Module 2: Deploying Resources with Azure Resource Manager

Contents:

Module overview

This module establishes a basic understanding of Azure Resource Manager and the core concepts of deployments, resources, templates, resource groups, and tags. The module will dive deeply into the automated deployment of resources using ARM templates.

Objectives

After completing this module, students will be able to:

- Create a resource group.
- Add resources to a resource group.
- Deploy an ARM template to a resource group.
- Filter resources using tags.
- Author a complex deployment using the Azure Building Blocks tools.

Lesson 1: ARM Templates

Azure has developed a great deal over the last few years, and with new services being released on a regular basis, there was a need to create a way to manage and deploy resources in a componentized and reusable manner. Azure Resource Manager was designed to represent each service in Azure as a resource provider and each

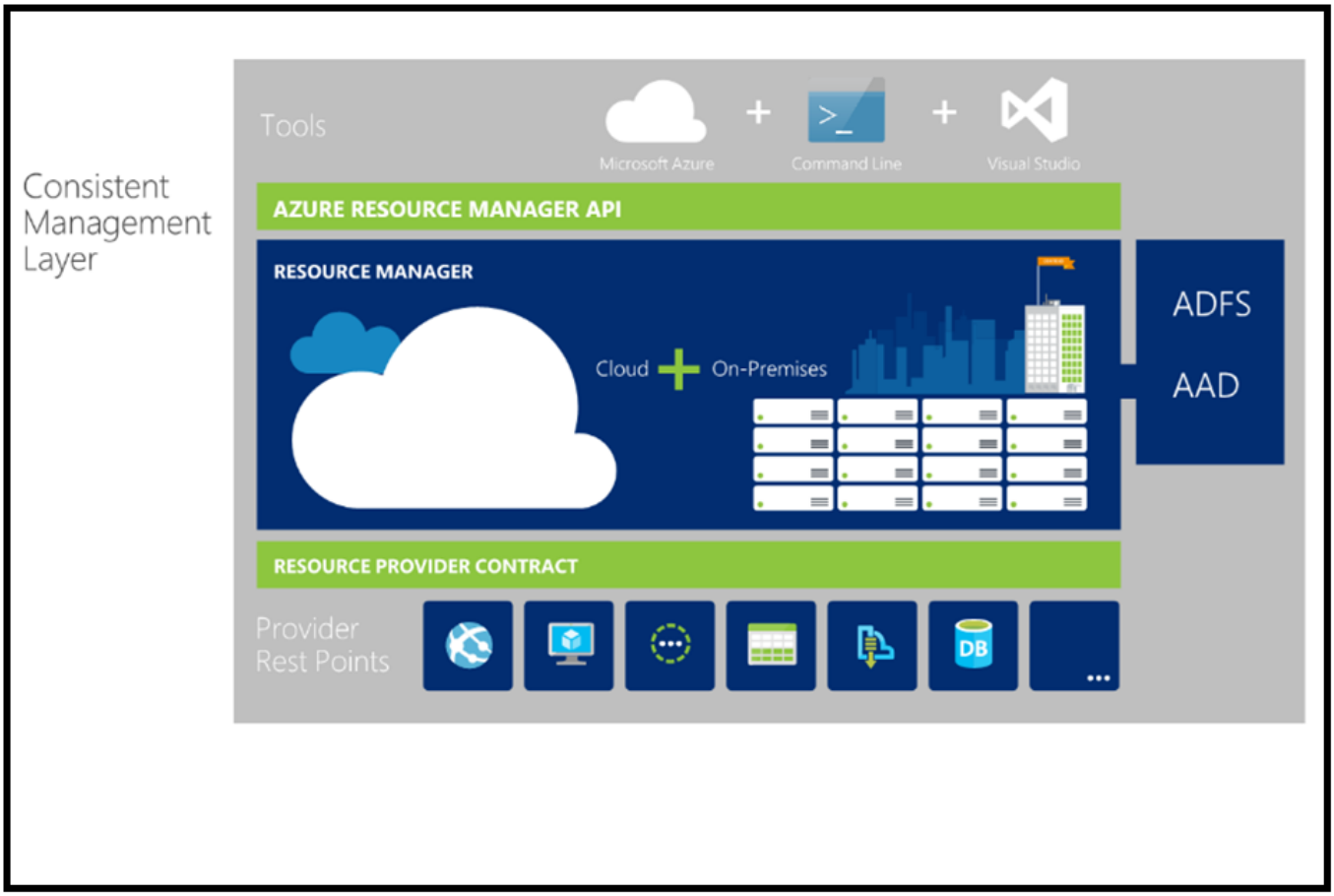
service instance in Azure as a modular resource. With Azure Resource Manager, JSON templates were used to deploy collections of resources using Infrastructure-as-Code concepts. Along with Azure Resource Manager, we saw that release of a new Azure Portal (https://portal.azure.com) that focused on the modular nature of resources. This lesson focuses on Azure Resource Manager (ARM) and how you can use JSON to deploy resources using ARM templates.

Lesson objectives

After completing this lesson, you will be able to:

- Describe ARM Templates.
- Understand the JSON format and how to author a JSON template in various ways
- Describe the format of a JSON file and where to obtain example templates for Azure deployments
- Deploy a resource using the Azure Quickstart templates on GitHub.

Azure Resource Manager



Azure Resource Manager (ARM) is the latest way to conceptualize your Azure resources in your subscription. Service instances are now resources, which are grouped as resource groups. Resource groups provide a common lifecycle for the child resources. They can be created, managed, monitored, or deleted together. The Resource Manager also offers the concept of resource group templates which enable you to define a service unit in advance, and then use the template to create as many resource groups as you need.

Resource groups and resource group templates are ideal for developer operations scenarios where you need to quickly build out development, test, quality assurance, or production environments that are homogenous in nature and can be managed with a shared lifecycle. Developers can quickly delete their environment and create a new environment by using the shared template. The resource groups can be monitored to determine the billing rate or resource usage at a higher level than monitoring individual service instances.

Before using ARM, you need to consider resource providers and ensure your subscription contains registrations for each provider you wish to use. Deploying a complex Virtual Machine template will fail at the first hurdle if you are not registered for the Microsoft.Compute provider. This provider controls access to all Virtual Machine creation.

Azure Resource Manager Objects

When you envision your solution using ARM, you must start by designing and conceptualizing your entire solution considering all components that may compose your solution. Once you have designed your entire solution, you can then identify individual units of functionality and find resources available on Azure that can facilitate the specific functionalities.

You use a template -- a resource model of the service -- to create a resource group with the resources you specified above. After you author the template, you can manage and deploy that entire resource group as a single logical unit. There are three primary concepts in Resource Manager:

- **Resource**: A resource is merely a single service instance in Azure. Most services in Azure have a direct representation as a resource. For example, a Web App instance is a resource. An App Service Plan is also a resource. Even a SQL Database instance is a resource.
- **Resource Group**: A resource group is a group of resources in the logical sense. For example, a Resource Group composed of a Network Interface Card (NIC), a Virtual Machine compute allocation, a Virtual Network, and a Public IP Address creates what we would logically consider a “Virtual Machine.”
- **Resource Group Template**: Every resource group deployment is completed using a JSON file known as the resource group template. This JSON file declaratively describes a set of resources. The deployment adds the new resources to a new or existing resource group. For example, a template could contain the configuration necessary to create 2 API App instances, a Mobile App instance and a Cosmos DB instance.

Interacting with Resource Manager

You can use Resource Manager in a variety of different ways including:

- **PowerShell**: There are PowerShell CmdLets already available to allow you to manage your services in the context of resources and resource groups.
- **Cross-Platform Command-Line Interface**: This CLI allows you to manage your Azure resources from many different operating systems.
- **Client Libraries**: There are client libraries already available for various programming frameworks/languages to create resources and resource groups in Azure.

- **Visual Studio**: Visual Studio 2015 ships with a Resource Manager project type that allows you to create a resource group template by either manually modifying JSON (with schema intellisense) or use scaffolding to update your JSON template automatically.
- **Portal template deployment**: In the portal, you can use the Template Deployment option in the Marketplace to deploy a Resource Group from a template.
- **REST API**: All the above options use the REST API to create your resources and resource groups. If you prefer to create resources without a library, you can always use the REST API directly.

> **Reference Links:** https://docs.microsoft.com/rest/api/resources/

ARM Templates

Azure supports the deployment of resources using Azure Resource Manager templates. In an Azure Resource Manager template, you specify the resource - for example, a virtual network, virtual machine, and so on - using a JSON object. Some or all of the properties of the resource can be parameterized so that you can customize your deployment by providing parameter values at deployment time.

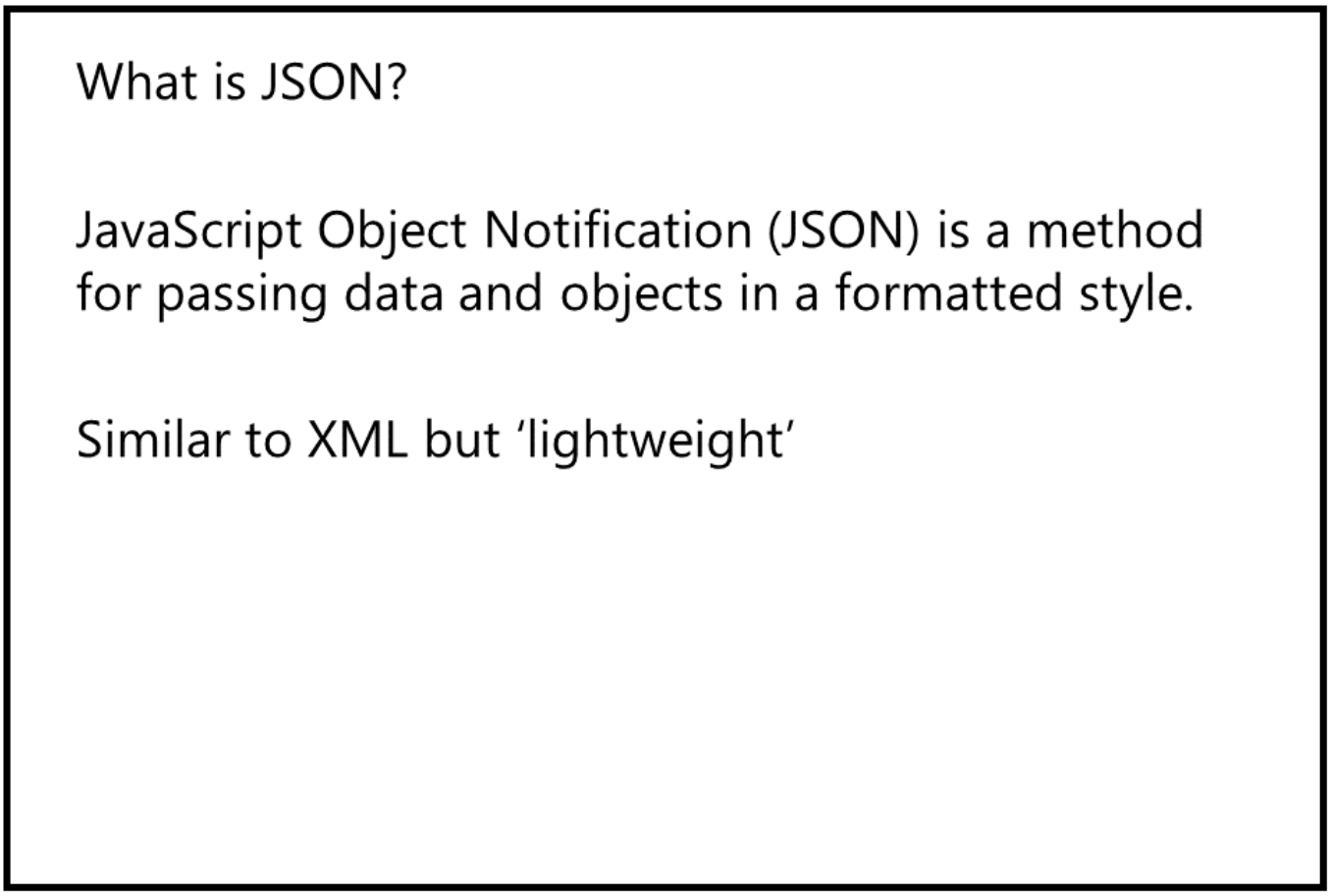
ARM Templates are deployed in a few ways. These depend on your aims, the result intended and your chosen method for development.

A developer may choose to use Visual Studio to create and deploy ARM templates directly and to manage the lifecycle of the resources through Visual Studio.

An administrator may choose to use PowerShell or the Azure Command Line to deploy resources and amend them.

An end user without command line or developer skills would choose to use the Azure Portal to deploy resources without realizing a template is involved. This deployment would typically be a less complicated group of resources unless the user chooses to deploy a marketplace offering.

JSON



What is JSON?

JavaScript Object Notification (JSON) is a lightweight formatted script designed for data transfer. Azure Resource Manager uses this as the foundation for every resource.

Empty ARM Template

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

The format of JSON is flexible in that several elements are optional.

The script must have a schema and a content section as well as a resource section. The other three, parameters, variables and output are optional. In day to day use, most Arm Template JSON contains all sections and even contains logic as well.

Resources can depend on other resources existing and even their names, locations and content. This resource dependency can lead to very complex nested JSON templates when building a sophisticated resource group, such as a SharePoint farm.

The Azure Quickstart templates on Github are an excellent resource for learning to structure and deploy your templates correctly. If you are just beginning on your JSON journey, then the Automation script element of every object in the Azure Portal allows you to learn the constructs and formatting of JSON.

Lesson 2: Role-Based Access Control (RBAC)

Azure role-based access control (RBAC) leverages existing Azure AD user, groups, and services to create a relationship between those identities and components of your Azure subscription. Using RBAC, you can assign roles to existing Azure AD identities that grants them pre-determined levels of access to an Azure subscription, resource group or individual resource.

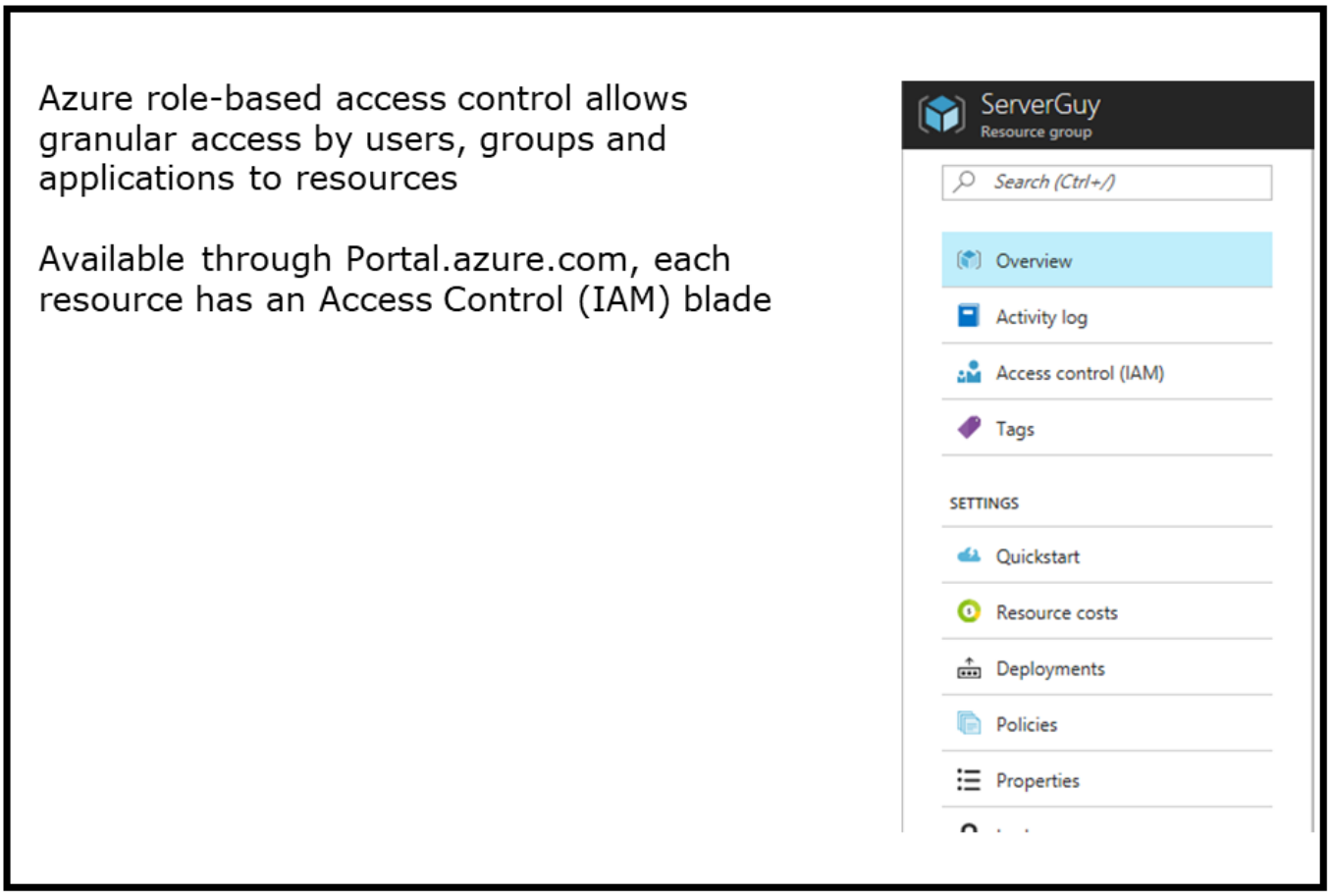
This lesson will discuss the best practices and possibilities of using Azure AD RBAC to manage access to your resources, resource groups, and application.

Lesson objectives

After completing this lesson, you will be able to:

- Describe Azure AD RBAC features.
- Decide how to provide access to your resources using RBAC.
- Allocate Roles to users and groups to provide access to resources.
- Deploy a custom role to an Azure resource group.

Role-Based Access Control



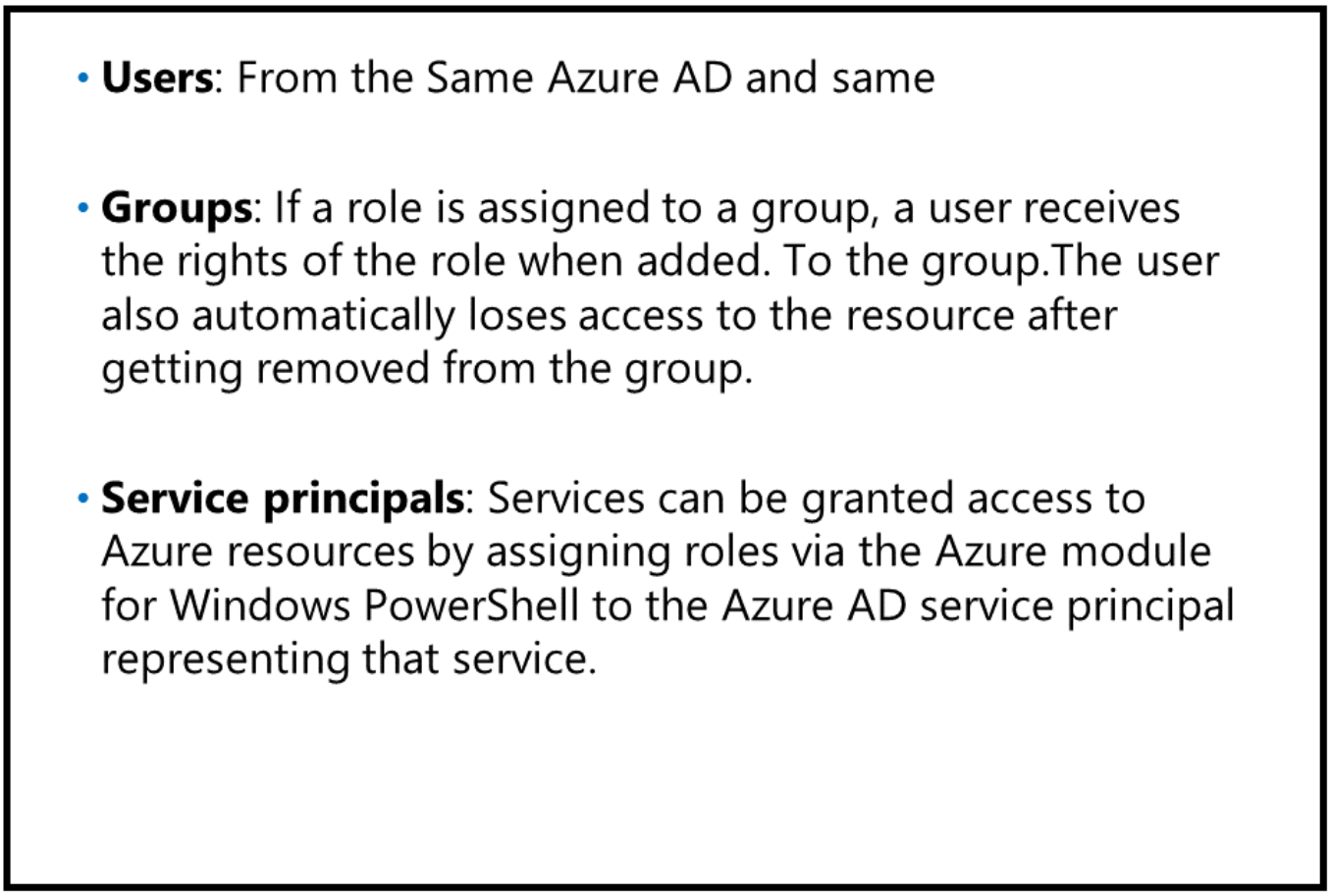
Azure role-based access control allows you to grant appropriate access to Azure AD users, groups, and services, by assigning roles to them on a subscription or resource group or individual resource level. The assigned role defines the level of access that the users, groups, or services have on the Azure resource.

Roles

A role is a collection of actions that can be performed on Azure resources. A user or a service is allowed to act on an Azure resource if they have been assigned a role that contains that action. There are built-in roles that include (but is not limited to):

ROLE NAME	DESCRIPTION
Contributor	Contributors can manage everything except access.
Owner	Owner can manage everything, including access.
Reader	Readers can view everything, but can't make changes.
User Access Administrator	Allows you to manage user access to Azure resources.
Virtual Machine Contributor	Allows you to manage virtual machines, but not access to them, and not the virtual network or storage account they are connected to.

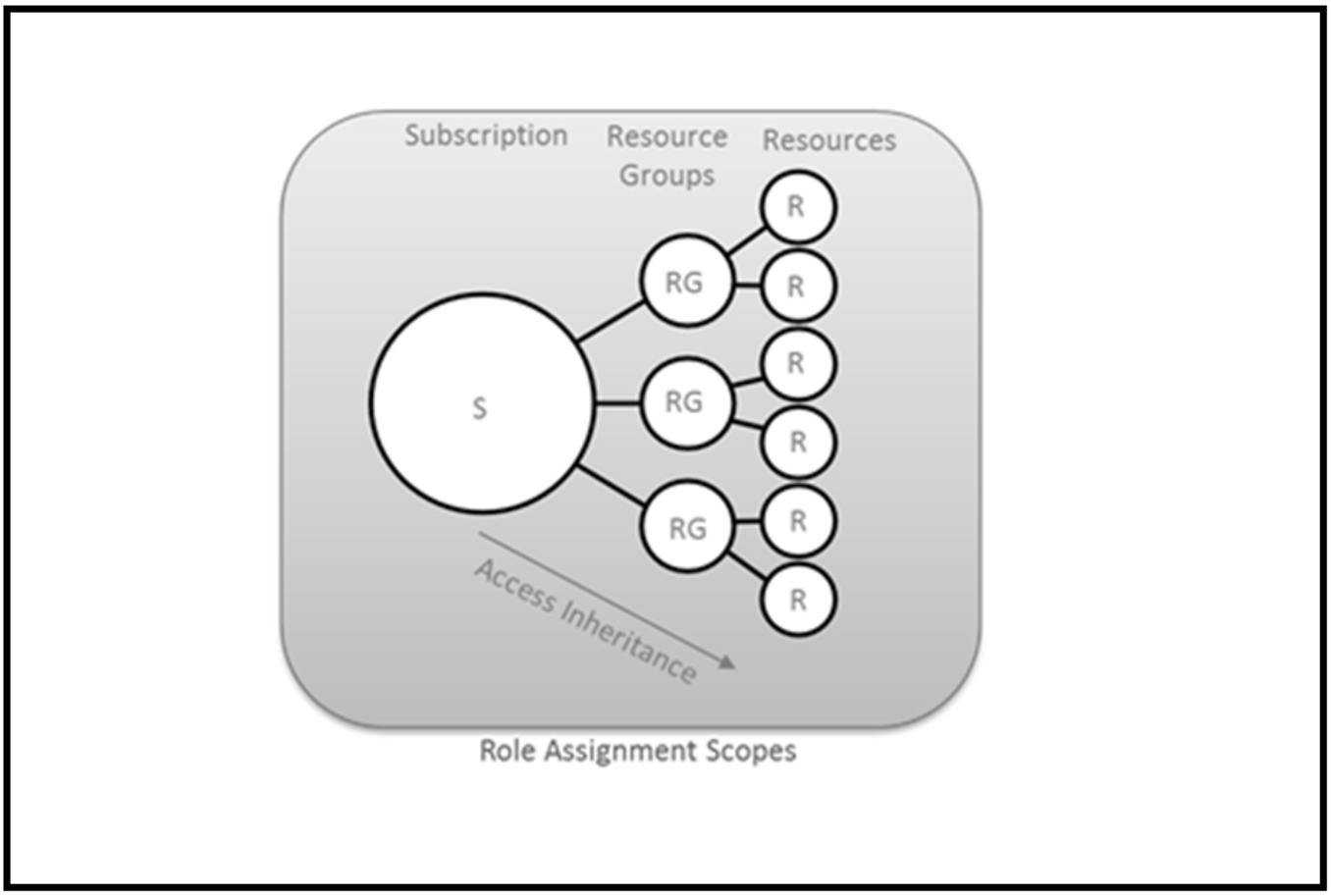
Role Assignment



A role assignment can be created that associates a security principal to a role. The role is further used to grant access to a resource scope. This decoupling allows you to specify that a specific role has access to a resource in your subscription and add/remove security principals from that role in a loosely connected manner. Roles can be assigned to the following types of Azure AD security principals:

- **Users**: roles can be assigned to organizational users that are in the Azure AD with which the Azure subscription is associated. Roles can also be assigned to external Microsoft accounts that exist in the same directory.
- **Groups**: roles can be assigned to Azure AD security groups. A user is automatically granted access to a resource if the user becomes a member of a group that has access. The user also automatically loses access to the resource after getting removed from the group. Managing access via groups by assigning roles to groups and adding users to those groups is the best practice, instead of assigning roles directly to users.
- **Service principals**: service identities are represented as service principals in the directory. They authenticate with Azure AD and securely communicate with one another. Services can be granted access to Azure resources by assigning roles via the Azure module for Windows PowerShell to the Azure AD service principal representing that service.

Resource Scope



Access does not need to be granted to the entire subscription. Roles can also be assigned to resource groups as well as for individual resources. In Azure RBAC, a resource inherits role assignments from its parent resources. So if a user, group, or service is granted access to only a resource group within a subscription, they will be able to access only that resource group and resources within it, and not the other resources groups within the subscription. As another example, a security group can be added to the Reader role for a resource group, but be added to the Contributor role for a database within that resource group.

Scoping to Resource Groups

While RBAC can easily be used to grant access to an individual resource, it is preferable to grant access to an entire Resource Group as opposed to individual resources. By scoping to a resource group, you can add/remove and modify resources quickly without having to recreate assignments and scopes. You can also give an individual owner or contributor access to a resource group so that they can create, recreate or destroy resources on their own without requiring involvement from the account administrator.

Let’s explore a resource group for an actual application.

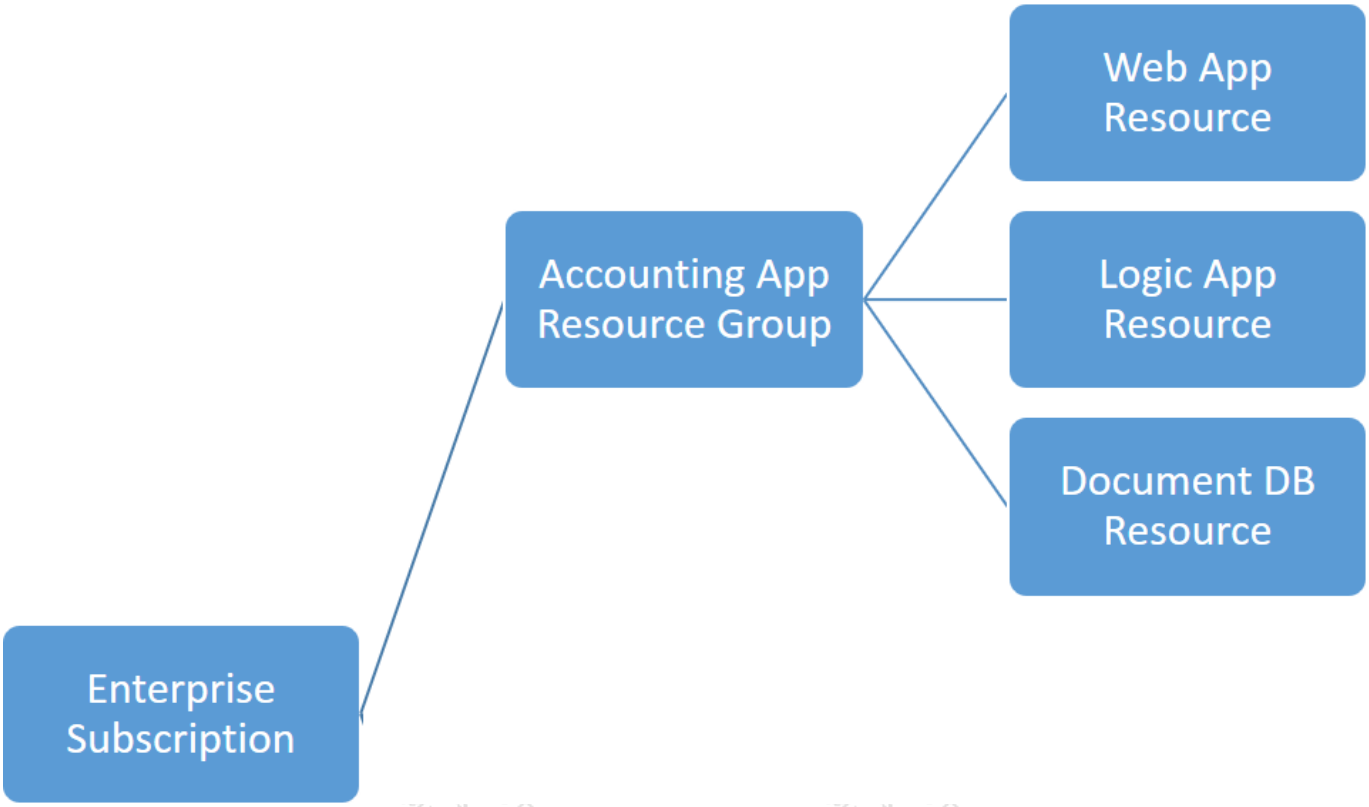


FIGURE 2.1: IN THIS EXAMPLE, THERE IS A RESOURCE GROUP FOR AN ACCOUNTING APPLICATION. THIS RESOURCE GROUP CONTAINS MULTIPLE RESOURCES THAT ARE USED IN THE APPLICATION.

By granting an individual owner or contributor access to the resource group, they can configure resources, create new deployments, add new resources or create automation scripts at will without requiring additional administrator assistance or having access to resources in other resource groups.

Custom Roles

If there is no role suitable for your purposes or granular enough to suit your needs, it is possible to create a custom role and apply that. To create a custom role, you must use either Azure PowerShell or the Azure Command-Line Interface.

It is also possible to use the REST API to create roles programmatically. Each Azure AD tenant is limited to 2000 custom roles.

To create a new custom role you run the New-AzureRmRoleDefinition Cmdlet, you can pass a JSON template to the CmdLet or use a PSRoleDefinitionObject.

JSON required for a new Custom Role

Custom Role

```
{
  "Name": "New Role 1",
  "Id": null,
  "IsCustom": true,
  "Description": "Allows for read access to Azure storage and compute resources",
  "Actions": [
    "Microsoft.Compute/*/read",
    "Microsoft.Storage/*/read",
```

```
  ],
  "NotActions": [
  ],
  "AssignableScopes": [
    "/subscriptions/c489345-9cd4-44c9-99a7-4gh6575315336g"
 ]
}
```

If you save the JSON to a file such as C:\CustomRole\newrole1.json, you can use the following PowerShell to add the custom role to a subscription:

Command to add the Custom role to the Subscription.

Custom Roles in PowerShell

```
New-AzureRmRoleDefinition -InputFile "C:\CustomRole\newrole1.json"
```

Lesson 3: Resource Policies

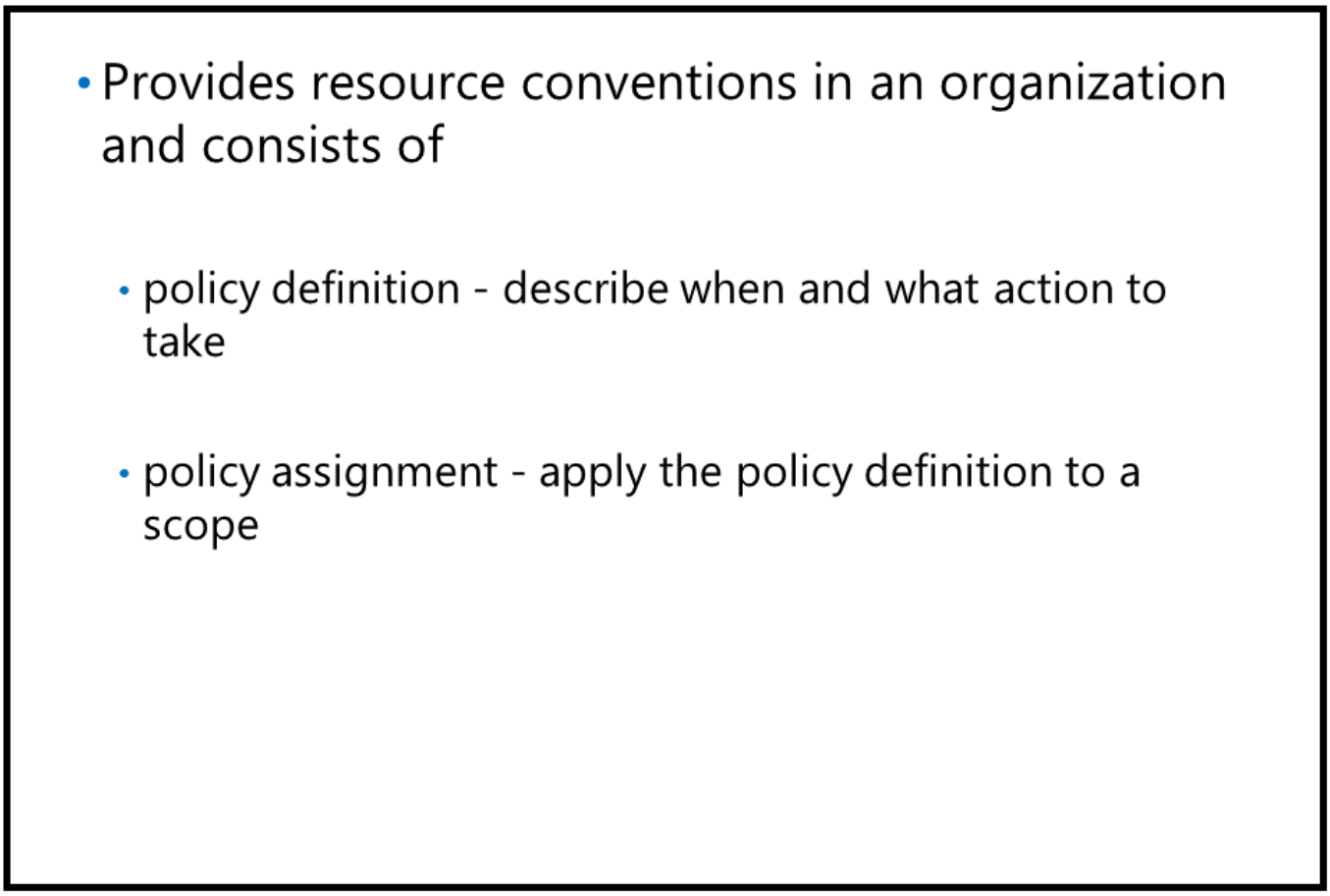
The use of RBAC controls users access to resources. Resource Policy is a preview service that controls which resources can be created and what for they take, such as naming conventions, locations, and sizes. This lesson will describe the features available and show the ways of defining, assigning and monitoring those policies.

Lesson objectives

After completing this lesson, you will be able to:

- Describe Azure Policy service.
- Decide when to apply policy definitions and to what scope.
- Author an Azure Policy definition.
- Assign and monitor an Azure resource policy.

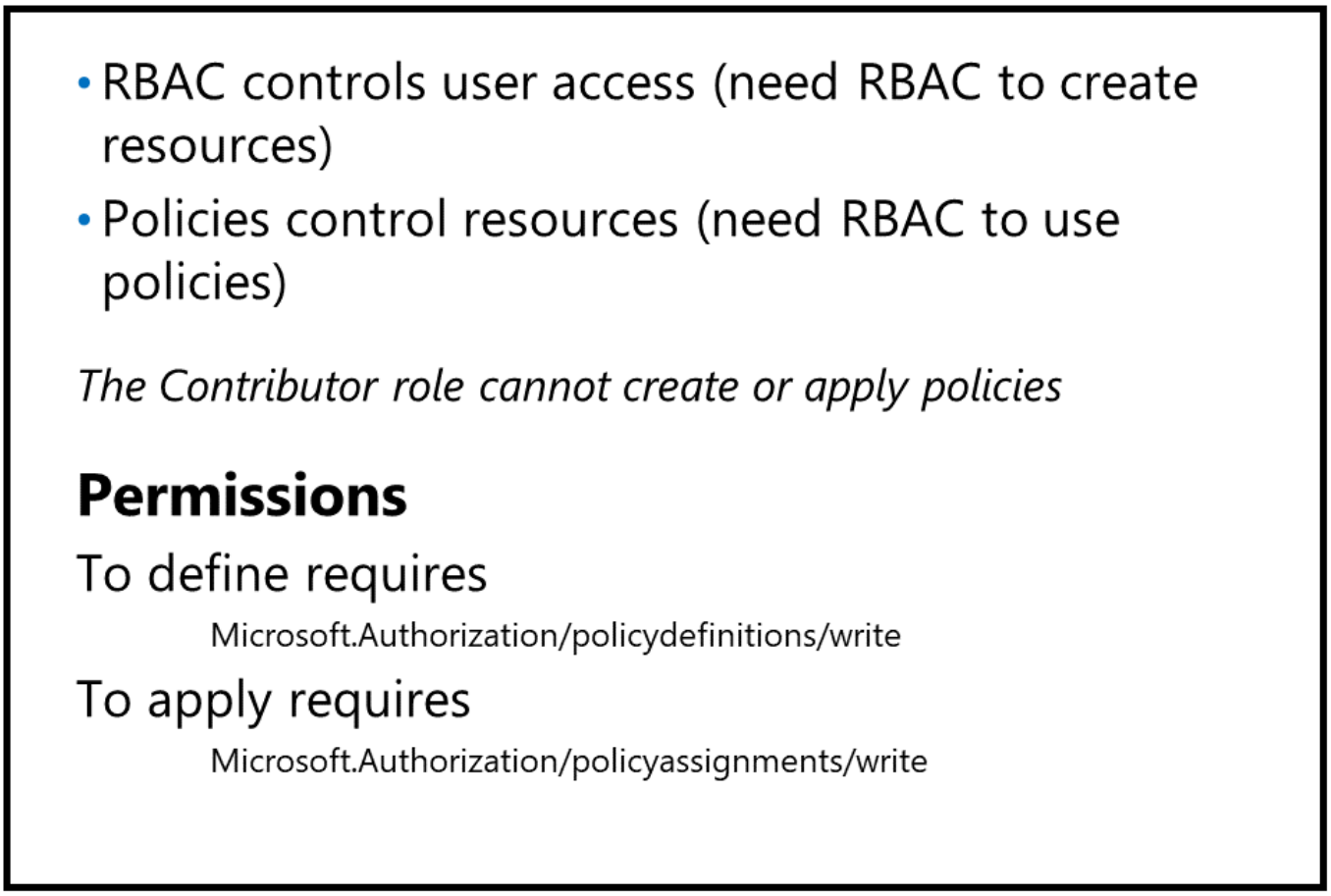
Azure Resource Policies



Azure Policy is a new service designed to allow an organization to ensure that resources created in the cloud comply with corporate standards and service level agreements. As an example, preventing users from creating resources in specific locations and of specific types and sizes. If your organization has no need for a Virtual Machine with 32 CPU cores and 500GB of RAM in the more expensive UK South region, then this can be monitored and prevented with Azure Policy.

An Azure policy contains two elements, a policy definition, and a policy assignment. This design allows an organization to create a library of policy definitions to be assigned later. There are many pre-defined policy definitions built into Azure.

Policy vs RBAC



There are differences between policy and role-based access control (ABC). RBAC controls user access, permissions, privileges, and actions at different scopes. As an example, you could add a user to the Owner role for a resource group as the desired scope. The Owner role gives you full control of that resource group.

Policy evaluates resource properties for already existing resources and during deployment. As an example, using policies, you can control which type of resource is available for deployment in your organization. Other policies can limit the locations in which you deploy resources or require your resources to follow naming conventions.

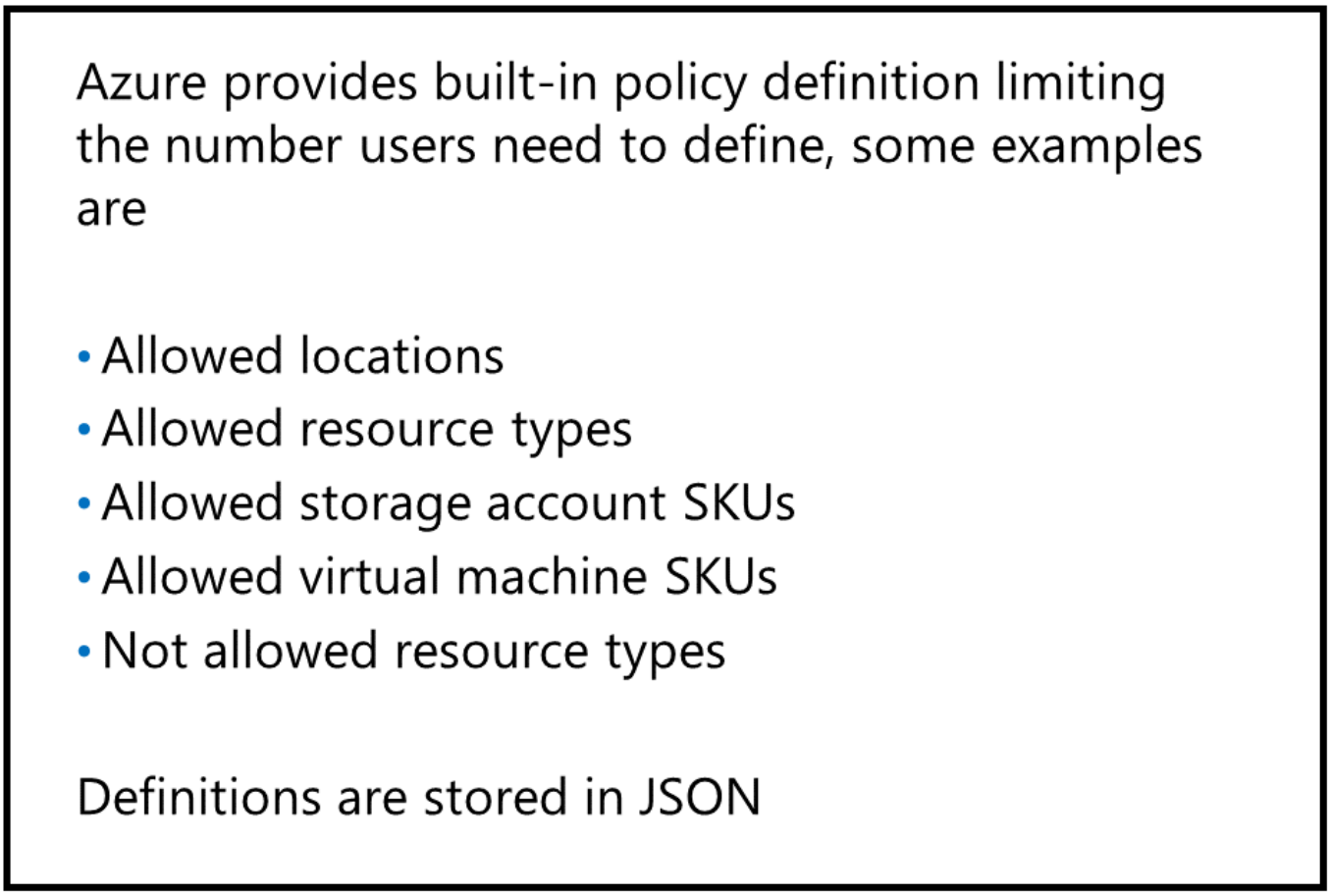
Azure policy is a default allow, and explicit deny system. This is not the same as RBAC.

To be able to create and assign Azure policies, you need to be assigned permissions using RBAC; the contributor role does not contain the necessary permissions.

The permissions required are:

- Microsoft.Authorization/policydefinitions/write permission to define a policy.
- Microsoft.Authorization/policyassignments/write permission to assign a policy.

Built-In Policies

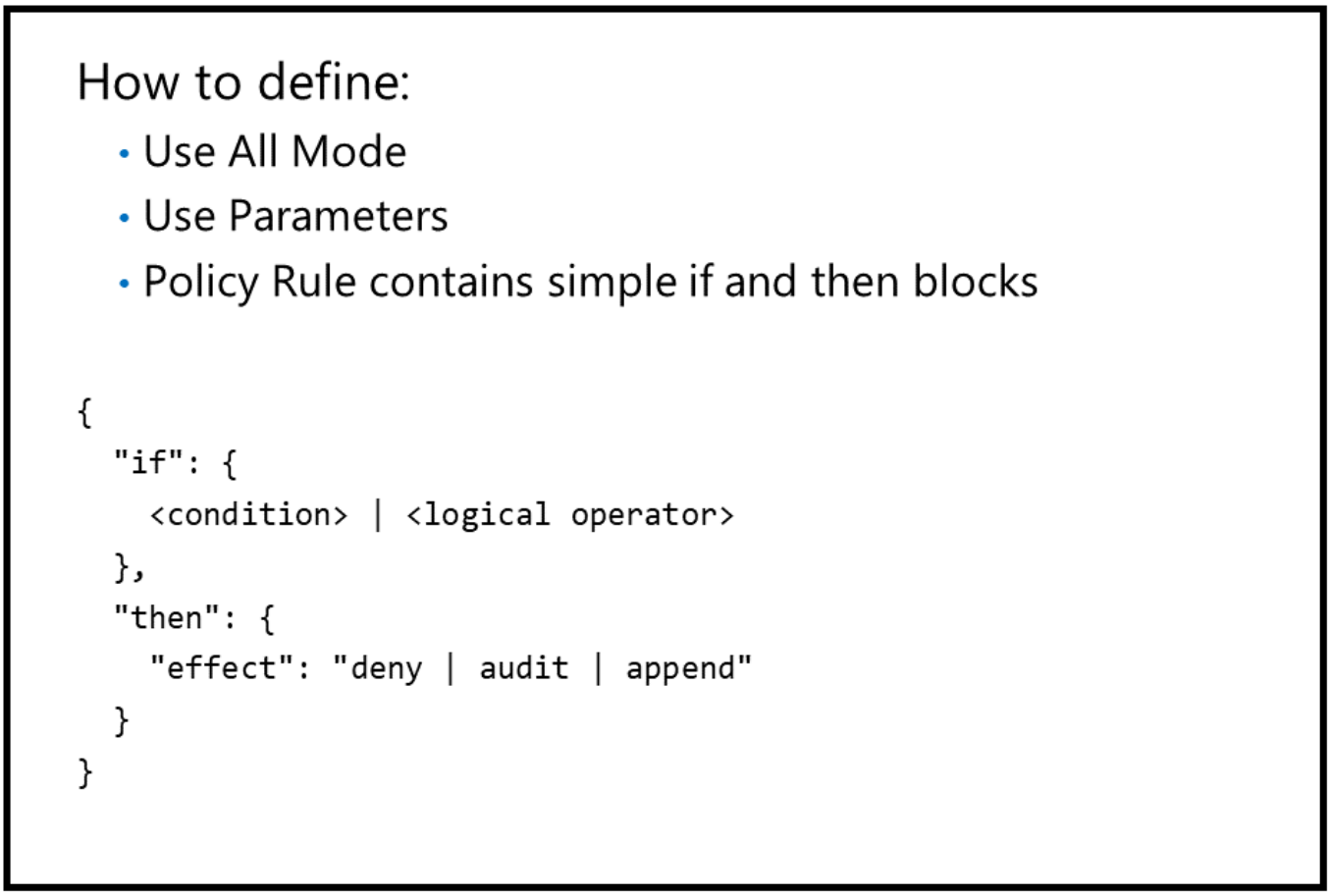


Azure has a built-in library of Policies to allow organizations to control their resources. These include:

- Allowed locations
- Allowed resource types
- Allowed storage account SKUs
- Allowed virtual machine SKUs
- Apply tag and default value
- Enforce tag and value
- Not allowed resource types
- Require SQL Server version 12.0
- Require storage account encryption

The inclusion of these built-in policy definitions limits the number a user is required to create to manage their subscription efficiently.

Policy Definition



Every policy definition contains a minimum of two elements, first the conditions under which it is enforced. Secondly, it has an action that triggers if the conditions are met.

Policy definitions are created using JSON. A policy definition contains elements to define:

- mode
- parameters
- display name
- description
- policy rule
- logical evaluation
- effect

The following example shows a policy that defines not allowed resources (This is a built-in policy definition):

policy not allowed.json

```
{
  "if": {
    "field": "type",
    "in": "[parameters('listOfResourceTypesNotAllowed')]"
  },
  "then": {
    "effect": "Deny"
  }
}
```

FIGURE 2.2: EXAMPLE POLICY

At assignment the list of resource types not allowed parameter is populated and evaluated against all current resources ins cope for the assignments. Any non-compliant resources will be identified. When creating new resources, the deployment will fail if they are non-compliant. The above is a straightforward policy definition. The JSON for policy definitions can be much more complicated.

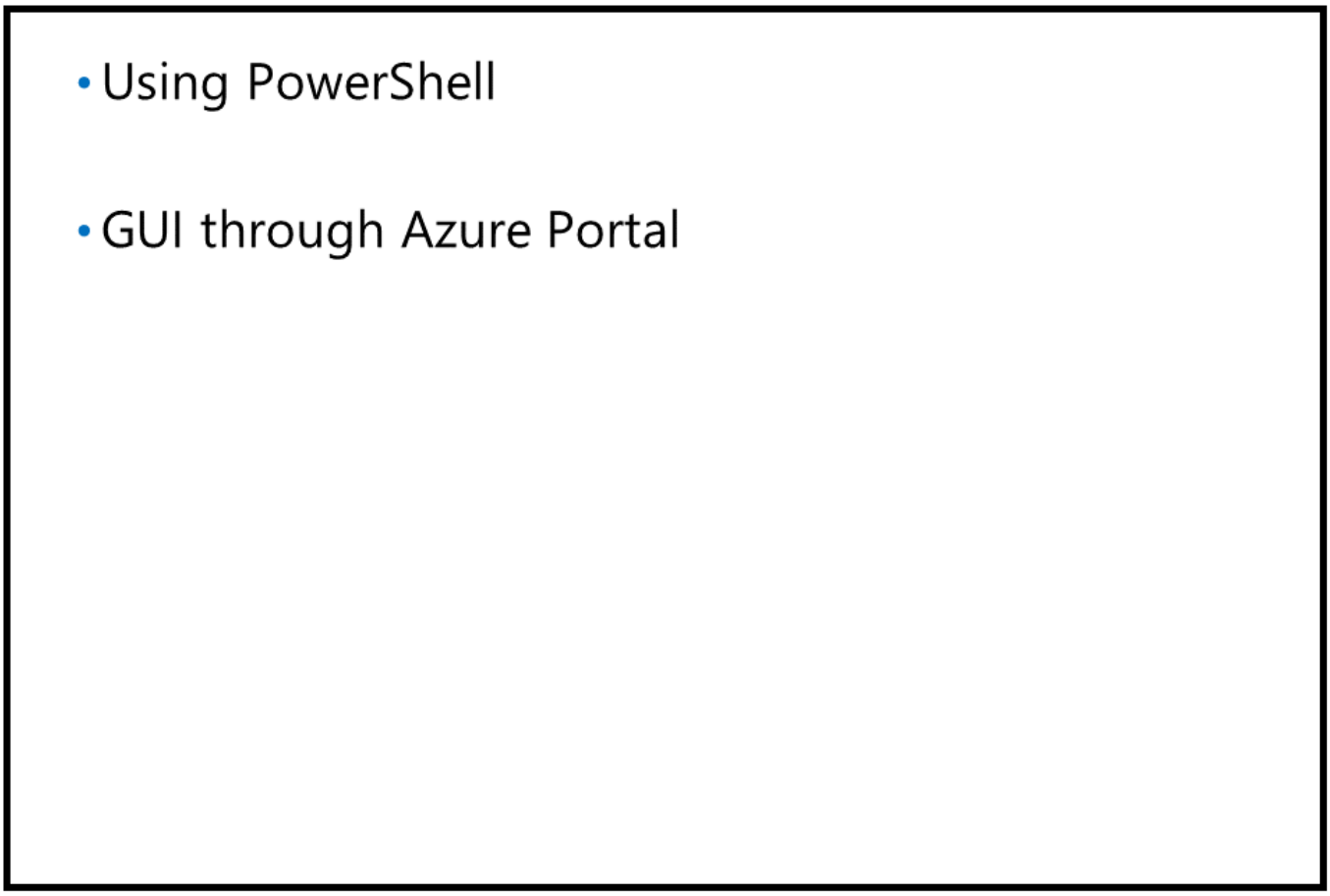
The definition can contain multiple if, then blocks and combines logical operators, conditions, and fields to build out the policy. In addition, the policy can define alternative effects.

The available effects are:

- Deny
- Audit
- Append
- AuditIfNotExists
- DeployIfNotExists

You can assign any of these policy definitions through the Azure portal, PowerShell, or Azure CLI.

Policy Assignment



Once a policy definition has been created, this is known as a custom policy definition and can be assigned to take effect over a specific scope. This scope could range from several subscriptions within an enterprise (known as a management group) to a resource group. A scope is the list of all the resource groups, subscriptions, or management groups that the policy definition is assigned to.

If a policy is assigned to a resource group, the same policy applies to all the resources contained in that group. Policy assignments are inherited by all child resources. This can be altered or prevented using the exclusion option at the time of assignment. As an example, a policy assignment could prevent the creation of SQL databases within your subscription. An exclusion could allow this in a single Resource group, which you want to keep as your database Resource Group. RBAC is then used to restrict access to the SQL database resource group to the trusted group of users that have the necessary skills and requirements for these resources.

A policy definition will create parameters to be applied at the time of Policy assignment. This allows for one definition to be re-used across several resource groups, locations, and subscriptions.

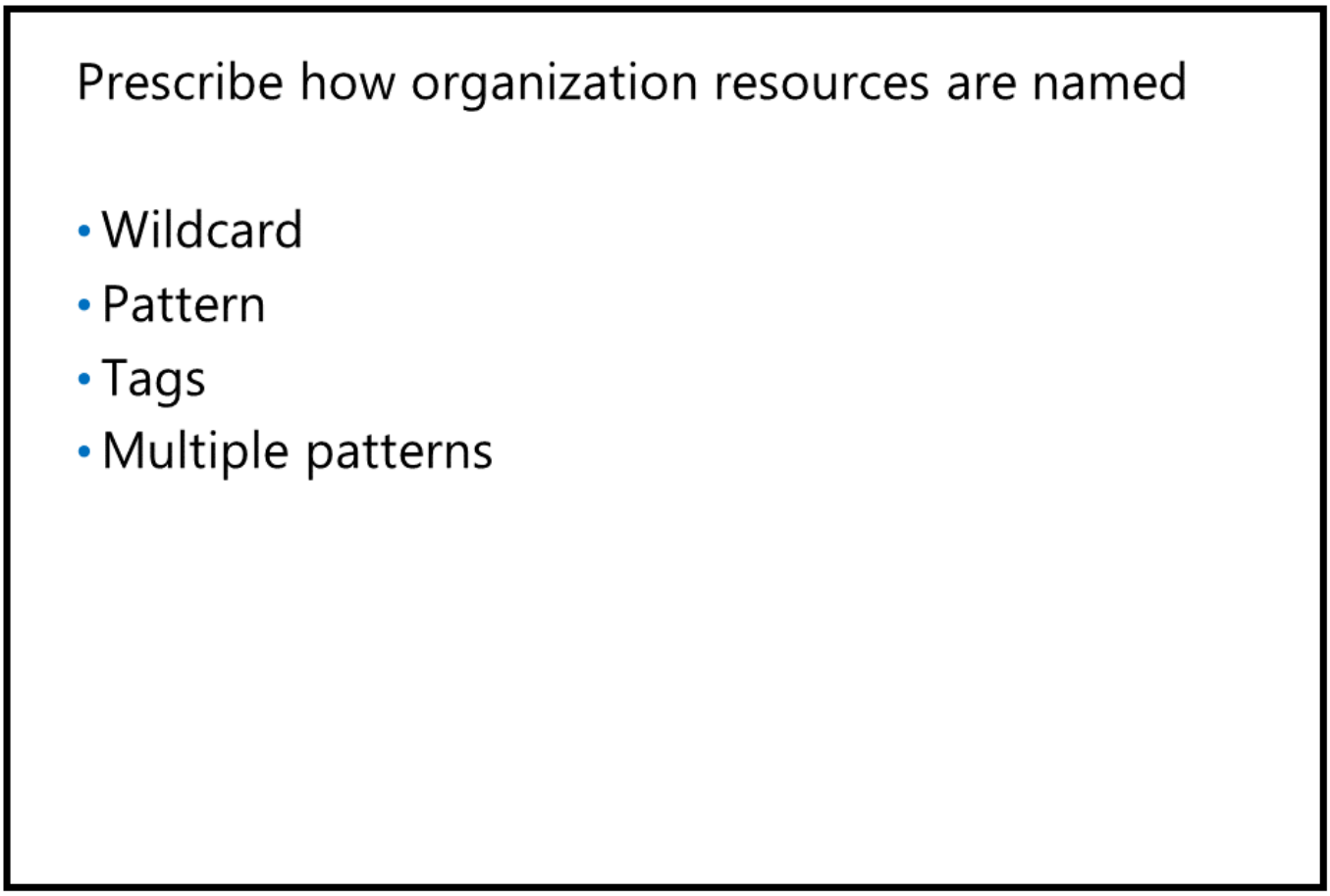
The scope, exclusions, and parameters of the Not Allowed resource types are shown in the policy assignments blade when assigning a definition.

Azure policies can be assigned using the Azure Portal, Azure PowerShell, and the Azure Command Line Interface.

Initiative definition

An initiative definition is a collection of policy definitions that are designed to achieve a design goal. Initiative definitions contain policy definitions and can be assigned as a group to simplify the process of achieving that goal. The definition appears as a single item rather than multiple definitions.

Policies for Naming Conventions



Azure policies can be used to maintain control on the naming conventions for all resources within your subscriptions.

A policy definition can be created to enforce naming conventions. These can use wildcards, patterns tags, and multiple patterns to apply restrictions to your Azure resource names.

The example below applies a pattern match asserting that is the resource name does not begin Contoso and have six characters as the rest of the name then it will be non-compliant.

Example Naming Pattern

```
{
  "if": {
    "not": {
      "field": "name",
      "match": "contoso??????"
    }
  },
  "then": {
    "effect": "deny"
  }
}
```

The effect of non-compliance is deployment failure on the creation of a resource or listing in the non-compliant reporting if the resource is already in existence.

Lesson 4: Security

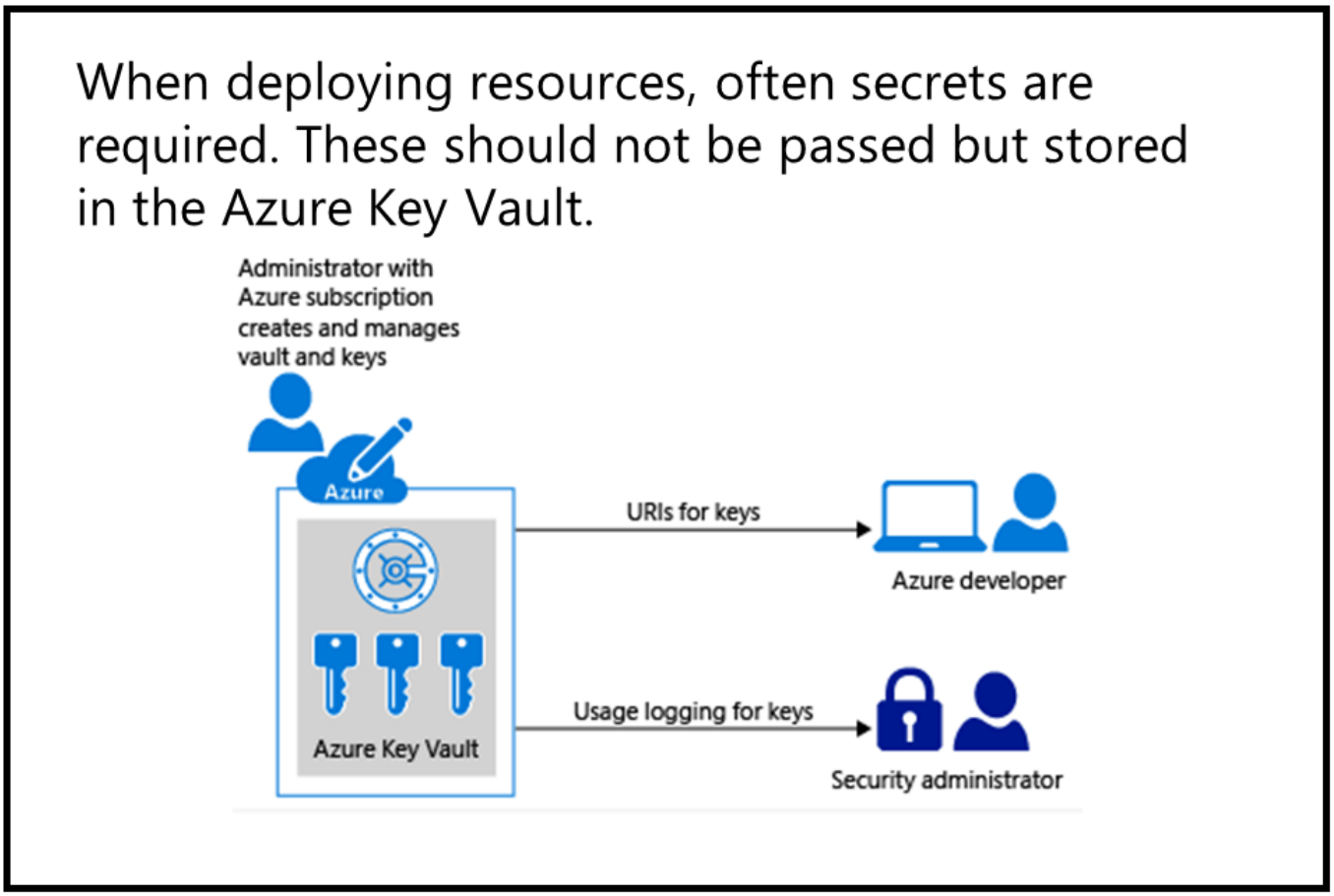
When deploying resources using Arm templates and automating that deployment, it is best practice to use a Service Principal, the Azure equivalent of an Active Directory Service Account. This removes the risk whereby an administrator account is stored and used to deploy resources. To facilitate this in a secure manner, Azure Resource Manager can use the Azure Key Vault to store the service principal secrets. This lesson describes the Key Vault and its use in deploying Arm templates securely.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Azure Key Vault service.
- Describe the secure deployment of templates using Service Principals and the Key Vault.
- Create a Service Principal.
- Deploy a resource ARM templates and the Azure Key Vault

Azure Key Vault



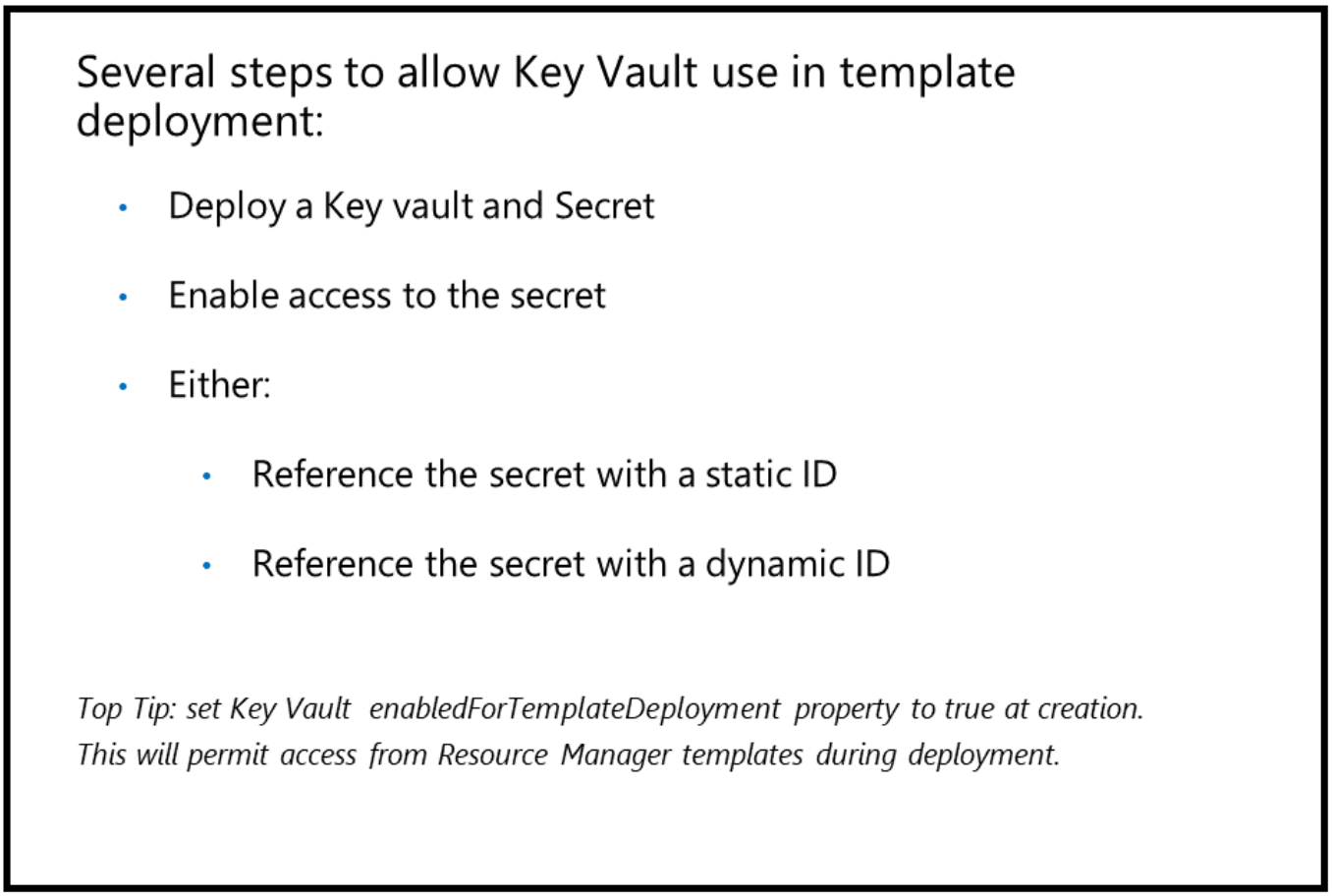
Azure Key Vault is a two-tier service that allows secure storage of cryptographic keys, certificates, and secrets. These can be used by applications, services, and users. The premium tier allows storage of these secrets in a Hardware Security Module, a physical device to contain all your secrets.

Key Vault makes the process of creation, importing, maintaining and recycling secrets much easier. The developer can easily create dev and test keys that can be migrated to production use at deployment. Security administrators can manage all the secrets in the Key Vault with ease and grant and revoke access when required. Key Vault can be used for several scenarios in Azure.

App developers want to use keys for signing and encryption but want all keys outside of the application. App developers do not want to manage keys but wants their customers to bring their own keys and certificates.

Security admins want to ensure that company applications comply with regulatory bodies requirements such as FIPS. Key Vaults can be created in any Azure subscription by any contributor or Owner. Key Vaults can be used to create, manage and import secrets, keys, and certificates for applications, services and users.

Key Vault Use in ARM Templates



When deploying resources into Azure using ARM Templates, a user credential is often required to allow the resources to be created. Best practices dictates that embedding credentials and passwords inside a template are unwise. Key vault can be used to pass a secure value as a parameter during deployment. The secure value, in this instance a password can be retrieved from the Key Vault, retrieval is accomplished by referencing both the key vault id and the secret in the parameter file for the template. The benefit is that the value of this secret is never exposed. To further secure the deployment, it is advised to create an Azure Service Principal, which is the equivalent of an Active Directory Service Account. This reduces risk by allowing you to limit the privileges of the Service Principal to only what is needed.

The value is never exposed because you only reference its key vault ID. At deployment time you are not required to enter credentials. There are several steps required to allow a Key Vault to be used during template deployment. First, it is essential to set the enabledForTemplateDeployment property to true when you create the Key Vault.This setting allows access to the key Vault from Resource manager at deployment time.

Next, you must create a secret using the Azure Portal, PowerShell or Azure CLI. Having created a secret for use by the template you must give the template access to the vault and the secret. This is achieved by ensuring the service principal, user or template has the Microsoft.KeyVault/vaults/deploy/action permission for the correct Key Vault. The Contributor built-in role already has this permission. The last step is to reference the secret using a static ID in the template parameter file. Sometimes the Key Vault secret will change from deployment to deployment; this requires the use of a dynamic ID reference which cannot go in the parameter file and calls for a nested template to achieve this.

Lesson 5: Building Blocks

While authoring ARM templates can give you the most flexibility when automating deployments, ARM templates can become very complex in a short amount of time. Additionally, it may be difficult to ensure that Microsoft best practices for Azure infrastructure are reflected in every template authored by your team.

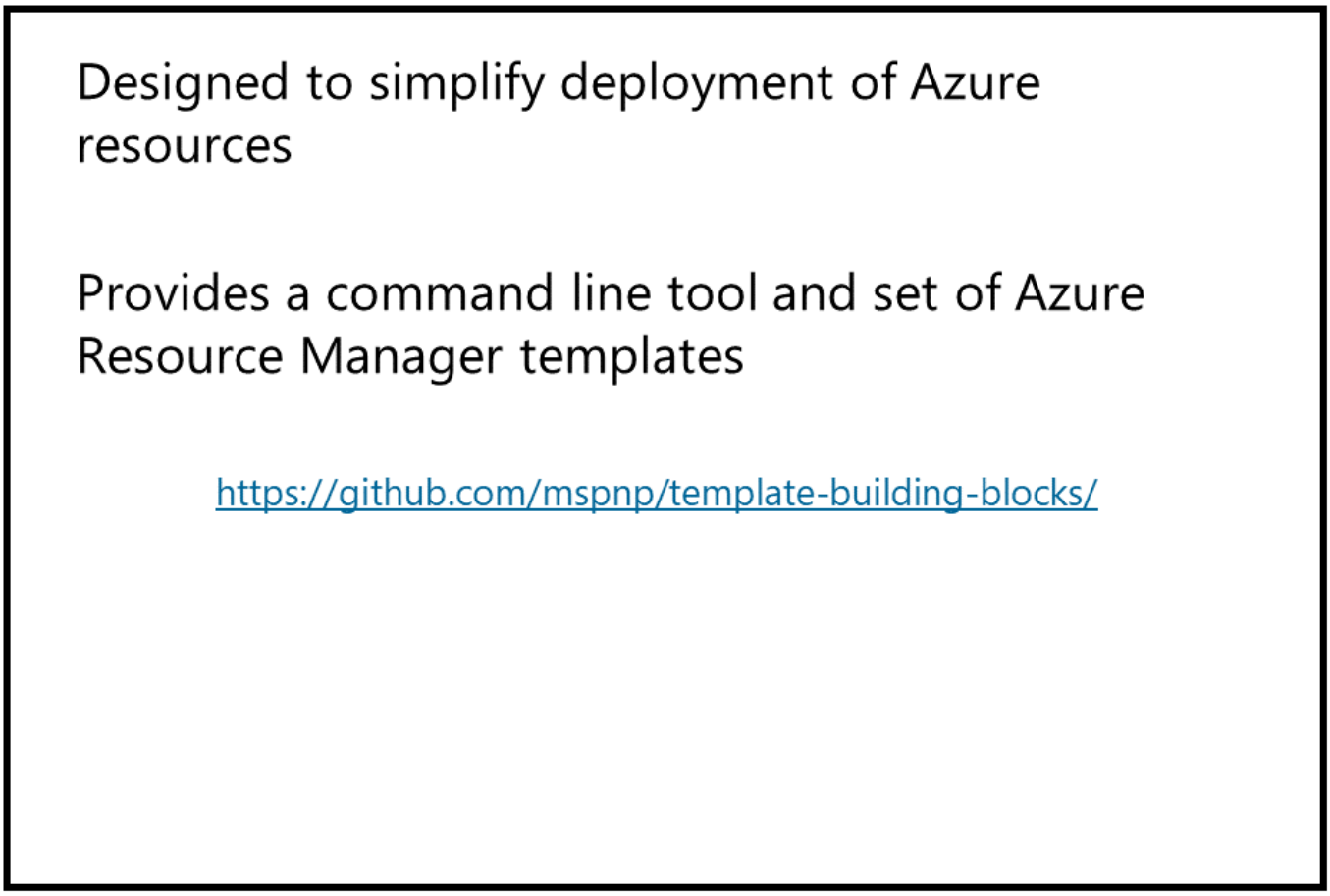
This lesson introduces the Azure Building Blocks, an open-source tool and repository to help simplify the authoring and deployment of ARM Templates. This tool includes ARM templates that already reflect best practices as prescribed by the Patterns & Practices team at Microsoft.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Azure Building Blocks and how they fit into a deployment automation solution.
- Decide when to author an ARM template manually or when to use the Azure Building Blocks.
- Author an Azure Building Blocks parameters file.
- Deploy a resource using the Azure Building Blocks command-line tool.

Azure Building Blocks



Azure supports the deployment of resources using Azure Resource Manager templates. In an Azure Resource Manager template, you specify the resource - for example, a virtual network, virtual machine, and so on - using a JSON object. Some or all of the the properties for the resource can be parameterized so that you can customize your deployment by providing parameter values at deployment time.

If you have spent time developing and maintaining Azure Resource Manager templates, you may have noticed that development of your templates follows a pattern. First you specify the JSON object for a virtual network, then you specify the JSON object for virtual machines that are deployed into the virtual network. Then you specify a JSON object for a network security group to secure the virtual network, and you might specify the JSON object for a load balancer to distribute incoming requests to the virtual machines. Perhaps you have parameterized some of the property values so you can customize the deployment.

While Azure Resource Manager templates are very powerful and allow you to deploy very large and complex architectures to Azure, they require a great deal of knowledge about Azure Resource Manager and the resources themselves. This leads to difficulty maintaining your templates because any modification can lead to unforeseen issues.

The Azure Building Blocks project solves this problem by providing a command line tool and set of Azure Resource Manager templates designed to simplify deployment of Azure resources. You specify settings for Azure resources using the Building Blocks JSON schema, and the command line tool merges these settings with best practice defaults to produce a set of parameter files. The command line tool deploys these parameter files using a set of pre-built Azure Resource Manager templates.

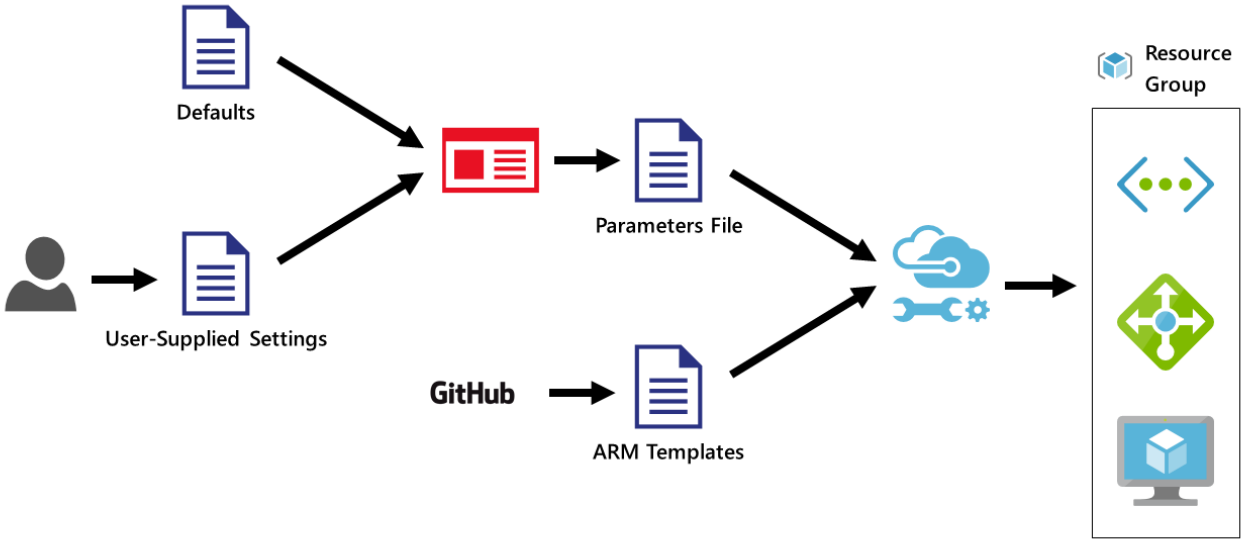


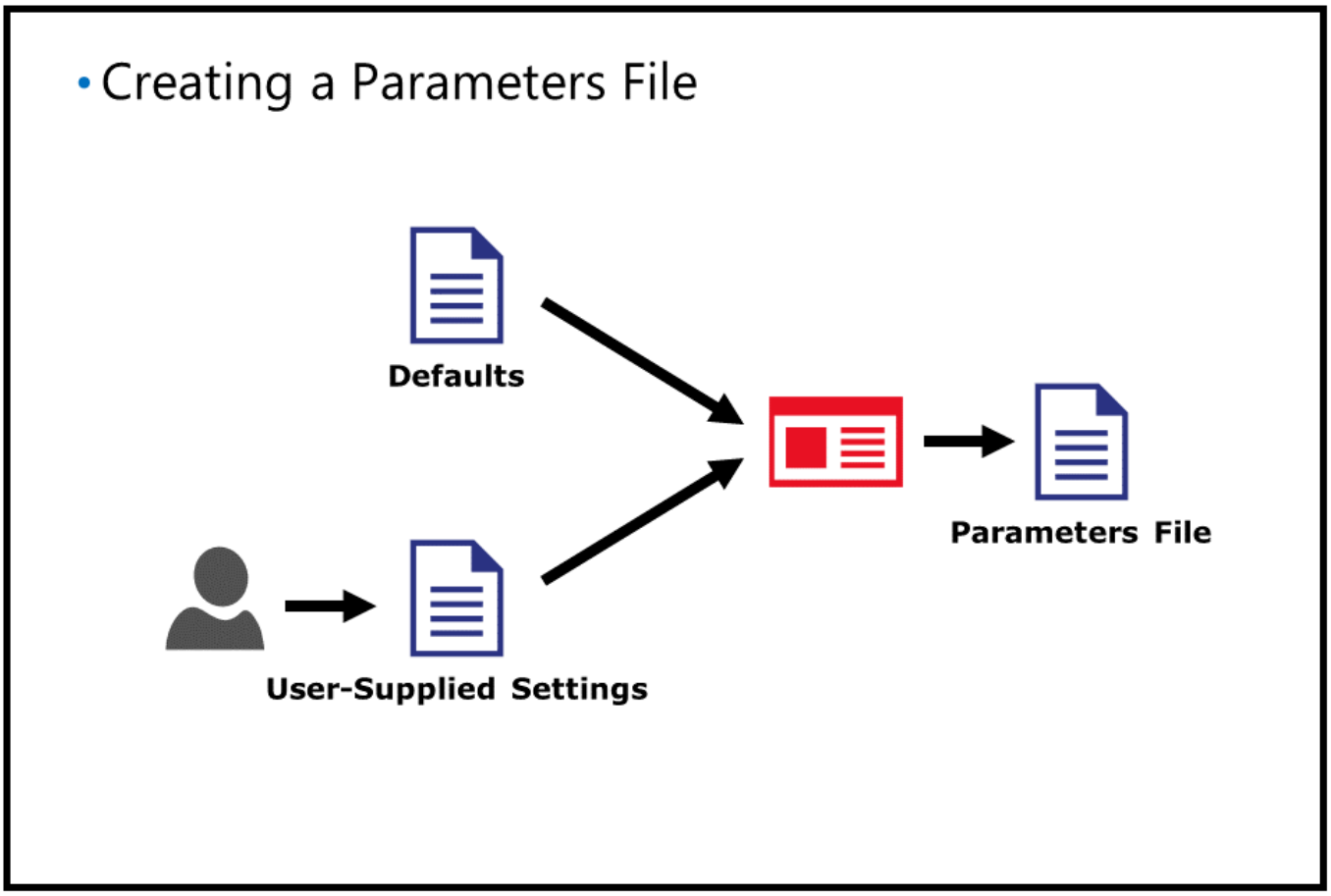
FIGURE 2.3: BUILDING BLOCKS WORKFLOW

The Building Blocks JSON schema is designed to be flexible. You can either specify your resources settings in one large file or several small files.

The Template Building Blocks currently support the following resource types:

- Virtual Networks
- Virtual Machines (including load balancers)
- Virtual Machine Extensions
- Route Tables
- Network Security Groups
- Virtual Network Gateways
- Virtual Network Connection

Deploying Resources using Building Blocks



To demonstrate how to use the Azure Building Blocks, we will use the tool to deploy a simple virtual network. This example assumes the azbb command line tool and the Azure CLI are both installed on your local machine.

Example Architecture

We'll begin our architecture with a simple virtual network:

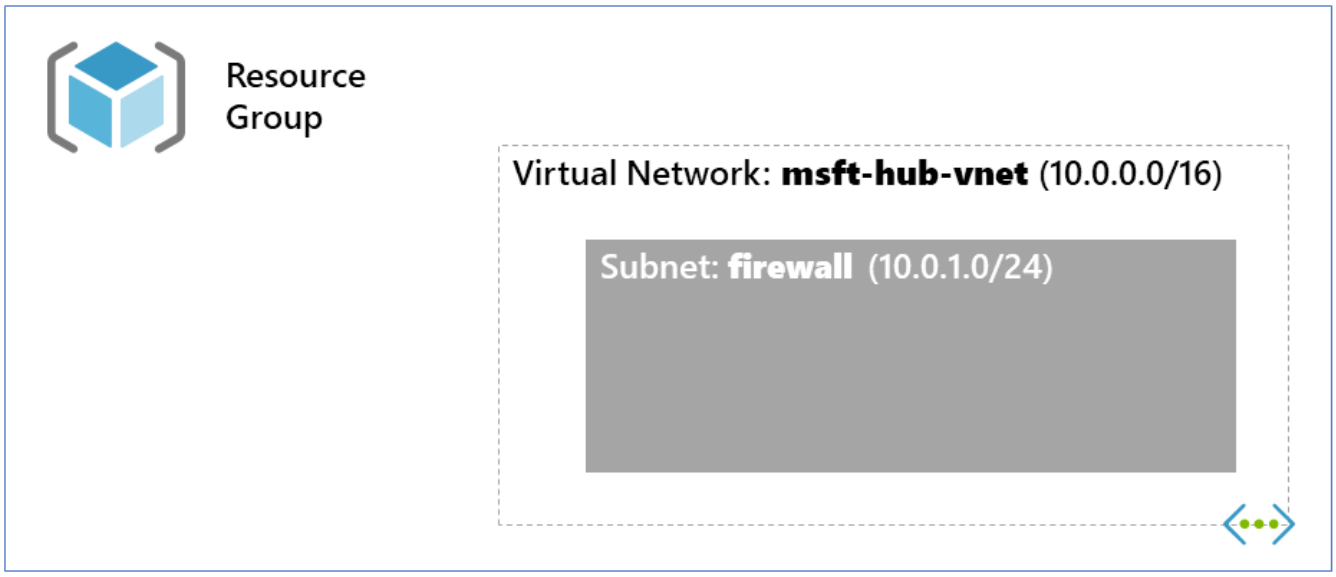


FIGURE 2.4: SAMPLE RESOURCE GROUP

The architecture includes the following:

- A VNet named msft-hub-vnet with an address space of 10.0.0.0/16.

- A subnet within msft-hub-vnet named firewall with an address space of 10.0.1.0/24.

Building Block Resource

Every block can be represented as a JSON object. For our example, we will use the following JSON object to represent our simple Virtual Network:

The type property is used by the Azure Building Blocks to identify the type of building block. We're going to deploy a VNet, so we have to specify a VirtualNetwork building block type. This property is required for each of the building blocks.

Every Azure Building block also requires a settings object where the properties for the building block are specified.

Let's look at each property for a simple VNet:

- **Name**

  In Azure, each resource requires a name to uniquely identify the resource within a resource group. In Azure Building Blocks, you specify a name property for each resource type to provide this unique name. When we deploy this settings file using the command line tool, this is the name that we'll see in the Azure portal user interface. In this settings file we've named the VNet msft-hub-vnet because this in future tutorials this will become the central "hub" VNet for our complex architecture.

- **addressPrefixes**

  Next, we specify the address space for our virtual network using the addressPrefixes property. The address space is specified using CIDR notation. In our example settings file, we've specified the address space to be 10.0.0.0/16. This means Azure Resource Manager allocates 65536 IP addresses beginning at 10.0.0.0 and ending at 10.0.255.255.

  Notice that the field for specifying the virtual network address space is an array. The reason for this is because we can specify multiple address ranges. For example, in addition to 10.0.0.0/16 we could have also specified 11.0.0.0/16 to specify everything between 11.0.0.0 and 11.0.255.255 as well:

  "addressPrefixes": [

  "10.0.0.0/16",

  "11.0.0.0/16"

  ]

- **subnets**

Now that we have specified the address space for our virtual network, we can begin to create named network segments known as subnets. Subnets are used to manage security, routing, and user access for each subnet independently of the entire VNet. Subnets are also used to segment VMs into back-end pools for load balancers and application gateways.

As you can see from our settings file, we've specified a single subnet named firewall with an address space of 10.0.1.0/24. Note that the subnets property is also an array - we can specify up to 1,000 subnets for each VNet.

Settings File

To deploy this virtual network, we will need to create a settings file for the azbb command line tool. An empty settings file looks like this:

```
{

"$schema": "https://raw.githubusercontent.com/mspnp/template-building-blocks/master/schemas/buildingBlocks.json",

"contentVersion": "1.0.0.0",

"parameters" : {

"buildingBlocks": {

"value": [

{}

]

}

}

}
```

In the example above, the **buildingBlocks** array is empty. For our deployment, we will add the simple Virtual Network building block that we discussed earlier in this topic. The settings file for our deployment would look like this:

```
{

"$schema": "https://raw.githubusercontent.com/mspnp/template-building-blocks/master/schemas/buildingBlocks.json",

"contentVersion": "1.0.0.0",

"parameters": {

"buildingBlocks": {
```

```
"value": [
{
"type": "VirtualNetwork",
"settings": [
{
"name": "msft-hub-vnet",
"addressPrefixes": [
"10.0.0.0/16"
],
"subnets": [
{
"name": "firewall",
"addressPrefix": "10.0.1.0/24"
}
]
}
]
}
]
}
}
```


Deployment

To deploy your settings file, you will need to ensure that you are logged in to the Azure CLI. Then, you'll need a few things before you can deploy the VNet using the settings file:

- You need your Azure subscription ID. You can find your subscription ID using the Azure CLI

command **az account list**, or, by going to the Azure Portal and opening the subscriptions blade.

- You'll need to consider the resource group to which the VNet will be deployed. You can deploy to either an existing or new resource group. The Azure Building Blocks command line tool determines if the resource group name you pass with the **-g** option exists or not. If the resource group exists, the command line tool deploys the VNet to the existing resource group. If it doesn't exist, the command line tool creates the resource group for you and then deploys the VNet to the new resource group.
- You'll also need to consider the Azure region where the VNet will be deployed.

Once you have your settings file and the information listed above, you can create a deployment using the following command:

azbb -g <new or existing resource group> -s <subscription ID> -l <region> -p <path to your settings file> --deploy

The command line tool will parse your settings file and deploy it to Azure using Azure Resource Manager. To verify that the VNet was deployed, visit the Azure Portal, click on **Resource Groups** in the left-hand pane to open the **Resource Groups** blade, then click on the name of the resource group you specified above. The blade for that resource group will open, and you should see the **msft-hub-vnet** in the list of resources.

Lab: Getting Started with Azure Resource Manager

Scenario

As part of your onboarding as a Fabrikam consultant, your team has asked you to become familiar with the Azure Resource Manager features and to deploy your first ARM templates.

Objectives

- Create a resource group.
- Deploy an ARM template to a resource group.
- View historical deployment metadata for a resource group.
- Delete a resource group and the related resources.

Lab setup

Estimated Time: 90 minutes

Virtual machine: **20535A-SEA-ARCH**

User name: **Admin**

Password: **Pa55w.rd**

The lab steps for this course change frequently due to updates to Microsoft Azure. Microsoft Learning updates the lab steps frequently, so they are not available in this manual. Your instructor will provide you with the lab documentation.

Exercise 1: Create Resource Groups

Exercise 2: Deploy an Empty Template

Exercise 3: Deploy a Simple Template

Exercise 4: Cleanup Subscription

Review Question(s)

Module review and takeaways

Review Question(s)