Hands-On Azure PowerShell Lab Workbook

60 Automation Scenarios for Managing, Automating, and Securing Azure with PowerShell (Az Module)

Preface

Why This Book Exists

The cloud doesn't wait for you to finish reading documentation. It moves fast, it changes constantly, and it rewards those who can *do*—not just those who can *describe*. That conviction is the beating heart of this book.

Hands-On Azure PowerShell Lab Workbook was born from a simple observation: too many Azure professionals understand concepts in theory but freeze when they open a terminal. They can explain what a Network Security Group does but can't script one from scratch. They know backups matter but have never automated a recovery vault with code. This book exists to close that gap—permanently—through **60 structured, hands-on labs** that put your fingers on the keyboard and keep them there.

Who This Book Is For

This workbook is designed for **Azure administrators, DevOps engineers, cloud architects, and IT professionals** who want to move beyond portal clicking and into the world of PowerShell-driven automation using the **Az module**. Whether you're preparing for a certification, transitioning into a cloud role, or looking to sharpen your production scripting skills, the hands-on approach of this book meets you where you are and pushes you forward.

You don't need to be a PowerShell expert to begin. You *will* be a far more capable one by the end.

What Makes This Book Different

This is not a reference manual. It is not a theory textbook. It is a **hands-on lab workbook**—every chapter is built around practical scenarios you will encounter in real Azure environments. Each lab presents a clear objective, walks you through the automation step by step, and challenges you to extend what you've built. The learning happens *in the doing*.

The 60 labs span the full spectrum of Azure administration:

- **Chapters 1–3** establish your hands-on lab environment, introduce governance fundamentals like resource groups, tags, and policies, and dive into identity automation with Azure AD.
- **Chapters 4–6** tackle core infrastructure—networking, virtual machine provisioning, and storage automation—through progressively complex hands-on exercises.
- **Chapters 7–9** shift focus to operational excellence: security hardening, observability, and backup automation, all scripted and repeatable.
- **Chapters 10–12** elevate your craft. You'll learn to write professional-grade automation scripts, implement scaling operations, and execute **real-world end-to-end projects** that mirror production deployments.
- **Appendices A–E** serve as your ongoing toolkit: cheat sheets, naming standards, script templates, a troubleshooting guide, and a **60-lab progress tracker** so you can measure every hands-on milestone.

The Philosophy Behind the Labs

Every lab in this book follows a principle I hold deeply: **you learn cloud automation by automating the cloud.** Passive reading builds familiarity; hands-on practice builds competence. Each scenario is designed to be *safe enough to experiment with* and *realistic enough to matter*. You will make mistakes in these labs—and that is by design. The troubleshooting guide in Appendix D exists because debugging is not a failure; it is the most valuable hands-on learning you'll do.

How to Use This Book

Start with Chapter 1 to configure your lab environment safely. Then work through the labs sequentially or jump to the chapters most relevant to your immediate needs. Use the progress tracker in **Appendix E** to hold yourself accountable. Most importantly: **type every command yourself.** Copy-pasting teaches your clipboard. Typing teaches your brain.

The terminal is open. The labs are waiting.

Let's get hands-on.

Laszlo Bocso (MCT)

Table of Contents

Chapter 1: Lab Environment and Safety

Labs 1–3

Before you begin automating, managing, and securing Azure resources through PowerShell, you must establish a solid foundation. This chapter walks you through every step of preparing your environment, understanding the tools at your disposal, and adopting the safety mindset that separates a competent Azure administrator from a reckless one.

Understanding the Lab Environment Architecture

Component	Purpose	Why It Matters
Azure Subscription	Billing and access boundary	Every command targets a specific subscription. A dedicated lab subscription prevents accidental production changes.
Resource Groups	Logical containers for resources	Labs use dedicated resource groups for easy create, manage, and teardown.
Azure PowerShell Az Module	Official module for managing Azure	Your primary tool throughout all 60 labs.

PowerShell 7.x	Cross-platform PowerShell	Consistent behavior across Windows, macOS, and Linux.
Azure Cloud Shell	Browser-based shell	Backup environment with pre-installed tools.
Visual Studio Code	Code editor with PowerShell extension	IntelliSense, debugging, integrated terminal.
Cost Management and Budgets	Spending controls	Prevents unexpected charges during labs.
Tags and Naming Conventions	Metadata and standardized names	Every lab resource is identifiable and cleanable.

Lab 1: Installing and Configuring the Az PowerShell Module

Objective: Install PowerShell 7.x and the Az module, configure execution policy, and verify the complete toolchain.

Check your current PowerShell version:

```
$PSVersionTable.PSVersion
```

If the Major version is 5 or lower, install PowerShell 7. On Windows:

```
winget install --id Microsoft.PowerShell --source winget
```

On macOS:

```
brew install powershell/tap/powershell
```

On Linux (Ubuntu):

```
sudo apt-get update
sudo apt-get install -y powershell
```

Launch PowerShell 7 and verify:

```
pwsh
$PSVersionTable.PSVersion
```

You should see output similar to:

```
Major  Minor  Patch  PreReleaseLabel BuildLabel
-----  -----  -----  --------------- ----------
7      4      6
```

Set the execution policy if needed:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope
CurrentUser
```

Install the Az module:

```
Install-Module -Name Az -Repository PSGallery -Force
-AllowClobber -Scope CurrentUser
```

Parameter	Explanation
`-Name Az`	The umbrella module including all Azure sub-modules
`-Repository PSGallery`	Official public repository
`-Force`	Overwrites existing installation without prompting
`-AllowClobber`	Permits installation even if cmdlet names conflict with other modules
`-Scope CurrentUser`	Does not require administrator privileges

Verify the installation:

```
Get-InstalledModule -Name Az
Get-Module -Name Az.* -ListAvailable | Select-Object Name,
Version | Sort-Object Name
```

Set up Visual Studio Code:

```
code --install-extension ms-vscode.powershell
```

Create your workspace:

```
New-Item -Path "$HOME/AzurePowerShellLabs" -ItemType Directory
-Force
New-Item -Path "$HOME/AzurePowerShellLabs/Chapter01" -ItemType
Directory -Force
```

Lab 2: Authenticating to Azure and Managing Contexts

Objective: Authenticate to Azure, understand contexts, switch subscriptions, and explore authentication methods.

Initiate interactive login:

```
Connect-AzAccount
```

Confirm your context:

```
Get-AzContext
```

List all available subscriptions:

```
Get-AzSubscription | Format-Table Name, Id, State
```

Switch to your lab subscription:

```
Set-AzContext -SubscriptionName "Your-Lab-Subscription-Name"
```

Or by ID:

```
Set-AzContext -SubscriptionId "xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx"
```

Authentication Method	Use Case	Command
Interactive Login	Daily lab work	`Connect-AzAccount`
Service Principal	Automated scripts and CI/CD	`Connect-AzAccount -ServicePrincipal -Credential $cred -Tenant $tenantId`
Managed Identity	Scripts inside Azure VMs	`Connect-AzAccount -Identity`
Device Code Flow	No browser available	`Connect-AzAccount -UseDeviceAuthentication`

Lab 3: Setting Up the Lab Environment with Cost Controls and Safety

Objective: Create a dedicated lab resource group, configure budget alerts, implement safety practices, and build a verification script.

Create your primary lab resource group:

```
New-AzResourceGroup -Name "rg-lab-powershell-001" -Location
"eastus" -Tag @{
    Environment = "Lab"
    Project     = "AzurePowerShellWorkbook"
    Owner       = "YourName"
    CreatedBy   = "PowerShell"
}
```

Verify creation:

```
Get-AzResourceGroup -Name "rg-lab-powershell-001" | Format-List
```

Check current spending:

```
Get-AzConsumptionUsageDetail -StartDate (Get-Date).AddDays(-7)
-EndDate (Get-Date) |
    Select-Object InstanceName, ConsumedService, PretaxCost |
    Sort-Object PretaxCost -Descending |
    Format-Table -AutoSize
```

Create a budget:

```
$startDate = Get-Date -Day 1 -Hour 0 -Minute 0 -Second 0
$endDate = $startDate.AddYears(1)

New-AzConsumptionBudget -Name "LabBudget" `
    -Amount 50 `
    -Category "Cost" `
    -TimeGrain "Monthly" `
    -StartDate $startDate `
    -EndDate $endDate `
    -ContactEmail @("your-email@example.com") `
    -NotificationKey "Alert1" `
    -NotificationEnabled `
    -NotificationThreshold 50
```

Threshold	Action
50%	Review running resources
80%	Delete resources from completed labs
100%	Stop all lab work, audit every resource

Practice safety commands:

```
# Preview destructive commands with -WhatIf
Remove-AzResourceGroup -Name "rg-lab-powershell-001" -WhatIf

# Require confirmation with -Confirm
```

```
Remove-AzVM -ResourceGroupName "rg-lab-powershell-001" -Name "vm-
test-001" -Confirm

# Lock resources against accidental deletion
New-AzResourceLock -LockLevel CanNotDelete `
    -LockName "ProtectLab" `
    -ResourceGroupName "rg-lab-powershell-001" `
    -LockNotes "Prevent accidental deletion during active labs"

# Remove lock when ready to clean up
$lock = Get-AzResourceLock -ResourceGroupName "rg-lab-
powershell-001"
Remove-AzResourceLock -LockId $lock.LockId -Force

# Audit resources and clean up
Get-AzResource -ResourceGroupName "rg-lab-powershell-001" |
Format-Table Name, ResourceType, Location
Remove-AzResourceGroup -Name "rg-lab-powershell-001" -Force
-AsJob
```

Safety Command	Purpose	When to Use
`-WhatIf`	Simulates without changes	Before any destructive operation
`-Confirm`	Prompts for confirmation	When uncertain about target resources
`New-AzResourceLock`	Prevents deletion	On resources that must not be removed
`Get-AzResource`	Lists all resources	Beginning and end of every lab
`-AsJob`	Runs in background	During long operations like deletion

Build and run the verification script:

```
Write-Host "=== Lab Environment Verification ==="
-ForegroundColor Cyan
```

```
Write-Host "`n[1] PowerShell Version:" -ForegroundColor Yellow
$PSVersionTable.PSVersion | Format-Table

Write-Host "[2] Az Module Version:" -ForegroundColor Yellow
Get-InstalledModule -Name Az | Select-Object Name, Version |
Format-Table

Write-Host "[3] Azure Connection:" -ForegroundColor Yellow
$context = Get-AzContext
if ($context) {
    Write-Host "  Connected to: $($context.Subscription.Name)"
-ForegroundColor Green
    Write-Host "  Subscription ID: $($context.Subscription.Id)"
-ForegroundColor Green
    Write-Host "  Tenant ID: $($context.Tenant.Id)"
-ForegroundColor Green
} else {
    Write-Host "  Not connected. Run Connect-AzAccount."
-ForegroundColor Red
}

Write-Host "`n[4] Lab Resource Group:" -ForegroundColor Yellow
$rg = Get-AzResourceGroup -Name "rg-lab-powershell-001"
-ErrorAction SilentlyContinue
if ($rg) {
    Write-Host "  Resource Group exists in $($rg.Location)"
-ForegroundColor Green
    Write-Host "  Tags: $($rg.Tags | ConvertTo-Json -Compress)"
-ForegroundColor Green
} else {
    Write-Host "  Resource group not found. Create it before
proceeding." -ForegroundColor Red
}

Write-Host "`n[5] Execution Policy:" -ForegroundColor Yellow
Get-ExecutionPolicy -List | Format-Table

Write-Host "`n=== Verification Complete ===" -ForegroundColor
Cyan
```

Save as `Verify-LabEnvironment.ps1` and run it. Every item should show green.

Chapter 2: Resource Groups, Tags, and Policy Mindset

Labs 4-9

Every well-architected Azure environment begins with a solid organizational foundation. This chapter takes you deep into the hands-on practice of working with Resource Groups, Tags, and Azure Policy through PowerShell.

Cmdlet	Purpose	Common Parameters
`New-AzResourceGroup`	Creates a new Resource Group	`-Name`, `-Location`, `-Tag`
`Get-AzResourceGroup`	Retrieves Resource Groups	`-Name`, `-Location`, `-Tag`
`Set-AzResourceGroup`	Updates properties	`-Name`, `-Tag`
`Remove-AzResource-Group`	Deletes a Resource Group and all contents	`-Name`, `-Force`, `-AsJob`
`Get-AzResource`	Lists all resources in a scope	`-ResourceGroupName`, `-ResourceType`, `-Tag`
`New-AzTag`	Creates or updates tags	`-ResourceId`, `-Tag`
`Get-AzTag`	Retrieves tag information	`-Name`, `-ResourceId`
`Remove-AzTag`	Removes tags	`-ResourceId`, `-Tag`

Lab 4: Creating and Managing Resource Groups

Objective: Create multiple resource groups following naming conventions, query and filter them with PowerShell.

```
New-AzResourceGroup -Name "rg-dev-project-alpha" -Location
"eastus"
New-AzResourceGroup -Name "rg-staging-project-alpha" -Location
"eastus"
New-AzResourceGroup -Name "rg-prod-project-alpha" -Location
"eastus"
```

Verify:

```
Get-AzResourceGroup | Where-Object { $_.ResourceGroupName -like
"*project-alpha*" } |
    Format-Table ResourceGroupName, Location, ProvisioningState
```

Lab 5: Working with Tags at Scale

Objective: Apply comprehensive tag sets to resource groups and query by tag.

```
$tags = @{
    "Environment" = "Development"
    "Project"     = "Alpha"
    "Owner"       = "CloudOps Team"
    "CostCenter"  = "CC-4200"
    "CreatedBy"   = "PowerShell Automation"
}

Set-AzResourceGroup -Name "rg-dev-project-alpha" -Tag $tags

$stagingTags = @{
    "Environment" = "Staging"
    "Project"     = "Alpha"
```

```
    "Owner"       = "CloudOps Team"
    "CostCenter"  = "CC-4200"
    "CreatedBy"   = "PowerShell Automation"
}

$prodTags = @{
    "Environment" = "Production"
    "Project"     = "Alpha"
    "Owner"       = "CloudOps Team"
    "CostCenter"  = "CC-4200"
    "CreatedBy"   = "PowerShell Automation"
}

Set-AzResourceGroup -Name "rg-staging-project-alpha" -Tag
$stagingTags
Set-AzResourceGroup -Name "rg-prod-project-alpha" -Tag $prodTags
```

Verify and query:

```
(Get-AzResourceGroup -Name "rg-dev-project-alpha").Tags

Get-AzResourceGroup -Tag @{ "Project" = "Alpha" } |
    Format-Table ResourceGroupName, Location
```

Note: Tags are not inherited by default. Tagging a Resource Group does not automatically tag its child resources. Use Azure Policy to enforce tag inheritance.

Lab 6: Bulk Tagging with PowerShell Loops

Objective: Write a script that adds a mandatory tag to every resource group missing it, while preserving existing tags.

```
$allResourceGroups = Get-AzResourceGroup

foreach ($rg in $allResourceGroups) {
    $currentTags = $rg.Tags
```

```
    if ($null -eq $currentTags) {
        $currentTags = @{}
    }

    if (-not $currentTags.ContainsKey("ManagedBy")) {
        $currentTags["ManagedBy"] = "CloudOps"
        Set-AzResourceGroup -Name $rg.ResourceGroupName -Tag
$currentTags
        Write-Output "Added 'ManagedBy' tag to $
($rg.ResourceGroupName)"
    } else {
        Write-Output "'ManagedBy' tag already exists on $
($rg.ResourceGroupName)"
    }
}
```

Note: Always retrieve existing tags before modifying. `Set-AzResourceGroup -Tag` replaces the entire tag collection. Forgetting this is the number one cause of accidentally deleted tags.

Lab 7: Generating a Tag Compliance Report

Objective: Build a compliance report that identifies resource groups missing required tags and export it to CSV.

```
$requiredTags = @("Environment", "Project", "Owner",
"CostCenter")
$report = @()

$allResourceGroups = Get-AzResourceGroup

foreach ($rg in $allResourceGroups) {
    $missingTags = @()
```

```
    foreach ($requiredTag in $requiredTags) {
        if ($null -eq $rg.Tags -or -not
$rg.Tags.ContainsKey($requiredTag)) {
            $missingTags += $requiredTag
        }
    }

    $report += [PSCustomObject]@{
        ResourceGroupName = $rg.ResourceGroupName
        Location          = $rg.Location
        TotalTags         = if ($null -ne $rg.Tags)
{ $rg.Tags.Count } else { 0 }
        MissingTags       = if ($missingTags.Count -gt 0)
{ $missingTags -join ", " } else { "None" }
        Compliant         = if ($missingTags.Count -eq 0) { "Yes"
} else { "No" }
    }
}

$report | Format-Table -AutoSize
$report | Export-Csv -Path ".\TagComplianceReport.csv"
-NoTypeInformation
```

Lab 8: Assigning and Managing Azure Policy

Objective: Explore built-in policy definitions, assign a tag enforcement policy using splatting, and test enforcement.

Explore built-in tag policies:

```
Get-AzPolicyDefinition -BuiltIn |
    Where-Object { $_.Properties.DisplayName -like "*tag*" } |
    Select-Object -Property
@{N='DisplayName';E={$_.Properties.DisplayName}},

@{N='Description';E={$_.Properties.Description}} |
```

```
    Format-Table -Wrap
```

Assign a policy requiring the CostCenter tag:

```
$policyDefinition = Get-AzPolicyDefinition -BuiltIn |
    Where-Object { $_.Properties.DisplayName -eq "Require a tag
on resource groups" }

$subscription = Get-AzSubscription | Select-Object -First 1

$assignmentParams = @{
    Name                 = "require-costcenter-tag-rg"
    DisplayName          = "Require CostCenter tag on Resource
Groups"
    Description          = "This policy requires all Resource
Groups to have a CostCenter tag."
    PolicyDefinition     = $policyDefinition
    Scope                = "/subscriptions/$($subscription.Id)"
    PolicyParameterObject = @{
        tagName = "CostCenter"
    }
}

New-AzPolicyAssignment @assignmentParams
```

Check compliance:

```
Get-AzPolicyState -SubscriptionId $subscription.Id |
    Where-Object { $_.ComplianceState -eq "NonCompliant" } |
    Select-Object ResourceId, PolicyAssignmentName,
ComplianceState |
    Format-Table -AutoSize
```

Lab 9: Creating Custom Policy Definitions and Cleanup

Objective: Create a custom audit policy, assign it, then clean up all resources and policy assignments.

Create the custom policy:

```
$policyRule = @'
{
    "if": {
        "allOf": [
            {
                "field": "type",
                "equals": "Microsoft.Resources/subscriptions/
resourceGroups"
            },
            {
                "field": "tags['Owner']",
                "exists": "false"
            }
        ]
    },
    "then": {
        "effect": "audit"
    }
}
'@

$customPolicyParams = @{
    Name        = "audit-missing-owner-tag"
    DisplayName = "Audit Resource Groups missing Owner tag"
    Description = "This policy audits any Resource Group that
does not have an Owner tag."
    Policy      = $policyRule
    Mode        = "All"
}

New-AzPolicyDefinition @customPolicyParams
```

```
$customPolicy = Get-AzPolicyDefinition -Name "audit-missing-
owner-tag"

New-AzPolicyAssignment -Name "audit-owner-tag-assignment" `
    -DisplayName "Audit missing Owner tag on Resource Groups" `
    -PolicyDefinition $customPolicy `
    -Scope "/subscriptions/$($subscription.Id)"
```

Cleanup:

```
$groupsToDelete = @("rg-dev-project-alpha", "rg-staging-project-
alpha", "rg-prod-project-alpha")

foreach ($groupName in $groupsToDelete) {
    Remove-AzResourceGroup -Name $groupName -Force -AsJob
    Write-Output "Deletion initiated for $groupName"
}

Get-Job | Format-Table Name, State, HasMoreData

Remove-AzPolicyAssignment -Name "require-costcenter-tag-rg"
Remove-AzPolicyAssignment -Name "audit-owner-tag-assignment"
Remove-AzPolicyDefinition -Name "audit-missing-owner-tag" -Force
```

Concept	Key Takeaway
Resource Groups	Every resource must belong to exactly one. Logical containers for lifecycle management.
Naming Conventions	Use prefixes like `rg-` and include environment and project identifiers.
Tags	Not inherited by child resources. Always retrieve before modifying.
Azure Policy	Enables mandatory enforcement. Effects: Deny, Audit, Modify.
Splatting	Use hashtables with `@` prefix for clean parameter passing.
Background Jobs	Use `-AsJob` for long-running operations.

Chapter 3: Azure Identity Automation

Labs 10-12

Identity management is the cornerstone of every secure cloud environment. This chapter covers bulk user provisioning, RBAC role assignments, and service principal management.

Lab 10: Bulk Creating Azure AD Users with PowerShell

Objective: Create a bulk user provisioning script that reads from a CSV and creates Azure AD users programmatically.

Install and connect to Microsoft Graph:

```
Install-Module -Name Microsoft.Graph -Scope CurrentUser -Force
Connect-MgGraph -Scopes "User.ReadWrite.All",
"Directory.ReadWrite.All"
```

Create `NewUsers.csv`:

```
DisplayName,UserPrincipalName,MailNickname,Department,JobTitle,Us
ageLocation
Sarah
Mitchell,sarah.mitchell@yourdomain.onmicrosoft.com,sarah.mitchell
,Engineering,Software Engineer,US
James
Rodriguez,james.rodriguez@yourdomain.onmicrosoft.com,james.rodrig
uez,Marketing,Marketing Analyst,US
```

```
Priya
Sharma,priya.sharma@yourdomain.onmicrosoft.com,priya.sharma,Finan
ce,Financial Analyst,US
David
Chen,david.chen@yourdomain.onmicrosoft.com,david.chen,Engineering
,DevOps Engineer,US
Emma
Thompson,emma.thompson@yourdomain.onmicrosoft.com,emma.thompson,H
uman Resources,HR Specialist,US
```

Field	Description	Required
DisplayName	Full name in directory	Yes
UserPrincipalName	Sign-in name in email format	Yes
MailNickname	Mail alias	Yes
Department	Organizational department	No
JobTitle	Professional title	No
UsageLocation	Two-letter ISO country code	Yes for licensing

Bulk provisioning script:

```
$csvPath = ".\NewUsers.csv"
$users = Import-Csv -Path $csvPath

$passwordProfile = @{
    Password                      = "TempP@ssw0rd2024!"
    ForceChangePasswordNextSignIn = $true
}

$successCount = 0
$failCount = 0
$results = @()

foreach ($user in $users) {
    try {
        $newUser = New-MgUser -DisplayName $user.DisplayName `
            -UserPrincipalName $user.UserPrincipalName `
            -MailNickname $user.MailNickname `
```

```
            -Department $user.Department `
            -JobTitle $user.JobTitle `
            -UsageLocation $user.UsageLocation `
            -PasswordProfile $passwordProfile `
            -AccountEnabled:$true

        Write-Host "Successfully created user: $
($user.DisplayName)" -ForegroundColor Green
        $successCount++

        $results += [PSCustomObject]@{
            UserPrincipalName = $user.UserPrincipalName
            DisplayName       = $user.DisplayName
            Status            = "Created"
            ObjectId          = $newUser.Id
            Timestamp         = Get-Date -Format "yyyy-MM-dd
HH:mm:ss"
        }
    }
    catch {
        Write-Host "Failed to create user: $($user.DisplayName) -
$($_.Exception.Message)" -ForegroundColor Red
        $failCount++

        $results += [PSCustomObject]@{
            UserPrincipalName = $user.UserPrincipalName
            DisplayName       = $user.DisplayName
            Status            = "Failed"
            ObjectId          = "N/A"
            Timestamp         = Get-Date -Format "yyyy-MM-dd
HH:mm:ss"
        }
    }
}

Write-Host "`nProvisioning Summary:" -ForegroundColor Cyan
Write-Host "Total Users Processed: $($users.Count)"
Write-Host "Successfully Created: $successCount" -ForegroundColor
Green
Write-Host "Failed: $failCount" -ForegroundColor Red
```

```
$results | Export-Csv -Path ".\UserCreationReport.csv"
-NoTypeInformation
```

Cleanup:

```
$users = Import-Csv -Path ".\NewUsers.csv"
foreach ($user in $users) {
    Remove-MgUser -UserId $user.UserPrincipalName -Confirm:$false
    Write-Host "Removed user: $($user.DisplayName)"
}
```

Lab 11: Assigning Azure RBAC Roles Programmatically

Objective: Assign granular RBAC permissions across different scopes, audit existing assignments, and generate a report.

Concept	Description	Cmdlet
Security Principal	Identity requesting access	Get-AzADUser, Get-AzADGroup
Role Definition	Collection of permissions	Get-AzRoleDefinition
Scope	Boundary for access	String path format

Explore built-in roles:

```
Get-AzRoleDefinition | Where-Object { $_.IsCustom -eq $false } |
    Select-Object Name, Description |
    Sort-Object Name |
    Format-Table -AutoSize -Wrap

Get-AzRoleDefinition -Name "Contributor" | Format-List Name,
Description, Actions, NotActions
```