**Summary**

* 5 minutes

Nice work! In this module, you learned how infrastructure as code enables you to describe the infrastructure that you need for your application. You also learned the difference between declarative and imperative code, as well as a few tools that you can use to automate your deployments.

Terraform is just one tool for automating your deployments. Terraform operations are idempotent. That means you can run Terraform as many times as you want without affecting infrastructure resources that haven't changed.

You can run Terraform locally or from Azure Cloud Shell to experiment and incrementally build out your infrastructure configuration. After you have the configuration, you can integrate automated provisioning in the pipeline. Maintaining your infrastructure code along with your application code gives you everything that you need to build, test, and deploy your application as a unit.

**Challenge**

If you're looking for a challenge, see if you can modify the pipeline to provision infrastructure for all three deployment stages: *Dev*, *Test*, and *Staging*. See if you can modify it just like the release pipeline that you set up in the [Create a multi-stage pipeline with Azure Pipelines](https://docs.microsoft.com/en-us/learn/modules/create-multi-stage-pipeline) module.

One approach might be to modify your Terraform plan to provision all three App Service environments. You can add a stage that runs after the *Build* stage and that provisions all three App Service environments before you run your deployment stages. You can use the [for\_each](https://www.terraform.io/docs/configuration/resources.html#for_each-multiple-resource-instances-defined-by-a-map-or-set-of-strings&azure-portal=true) syntax to provision all three App Service environments from a single block of code.

Another approach might be to use the same pattern that you used in this module. Here, you can specify a Terraform variable that defines the App Service name for the corresponding environment: **dev**, **test**, or **staging**. You'll also need a separate state file to manage each deployment. For example, you might name the state file for the *Dev* stage *terraform-dev.tfstate* instead of just *terraform.tfstate*.

As a bonus, you can map each App Service instance to an App Service plan that more closely matches your infrastructure requirements. For example, you might use the B1 Basic plan for your **dev** and **test** environments. The the B1 Basic plan is intended for apps that have lower traffic requirements. Then, you might use the Standard or Premium plan for your **staging** environment. The Standard and Premium plans are for production workloads, and the **staging** environment more closely matches your production environment.

**Learn more**

In this module, you saw a few tools that you can use to automate your deployments. Here's more information about each option.

**Azure CLI**

* [Control Azure services with the CLI](https://docs.microsoft.com/en-us/learn/modules/control-azure-services-with-cli/)
* [Azure CLI](https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest)

**Azure PowerShell**

* [Automate Azure tasks using scripts with PowerShell](https://docs.microsoft.com/en-us/learn/modules/automate-azure-tasks-with-powershell/)
* [Azure PowerShell](https://docs.microsoft.com/en-us/powershell/azure/)

**Azure Resource Manager templates**

* [Build Azure Resource Manager templates](https://docs.microsoft.com/en-us/learn/modules/build-azure-vm-templates/)
* [Control and organize Azure resources with Azure Resource Manager](https://docs.microsoft.com/en-us/learn/modules/control-and-organize-with-azure-resource-manager/)
* [Azure Resource Manager](https://docs.microsoft.com/en-us/azure/azure-resource-manager/)

**Ansible**

* [Ansible](https://www.ansible.com/)
* [Ansible on Azure documentation](https://docs.microsoft.com/en-us/azure/ansible/)

**Terraform**

* [Terraform](https://www.terraform.io/)
* [Terraform on Azure documentation](https://docs.microsoft.com/en-us/azure/terraform/)

**Appendix: Inspecting and validating infrastructure for compliance**

We've seen the team take its first steps toward treating their infrastructure as code. Just as application code needs to be secure and comply with regulations, so does your infrastructure code. What are some concepts and tools to consider for inspecting and validating infrastructure for compliance?

Early attempts at automating the creation of virtual machines and entire environments focused on writing procedural code in languages like PowerShell. The problem with this type of coding was that it was hard to write idempotent code, in other words, code that produces the same outcome no matter how many times you run it. [Windows PowerShell Desired State Configuration (DSC)](https://docs.microsoft.com/en-us/powershell/scripting/dsc/overview/overview?view=powershell-6%2F%3Fazure-portal%3Dtrue) was a big step forward because it allowed you to define the desired outcome itself, instead of describing the process for achieving that outcome. It was a declarative mechanism rather than a procedural one.

Today, a variety of tools enable you to define infrastructure as code, and these have already created significant benefits for DevOps teams. Reliable environments (including operating systems and services) can be created on demand, and in a completely repeatable way. The same code can be run again to ensure that the target state is present, and to remove any drift from that state.

Rather than just configuration of the environments though, there is a need to validate the environments, particularly in relation to organizational or regulatory policies that might relate to them.

**Validation tooling**

[Azure Automation State Configuration](https://docs.microsoft.com/en-us/azure/automation/automation-dsc-overview/) builds on DSC to make management easier. It allows you to author and manage DSC Configurations, import DSC Resources, and generate DSC Node Configurations (MOF documents), all in the cloud. These items are placed on an Automation server so that target nodes in the cloud or on-premises can retrieve them, and automatically conform to the desired state, and report back on their compliance.

Another tool, [InSpec](https://www.inspec.io/) is an open-source testing framework for infrastructure with a language for specifying compliance, security and other policy requirements. It works by comparing the actual state of your system with the desired state that you express in InSpec code. It then detects violations and displays findings in the form of a report. The remediation is then done by the administrator.