**Introduction**

This module is about expressing your infrastructure requirements as code. By adopting this practice, you can automatically provision the infrastructure that you need for your application or service.

In the [Deploy applications with Azure DevOps](https://docs.microsoft.com/en-us/learn/paths/deploy-applications-with-azure-devops) learning path, you built a release pipeline in Azure Pipelines. You deployed a web application to Azure App Service instances that you brought up manually through the Azure portal and through the Azure CLI.

Keeping up with changes in your infrastructure is a challenge. New resources need to be built. Old resources need maintenance. Environments need to be consistent. *Infrastructure as code* enables you to describe, through code, the infrastructure that you need for your application.

The Tailspin team is making great progress. But how can they further automate their processes to allow them to scale and release even faster?

In this module, you continue your journey with the Tailspin team as they add automatic provisioning to their CI/CD pipeline for the *Space Game* website.

**Learning objectives**

In this module, you'll:

* Learn what we mean by *infrastructure as code* and why it's a best practice in DevOps.
* Apply a Terraform plan that provisions everything you need to run your web application.
* Add a job to your Azure Pipelines configuration that provisions your infrastructure when the pipeline runs.

**Prerequisites**

The modules in this learning path form a progression.

To follow the progression from the beginning, be sure to first complete these learning paths:

* [Evolve your DevOps practices](https://docs.microsoft.com/en-us/learn/paths/evolve-your-devops-practices)
* [Build applications with Azure DevOps](https://docs.microsoft.com/en-us/learn/paths/build-applications-with-azure-devops)
* [Deploy applications with Azure DevOps](https://docs.microsoft.com/en-us/learn/paths/deploy-applications-with-azure-devops)

If you want to start with this learning path, you need to set up a development environment on your Windows, macOS, or Linux system. You need:

* An [Azure DevOps organization](https://dev.azure.com/)
* An [Azure subscription](https://azure.microsoft.com/free/)
* A [GitHub](https://github.com/join" \t "az-portal) account
* [Visual Studio Code](https://code.visualstudio.com/)
* [.NET Core 3.1 SDK](https://dotnet.microsoft.com/download/dotnet-core/3.1)
* [Git](https://git-scm.com/downloads)

You can get started with Azure and Azure DevOps for free. You don't need an Azure subscription to work with Azure DevOps, but here you'll use Azure DevOps to deploy to resources that exist in your Azure subscription.

This environment lets you complete the exercises in this and future modules. You can also use it to apply your new skills to your own projects.

**Note**

Keep in mind that you can use Azure DevOps to build and deploy almost any kind of application written in any language. In this module, you'll be working with a .NET Core application written in C#.

You don't need to be an expert in .NET or C# to complete this module. You can apply the patterns you learn here to your own projects that use your favorite programming languages and frameworks.

# What is infrastructure as code?

* 12 minutes

Infrastructure as code enables you to describe, through code, the infrastructure that you need for your application.

Infrastructure as code enables you to maintain both your application code and everything you need to deploy your application in a central code repository. In that repository, you can build, test, and deploy as a unit.

Tim has called the team together. He has a problem, and he knows that the best way to solve it is to discuss it with everyone.

## Weekly meeting

**Andy:** Tim, you had a topic to discuss. What's up?

**Tim:** I have a few things on my mind. We're getting ready to launch Space Game to our preview users, which means that the website needs to be ready as well. Irwin wants us to expand our operations so that the website is scaled geographically. Geographic scaling means better response times for users and makes us more competitive.

Irwin added a few more requirements:

* The game development team has a number of back-end systems that perform in-game matchmaking, collect leaderboard data, and so on. These systems are virtual machines that run on Azure. We need the ability to scale out these systems to accommodate any number of active users playing the game. Each system needs to come up quickly and be configured identically.
* Our security officer is running an audit on our systems. They require that we adhere not only to application security requirements, but also to infrastructure security requirements.
* Currently, we deploy one Azure SQL database that connects to each of our **dev**, **test**, and **staging** environments. We need each environment to have its own database server so that we can test and roll out changes incrementally. A change made in **dev** might break other environments, such as **test** and **staging**, until that change reaches those stages.

All this means that there's a lot more infrastructure to manage. I'm just one person, and we don't plan on hiring others to help. I honestly don't know what to do. I know that some of these issues, like our back-end systems, aren't your problem. But I'd appreciate the help.

**Andy:** OK. From what I'm hearing, you need a way to:

* Automatically provision Azure App Service environments and database servers that run the Space Game website.
* Configure multiple systems in the exact same way, and keep them configured that way.
* Prove that all the production sites are compliant with various regulations.

From these requirements, I see one common theme: the need to automate your deployments. Is that fair?

**Tim:** Sounds right.

**Andy:** OK, Tim. Why don't you and I put our heads together after the meeting and then we'll let the rest of the team know what we come up with?

**Mara:** It sounds like you two are on this. But before we end the meeting, let's capture these tasks as part of our upcoming sprint plan.

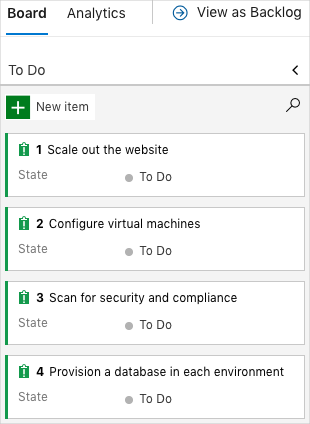
## The sprint plan

The team does more brainstorming. As they do every few weeks, they identify the tasks that they need to perform in the next sprint. For the issues Tim raised, they identify these four tasks:

* Scale out the website
* Configure virtual machines
* Scan for security and compliance
* Provision a database server for each environment

**Mara:** If that seems right, I'll add these tasks to Azure Boards.

The team agrees. Mara sets up a sprint iteration and adds these tasks:



Each of these tasks relates to what you'll do in this and the following modules. You don't need to add these tasks to Azure Boards now. You'll run a template that populates Azure Boards with these tasks, just as you did in previous modules.

## Express infrastructure as code to automate your deployments

Tim and Andy are sitting at the coffee shop. Andy, who has had three espressos, talks while scribbling on napkins.

**Andy:** The answer to all your problems is infrastructure as code.

**Tim:** Well, that was simple. You'll have to tell me more!

**Andy:** OK. When something goes wrong with a server, what do you do?

**Tim:** I SSH in and fix it. I've also got a lot of Bash scripts that I've written over the years, although they're not exactly organized.

**Andy:** Right, I believe I've seen many of them taped to the walls in your office.

Tim shrugs.

**Andy:** Infrastructure as code, which is sometimes called IaC, uses automation to manage infrastructure. For example, instead of configuring your servers one at a time, you use a tool to configure all of them at once. You express the infrastructure you need for your application as code, such as a configuration file or a script.

Here's a short video where Abel Wang, Cloud Advocate at Microsoft, explains the concepts of infrastructure as code.

When you need to change something, like a firewall setting, you update the configuration file and run it one time to update all of the servers. You can see why this makes scaling out a lot easier.

For the Space Game website, we started by bringing up App Service environments manually through the Azure portal. Then we moved to using the Azure CLI in Azure Cloud Shell to accomplish the same thing, but in a more consistent and predicable way. To take the next step, which automates the process even more, we can run the provisioning process through Azure Pipelines.

As a bonus, your infrastructure and configuration code is stored in version control, along with the application code. You'll have lots more wall space.

As you move to infrastructure as code, you're going to see a lot of improvements.

Andy scribbles them down on a napkin:

* Consistent configurations
* Improved scalability
* Faster deployments
* Less documentation because the scripts replace it
* Better traceability

A cup of coffee and a napkin that lists the benefits of infrastructure as code

Infrastructure as code is a best practice in DevOps because it helps bring teams together. The development team gets a better understanding of where their code is being deployed to and run. Operations teams get involved in the process much earlier and can use infrastructure choices to help developers make the right design decisions.

**Andy:** You have a lot on your plate. I think the best way to get started is for us to find a way to scale out the website. You can then apply what you've learned to your other problems.

**Tim:** Sounds great. But I have one question first. You mentioned that we can run the provisioning process through Azure Pipelines. Won't the process re-create our existing environments? I don't think we would want that.

**Andy:** The key here is idempotency.

## What is idempotency?

Idempotency is a term that's used in both mathematics and computer science.

An idempotent operation is one that provides the same result each time you apply it.

Consider the Azure CLI as an example. The following az group create command creates an Azure resource group that's named "my-rg" in the "westus" region.

**Azure CLI**

az group create --name my-rg --location westus

The command produces output similar to this:

**Output**Copy

{

"id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/my-rg",

"location": "westus",

"managedBy": null,

"name": "my-rg",

"properties": {

"provisioningState": "Succeeded"

},

"tags": null,

"type": "Microsoft.Resources/resourceGroups"

}

If you run this command a second time, you receive the exact same output. You don't receive an error or a duplicate resource group. That's because this Azure CLI command was designed to be idempotent.

### What if an operation isn't idempotent?

Of course, not all operations are idempotent. For example, say you have a Bash script that creates a directory to hold service logs:

**Bash**

#!/bin/bash

mkdir /var/log/my-service

This script creates the /var/log/my-service directory the first time you run it.

But when you run the script for the second time, you get an error message:

**Output**

mkdir: /var/log/my-service: File exists

Because the mkdir command failed, the script produces a nonzero status code. This status code can indicate a failure to another process.

When an operation isn't idempotent, you can use a condition to make it behave like one. Here's an example that uses **if..then** syntax to create the directory, but only if that directory does not exist:

**Bash**

#!/bin/bash

if [ ! -d "/var/log/my-service" ]

then

mkdir /var/log/my-service

fi

## What is imperative vs. declarative code?

Generally speaking, imperative code defines both what the program should accomplish and how to achieve the result.

In contrast, declarative code defines what the program should accomplish but does not define how to achieve the result.

Think about how you write code in languages like Python, C#, or Java. Each of these languages is imperative. You use variables, conditional statements, and loops to control the state of your program and express the steps you need.

Compare these languages to HTML, which is declarative. HTML describes what elements appear on the page, but it doesn't describe how to display them. The "how" is the web browser's responsibility.

When you're relating imperative and declarative code to infrastructure, consider a few examples. This example shows a Bash script that creates an Azure resource group, an App Service plan, and an App Service instance that uses that plan:

**Bash**

#!/bin/bash

# Create the resource group.

az group create \

--name my-rg \

--location westus

# Create the app service plan.

az appservice plan create \

--resource-group my-rg \

--name my-asp

# Create the app service.

az webapp create \

--resource-group my-rg \

--plan my-asp \

--name my-webapp

Here, you have full control over the order of each step and how to run each command. Compare this to the following Terraform plan, which accomplishes the same task. (You'll learn more about Terraform shortly.)

**Terraform**

resource "azurerm\_resource\_group" "my" {

name = "my-rg"

location = "westus"

}

resource "azurerm\_app\_service\_plan" "my" {

name = "my-asp"

location = "westus"

resource\_group\_name = "my-rg"

kind = "Linux"

reserved = true

sku {

tier = "Basic"

size = "B1"

}

}

resource "azurerm\_app\_service" "my" {

name = "my-appsvc"

location = "westus"

resource\_group\_name = "my-rg"

app\_service\_plan\_id = azurerm\_app\_service\_plan.my.id

}

This code is written in Hashicorp Configuration Language (HCL), a declarative language that supports Terraform. Here, you have less control over the order of the steps or how Terraform accomplishes each step. The advantage is that you need only to specify the infrastructure that you want. Terraform figures out the details.

### Which approach is better?

When you're considering an imperative versus a declarative approach to your infrastructure, don't focus on which approach is better. Instead, consider the tools that you can use. Choose the tool that matches your skills and best meets your needs.

Some prefer an imperative approach because it gives you fuller control over how your infrastructure is managed. But consider the Bash script that you saw earlier. Say you want to modify the resource group's location from "westus" to "northeurope."

**Bash**

#!/bin/bash

# Create the resource group.

az group create \

--name my-rg \

--location northeurope

# Create the app service plan.

az appservice plan create \

--resource-group my-rg \

--name my-asp

# Create the app service.

az webapp create \

--resource-group my-rg \

--plan my-asp \

--name my-webapp

When you run this Bash script, you get an error that says the resource group already exists in another location.

**Output**

Invalid resource group location 'northeurope'. The Resource group already exists in location 'westus'.

To fix this error, you might first delete the existing resource group and then run the Bash script a second time to create a new set of resources. Or you might run the az resource move command to move the existing resources within that resource group to the new location.

In the corresponding Terraform plan, Terraform creates a dependency graph for all the resources that you describe. The graph creates an overall description of what you're building. In doing so, Terraform can often adjust your resources to match your desired state.

**Terraform**Copy

resource "azurerm\_resource\_group" "my" {

name = "my-rg"

location = "northeurope"

}

resource "azurerm\_app\_service\_plan" "my" {

name = "my-asp"

location = "northeurope"

resource\_group\_name = "my-rg"

kind = "Linux"

reserved = true

sku {

tier = "Basic"

size = "B1"

}

}

resource "azurerm\_app\_service" "my" {

name = "my-appsvc"

location = "northeurope"

resource\_group\_name = "my-rg"

app\_service\_plan\_id = azurerm\_app\_service\_plan.my.id

}

In this case, Terraform must re-create the resource group because you can't reassign a resource group's location. Terraform must also re-create the App Service plan and the App Service instance because it understands that they depend on the resource group.

Terraform takes care of these details for you. All you needed to do was define your requirements.

## What tools can I use?

**Tim:** Infrastructure as code sounds like something that can answer many of my problems. But I'll need to try it out. What tools can you use to apply infrastructure code?

**Andy:** As you'd expect, there are many options. Some are from Microsoft and are designed for Azure, but there are other options too.

Andy writes down a few options on a napkin.

* Azure CLI
* Azure PowerShell
* Azure Resource Manager templates
* Ansible
* Terraform
* Azure SDK

### **Azure CLI**

You can use the Azure CLI to connect to Azure and run administrative commands on Azure resources. As with other command-line interfaces, you can run commands directly from a terminal, or you can add commands to a Bash script or a PowerShell script. The Azure CLI runs on Windows, macOS, or Linux. You can also access it from Cloud Shell.

Here's a basic example that uses the az group create command to create an Azure resource group named "my-rg" in the "westus" location:

**Azure CLI**

az group create \

--name my-rg \

--location westus

If you run this command multiple times, you get the same resource group back each time. So this command becomes an idempotent operation.

### Azure PowerShell

Azure PowerShell is a module that you add to Windows PowerShell or PowerShell Core. You can then connect to your Azure subscription and manage resources.

Azure PowerShell requires PowerShell to function. PowerShell provides services like the shell window and command parsing. Azure PowerShell adds the Azure-specific commands.

Here's a basic example that uses the New-AzResourceGroup cmdlet to create an Azure resource group that's named "my-rg" in the "West US" location:

**PowerShell**

New-AzResourceGroup `

-Name my-rg `

-Location "West US"

If you run this command multiple times, you get the same resource group back each time. So this command becomes an idempotent operation.

Azure PowerShell is also available two ways:

* Inside a browser via Cloud Shell
* Locally on Windows, macOS, or Linux

In both cases, you have two modes to choose from. You can use Azure PowerShell in interactive mode, in which you manually issue one command at a time. Or you can use it in scripting mode, where you run a script that consists of multiple commands.

The Azure CLI and Azure PowerShell are roughly equal in terms of what they can do. When you're deciding between the Azure CLI and Azure PowerShell, choose the one that best fits your team's skills. For example, if you use PowerShell to administer Windows, you'll quickly become comfortable with using Azure PowerShell. If you write Bash scripts to administer Linux, you might prefer the Azure CLI.

### Azure Resource Manager templates

Azure Resource Manager is the interface for managing and organizing cloud resources. Think of Resource Manager as a way to deploy cloud resources.

If you're familiar with Azure resource groups, you know that they enable you to treat sets of related resources as a single unit. Resource Manager is what organizes the resource groups that let you deploy, manage, and delete all of the resources together in a single action.

A Resource Manager template precisely defines all the Resource Manager resources in a deployment. You can deploy a Resource Manager template into a resource group as a single operation.

A Resource Manager template is a JSON file, making it a form of declarative automation. Here's a basic example that defines an Azure resource group named "my-rg" in the "westus" location:

**JSON**Copy

{

"$schema": "https://schema.management.azure.com/schemas/2018-05-01/subscriptionDeploymentTemplate.json#",

"contentVersion": "1.0.0.1",

"resources": [

{

"type": "Microsoft.Resources/resourceGroups",

"apiVersion": "2018-05-01",

"location": "westus",

"name": "my-rg",

"properties": {}

}

]

}

You can run this template from Azure PowerShell, the Azure CLI, or the Azure portal. Or you can run it from application code through a REST service or API. Here's an example that uses the Azure CLI command az deployment create to deploy the template:

**Azure CLI**

az deployment create --template-file test.json

If you run this command multiple times, you get the same resource group back each time. So this command becomes an idempotent operation.

**Note**

You might hear others refer to Resource Manager templates as "ARM templates." We prefer the full name "Azure Resource Manager templates" or "Resource Manager templates."

### Ansible

Ansible is an open-source product that's sponsored by Red Hat. It automates cloud provisioning, configuration management, and application deployments. You can use Ansible to provision Azure resources such as virtual machines, containers, networks, and even complete cloud infrastructures. You can also use Ansible to configure your Azure resources after they're provisioned.

In addition to Azure, Ansible supports other public clouds and private cloud frameworks.

You can run Ansible commands directly, or you can write playbooks to deploy more complex configurations. A playbook is a YAML file, making it a form of declarative automation. Here's a basic example that defines an Azure resource group named "my-rg" in the "westus" location:

**yml**

- name: Example configuration

hosts: localhost

connection: local

tasks:

- name: Create resource group

azure\_rm\_resourcegroup:

name: my-rg

location: westus

You use the ansible-playbook command to apply this configuration, like this:

**Bash**

ansible-playbook example.yml

If you run this command multiple times, you get the same resource group back each time. So this command becomes an idempotent operation.

### Terraform

Terraform, by HashiCorp, is an open-source tool that provisions and manages cloud infrastructure. Terraform's command-line interface provides you with a simple way to provision Azure resources such as virtual machines, containers, and networks.

Terraform generates an execution plan that describes what it will do to reach the desired state. You can review this execution plan and then run the plan to apply the configuration. As your configuration changes, Terraform determines what has changed and creates incremental execution plans to update only what's necessary.

In addition to Azure, Terraform supports other public clouds and private cloud frameworks.

You typically write Terraform code in HCL. But you can also express your Terraform configuration by using JSON. HCL and JSON are forms of declarative automation. Here's a basic example written in HCL that defines an Azure resource group named "my-rg" in the "westus" location:

**Terraform**

resource "azurerm\_resource\_group" "my" {

name = "my-rg"

location = "westus"

}

You use the terraform apply command to apply this configuration, like this:

**Bash**

terraform apply

If you run this command multiple times, you get the same resource group back each time. So this command becomes an idempotent operation.

### Azure SDK

The Azure SDK enables you to access and manage Azure resources from application code by using your favorite programming languages.

The Azure SDK is available for languages such as Java, Node, Python, Ruby, PHP, .NET, and Go. You can also use a REST API to access Azure resources from any programming language.

Here's an example that uses the Azure SDK for .NET to create an Azure resource group named "my-rg" in the "westus" location. This example is written in C#.

**C#**

IResourceGroup resourceGroup = azure.ResourceGroups

.Define("my-rg")

.WithRegion(Region.USWest)

.Create();

## Where can I run my infrastructure code?

You can run your infrastructure code from:

* A terminal window or PowerShell
* Cloud Shell
* A CI/CD pipeline, including Azure Pipelines

Writing infrastructure code can be an iterative process. As you test new ideas, you might need to bring up and tear down your configurations multiple times. For more complex configurations, you might define your infrastructure in smaller pieces, and then fit those pieces together to form your complete cloud infrastructure.

During the development phase, you typically run your infrastructure code directly from a terminal window or Cloud Shell. Visual Studio Code provides add-ins that help you write configuration code. You can run your configuration code from the integrated terminal in Visual Studio Code. Cloud Shell is also a popular choice because it comes with everything you need to run any of the options that we discussed here: Azure CLI, Azure PowerShell, Azure Resource Manager templates, Ansible, and Terraform.

After you have the working configuration that you need, you typically check in your infrastructure code to source control along with your application code. You can then version, build, test, and deploy your applications as a unit.

Because your infrastructure code is maintained along with your application code, you can run your infrastructure code in the pipeline. You might configure the pipeline to apply your infrastructure code after it builds the application, but before you deploy your application to a running environment.

In this module, you first apply a Terraform plan from Cloud Shell to see it working. Then you run it from Azure Pipelines to provision infrastructure for your web application.

## The decision

Andy sits back in his chair.

**Andy:** Tim, of these options, which tool feels like the right choice to start with?

**Tim:** Each option is appealing. I'm a Linux user, so running Azure CLI commands or Resource Manager templates through Bash scripts most closely matches my skills. But remember that we operate a number of services on-premises that power our game servers and payment systems. Until we migrate these systems to the cloud, we still need to provision and configure those systems. Terraform enables us to do that.

**Andy:** Great idea. You're always forward-looking. If you like Terraform, you can use it to provision systems in our datacenter as well. Then, as we migrate to the cloud, you can modify the infrastructure code to provision cloud resources because you already know the language and the infrastructure requirements.

**Andy:** Let's work together on this one. You can take the lead. Let me just grab one more coffee to go.