DevOps

Week 06

Murtaza Munawar Fazal

Infrastructure as Code (IaC)

- Infrastructure as code is the concept of managing your operations environment like you do applications or other code for general release.
- Rather than manually making configuration changes or using one-off scripts to make infrastructure changes, the operations infrastructure is managed instead using the same rules and strictures that govern code development—particularly when new server instances are spun up.
- That means that the core best practices of DevOps—like version control, virtualized tests, and continuous monitoring—are applied to the underlying code that governs the creation and management of your infrastructure.

Infrastructure as Code (IaC)

- With infrastructure as code, you capture your environment (or environments) in a text file (script or definition).
- Your file might include any networks, servers, and other compute resources.
- You can check the script or definition file into version control and then use it as the source for updating existing environments or creating new ones.
- For example, you can add a new server by editing the text file and running the release pipeline rather than remoting it into the environment and manually provisioning a new server.

Manual deployment versus IaC

- A common analogy for understanding the differences between manual deployment and infrastructure as code is the distinction between owning pets and owning cattle.
- When you have pets, you name each one and regard them as individuals; if something terrible happens to one of your pets, you're inclined to care a lot.
- If you have a herd of cattle, you might still name them, but you consider their herd.
- In infrastructure terms, there might be severe implications with a manual deployment approach if a single machine crash and you need to replace it (pets).
- If you adopt infrastructure as a code approach, you can more easily provision another machine without adversely impacting your entire infrastructure (cattle) if a single machine goes down.

Manual deployment versus IaC

Manual deployment	Infrastructure as code
Snowflake servers.	A consistent server between environments.
Deployment steps vary by environment.	Environments are created or scaled easily.
More verification steps and more elaborate manual processes.	Fully automated creation of environment Updates.
Increased documentation to account for differences.	Transition to immutable infrastructure.
Deployment on weekends to allow time to recover from errors.	Use blue/green deployments.
Slower release cadence to minimize pain and long weekends.	Treat servers as cattle, not pets.

Benefits of infrastructure as code

- Promotes auditing by making it easier to trace what was deployed, when, and how.
- Provides consistent environments from release to release.
- Greater consistency across development, test, and production environments
- Automates scale-up and scale-out processes.
- Allows configurations to be version-controlled.
- Provides code review and unit-testing capabilities to help manage infrastructure changes.

Benefits of infrastructure as code

- Uses immutable service processes, meaning if a change is needed to an environment, a new service is deployed, and the old one was taken down; it isn't updated.
- Allows blue/green deployments. This release methodology minimizes downtime, where two identical environments exist—one is live, and the other isn't. Updates are applied to the server that isn't live. When testing is verified and complete, it's swapped with the different live servers. It becomes the new live environment, and the previous live environment is no longer the live. This methodology is also referred to as A/B deployments.
- Treats infrastructure as a flexible resource that can be provisioned, de-provisioned, and reprovisioned as and when needed.

Configuration Management

- Configuration management (also known as configuration as code) refers to automated configuration management, typically in version-controlled scripts, for an application and all the environments needed to support it.
- Configuration management means lighter-weight, executable configurations that allow us to have configuration and environments as code.
- For example, adding a new port to a Firewall could be done by editing a text file and running the release pipeline, not by remoting into the environment and manually adding the port.

Manual configuration v/s configuration as code

- Manually managing the configuration of a single application and environment can be challenging.
- The challenges are even more significant for managing multiple applications and environments across multiple servers.
- Automated configuration, or treating configuration as code, can help with some of the manual configuration difficulties.

Manual configuration versus configuration as code

Manual configuration	Configuration as code
Configuration bugs are challenging to identify.	Bugs are easily reproducible.
Error-prone.	Consistent configuration.
More verification steps and more elaborate manual processes.	Increase deployment cadence to reduce the amount of incremental change.
Increased documentation.	Treat environment and configuration as executable documentation.
Deployment on weekends to allow time to recover from errors.	
Slower release cadence to minimize the requirement for long weekends.	

Imperative versus declarative configuration

- Declarative (functional)
  - The declarative approach states what the final state should be. When run, the script or definition will initialize or configure the machine to have the finished state declared without defining how that final state should be achieved.
  - A declarative approach would generally be the preferred option where ease of use is the primary goal. Azure Resource Manager template files are an example of a declarative automation approach.
- Imperative (procedural)
  - In the imperative approach, the script states the how for the final state of the machine by executing the steps to get to the finished state. It defines what the final state needs to be but also includes how to achieve that final state. It also can consist of coding concepts such as for, *if-then, loops, and matrices.
  - An imperative approach may have some advantages in complex scenarios where changes in the environment occur relatively frequently, which need to be accounted for in your code.

Azure Resource Manager templates

- Using Resource Manager templates will make your deployments faster and more repeatable.
- For example, you no longer must create a VM in the portal, wait for it to finish, and then create the next VM. The Resource Manager takes care of the entire deployment for you.

Parameters – ARM Templates

- Here's an example that illustrates two parameters: one for a virtual machine's (VMs) username and one for its password:

```
"parameters": {
 "adminUsername": {
  "type": "string",
  "metadata": {
   "description": "Username for the Virtual Machine."
  }
 },
 "adminPassword": {
  "type": "securestring",
  "metadata": {
   "description": "Password for the Virtual Machine."
  }
 }
```

Variables - ARM Templates

- Here's an example that illustrates a few variables that describe networking features for a VM:

```
"variables": {
  "nicName": "myVMNic",
  "addressPrefix": "10.0.0.0/16",
  "subnetName": "Subnet",
  "subnetPrefix": "10.0.0.0/24",
  "publicIPAddressName": "myPublicIP",
  "virtualNetworkName": "MyVNET"
}
```

Functions - ARM Templates

- Here's an example that creates a function for creating a unique name to use when creating resources that have globally unique naming requirements:

```
"functions": [
 {
  "namespace": "contoso",
  "members": {
   "uniqueName": {
    "parameters": [
     {
      "name": "namePrefix",
      "type": "string"
     } ],
    "output": {
     "type": "string",
     "value": "[concat(toLower(parameters('namePrefix')), uniqueString(resourceGroup().id))]"
    } } } } ]
```

Resources - ARM Templates

- Here's an example that creates a public IP address resource:

```
{
  "type": "Microsoft.Network/publicIPAddresses",
  "name": "[variables('publicIPAddressName')]",
  "location": "[parameters('location')]",
  "apiVersion": "2018-08-01",
  "properties": {
    "publicIPAllocationMethod": "Dynamic",
    "dnsSettings": {
      "domainNameLabel": "[parameters('dnsLabelPrefix')]"
    } } }
```

Outputs – ARM Templates

- Here's an example that illustrates an output named hostname.
- The FQDN value is read from the VM's public IP address settings:

```
"outputs": {
  "hostname": {
    "type": "string",
    "value": "[reference(variables('publicIPAddressName')).dnsSettings.fqdn]"
  }
}
```

Circular dependencies

- A circular dependency is a problem with dependency sequencing, resulting in the deployment going around in a loop and unable to continue.
- As a result, the Resource Manager can't deploy the resources.
- Resource Manager identifies circular dependencies during template validation.

Deployments modes - ARM Templates

- Validate
  - This option compiles the templates, validates the deployment, ensures the template is functional (for example, no circular dependencies), and correct syntax.
- Incremental mode (default).
  - This option only deploys whatever is defined in the template. It doesn't remove or modify any resources that aren't defined in the template. For example, if you've deployed a VM via template and then renamed the VM in the template, the first VM deployed will remain after the template is rerun. It's the default mode.
- Complete mode:
  - Resource Manager deletes resources that exist in the resource group but isn't specified in the template. For example, only resources defined in the template will be present in the resource group after the template deploys. As a best practice, use this mode for production environments to achieve idempotency in your deployment templates.

Azure Bicep

- Azure Bicep is the next revision of ARM templates designed to solve some of the issues developers were facing when deploying their resources to Azure. It's an Open Source tool and, in fact, a domain-specific language (DSL) that provides a means to declaratively codify infrastructure, which describes the topology of cloud resources such as VMs, Web Apps, and networking interfaces. It also encourages code reuse and modularity in designing the infrastructure as code files.
- The new syntax allows you to write less code compared to ARM templates, which are more straightforward and concise and automatically manage the dependency between resources. Azure Bicep comes with its command line interface (CLI), which can be used independently or with Azure CLI. Bicep CLI allows you to transpile the Bicep files into ARM templates and deploy them and can be used to convert an existing ARM template to Bicep.

Scope – Bicep

- By default the target scope of all templates is set for resourceGroup, however, you can customize it by setting it explicitly. As other allowed values, subscription, managementGroup, and tenant.

Parameters - Bicep

- They allow you to customize your template deployment at run time by providing potential values for names, location, prefixes, etc.
- Parameters also have types that editors can validate and also can have default values to make them optional at deployment time. Additionally, you can see they can have validation rules to make the deployment more reliable by preventing any invalid value right from the authoring. For more information, see Parameters in Bicep.

```
param location string = resourceGroup().location
```

Variables - Bicep

- Similar to parameters, variables play a role in making a more robust and readable template. Any complex expression can be stored in a variable and used throughout the template. When you define a variable, the type is inferred from the value.

- var uniqueStorageName = '${storagePrefix}${uniqueString(resourceGroup().id)}'

Resources - Bicep

- The resource keyword is used when you need to declare a resource in your templates. The resource declaration has a symbolic name for the resource that can be used to reference that resource later either for defining a subresource or for using its properties for an implicit dependency like a parent-child relationship.
- There are certain properties that are common for all resources such as location, name, and properties. There are resource-specific properties that can be used to customize the resource pricing tier, SKU, and so on.

```
resource storage 'Microsoft.Storage/storageAccounts@2021-02-01' = {
    name: 'examplestorage'
    location: resourceGroup().location
    kind: 'StorageV2'
    sku: {
        name: 'Standard_LRS'
    }
}
```

Modules - Bicep

- Modules enable you to reuse a Bicep file in other Bicep files. In a module, you define what you need to deploy, and any parameters needed and when you reuse it in another file, all you need to do is reference the file and provide the parameters. The rest is taken care of by Azure Bicep.

```
module storageModule './storage.bicep' = {
  name: 'linkedTemplate'
  params: {
    location: location
    storageAccountName: storageAccountName
  }
}
```

Outputs - Bicep

- You can use outputs to pass values from your deployment to the outside world, whether it is within a CI/CD pipeline or in a local terminal or Cloud Shell. That would enable you to access a value such as storage endpoint or application URL after the deployment is finished.
- All you need is the output keyword and the property you would like to access:

- output storageEndpoint endpoints = stg.properties.primaryEndpoints