



Getting started with Azure Resource Manager

Author: Marc van Eijk

Reviewers: Mark Russinovich, John Gossman, Ryan Jones, Ralph Squillace

Published: September 2016, Version 1.0

Abstract

This whitepaper is intended for IT Professionals and Developers. Whether you currently have an existing Infrastructure as Code solution, you are evaluating the available solutions or already decided to start with Microsoft Azure services, this whitepaper is written for you. The content covers the end to end process for creating, deploying and managing applications with Azure Resource Manager, the foundation of the Microsoft cloud.

This whitepaper applies to Microsoft Azure and Microsoft Azure Stack.

© 2016 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples are for illustration only and are fictitious. No real association is intended or inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Contents

| | |
|--|----|
| Overview | 5 |
| Microsoft cloud platform | 6 |
| End-user experiences | 6 |
| Unified Application model | 7 |
| Extensible Services Framework | 7 |
| Cloud Infrastructure | 7 |
| Infrastructure as Code | 8 |
| Imperative and declarative | 9 |
| Azure Resource Manager | 10 |
| Resources | 10 |
| Resource Groups | 10 |
| Role based access control | 11 |
| Custom Policies | 11 |
| Tagging | 11 |
| Template Orchestration | 12 |
| Auditing | 12 |
| Consistency | 12 |
| Template authoring | 13 |
| Language | 13 |
| Element | 14 |
| Tooling | 14 |
| Versioning control | 15 |
| Adding resources to your template | 15 |
| Adding parameters to your template | 16 |
| Adding variables to your template | 17 |
| Template functions | 18 |
| Common examples of template function usage | 18 |
| Adding outputs to your template | 19 |
| Template deployment | 21 |
| Parameter file | 22 |
| Tooling | 22 |
| Troubleshooting | 26 |

| | |
|--|----|
| Template reusability across clouds..... | 29 |
| Endpoints | 29 |
| Resource location..... | 29 |
| API versions..... | 30 |
| Resource provider availability..... | 30 |
| Adding additional resources to the template..... | 32 |
| Dependencies..... | 32 |
| Inside a resource..... | 32 |
| Azure Quickstart Templates..... | 33 |
| More advanced scenarios | 34 |
| Decomposing templates | 34 |
| Multiple Resource Groups | 34 |
| Summary | 36 |

Overview

The cloud is one of the biggest changes in our industry. Regardless of your role in IT, the cloud impacts us all. It introduces new capabilities, opens new opportunities and involves different cost models. Businesses demand extreme agility and the cloud can help organizations to achieve that level of agility.

Transitioning to a cloud model requires investment in time and resources. How do you know if you are making the right investments? Consistency across clouds and physical locations makes it possible to not only see cloud as a model, but to leverage your investment in that model and have return on investment everywhere. By minimizing the investments and maximizing the opportunities from that investment, you can meet the agile demand from your business and achieve more.

This whitepaper explains how to get started with the infrastructure as code capabilities in Azure Resource Manager. Azure Resource Manager allows you to make your investment once and reuse it across the hyper-scale public cloud, customer datacenters and service providers. Microsoft is unique with a cloud platform that is consistent across physical boundaries, proven in the hyper-scale public cloud Microsoft Azure and powering datacenters in private and hosted clouds with Microsoft Azure Stack.

Starting with an introduction to the Microsoft cloud platform and some background on Infrastructure as Code, this whitepaper will cover the capabilities of Azure Resource Manager. Throughout this whitepaper we will make use of a single resource (a storage account) as a basic example. This example will explain how to author a template, ensure reusability of a template across clouds, deploy a template and troubleshoot deployments. The basic example can be replaced with any resources that compose your application and the same scenarios will still be applicable. At the end of this whitepaper, more advanced scenarios will be addressed.

Considering the versatile topic, the whitepaper is intentionally kept relatively short to ensure you can use this whitepaper to get started with Azure Resource Manager quickly. Where possible the document contains references to more detailed information on a topic. External references are formatted with an icon and a hyperlink.

 <https://azure.microsoft.com/en-us/features/resource-manager/>

Microsoft cloud platform

Microsoft provides a consistent cloud platform that unifies how cloud services operate in Microsoft Azure, enterprise datacenters and service provider datacenters. Microsoft Azure provides hyper-scale public cloud services for the enterprise. Microsoft Azure Stack brings consistent Azure services to the datacenter, allowing organizations to deliver Azure services from their datacenter or service provider to run these services on their behalf. This consistency enables powerful hybrid scenarios;

- Applications can span clouds, using the elasticity and scalability of the public cloud while sensitive data remains in the organization owned datacenters
- Moving applications between clouds based on capacity requirements
- Continues delivery across clouds

The consistent cloud platform provided by Microsoft can be divided into four layers.

- End-user experiences
- Unified Application model
- Services
- Cloud infrastructure

End-user experiences

Microsoft enables Developers and IT Pros to continue to use the tools they are comfortable with, both in Microsoft and open source communities. By embracing the technologies that the customers use today and enabling integration points on the cloud platform for those technologies, a wide variety of tools and interfaces can be used to interact with the platform.



Portal



Cross Platform CLI



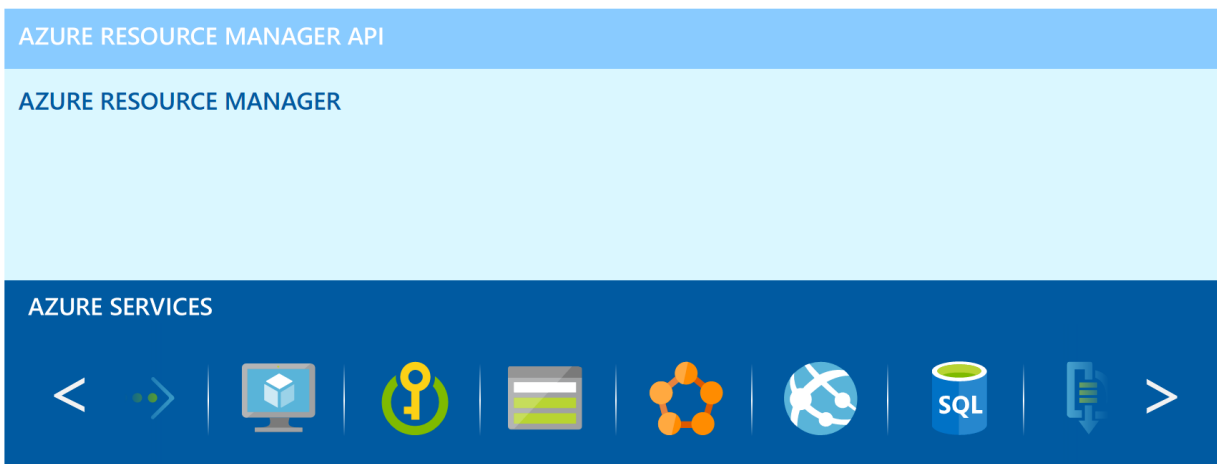
PowerShell



Client Libraries



Visual Studio



Unified Application model

At the core of Microsoft cloud platform, Azure Resource Manager provides a RESTful API that allows the wide variety of user interfaces to communicate with the platform. The capabilities in Azure Resource Manager ensure that the tenant experience is consistent regardless what tool is used. Azure Resource Manager applies a set of policies and features at its core, ensuring all user interfaces can leverage the same capabilities and are restricted by the same policies.

Extensible Services Framework

The Microsoft cloud platform provides an extensive range of IaaS and PaaS services. The advantage of the cloud is that new services are introduced and existing services are updated with new features very frequently, allowing organizations to benefit from the new capabilities without making any investments in the platform itself. These services form the building blocks for any type of application, either available as “ready to deploy” items from the marketplace or created by yourself with tooling available cross-platform.

Cloud Infrastructure

Hyper-scale datacenters power the Microsoft cloud platform. The scalability, stability and reliability of the cloud infrastructure can be utilized without any upfront investment in hardware. If you prefer to run Azure services from your own datacenter, you can power your datacenter with Microsoft Azure Stack to leverage the knowledge learned by Microsoft from running Azure services at hyper-scale.

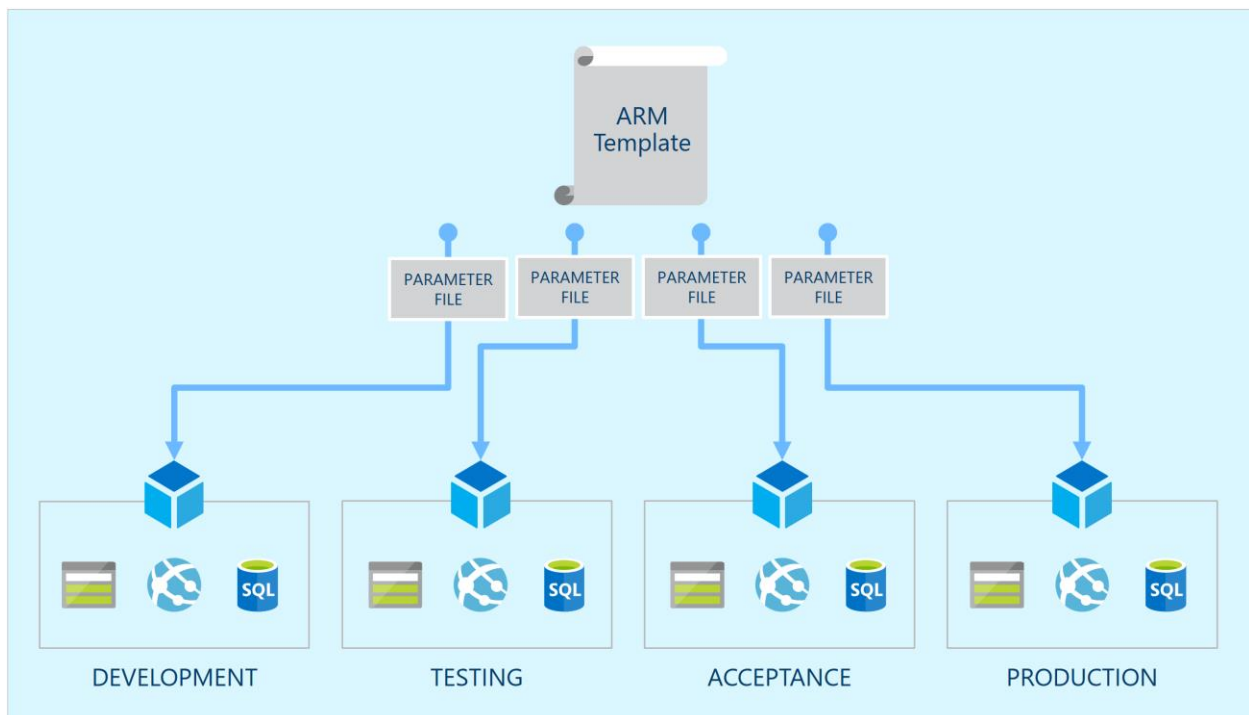
Infrastructure as Code

Organizations need to manage and operate their existing applications in a modern way, while also taking advantage of the cloud model whenever possible. This transformation shifts the organization from a traditional model to a cloud model.

In the traditional model applications are configured manually; with scripts, user interfaces, management utilities or likely a combination of all these tools. The result of this process is an application that is released to a specific environment. During the lifecycle of the application different tooling manages different aspects of the application. When changes are made, they are performed in the management tooling, locally on the resource or in related resources that may be shared with other applications. Deploying two identical instances of an application with the same scripts on the same date, will most likely not have identical configurations over their complete lifecycle, even if that is the desired state.

To overcome these challenges, the concept of Infrastructure as Code was introduced. It allows you to define the desired state of your application in a template. The template is then used to deploy the application. The template provides the ability to repeat a deployment exactly but it can also ensure that a deployed application stays consistent with the desired state defined in the template over time. If you want to make a change to the application, you would make that change in the template. The template can then be used to apply the desired state to the existing application instance over its complete lifecycle.

Templates can be parameterized; creating a reusable artifact that is used to deploy the same application to different environments, accepting the relevant parameters for each environment.



Imperative and declarative

Deploying and configuring resources can be a challenging task. An average application that is running in production has a complex architecture with many configuration settings. For a single deployment it might seem more appealing and time effective to configure all these settings manually instead of investing in automation, but it is inevitable that a shortcut will eventually be inefficient.

In general, we can distinguish two different programming types.

- **Imperative syntax** describes the how. It requires you to think about how to configure all the individual components of the application, and define that in a programming language.
- **Declarative syntax** describes the what. It requires you to define the desired state of your application and let a system determine the most efficient way to reach that state.

In the remainder of this whitepaper is explained how the Azure Resource Manager API accepts both declarative and imperative programming.

Azure Resource Manager

The Azure Resource Manager API is flexible allowing a range of different tools and languages to interact with the platform. Azure Resource Manager provides resource groups, template orchestration, role based access control, custom policies, tagging and auditing features to make it easier for you to build and manage your applications.

Resources

The Microsoft cloud platform delivers IaaS and PaaS services. These services are referred to as resources in the ARM model. A typical application consists of various resources in a defined composition. Each type of resource is managed by a resource provider. Each resource provider serves resources of a similar type.

For example, the resource provider for networking serves resource types like virtual networks, network interfaces and load balancers. The resource provider for storage serves a storage accounts resource type. Each resource provider implements an API to Azure Resource Manager that complies with a common contract, therefore allowing a consistent unified application model across all resource providers. When a new resource or even a completely new resource provider is introduced, the common contract ensures that your existing investments remain applicable as new services are introduced.

Beside the common properties that are shared across all services, a resource also has properties specific to each resource type. For example, a virtual network contains a concept of subnets, that is not relevant for storage accounts. Inversely, a storage account contains different redundancy options that are not relevant for a virtual network. To allow resources to contain their own properties and enable the introduction of new features to existing resources, each resource has its own API version. API versions ensure that your current configuration remains valid as new updates to a resource type are introduced.

The combination of the common contract and resource specific properties provides a solid foundation for future developments, while ensuring your current investments are applicable to new features and services for the public, hosted and private cloud. Azure Resource Manager is a unified application model that provides consistent end user experiences while interacting with the resource providers on the user behalf.

Resource Groups

An application usually consists of multiple resources that are related to each other. Related application resources conceptually share a common lifecycle. Maintaining an application for its complete lifetime requires lifecycle management capabilities. In Azure Resource Manager, these capabilities are provided with Resource Groups. A resource group is an end-user facing concept that groups resources into a logical unit. A resource must always be part of a resource group and can only be part of a single resource group. By grouping resources into a resource group, an entire application can be managed as a single entity instead of a range of scattered individual resources.

For example, a virtual machine consists of a virtual hard disk that is stored in a storage account, a network interface card connected to a virtual network and an allocation of compute (CPU & RAM) resources. These components of a virtual machine are distinct resources in Azure. A resource group allows you to manage these related resources as a single logical entity with a shared common lifecycle.

A resource group can exist with no resources, a single resource or many resources. These resources can be sourced from one or multiple resource providers, spanning both IaaS and PaaS services.



A resource group cannot contain another resource group. All other Azure Resource Manager features integrate tightly with the resource group concept.

Role based access control

To create resources in the Microsoft cloud platform, an Azure subscription is required. A subscription is a top level container for all resources and operations of those resources within that subscription. These actions require an account associated with Azure Active Directory. It is possible to manage multiple subscriptions with the same account. Before designing your application on the Microsoft cloud platform, it is important to understand that a subscription also has quotas and limitations¹.

Role-based access control (RBAC) provides granular access management for Azure resources. For example, you can create a virtual machine that can only be deleted by the administrator of that machine. Role based access control is configured by assigning a role definition to a scope. The scope can be a subscription, a resource group or an individual resource. The assignment is inherited from the parent container. If you assign a role definition to a resource group, it applies to all resources in that resource group.

<https://azure.microsoft.com/en-us/documentation/articles/role-based-access-control-configure/>

Custom Policies

Many organizations have additional requirements when it comes to controlling possibilities during deployment. For example, you may wish to only allow deployment in certain regions, you may enforce a naming convention or you might be required to deny deployment of particular resource types. Custom policies are complementary to RBAC and enable the enforcement of policies at deployment runtime. Combined, they allow organizations to utilize the platform in advanced ways while still complying with their existing policy requirements and conventions.

<https://azure.microsoft.com/en-us/documentation/articles/resource-manager-policy/>

Tagging

Applications can consist of common resources that benefit from shared metadata. A tag is a key value pair, that can be assigned to resource groups or to resources. The key/value taxonomy is shared within a subscription, allowing you to use a tag from an existing resource and assign that to another resource within the same subscription. Using tags to add metadata to resources allows you to add additional

¹ <https://azure.microsoft.com/en-us/documentation/articles/azure-subscription-service-limits/>

information about your resources. A single tag provides an overview of all resources spanning the resource groups in a subscription, that have the tag applied. A good use case for tags is adding metadata to a resource for billing detail purposes.

<https://azure.microsoft.com/en-us/documentation/articles/resource-group-using-tags/>

Template Orchestration

Azure Resource Manager provides powerful Infrastructure as Code capabilities. Any type of resource that is available on the Microsoft cloud platform can be deployed and configured with Azure Resource Manager. With a single template, you can deploy and configure multiple resources as a single deployment operation. Templates allow you to deploy your complete application to an ideal operational end state.

Auditing

Any write operation to a resource is logged. The audit log can be used for troubleshooting purposes or user activity monitoring.

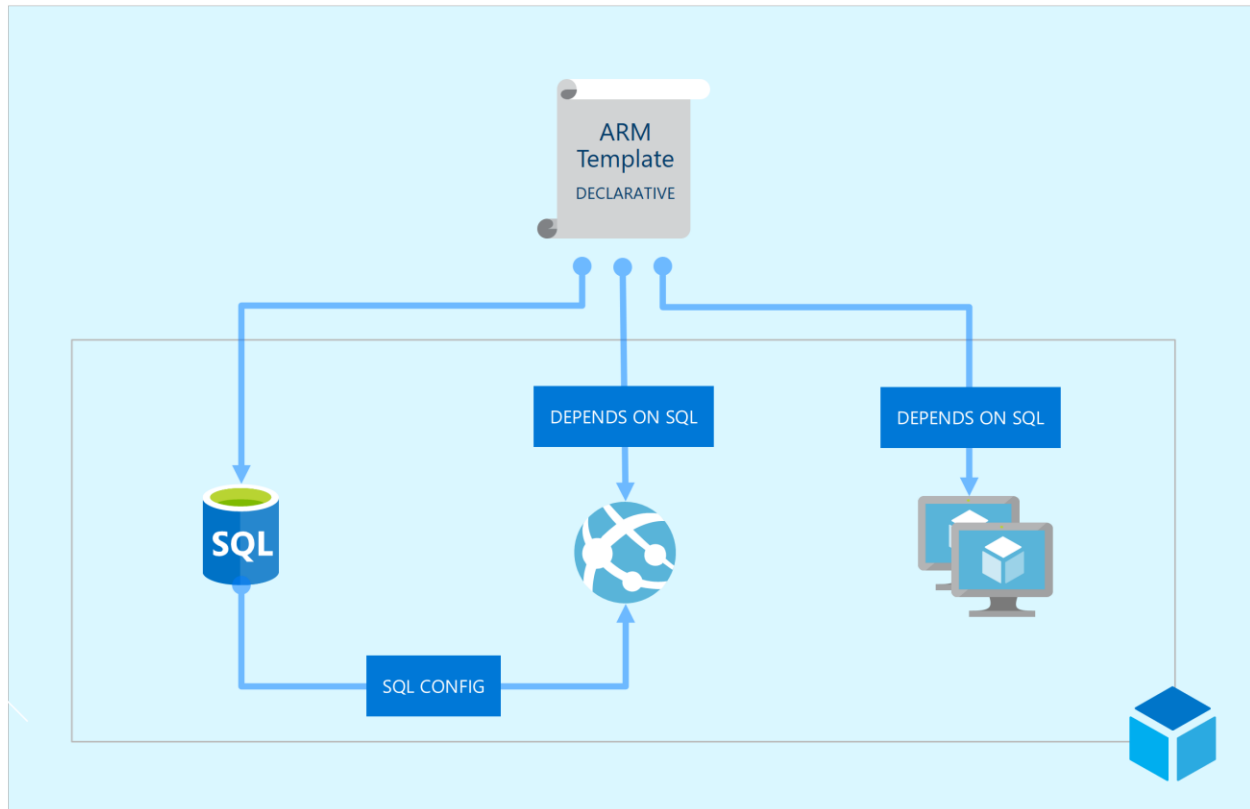
<https://azure.microsoft.com/en-us/documentation/articles/resource-group-audit/>

Consistency

The consistency of the public cloud, hosted cloud and cloud in your datacenter not only allows you to benefit from all the features within Azure Resource Manager, but also enables you to use your investments into the Infrastructure as Code capabilities of Azure Resource Manager across these clouds.

Template authoring

An application consists of different building blocks. A virtual machine is not a single resource but contains different individual resources like a virtual hard disk and a virtual network interface card. These resources can depend on other resources or can have other resources depending on it. This decomposed model allows an application to be constructed completely to your needs. Before you start creating your template it is a good idea to create a graphical design of your application first. You can think of the graphical design like an index of a book. The graphical design helps you to understand the required resources and the dependencies of between these resources.



A SQL DB instance is created for a Web App and Virtual Machines that depend on it. The connection details (SQL CONFIG) configures the WebApp to connect to the SQL database. Beyond the resources shown in the diagram, each virtual machine requires a location to store its disks and contains a virtual network interface card that requires a network for connectivity. An application within the virtual machine is configured with the use of a VM extension resource, which will be explained in more detail in the adding additional resources section in this whitepaper.

You can create a script that calls the Azure Resource Manager REST API to create each resource as an independent action, but you also create a template that describes the final state of your application and the template orchestration will perform the steps to achieve the desired state.

Language

Azure Resource Manager accepts JavaScript Object Notation (JSON) templates that comply with a JSON schema. JSON is an industry standard, human readable language.

<http://www.json.org/>

You can follow along in this whitepaper as we build a simple template as the basis for this whitepaper. For our example create a new template file called `azuredeploy.json`. Copy and paste the following code, that contains all the top level elements, in to the file.

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

An Azure Resource Manager template uses 6 top level elements. Each element has a distinct role in the template.

| ELEMENT | REQUIRED | DESCRIPTION |
|-----------------------------|----------|--|
| <code>\$schema</code> | ✓ | Location of the JSON schema file that describes the version of the template language. You should use the standard URL: <code>https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#</code> |
| <code>contentVersion</code> | ✓ | 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. |
| <code>parameters</code> | | Values that are provided when deployment is executed to customize resource deployment. |
| <code>variables</code> | | Values that are used as JSON fragments in the template to simplify template language expressions. |
| <code>resources</code> | ✓ | Resource types that are deployed or updated in a resource group. |
| <code>outputs</code> | | Values that are returned after deployment. |

Conventionally, the first single word in an attribute name is specified in lowercase while additional words are appended to the first word without spaces and start with an Uppercase (e.g. `contentVersion`). This convention is commonly referred to as “Camel case”.

Tooling

Although it is possible to create a template for a complete application in simple text editor, there is sophisticated tooling available that provides you with a better authoring experience for Azure Resource Manager templates.

Visual Studio

Visual Studio is an integrated development environment (IDE) for creating applications. Combined with the Azure Software Development Kit (SDK), Visual Studio provides an advanced Azure Resource Manager template authoring experience. Intellisense enables autocomplete for ARM templates and resource specific properties, based on the referenced JSON schema. You can deploy and debug your application directly in Visual Studio.

<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-resource-groups-deployment-projects-create-deploy/>

VS Code

Visual Studio Code (VS Code) is a free code editing tool. The tool is open source and runs on Linux, OS X, and Windows. With the Azure Resource Manager Tools extension installed, this code editor also provides Intellisense for ARM templates.

<https://azure.microsoft.com/en-us/documentation/articles/resource-manager-vs-code/>

Versioning control

It is important to track changes to a template over time or review mistakes made in past versions. Version control provides a solution to this common problem by versioning your template over time. By creating multiple versions of the template during your authoring process, you are able to look at an older version for reference or even revert to an older version.

There are multiple version control systems available that enhance the experience even more, by allowing different authors to contribute to the same template source in an orchestrated workflow.

As an example, Git is a distributed version control system that is widely used for software development. By using a distributed setup, every contributor operates on a complete local copy of the source code making their local efforts fast and reliable. Changes be merged into the master repository after evaluation by the owner. Tools like Visual Studio and VS Code are already fully integrated with Git.

Adding resources to your template

For the example we will use in this whitepaper, a storage account is the first resource that will be added to the template. The code blocks in this whitepaper will only contains the changes to our code and not the complete template. This will help to emphasize the differences in each step.

```
"resources": [  
  {  
    "name": "myStorageAccount",  
    "type": "Microsoft.Storage/storageAccounts",  
    "apiVersion": "2015-06-15",  
    "location": "East US",  
    "properties": {  
      "accountType": "Standard_LRS"  
    }  
  }  
],
```

Within the resources array we added a new object with an open and closing curly bracket. Within that object the common attributes (`name`, `type`, `apiVersion`, `location` and `properties`) are specified. The

value of the `properties` attribute is enclosed in curly brackets, indicating that the value allows for multiple different child attributes. This template deploys a storage account configured with the type set to the Local Redundant Storage option in the East US region.

All resource types require some common attributes like "name" and "type". Azure Resource Manager uses these attributes to ensure that the correct resource provider is handling the request. The following common attributes are required across all resources.

- **name** is used to name the deployed resource. Each resource of the same resource type within a single resource group must be uniquely named. Depending on the resource type it may also require uniqueness at either subscription, tenant or global scopes.
- **type** contains the resource provider and the resource type within that resource provider.
- **apiVersion** is used to identify the API version for the resource type. Multiple API versions can exist for a single resource type. These API versions are used to identify the available properties of a resource type.
- **location** sets the region for the resource to be deployed into. This can be a region in Microsoft Azure or a region in Microsoft Azure Stack, hosted by a service provider or running in your datacenter

Besides these required common attributes, each different resource can have optional common attributes like `tags` or `comments` and will have `properties` that are resource specific. For example, a storage account requires a replication policy while a virtual network requires a subnet. Resource specific properties are configured in the properties attribute.

- **properties** contain resource specific information

[Adding parameters to your template](#)

Consistency of Azure Resource Manager across clouds allows the template to be used for different environments. A scaled down version of your application can be sufficient for a test environment, while a production environment requires a more robust version. Depending on the desired end state, specific options might require other settings. For the example in this whitepaper, you may want to use a different replication mechanism for the storage account when the template is deployed to a different environment. This can be achieved by specifying parameters. The values for these parameters are requested from the tenant when a template is deployed. The values for the parameters can be passed to the template deployment in different ways.


```

"parameters": {
  "storageAccountType": {
    "type": "string",
    "defaultValue": "Standard_LRS",
    "allowedValues": [
      "Standard_LRS",
      "Standard_GRS",
      "Standard_RAGRS"
    ]
  }
},

```

A parameter requires a type, but can optionally contain a default value and allowed values. The type attribute specifies the expected type of input (string, int, bool, array, object, secureString). The type can also be used to render a user interface field when deploying the template from the tenant portal. The default value is used if no value is specified by the tenant at deployment time. If a parameter does not contain a default value, the tenant is required to submit a value when deploying the template.

While the parameter has been specified, the parameter is not used in the resource yet. The hard-coded value of the storage account type needs to be replaced with the parameter. The value must refer to the correct parameter in the format `parameters('parameterName')` enclosed by two square brackets. The square brackets allow Azure Resource Manager to identify the content as a function to be evaluated instead of a static string.

```

"resources": [
  {
    "name": "storageAccount",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2015-06-15",
    "location": "East US",
    "properties": {
      "accountType": "[parameters('storageAccountType')]"
    }
  }
],

```

By adding a parameter to your template and replacing the fixed value in a resource with that parameter, the template is now more dynamic. While the concept of replacing fixed values by parameters is applicable throughout the template, just be aware of the effect that a larger number of parameters can have on the deployment experience. If a tenant is asked to submit many values for deploy a template, the experience will not be very user friendly. The deployment experience can be improved by specifying a default value where possible and minimize the number of parameters overall. Combining parameters with variables can also improve the experience while retaining the dynamic nature of a template.

[Adding variables to your template](#)

Where parameters are used to request a value from the tenant at deployment time, a variable is a value that can be reused throughout the template without requiring user input. For example, a storage account resource can be used by a virtual machine resource to store its virtual disks. The virtual machine resource will need to reference the storage account resource name in its configuration.

You could specify the name for the storage account as a static value for both the storage account resource and the virtual machine resource. This is problematic as the name can be potentially changed in one location and not the other. Instead, it is much easier to create a single variable that is referenced by all the relevant resources.

```
"variables": {
  "storageAccountName": "myStorageAccountName"
},
```

The variable can be referenced using the function: `variables('variableName')`. The function, just like the parameter function, is enclosed by two square brackets indicating that it should be evaluated by Azure Resource Manager accordingly.

```
"resources": [
  {
    "name": "[variables('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2015-06-15",
    "location": "East US",
    "properties": {
      "accountType": "[parameters('storageAccountType')]"
    }
  }
],
```

Template functions

Azure Resource Manager provides template functions that make the template orchestration engine very powerful. Template functions enable operations on values within the template at deployment time. A simple example of a template function is to concatenate two strings into a single string. You could use a function that concatenate strings and pass in the two strings as parameters to the functions.

Template functions can be used in variables, resources and outputs in Azure Resource Manager templates. Template functions can reference parameters, other variables or even objects related to the deployment. For example, a template function can reference the ID of the resource group.

<https://azure.microsoft.com/en-us/documentation/articles/resource-group-template-functions/>

Common examples of template function usage

The example we used in this whitepaper currently contains a variable for the storage account name. This variable is set to the value `"myStorageAccountName"`. Besides other requirements, a storage account requires a globally unique fully-qualified domain name (FQDN). After you create the storage account, this domain is used to allow you to access your storage resources using a URL. In our example, the chance that the name `myStorageAccountName` is not used yet is very small. If we try to deploy a storage account with that name, the deployment will likely fail. The best practice for a storage account name is to use a variable that generates a unique name. To accomplish this, we can concatenate a unique string to the text `'storage'`. This can be achieved by using the `concat()`, `resourceGroup()` and `uniqueString()` template functions.

The storage account name is generated by concatenating a unique string that is derived from the id of the resource group that the template is being deployed to and the string 'storage'. Because the storage account resource already references the variable for the `storageAccountName`, the resource itself does not require a change to reflect the new value of the variable.

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

Although we have created a parameter for the `storageAccountType`, each time the template is deployed the storage account will always be created in "East US". That fixed value is specified in the `location` attribute of the resource. The obvious solution would be to define parameter for the location and ask the user for the region. To make this less error prone a list of allowed values would ensure a valid selection, but it also limits the regions you can deploy the resource to. As new regions are added to the platform, it can be challenging when deploying to Microsoft Azure and even more challenging for Microsoft Azure Stack, where you might specify your own regions. A template function solves this challenge. You can configure each resource to deploy to the same location as the target resource group using the template function `resourceGroup().location`, the template remains reusable across clouds and requires one less input from the tenant. Using this function each resources location ensures that all resources will be deployed in the same region as the resource group.

```
"resources": [  
  {  
    "name": "[variables('storageAccountName')]",  
    "type": "Microsoft.Storage/storageAccounts",  
    "apiVersion": "2015-06-15",  
    "location": "[resourceGroup().location]",  
    "properties": { "accountType": "[parameters('storageAccountType')]" }  
  }  
],
```

Some applications can require a distributed setup across regions. In these cases, create a single variable for the deviating location and reference that variable for the resources that require placement in region other than the resource group.

[Adding outputs to your template](#)

A template can also contain outputs. The outputs and their values are displayed and recorded when the deployment is finished. Outputs can be useful to display properties of a deployed resource (e.g. the FQDN of a web server). Each output requires a type (`string`, `int`, `bool`, `array`, `object`, `secureString`) and can contain template functions, parameters and variables.

```
"outputs": {
  "storageAccountEndpoint": {
    "type": "string",
    "value": "[variables('storageAccountName')]"
  }
}
```

You might be accustomed to the option to get the content of a variable. It is possible to create a template without any resources, but with outputs to validate the content of more complex template functions in the template. This template will deploy very fast, since there are no resources deployed, but still allow you to verify the output of the template function.

Template deployment

The template we just created only contains a single storage account resource. Before adding new resources to the template, it is a good idea to commit changes to the template in our version control system. This allows us to compare future edits to this version of the template and also enables the option to revert this version, undoing possible unwanted changes.

After committing changes in the template to version control, we are ready to deploy the template. The template only contains a storage account at this point.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountType": {
      "type": "string",
      "defaultValue": "Standard_LRS",
      "allowedValues": [
        "Standard_LRS",
        "Standard_GRS",
        "Standard_RAGRS"
      ]
    }
  },
  "variables": {
    "storageAccountName": "[concat(uniquestring(resourceGroup().id), 'storage')]"
  },
  "resources": [
    {
      "name": "[variables('storageAccountName')]",
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2015-06-15",
      "location": "[resourceGroup().location]",
      "properties": {
        "accountType": "[parameters('storageAccountType')]"
      }
    }
  ],
  "outputs": {
    "storageAccountEndpoint": {
      "type": "string",
      "value": "[variables('storageAccountName')]"
    }
  }
}
```

The tooling section in this chapter explains the different deployment methods.

You can stop and test your template as you complete parts of it. This enables you to validate early and iteratively test additions to the template. We still have more resources to add and we can do this at any time even after the template has been run. The same template can be reused targeting the same resource group. Azure Resource Manager is idempotent. Based on the referenced resource type, each resource in the template will be validated by the relevant resource provider at deployment runtime. If the

resource does not exist, it is created. If the resource already exists, the resource provider verifies if any changes to the resource present in the template and updates the resource accordingly.

The resource providers perform exact matching. For example, if your template contains a storage account with the exact same name as an existing storage account in that resource group, Azure Resource Manager will skip the creation of that resource. If your template contains a storage account with a different name as an existing storage account in that resource group, Azure Resource Manager will create a new storage account, resulting in two storage accounts. By tracking changes to your template with version control as you deploy your template to the same resource group during the authoring process it is easy to identify issues.

All resources of a deployment will be created in the same resource group, even if you nest templates for more advanced scenarios. We will cover nested templates later in this whitepaper.

Parameter file

The example template in this whitepaper currently contains one parameter for the storage account type. A template for a complete application can contain various different parameters. If you are frequently deploying your template during the authoring process, the values for the parameters need to be resubmitted each time. A parameter file can automate this process. A parameter file contains values for the parameters in the template, and can be parsed when the template is deployed. Instead of manually typing in the parameters, the values are pulled from the parameter file. You can create multiple parameter files, representing different stages of an environment (Development, Testing, Acceptance and Production – or, DTAP), different regions or different clouds.

You can create a parameter file with the name `azuredploy.parameters.json`. For the storage template, an example parameter file could contain the following code.

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

Notice that the file has a different schema? This new schema has a `contentVersion` attribute like the previous schema but it also has a `parameter` attribute where you specify parameter names and values for your template deployment.

This parameter file needs to be updated over time in accordance with the changes to the template. At a minimum, each parameter in the template that does not contain a default value must be present in the `parameters` attribute of the parameter file. Each parameter in the template that contains a default value can optionally be overwritten by a value specified in the parameter file, but it is not required.

Tooling

There are multiple ways to deploy a template. You can choose the approach that fits your needs best.

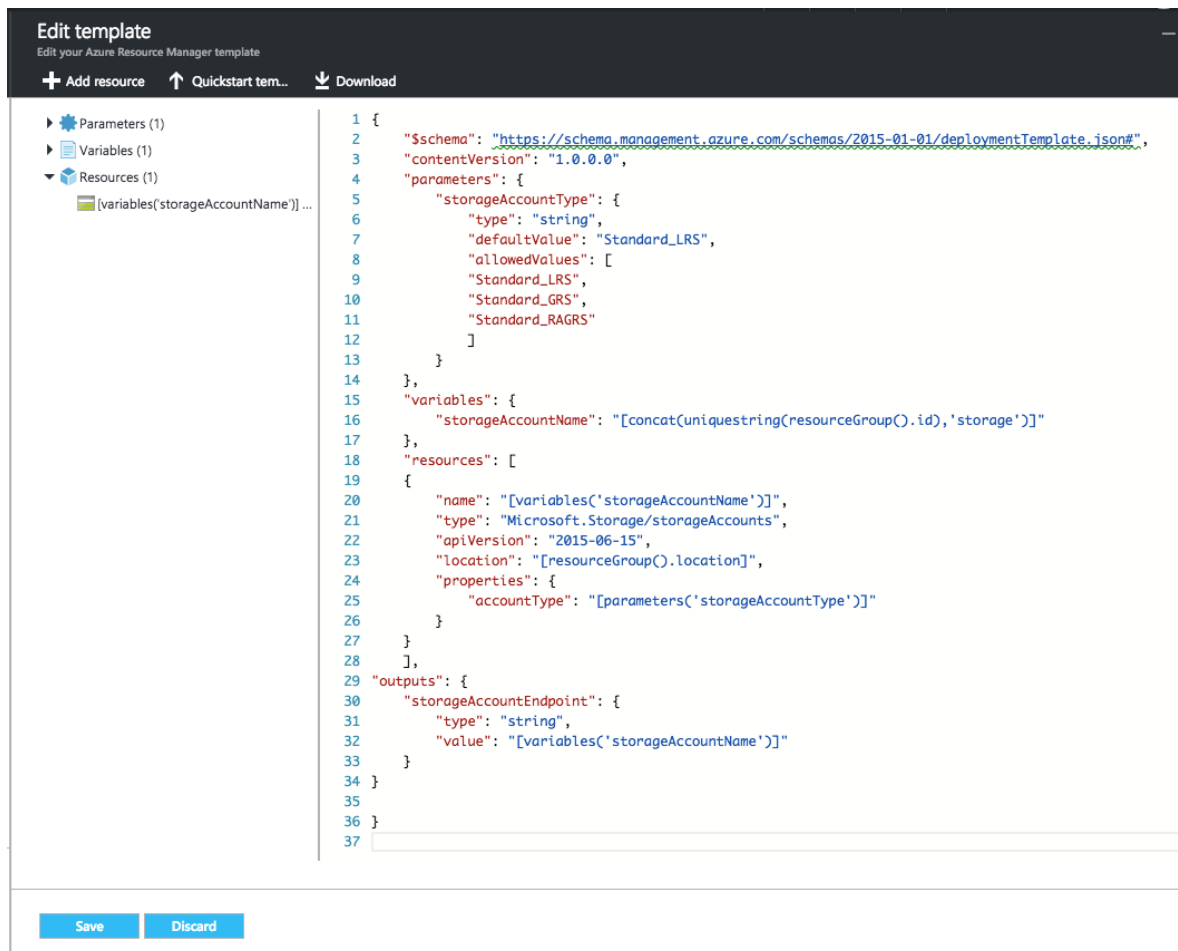
Portal

The Microsoft cloud platform allows you to build, manage, and monitor everything from simple web apps to complex cloud applications in a single, unified web based portal.

The Azure portal deploys most resources using the Azure Resource Manager and various templates. When using the portal, you may notice that as you create resources imperatively, the URL of the dynamic web interface updates to indicate which resources you are creating using ARM. If you want to create a Web App using the portal and click the appropriate menu options, you will notice that the URL in the portal changes from <https://portal.azure.com/> to <https://portal.azure.com/#create/Microsoft.WebSite>. You can alternatively paste this URL into any browser and you will be taken directly to the appropriate interface to create a new Web App.

Besides the enormous gallery of preconfigured applications, you can deploy from the marketplace, you can also deploy a custom ARM template directly from the portal. The marketplace contains a Template deployment item, that can consume an ARM template, render its parameters for input and start deployment. When you select this option, the portal will update to the URL: <https://portal.azure.com/#create/Microsoft.Template>.

The template deployment option in the portal contains an authoring window where you can directly edit your JSON template. You can also copy your template and paste it directly into the authoring window.



```
1 {
2   "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "storageAccountType": {
6       "type": "string",
7       "defaultValue": "Standard_LRS",
8       "allowedValues": [
9         "Standard_LRS",
10        "Standard_GRS",
11        "Standard_RAGRS"
12      ]
13    }
14  },
15  "variables": {
16    "storageAccountName": "[concat(uniquestring(resourceGroup().id), 'storage')]"
17  },
18  "resources": [
19    {
20      "name": "[variables('storageAccountName')]",
21      "type": "Microsoft.Storage/storageAccounts",
22      "apiVersion": "2015-06-15",
23      "location": "[resourceGroup().location]",
24      "properties": {
25        "accountType": "[parameters('storageAccountType')]"
26      }
27    }
28  ],
29  "outputs": {
30    "storageAccountEndpoint": {
31      "type": "string",
32      "value": "[variables('storageAccountName')]"
33    }
34  }
35 }
36 }
37 }
```

Alternatively, you can create a hyperlink to open the deployment option in the portal and use an existing template that is available via a publicly accessible URL. For this option, you commonly see template authors use a URL to the latest version of the ARM template hosted in a version control system).

You can create a URL with a hyperlink to deploy your template.

```
https://portal.azure.com/#create/Microsoft.Template/uri/<replace with the encoded raw path of your azuredeploy.json>
```

You can also create a HTML button with a hyperlink to deploy your template with the following HTML code. This button can also use an image hosted at <http://azuredeploy.net> that reads "Deploy to Azure".

```
<a href="https://portal.azure.com/#create/Microsoft.Template/uri/<replace with the encoded raw path of your azuredeploy.json"> target="_blank">  
    
</a>
```

Please note that you will need to still specify the parameters for each deployment. It is not possible to parse a parameter file in the template deployment item.

Cross Platform CLI

The cross platform command line interface provides (xPlat CLI) open sourced shell based command to operate resources in Microsoft Azure different operating systems. After installing the cross platform cli you can connect to your subscription from a Mac, Linux or Windows machine using console interfaces like bash, terminal and command prompt. After you have logged on to your subscription with `azure login`, you need to switch to ARM context with `azure config mode arm`. In the ARM context a template can be deployed with the following command.

```
azure group deployment create -f azuredeploy.json -e azuredeploy.parameters.json testRG  
testRGDeploy
```

<https://azure.microsoft.com/en-us/documentation/articles/xplat-cli-install/>

PowerShell

It is possible to use PowerShell to deploy an ARM template using the declarative programming model. This allows us to deploy the template from the example used in this whitepaper. You can specify the parameters for the template manually or reference a parameter file.

```
New-AzureRmResourceGroupDeployment -Name myDeployment -ResourceGroupName  
"myResourceGroup" -TemplateFile c:\myApplication\azuredeploy.json -  
TemplateParameterFile c:\myApplication\azuredeploy.parameters.json
```

The command runs for the duration of the deployment giving status updates of the current action in the output.

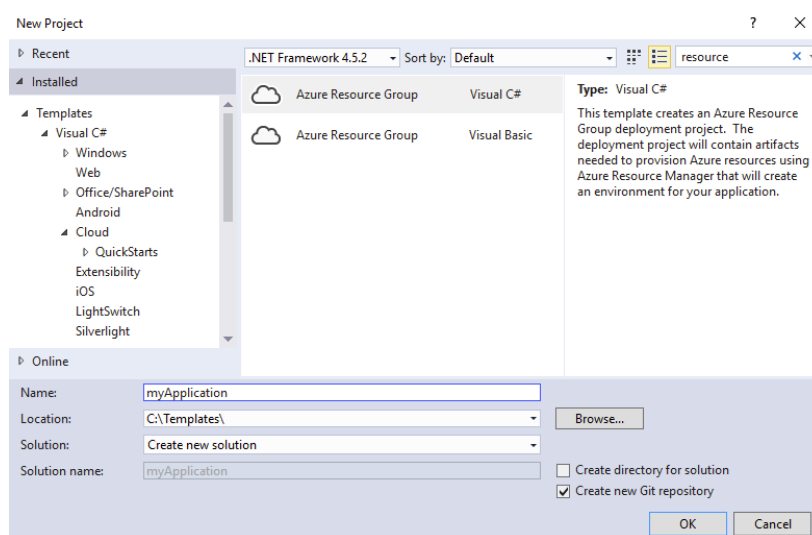
With PowerShell you can also create resources using the imperative programming model by running the relevant PowerShell cmdlet. For example, a storage account is created with the following snippet after signing in to your subscription.

```
New-AzureRmStorageAccount -ResourceGroupName "myResourceGroup" -Name "mystorageaccount" -Location "West US" -Type Standard_LRS
```

This snippet creates a new storage account of type `Standard_LRS` named `mystorageaccount` in a resource group named `myResourceGroup` located in the `West US` Azure region.

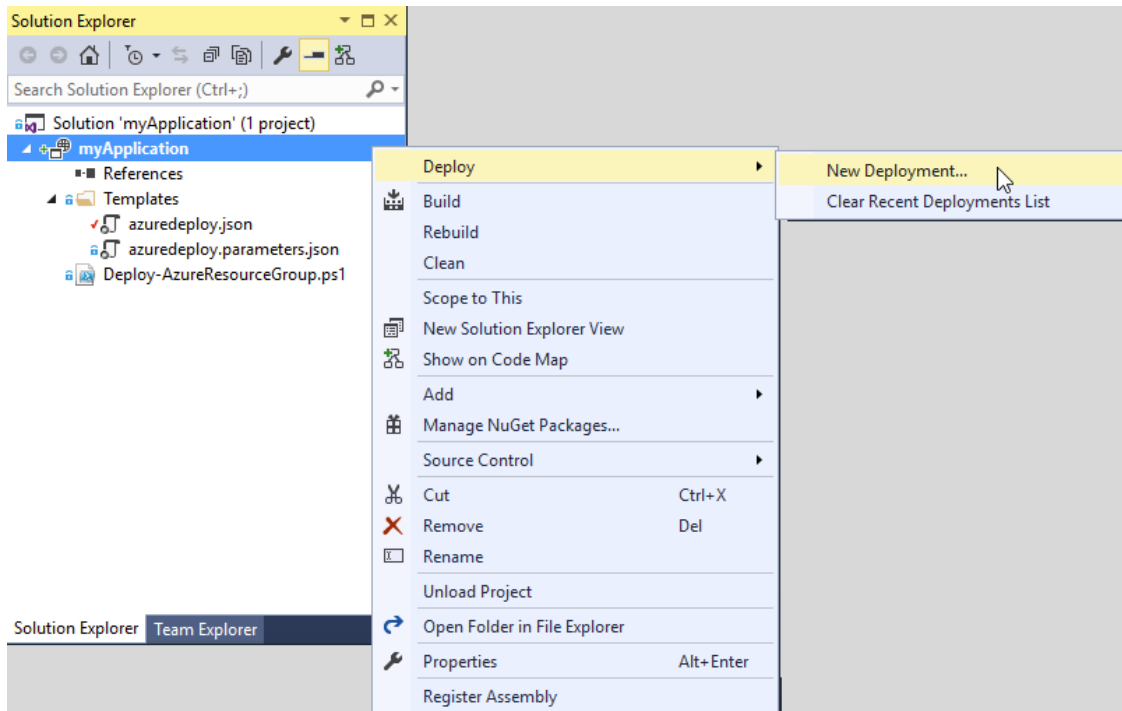
Visual Studio

Combined with the Azure software development kit (SDK), Visual Studio provides an integrated solution for creating, deploying and troubleshooting your templates. Visual Studio can create an Azure Resource Group project, containing all the artifacts for your deployment.



It contains a JSON outline to add resources to your template or create a new template from a quick start item, preconfigured with content. Intellisense in Visual Studio enhances the authoring experience and allows you to quickly identify any issues in your template.

A template can be deployed directly from Visual Studio without ever logging on to the portal. Visual Studio uses PowerShell to deploy the template and the output is directly visible in the same interface as your template. By integrating authoring, deployment, output and optionally even version control, Visual Studio provides a unified solution for creating your applications.



Deploy from a private storage account

The root of the **azure-quickstart-templates** repository on GitHub contains a script, that will copy an ARM template and all artifacts from a single folder (including scripts) and pre-stage it in a private storage account. The deployment will then be started automatically from the storage account using the storage endpoint URL and the SAS token as an input parameter.

If your template contains application specific intellectual property a private storage account in your subscription can protect the IP. Future deployments can be performed from the same storage account with the same script.

<https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/README.md#deploying-samples>

Troubleshooting

During the authoring process for an application deployment, the author will most likely make various changes to get to the desired end result. A change can impact the deployment and potentially even result in a failure during deployment. Versioning control helps you to revert to a previous stable version of your template. The same tools used for the deployment allow you to discover the reason for a failed deployment. You can then use this reason to determine the required changes to the template necessary for a successful deployment.

You can use these tools interchangeably since they all communicate with Azure Resource Manager. For example, you can deploy a template from the cross platform CLI and retrieve the deployment logs with PowerShell. You could also use the portal to look at the deployment logs from a deployment performed from Visual Studio. Each deployment to a resource group is logged with its own unique entry within that

resource group. Each deployment entry in a resource group consists of one or multiple deployment operations.

Query returned 4 items. [Click here to download all the items as csv.](#)

| OPERATION NAME | STATUS | TIME |
|---|-----------|-----------|
| ▶  Write Deployments | Succeeded | 4 min ago |
|  Validate | Succeeded | 4 min ago |
|  Update resource group | Succeeded | 4 min ago |
|  Update resource group | Succeeded | 4 min ago |


PowerShell

The Azure PowerShell module contains cmdlets for retrieving deployment logs from resource groups. The `Get-AzureRmResourceGroupDeployment` cmdlet retrieves the deployment entries from a resource group and the `Get-AzureRmResourceGroupDeploymentOperation` displays the details of an individual deployment operation. The following script retrieves the operations from the latest deployment in a sample resource group named `myResourceGroup`.

```
$RG = Get-AzureRmResourceGroup -Name "myResourceGroup"

$RGD = (Get-AzureRmResourceGroupDeployment -ResourceGroupName
    $RG.ResourceGroupName)[0].DeploymentName

((Get-AzureRmResourceGroupDeploymentOperation -ResourceGroupName $RG.ResourceGroupName -
    DeploymentName $RGD).Properties).StatusMessage.error | FL
```


 <https://azure.microsoft.com/en-us/documentation/articles/resource-manager-troubleshoot-deployments-powershell/>

Visual Studio

The output window in Visual Studio display the actual status of a deployment made using Visual Studio. When the deployment is successful, the outputs from the ARM template are displayed with their values. If the deployment fails during execution, the output window will display the error messages. The output window in Visual Studio only displays the log of the current deployment. If you want to retrieve logs from previous deployments, you can use one of the other tools.

Portal

From the portal the deployment entries and operations can be retrieved through a graphical UI. The deployment tab in the settings blade of a resource group shows a list of all recent deployments to that resource group (both completed and active). Selecting a deployment from the list displays the individual operations. Each operation will open its details in when selected.

 <https://azure.microsoft.com/en-us/documentation/articles/resource-manager-troubleshoot-deployments-portal/>

Cross Platform CLI

Azure supports a command line tool that allows you to manage resources in a consistent manner across many different operating systems. This tool is commonly referred to as the Cross-Platform CLI or xPlat CLI and provides an unprecedented consistency across tools and platforms.

You can retrieve the deployments to a resource group named `myResourceGroup` from a Mac, Linux or Windows machine with the following command. You can replace `myResourceGroup` in this example with your resource group name.

```
azure group deployment show --resource-group myResourceGroup
```

To retrieve the details of a deployment operation, the following command can be used. For your own resource groups, replace `myResourceGroup` in this example with your resource group name and `myDeployment` with the relevant deployment from the previous command.

```
azure group deployment operation list --resource-group myResourceGroup --name  
myDeployment --json
```

<https://azure.microsoft.com/en-us/documentation/articles/resource-manager-troubleshoot-deployments-cli/>

Template reusability across clouds

Each Azure Resource Manager template needs to comply with a JSON schema. A JSON schema details a structure for JSON data. In the context of Azure Resource Manager, the schema is a contract that details the valid structure for an ARM template which is written in JSON. Each individual resource provider has their own schema referenced from the master schema. These individual schemas detail properties that are unique to each resource provider.

Within the JSON schema you still have some flexibility. You can decide to use a parameter or a variable for a reusable value throughout the template. You can specify a fixed value or create a variable for everything. However, to ensure reusability of a template across clouds you need to avoid region specific dependencies in a template. Region specific dependencies are relevant when dealing with the location of a resource, endpoints and API versions.

Endpoints

Some services use a public endpoint that can be referenced in an ARM template. For example, when you add a virtual machine resource to a template, the disks reference a URI of a blob hosted in a specific storage account. It is possible to concatenate the storage account name with the public endpoint (e.g. blob.core.windows.net) as shown below.

```
"diskUri": "[concat('http://',variables('storageAccountName'),'.blob.core.windows.net/',  
variables('vmStorageAccountContainerName'),'/',variables('OSDiskName'),'.vhd')]"
```

This technique creates a dependency on that public endpoint. Deploying the same template to Microsoft Azure Stack would fail, because that uses a different endpoint FQDN for storage. Instead of hardcoding the endpoint, use a reference function to get the public endpoint FQDN dynamically at runtime. This method allows you to reuse the same template across clouds.

```
"diskUri": "[concat(reference(concat('Microsoft.Storage/storageAccounts/',  
variables('storageAccountName')), '2015-06-15').primaryEndpoints.blob,  
variables('vmStorageAccountContainerName'),'/',variables('OSDiskName'),'.vhd')]"
```

Resource location

A resource group is always created in a location. The resources in a resource group can be created in the same location as the resource group or in a different location, as long as Azure Resource Manager is the control plane for that location. Microsoft Azure is composed of many regions today and new regions are constantly added to this list. In the context of Microsoft Azure Stack, locations will refer specifically to your datacenters.

Because of the relationship between resources, resource groups and regions; you can easily create a dependency on a specific region within a template. A template function can help prevent a dependency on a region in a template. By configuring the resource to use the same location as your resource group, you can now ensure that the template can be deployed to many different regions.

```
"resources": [
  {
    "name": "[variables('storageAccountName')]",
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2015-06-15",
    "location": "[resourceGroup().location]",
    "properties": { "accountType": "[parameters('storageAccountType')]" }
  }
],
```

API versions

Each resource type within a resource provider support a range of API versions. An API version details the available properties of the resource type at a specific point in time. This allows resources to be updated independently of other resources. This also allows you to write templates today that will still function correctly as new versions of the resource providers are released.

It is important to strongly version your resources as different API versions for the same resource type can have different properties. Strong versioning means that the API version is always statically specified instead of retrieving the latest API version dynamically at runtime. In the cloud-first, mobile-first world; new API versions will always be introduced in Microsoft Azure. When Microsoft Azure Stack brings an API version for a resource from Microsoft Azure to your datacenter, the properties of the resource for that API will be consistent.

If a template references a new resource type API version that is not (yet) available in Microsoft Azure Stack the template deployment will fail on Microsoft Azure Stack. This can happen if a template is referencing a version of a resource provider that is available on the public cloud (Microsoft Azure) but not yet on your Microsoft Azure Stack implementation.

If you want to ensure the reusability of your template across clouds, you should use an API version that is present across both public and private clouds. To verify what API versions are available for a resource type, you can use the following PowerShell command after connecting to your subscription from the PowerShell session.

```
Get-AzureRmResourceProvider | select-object ProviderNamespace -ExpandProperty
ResourceTypes | ft ProviderNamespace, ResourceType, ApiVersions
```

These details can also be retrieved from the cross platform CLI. After authenticating from the command line, the following command exports the full details of all resource providers to a JSON file.

```
azure provider list -vv --json > c:\temp.json
```

Resource provider availability

New resource providers are initially introduced to the platform as a preview for evaluation purposes. These providers are only available in specific regions during their preview. Eventually, more regions will be added as the resource provider moves to general availability. When designing templates, it is important to verify that the region you are using for a new resource supports your desired resource type. By running the following script, you can verify what regions support a specific resource type:

```
Get-AzureRmResourceProvider | select-object ProviderNamespace -ExpandProperty  
ResourceTypes | ft ProviderNamespace, ResourceType, Locations
```

Microsoft Azure Stack contains a subset of the resource providers available in Microsoft Azure. A Microsoft Azure Stack operator may decide to only implement some of the available resource providers in their datacenters. A template deployment will fail if the template contains resources from a resource provider that is not available in the desired datacenter (region). The following script retrieves the resource providers available in a specific cloud:

```
Get-AzureRmResourceProvider -ListAvailable
```

Adding additional resources to the template

The first version of our ARM template successfully deploys a storage account resource. Ideally, the changes to template would be committed to version control at this point. In a typical resource design workflow, we are ready to add additional resources to the template. These additional resources can be added by typing the JSON code directly into the template or use a tool like Visual Studio to add resources through a wizard. Visual Studio also includes a JSON Outline pane that allows you to add resources to your template through a graphical user interface that presents the most common resource types. If you add a resource using this tool, it adds the required code for the resource, its parameters and variables to the existing template.

Dependencies

Real applications are often connected in various ways that creates dependencies. The creation of a resource in Azure can also depend on the existence of another resource. Creating a virtual machine, requires a storage account and a virtual network to be present. When all the dependent resources are deployed as part of the same template, the dependencies need to be defined. Without defining the dependencies, the platform cannot determine the correct order necessary to deploy the resources. To specify a dependency for a resource the following `dependsOn` attribute is used.

```
"resources": [
  {
    "name": "[variables('vmName')]",
    "type": "Microsoft.Compute/virtualMachines",
    "apiVersion": "2015-06-15",
    "location": "[resourceGroup().location]",
    "dependsOn": [
      "[concat('Microsoft.Storage/storageAccounts/',
        variables('storageAccountName'))]"
    ],
    "properties": {}
  },
],
```

In this example, a resource for a virtual machine (identified by the `type` field) is added to the template. The `name` attribute contains a variable that defines the name of the virtual machine. The `properties` are left empty to simplify this example. In a real deployment, they should minimally contain the required properties. The resource for the virtual machine consists of the same common attributes (`name`, `type`, `apiVersion` and `location`), with the exception that we have added a new `dependsOn` attribute. The `dependsOn` attribute is applied to ensure that the resources within the resource group are deployed in the correct order based on a dependency tree. In this example, the `dependsOn` attribute specifies that the storage account with the name `variable('storageAccountName')`, is deployed before the virtual machine resource is deployed.

<https://azure.microsoft.com/en-us/documentation/articles/resource-group-define-dependencies/>

Inside a resource

The activities performed by the template deployment involve the creation of new resources and modification of existing resources. Some resources require configuration inside their container.

A good example of a resource configuration is an application that makes use of virtual machines. A virtual machine resource is deployed by Azure Resource Manager, but the virtual machine typically requires an application that needs to be installed and configured. For this reason, VM extension resources allow you to perform configuration actions within the virtual machine from a multitude of imperative or declarative technologies. Extensions can perform actions such as deploy a Docker image, apply a Desired State Configuration (DSC) document, run a PowerShell script or configure Puppet agents.

<https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-windows-extensions-features/>

Extension resources are also available for other Azure resources. An Azure SQL Database instance can have an extension that installs an application-tier database using a .bacpac file. An Azure App Service Web App instance can have an extension that deploys a project from GitHub or installs a WebDeploy package. Because an extension is a first class resource in Azure Resource Manager, extensions can even have their own dependencies.

<https://azure.microsoft.com/en-us/documentation/articles/app-service-web-arm-from-github-provision/>

Azure Quickstart Templates

After reading this whitepaper you'll likely be curious how you can configure all kinds of resource types. You can use the Azure Quickstart Templates repository as a great source of information. Microsoft started this community initiative on GitHub as a way to create, maintain and contribute templates in a public manner.

<https://github.com/Azure/azure-quickstart-templates>

As of writing this whitepaper, the **azure-quickstart-templates** repository contains close to 400 different deployment templates to use with Azure Resource Manager. These templates range from simple examples to full multi-tier applications with a wide variety of infrastructure and platform resources.

Besides the variety of templates, the repository contains guidelines that help you create and maintain your own ARM templates. These guidelines can be applied whether you prefer to store your templates on a public repository such as GitHub or not. The guidelines are split into three documents.

| | | |
|---------------------------|---|---|
| CONTRIBUTION GUIDE | Describes the minimal guidelines for contributing. | https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/README.md |
| BEST PRACTICES | Best practices for improving the quality of your template design. | https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/best-practices.md |
| GIT TUTORIAL | Step by step to get you started with Git. | https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/git-tutorial.md |

Deployment templates are typically validated before they are merged into the public repo against guidelines specified in the contribution guide. This ensures that the templates are also automatically displayed on a gallery on the public Microsoft Azure website.

More advanced scenarios

Decomposing templates

It is obvious to create a single deployment template to deploy a single resource. A deployment template for more advanced scenarios can benefit from a design that decomposes your related resources. Azure Resource Manager allows you to link multiple templates, in a practice referred to as nesting templates. Consider the following guidance to help you to decide between a single template or a decomposed nested template design:

- Create a single template for a single tier application
- Create a nested template deployment for a multitier application
- Use nested templates for conditional deployment (based on a parameter selection a different nested template URL can be constructed)

Within a deployment template, create a resource of the type `Microsoft.Resources/deployments`. The `templateLink` property of this resource allows you to reference another deployment template.

```
"resources": [  
  {  
    "name": "myNestedTemplate",  
    "type": "Microsoft.Resources/deployments",  
    "apiVersion": "2015-01-01",  
    "properties": {  
      "mode": "Incremental",  
      "templateLink": {  
        "uri": " https://raw.githubusercontent.com/Azure/azure-quickstart-  
templates/master/wordpress-mysql-replication/nested/website.json ",  
        "contentVersion": "1.0.0.0"  
      }  
    }  
  }  
]
```

The template that is referenced in the `templateLink` attribute is deployed as part of the initial (*master*) deployment. All resources, from both the master and the linked template, will be deployed in the same resource group. You can define multiple resources of the type `Microsoft.Resources/deployments` in a single deployment template, each pointing to a different nested template. This can be useful if you are deploying a multi-tier application. Each tier can have its own template that is referenced from the master template. By setting the `dependsOn` attribute for the `Microsoft.Resources/deployments` resource type you can control the nested templates deployment order.

<https://azure.microsoft.com/en-us/documentation/articles/resource-group-linked-templates/>

Multiple Resource Groups

A deployment will always be performed against a single resource group, even when linked templates are used. It is not uncommon that a resource in one resource group has a dependency to a resource in another resource group. For example, two applications that consist of multiple virtual machines require connectivity to each other. The virtual network that enables this connectivity can only be part of one resource group. In this design the virtual network could be configured in its own resource group. The

virtual network adapters of the virtual machines in the other resource groups can reference the virtual network in the separate resource group.

```
"resources": [
  {
    "name": "vNic",
    "type": "Microsoft.Network/networkInterfaces",
    "apiVersion": "2015-06-15",
    "location": "[resourceGroup().location]",
    "properties": {
      "ipConfigurations": [
        {
          "name": "ipconfig1",
          "properties": {
            "privateIPAllocationMethod": "Dynamic",
            "subnet": {
              "id": "[concat(resourceId(parameters('existingVirtualNetworkResourceGroup'),
                'Microsoft.Network/virtualNetworks',
                parameters('existingVirtualNetworkName')), '/subnets/',
                parameters('existingSubnetName'))]"
            }
          }
        }
      ]
    }
  }
],
```

In this example the id of the virtual network adapter references a subnet, that is part of an existing virtual network in an existing resource group. The names of the existing values are retrieved from the tenant as parameters.

To better understand more advanced deployment scenarios, you can review a detailed whitepaper called World Class ARM Templates Considerations and Proven Practices.

<http://download.microsoft.com/download/8/E/1/8E1DBEFA-CECE-4DC9-A813-93520A5D7CFE/World%20Class%20ARM%20Templates%20-%20Considerations%20and%20Proven%20Practices.pdf>

Summary

Congratulations, you just made a valuable investment by reading this whitepaper explaining Azure Resource Manager and its Infrastructure as Code capabilities. While this whitepaper took you through a high level tour of Azure Resource Manager capabilities to give you some foundation and get you started, there are a number of relevant links to additional resources throughout this whitepaper that will help you as you dive in to the details. In addition, the Azure Quickstart Templates repository is your one-stop resource for numerous rich template examples you can draw from for your own scenarios.

Microsoft provides a unique platform that is consistent across clouds – and Azure Resource Manager is core to enabling this consistency. By further investing in a platform that allows you to reuse your investment for different environments you will achieve a greater return on investment.