SDN MAGAZINE

Nummer
145
dec 2022

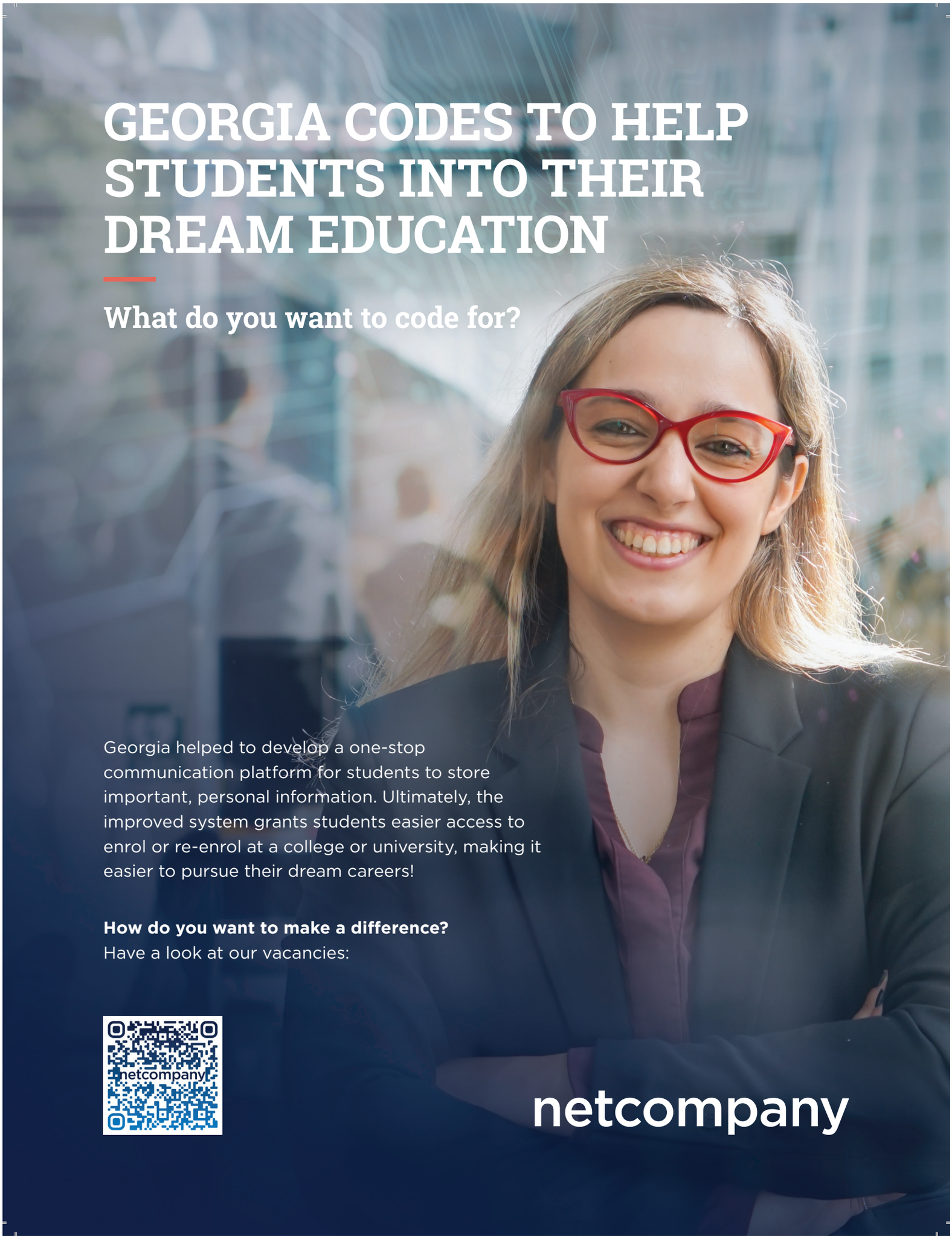
SDN Magazine is
een uitgave van:
SDN

Van en voor Microsoft & .NET professionals

> .NET 7 is hier!

> Duik in de wereld van Infrastructure-as-Code

> Word Tarzan in de jungle van DevOps


GEORGIA CODES TO HELP STUDENTS INTO THEIR DREAM EDUCATION
What do you want to code for?
Georgia helped to develop a one-stop communication platform for students to store important, personal information. Ultimately, the improved system grants students easier access to enrol or re-enrol at a college or university, making it easier to pursue their dream careers!
How do you want to make a difference?
Have a look at our vacancies:
netcompany
netcompany

Colofon

Uitgave
Software Development Network
26ste jaargang
Nr. 145 • december 2022

Redactiecommissie:
Annejan Barelds, Nadine Wolff, Jan de Vries, John Bruin, Menno Jongerius, Marcel Meijer, Roy Janssen, Roelant Dieben, Vincent Hendriks.

Auteurs:
Bart van Nierop, Peter Rombouts, David de Hoop, Rob van Geloven, Roelant Dieben, Patrick Vroegh, Thomas Vieveen, Yaël Keemink & Jasper Sprengers.

Listings:
Zie de website www.sdn.nl voor eventuele source files uit deze uitgave

Contact:
Software Development Network
info@sdn.nl
redactie@sdn.nl

Adverteren:
Informatie over adverteren en de tarieven kunt u opvragen bij
Kelly Verschoor
kelly.verschoor@sdn.nl

Beste SDN-ers,

Het einde van het jaar is in zicht, de Sint is alweer even het land uit en de huizen en straten worden weer versierd met kerstbomen en lichtjes.

Vorig jaar rond deze tijd kwam het eerste vernieuwde magazine uit, heel gek om te bedenken dat het jaar voorbijgevlogen is waarin dit alweer de 5e editie is van het vernieuwde magazine. Dus heel veel dank aan alle auteurs en redactiecommissieleden van het afgelopen jaar! Zonder jullie was dit allemaal niet mogelijk geweest.

Maar naast het magazine hebben we nog andere mooie dingen mogen doen dit jaar met de SDN.

Ondanks dat we aan het begin van het jaar allemaal nog een beetje in de tang van COVID gehouden werden en de geplande Future Tech conferentie van maart nog net niet door kon gaan, werd het daarna al snel een stuk beter. In juni mochten we dan ook Future Tech eindelijk weer in-person organiseren. Het was een groot succes met veel mooie sessies, een goed gevulde beursvloer en blije deelnemers, sprekers & exposanten.

Dit jaar hebben we ook de SDN Cast in een nieuw jasje gestoken. Er zijn veel nieuwe mooie casts online gezet en we waren zelfs aanwezig op Techorama. Ook daarvan hebben we een aantal mooie casts online weten te zetten.

2023 klopt alweer aan onze deuren, dus tijd om nog heel even vooruit te kijken en alvast een belangrijke datum in jullie agenda's te zetten: 15 maart! Op deze dag wordt Future Tech 2023 georganiseerd, in de Jaarbeurs, Utrecht. Wij kijken er alweer erg naar uit, hopelijk jullie ook. Willen jullie alvast jullie kaartje bemachtigen? Dat kan! Alle leden van de SDN hebben op 12 december een email ontvangen met een link naar hun gratis ticket. Ben je geen lid? Meldt je dan snel aan via sdn.nl en dan ontvang jij jouw gratis ticket ook! Mailtje kwijt? Geen probleem, de gratis ticket wordt nogmaals in januari verstuurd.

Genoeg over 2022 & 2023 en tijd om lekker onderuitgezakt bij de kerstboom te gaan zitten en ook deze editie van het magazine door te lezen.

Even een speciale dank aan onze auteurs die deze editie van het magazine voorzien hebben met een mooi en leerzaam artikel: Bart van Nierop, Peter Rombouts, David de Hoop, Rob van Geloven, Roelant Dieben, Patrick Vroegh, Thomas Vieveen, Yaël Keemink & Jasper Sprengers.
En natuurlijk een grote dank aan onze redactiecommissie.

Dan rest mij alleen nog te zeggen: Veel leesplezier, Fijne Feestdagen, een mooie en gezellige jaarwisseling en een heel voorspoedig en gezond 2023!

Tot volgende jaar.
Kelly Verschoor

DEVITJOBS.NL

Senior Software Developer

Salarisindicatie:
45.000 - 65.000 EUR

Locatie:
Delft

Technologieën:
C#, Java, PHP, Azure, Kubernetes, Docker, JavaScript, Angular, NodeJS, React, .NET

netcompany

Senior Software Developer

Salarisindicatie:
45.000 - 75.000 EUR

Locatie:
Delft

Technologieën:
Java, C++, C#, Cloud, DevOps, Docker, Vue, React, Angular, JavaScript

TOPdesk
Your guides to service excellence

Azure DevOps Engineer

Salarisindicatie:
60.000 - 85.000 EUR

Locatie:
Nieuwegein

Technologieën:
Azure, Power BI, PowerShell, C#, DevOps, CI/CD, .NET

ORDINA
Software Development

Medior .NET developer

Salarisindicatie::
45.000 - 60.000 EUR

Locatie:
Amstelveen

Technologieën:
C#, MVC, .NET Core, Angular, JavaScript, HTML5, CSS, CI/CD, Git, Jenkins, Azure, JIRA, Confluence, Kubernetes, Docker

Atos

Find all these vacancies and more

at Devitjobs.nl

In dit nummer

11 In de huidige wereld van cloud native, PaaS en SaaS zou je haast vergeten dat nog heel veel workloads draaien op virtuele machines (VMs). Voor software developers vaak een blok aan het been, en vaak ook niet binnen het blikveld van een team. Hoe kun je nu in je volledig geautomatiseerde CICD omgeving deze machines toch een plek geven?

20 Een vraag waar we in onze dagelijkse werkzaamheden misschien niet direct stil bij staan, maar waarom testen we nu eigenlijk? De antwoorden hierop zijn uiteenlopend en variëren van het wel heel pragmatische “om de code coverage te verhogen” tot de beschrijving van mogelijke gevolgen “om zo min mogelijk bugs te hebben”.

26 Begin november is er weer een nieuwe versie van .NET gereleased. Versie 7 van .NET is een current version, waarbij de support dus korter is dan die van een Long-Term Support (LTS) versie. Met de 18 maanden support die we op .NET 7 krijgen, loopt deze dus een half jaar eerder af dan de support op .NET 6. Dit betekent niet dat je deze versie van .NET links moet laten liggen, want er zijn weer tal van verbeteringen en optimalisaties doorgevoerd. In dit artikel neem ik je mee door de belangrijkste.

6 redenen om voor je volgende project F# te kiezen

Auteur: Bart van Nierop

Functional programming is in. Als je tech nieuws bijhoudt, dan lees je met enige regelmaat over bijvoorbeeld Haskell, OCaml, Reason, Clojure, Scala of F#. Je leest dat je met een functionele programmeertaal leesbaardere, eenvoudigere en kortere code kunt schrijven waar je makkelijker over kunt redeneren.

Echter, in de praktijk zien we deze programmeertalen niet vaak gebruikt worden door bedrijven. "Selection bias" kan tot het beeld leiden dat functionele progammeertalen vaak gebruikt worden, maar in vergelijking met meer traditionele objectgeoriënteerde talen valt dat best mee. Functionele programmeertalen scoren laag op populariteitslijstjes. [1][2]
Veel traditionele programmeertalen nemen echter steeds meer functies over van functionele programmeertalen, dus er zit wel enige waarde in die functies.
Dit artikel beschrijft de belangrijkste functies van functionele programmeertalen in het algemeen en F# in het bijzonder. Het doel is om te laten zien dat gebruik van functionele programmeertalen nuttig kan zijn en dat het geen kwaad kan om F# te overwegen voor een aankomend project.

Immutability

Bij functionele programmeertalen wordt de voorkeur gegeven aan waardes die niet kunnen veranderen. Niet alle talen zijn daar even streng in, maar dat is wel altijd de insteek. Het idee is dat de output van een functie, gebaseerd is op de input. Een functie past zijn input niet aan, verzamelt zelf geen eigen input behalve wat is opgegeven en maakt geen aanpassingen aan de buitenwereld (bijvoorbeeld `Console.WriteLine` of `Random.Shared.Next()`).
Dit lijkt een redelijk simpel idee en de voordelen van functies zonder bijwerkingen (ook wel *pure* genoemd) zijn groot. Als gebruiker van een functie zonder bijwerkingen kun je ervan uitgaan dat alle objecten of data die meegegeven worden (en ook alle objecten en data die *niet* meegegeven wordt), onveranderd blijven.
Neem bijvoorbeeld de volgende implementatie van een functie die de waardes in een lijst met getallen verdubbelt. **Listing 1**

In het voorbeeld hierboven is het aan de gebruiker van de functie om te bepalen of de input verandert. Vergelijk dit met dezelfde functie in een functionele programmeertaal, waar het veranderen van de inputlijst onmogelijk is zonder daarvoor je best te doen: **Listing 2**
`List.map` is een functie die zijn eerste argument, een functie, uitvoert op alle waardes in het tweede argument, een lijst, en het resultaat teruggeeft zonder dat de input wordt aangepast. Het equivalent in C# ziet er als volgt uit: **Listing 3**
C# heeft de laatste jaren een heleboel functionaliteit overgenomen uit F# en dit voorbeeld geeft aan hoeveel de talen op sommige punten op elkaar lijken.
Dit is een erg geforceerd voorbeeld, maar dingen die onverwachts veranderen in een flinke code base zijn een veelvoorkomende oorzaak voor lange debugsessies.

Type inference

Functionele programmeertalen leiden in het algemeen zelf de types van functies en hun arguments af zonder dat deze opgegeven hoeven te worden. Het resultaat daarvan is

Functionele programmeertalen scoren laag op populariteitslijstjes.

```
List<int> Double(List<int> list)
{
  for (int i = 0; i < list.Count; ++i) {
    list[i] *= 2;
  }
  return list;
}

// Verderop …

List<int> numbers = new() { 1, 2, 3 };
List<int> doubled = Double(list);
```

1

```
let double list = List.map ((*) 2) list
```

2

```
List<int> Double(List<int list) => list.Select(x => x * 2);
```

3

```
int Square(int x) => x * x; // C#
let square x = x * x // F#
```

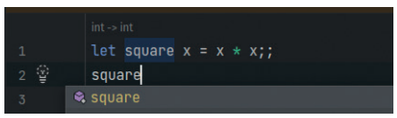
4

```
type Shape =
| Circle of float              // Cirkel met radius
| Rect of (int * int)          // Rechthoek met hoogte * breedte
| Composed of (Shape * Shape) // Combinatie van twee figuren
```

5

> Vind deze en de andere listingen uit het artikel op sdn.nl!

dat deze types niet geschreven hoeven te worden. Dat houdt dan weer in dat een functionele programmeertaal minder woorden nodig heeft om hetzelfde uit te drukken als in een taal die deze types niet zelf afleidt. Neem als voorbeeld onderstaande code in zowel C# als F#. Het is een definitie van een functie die als argument een integer accepteert en als output een integer teruggeeft. Beide voorbeelden zijn strongly typed, maar in het F#-voorbeeld hebben we dat niet aan hoeven geven. **Listing 4**
Je zou kunnen beargumenteren dat dit de code, zeker in een complexe codebase, minder leesbaar maakt, maar niets is minder waar. Het type van de functie wordt gewoon getoond door de editor.



Sum types & Pattern matching

Een sum type is een datastructuur gemaakt om een waarde te bevatten die een van verschillende, voorgedefiniëerde types kan zijn. In de praktijk lijkt dit op een enum met data van een bepaald type voor iedere mogelijke waarde van de enum. Dit kan er bijvoorbeeld als volgt uitzien: **Listing 5**

Aan de hand hiervan kunnen we deze waardes gemakkelijk creëren en beslissingen nemen aan de hand van het type figuur. Om dezelfde structuur te krijgen in een object georiënteerde taal zouden we een aparte klasse moeten maken voor ieder van de figuren. Ook is ofwel een `typeof`-check, ofwel functies toevoegen aan `Shape` die niet altijd logisch zijn voor subclasses, óf bijvoorbeeld het Visitor design pattern nodig. Daar is niks mis mee, maar het is wel behoorlijk meer code dan de functionele optie.
Als een sum type een super-enum is, is pattern matching een super-`switch`. Een voorbeeld, gebasseerd op bovestaande `Shape`. **Listing 6**

```
let rec area shape =
  match shape with
  | Circle radius -> radius * radius * 3.14
  | Rect (width, height) -> float (width * height)
  | Composed (Circle radiusA, Circle radiusB) -> (radiusA * radiusA * 3.14) +
(radiusB * radiusB * 3.14)
  | Composed (a, b) -> (area a) + (area b)
```

6

```
abstract record Shape();
record Circle(double Radius) : Shape;
record Rect(int Width, int Height) : Shape;
record Composed(Shape A, Shape B) : Shape;

static double Area(Shape shape)
{
    return shape switch {
        Circle c => c.Radius * c.Radius * 3.14,
        Rect r => r.Width * r.Height,
        Composed { A: Circle c1, B: Circle c2 } =>
            c1.Radius * c1.Radius * 3.14 + c2.Radius * c2.Radius * 3.14,
        Composed c => Area(c.A) + Area(c.B),
        _ => throw new ArgumentOutOfRangeException(nameof(shape), shape, null)
    };
}
```

7

Dit is redelijk duidelijke code, maar er gebeurt toch een hoop.
Regel (1) definiëert een recursieve functie, `area`, die de oppervlakte van de figuren berekent. Op regel (2) begint de match op de argument, `shape`.
Regel (3) matcht een `Circle`. De waarde die daarbij hoort, een float, kennen we toe aan de naam `radius`. Achter de -> staat de expressie die `match` uiteindelijk zal teruggeven.

Regel (4) matcht een `Rect` en kent de waarde toe. De waarde van een `Rect` en een tuple met twee integers. De integers worden toegekend aan de namen `width` en `height`.
Regel (5) matcht een `Composed`, enkel als de `Composed` bestaat uit twee `Circle s` Deze regel toont dat `match` meerdere clausules kan hebben voor hetzelfde type en voegt verder weinig toe
Regel (6) tenslotte matcht de overige waardes van `Composed`. Zouden we deze regel weglaten, dan klaagt de F#-compiler dat niet alle mogelijkheden gematcht worden. Doordat alle mogelijkheden afgehandeld worden is een default case niet nodig.
C# heeft sinds versie 7 ook pattern matching, al is het niet even krachtig als de pattern matching in F#. **Listing 7**
Het belangrijkste verschil in de C#-versie is de laatste regel.

De C#-compiler weet *niet* dat, zonder de laatste regel, alle mogelijke combinaties zijn gecontroleerd. Daardoor moet de laatste clausule, de wildcard _, ook worden toegevoegd. Het eerste nadeel daarvan is dat erover moet worden nagedacht *wat* erin gedaan moet worden in het geval er geen match is. Callers moeten in dit geval de `ArgumentOutOfRange-Exception` afhandelen. Het tweede nadeel is dat wanneer we een extra type `Shape` toevoegen, de C#-compiler niet zal klagen en deze ArgumentOutOfRange-Exception ook echt voor gaat komen.
De F#-compiler klaagt wel.
In C# kan deze eenvoudige case natuurlijk ook worden opgelost door een abstract method op `Shape`. En daar is niets mis mee. Polymorphism en Sum Types lossen hetzelfde probleem op een andere manier op. Beiden hebben voor- en nadelen.
In het geval van polymorphism is het makkelijker om nieuwe types toe te voegen. Sum Types maken het makkelijker om nieuwe functies toe te voegen.

Units of Measure

Het is in functionele programmeertalen door de combinatie van type inference en de eenvoudige syntax om types te definiëren, erg makkelijk om kleine types te maken voor type safety. Units of Measure zijn een built-in manier van F# om dit te doen voor getallen.

```
bool CheckSpeedLimit(int speedInKmh) => speedInKmh <= 100;
// Verderop ...
int speedInMph = 80;
CheckSpeedLimit(speedInMph); // Returns true!
```

8

```
[<Measure>] type km
[<Measure>] type mile
[<Measure>] type h

let checkSpeedLimit speed = speed <= 100.0<km/h>

checkSpeedLimit 80.0<km/h>
checkSpeedLimit 80.0<mile/h>
checkSpeedLimit 80.0
```

9

```
let milesPerHourToKmPerHour (mph : float<mile/h>) = mph * 1.6<km> / 1.0<mile>

checkSpeedLimit (milesPerHourToKmPerHour 80.0<mile/h>)
.NET Interop
```

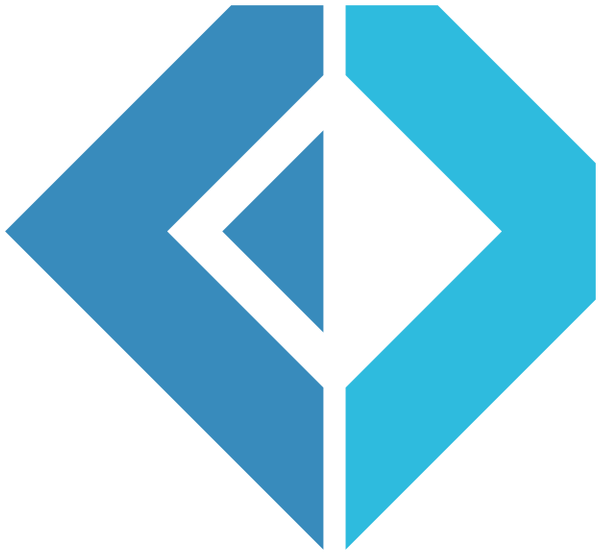
10

BIO

Bart van Nierop

Bart van Nierop is een bevlogen software-ontwikkelaar die altijd op zoek is naar nieuwe ideeën en technieken om software en het leven een beetje beter te maken.

Een module geschreven in F# kan in C#-code gebruikt worden. Je zit dus nooit vast aan de taal.



Stel je voor dat je werkt met snelheden. Er zijn landen op de wereld waar ze nog altijd niet inzien dat kilometers per uur de makkelijkste manier is om te rekenen, maar waar nog altijd in mijlen (per uur) gewerkt wordt.
Als we de twee door elkaar heen gebruiken, kan dat voor problemen zorgen. **Listing 8**
F# geeft ons de mogelijkheid om dit (eenvoudig) te voorkomen: **Listing 9**
Regels (1-3) definiëren types voor kilometers, mijlen en uren. Regel (5) definiëert een functie die een snelheid (in km/h) als input heeft en een boolean als output.
Deze functie wordt op regel (8) aangeroepen met een snelheid in km/h. Deze aanroep geeft `true` terug, zoals verwacht.
Op regel (9) wordt een poging gedaan om de functie aan te roepen met een snelheid in miles per uur. Dat lukt echter niet. F# geeft een foutmelding:
error FS0001: Type mismatch. Expecting a 'float<km/h>' but given a 'float<mile/h>' The unit of measure 'km/h' does not match the unit of measure 'mile/h'
Ook de call op regel (10), zonder unit of measure, gaat fout:
error FS0001: This expression was expected to have type 'float<km/h>' but here has type 'float'
Gelukkig is het mogelijk om te converteren tussen Units of Measure. De onderstaande code definieert een functie om te converteren van mile/h naar km/h en roept vervolgens `checkSpeedLimit` aan met deze conversie. Deze chauffeur rijdt duidelijk te snel. **Listing 10**

.NET Interop

In de introductie worden programmeertalen als Haskell en OCaml genoemd. Hoewel dat prima programmeertalen zijn, hebben ze ook een groot nadeel. Omdat relatief weinig mensen ze gebruiken, zijn er relatief weinig (open source) libraries.
Een groot voordeel van F# op dit gebied is dat het een .NET-taal is. Daardoor kan gebruik gemaakt worden van het volledige .NET-ecosysteem. Inclusief .NET Framework libraries, .NET6 libraries en alles op NuGet.
Een nadeel is dat libraries geschreven voor C# over het algemeen niet de best practices voor F# hanteren, zoals immutability en pure functions. Dat is iets om in de gaten te houden bij het gebruik van een third party library. Desalnietemin is het feit dat er veel beschikbaar is, een groot voordeel.
Dit werkt twee kanten op. Een module geschreven in F# kan in C#-code gebruikt worden. Je zit dus nooit vast aan de taal.

Multi paradigm

Hoewel de focus van dit artikel ligt op functional programming, is F# geen pure functionele programmeertaal. De taal zelf ondersteunt mutatie en object georiënteerd programmeren.
Dat betekent dat, om te beginnen met F#, je kunt beginnen door simpelweg de syntax van de taal te leren, zonder je bezig te hoeven houden met het leren van functional design patterns en de manier van denken die hoort bij het schrijven van code zonder side effects, terwijl je wel direct het voordeel hebt van pattern matching, sum types en units of measure.
Wanneer je eenmaal comfortabel bent met de syntax kun je dan functionele concepten toevoegen aan je modules. Binnen een .NET-solution kunnen C#-projecten en F#-projecten door elkaar heen gebruikt worden. Binnen een project niet.

Conclusie

We hebben gekeken naar zes features van functionele programmeertalen en F# in het bijzonder. Deze features maken dat het code geschreven in een functionele programmeertaal voorspelbaarder, korter en eenvoudiger is dan dezelfde code in traditionele programmeertalen. Ook kan er door het type-system van functionele programmeertalen meer verantwoordelijkheid bij de compiler gelegd worden.

Hopelijk geeft dit artikel inspiratie om F# in een volgend project als serieuze kandidaat te overwegen. ●

Referenties
[1]: https://www.tiobe.com/tiobe-index/
[2]: https://pypl.github.io/PYPL.html

Voor sponsorships neem contact op met Kelly Verschoor - kverschoor@reshift.nl

Een plek voor VMs binnen je Cloud Native landschap? **Het kan met Packer!**

In de huidige wereld van cloud native, PaaS en SaaS zou je haast vergeten dat nog heel veel workloads draaien op virtuele machines (VMs). Voor software developers vaak een blok aan het been, en vaak ook niet binnen het blikveld van een team. Hoe kun je nu in je volledig geautomatiseerde CICD omgeving deze machines toch een plek geven?

Auteur: Peter Rombouts

In dit artikel laat ik zien hoe je HashiCorp Packer kan gebruiken om je VMs kan opnemen in je automated deploy en dit herhaalbaar en volledig geautomatiseerd kan neerzetten.

Probleemstelling

Stel je bent met je team een complete applicatie aan het bouwen in de Azure cloud. Een front end in Static Web Apps, een CosmosDB voor je opslag en een ServiceBus voor communicatie. Echter is er één COTS (Commercial-off-the-shelf) applicatie die je moet gebruiken voor wat logica. In dit voorbeeld hebben we niet de ruimte of tijd om deze WindowsService opnieuw te schrijven, en het hosten van een dergelijke service is (buiten VMs) heel erg lastig. Uiteraard neem je in je roadmap op dat je dit component graag wil vervangen, maar dat is wegens licentie redenen niet haalbaar binnen een jaar.
Nu kun je er voor kiezen om de service te laten hosten door een managed omgeving. Dit uit te besteden aan je partner of een andere partij.

Echter is dat complex. Denk aan bijvoorbeeld connecties die de service moet maken naar je landschap. Je zit met peering, networking, etc. Idealiter wil je dit dus gewoon opnemen binnen je team, en zelf je broek ophouden als het gaat om de hosting, updates en security van het geheel.

De oplossing

VMs uitrollen, compleet met software is relatief eenvoudig, als de image reeds bestaat. Denk aan een DevTest lab in Azure, of een marketplace image in Azure. Dat zijn voorbeelden van images compleet met voorgeïnstalleerde software. Deze rol je uit en je kan ze gebruiken.

In de wereld van containers lijken VMs maar lastig, groot en oud, maar een container baseer je simpelweg ook op een image. En ja, de image van een VM is doorgaans groter maar de cloud providers spinnen dat snel op, dus het is een prima oplossing om een image te maken en die te koppelen aan een machine in Azure. In dit hoofdstuk laat ik zien hoe je een image kan opslaan en gebruiken in Azure, en hoe je die maakt met behulp van de HashiCorp tool Packer.

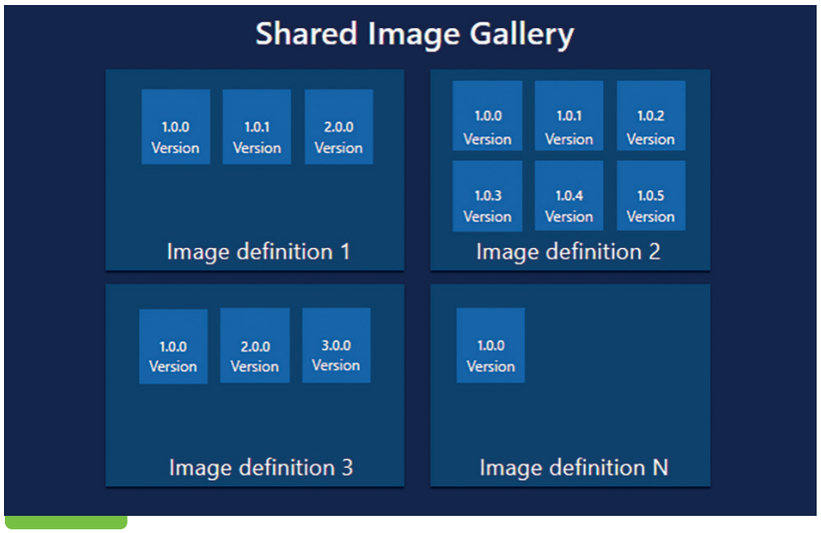
Shared Images

in basis kan een image niet meer zijn dan een .VHD bestand op je harddisk. Maar in Azure kun je een Shared Image Gallery (SIG) maken. Dat is een opslagplek voor images die je daarna kan gebruiken. Dat scheelt lokaal opslag, en het is ook eenvoudig toegankelijk voor je VMs. Voorbeelden van images zijn standaard Windows of Linux omgevingen, tot en met een hardened versie van je omgeving. De basis van je applicatie is dus zo'n image.

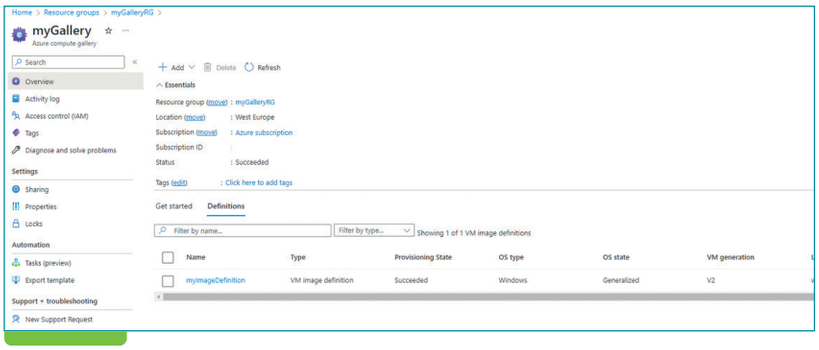
1

```
# Resource group en image gallery
az group create --name myGalleryRG --location westeurope
az sig create --resource-group myGalleryRG --gallery-name myGallery

# Standaard Windows definitie
az sig image-definition create \
   --resource-group myGalleryRG \
   --gallery-name myGallery \
   --gallery-image-definition myImageDefinition \
   --publisher myPublisher \
   --offer myOffer \
   --sku mySKU \
   --os-type Windows \
   --hyper-v-generation V2 \
   --os-state Generalized
```

> Vind deze en de andere listingen uit het artikel op sdn.nl!



Afbeelding 1



Afbeelding 2. Tip: Open een Azure Cloud shell om dit voorbeeld direct na te spelen.

Omdat wij een COTS applicatie willen neerzetten zonder enige handmatige configuratie, zullen we dus een begin moeten maken met een Shared Image Gallery. Om niet te diep in te gaan op dit concept, slaan we even wat concepten plat door middel van **afbeelding 1**. Hier zien we in concept een SIG met daarin een viertal definities. Elke definitie kan ook weer een versie hebben. Denk aan een image voor Windows 10 en een voor Windows 11. Conceptueel niet erg ingewikkeld. Voor dit voorbeeld gaan we uit van één definitie, met één versie. Let wel; dit

```
build.pkr.hcl
build {
  sources = ["source.azure-arm.windows2022"]

  provisioner "powershell" {
    script = "scripts/main.ps1"
  }
}
sources.pkr.hcl
source "azure-arm" "windows2022" {
  location = "westeurope"

  tenant_id       = var.tenant_id
  subscription_id = var.subscription_id
  client_id       = var.client_id
  client_secret   = var.client_secret

  managed_image_resource_group_name = "myGalleryRG"
  managed_image_name                = "myImageDefinition-1.0.0"

  image_offer     = "WindowsServer"
  image_publisher = "MicrosoftWindowsServer"
  image_sku       = "2022-datacenter-azure-edition"
  os_type         = "Windows"
  vm_size         = "Standard_D2s_v4"

  communicator   = "winrm"
  winrm_use_ssl  = true
  winrm_insecure = true
  winrm_timeout  = "10m"
  winrm_username = "packer"

  subscription      = var.subscription_id
  resource_group    = "myGalleryRG"
  gallery_name      = "myGallery"
  image_name        = "myImageDefinition"
  image_version     = "1.0.0"
}
```

2

```
scripts/main.ps1
# Start of script
$InstallSourcePath = 'C:\Install'
New-Item $InstallSourcePath -ItemType "directory" -Force

# Start downloading Azure CLI
Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows -OutFile $In-
stallSourcePath\AzureCLI.msi

# Start install Azure CLI
Start-Process -filepath msiexec.exe -Wait -ErrorAction Stop -ArgumentList "/I $In-
stallSourcePath\AzureCLI.msi /qn"

# SysPrep machine
Write-Log "Starting Sysprep"
& $env:SystemRoot\\System32\\Sysprep\\Sysprep.exe /oobe /generalize /quiet /quit /
mode:vm
    while($true) { $imageState = Get-ItemProperty HKLM:\\SOFTWARE\\Microsoft\\
Windows\\CurrentVersion\\Setup\\State | Select-Object ImageState; if($imageState.
ImageState -ne 'IMAGE_STATE_GENERALIZE_RESEAL_TO_OOBE') { Write-Output $imageState.
ImageState; Start-Sleep -s 10  } else { break } }
}

# Cleanup downloads
Remove-Item -Path "$InstallSourcePath" -Recurse -Force
```

3

zijn allemaal nog lege items.
De daadwerkelijke image moet gemaakt worden met een tool!
Het aanmaken van een SIG kan met behulp van AZ CLI zeer eenvoudig:
Listing 1
Deze code maakt de resourcegroup aan, met een shared image gallery en daarin één definitie, voor een Windows image.
Let op dat je voor sommige base images een **V1** of een **V2 hyper-v-generation** in moet stellen. By default wordt de V1 genomen. In ons voorbeeld is de basis image waar we de COTS op installeren een **2022-datacenter-azure-edition**, en dat is een V2 hyper-v-generation image.
Het eindresultaat in de Azure portal ziet eruit als **afbeelding 2.**

HashiCorp Packer

Waarschijnlijk gebruik je voor het deployen van je infrastructuur en componenten een tool als bicep, Pulumi of Terraform. Die laatste is van HashiCorp, en zij maken ook de tool Packer. Deze tool maakt gebruik van dezelfde taal (HashiCorp Configuration Language) en zorgt voor een zeer eenvoudige en overzichtelijke manier van images maken.
In dit artikel gaan we niet in op de installatie van Packer, want die is dermate simpel en te vinden op de site https://packer.io/downloads. Daar zijn ook talloze voorbeelden hoe images te maken voor AWS, Azure, Google en alle andere grote platformen die je kan gebruiken als doelomgeving.
In dit voorbeeld gaan we uit van Azure, waar we de SIG hebben gemaakt, met één image definitie met daarin één versie. Hoe maken we nu de Packer code, om deze image te bouwen? Er zijn twee HashiCorp Configuration Language (HCL) bestanden nodig om de meest simpele setup te maken voor een image, en een PowerShell voor alle zaken die in Windows moeten gebeuren. Allereerst de HCL:
Listing 2
De voornoemde HCL code genereert het image op de Azure plek waar we de shared image gallery hebben aangemaakt. Packer doet dat door 'onder water' een VM op te spinnen, uit te rollen en daarna te packagen. Wellicht vraag je je af waarom er een VM size in staat. Dit is dus de VM size waarmee Packer gaat builden. Als de image gereed is kun je gewoon een ander type VM size

VMs uitrollen, compleet met software is relatief eenvoudig, als de image reeds bestaat.

pakken, naar gelang je nodig hebt. Kies nooit een te kleine VM voor builden, immers is tijd geld in de cloud, en een zwaardere VM zal sneller klaar zijn met alle stappen. De provisioner die in de build file zit, start een PowerShell file. Hieronder de contents van die PowerShell. Mocht je een Linux image opbouwen heb je uiteraard andere opties, bijvoorbeeld een bash script. **Listing 3**

```
# User logged in using AZ CLI
packer validate .
```

4

```
# User logged in using AZ CLI
packer build .
```

5

```
az group create --name myResourceGroup --location eastus
az vmss create \
   --resource-group myResourceGroup \
   --name myScaleSet \
   --image "/subscriptions/<Subscription ID>/resourceGroups/myGalleryRG/providers/
Microsoft.Compute/galleries/myGallery/images/myImageDefinition" \
   --generalized
```

6

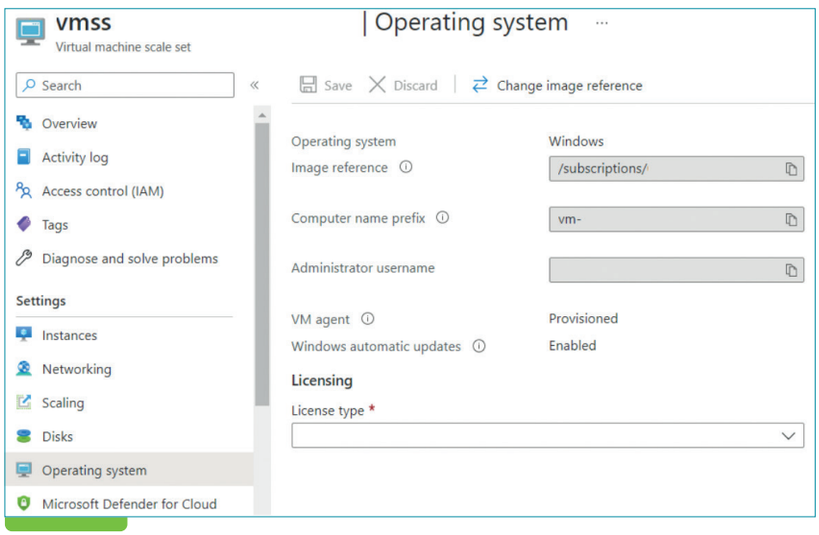
```
azure-arm.windows2022: output will be in this color.

==> azure-arm.windows2022: Running builder ...
==> azure-arm.windows2022: Getting tokens using client secret
==> azure-arm.windows2022: Getting tokens using client secret
    azure-arm.windows2022: Creating Azure Resource Manager (ARM) client ...
==> azure-arm.windows2022: Getting source image id for the deployment ...
==> azure-arm.windows2022:  -> SourceImageName: '/subscriptions/<sensitive>/providers/Microsoft.Compute/locations/westeurope/publis
==> azure-arm.windows2022: Creating resource group ...
==> azure-arm.windows2022:  -> ResourceGroupName : 'pkr-Resource-Group-b350uslay1'
==> azure-arm.windows2022:  -> Location          : 'westeurope'
==> azure-arm.windows2022:  -> Tags              :
==> azure-arm.windows2022: Validating deployment template ...
==> azure-arm.windows2022:  -> ResourceGroupName : 'pkr-Resource-Group-b350uslay1'
==> azure-arm.windows2022:  -> DeploymentName    : 'pkrdpb350uslay1'
==> azure-arm.windows2022: Deploying deployment template ...
==> azure-arm.windows2022:  -> ResourceGroupName : 'pkr-Resource-Group-b350uslay1'
==> azure-arm.windows2022:  -> DeploymentName    : 'kvpkrdpb350uslay1'
```

Afbeelding 3

In bovenstaande code zien we een simpel voorbeeld van het installeren van de Azure CLI op de Windows machine. Daarna wordt er een SysPrep gedaan op de machine, nodig om deze voor te bereiden om als image te kunnen worden uitgerold, en als laatste ruimen we netjes de installatiefolder op.
In het real world scenario werd hier de COTS applicatie geïnstalleerd, en daarbij ook firewall poorten opengezet in Windows.

Packer uitvoeren

Als alle code klaar is, kun je Packer uitvoeren om daadwerkelijk de image te maken en te deployen. Omdat er feitelijk een VM wordt gestart, alle software wordt geïnstalleerd en daarna een image wordt getrokken, kan dit proces best wat tijd in beslag nemen. Een half uur is niet uitzonderlijk. Let ook op de VM size in je script, daarmee spelen kan leiden tot betere, snellere builds.
Omdat we in dit voorbeeld niet werken met variabelen in bestanden, is een simpel commando in Packer voldoende om de configuratie te valideren: **Listing 4**
Door het bovenstaande commando valideert Packer alle bestanden in



Afbeelding 3

BIO

Peter Rombouts

Peter Rombouts is een Cloud Software Architect met bijna 20 jaar ervaring, met een brede achtergrond in infrastructuur, firewalling, networking en software architectuur. Ook schrijft hij regelmatig eBooks en whitepapers voor Microsoft, en is hij Udemy auteur met twee gepubliceerde trainingen omtrent Cognitive Services.

Zijn werkgebied bevindt zich momenteel op het snijvlak van cloud, software engineering, low-code en integratie. Hij is uitgesproken fan van alles met een stekker, accu of wielen en specialist op Azure, HashiCorp Terraform, Cloud Native en Integration Services.

LinkedIn: https://www.linkedin.com/in/peterrombouts78/
Blog: https://peterrombouts.nl

Packer wordt ook intern bij Microsoft gebruikt voor Azure Image Builder

de huidige directory. Net zoals bij Terraform maakt het niet uit of je alles in één bestand plaatst of juist in losse. Voor de overzichtelijkheid is een scheiding wel erg makkelijk. En uiteraard is het PowerShell script wel een apart bestand (in dit voorbeeld in de submap scripts).
Nu rest ons slechts nog de start van de build, met behulp van onderstaand commando: **Listing 5**
Ook hier wordt met de punt aangegeven dat de code in de huidige directory aanwezig is.
Een voorbeeld van zo'n build in Azure DevOps is weergegeven in **afbeelding 3**.
Hier zie je ook de tijdelijke resourcegroup naam die Packer gebruikt om de machine op te bouwen waarvan een image wordt gemaakt. In dit geval duurt het circa 25 minuten voor de gehele build klaar is, waarbij dus ook de image in de SIG is geplaatst.

Virtual Machine ScaleSet

De image komt natuurlijk pas tot leven als we deze uitrollen. Een van de meest gebruikte oplossingen is een Virtual Machine ScaleSet (VMSS). Het grote voordeel van deze setup is dat we eenvoudig de VMSS kunnen laten wijzen naar een SIG. Daarbij kun je dus automatisch wijzen naar je image, waardoor de VMSS altijd up-to-date is, en de image die je gemaakt hebt met Packer zal gebruiken als de basis.
Met de Azure CLI kunnen we nu heel gemakkelijk een VMSS opspinnen met de volgende commando's: **Listing 6**
Als voorbeeld kan je een screenshot een soortgelijke opzet vinden in **afbeelding 4**.
Hier is de zien dat het Operating system wijst naar een image reference. Bijkomend voordeel; zodra er een nieuwe image beschikbaar is, is het eenvoudig om deze weer uit te rollen in de scale set.
Er moet wel een kanttekening geplaatst worden met deze opzet. Als je COTS applicatie naar een database wijst, moet deze wel geschikt zijn om meerdere instanties toe te laten. Het is daarom wel belangrijk om te controleren dat je zonder problemen met meerdere COTS instanties kan runnen.

Vervolg; Automation en CICD

Alle stappen die werden getoond in dit document zijn eenvoudig te automatiseren. Een van de screenshots is zelfs van een Azure Pipeline runner. De tool Packer wordt ook intern bij Microsoft gebruikt voor Azure Image Builder, en alle hosted runners hebben Packer (maar ook Terraform) pre-installed. De stap naar CICD is dus erg klein.
Later dit jaar komt een whitepaper uit waarin we (Sogeti en HashiCorp) diep ingaan op dit concept, en zelfs images maken op basis van andere images in de SIG, Inception eat your heart out! Hierbij komt ook een complete GitHub repository online waar alle broncode met werkende voorbeelden te vinden zullen zijn. Ook gaan we daar met Terraform te werk om alle componenten neer te zetten in de Azure Cloud, houdt dus de publicaties vooral in de gaten! ●

De Jungle der DevOps functies

Auteur: David de Hoop

DevOps Consultant, Platform Engineer, SRE? Je ziet tegenwoordig door de bomen het bos niet meer. De ene na de andere vacature komt voorbij in je LinkedIn inbox met deze nietszeggende functietitels. Uit de omschrijving haal je waarschijnlijk dat ze allemaal iets met DevOps te maken hebben, máár wat is nu echt het verschil?

Functietransformatie

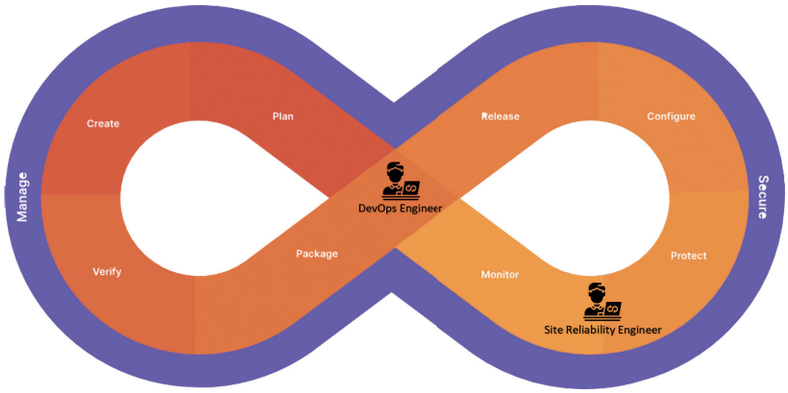
DevOps wint nog altijd aan terrein. Veel organisaties plannen of zitten inmiddels middenin een DevOps transformatie. Dit is niet gek, immers werken volgens de DevOps filosofie is ongekend populair en zorgt voor enorm veel voldoening en werkplezier bij developers, mits goed geïmplementeerd. In organisaties met een grote IT-afdeling en veel componenten die gedeeld worden met meerdere teams kunnen vaak niet alle verantwoordelijkheden verschoven worden naar het development team. In deze DevOps transformaties is vaak behoefte aan mensen die de verschillende IT platformen bouwen en beheren, en die de operatie van de vaak complexe integraties tussen meerdere systemen bewaken. Kortom er is behoefte aan een meer hybride rol binnen de IT operatie wiens focus niet primair bij één team of applicatie ligt. Hier ligt de geboorte van de Platform en Site Reliability Engineer (SRE) aan ten grondslag.

Site Reliability Engineering

De term Site Reliability Engineering (SRE) komt van oorsprong bij Google vandaan en werd al in 2003 geïntroduceerd. Net als DevOps is SRE niet bedacht als rol of functie maar is het een combinatie van principes en best practises uit Software Engineering en past deze toe binnen het IT operations domein. De populariteit van SRE als aparte functie is sterk toegenomen sinds 2016 wanneer steeds meer multinationals een

Het verschil tussen een DevOps en Site Reliability Engineer zit hem … in de plek binnen de DevOps lifecycle waar hun werkzaamheden zich bevinden.

DevOps transformatie ondergingen. Zij hebben behoefte aan aparte teams binnen de IT afdelingen die taken onafhankelijk van de development teams kunnen oppakken, werkzaamheden die vooral met de dagelijkse operatie van de software en de systemen te maken hebben. Het verschil tussen een DevOps en Site Reliability Engineer zit hem dan ook met name in de plek binnen de DevOps lifecycle waar hun werkzaamheden zich bevinden. Waar een DevOps engineer zich vooral bezig houdt met het opzetten van CI/CD en het in productie krijgen van de software en vooral in de Package en Release fase actief is, begint het werk van de Site Reliability Engineer eigenlijk pas wanneer de software in productie draait. Deze houdt zich vooral bezig met Observability en monitoring van de applicatie, capaciteits- en kostenmanagement van de infrastructuur en met SLA's en het incident management proces.



De positie van de DevOps en Site Reliability Engineer binnen de DevOps lifecycle.

Een DevOps transformatie binnen een grote organisatie raakt namelijk ook de traditionele ITIL processen. Deze worden na de transformatie ook ingevuld door de SRE. Daarnaast houden SREs zich ook bezig met het verbeteren van de betrouwbaarheid van de applicaties die zijn monitoren, bijvoorbeeld door het toepassen van Chaos Engineering.

Het "Platform" als interne service provider

Naast het inzetten van SRE is er bij grotere bedrijven vaak nóg een probleem bij een DevOps transformatie. Je wilt niet dat elk development team een eigen tenant in de public cloud gaat hosten of dat elk development team het wiel opnieuw uitvind. Daarnaast zijn er sectoren die streng gereguleerd worden waarbij centraal geregelde governance gewoon een vereiste is. Om dit probleem toch op een agile manier te kunnen oplossen zijn veel van deze bedrijven begonnen met één of meerdere projecten waarbij een aantal development teams werken aan een intern platform. Dit kan puur cloud infrastructuur zijn, bijvoorbeeld in de vorm van een Cloud Center of Excellence (CCoE), maar ook een integratieplatform, centrale API gateway of datawarehouse. Deze projecten functioneren eigenlijk volledig als een aparte leverancier voor de organisatie, alleen dan binnen diezelfde organisatie. De development teams zijn dan klant en tevens afnemer van dit platform. Op deze manier is het binnen deze teams mogelijk om agile te werken op een vraag gestuurde manier én kun je toch een mate van gecentraliseerde controle houden op de operatie van de IT infrastructuur en kun je zaken als governance, kostenbeheersing en heel belangrijk, security centraal regelen.

BIO

David de Hoop

David de Hoop is Special Agent (thought leader) bij Team Rockstars IT. Als solution architect adviseert hij organisaties op het gebied van Public Cloud, DevOps transformaties en Microsoft Azure. David is een fervent Agile aanhanger en DevOps evangelist. Daarnaast haalt hij veel energie uit het geven van workshops en spreken op conferenties over de hele wereld zoals laatst in Hamburg op de ContainerDays en in India op de Azure Community Conference.

Twitter: https://twitter.com/davidshadoow
LinkedIn: https://www.linkedin.com/in/daviddehoop/
Website: https://www.teamrockstars.nl/developers/special-agent-david-de-hoop/

	DevOps Engineer	Site Reliability Engineer	Platform Engineer
Verantwoordelijkheden	Is verantwoordelijk voor het in productie krijgen van nieuwe applicaties.	Is verantwoordelijk voor het draaiend houden en monitoren van bestaande applicaties.	Is verantwoordelijk voor de ontwikkeling van de onderliggende infra- en/of datastructuur.
Key Focus	Focust zich op automatisering van het build en release proces.	Focust zich op de observability, monitoring en stabiliteit van draaiende applicaties.	Focust zich op de capaciteit, beveiliging en governance van de infrastructuur en/of datastore.
Plaats in de organisatie	Onderdeel van een development team	Onderdeel van een development team óf operationele afdeling	Onderdeel van een interne service provider.

Vergelijking tussen DevOps, SRE en Platform Engineers.

De engineers die aan zo'n platform werken worden dan steevast Platform Engineers genoemd. Een nogal niets-zeggende functietitel in een vacature als je niet weet om wat voor platform het gaat. Vaak zijn dit functies waarbij zowel de competenties van een cloud- of netwerk engineer gevraagd wordt samen met enkele competenties van een meer traditionele DevOps rol. Want ook op het interne platform wil je zoveel mogelijk automatiseren. Misschien wil je voor deze platforms zelfs wel aparte SREs hebben die zich ontfermen over de monitoring en observability ervan. Een van de kenmerken van deze interne service providers is dan dat je ze in de DevOps transformatie net zo behandeld als elke andere applicatie. Alleen gaat het in dit geval dan niet om een webapplicatie of mobiele app, maar om een infrastructuur platform, datawarehouse of API gateway. Trouwens een dergelijk platform heeft vaak ook API's op de back-end of een webapplicatie als front-end. Dus eigenlijk is het ook helemaal niet zo raar om deze net zo te behandelen als de applicaties die uiteindelijk van dit platform gebruik gaan maken.

Dé DevOps Engineer bestaat niet

Kortom zoals je hebt kunnen lezen bestaat dé DevOps Engineer niet. Zowel Platform als Site Reliability Engineering zijn onderdeel van de DevOps lifecycle en vervullen zeker in de wat grotere organisaties een belangrijke rol. Waar de competenties voor deze functies ook gelijkenissen vertonen zijn de dagelijkse werkzaamheden behoorlijk verschillend, zoals ook in de volgende vergelijking te zien is.
In de kern houden ze zich allemaal bezig met een bepaalde mate van automatisering, maar waar de DevOps engineer zich vaak toch dichter bij het development team begeeft, zitten de SRE en Platform Engineer meer ingebed in de IT operatie. In die zin zou je kunnen zeggen dat de SRE en Platform Engineer de moderne evolutie van de traditionele systeem- en applicatiebeheerders zijn.
Deze analogie laat zien dat de rol van zowel de SRE als Platform engineer binnen grote organisaties belangrijk zijn voor een succesvolle DevOps transformatie. Door de hedendaagse complexiteit van microservices architecturen en public cloud platformen is er behoefte aan meer diepgaande technische kennis binnen het operationele IT domein. Het ontwikkelen van goede monitoring met betrouwbare en betekenisvolle informatie en het snel kunnen opvolgen van security incidenten is van levensbelang voor een soepele operatie van de gehele bedrijfsvoering die steeds meer door IT gedreven wordt.
Daarnaast is wendbaarheid van de IT operatie in de snel veranderende wereld van public cloud een vereiste. Hier is in het heden en op de korte termijn dan ook een belangrijke rol weggelegd voor de SRE en de Platform Engineer. ●

In de kern houden ze zich allemaal bezig met een bepaalde mate van automatisering.

Waarom testen we?

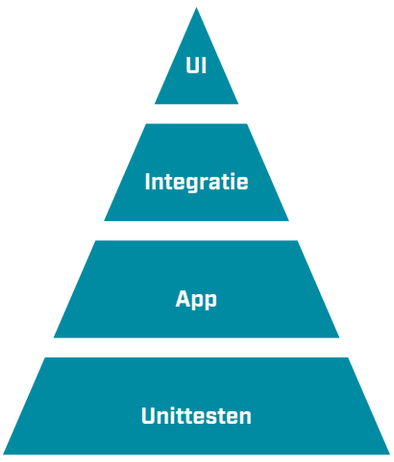
Een vraag waar we in onze dagelijkse werkzaamheden misschien niet direct stil bij staan, maar waarom testen we nu eigenlijk? De antwoorden hierop zijn uiteenlopend en variëren van het wel heel pragmatische "om de code coverage te verhogen" tot de beschrijving van mogelijke gevolgen "om zo min mogelijk bugs te hebben".

Auteur: Rob van Geloven

Het antwoord op deze vraag is wat mij betreft "we testen om een optimale dienstverlening voor onze klanten te kunnen garanderen". Optimaal is hier wat mij betreft het sleutelwoord: we hoeven niet alles perfect te testen. We testen enkel wat noodzakelijk is om de dienstverlening optimaal te kunnen garanderen.
Helaas zien we in de praktijk vaak dat er vooral de focus ligt op het proces: de code coverage moet aan een bepaald percentage voldoen of pull-requests die worden pas goedgekeurd als de nieuwe code ook bijbehorende testen heeft. En hoewel ik een voorstander ben van een goed proces ligt er te vaak de nadruk op randvoorwaarden ten koste van het resultaat.

De testpiramide

Ik merk vaak dat de onderbouwing voor dit soort processen volgt uit de aard van de testen: we gebruiken in de praktijk eigenlijk alleen unittesten. Maar waarom? Als we de testpiramide erbij pakken, zien we dat dit het fundament is, maar dat er nog een hele piramide aan testen op staat:
Vaak vloeit dit voort uit het feit dat unittesten doorgaans goed onderhoudbaar zijn en zich ook uitstekend lenen om volledig te automatiseren. Hoe hoger je in de testpiramide komt, hoe meer je bezig bent om de 'buitenkant' van de applicatie te testen en minder de 'binnenkant'. Dat betekent vaak ook dat er allerlei externe afhankelijkheden moeten worden weggemockt.



Wat zijn destructieve testen?

Destructieve testen zijn testen waarbij de staat van een systeem onherroepelijk wordt gewijzigd. Waarbij je dus, om in database termen te blijven, de transactie ook daadwerkelijk commit naar een store. Testen waarbij je de oude files ook daadwerkelijk verwijderd van een file share. Testen die dus niet zo makkelijk nog een keer zijn uit te voeren.
Regelmatig zie je dat hier weken voorbereidend werk nodig is om alle testdata goed op te zetten in alle onderliggende systemen om vervolgens na een dag testen eigenlijk weer opnieuw te kunnen beginnen. Dat moet toch anders kunnen!

Een eerste opzet

Maar hoe pakken we zoiets nu aan in de praktijk? Laten we eens kijken naar een veelvoorkomend probleem: het mocken van datastores. Als voorbeeld pakken we een simpele webshop met een database. In deze database zit de volgende tabel: **Lisiting 1**
We gebruiken ORM tooling in de vorm van Entity Framework Core voor het benaderen van onze database. Ons doel is om onze eerste applicatietest te gaan schrijven voor een nieuw stuk functionaliteit waar onze verkopers behoefte aan hebben: het kunnen verwijderen van accounts die nooit actief zijn geweest in de webshop. We maken gebruik van Specflow om onze testgevallen te beschrijven. Specflow is een .NET port van de open source sofware Cucumber. Deze software maakt gebruik van de Gherkin syntax om in natuurlijke taal testgevallen te kunnen beschrijven. Onze tester is al zo vriendelijk geweest om een SpecFlow file aan te leveren waarin de test beschreven staat. **Lisiting 2**
Omdat we onze database aanspreken via Entity Framework core lijkt het voor de hand te liggen dat we deze afhankelijkheid mocken. Een korte inventarisatie leert ons dat we gelukkig niet DbSet hoeven te mocken maar dat Entity Framework Core een in-memory implementatie heeft die gebruikt kan worden voor testen. **Lisiting 3**
We schrijven onze specflow test en zorgen ervoor dat onze code doet

```
CREATE TABLE IF NOT EXISTS accounts (
    user_id INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    email VARCHAR ( 255 ) UNIQUE NOT NULL,
    password VARCHAR ( 50 ) NOT NULL,
    first_name VARCHAR ( 50 ) NOT NULL,
    last_name VARCHAR ( 50 ) NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);
```

1

```
#language: nl-NL

Functionaliteit: Het klantenbestand kan worden opgeschoond

Scenario: Een medewerker kan het klantenbestand opschonen
    Gegeven de medewerker Isabella
    Als de medewerker het klantenbestand opschoont
    Dan bevat het klantenbestand geen inactieve klanten meer die zijn aangemaakt
voor 2020-01-01
```

2

```
[When(@"de medewerker het klantenbestand opschoont")]
public void WhenDeMedewerkerHetKlantenbestandOpschoont()
{
_numberOfAccountsRemoved = _numberOfAccountsRemoved = _accountService.RemoveAll
StaleAccounts();
}
Waarbij de functie "RemoveAllStaleAccounts()" het volgende SQL commando uitvoert:
"DELETE FROM Accounts WHERE last_login is null AND created_on < '2020-01-01'"
```

3

```
using DotNet.Testcontainers.Builders;
using DotNet.Testcontainers.Containers;

var testcontainersBuilder =
    new TestcontainersBuilder<TestcontainersContainer>()
    .WithImage("hello-world")
    .WithName("hello-world")
    .WithOutputConsumer(Consume.RedirectStdoutAndStderrToConsole());

await using (var testcontainers = testcontainersBuilder.Build())
{
    await testcontainers.StartAsync();
    Console.ReadKey();
}
```

4

> Vind deze en de andere listingen uit het artikel op sdn.nl!

wat hij zou moeten doen. Echter staat onze applicatie nog geen dag live of de functionaliteit, die we dachten goed getest te hebben, blijkt niet te werken.

Wat is er aan de hand?

Omdat we niet gebruik hebben gemaakt van een echte database in onze testen maar slechts een mock hiervan is er een stuk gedrag van de uiteindelijke database ten opzichte van onze mock niet naar voren gekomen: onze mock is niet case sensitive, de database wel. Waarbij in de mock de tabel "Accounts" wel bestaat is deze niet beschikbaar in de echte database waar hij "accounts" heet met enkel kleine letters. Een makkelijk te missen verschil, maar eentje waar we eigenlijk van verwachten dat we deze hadden kunnen ontdekken via onze testen.

Dit onderstreept ook direct het probleem van mocks: hoe goed ze ook zijn en hoeveel ze ook lijken op de echte afhankelijkheden in onze code, uiteindelijk zijn ze slechts representaties van de werkelijkheid en zullen ze nooit volledig sluitend zijn in hun gedrag.

Maar een echte database gebruiken in testen wordt vaak gezien als een moeilijke, omslachtige of zelfs onmogelijke stap om te maken, maar is dat ook zo?

Er zijn tegenwoordig een veelvoud aan ondersteunende tools om juist de externe afhankelijkheden te mocken in plaats van de code. Voorbeelden hiervan zijn de Azurite emulator voor het testen van software met een afhankelijkheid van Azure Storage of WireMock voor het testen van software met een afhankelijkheid op externe HTTP servers.

Testcontainers

Testcontainers zijn een mogelijk antwoord op het fundamentele probleem dat we niet testen met echte afhankelijkheden. Met Testcontainers kunnen we lichtgewicht instanties maken van zaken als databases, API's, of wat er ook maar in een Docker container past. Vervolgens kunnen we deze instanties gebruiken in onze, al dan niet, geautomatiseerde testen.

Dat klinkt misschien nog wat hoog over, maar laten we het voorbeeld

Dit onderstreept ook direct het probleem van mocks … uiteindelijk zijn ze slechts representaties van de werkelijkheid en zullen ze nooit volledig sluitend zijn in hun gedrag.

```
public PostgreSqlFixture()
{

var testcontainersBuilder =
new TestcontainersBuilder<PostgreSqlTestcontainer>()
.WithDatabase(new PostgreSqlTestcontainerConfiguration()
{
Database = “db”,
                    Username = “db_user”,
                    Password = “db_password”,
})
.WithOutputConsumer(Consume.RedirectStdoutAndStderrToConsole())
.WithWaitStrategy(Wait.ForUnixContainer().UntilCommandIsCompleted($”pg_isready -h
‘localhost’ -p ‘5432’”));

            Container = testcontainersBuilder.Build();
}

public async Task UseBackupFile(byte[] backupFile)
{
await Container.CopyFileAsync(“/tmp/db_backup.dump”, backupFile);

var command = “pg_restore --username=db_user --dbname=db -1 /tmp/db_backup.dump”;

await Container.ExecAsync(command.Split(‘ ‘));
}
```

5

```
In de “BeforeFeature” kunnen we dan dit stuk code toevoegen:
[BeforeFeature]
public static async Task BeforeFeature(FeatureContext featureContext)
{
PostgreSqlFixture postgreSqlFixture = new();

await postgreSqlFixture.InitializeAsync();
await postgreSqlFixture.UseBackupFile(await File.ReadAll
BytesAsync(“Support/db_backup.dump”));

var optionsBuilder = new DbContextOptionsBuilder<StoreContext>();
optionsBuilder.UseNpgsql(postgreSqlFixture.Connection!);

featureContext.Set(postgreSqlFixture);
}
```

6

van onze webshop pakken om er een concreet voorbeeld van te maken.
De eerste vraag is natuurlijk: hoe maak je een Testcontainer en hoe start je deze vervolgens? Allereerst moeten we het package Testcontainers installeren via het commando ‘dotnet add package Testcontainers --version 2.2.0’
Vervolgens hebben we met deze kleine console applicatie al onze eerste Testcontainer draaiend: **Lisiting 4**
Meer is er niet nodig: onder de motorkap zorgt Testcontainers voor het binnenhalen van het Docker image en het instantiëren van de container.
Als we vervolgens de database van de webshop erbij pakken dan zou dat er als volgt uit kunnen zien:
Lisiting 5
Laten we kijken wat dit stuk code voor ons doet, allereerst is het belangrijk om te zien dat we in plaats van de generieke “TestcontainersContainer” in de TestContainersBuilder we nu gebruik maken van de class “PostgreSqlTestcontainer”. Dit is een class van het Testcontainers package die een aantal PostgreSql specifieke zaken al voor ons heeft geconfigureerd.
Met het “WithDatabase” commando specificeren we een aantal extra opties de willen gebruiken voor onze database, namelijk welke database we willen gebruik, en met welke gebruikersnaam en wachtwoord we straks op de database kunnen inloggen.
De regel “WithOutputConsumer” is voor productie scenarios niet per se nodig, deze regel zorgt er voor dat we via de console de output van het Docker image kunnen zien. Dit is vooral handig tijdens debuggen en niet nodig tijdens het uitvoeren van de geautomatiseerde testen.
De regel “WithWaitStrategy(Wait.ForUnixContainer().UntilCommandIsCompleted($"pg_isready -h 'localhost' -p '5432'"))” is een belangrijke. Dit commando blokkeert executie van ons programma totdat de database opgestart is. Dit is belangrijk want als we direct verder zouden gaan en willen interacteren met de container dan zou dit onherroepelijk leiden tot fouten aangezien de database nog niet beschikbaar is.
Met het commando “StartAsync” starten we de container om vervolgens met het commando “CopyFileAsync” een bestand van de lokale storage naar de container te kopiëren.
Als laatste stap herstellen wij de database met het backup bestand “db_backup.dump” via het commando “ExecAsync”.

Integratie met een geautomatiseerde test

Nu we een instantie hebben van onze database, willen we deze graag gebruiken in onze SpecFlow test. Belangrijke tip hierbij is om deze container bij voorkeur slechts één keer per feature op te starten. Ondanks dat de container binnen een aantal seconden gestart is en de backup is teruggezet kan het bij grote hoeveelheden testgevallen al snel vertragend gaan werken.
Het continue downloaden van alle

Belangrijke tip hierbij is om deze container bij voorkeur slechts één keer per feature op te starten.

Docker images kan al snel oplopen tot vele minuten extra tijd tijdens het testen, afhankelijk van hoeveel images er benodigd zijn. **Lisiting 6** Dit stuk code instantieert onze Entity Framework Core DbContext met connectionstring van onze Testcontainer. Hiermee kunnen we vervolgens onze SpecFlow test onveranderd uitvoeren met een kopie van onze live database, die we conform AVG wetgeving hebben geanonimiseerd, in plaats een mock. De test zal uiteraard falen omdat we niet een juist SQL commando proberen uit te voeren, maar hiermee hebben we de bug gevonden voordat hij naar productie ging.

Testontainers als testomgeving

Met Testcontainers kunnen we meer dan slechts een enkele container opstarten, we kunnen er in principe alles mee wat Docker ondersteunt. Als voorbeeld hiervan gaan we ook de UI testen van onze applicatie als Testcontainers opzetten. De volledige broncode van dit voorbeeld is te vinden op: https://github.com/XPRTZ/testcontainers-workshop

Allereerst moeten we er voor zorgen dat onze webapplicatie als Docker container beschikbaar is om te gebruiken, dit kan door simpelweg een dockerfile in de directory te plaatsen waar ook de solution file staat. Vervolgens creëren we een class die de interface IDockerImage van Testcontainers en de interface IAsyncLifetime van XUnit implementeert. In de functie "InitializeAsync" voegen we vervolgens deze code toe: **Lisiting 7** Met behulp van deze class hebben we nu een Testcontainer image

7
```
public async Task InitializeAsync()
{
    await _semaphoreSlim.WaitAsync();

    try
    {
        await new ImageFromDockerfileBuilder()
            .WithName(this)
            .WithDockerfileDirectory(CommonDirectoryPath.GetSolutionDirectory(), string.Empty)
            .WithDockerfile("Dockerfile")
            .WithBuildArgument("RESOURCE_REAPER_SESSION_ID", ResourceReaper.DefaultSessionId.ToString("D"))
        .WithDeleteIfExists(false)
        .Build();
    }
    finally
    {
        _semaphoreSlim.Release();
    }
}
```

8
```
_demoAppNetwork = new TestcontainersNetworkBuilder()
          .WithName(Guid.NewGuid().ToString("D"))
          .Build();
```

9
```
new TestcontainersBuilder<TestcontainersContainer>()
          .WithImage(Image)
          .WithNetwork(_demoAppNetwork)
          .WithPortBinding(DemoAppImage.HttpsPort, true)
          .WithEnvironment("ASPNETCORE_URLS", "https://+")
          .WithEnvironment("ASPNETCORE_Kestrel__Certificates__Default__Path", DemoAppImage.CertificateFilePath)
          .WithEnvironment("ASPNETCORE_Kestrel__Certificates__Default__Password", DemoAppImage.CertificatePassword)
          .WithEnvironment("ConnectionStrings__StoreConnectionString", connectionString)
          .WithWaitStrategy(Wait.ForUnixContainer().UntilPortIsAvailable(DemoAppImage.HttpsPort))
          .Build();
```

10
```
public async Task Get_Accounts_Should_Return_100_Pages_Of_Accounts()
{
    // Arrange
    string ScreenshotFileName() => $"{nameof(Get_Accounts_Should_Return_100_Pages_Of_Accounts)}_{DateTimeOffset.UtcNow.ToUnixTime-
Milliseconds()}.png";

    using var chrome = new ChromeDriver(_chromeOptions);

    // Act
    chrome.Navigate().GoToUrl(_demoAppContainer.BaseAddress);

        chrome.GetScreenshot().SaveAsFile(Path.Combine(CommonDirectoryPath.GetProjectDirectory().DirectoryPath, ScreenshotFileNa-
me()));

        chrome.FindElement(By.Id("accounts_link")).Click();

        await Task.Delay(TimeSpan.FromSeconds(5));

    chrome.GetScreenshot().SaveAsFile(Path.Combine(CommonDirectoryPath.GetProjectDirectory().DirectoryPath, ScreenshotFileNa-
me()));

        // Assert
        var span = chrome.FindElement(By.ClassName("pager-display")).FindElement(By.TagName("span"));

        span.Text.Should().Be("1 of 100");
}
```

gedefinieerd welke we nu kunnen gebruiken in onze testen. Het stuk code voor het initialiseren van de database container breiden we uit met het statement "WithNetworkAliases" wat er voor zorgt dat we de database niet alleen kunnen benaderen met een ip-adres maar ook met een hostnaam.
Vervolgens creëren we een Docker network zodat de twee containers met elkaar kunnen communiceren: **Lisiting 8**
Als laatste stap creëren we de Testcontainer voor de frontend: **Listing 9**
Nu we alle onderdelen hebben kunnen we deze stack gebruiken in onze testen: **Listing 10**

Door het runnen van deze test worden de volgende stappen uitgevoerd:
> Er wordt een network aangemaakt
> De database wordt opgestart, gerestored en aangemeld op het netwerk
> De frontend wordt gecompileerd en er wordt een binary van gemaakt
> De frontend wordt opgestart en aangemeld op het netwerk
> De automatische testen worden uitgevoerd
> Het network, de database en de frontend worden opgeruimd

Hiermee hebben wij dus on-the-fly een testomgeving gecreëerd waarin we representatieve testen kunnen uitvoeren!
Deze test duurt echter wel langer dan de voorgaande test, namelijk 20 a 30 seconden in totaal. Deze extra tijd zit voornamelijk in het aanmaken van een eigen image en het opstarten van de extra container. Eenmaal gestart verlopen de testen net zo snel alsof we alle afhankelijkheden hadden weggemockt. Om deze reden is het dan ook zeer aan te raden om hier gebruik te maken van één Testcontainer set per logische set aan testen.
Een andere punt van aandacht is het cachen van Docker images: indien gebruik wordt gemaakt van een CI/CD pipeline kan het gebeuren dat bij iedere run de images opnieuw worden gedownload wat resulteert in tests die, afhankelijk van de netwerksnelheid, een flink aantal minuten langer kunnen duren.

Tot slot

Met behulp van testcontainers is het mogelijk om volledig geautomatiseerd een complete en complexe testomgeving op te zetten. Deze kan vervolgens gebruikt worden om geautomatiseerde testgevallen uit te voeren die de applicatie in een "productie like" omgeving testen. Met een integratie in een CI/CD pipeline is het vervolgens mogelijk om deze testgevallen al in een vroeg stadium uit te voeren en kunnen we al veel eerder zien hoe onze applicatie zich gedraagt wanneer deze op een live omgeving deployed is.
En hiermee komen we tot een belangrijke vraag: hebben we nog wel een testomgeving nodig?
We hebben met Testcontainers de mogelijkheid om een complete testomgeving geautomatiseerd op te zetten en deze alleen gebruiken wanneer we ook daadwerkelijk testen. Hiermee kunnen we voorspelbaarder en goedkoper testen: we hebben tenslotte de omgeving alleen "aan" staan wanneer hij in gebruik is en hebben volledige controle over de data die aanwezig is in de omgeving.
Ik denk dat de tijd is aangebroken om van de klassieke "ontwikkel, test, acceptatie en productie" omgevingen over te stappen op "ontwikkel, acceptatie en productie" omgevingen. Wie doet er mee? ●

Een andere punt van aandacht is het cachen van Docker images

BIO

Rob van Geloven
Met bijna twintig jaar aan ervaring in de software industrie heb ik een veelvoud aan rollen gehad, van technisch applicatiebeheer tot CTO en van DevOps engineer tot technisch tester. Tegenwoordig ben ik meestal werkzaam als een software architect met een speciale focus op de .NET stack. Mijn belangrijkste drijfveer is om mensen en bedrijven te helpen van goed tot beter te gaan: hoe mensen samenwerken, hoe code wordt ontwikkeld en getest en hoe uiteindelijk deze software wordt opgevelerd aan klanten.


Hybrid work teaches you to listen better
Discover IT & the digital way of working at **rabobank.jobs**

Rabobank

.NET 7

Auteur: Roelant Dieben

Begin november is er weer een nieuwe versie van .NET gereleased. Versie 7 van .NET is een current version, waarbij de support dus korter is dan die van een Long-Term Support (LTS) versie. Met de 18 maanden support die we op .NET 7 krijgen, loopt deze dus een half jaar eerder af dan de support op .NET 6. Dit betekent niet dat je deze versie van .NET links moet laten liggen, want er zijn weer tal van verbeteringen en optimalisaties doorgevoerd. In dit artikel neem ik je mee door de belangrijkste.

Compilerverbeteringen

Als ontwikkelaar maak ik graag gebruik van een hogere abstractie taal als C# en het bijbehorende .NET platform, zodat ik me geen zorgen hoef te maken over tal van zaken rond de compiler en de runtime. Hierdoor voelen verbeteringen aan de compiler soms ook minder relevant, terwijl hier op een dergelijk niveau vaak de grootste winst ten aanzien van performance wordt geboekt. Ook in .NET 7 zitten weer tal van deze verbeteringen, zoals *Native Ahead-of-Time (AoT) compilation*, welke het mogelijk maakt om ook op omgevingen uit te rollen, waar *Just-in-Time (JiT) compilation* niet wordt ondersteund. Daarnaast starten je applicaties ook nog eens sneller. Naast tal van andere compiler verbeteringen rond geheugenallocatie, ondersteuning van diverse runtimes, algoritme optimalisatie, veiligheidsupdates en verbeteringen aan de *command-line interface (CLI)*, zijn er ook concretere verbeteringen te ontdekken in .NET 7 die een meer functionele impact hebben.

DateTime.Nanosecond

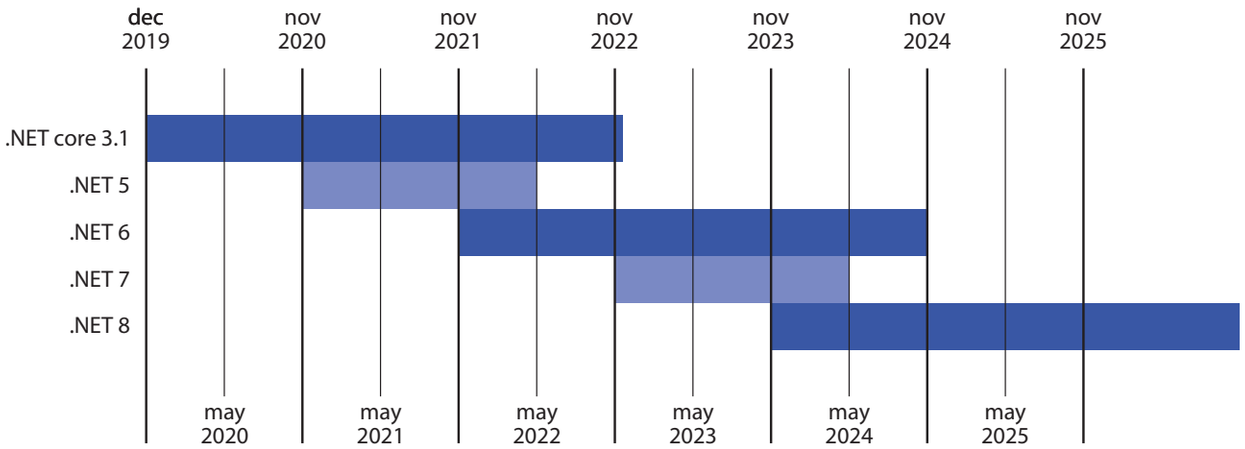
Het DateTime type wordt uitgebreid met properties die meer zeggen dan het abstracte Ticks en ook nog met een grotere nauwkeurigheid. Vanaf .NET 7 worden namelijk ook micro- en nanoseconden ondersteund.

JSON serialisatie

Jarenlang werd de discussie Newtonsoft.Json vs System.Text.Json

.NET 7 is een current version met 18 maanden support

.NET support windows



met bijna net zoveel passie gevoerd als de discussie tabs vs spaties. Fervente voorvechters van beide bibliotheken kwamen met allerhande voors en tegens, waarbij vaak het doorslaggevende argument was dat System.Text.Json niet *feature complete* was. Met .NET 7 worden er een aantal veelgevraagde en langverwachte features toegevoegd aan System.Text.Json zoals *polymorphism* en meer controle over het niveau en opties van serialisatie. Hoewel *feature parity* geen doel op zich is voor het System.Text.Json team, worden de scenarios die enkel in Newtonsoft.Json worden ondersteund wel steeds exotischer.

Generic math

In .NET 7 is het door de *feature static virtual members* in interfaces mogelijk om generieke implementaties te schrijven waarbij de input wordt beperkt tot numerieke waarden en je dus niet voor elk numeriek type een vergelijkbare implementatie hoeft te schrijven, maar dit op kan lossen middels *generics*. Waar je dus voorheen voor een double, een integer en een decimal drie verschillende implementaties moest schrijven, kan je nu af met: **Afbeelding 1**

C# 11 – Syntactic sugar

Zoals gebruikelijk worden er ook tal van C# verbeteringen doorgevoerd in de nieuwe versie van .NET. Veel van deze verbeteringen zitten in de sfeer van *syntactic sugar*, zoals generieke attributen, automatische initialisatie van *defaults* in *structs* en de ondersteuning van *new line breaks* in *string interpolation* zonder deze expliciet te escapen. **Afbeelding 2**

C#11 – List patterns

Een nieuwe functionaliteit die verder gaat dan slechts *syntactic sugar* is het kunnen matchen op *list patterns*. Hierbij kunnen we voor een array van [1, 2, 3]:

1
```
static T Add<T>(T left, T right) where T : INumber<T> => left + right;
```

2
```
public class GenericAttribute<T> : Attribute { }
```

3
```
Console.WriteLine(new[] { 1, 2, 3 } is [1, 2, 3]); // True
Console.WriteLine(new[] { 1, 2, 3 } is [1, 2, 4]); // False
Console.WriteLine(new[] { 1, 2, 3 } is [1, 2, 3, 4]); // False
```

4
```
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]); // True
```

5
```
Console.WriteLine(new[] { 1, 2, 3 } is [0 or 1, <= 2, >= 3]); // True
```

6
```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ..]); // True
```

exact matchen op een *array*, **Afbeelding 3**
matchen op een *sequence*, **Afbeelding 4**
generiek matchen middels een discard pattern. **Afbeelding 5**
aan het begin of einde van een array matchen middels het slice pattern. **Afbeelding 6**

C#11 – Spans to rule them all (or at least for System.LINQ)

Binnen C# 11 is de support voor Span weer verbeterd. De Span, of voluit de System.Span<T>, zit er al sinds C# 7.2 in, maar in de praktijk lijkt nog niet iedereen de Span te hebben omarmd. Doordat Spans op de *stack* leven en niet op de *managed heap* kan de compiler hierop optimaliseren en is het aanzienlijke malen sneller dan collecties op de *heap*. Daarnaast bespaar je uiteraard ook rekenkracht doordat er geen *garbage collection* nodig is voor de *stack*, dus ook niet voor de Span<T>. Dit voordeel is ook direct een potentieel nadeel, want het geheugen wat tijdens compileren wordt gealloceerd, blijft gealloceerd. Er zitten daarnaast ook beperkingen aan het leven op de *stack*. Er is geen *boxing* mogelijk, waardoor code die onderliggend leunt op *reflection* geen gebruik kan maken van Spans, aangezien daar *boxing* nodig is. Ook het gebruik in asynchrone code, of in lambdas, is hierdoor niet altijd mogelijk. De afgelopen vijf jaar, sinds de introductie van Spans, zien we ondanks deze uitdagingen de Span steeds

```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22000.1098/21H2)
Intel Core i9-9900KF CPU 3.60GHz (Coffee Lake), 1 CPU, 16 logical and 8 physical cores
.NET SDK=7.0.100-rc.2.22454.1
  [Host]   : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2  [AttachedDebugger]
  .NET 6.0 : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2
  .NET 7.0 : .NET 7.0.0 (7.0.22.42212), X64 RyuJIT AVX2


|  Method |      Job |  Runtime | Size |      Mean |     Error |    StdDev | Allocated |
|-------- |--------- |--------- |----- |----------:|----------:|----------:|----------:|
|     Min | .NET 6.0 | .NET 6.0 |  250 | 966.55 ns | 13.802 ns | 12.235 ns |      32 B |
|     Max | .NET 6.0 | .NET 6.0 |  250 | 952.08 ns | 15.918 ns | 14.111 ns |      32 B |
| Average | .NET 6.0 | .NET 6.0 |  250 | 924.48 ns | 18.498 ns | 32.881 ns |      32 B |
|     Sum | .NET 6.0 | .NET 6.0 |  250 | 961.15 ns | 19.048 ns | 24.767 ns |      32 B |
|     Min | .NET 7.0 | .NET 7.0 |  250 |  26.33 ns |  0.540 ns |  0.554 ns |         - |
|     Max | .NET 7.0 | .NET 7.0 |  250 |  25.86 ns |  0.502 ns |  0.469 ns |         - |
| Average | .NET 7.0 | .NET 7.0 |  250 |  31.36 ns |  0.637 ns |  0.828 ns |         - |
|     Sum | .NET 7.0 | .NET 7.0 |  250 |  84.28 ns |  1.606 ns |  1.502 ns |         - |
```

Figuur 1 Benchmark LINQ .NET 6 vs .NET 7

BIO

Roelant Dieben

Roelant is Microsoft Azure MVP en oprichter van XPRTZ.cloud. Zijn roots liggen in de software engineering en cloud architectuur en hij heeft een gezonde passie voor AI en machine learning. Ook is hij betrokken bij diverse communities als SDN, DevNetNoord en Azure Thursday.

vaker terugkomen in de interne methodes van C#, waarbij we ongemerkt onder de motorkap dus toch gebruik maken van de hierboven genoemde voordelen. In C# 11 zorgt deze verbeterde support voor een potentiële *gamechanger*, namelijk het gebruik in System.LINQ.
Sinds C# 3 ben ik al groot voorstander van het gebruik van *Language INtegrated Queries (LINQ)* die in mijn ogen bijna alle code leesbaarder en beter onderhoudbaar kan maken. Aan die leesbaar- en onderhoudbaarheid hing eigenlijk altijd het prijskaartje van een verminderde performance. Wilde je een goede performance, dan was het devies om vooral geen LINQ te gebruiken.

Voor een aantal operaties in LINQ is deze matige performance nu verleden tijd. Door het onderliggende en interne gebruik van spans, is de performance van LINQ nu vele malen beter geworden. We hebben het dan niet over 2 keer zo snel, of 10 keer zo snel, maar onder de juiste omstandigheden wel 40 keer zo snel. Daarnaast is de geheugenconsumptie, door het gebruik van de *stack*, vrijwel nihil.
De kans is groot dat met de volgende .NET en C# versies de ondersteuning van spans verbeterd blijft worden, dus de volgende keer dat je een collectie in C# nodig hebt, overweeg dan ook een keer een span en laat LINQ niet links liggen omwille van performance. ●

LINQ maakt code veel leesbaarder en beter onderhoudbaar

Infrastructure-as-Code

Het is 2002, je bent programmeur. Na maanden zwoegen ben je eindelijk zover om dat stuk software uit te rollen naar productie. Je brandt een gecompileerde versie, inclusief een MSI installatiebestand en uitgebreide installatie documentatie op een CD waarop je met een marker hebt geschreven: "versie 1.0". Je loopt langs bij de systeembeheer afdeling om de CD af te geven en bij het afgeven krijg je de terugkoppeling dat ze pas over een week tijd hebben om de software uit te rollen naar de server en alle werkstations. Na een week krijg je een telefoontje van systeembeheer met de boodschap dat ze niet genoeg capaciteit hebben op de server. Daarom hebben ze een nieuwe server besteld waarop een levertijd van ongeveer een maand zit! Enzovoort, enzovoort.

Auteur: Patrick Vroegh

Fast-forward naar 2022, je bent software ontwikkelaar. Je hebt zojuist een nieuw stukje functionaliteit ontwikkeld. Alle testen slagen lokaal, je 'commit' de code en 'pushed' deze naar de Azure DevOps omgeving. Je gaat koffie halen en voordat je je eerste slok neemt, is de nieuwe functionaliteit uitgerold en wereldwijd beschikbaar voor de eindgebruiker.
We zijn van ver gekomen en in die 20 jaar is de gereedschapskist van de software ontwikkelaar steeds meer voorzien van indrukwekkende hulpmiddelen waarvan we 20 jaar geleden niet eens wisten dat het mogelijk zou zijn. In deze blog zoomen we graag in op één van deze hulpmiddelen en dat is **Infrastructure-as-Code.**

Definitie

Wat is Infrastructure-as-Code precies? In principe dekt de term al veel lading, maar een stukje extra definitie is nog wel benodigd:
Infrastructure-as-Code is het proces voor het opzetten en onderhouden van infrastructuur middels declaratieve of imperatieve code, op een geautomatiseerde manier in plaats van handmatig.
Vooral de termen 'code' en 'geautomatiseerd' zijn de sleutel in de definitie. In de code wordt exact gespecificeerd door de ontwikkelaar waaraan de infrastructuur moet voldoen (declaratief) of wordt geprogrammeerd welke stappen uitgevoerd moeten worden om de

Infrastructure-as-Code is het proces voor het opzetten en onderhouden van infrastructuur middels code; geautomatiseerd in plaats van handmatig.

ARM staat voor Azure Resource Manager en is het mechanisme voor automatisch opzetten en afbreken van resources in Azure.

infrastructuur op te zetten (imperatief). Deze code wordt op een geautomatiseerde manier geïnterpreteerd en uitgevoerd, met als resultaat functionerende infrastructuur die toegespitst is op de vooraf gestelde requirements.
Aangezien de declaratieve or imperatieve code leesbare tekst is, is deze makkelijk op te slaan in een Sourcecode Repository, met alle gemakken daarvan zoals versiebeheer, de mogelijkheid om gemakkelijk samen te werken op dezelfde code en dat elke versie van je applicatie onlosmakelijk verbonden is met een versie van de infrastructuur.
Dit heeft deuren geopend om Infrastructure-as-Code te integreren met bestaande CI/CD (Continuous Integration/Continuous Deployment) processen.

Geschiedenis

In principe verklappen de twee inleidende verhaaltjes waar Infrastructure-as-Code vandaan komt. Het is onder andere ontstaan uit een behoefte om sneller feedback te krijgen, een snellere time-to-market te hebben en minder te hoeven bekommeren over de infrastructuur. Verder hebben virtualisatie van computers en applicaties ook een grote rol gespeeld in de opkomst van Infrastructure-as-Code, vanwege het feit dat het opzetten van virtuele servers en/of applicaties zich heel goed laat automatiseren. Dat in tegenstelling van het onderhouden van fysieke hardware waar de software zonder virtualisatie op draait. Het fysiek uitbreiden van hardware is toch echt een handmatige klus.
De opkomst van Cloud Computing heeft Infrastructure-as-Code verder op de kaart gezet. Bij alle grote Cloud platforms speelt Infrastructure-as-Code een grote rol en is het dan ook goed geïmplementeerd.

De rol in DevOps

Infrastructure-as-Code speelt een belangrijke rol in een DevOps team wanneer er gebruik gemaakt wordt van virtuele hardware. Door deze code en/of definitie bestanden op te nemen in de Sourcecode Repository en in de CI/CD pipelines, wordt de feedback-loop kleiner gemaakt. Dit komt omdat de infrastructuur op een consistente en herhaalbare manier geautomatiseerd in wordt voorzien. Aangezien er geen handen aan te pas komen, maakt dit de tijdslijnen van idee tot implementatie veel korter. Hierdoor krijg je dus veel snellere terugkoppeling of het idee ook daadwerkelijk een goed idee blijkt te zijn.

Declaratief versus imperatief

De code die gebruikt wordt om in de infrastructuur te voorzien heb je in twee smaken, declaratief en imperatief. Imperatief is als een recept voor het maken van bijvoorbeeld een chocolade cake; het bevat de uitgebreide instructies om de cake te maken. Declaratief is als een boodschappenlijstje waarop staat: “Koop 500gr Chocolade Cake”.
Je zou kunnen zeggen dat imperatieve code de complexiteit van het opzetten van de infrastructuur (de chocolade cake) deels bij de ontwikkelaar legt, waarbij de complexiteit bij declaratieve code wordt verlegd naar de tooling die gebruikt wordt om de code te interpreteren en uit te voeren.
Imperatief geeft meer controle over hoe in de infrastructuur wordt voorzien, waarbij je bij declaratief weinig tot geen invloed hebt over hoe dit gedaan wordt; je hoeft slechts aan te geven wat je wilt hebben en hoe dit gedaan wordt, is onbelangrijk voor de ontwikkelaar.

Tools

De code die de infrastructuur definieert is niet het enige wat nodig is om Infrastructure-as-Code mogelijk te maken. Er zijn ook tools nodig die deze code kunnen lezen en uitvoe-

BIO

Patrick Vroegh

Patrick Vroegh is Technical Lead Consultant voor Bergler Software Solutions. Hij helpt organisaties bij het toekomstbestendig maken van hun software en ontwikkelprocessen.

ren. Gelukkig zijn er genoeg tools beschikbaar die dit mogelijk maken. Hieronder staat een (niet definitieve) lijst met populaire tools, inclusief een code voorbeeld.

ARM

ARM staat voor Azure Resource Manager en is het mechanisme van Microsoft's Cloud platform voor het automatisch opzetten (maar ook afbreken) van resources in Azure. Waar ARM de tool is, zijn ARM templates de code waarin gedefinieerd wordt hoe de infrastructuur er uit moet komen te zien. ARM templates zijn declaratief van aard, dus geven de gewenste staat van de infrastructuur aan. **Figuur 1**

Het grote voordeel van ARM templates is dat elke Azure resource tot in elk kleinste detail gespecificeerd kan worden. Dit is overigens ook tegelijkertijd het grootste nadeel; ARM templates zijn vaak langdradig en moeilijk te lezen. Om dit nadeel te overwinnen heeft Microsoft Bicep geïntroduceerd.

Meer informatie over ARM is hier te vinden: https://tinyurl.com/ycxfjc2w.

Bicep

Bicep is een feitelijk een vertaler tussen de Bicep syntax naar ARM templates en heeft als doel om de Infastructure-as-Code veel minder langdradig en dus beter leesbaar te maken. In tegenstelling tot ARM templates is het ook mogelijk in Bicep om control-of-flow structuren zoals if- en for-statements te gebruiken. De Bicep syntax is declaratief van aard. Wanneer een Bicep file uitgerold word, wordt het eerst vertaald naar een ARM template om deze vervolgens door de ARM uit te laten voeren. **Figuur 2**

```
param location string = resourceGroup().location
param storageCount int = 2

resource storageAcct 'Microsoft.Storage/storageAccounts@2021-06-01' = [for i in range(0, storageCount): {
  name: '${i}storage${uniqueString(resourceGroup().id)}'
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'Storage'
}]

output storageInfo array = [for i in range(0, storageCount): {
  id: storageAcct[i].id
  blobEndpoint: storageAcct[i].properties.primaryEndpoints.blob
  status: storageAcct[i].properties.statusOfPrimary
}]
```

2

Bicep syntax voorbeeld. Meer information over Bicep is hier te vinden: https://tinyurl.com/3vwvcczt.

```
"user strict";
const pulumi = require("@pulumi/pulumi");
const aws = require("@pulumi/aws");

const group = new aws.ec2.SecurityGroup("web-sg", {
    description: "Enable HTTP access",
    ingress: [{ protocol: "tcp", fromPort: 80, toPort: 80, cidrBlocks: ["0.0.0.0/0"] }],
});

const server = new aws.ec2.Instance("web-server", {
    ami: "ami-6869aa05",
    instanceType: "t2.micro",
    vpcSecurityGroupIds: [ group.name ], // reference the security group resource above
});

export const publicIp = server.publicIp;
export const publicDns = server.publicDns;
```

3

JavaScript code dat gebruikt maakt van de Pulumi API Meer informatie over Pulumi is te vinden op: https://tinyurl.com/5662f24n.

Pulumi

Pulumi is een vreemde eend in de bijt als het gaat om tools voor Infrastructure-as-Code. Het heeft geen eigen syntax definitie, maar maakt juist gebruik van bestaande programmeertalen. Dit heeft als voordeel dat ontwikkelaars hun bestaande kennis van hun favoriete programmeertaal kunnen hergebruiken om in de infrastructuur te voorzien.

Pulumi gebruikt eigenlijk een mix van declaratieve en imperatieve Infrastructure-as-Code, juist vanwege het feit dat het bestaande programmeertalen gebruikt. Hierdoor kun je je eigen logica implementeren in de code, maar je kunt ook gewoon definiëren hoe in de infrastructuur voorzien moet worden. **Figuur 3**

Tot slot

Infrastructure-as-Code is een waardevolle tool in de gereedschapskist van een software ontwikkelaar. Het brengt de werelden van het ontwikkelen van software en het operationele aspect van het uitvoeren van software dichter bij elkaar. ●

```
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2018-07-01",
    "name": "[concat('storage', uniqueString(resourceGroup().id))]",
    "comments": "Storage account used to store VM disks",
    "location": "[parameters('location')]",
    "metadata": {
      "comments": "These tags are needed for policy compliance."
    },
    "tags": {
      "Dept": "[parameters('deptName')]",
      "Environment": "[parameters('environment')]"
    },
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "Storage",
    "properties": {}
  }
]
```

1

ARM Template voorbeeld


"Wij zijn Vitas Young Talent! We hebben een geweldig opleidingsprogramma en bijbehorende intensieve begeleiding, binnen het ICT-werkveld, voor pas afgestudeerden en Young Professionals."
Vitas Young Talent Programma
Het doel van het programma is om jou met veel vertrouwen, enthousiasme en werkplezier een prachtige start van je carrière te laten maken.
In onze eigen Vitas Academy stellen we samen je trainingsdoelen op en bereiden je voor op het werken binnen het domein van onze opdrachtgevers.
VOOR DEELNEMERS
De inzet van de Vitas Academy
Behalen van 2 vakcertificeringen
Communicatietrainingen en trainingen gericht op samenwerking
En meer: vertellen we graag onder het genot van een kop koffie!
VOOR WERKGEVERS
Versnelde aansluiting van benodigde kennis
Regie en begeleiding vanuit Vitas Young Talent georganiseerd
Begeleiding vanuit een HR-invalshoek en vanuit de inhoud
En meer: vertellen we graag onder het genot van een kop koffie!
SCAN DE QR CODE VOOR MEER INFORMATIE EN SOLLICITEER DIRECT
MICROSOFT
ANGULAR
AZURE
.NET
C#
VITAS

Add KeyVault secrets to an Azure App Configuration in Bicep

Auteur: Thomas Vieveen

Since security is becoming an increasingly important topic nowadays, I figured.. Why not write an article about it? In this article I am going to explain how you can remove all important (generated) secrets, such as connection strings, access keys, and more from your appsettings files. Instead, we're going to import them straight into the Key Vault as soon as they roll out of the infrastructure as code (IaC) templates.

Assumptions

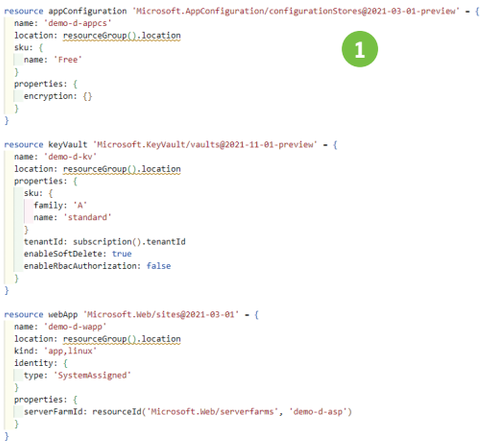
Before we continue, I want to define some assumptions. These are the things I expect you have already set up.

- An Azure environment + credits
- You have a Web App or Function already hooked up to Azure App Config + Azure Key Vault
- You are currently deploying your IaC with Bicep

If you don't have Azure App Configuration set up yet, visit the Azure App Configuration documentation page, or take a look at some examples provided by Microsoft.

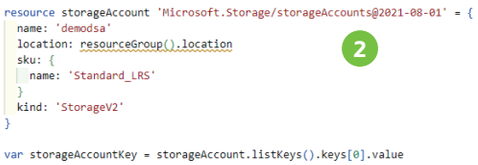
If you need an introduction to Bicep, please take a look at this introduction on Microsoft Learn, or consider buying my colleague Eduard Keiholz his book about Azure Infrastructure as Code.

Okay, let's dive into the Bicep templates. I'd assume your master template looks something like this. There should at least be a resource or template for an App Service, an App Configuration and a Key Vault. Take a look at Example 1. In this article I will not explain how to set up access control between these resources. **Example 1 – application.bicep (part 1)**



```
resource appConfiguration 'Microsoft.AppConfiguration/configurationStores@2021-03-01-preview' = {
  name: 'demo-d-appcs'
  location: resourceGroup().location
  sku: {
    name: 'Free'
  }
  properties: {
    encryption: {}
  }
}

resource keyVault 'Microsoft.KeyVault/vaults@2021-11-01-preview' = {
  name: 'demo-d-kv'
  location: resourceGroup().location
  properties: {
    sku: {
      family: 'A'
      name: 'standard'
    }
    tenantId: subscription().tenantId
    enableSoftDelete: true
    enableRbacAuthorization: false
  }
}

resource webApp 'Microsoft.Web/sites@2021-03-01' = {
  name: 'demo-d-wapp'
  location: resourceGroup().location
  kind: 'app,linux'
  identity: {
    type: 'SystemAssigned'
  }
  properties: {
    serverFarmId: resourceId('Microsoft.Web/serverfarms', 'demo-d-asp')
  }
}
```

1



```
resource storageAccount 'Microsoft.Storage/storageAccounts@2021-08-01' = {
  name: 'demodsa'
  location: resourceGroup().location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
}

var storageAccountKey = storageAccount.listKeys().keys[0].value
```

2

Adding a storage account

For example, you decide you want to add a storage account to store some pictures and documents uploaded by your users. You don't want your developers throwing the access keys around, since the storage account contains sensitive data. The access key should be added to the Key Vault, without you (the developer) ever knowing what the value is.

First, we create a storage account using Bicep. We can then access the access key by using the code on the last line of Example 2. This is the value that we want to add to our Azure Key Vault. **Example 2**

Adding the value to Key Vault

I have to admit this took the most time. Lucky for you, I invested the time so you don't have to! I've created a simple template that allows you to add a single key-value pair to your Key Vault. You just need to provide the keyVault name, key, and value.

Pay attention when inserting secrets from Bicep

Adding KeyVault reference to Azure App Configuration must be done in an object with a uri property.

3

Example 4 – secrets.bicep

```
param keyVaultName string

@secure()
param keyVaultValue string
param keyVaultKey string

resource KeyVaultSecret 'Microsoft.KeyVault/vaults/secrets@2021-11-01-preview' = {
  name: '${keyVaultName}/${keyVaultKey}'
  properties: {
    value: keyVaultValue
  }
}

var keyVaultRef = {
  uri: KeyVaultSecret.properties.secretUri
}

output keyVaultUri object = keyVaultRef
```

Potential security risk!

Be wary, all output variables are listed in the 'Deployments' blade in your resource group in Azure. Make sure you don't enter any real secrets here or you'll still be running a security risk.

4

```
resource storageAccount 'Microsoft.Storage/storageAccounts@2021-08-01' = {
  name: 'demodsa'
  location: resourceGroup().location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
}

module storageKeyKeyVault 'templates/secrets.bicep' = {
  name: 'StorageKeyKeyVault'
  params: {
    keyVaultName: keyVault.name
    keyVaultKey: 'StorageAccessKey'
    keyVaultValue: storageAccount.listKeys().keys[0].value
  }
}

module appConfigurationKeyValues 'templates/singleKeyValue.bicep' = {
  name: 'KeyValues'
  params: {
    appConfigName: appConfiguration.name
    appConfigKey: 'Storage:AccessKey'
    appConfigValue: storageKeyKeyVault.outputs.keyVaultUri
    contentType: 'application/vnd.microsoft.appconfig.keyvaultref+json;charset=utf-8'
  }
}
```

5

The important lesson-learned is at the bottom of Example 4. Take a look at 'var keyVaultRef'. Whenever we're adding a KeyVault reference to Azure App Configuration, Azure requires it to be in an object with a uri property. If you forget this object, your reference to Key Vault will not work and your application can crash. As you can see in Example 3, the first reference is invalid. The other two are correct. You can copy-and-paste the template in **Example 4** to your own project. **Example 3**

Preventing potential security risks

Potential security risks arise when you starting using templates to create your Azure resources in. I get that you don't want to put all your Azure resource creations in one single file. That would become a total mess on bigger projects. However, when you want to output a secret variable back to your main template, these variables show up in your Outputs blade in Azure. Fixing this issue is actually quite simple. In the template that we want to access the secret, we can reference this resource using the '**existing**' keyword We can then retrieve the keys without passing them back through the output variables.

This security risk only occurs when outputting variables back to the template they're called from. However, when you need to pass values on to the templates you want to use, these also show up in Azure under the Inputs blade. Luckily, Bicep already has you covered. We can simply add a '@secure()' above the parameter, and the variable won't show up in your Inputs blade anymore (Example 4).

Adding the Key Vault reference to Azure App Configuration

Now, for the final part. We're going to add our reference to App Config. With the following template, we can decide whether to add a normal string value, or a key vault reference to our App Configuration.

How will it look when everything is put together?

In Example 5, I will show you the code that we can add to Example 1 to make it complete. You can start expanding this simple example in your own project. You'll get rid of your secrets in no-time! I hope you enjoyed reading my article and that I inspired you to start implementing this into your own projects!

Example 5 – application.bicep (part 2) ●

BIO

Thomas Vieveen

Thomas werkt bij 4DotNet als DevOps engineer. Hij onderzoekt graag nieuwe Azure en .NET features. Hij haalt plezier uit het efficiënter maken van de processen bij zijn opdrachtgevers, zoals het neerzetten van een goede CI-CD pipeline. Buiten zijn werk is hij graag aan het ondernemen en ontwikkelt hij zich in IT/business architectuur.

Mutation testing in .NET with Stryker

Many of us nowadays use unit tests to automatically check the quality of our code. The great thing about unit testing is that we have found a way to automate manual testing. Automated tests have a lot of advantages over manual testing. They are way faster to run, repeatable and all the developers can run those tests. But have you ever considered that unit tests are just code.

Auteur: Yaël Keemink

Which raises the question, "if unit testing is for ensuring the quality of my code and my unit tests is code, how do I know the quality of my testing code?"
That is what code coverage is for right? Why would you need mutation testing when you already have a 90% or higher code coverage. Code coverage tells you if you have tested all your code paths right? Well, that is not entirely true. Code coverage only tells you if your code has been hit by a test. Not if your test actually tests your code. That is where mutation testing will come into play, mutation testing will tell you if you have tests that... test your code.

How does it work

It probably all sounds a bit abstract, but stay with me. I will try my best to explain how it works, what tool to use for .NET and how easy it is to do mutation testing.
Mutation testing will start off by running all your unit tests to verify all your tests succeed. After it has done this the mutation test tooling will try to break your tests. It will do so by inserting bugs, or mutants as they are called, in your code. When these mutants are introduced at least one of your tests should fail. If no tests fail it is an indication your tests might not be sufficient.
Let's put that into practice, consider the following code with unit test:

Lisiting 1

```
namespace Demo

{
      public class DemoCode
      {
        public int AddNumbers(int v1, int v2)
        {
                return v1 + v2;
        }
      }

      [TestClass]
      public class DemoTest
      {
        [TestMethod]
        public void MyTestMethod()
        {
                // Arrange
                var sut = new DemoCode();

                // Act
                int result = sut.AddNumbers(1, 1);

                 //Assert
                Assert.AreEqual(result, result);
        }
       }
}
```

1

The class DemoCode has one method which will add numbers and return the sum. Our test method however will compare the result to the result and therefore will always return true. When we run our test it will tell us our code works as intended.
Now we run our mutation test, this will alter the code of our AddNumbers method as follows.

Lisiting 2

Notice that the '+' operator has changed to the '-' operator. The problem we now face is that our test will still pass while we are now subtracting instead of adding the numbers. This means our test is not sufficient to the purpose of the method and bugs can be easily introduced.
Mutations can be done on multiple different operators and value types. It can flip Booleans, change the number of an integer, empty a string or even declare a variable NULL. It also works on collections and even regex statements.

Who ensures the quality of the mutation testing code

You are probably wondering who will ensure the quality of the

> Vind deze en de andere listingen uit het artikel op sdn.nl!

```
public class DemoCode
{
        public int AddNumbers(int v1, int v2)
        {
                return v1 - v2;
        }
}
```

2

mutation testing framework since that too is still code. Well, unit tests and mutation tests do. The wonderful thing about mutation testing is that with it we are finally able to close the testing loop. The Stryker team and community write unit tests, integration tests and are running Stryker to test Stryker.

Use Stryker for .NET

Now the moment you have been waiting for, or just skipped to because you just wanted to know which tool to use. For .NET I would recommend to use Stryker as your mutation testing tool. Stryker is an open source tool which is easy to install, use locally and implement in your pipeline. You can also use Stryker for Scala and Javascript.

BIO

Yaël Keemink

Yaël is a snowboarding software engineer who loves to automate everything he has to do twice. He has been automating tedious jobs like manual testing and updating dependencies since 2017. Yaël is a critical thinker and is always looking to improve the way of working in his team.
At Info Support he has worked as a DevOps engineer and is now employed at Ordina.
Even though he is still at the beginning of his career Yaël has already had a big impact on the projects he has worked on by always asking questions.
It is only by asking the right questions, you can improve yourself and your team.

> For .NET I would recommend to use Stryker as your mutation testing tool.

All nice and well, but you want to know how to actually use Stryker. It is quite easy, first you will have to install Stryker, you can do so with the 'install stryker-dotnet --tool-path'. After you have installed Stryker you can easily run it by changing the working directory to the path where your test project lives and running Stryker with the command dotnet-stryker from the installation location.
Stryker can create a multitude of reports, it can create a html report, a json report and you can even create a dashboard to see the difference with another branch. This will enable you to see if the pull request you've created will lower your so-called mutation score, a score given on how many mutations have survived, or maybe improve your score.
To make it even easier to use the Stryker in a pull request is the integration with SonarQube. You can let Stryker create an issue for every surviving mutation. Together with the fact that SonarQube has Pull request decoration, you don't even have to look at the report manually or go to SonarQube to see if there has been any changes. You can simply wait for the build to finish and see if there are any comments on your pull request.

Who is it for

Mutation testing is for everyone who has unit tests and wants to build high quality software. It is one of those tools that give you more control and insight on what the quality of your code is and it can prevent bugs from being introduced in your software.
In summary the added value of using Stryker, or any mutation testing tool, is the ability to determine the quality of your tests which in turn will tell something about the quality of your application. This is in contrast to code coverage which only tells you if your code has been hit by a test instead of telling you if your codeMutatio has been tested. Mutation testing can be a great asset in preventing bugs from entering your code and therefore should be in the CI-pipeline of every project.
For more information on how to use Stryker and other functionality it has to offer, please go to their website at https://stryker-mutator.io/ ●

75% of our IT colleagues agree

ing.nl/careers/it

Jasper Sprengers

Practice what you preach

Deze column gaat over de waarde van het programmeren als ambacht. Ook voor managers is het belangrijk daar voeling mee te blijven houden. Maar ik begin met een muziekvoorbeeld. Wijlen Steve Jobs was geen programmeur of ontwerper maar zag zich graag als de dirigent die het beste uit zijn orkest haalde. Of perste, als je zijn biografie leest. Als Apple het New York Philharmonic was, dan had Jobs wel wat van sterdirigent Leonard Bernstein, net zo berucht om zijn temperament.

Uitvinder en uitvoerder waren van oudsher dezelfde persoon. Wat componisten als Mozart en Chopin bedachten konden ze zelf ook virtuoos uitvoeren. Maar om hun orkestwerken voor vijftig of meer muzikanten in goede banen te leiden was een nieuwe functie nodig: die van dirigent, tegenwoordig een volwaardige masterstudie. Dirigenten bespelen ook wel een instrument, maar meestal niet op hetzelfde topniveau als de muzikanten die zij muzikaal leiden. Qua vakmanschap zijn de vereisten voor beide beroepen even hoog, maar maatschappelijk waarderen we ze wel degelijk anders. Wie leiding geeft of de grote lijnen uitzet staat hoger op de ladder, met overeenkomstig salaris.

Terug naar de software, want in onze branch is het niet anders. Hiërarchisch gezien zijn de programmeurs de muzikanten, is de dirigent de development lead en de enterprise architect de grote componist. Maar tussen de rollen bestaat veel minder onderscheid tussen uitvinder en uitvoerder dan je zou denken. Programmeren is niet het volgen van een recept, zoals brood bakken. De uitdagingen liggen weliswaar op meer gedetailleerd niveau (nl. de broncode) dan de grote lijnen waar de architect mee worstelt, maar vergen net zoveel creativiteit. Waarom vinden veel bedrijven het echter normaal dat een developer minder code schrijft naarmate haar ervaring groeit? De progressie van senior/lead developer naar architectuur verraadt een mening dat code schrijven op zeker moment te min voor je is. Grote onzin, en wel om drie redenen.

Mensen die de grote lijnen uitzetten voor grote systemen hebben een streepje voor als ze ook in detail begrijpen wat anderen voor ze bouwen. Ze zouden op elk moment moeten kunnen bijspringen – al is het wel belangrijk dat ze die neiging onderdrukken, want niemand houdt van micromanagers. De naam Elon Musk dringt zich hier op.

Ten tweede kun je ook na twintig jaar steeds een beetje beter worden in programmeren. Daarbij verouderen talen en trends sneller dan je fysieke lichaam. Wie beweert dat hij alles inmiddels wel weet is waarschijnlijk een one-trick pony. Niet zo verstandig voor je inzetbaarheid, en saai bovendien. De noodzaak om bij te blijven houdt mij juist gemotiveerd.

In kunst en wetenschap brengt jeugdig talent en geldingsdrang vaak vroeg succes. Daarom vindt Mark Zuckerberg jonge mensen gewoon slimmer. Misschien, maar het tomeloos enthousiasme en hoog IQ van de jonge honden levert ook onbegrijpelijke staaltjes op van pocherige 'kijk mij eens' code. Mijn derde reden om te blijven coderen is dat ervaren krachten bescheiden zijn en bewust van hun beperkte hersenpan. Ze hebben de pest aan slimme truukjes en ze weten meteen welke beroemde programmeur ik hier zojuist citeerde.

Edsger W. Dijkstra The Humble Programmer (1972) https://tinyurl.com/3wfp3akp

Jasper Sprengers

Wil je meer weten over deze Microsoft opleidingen, groeipaden of sociale activiteiten?
Neem contact op met Cindy Davids (Cindy.davids@ordina.nl).