations for working with the activity log are part of the [Insights REST API](#). To retrieve activity log events, see [List the management events in a subscription](#).

Next steps

- Azure Activity logs can be used with Power BI to gain greater insights about the actions in your subscription. See [View and analyze Azure Activity Logs in Power BI and more](#).
- To learn about setting security policies, see [Azure Role-based Access Control](#).
- To learn about the commands for viewing deployment operations, see [View deployment operations](#).
- To learn how to prevent deletions on a resource for all users, see [Lock resources with Azure Resource Manager](#).

View deployment operations with Azure Resource Manager

3/6/2017 • 4 min to read • [Edit Online](#)

You can view the operations for a deployment through the Azure portal. You may be most interested in viewing the operations when you have received an error during deployment so this article focuses on viewing operations that have failed. The portal provides an interface that enables you to easily find the errors and determine potential fixes.

You can troubleshoot your deployment by looking at either the audit logs, or the deployment operations. This topic shows both methods. For help with resolving particular deployment errors, see [Resolve common errors when deploying resources to Azure with Azure Resource Manager](#).

Portal

To see the deployment operations, use the following steps:

1. For the resource group involved in the deployment, notice the status of the last deployment. You can select this status to get more details.

The screenshot shows the Azure portal's resource group management interface. At the top, it displays 'ExampleGroup' as a 'Resource group'. Below the title, there are three buttons: 'Settings', 'Add', and 'Delete'. Under the 'Essentials' section, it shows the 'Subscription name' as 'Windows Azure MSDN - Visual Studio Ulti...'. The 'Last deployment' section is highlighted with a red box, showing the date '3/16/2016 (Failed)'.

2. You see the recent deployment history. Select the deployment that failed.

The screenshot shows the 'Deployment history' page for the 'examplegroup' resource group. It displays a single deployment entry: 'Microsoft.Template' from '6/14/2016 12:34:23 PM'. A red exclamation icon is next to the deployment entry, indicating that it failed.

3. Select the link to see a description of why the deployment failed. In the image below, the DNS record is not unique.

Microsoft.Template
Deployment

Error details

! Failed. Click here for details →

At least one resource deployment operation failed. Please list deployment operations for details. Please see <https://aka.ms/arm-debug> for usage details. (Code: DeploymentFailed)

- DNS record
dns.centralus.cloudapp.azure.com is already used by another public IP. (Code: DnsRecordInUse)

Summary
DEPLOYMENT DATE 6/14/2016 12:15:04 PM
STATUS Failed
RESOURCE GROUP examplegroup

This error message should be enough for you to begin troubleshooting. However, if you need more details about which tasks were completed, you can view the operations as shown in the following steps.

4. You can view all the deployment operations in the **Deployment** blade. Select any operation to see more details.

Operation details					
RESOURCE	TYPE	STATUS	TIMESTAMP		
tfstorage9856	Microsoft.Storage/storageAcco...	OK	2016-06-14T21:18...		
myPublicIP	Microsoft.Network/publicIPAdd...	BadRequest	2016-06-14T21:18...		
myVNET	Microsoft.Network/virtualNetw...	OK	2016-06-14T21:18...		
myAvSet	Microsoft.Compute/availabilityS...	OK	2016-06-14T21:18...		

In this case, you see that the storage account, virtual network, and availability set were successfully created. The public IP address failed, and other resources were not attempted.

5. You can view events for the deployment by selecting **Events**.

Microsoft.Template
Deployment

! Failed. Click here for details →

Summary
DEPLOYMENT DATE 6/14/2016 2:18:21 PM
STATUS Failed
RESOURCE GROUP examplegroup
RELATED Events

6. You see all the events for the deployment and select any one for more details. Notice too the correlation IDs. This value can be helpful when working with technical support to troubleshoot a deployment.

Microsoft.Resources/deployments/write

LEVEL	Error
STATUS	Failed
TIME	Tuesday, June 14, 2016 2:18:21 PM
CALLER	someone@example.com
CORRELATION IDS	a1ffc1c1-5b71-487f-9306-4257f6a7444c

EVENT	LEVEL	STATUS	TIME
Microsoft.Resources/de...	>Error	Failed	5...
Microsoft.Storage/stora...	Informational	Succeeded	5...
Microsoft.Network/public...	>Error	Failed	5...
Microsoft.Network/public...	>Error	Failed	5...
Microsoft.Network/virtu...	Informational	Succeeded	5...
Microsoft.Compute/avail...	Informational	Succeeded	5...

PowerShell

- To get the overall status of a deployment, use the **Get-AzureRmResourceGroupDeployment** command.

```
Get-AzureRmResourceGroupDeployment -ResourceGroupName ExampleGroup
```

Or, you can filter the results for only those deployments that have failed.

```
Get-AzureRmResourceGroupDeployment -ResourceGroupName ExampleGroup | Where-Object ProvisioningState -eq Failed
```

- Each deployment includes multiple operations. Each operation represents a step in the deployment process. To discover what went wrong with a deployment, you usually need to see details about the deployment operations. You can see the status of the operations with **Get-AzureRmResourceGroupDeploymentOperation**.

```
Get-AzureRmResourceGroupDeploymentOperation -ResourceGroupName ExampleGroup -DeploymentName vmDeployment
```

Which returns multiple operations with each one in the following format:

```

Id      :
/subscriptions/{guid}/resourceGroups/ExampleGroup/providers/Microsoft.Resources/deployments/Microsoft.Template/operations/A3EB2DA598E0A780
OperationId   : A3EB2DA598E0A780
Properties    : @{provisioningOperation=Create; provisioningState=Succeeded; timestamp=2016-06-14T21:55:15.0156208Z;
                  duration=PT23.0227078S; trackingId=11d376e8-5d6d-4da8-847e-6f23c6443fbf;
                  serviceRequestId=0196828d-8559-4bf6-b6b8-8b9057cb0e23; statusCode=OK; targetResource=}
PropertiesText : {duration:PT23.0227078S, provisioningOperation:Create, provisioningState:Succeeded,
                  serviceRequestId:0196828d-8559-4bf6-b6b8-8b9057cb0e23...}

```

- To get more details about failed operations, retrieve the properties for operations with **Failed** state.

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName Microsoft.Template -ResourceGroupName ExampleGroup).Properties | Where-Object ProvisioningState -eq Failed
```

Which returns all the failed operations with each one in the following format:

```

provisioningOperation : Create
provisioningState     : Failed
timestamp            : 2016-06-14T21:54:55.1468068Z
duration             : PT3.1449887S
trackingId           : f4ed72f8-4203-43dc-958a-15d041e8c233
serviceRequestId     : a426f689-5d5a-448d-a2f0-9784d14c900a
statusCode           : BadRequest
statusMessage        : @{error=}
targetResource       : @{id=/subscriptions/{guid}/resourceGroups/ExampleGroup/providers/Microsoft.Network/publicIPAddresses/myPublicIP;
                      resourceType=Microsoft.Network/publicIPAddresses; resourceName=myPublicIP}

```

Note the serviceRequestId and the trackingId for the operation. The serviceRequestId can be helpful when working with technical support to troubleshoot a deployment. You will use the trackingId in the next step to focus on a particular operation.

- To get the status message of a particular failed operation, use the following command:

```
((Get-AzureRmResourceGroupDeploymentOperation -DeploymentName Microsoft.Template -ResourceGroupName ExampleGroup).Properties | Where-Object trackingId -eq f4ed72f8-4203-43dc-958a-15d041e8c233).StatusMessage.error
```

Which returns:

code	message	details
---	-----	-----
DnsRecordInUse	DNS record dns.westus.cloudapp.azure.com is already used by another public IP.	{}

- Every deployment operation in Azure includes request and response content. The request content is what you sent to Azure during deployment (for example, create a VM, OS disk, and other resources). The response content is what Azure sent back from your deployment request. During deployment, you can use **DeploymentLogLevel** parameter to specify that the request and/or response are retained in the log.

You get that information from the log, and save it locally by using the following PowerShell commands:

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName "TestDeployment" -ResourceGroupName "Test-RG").Properties.request | ConvertTo-Json | Out-File -FilePath <PathToFile>
```

```
(Get-AzureRmResourceGroupDeploymentOperation -DeploymentName "TestDeployment" -ResourceGroupName "Test-RG").Properties.response | ConvertTo-Json | Out-File -FilePath <PathToFile>
```

Azure CLI

1. Get the overall status of a deployment with the **azure group deployment show** command.

```
azure group deployment show --resource-group ExampleGroup --name ExampleDeployment --json
```

One of the returned values is the **correlationId**. This value is used to track related events, and can be helpful when working with technical support to troubleshoot a deployment.

```
"properties": {  
    "provisioningState": "Failed",  
    "correlationId": "4002062a-a506-4b5e-aaba-4147036b771a",
```

2. To see the operations for a deployment, use:

```
azure group deployment operation list --resource-group ExampleGroup --name ExampleDeployment --json
```

REST

1. Get information about a deployment with the [Get information about a template deployment](#) operation.

```
GET https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group-name}/providers/microsoft.resources/deployments/{deployment-name}?api-version={api-version}
```

In the response, note in particular the **provisioningState**, **correlationId**, and **error** elements. The **correlationId** is used to track related events, and can be helpful when working with technical support to troubleshoot a deployment.

```
{  
    ...  
    "properties": {  
        "provisioningState": "Failed",  
        "correlationId": "d5062e45-6e9f-4fd3-a0a0-6b2c56b15757",  
        ...  
        "error": {  
            "code": "DeploymentFailed", "message": "At least one resource deployment operation failed. Please  
list deployment operations for details. Please see http://aka.ms/arm-debug for usage details.",  
            "details": [{"code": "Conflict", "message": "{\r\n    \"error\": {\r\n        \"message\": \"Conflict\"}\r\n}"}]  
        }  
    }  
}
```

2. Get information about deployment operations with the [List all template deployment operations](#) operation.

```
GET https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group-name}/providers/microsoft.resources/deployments/{deployment-name}/operations?$skiptoken={skiptoken}&api-version={api-version}
```

The response includes request and/or response information based on what you specified in the **debugSetting** property during deployment.

```
{
  ...
  "properties": {
    {
      ...
      "request": {
        "content": {
          "location": "West US",
          "properties": {
            "accountType": "Standard_LRS"
          }
        }
      },
      "response": {
        "content": {
          "error": {
            "message": "Conflict", "code": "Conflict"
          }
        }
      }
    }
  }
}
```

Next steps

- For help with resolving particular deployment errors, see [Resolve common errors when deploying resources to Azure with Azure Resource Manager](#).
- To learn about using the activity logs to monitor other types of actions, see [View activity logs to manage Azure resources](#).
- To validate your deployment before executing it, see [Deploy a resource group with Azure Resource Manager template](#).

Azure Resource Manager template functions

4/14/2017 • 21 min to read • [Edit Online](#)

This topic describes all the functions you can use in an Azure Resource Manager template.

Template functions and their parameters are case-insensitive. For example, Resource Manager resolves **variables('var1')** and **VARIABLES('VAR1')** as the same. When evaluated, unless the function expressly modifies case (such as `toUpperCase` or `toLowerCase`), the function preserves the case. Certain resource types may have case requirements irrespective of how functions are evaluated.

Numeric functions

Resource Manager provides the following functions for working with integers:

- [add](#)
- [copyIndex](#)
- [div](#)
- [int](#)
- [mod](#)
- [mul](#)
- [sub](#)

add

```
add(operand1, operand2)
```

Returns the sum of the two provided integers.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand1	Yes	Integer	First number to add.
operand2	Yes	Integer	Second number to add.

The following example adds two parameters.

```

"parameters": {
    "first": {
        "type": "int",
        "metadata": {
            "description": "First integer to add"
        }
    },
    "second": {
        "type": "int",
        "metadata": {
            "description": "Second integer to add"
        }
    }
},
...
"outputs": {
    "addResult": {
        "type": "int",
        "value": "[add(parameters('first'), parameters('second'))]"
    }
}
}

```

copyIndex

`copyIndex(offset)`

Returns the index of an iteration loop.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
offset	No	Integer	The number to add to the zero-based iteration value.

This function is always used with a **copy** object. If no value is provided for **offset**, the current iteration value is returned. The iteration value starts at zero. For a complete description of how you use **copyIndex**, see [Create multiple instances of resources in Azure Resource Manager](#).

The following example shows a copy loop and the index value included in the name.

```

"resources": [
{
    "name": "[concat('examplecopy-', copyIndex())]",
    "type": "Microsoft.Web/sites",
    "copy": {
        "name": "websitetscopy",
        "count": "[parameters('count')]"
    },
    ...
}
]

```

div

`div(operand1, operand2)`

Returns the integer division of the two provided integers.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand1	Yes	Integer	The number being divided.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand2	Yes	Integer	The number that is used to divide. Cannot be 0.

The following example divides one parameter by another parameter.

```
"parameters": {
  "first": {
    "type": "int",
    "metadata": {
      "description": "Integer being divided"
    }
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Integer used to divide"
    }
  },
  ...
},
"outputs": {
  "divResult": {
    "type": "int",
    "value": "[div(parameters('first'), parameters('second'))]"
  }
}
```

int

```
int(valueToConvert)
```

Converts the specified value to an integer.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
valueToConvert	Yes	String or Integer	The value to convert to an integer.

The following example converts the user-provided parameter value to Integer.

```
"parameters": {
  "appId": { "type": "string" }
},
"variables": {
  "intValue": "[int(parameters('appId'))]"
}
```

mod

```
mod(operand1, operand2)
```

Returns the remainder of the integer division using the two provided integers.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand1	Yes	Integer	The number being divided.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand2	Yes	Integer	The number that is used to divide, Cannot be 0.

The following example returns the remainder of dividing one parameter by another parameter.

```
"parameters": {
  "first": {
    "type": "int",
    "metadata": {
      "description": "Integer being divided"
    }
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Integer used to divide"
    }
  },
  ...
},
"outputs": {
  "modResult": {
    "type": "int",
    "value": "[mod(parameters('first'), parameters('second'))]"
  }
}
```

mul

```
mul(operand1, operand2)
```

Returns the multiplication of the two provided integers.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand1	Yes	Integer	First number to multiply.
operand2	Yes	Integer	Second number to multiply.

The following example multiplies one parameter by another parameter.

```

"parameters": {
    "first": {
        "type": "int",
        "metadata": {
            "description": "First integer to multiply"
        }
    },
    "second": {
        "type": "int",
        "metadata": {
            "description": "Second integer to multiply"
        }
    }
},
...
"outputs": {
    "mulResult": {
        "type": "int",
        "value": "[mul(parameters('first'), parameters('second'))]"
    }
}
}

```

sub

```
sub(operand1, operand2)
```

Returns the subtraction of the two provided integers.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
operand1	Yes	Integer	The number that is subtracted from.
operand2	Yes	Integer	The number that is subtracted.

The following example subtracts one parameter from another parameter.

```

"parameters": {
    "first": {
        "type": "int",
        "metadata": {
            "description": "Integer subtracted from"
        }
    },
    "second": {
        "type": "int",
        "metadata": {
            "description": "Integer to subtract"
        }
    }
},
...
"outputs": {
    "subResult": {
        "type": "int",
        "value": "[sub(parameters('first'), parameters('second'))]"
    }
}
}

```

String functions

Resource Manager provides the following functions for working with strings:

- [base64](#)
- [concat](#)
- [length](#)
- [padLeft](#)
- [replace](#)
- [skip](#)
- [split](#)
- [string](#)
- [substring](#)
- [take](#)
- [toLowerCase](#)
- [toUpperCase](#)
- [trim](#)
- [uniqueString](#)
- [uri](#)

base64

```
base64 (inputString)
```

Returns the base64 representation of the input string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
inputString	Yes	String	The value to return as a base64 representation.

The following example shows how to use the base64 function.

```
"variables": {
    "usernameAndPassword": "[concat(parameters('username'), ':', parameters('password'))]",
    "authorizationHeader": "[concat('Basic ', base64(variables('usernameAndPassword')))]"
}
```

concat - string

```
concat (string1, string2, string3, ...)
```

Combines multiple string values and returns the concatenated string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
string1	Yes	String	The first value for concatenation.
additional strings	No	String	Additional values in sequential order for concatenation.

This function can take any number of arguments, and can accept either strings or arrays for the parameters. For an example of concatenating arrays, see [concat - array](#).

The following example shows how to combine multiple string values to return a concatenated string.

```

"outputs": {
    "siteUri": {
        "type": "string",
        "value": "[concat('http://', reference(resourceId('Microsoft.Web/sites', parameters('siteName'))).hostNames[0])]"
    }
}

```

length - string

`length(string)`

Returns the number of characters in a string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
string	Yes	String	The value to use for getting the number of characters.

For an example of using length with an array, see [length - array](#).

The following example returns the number of characters in a string.

```

"parameters": {
    "appName": { "type": "string" }
},
"variables": {
    "nameLength": "[length(parameters('appName'))]"
}

```

padLeft

`padLeft(valueToPad, totalLength, paddingCharacter)`

Returns a right-aligned string by adding characters to the left until reaching the total specified length.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
valueToPad	Yes	String or Integer	The value to right-align.
totalLength	Yes	Integer	The total number of characters in the returned string.
paddingCharacter	No	Single character	The character to use for left-padding until the total length is reached. The default value is a space.

The following example shows how to pad the user-provided parameter value by adding the zero character until the string reaches 10 characters. If the original parameter value is longer than 10 characters, no characters are added.

```

"parameters": {
    "appName": { "type": "string" }
},
"variables": {
    "paddedAppName": "[padLeft(parameters('appName'),10,'0')]"
}

```

replace

```
replace(originalString, oldCharacter, newCharacter)
```

Returns a new string with all instances of one character in the specified string replaced by another character.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
originalString	Yes	String	The value that has all instances of one character replaced by another character.
oldCharacter	Yes	String	The character to be removed from the original string.
newCharacter	Yes	String	The character to add in place of the removed character.

The following example shows how to remove all dashes from the user-provided string.

```

"parameters": {
    "identifier": { "type": "string" }
},
"variables": {
    "newIdentifier": "[replace(parameters('identifier'),'-',''))]"
}

```

skip - string

```
skip(originalValue, numberToSkip)
```

Returns a string with all the characters after the specified number in the string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
originalValue	Yes	String	The string to use for skipping.
numberToSkip	Yes	Integer	The number of characters to skip. If this value is 0 or less, all the characters in the string are returned. If it is larger than the length of the string, an empty string is returned.

For an example of using skip with an array, see [skip - array](#).

The following example skips the specified number of characters in the string.

```

"parameters": {
  "first": {
    "type": "string",
    "metadata": {
      "description": "Value to use for skipping"
    }
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Number of characters to skip"
    }
  }
},
"resources": [
],
"outputs": {
  "return": {
    "type": "string",
    "value": "[skip(parameters('first'),parameters('second'))]"
  }
}
}

```

split

```
split(inputString, delimiterString)
```

```
split(inputString, delimiterArray)
```

Returns an array of strings that contains the substrings of the input string that are delimited by the specified delimiters.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
inputString	Yes	String	The string to split.
delimiter	Yes	String or Array of strings	The delimiter to use for splitting the string.

The following example splits the input string with a comma.

```

"parameters": {
  "inputString": { "type": "string" }
},
"variables": {
  "stringPieces": "[split(parameters('inputString'), ',')]"
}

```

The next example splits the input string with either a comma or a semi-colon.

```

"variables": {
  "stringToSplit": "test1,test2;test3",
  "delimiters": [ ",", ";" ]
},
"resources": [ ],
"outputs": {
  "exampleOutput": {
    "value": "[split(variables('stringToSplit'), variables('delimiters'))]",
    "type": "array"
  }
}

```

string

<code>string(valueToConvert)</code>

Converts the specified value to a string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
valueToConvert	Yes	Any	The value to convert to string. Any type of value can be converted, including objects and arrays.

The following example converts the user-provided parameter values to strings.

```

"parameters": {
  "jsonObject": {
    "type": "object",
    "defaultValue": {
      "valueA": 10,
      "valueB": "Example Text"
    }
  },
  "jsonArray": {
    "type": "array",
    "defaultValue": [ "a", "b", "c" ]
  },
  "jsonInt": {
    "type": "int",
    "defaultValue": 5
  }
},
"variables": {
  "objectString": "[string(parameters('jsonObject'))]",
  "arrayString": "[string(parameters('jsonArray'))]",
  "intString": "[string(parameters('jsonInt'))]"
}

```

substring

<code>substring(stringToParse, startIndex, length)</code>

Returns a substring that starts at the specified character position and contains the specified number of characters.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
stringToParse	Yes	String	The original string from which the substring is extracted.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
startIndex	No	Integer	The zero-based starting character position for the substring.
length	No	Integer	The number of characters for the substring. Must refer to a location within the string.

The following example extracts the first three characters from a parameter.

```
"parameters": {
    "inputString": { "type": "string" }
},
"variables": {
    "prefix": "[substring(parameters('inputString'), 0, 3)]"
}
```

The following example will fail with the error "The index and length parameters must refer to a location within the string. The index parameter: '0', the length parameter: '11', the length of the string parameter: '10'".

```
"parameters": {
    "inputString": { "type": "string", "value": "1234567890" }
},
"variables": {
    "prefix": "[substring(parameters('inputString'), 0, 11)]"
}
```

take - string

```
take(originalValue, numberToTake)
```

Returns a string with the specified number of characters from the start of the string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
originalValue	Yes	String	The value to take the characters from.
numberToTake	Yes	Integer	The number of characters to take. If this value is 0 or less, an empty string is returned. If it is larger than the length of the given string, all the characters in the string are returned.

For an example of using take with an array, see [take - array](#).

The following example takes the specified number of characters from the string.

```

"parameters": {
  "first": {
    "type": "string",
    "metadata": {
      "description": "Value to use for taking"
    }
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Number of characters to take"
    }
  }
},
"resources": [
],
"outputs": {
  "return": {
    "type": "string",
    "value": "[take(parameters('first')), parameters('second'))]"
  }
}
}

```

toLowerCase

<code>toLowerCase(stringToChange)</code>
--

Converts the specified string to lower case.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
stringToChange	Yes	String	The value to convert to lower case.

The following example converts the user-provided parameter value to lower case.

```

"parameters": {
  "appName": { "type": "string" }
},
"variables": {
  "lowerCaseAppName": "[toLowerCase(parameters('appName'))]"
}

```

toUpperCase

<code>toUpperCase(stringToChange)</code>
--

Converts the specified string to upper case.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
stringToChange	Yes	String	The value to convert to upper case.

The following example converts the user-provided parameter value to upper case.

```

"parameters": {
    "appName": { "type": "string" }
},
"variables": {
    "upperCaseAppName": "[toUpperCase(parameters('appName'))]"
}

```

trim

```
trim (stringToTrim)
```

Removes all leading and trailing white-space characters from the specified string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
stringToTrim	Yes	String	The value to trim.

The following example trims the white-space characters from the user-provided parameter value.

```

"parameters": {
    "appName": { "type": "string" }
},
"variables": {
    "trimAppName": "[trim(parameters('appName'))]"
}

```

uniqueString

```
uniqueString (baseString, ...)
```

Creates a deterministic hash string based on the values provided as parameters.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
baseString	Yes	String	The value used in the hash function to create a unique string.
additional parameters as needed	No	String	You can add as many strings as needed to create the value that specifies the level of uniqueness.

This function is helpful when you need to create a unique name for a resource. You provide parameter values that limit the scope of uniqueness for the result. You can specify whether the name is unique down to subscription, resource group, or deployment.

The returned value is not a random string, but rather the result of a hash function. The returned value is 13 characters long. It is not globally unique. You may want to combine the value with a prefix from your naming convention to create a name that is meaningful. The following example shows the format of the returned value. The actual value varies by the provided parameters.

```
tcvhiyu5h2o5o
```

The following examples show how to use uniqueString to create a unique value for commonly used levels.

Unique scoped to subscription

```
"[uniqueString(subscription().subscriptionId)]"
```

Unique scoped to resource group

```
"[uniqueString(resourceGroup().id)]"
```

Unique scoped to deployment for a resource group

```
"[uniqueString(resourceGroup().id, deployment().name)]"
```

The following example shows how to create a unique name for a storage account based on your resource group. Inside the resource group, the name is not unique if constructed the same way.

```
"resources": [{}  
    "name": "[concat('storage', uniqueString(resourceGroup().id))]",  
    "type": "Microsoft.Storage/storageAccounts",  
    ...
```

uri

```
uri (baseUri, relativeUri)
```

Creates an absolute URI by combining the baseUri and the relativeUri string.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
baseUri	Yes	String	The base uri string.
relativeUri	Yes	String	The relative uri string to add to the base uri string.

The value for the **baseUri** parameter can include a specific file, but only the base path is used when constructing the URI. For example, passing `http://contoso.com/resources/azuredeploy.json` as the baseUri parameter results in a base URI of `http://contoso.com/resources/`.

The following example shows how to construct a link to a nested template based on the value of the parent template.

```
"templateLink": "[uri(deployment().properties.templateLink.uri, 'nested/azuredeploy.json')]"
```

Array functions

Resource Manager provides several functions for working with array values.

- [concat](#)
- [length](#)
- [skip](#)
- [take](#)

To get an array of string values delimited by a value, see [split](#).

concat - array

```
concat (array1, array2, array3, ...)
```

Combines multiple arrays and returns the concatenated array.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
array1	Yes	Array	The first array for concatenation.
additional arrays	No	Array	Additional arrays in sequential order for concatenation.

This function can take any number of arguments, and can accept either strings or arrays for the parameters. For an example of concatenating string values, see [concat - string](#).

The following example shows how to combine two arrays.

```
"parameters": {
    "firstarray": {
        "type": "array"
    }
    "secondarray": {
        "type": "array"
    }
},
"variables": {
    "combinedarray": "[concat(parameters('firstarray'), parameters('secondarray'))]"
}
```

length - array

```
length(array)
```

Returns the number of elements in an array.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
array	Yes	Array	The array to use for getting the number of elements.

You can use this function with an array to specify the number of iterations when creating resources. In the following example, the parameter **siteNames** would refer to an array of names to use when creating the web sites.

```
"copy": {
    "name": "websitetescopy",
    "count": "[length(parameters('siteNames'))]"
}
```

For more information about using this function with an array, see [Create multiple instances of resources in Azure Resource Manager](#).

For an example of using length with a string value, see [length - string](#).

skip - array

```
skip(originalValue, numberToSkip)
```

Returns an array with all the elements after the specified number in the array.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
originalValue	Yes	Array	The array to use for skipping.
numberToSkip	Yes	Integer	The number of elements to skip. If this value is 0 or less, all the elements in the array are returned. If it is larger than the length of the array, an empty array is returned.

For an example of using skip with a string, see [skip - string](#).

The following example skips the specified number of elements in the array.

```
"parameters": {
  "first": {
    "type": "array",
    "metadata": {
      "description": "Values to use for skipping"
    },
    "defaultValue": [ "one", "two", "three" ]
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Number of elements to skip"
    }
  }
},
"resources": [
],
"outputs": {
  "return": {
    "type": "array",
    "value": "[skip(parameters('first'), parameters('second'))]"
  }
}
```

take - array

```
take(originalValue, numberToTake)
```

Returns an array with the specified number of elements from the start of the array.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
originalValue	Yes	Array	The array to take the elements from.
numberToTake	Yes	Integer	The number of elements to take. If this value is 0 or less, an empty array is returned. If it is larger than the length of the given array, all the elements in the array are returned.

For an example of using take with a string, see [take - string](#).

The following example takes the specified number of elements from the array.

```
"parameters": {
  "first": {
    "type": "array",
    "metadata": {
      "description": "Values to use for taking"
    },
    "defaultValue": [ "one", "two", "three" ]
  },
  "second": {
    "type": "int",
    "metadata": {
      "description": "Number of elements to take"
    }
  }
},
"resources": [
],
"outputs": {
  "return": {
    "type": "array",
    "value": "[take(parameters('first'),parameters('second'))]"
  }
}
```

Deployment value functions

Resource Manager provides the following functions for getting values from sections of the template and values related to the deployment:

- [deployment](#)
- [parameters](#)
- [variables](#)

To get values from resources, resource groups, or subscriptions, see [Resource functions](#).

deployment

```
deployment()
```

Returns information about the current deployment operation.

This function returns the object that is passed during deployment. The properties in the returned object differ based on whether the deployment object is passed as a link or as an in-line object.

When the deployment object is passed in-line, such as when using the **-TemplateFile** parameter in Azure PowerShell to point to a local file, the returned object has the following format:

```
{
  "name": "",
  "properties": {
    "template": {
      "$schema": "",
      "contentVersion": "",
      "parameters": {},
      "variables": {},
      "resources": [
      ],
      "outputs": {}
    },
    "parameters": {},
    "mode": "",
    "provisioningState": ""
  }
}
```

When the object is passed as a link, such as when using the **-TemplateUri** parameter to point to a remote object, the object is returned in the following format:

```
{
  "name": "",
  "properties": {
    "templateLink": {
      "uri": ""
    },
    "template": {
      "$schema": "",
      "contentVersion": "",
      "parameters": {},
      "variables": {},
      "resources": [],
      "outputs": {}
    },
    "parameters": {},
    "mode": "",
    "provisioningState": ""
  }
}
```

The following example shows how to use deployment() to link to another template based on the URI of the parent template.

```
"variables": {
  "sharedTemplateUrl": "[uri(deployment().properties.templateLink.uri, 'shared-resources.json')]"
}
```

parameters

`parameters (parameterName)`

Returns a parameter value. The specified parameter name must be defined in the parameters section of the template.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
parameterName	Yes	String	The name of the parameter to return.

The following example shows a simplified use of the parameters function.

```
"parameters": {  
    "siteName": {  
        "type": "string"  
    }  
},  
"resources": [  
    {  
        "apiVersion": "2014-06-01",  
        "name": "[parameters('siteName')]",  
        "type": "Microsoft.Web/Sites",  
        ...  
    }  
]
```

variables

```
variables (variableName)
```

Returns the value of variable. The specified variable name must be defined in the variables section of the template.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
variableName	Yes	String	The name of the variable to return.

The following example uses a variable value.

```
"variables": {  
    "storageName": "[concat('storage', uniqueString(resourceGroup().id))]"  
},  
"resources": [  
    {  
        "type": "Microsoft.Storage/storageAccounts",  
        "name": "[variables('storageName')]",  
        ...  
    }  
],
```

Resource functions

Resource Manager provides the following functions for getting resource values:

- [listKeys and list{Value}](#)
- [providers](#)
- [reference](#)
- [resourceGroup](#)
- [resourceId](#)
- [subscription](#)

To get values from parameters, variables, or the current deployment, see [Deployment value functions](#).

listKeys and list{Value}

```
listKeys (resourceName or resourceIdentifier, apiVersion)
```

```
list{Value} (resourceName or resourceIdentifier, apiVersion)
```

Returns the values for any resource type that supports the list operation. The most common usage is **listKeys**.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
resourceName or resourceId	Yes	String	Unique identifier for the resource.
apiVersion	Yes	String	API version of resource runtime state. Typically, in the format, yyyy-mm-dd .

Any operation that starts with **list** can be used as a function in your template. The available operations include not only **listKeys**, but also operations like **list**, **listAdminKeys**, and **listStatus**. To determine which resource types have a list operation, see the [REST API operations](#) for the resource provider.

To find the list operations for a resource provider, use the following PowerShell cmdlet:

```
Get-AzureRmProviderOperation -OperationSearchString "Microsoft.Storage/*" | where {$_.Operation -like "*list*"} | FT Operation
```

To find the list operations for a resource provider, use the following Azure CLI command, and the JSON utility [jq](#) to filter only the list operations:

```
azure provider operations show --operationSearchString */apiapps/* --json | jq ".[] | select (.operation | contains(\"list\"))"
```

The resourceId can be specified by using the [resourceId function](#) or by using the format **{providerNamespace}/{resourceType}/{resourceName}**.

The following example shows how to return the primary and secondary keys from a storage account in the outputs section.

```
"outputs": {
  "listKeysOutput": {
    "value": "[listKeys(resourceId('Microsoft.Storage/storageAccounts', parameters('storageAccountName')), '2016-01-01')]",
    "type" : "object"
  }
}
```

The returned object from **listKeys** has the following format:

```
{
  "keys": [
    {
      "keyName": "key1",
      "permissions": "Full",
      "value": "{value}"
    },
    {
      "keyName": "key2",
      "permissions": "Full",
      "value": "{value}"
    }
  ]
}
```

```
providers (providerNamespace, [resourceType])
```

Returns information about a resource provider and its supported resource types. If you do not provide a resource type, the function returns all the supported types for the resource provider.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
providerNamespace	Yes	String	Namespace of the provider
resourceType	No	String	The type of resource within the specified namespace.

Each supported type is returned in the following format. Array ordering is not guaranteed.

```
{
    "resourceType": "",
    "locations": [ ],
    "apiVersions": [ ]
}
```

The following example shows how to use the provider function:

```
"outputs": {
    "exampleOutput": {
        "value": "[providers('Microsoft.Storage', 'storageAccounts')]",
        "type" : "object"
    }
}
```

reference

```
reference (resourceName or resourceIdentifier, [apiVersion])
```

Returns an object representing another resource's runtime state.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
resourceName or resourceIdentifier	Yes	String	Name or unique identifier of a resource.
apiVersion	No	String	API version of the specified resource. Include this parameter when the resource is not provisioned within same template. Typically, in the format, yyyy-mm-dd .

The **reference** function derives its value from a runtime state, and therefore cannot be used in the variables section. It can be used in outputs section of a template.

By using the reference function, you implicitly declare that one resource depends on another resource if the referenced resource is provisioned within same template. You do not need to also use the **dependsOn** property. The function is not evaluated until the referenced resource has completed deployment.

The following example references a storage account that is deployed in the same template.

```
"outputs": {
    "NewStorage": {
        "value": "[reference(parameters('storageAccountName'))]",
        "type" : "object"
    }
}
```

The following example references a storage account that is not deployed in this template, but exists within the same resource group as the resources being deployed.

```
"outputs": {
    "ExistingStorage": {
        "value": "[reference(concat('Microsoft.Storage/storageAccounts/', parameters('storageAccountName')), '2016-01-01'))]",
        "type" : "object"
    }
}
```

You can retrieve a particular value from the returned object, such as the blob endpoint URI, as shown in the following example:

```
"outputs": {
    "BlobUri": {
        "value": "[reference(concat('Microsoft.Storage/storageAccounts/', parameters('storageAccountName')), '2016-01-01').primaryEndpoints.blob]",
        "type" : "string"
    }
}
```

The following example references a storage account in a different resource group.

```
"outputs": {
    "BlobUri": {
        "value": "[reference(resourceId(parameters('relatedGroup'), 'Microsoft.Storage/storageAccounts/' , parameters('storageAccountName')), '2016-01-01').primaryEndpoints.blob]",
        "type" : "string"
    }
}
```

The properties on the object returned from the **reference** function vary by resource type. To see the property names and values for a resource type, create a simple template that returns the object in the **outputs** section. If you have an existing resource of that type, your template just returns the object without deploying any new resources. If you do not have an existing resource of that type, your template deploys only that type and returns the object. Then, add those properties to other templates that need to dynamically retrieve the values during deployment.

resourceGroup

```
resourceGroup()
```

Returns an object that represents the current resource group.

The returned object is in the following format:

```
{
  "id": "/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}",
  "name": "{resourceGroupName}",
  "location": "{resourceGroupLocation}",
  "tags": {},
  "properties": {
    "provisioningState": "{status}"
  }
}
```

The following example uses the resource group location to assign the location for a web site.

```
"resources": [
  {
    "apiVersion": "2014-06-01",
    "type": "Microsoft.Web/sites",
    "name": "[parameters('siteName')]",
    "location": "[resourceGroup().location]",
    ...
  }
]
```

resourceId

```
resourceId ([subscriptionId], [resourceGroupName], resourceType, resourceName1, [resourceName2]...)
```

Returns the unique identifier of a resource.

PARAMETER	REQUIRED	TYPE	DESCRIPTION
subscriptionId	No	String (In GUID format)	Default value is the current subscription. Specify this value when you need to retrieve a resource in another subscription.
resourceGroupName	No	String	Default value is current resource group. Specify this value when you need to retrieve a resource in another resource group.
resourceType	Yes	String	Type of resource including resource provider namespace.
resourceName1	Yes	String	Name of resource.
resourceName2	No	String	Next resource name segment if resource is nested.

You use this function when the resource name is ambiguous or not provisioned within the same template. The identifier is returned in the following format:

```
/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/{resourceProviderNamespace}/{resourceType}
/{resourceName}
```

The following example shows how to retrieve the resource ids for a web site and a database. The web site exists in a resource group named **myWebsitesGroup** and the database exists in the current resource group for this template.

```
[resourceId('myWebsitesGroup', 'Microsoft.Web/sites', parameters('siteName'))]
[resourceId('Microsoft.SQL/servers/databases', parameters('serverName'), parameters('databaseName'))]
```

Often, you need to use this function when using a storage account or virtual network in an alternate resource group. The storage account or virtual network may be used across multiple resource groups; therefore, you do not want to delete them when deleting a single resource group. The following example shows how a resource from an external resource group can easily be used:

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "virtualNetworkName": {
      "type": "string"
    },
    "virtualNetworkResourceGroup": {
      "type": "string"
    },
    "subnet1Name": {
      "type": "string"
    },
    "nicName": {
      "type": "string"
    }
  },
  "variables": {
    "vnetID": "[resourceId(parameters('virtualNetworkResourceGroup'), 'Microsoft.Network/virtualNetworks', parameters('virtualNetworkName'))]",
    "subnet1Ref": "[concat(variables('vnetID'), '/subnets/', parameters('subnet1Name'))]"
  },
  "resources": [
    {
      "apiVersion": "2015-05-01-preview",
      "type": "Microsoft.Network/networkInterfaces",
      "name": "[parameters('nicName')]",
      "location": "[parameters('location')]",
      "properties": {
        "ipConfigurations": [
          {
            "name": "ipconfig1",
            "properties": {
              "privateIPAllocationMethod": "Dynamic",
              "subnet": {
                "id": "[variables('subnet1Ref')]"
              }
            }
          }
        ]
      }
    }
  ]
}
```

subscription

```
subscription()
```

Returns details about the subscription in the following format:

```
{  
  "id": "/subscriptions/#####",  
  "subscriptionId": "#####",  
  "tenantId": "#####"  
}
```

The following example shows the subscription function called in the outputs section.

```
"outputs": {  
  "exampleOutput": {  
    "value": "[subscription()]",  
    "type" : "object"  
  }  
}
```

Next Steps

- For a description of the sections in an Azure Resource Manager template, see [Authoring Azure Resource Manager templates](#)
- To merge multiple templates, see [Using linked templates with Azure Resource Manager](#)
- To iterate a specified number of times when creating a type of resource, see [Create multiple instances of resources in Azure Resource Manager](#)
- To see how to deploy the template you have created, see [Deploy an application with Azure Resource Manager template](#)

Throttling Resource Manager requests

1/17/2017 • 2 min to read • [Edit Online](#)

For each subscription and tenant, Resource Manager limits read requests to 15,000 per hour and write requests to 1,200 per hour. If your application or script reaches these limits, you need to throttle your requests. This topic shows you how to determine the remaining requests you have before reaching the limit, and how to respond when you have reached the limit.

When you reach the limit, you receive the HTTP status code **429 Too many requests**.

The number of requests is scoped to either your subscription or your tenant. If you have multiple, concurrent applications making requests in your subscription, the requests from those applications are added together to determine the number of remaining requests.

Subscription scoped requests are ones that involve passing your subscription id, such as retrieving the resource groups in your subscription. Tenant scoped requests do not include your subscription id, such as retrieving valid Azure locations.

Remaining requests

You can determine the number of remaining requests by examining response headers. Each request includes values for the number of remaining read and write requests. The following table describes the response headers you can examine for those values:

RESPONSE HEADER	DESCRIPTION
x-ms-ratelimit-remaining-subscription-reads	Subscription scoped reads remaining
x-ms-ratelimit-remaining-subscription-writes	Subscription scoped writes remaining
x-ms-ratelimit-remaining-tenant-reads	Tenant scoped reads remaining
x-ms-ratelimit-remaining-tenant-writes	Tenant scoped writes remaining
x-ms-ratelimit-remaining-subscription-resource-requests	Subscription scoped resource type requests remaining. This header value is only returned if a service has overridden the default limit. Resource Manager adds this value instead of the subscription reads or writes.
x-ms-ratelimit-remaining-subscription-resource-entities-read	Subscription scoped resource type collection requests remaining. This header value is only returned if a service has overridden the default limit. This value provides the number of remaining collection requests (list resources).
x-ms-ratelimit-remaining-tenant-resource-requests	Tenant scoped resource type requests remaining. This header is only added for requests at tenant level, and only if a service has overridden the default limit. Resource Manager adds this value instead of the tenant reads or writes.

RESPONSE HEADER	DESCRIPTION
x-ms-ratelimit-remaining-tenant-resource-entities-read	Tenant scoped resource type collection requests remaining. This header is only added for requests at tenant level, and only if a service has overridden the default limit.

Retrieving the header values

Retrieving these header values in your code or script is no different than retrieving any header value.

For example, in **C#**, you retrieve the header value from an **HttpWebResponse** object named **response** with the following code:

```
response.Headers.GetValues("x-ms-ratelimit-remaining-subscription-reads").GetValue(0)
```

In **PowerShell**, you retrieve the header value from an **Invoke-WebRequest** operation.

```
$r = Invoke-WebRequest -Uri https://management.azure.com/subscriptions/{guid}/resourcegroups?api-version=2016-09-01 -Method GET -Headers $authHeaders
$r.Headers["x-ms-ratelimit-remaining-subscription-reads"]
```

Or, if want to see the remaining requests for debugging, you can provide the **-Debug** parameter on your **PowerShell** cmdlet.

```
Get-AzureRmResourceGroup -Debug
```

Which returns many values, including the following response value:

```
...
DEBUG: ===== HTTP RESPONSE =====

Status Code:
OK

Headers:
Pragma : no-cache
x-ms-ratelimit-remaining-subscription-reads: 14999
...
```

In **Azure CLI**, you retrieve the header value by using the more verbose option.

```
azure group list -vv --json
```

Which returns many values, including the following object:

```
...
silly: returnObject
{
  "statusCode": 200,
  "header": {
    "cache-control": "no-cache",
    "pragma": "no-cache",
    "content-type": "application/json; charset=utf-8",
    "expires": "-1",
    "x-ms-ratelimit-remaining-subscription-reads": "14998",
    ...
  }
}
```

Waiting before sending next request

When you reach the request limit, Resource Manager returns the **429** HTTP status code and a **Retry-After** value in the header. The **Retry-After** value specifies the number of seconds your application should wait (or sleep) before sending the next request. If you send a request before the retry value has elapsed, your request is not processed and a new retry value is returned.

Next steps

- For more information about limits and quotas, see [Azure subscription and service limits, quotas, and constraints](#).
- To learn about handling asynchronous REST requests, see [Track asynchronous Azure operations](#).

Track asynchronous Azure operations

1/17/2017 • 4 min to read • [Edit Online](#)

Some Azure REST operations run asynchronously because the operation cannot be completed quickly. This topic describes how to track the status of asynchronous operations through values returned in the response.

Status codes for asynchronous operations

An asynchronous operation initially returns an HTTP status code of either:

- 201 (Created)
- 202 (Accepted)

When the operation successfully completes, it returns either:

- 200 (OK)
- 204 (No Content)

Refer to the [REST API documentation](#) to see the responses for the operation you are executing.

Monitor status of operation

The asynchronous REST operations return header values, which you use to determine the status of the operation. There are potentially three header values to examine:

- `Azure-AsyncOperation` - URL for checking the ongoing status of the operation. If your operation returns this value, always use it (instead of Location) to track the status of the operation.
- `Location` - URL for determining when an operation has completed. Use this value only when Azure-AsyncOperation is not returned.
- `Retry-After` - The number of seconds to wait before checking the status of the asynchronous operation.

However, not every asynchronous operation returns all these values. For example, you may need to evaluate the Azure-AsyncOperation header value for one operation, and the Location header value for another operation.

You retrieve the header values as you would retrieve any header value for a request. For example, in C#, you retrieve the header value from an `HttpWebResponse` object named `response` with the following code:

```
response.Headers.GetValues("Azure-AsyncOperation").GetValue(0)
```

Azure-AsyncOperation request and response

To get the status of the asynchronous operation, send a GET request to the URL in Azure-AsyncOperation header value.

The body of the response from this operation contains information about the operation. The following example shows the possible values returned from the operation:

```
{
  "id": "{resource path from GET operation}",
  "name": "{operation-id}",
  "status" : "Succeeded | Failed | Canceled | {resource provider values}",
  "startTime": "2017-01-06T20:56:36.002812+00:00",
  "endTime": "2017-01-06T20:56:36.002812+00:00",
  "percentComplete": {double between 0 and 100 },
  "properties": {
    /* Specific resource provider values for successful operations */
  },
  "error" : {
    "code": "{error code}",
    "message": "{error description}"
  }
}
```

Only `status` is returned for all responses. The error object is returned when the status is Failed or Canceled. All other values are optional; therefore, the response you receive may look different than the example.

provisioningState values

Operations that create, update, or delete (PUT, PATCH, DELETE) a resource typically return a `provisioningState` value. When an operation has completed, one of following three values is returned:

- Succeeded
- Failed
- Canceled

All other values indicate the operation is still running. The resource provider can return a customized value that indicates its state. For example, you may receive **Accepted** when the request is received and running.

Example requests and responses

Start virtual machine (202 with Azure-AsyncOperation)

This example shows how to determine the status of **start** operation for virtual machines. The initial request is in the following format:

```
POST  
https://management.azure.com/subscriptions/{subscription-id}/resourceGroups/{resource-group}/providers/Microsoft.Compute/virtualMachines/{vm-name}/start?api-version=2016-03-30
```

It returns status code 202. Among the header values, you see:

```
Azure-AsyncOperation : https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.Compute/locations/{region}/operations/{operation-id}?api-version=2016-03-30
```

To check the status of the asynchronous operation, sending another request to that URL.

```
GET  
https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.Compute/locations/{region}/operations/{operation-id}?api-version=2016-03-30
```

The response body contains the status of the operation:

```
{  
  "startTime": "2017-01-06T18:58:24.7596323+00:00",  
  "status": "InProgress",  
  "name": "9a062a88-e463-4697-bef2-fe039df73a02"  
}
```

Deploy resources (201 with Azure-AsyncOperation)

This example shows how to determine the status of **deployments** operation for deploying resources to Azure.

The initial request is in the following format:

```
PUT  
https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group}/providers/microsoft.resources/deployments/{deployment-name}?api-version=2016-09-01
```

It returns status code 201. The body of the response includes:

```
"provisioningState": "Accepted",
```

Among the header values, you see:

```
Azure-AsyncOperation: https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group}/providers/Microsoft.Resources/deployments/{deployment-name}/operationStatuses/{operation-id}?api-version=2016-09-01
```

To check the status of the asynchronous operation, sending another request to that URL.

```
GET  
https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group}/providers/Microsoft.Resources/deployments/{deployment-name}/operationStatuses/{operation-id}?api-version=2016-09-01
```

The response body contains the status of the operation:

```
{"status": "Running"}
```

When the deployment is finished, the response contains:

```
{"status": "Succeeded"}
```

Create storage account (202 with Location and Retry-After)

This example shows how to determine the status of the **create** operation for storage accounts. The initial request is in the following format:

```
PUT  
https://management.azure.com/subscriptions/{subscription-id}/resourceGroups/{resource-group}/providers/Microsoft.Storage/storageAccounts/{storage-name}?api-version=2016-01-01
```

And the request body contains properties for the storage account:

```
{ "location": "South Central US", "properties": {}, "sku": { "name": "Standard_LRS" }, "kind": "Storage" }
```

It returns status code 202. Among the header values, you see the following two values:

```
Location: https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.Storage/operations/{operation-id}?monitor=true&api-version=2016-01-01
Retry-After: 17
```

After waiting for number of seconds specified in Retry-After, check the status of the asynchronous operation by sending another request to that URL.

```
GET
https://management.azure.com/subscriptions/{subscription-id}/providers/Microsoft.Storage/operations/{operation-id}?monitor=true&api-version=2016-01-01
```

If the request is still running, you receive a status code 202. If the request has completed, you receive a status code 200, and the body of the response contains the properties of the storage account that has been created.

Next steps

- For documentation about each REST operation, see [REST API documentation](#).
- For information about managing resources through the Resource Manager REST API, see [Using the Resource Manager REST API](#).
- for information about deploying templates through the Resource Manager REST API, see [Deploy resources with Resource Manager templates and Resource Manager REST API](#).