Corndel DevOps Engineering Programme

in association with Softwire

Module 12: Cloud Infrastructure as Code

Corndel Digital.

DB47

Introduction

Last module we talked about setting up environments in the cloud. This is reasonably easy, as the cloud providers have invested heavily in portals for creating and managing instances of their services. However, provisioning resources manually using web portals isn't particularly scalable; creating a second environment means going through the same steps again, hoping that they have been documented exhaustively and clearly enough that the new environment is identical to the first.

Reliability and efficiency of creating environments is very important. If you have a system that is complete and fully tested on one environment but you cannot easily create a live environment, then the system is not ready to go live. When it does go live you can't be confident that subtle differences between the test and production environments won't lead to bugs in production that didn't occur in test. This risk remains for the lifetime of the environment. Prior to that stage an inability to create test environments reliably and on demand can seriously hamper development by creating testing bottlenecks, and wasting time ironing out bugs in the setup.

The key rationale behind developing software is that automating repetitive tasks makes them more reliable and require less effort to carry out. CI/CD already applies this principle to verifying and deploying updates, and tools are also available to achieve the same with setting up environments in the first place. By replacing these manual steps with code we have both completely documented how environments are set up and ensured that this is accurate and consistent between them. We also make it much quicker and more reliable to "spin up" new environments, including disaster recovery if needed and even "throwaway" environments for testing.

By defining your environment in source control, its current state and history are easily visible to anyone with the source code. It can also evolve with the source code: if your application gains the ability to accept uploaded files from the user, then the code changes implementing this feature can be added alongside the changes to add storage resources to the environment, and both these changes can be deployed simultaneously.

Table of Contents

Core Material

Introduction

Chapter 1:
Basic Principles

Chapter 2:
Options
- Declarative or imperative

Chapter 3:
Security in Infrastructure as Code
- Firewalls and virtual networks
- Web Application Firewalls and DDoS Protection
- API Management
- Securing data and credentials

Chapter 4:
Continuous Deployment
- Sharing Deployment State
- Considerations for Impact on Users
- Ordering deploys
- Managing secrets
- Handling failure

Chapter 1

Basic principles

From the previous module you will have seen deploying resources via the cloud provider's CLI. These could of course be put together into a script with parameters so you can name an environment. This would allow you to create a consistent and reproducible environment, but wouldn't allow configuration changes after deployment and instead you'd have to manually make these changes on any extant environments, again introducing the possibility of inconsistencies. Such a simple script wouldn't let you take down the environment when it is done with, or gracefully handle failures when deploying. The "code" of "Infrastructure as Code" is ultimately just a list of required resources with their configuration. Resources are simply anything that needs to be deployed in the cloud. These include your App Services and Databases, but also the compute and networking resources to support these such as Resource Groups and App Service Plans in Azure.

A more complete solution is to define the **desired state** of the resources instead of listing instructions to create and configure them. This could then be checked against what actually exists in the environment (if anything) and create, destroy, or reconfigure the resources as required. Such a system also needs to support dependencies between resources, as before creating an App Service you may need to create a virtual network for it to join, and pass its name into the configuration.

Chapter 2

Options

Each of the main cloud providers has its own implementation of Infrastructure as Code:

- AWS CloudFormation
- Azure Resource Manager Templates
- GCP Cloud Deployment Manager

There are also cross platform tools including:

- Terraform
- Pulumi

The cross platform tools allow resources to be created in a large number of different clouds providers. However each of these resources is specific to a particular cloud provider, for instance rather than using a general type for a SQL Server database, you might use an Azure SQL Database or Amazon RDS. This means the configuration you create will be specific to the cloud provider it was written for. The cross platform tools however enable at least knowledge sharing between these platforms even though the configuration created with them is not portable.

Declarative or imperative

Declarative code states what is wanted, imperative code is a set of instructions to achieve it. An example of the two might be the difference between a SQL query

```
SELECT * FROM Orders WHERE Status = 'OPEN'
```

and the C# code

```
foreach (var order in orders) {
    if(order.status == "OPEN") openOrders.add(order)
}
```

They both do approximately the same thing, but the former states what is wanted leaving it up to the database to work out how to get the answer, whereas the latter is a specific list of steps to calculate it. It is possible to write declarative style code in an imperative language e.g.

```
var openOrders = orders.Where(order => order.Status == "OPEN").
```

As Infrastructure as Code is about declaring the desired state of an environment it lends itself well to a declarative format such as JSON or YAML, but just as with the example above it is possible to express this in an imperative language.

Terraform, the AWS and Azure tools are purely declarative, that is the code that is checked in is a configuration file such as JSON rather than instructions that are run. Another cross platform option Pulumi instead is more imperative, you write code in a programming language to build the environment. GCP's Cloud Deployment Manager has a choice of either style. Ultimately both approaches build a model of what the environments should look like, which is then applied rather than method calls directly creating setting up the environment.

> *Declarative code abstracts away the complexity of actually creating the desired resources by delegating this to the framework; however it lacks some of the fine control of an imperative approach. Declarative approaches can simplify your setup; at the expense of some control.*

"Clean code" is as important here as it is elsewhere, as this code will need to be updated alongside the project and sometimes fixed if there are issues. For instance in a microservices architecture you may have a number of very similar web apps and databases. **Code reuse** keeps each resource simple by keeping common configuration in one place rather than needing each resource to have a copy of it all. **Encapsulation** keeps each web app together with its database and other dependent resources as these are just an implementation detail of the service rather than something that should be exposed more widely.

The imperative style allows code reuse and encapsulation using techniques and constructs already familiar to developers for example a function that returns a resource given the required parameters.

For instance in Pulumi we might create a database as follows:

```
  allowAllAzureIpsAccessToDb =
pulumi.Config().require("allowAllAzureIpsAccessToDb")

  sql_server = azure_native.sql.Server("sqlserver", ...)

  if allowAllAzureIpsAccessToDb:
    azure.sql.FirewallRule("allowAllAzureIpsFirewallRule",
      server_name=sql_server.name, ...)

  db = azure_native.sql.Database("db",
    server_name = sql_server.name, ...)

  connection_string = pulumi.Output.all(
    sqlserver.fully_qualified_domain_name, db.name) \
    .apply(lambda args:
        f'Server=tcp:{args[0]},1433;Initial Catalog={args[1]};'
    )
```

Note that if statement from imperative programming makes the code clear. However the apply method makes defining the connection string convoluted - this code is all run before anything is deployed so the value can't be a string but an object that keeps track of its dependencies ready to be calculated after deploying the database.

The declarative style is perhaps a more natural fit, but to achieve this requires new constructs to be learned such as ways of expressing conditions and Terraform's modules.

The example above in Terraform would be:

```
variable "prefix" {
  description = "The prefix used for all resources"
}

variable "allowAllAzureIpsAccessToDb" {
  type = bool
}

resource "azurerm_sql_server" "main" {
  name                 = "${var.prefix}-sqlserver"
  ...
}

resource "azurerm_sql_firewall_rule" "allow_all_azure" {
  name                 = "FirewallRule1"
  server_name          = azurerm_sql_server.main.name
  count                = var.allowAllAzureIpsAccessToDb ? 1 : 0
  ...
}

resource "azurerm_sql_database" "main" {
  name                 = "${var.prefix}-db"
  server_name          = azurerm_sql_server.main.name
  ...
}

locals {
  connection_string = "Server=tcp:$
{azurerm_sql_server.main.fully_qualified_domain_name},1433;Initial
Catalog=${azurerm_sql_database.main.name}"
}
```

Note the slightly hidden nature of the condition here. Terraform doesn't have if statements so instead we set count to 0 if we don't want to create the resource. The expressions however are very clear, and let Terraform build up a dependency tree by knowing that the database depends on the server being created first by virtue of depending on its name.

Chapter 3

Security in Infrastructure as Code

Whilst cloud providers take care of a number of security concerns for you, such as physical security and patching, there are still security considerations that need addressing when managing cloud services.

Traditionally with on-premises infrastructure the first line of defence is the corporate firewall preventing outside access to resources unless they are specifically allowed through. This was insufficient by itself as just one compromised device (or person!) on the network could gain access to a number of systems. By default cloud deployed resources are generally accessible to the whole internet not just your own organisation, so even this limited level of protection cannot be relied on to automatically protect your services. As such, particularly careful consideration needs to be paid to securing access. The 0-trust principle should be applied - rather than relying on the network being secure you should instead limit access to the minimum required to run the systems.

Firewalls and virtual networks

Rather than exposing your services to be accessed from anywhere on the internet, cloud services come with firewalls which can restrict the origin of requests.

If your service is directly accessed only by members of your organisation, whether via a web browser or a desktop or mobile application then you may be able to restrict access to devices on the organisations network. It is necessary to consider if all devices that access is required from will be on the network, as increasingly systems need to be accessed by homeworkers, or on mobile phones rather than just PCs in the office. If these devices all connect to a VPN then this won't be a problem, but if not it may be impossible to protect your service this way.

It is likely that most of the cloud resources you have will need to be accessed by other resources rather than users directly, for instance a web app accessing its database. By default there may not be a range of IP addresses you can permit without enabling all other customers on the cloud provider to connect to the database. There is little security advantage in, say, simply restricting your database to be accessed by the IP range Azure App Services use, as a potential hacker who was aware of your database could easily create their own App Service to gain access.

A dedicated public IP address for outbound requests from your resource would enable you to setup firewall rules, but this is not always an option. Alternatively the cloud providers allow you to create virtual networks of resources. It is possible to restrict traffic to only being within these networks, or better to only be between specific resources. It's also possible to connect this to on premises services, by linking your virtual and physical networks with a VPN.

Web Application Firewalls and DDoS Protection

Web Application Firewalls and **DDoS Protection** are additional 'bolt-on' forms of security that can be added onto existing applications relatively easily. They work by intercepting all incoming traffic *before* it hits the application and preventing obviously harmful traffic from reaching the application.

Examples include AWS WAF, Azure Web Application Firewall and Cloudflare.

A Web Application Firewall (WAF) is a specific kind of firewall for detecting and rejecting suspicious requests before they hit web apps. They can mitigate potential vulnerabilities in your web app, for instance the SQL Injection we mentioned in Module 10.

Suppose a multi-tenanted application mapped a request to `/users?role=admin` to the query

```
SELECT * FROM Users WHERE organisationId = <OrganisationId> AND
role = ‘admin'
```

then a request to `/users?role=admin'OR'1'='1` will result in

```
SELECT * FROM Users WHERE organisationId = <OrganisationId> AND
role = ‘admin'OR'1'='1'
```

This will result in all users returned including those from other organisations. A WAF may pick up this as a suspicious looking request and refuse to execute it. This is not an alternative to *preventing* (or *fixing*) injection attacks injection attacks properly in your codebase (e.g. through the use of parameterised queries)! A codebase with a simple injection vulnerability (such as this one) is likely to have more, and the WAF will not be able to detect every possible exploitation of such a vulnerability. In general, a tool such as a WAF is designed to add another layer of defence to your application, not be your only means of defence.

DDoS Protection is similar but blocks suspicious *groups* of (rather than individual) requests before they hit web apps, mitigating concerted effort to take web apps offline by overloading them with requests. In the case of a cloud app if you have autoscaling enabled this could also cost a lot of money for resources, on top of the lost revenue from downtime. There is some DDoS protection included by default with Azure DDoS Protection Basic and AWS Shield.

These can be purchased from the main cloud providers, and there are third party solutions such as Cloudflare. The cloud providers' own options are fully integrated and can be deployed and configured as just another part of your infrastructure as code, whereas third party services will need more setup, including ensuring only traffic from the WAF can access your application directly, rather than just anyone with the URL. The savings for using a third party can be quite substantial, particularly for DDoS protection as for instance Azure DDoS Protection Standard starts at over £2,000 a month (https://azure.microsoft.com/en-us/pricing/details/ddos-protection/).

API Management

For example Amazon API Gateway and Azure API Management.

While firewalls protect applications as a whole, API management tools let you expose and secure APIs on an endpoint by endpoint basis. If you have a service where some of the endpoints are for internal use only and others need to be exposed to third parties, you can use API management to set up different IP allow-lists for different endpoints, restricting the possible attack surface. You can also apply other protections such as rate limiting or caching endpoints so an overeager or malicious consumer cannot take down your application or run up large bills by sending too many requests. The API management tool can also be responsible for authentication and authorisation where it isn't necessary for your application itself to know the identity of its users, or as a second layer of defence preventing users who shouldn't have access from sending a request to your app in the first place.

Outside of security benefits, API management tools can improve the experience of consuming APIs, for instance by enabling mock APIs for a specification before the implementing application has been developed or bringing together a variety of services into a more consistent usable set of APIs.

Securing data and credentials

It's frequently in the news that a company has accidentally exposed sensitive customer data via unsecured cloud storage (https://www.theregister.com/2020/08/03/leaky_s3_buckets/). Whilst firewalls can help mitigate this, locking down the data so it is only accessible where legitimately required is essential to protecting data.

Firstly of course data should only be publicly readable if it is the intention that anyone in the world should be able to access it, for instance static resources on a public website, or freely available downloads. If you want users to download data directly from the storage service then it is possible to generate secure links that allow a single file to be downloaded for a limited period of time. For instance in Amazon S3 you can pre-sign URLs. This enables your application to serve large files without the performance cost whilst keeping control of the data.

The credentials that are used by applications to access data need securing too. Passwords or API Keys are the simplest way of protecting an account, and many services will generate a connection string with a long unguessable key in it. This can be inserted into the configuration of the application for example as an environment variable. This provides good protection against external threats to your systems, but anyone who has ever had access to this key can continue to use it anonymously themselves. This can be mitigated with key rotation, by making the keys people previously had access to useless after a period of time and by restricting the reading of the key further. For instance with Azure Key Vault you can store secrets and make them available to your applications without exposing credentials to everyone who has access to manage those applications.

The best solution is to get the cloud provider to manage the credentials for you in the first place. That way no person ever needs to directly access the credentials; the cloud itself issues, distributes, and cycles them securely. An example of this is using System Assigned Managed Identities in Azure. This creates a Service Principal, effectively a user account but for an application, and allows the application to generate auth tokens for required resources. There is no password to leak, and the tokens have a very limited lifespan. In some cases it is not necessary to rewrite the application to take advantage of this, for instance the standard drivers for Microsoft SQL Server support this out of the box so it is just a configuration change to use it. AWS has similar functionality in IAM.

Many databases and file storage services will be configured to always encrypt your data at rest, but this still allows those with access the ability to read the data. An additional layer of protection for data is encrypting it within the application before sending it to the database or file store. SQL Server and Azure Blob Storage support encrypting data within the application transparently doing so within the client library. With this in force you need to have both access to the database or storage account and the encryption key to access data. The encryption key will need to be stored in a secret store such as Azure Key Vault. This allows Dev Ops and support users limited access to do maintenance and investigation task whilst protecting the data, or possibly just particularly sensitive data. For instance on an employee appraisal system it would be possible to protect just the text of appraisals and any scores, whilst leaving the rest of the data accessible as usual to allow efficient querying and other actions. There are downsides to this approach, for instance for encrypted columns your applications can only query for exact matches, and then only if "deterministic" encryption is used. However deterministic encryption is weaker particularly if there are repeated values. For instance if there is an "IsOnImprovementProgramme" column with two values and only a small minority of employees have one value, you can confidently guess that they are on the programme.

Chapter 4

Continuous Deployment

Once the environment is defined in source control it can and should be tested and deployed alongside any other changes to the code. This means it should be added to the Continuous Deployment setup. This might seem scary, your build wont just update existing resources with new code, but also potentially create resources that cost money or destroy them if they are no longer needed. These are legitimate concerns, but ones that need to be addressed anyway. By including the Infrastructure as Code in your CD process you are in fact de-risking this as it will ensure it is frequently tested on all environments.

Sharing Deployment State

After you have first provisioned your infrastructure, to enable modification and destruction the IaC tools need to be aware of what is currently deployed. Some tools achieve this by comparing your template to actual resources that have been deployed, whereas others store a separate listing of what has been created, generally referred to as the **state**.

When applying Azure's ARM templates your configuration will be compared to the actual resources deployed in a 'Resource Group'. Any missing resources will be created and existing ones will be updated if needed, but by default if a resource is removed from an ARM Template it won't be deleted.

Terraform and Pulumi store the state of a deployment separately. This has a number of advantages, for instance it is possible to generate and store passwords as part of deploying the code for the first time. Pulumi can also generate unique names for resources, so that you can easily create different environments (stacks) and so stores the generated names in the state.

This state needs to be shared amongst the team, or at minimum be available to the CD server for it to be able to apply updates when not running on your local machine. Although it is recommended to store your IaC configuration in version-control alongside your source-code, this state information cannot be stored here. Firstly it will often contain extremely sensitive data, including specific resource ids and secrets, so it should be stored somewhere more secure and ideally encrypted. Secondly the state represents what is currently deployed, so unlike with code in source control the latest version must always be used.

These tools also seek to prevent simulations deployments, as these would conflict. This can be done by storing a lock in the state, or in the case of ARM Templates by generally having all the resources to be in a single Resource Group which can be locked for deployments.

To enable this, most IaC tools will offer configurable "backends", see the applicable docs for Pulumi and Terraform for more information. The backends they usually recommend are their own paid services, which have the added benefit of allowing you to also apply updates remotely as well as store state, keeping the data even more secure.

However, in practise it is usually easier (and cheaper) to use a simple storage blob in whichever cloud platform you are already using - all that is really required is the secure storage of a json blob that a deployment script can be given credentials to access. There are pre-configured connections between to the usual blob storage services, for example Pulumi to Azure BlobStorage] and Terraform to AWS S3 which can be quickly configured to enable this functionality.

Considerations for Impact on Users

Many of your resources such as web apps will be stateless. If they are taken down and recreated unnecessarily it may result in a longer downtime on releases but won't cause any data loss. Doing the same thing to a database or storage account will probably result in catastrophic data loss.

Set up Re-Deployments to Reduce Downtime

Rather than simply destroying and recreating your stateless resources when re-deploying during CD, most Cloud Providers have specific mechanisms for bringing live production up to the latest version of the source code whilst minimising downtime caused by fully re-provisioning the resource. This can vary from fully-managed services such as AWS CodeDeploy to simple webhooks that cause a app-refresh behind the scenes without any effects for the end-user.

Configure Stateful Resources to prevent destruction

By default all the resources described in Infrastructure As Code configurations are meant to be easily repeatable, reproducible and replaceable.

This is obviously not the case for your main live database, where replacing it without transferring data could be disastrous for users.

One approach is to avoid having the live database included in the IaC config, with it set up separately and connections strings passed in, with temporary databases only set up for test. Although potentially safer from a data-retention point of view, this runs the risk of the live and test environments' configurations becoming inconsistent over time.

A preferred approach is to specify within the IaC configuration that some resources are not to be destroyed unless specifically requested, protecting the live database but allowing identical temporary test databases to be provisioned. For example, this can be done by using AWS's DeletionPolicy or Terraform's Lifecycle meta-arguments.

Ordering deploys

The Infrastructure as Code tool is aware of dependencies between resources and will create or update them based on this order.

For instance a VM will depend on network infrastructure, referencing a network interface by id, which then references the subnet and finally virtual network also by their ids. These then all need to be created in the right order, which is worked out from the implicit dependency of these references. When destroying resources the opposite ordering is used, as the network cannot be destroyed whilst it has a VM on it. It is also possible to add explicit dependencies if needed.

When integrating the Infrastructure as Code template as a step in your deployment pipeline you'll need to work out what order the steps need to be run in. For instance if you build and deploy in the same pipeline you'll want to ensure the latest code builds and tests pass before making any changes to your environments. More generally if the deployment job is to fail it is very much preferable for it to happen before actually makes changes to the environment so that it is left in a working state. This doesn't however mean deploying resources should always be the last step, as you will need resources such as App Services to be deployed before you can push code to them.

Managing secrets

As a part of creating an environment you'll also need to create secrets to enable resources to connect to each other.

You can generate passwords as part of your deploy, for instance with a "random-password" resource in Terraform. This can then be used in configuring other resources, for instance adding it a secret in Azure Key

Vault, and granting access to the resource requiring it. With this setup it is possible to have secrets that are never visible to anyone.

Of course it is also possible to setup other security features like Service Principals, as mentioned in the previous chapter, and IP filtering rules so they apply right from creation.

The state generated by the Infrastructure as Code tool may contain secrets itself so it is important that it is store securely, for instance by encrypting it.

Handling failure

It will always be possible for something to go wrong during a deploy leaving your environment in an inconsistent state. Fortunately if something does go wrong part way the state should be updated to reflect what has successfully occurred. This means transient errors can be resolved by reapplying the changes, as it will know which resources have already been created or updated. If the error is in the configuration then fixing that and reapplying should make the required changes to existing resources so the deployment can continue. Alternatively if the changes that we applied were non-destructive, applying an earlier version of the template should bring back the system.

If however the state of the environment goes out of sync with the stored version of it then you will need to bring them into sync. It may be possible to do this automatically by running a command, or manual steps may be needed to get them carefully back into line.

Rolling back any system is tricky, it would normally only be required if something went wrong, and it is simply impossible to test every possible error path is recoverable in advance. Ensuring that changes have already been applied on production and taking care to only make changes via Infrastructure as Code, even if that is the quickest fix to an urgent bug, will reduce the chance of breaking environments during a deploy.

Unfortunately, the features support by IaC solutions tend to lag the actual cloud features available. It is often the case that specific resources or properties can only be created/updated directly in the cloud platform. This is usually only a temporary situation, but there will be a tradeoff between keeping all your changes in IaC and using the latest, most up to date features. If any infrastructure needs to be updated outside of IaC then ensure it is thoroughly documented to ensure important, manual changes are not lost.