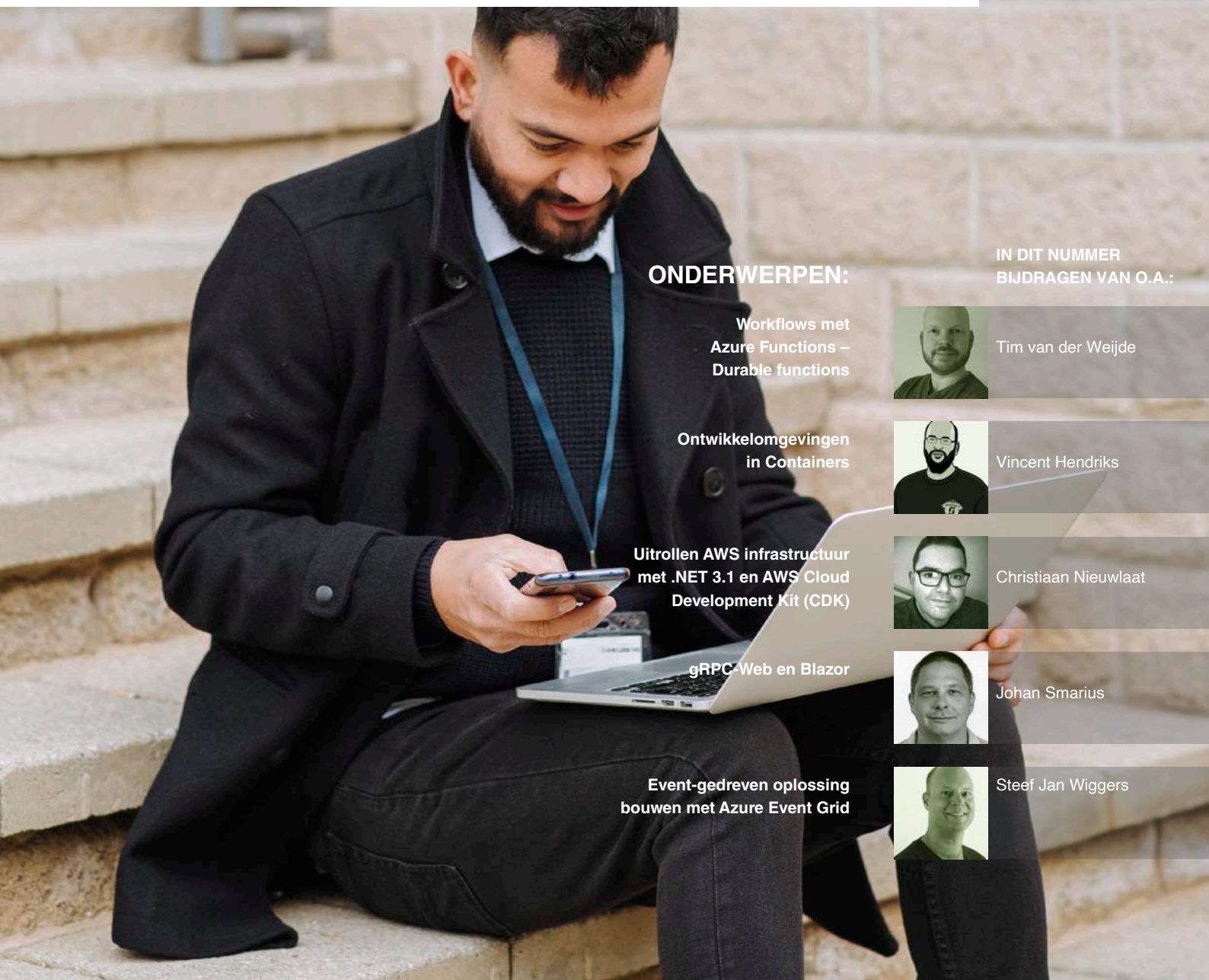
ISSN: 2211-6486

SDN MAGAZINE KNOW-LEDGE #140

Nummer 140 Januari 2021 SDN Magazine verschijnt elk kwartaal en is een uitgave van Software Development Network

SDN Magazine is hét lijfblad van en voor programmeurs, met artikelen over ontwikkelen voor Windows, Web, Mobile of Cloud. Of u nu ontwikkelt voor .NET, Delphi of iets anders, als lid van het SDN weet u alles.



ONDERWERPEN:

IN DIT NUMMER BIJDRAGEN VAN O.A.:

Tim van der Weijde

Vincent Hendriks

Christiaan Nieuwlaat

Johan Smarius

Steef Jan Wiggers

TERUGBLIK

FUTURE TECH

MASTERCLASSES LIVE

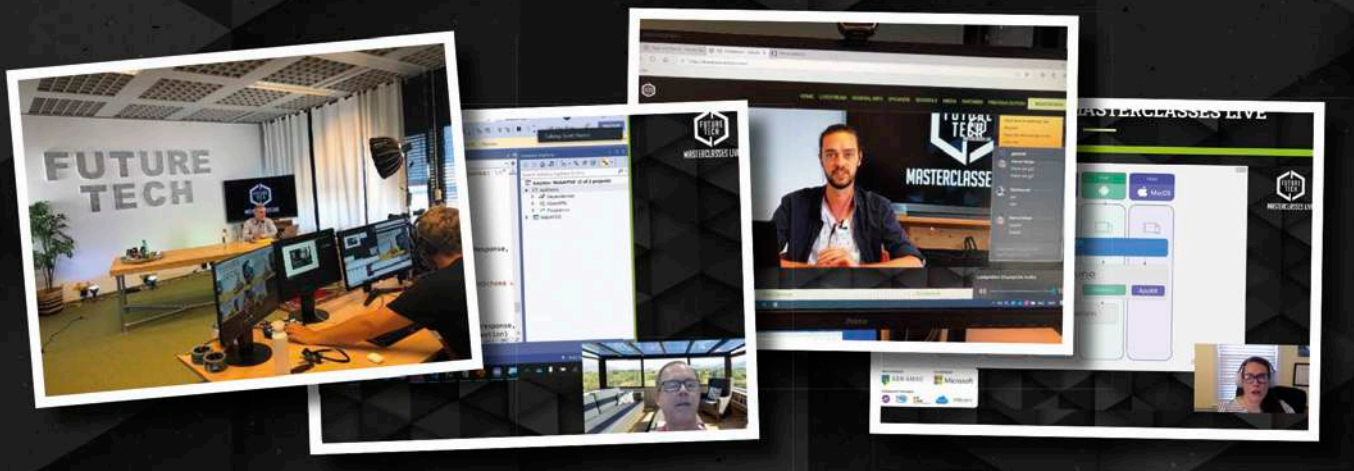
FUTURE TECH Masterclasses Live

Vanwege het Corona virus werd Future Tech de Developers conferentie voor Microsoft Technologies & .Net dit keer niet fysiek georganiseerd in de Jaarbeurs te Utrecht maar volledig succesvol virtueel.

Maar liefst 3 dagen lang kon de bezoeker vanaf 16:00 – 18:00 uur verschillende interactieve sessies volgen van sprekers van over de hele wereld.

Zo konden de bijna 600 bezoekers live bijdragen volgen van oa Scott Hunter (director of Program Management @ Microsoft), Kenzie Wahlen (Developer/MVP), Gian Paolo Santopaolo (MVP), Abel Wang, Michael R Bentley (ABN AMRO), Jyoti Singh en Rodrigo Dias.

De Future Masterclasses Live waren gratis te volgen. Het event is mogelijk gemaakt door ABN AMRO en Microsoft. SDN is 'member of the board' van de stichting Future Tech.



Bekijk alle video's nog een keer op https://futuretech.nl/livestream/

Alle activiteiten van Future Tech blijven volgen? Volg Future Tech op Facebook, Twitter of sluit aan bij de community via Meetup.com/Future-Tech-NL

Sponsored by

ABN·AMRO Microsoft

voorwoord

Beste SDN Magazine lezer

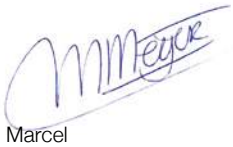
De beste wensen voor 2021! Dit jaar zal voor de SDN in het teken van verandering staan. We hebben grootste plannen en we willen de SDN naar een hoger plan tillen. Daarover zullen we in een later stadium nog meer vertellen, maar het magazine en de SDN Cast blijven een belangrijk onderdeel. Nu maar hopen dat we ook weer spoedig events kunnen organiseren en elkaar IRL (in real life) te kunnen spreken en samen te discussiëren/te sparren over technologie. Tenslotte zijn wij mensen kuddedieren en genieten we van samen dingen doen.

Je kunt met ons mee discussiëren tijdens onze tweewekelijkse SDN Cast via Twitch en Youtube. Iedere twee weken hebben wij een interessant onderwerp met een expert uit de community. Je kunt ze terugkijken op ons youtube kanaal. Mocht je graag iemand in onze cast willen zien, mail ons zijn/haar naam en wij zullen hem/haar benaderen om op treden als expert gast.

Voor je ligt het nieuwste SDN-magazine. In dit magazine artikelen over: AWS infrastructuur met .NET, gRPC en Blazor, Unit testen, Monolieten naar services, ontwikkelen in een container, Azure event grid en durabel functions. Onze collega's uit het community Christiaan Nieuwlaar, Johan Smarius, Marcel Meijer, Vincent Hendriks, Steef Jan Wiggers, Tim van der Weijde en Sander Hoogendoorn geven allemaal een inkijkje in hun ervaringen over de verschillende onderwerpen. Bedankt auteurs, super waardevolle verhalen.

Veel leesplezier en tot ons volgende event of uitzending van de SDN Cast Rebooted. Wil je schrijven of spreken of meedoen aan onze cast meld je dan op redactie@sdn.nl.

Groeten,



Marcel
Eindredacteur Magazine en Voorzitter SDN

SDN MAGAZINE KNOW-LEDGE#140

Over Marcel Meijer:
Marcel Meijer is Freelance (Enterprise) Architect, Teamlead, CTO, CTO Advisor of interim manager. Daarnaast is Marcel voorzitter, bestuurslid, event organisator en eindredacteur bij de SDN. Sinds 2010 mag hij zich MVP noemen.

Colofon

Uitgave
Software Development Network
Zesentwintigste jaargang
No. 140 • januari 2021

Bestuur van SDN
Marcel Meijer, voorzitter
Rob Suurland, penningmeester
Roy Janssen, secretaris

Redactie
Marcel Meijer (redactie@sdn.nl)

Aan dit magazine werd meegewerkt door
Aan dit magazine werd meegewerkt door Bob Swart, Maarten van Stam, Arjen Bos, Roy Janssen, Rob Suurland, Marcel Meijer en natuurlijk alle auteurs!

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

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

Vormgeving en opmaak
Reclamebureau Bij Dageraad
Winterswijk www.bijdageraad.nl

Adverteerders

Adverteren
Informatie over adverteren en de advertentietarieven kunt u vinden op www.sdn.nl onder de rubriek Magazine.

SDN MAGAZINE KNOW-LEDGE #140

AGENDA 2021

2-4 maart
Microsoft
Ignite

24 en 25 maart
Future Tech
Master Classes
Live

17-19 mei
Techorama 2021
The Virtual Edition

Inhoud

Vincent Hendriks

Ontwikkelomgevingen in Containers

Er is niks zo vervelend als het optuigen van een nieuwe ontwikkelomgeving. Het is zonde van de tijd. Daarnaast wil ik in de tijd dat ik niet aan het overleggen ben gewoon waarde kunnen leveren richting mijn opdrachtgever en collega's en me niet druk te moeten maken over het bijhouden van mijn tools. Ontwikkelomgevingen in containers gaan ons hierbij helpen.

INTRODUCTIE

De tijd dat ik bezig ben met het opzetten van mijn ontwikkelomgeving is, in mijn optiek, verloren tijd. Niet dat het niet belangrijk is, want een goede ontwikkelomgeving zorgt ervoor dat ik snel mijn werk kan doen, maar kan dit niet sneller en efficiënter?
Dit is toch wel iets wat je meerdere malen in je carrière zult moeten doen. Wat mijn ervaring hierbij is, is dat gedurende het opzetten van mijn ontwikkelomgeving (helemaal volgens de documentatie indien deze aanwezig is…) de tools en SDK's die in de documentatie staan inmiddels alweer verouderd zijn en niet meer te kloppen. Uiteraard werkte ik deze documentatie weer netjes bij, maar werkt iedereen nu op dezelfde versies? Gebruiken we allemaal dezelfde tools? En ja, die documentatie is de volgende dag alweer verouderd.
Of ken je die situatie dat je ontwikkelomgeving opeens niet meer werkt na het installeren van een laatste update? …
Zoals ik al aangaf: verloren tijd…

CONTAINERS

Container technologie bestaat al jaren, ruim voordat docker er mee kwam, maar het is pas de laatste jaren echt populair geworden. Het bracht de mogelijkheid om veel eenvoudiger applicaties te packagen en te deployen onder het motto: "Build, Ship, Run, Any App Anywhere". Het is veel meer lichtgewicht dan bijvoorbeeld een virtual machine. Dat betekent overigens niet dat virtual machines overbodig zijn geworden, maar ze worden nu vaak samen gebruikt: Virtual machines worden bijvoorbeeld ingericht als een kubernetes node waar vervolgens weer docker containers op draaien, etc.
Containers hebben veel gedaan voor onze hele OTAP straat. Met name voor de test, acceptatie en productie omgevingen. Vaak is het zelfs zo dat containers eigenlijk pas na het ontwikkelen relevant werden. Applicaties werden lokaal gemaakt en wanneer ze "klaar" waren werden de applicaties in docker containers gepackaged en door de hele Test/Acceptance/Production straat gehaald.

Alhoewel dit al flinke verbeteringen oplevert, ontbreekt hier wat mij betreft nog iets. Ik heb nog steeds developer tools nodig zoals Node.js, Npm, verschillende SDK's, een lokale SQL Server, een SMTP server, etc. En deze tools en services moeten ook regelmatig bijgewerkt worden en bij voorkeur op dezelfde versie als de productie omgeving. En dan hebben we het nog niet eens over het meest favoriete onderwerp van ontwikkelaars: Het bijhouden van de documentatie hoe je de ontwikkelomgeving in moet richten.
Eigenlijk wil je deze tools niet moeten beheren en wil je gewoon ontwikkelen. Het liefst in een kant-en-klare IDE lokaal ofwel in de cloud.

Waarom zou je niet gewoon je hele ontwikkelomgeving in de cloud of in een container draaien? Tegenwoordig kan dit!

Wanneer we containers kunnen maken voor productie runtimes, waarom kunnen we hier dan ook niet gewoon onze SDK's, tools, etc. in installeren? Een instant ontwikkelomgeving in een kwestie van minuten! En weet je wat? Je kunt ze ook nog eens delen met je hele team. Een nieuw teamlid hoeft ook niet meer lang bezig te zijn met het inrichten van zijn ontwikkelomgeving.

Een tijd geleden wilde ik wat kennis opdoen over Python en Machine Learning en had ik een dag vrijgehouden, gezorgd voor een goede werkplek om me hier 100% op te kunnen concentreren. Vervolgens was ik de helft van de tijd bezig om mijn ontwikkelomgeving in te richten. Dat is gewoon erg pijnlijk. Als ik toen wist wat ik nu wist had ik dat niet gehad, want tegenwoordig zijn er geweldige tools die je kunt gebruiken om kant-en-klare ontwikkelomgevingen te creëren.

DEV CONTAINERS

Een containerized ontwikkelomgeving is eigenlijk niet meer een collectie van 1 of meerdere containers waarin alle tools, SDK's, services, etc zijn geïnstalleerd. De hele setup hiervan is een collectie van json, docker en yaml files wat weer perfect opgeslagen kan worden in source control. Eigenlijk op precies dezelfde plek waar je broncode en je CI/CD configuratie zit. Aanpassingen zullen dezelfde procedures volgen zoals de rest van de code zoals bijvoorbeeld peer reviews.

Nu zul je wellicht denken: Leuk als die SDK's in een container zitten, maar hoe werk ik hier dan mee?

Hiervoor zijn verschillende opties:

- Je kunt bijvoorbeeld Visual Studio Code gebruiken die d.m.v. een extensie enkele tools installeert in de container om vervolgens met een lokale VS Code instantie te kunnen werken.
- Je kunt een web based UI gebruiken door gebruik te maken van Visual Studio Online/Github Codespaces. Visual Studio Codespaces is overigens depreciated en Github Codespaces zal de nieuwe dienst zijn die daarvoor in de plaats komt. Github Codespaces is, op het moment van schrijven, ook nog in private preview.

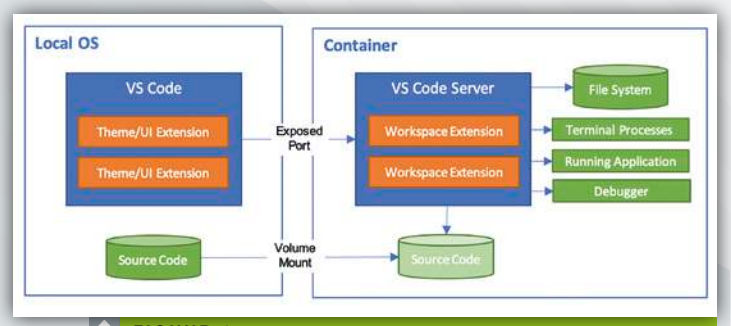
Ondersteuning voor Visual Studio i.c.m. GitHub Codespaces wordt op dit moment ook al aan gewerkt en is in private preview stadium. Als iemand van Microsoft dit leest: Voeg me aub even toe… Ik ben een fan!

Ontwikkelomgevingen in Containers

VISUAL STUDIO CODE (VS CODE)

Laten we eerst kijken naar de oplossing die Microsoft met VS Code biedt en hoe dit exact werkt. Zoals jullie wellicht al weten is VS Code een uitgebreide code editor met een legio aan extensies. Niet te verwarren met Visual Studio wat een full-fledged IDE is. Visual Studio komt typisch met een compiler, debugger, linker en SDK's. Bij Visual Studio Code moet je dit er allemaal los bij installeren.

De extensie die het mogelijk maakt om te ontwikkelen in een container vanuit VS Code is "Visual Studio Code Remote - Containers". VS Code is gebouwd op basis van het Electron framework wat bedoeld is om Node.js webapplicaties naar de desktop te brengen. De architectuur die daarvoor nodig is heeft het relatief makkelijk gemaakt om de UI van Visual Studio Code los te trekken van de backend. Dit is dan ook waar deze extensie gebruik van maakt. De extensie start een docker container waarbij alle benodigde tools, SDK's, etc in geïnstalleerd zijn, maakt een volume mount aan die ervoor zorgt dat de broncode op je machine beschikbaar is in de container en installeert vervolgens de backend van VS Code in de container (VS Code Server). Welke container(s), extensies, etc nodig zijn staat allemaal geconfigureerd in een ".devcontainer" directory.



FIGUUR 1

CONFIGURATIE VAN EEN DEV CONTAINER

Alle configuratie van een devcontainer staat in dus in een ".devcontainer" directory waarin een "devcontainer.json" file staat. In deze JSON staan alle benodigdheden die VS Code nodig heeft. Denk aan welke Dockerfile gebruikt moet worden, welke extensies en shell gebruikt moeten worden en onder welke user alles moet draaien.
Een voorbeeld van een devcontainer setup voor C#/.NET Core:

```
{
        "name": "C# (.NET)",
        "build": {
                "dockerfile": "Dockerfile",
                "args": {
                        "VARIANT": "5.0",
                        // Options
                        "INSTALL_NODE": "true",
                        "NODE_VERSION": "lts/*",
                        "INSTALL_AZURE_CLI": "false"
                }
        },

        "settings": {
                "terminal.integrated.shell.linux": "/bin/bash"
        },
        "extensions": [
                "ms-dotnettools.csharp"
        ],

        "remoteUser": "vscode"
}
```

De exacte inrichting van de container is vervolgens te vinden in de "Dockerfile". In deze Dockerfile zou je alle tools kunnen toevoegen. Denk in dit geval bijvoorbeeld ook aan een Azure CLI of Node.js

MEERDERE SERVICES

In bovenstaand voorbeeld spreken we over een enkele container met daarin alles wat je nodig hebt, maar dat is niet altijd voldoende. Wanneer je bijvoorbeeld gebruik wilt maken van services waar al bestaande container images van aanwezig zijn is het vaak eenvoudiger om deze er gewoon naast te draaien. Denk bijvoorbeeld aan een SQL Server of Redis container. Daarvoor hebben ze ook de mogelijkheid gemaakt om een docker-compose.yaml file te gebruiken waarin al deze services gespecificeerd zijn. Zodra je de devcontainer opent vanuit VS Code dan zullen ook alle bijbehorende services, zoals in de docker-compose.yml staan, opgestart worden. Ben je klaar met ontwikkelen en sluit je de devcontainer? Dan zullen de services ook gestopt worden.
Een voorbeeld van een docker-compose.yaml:

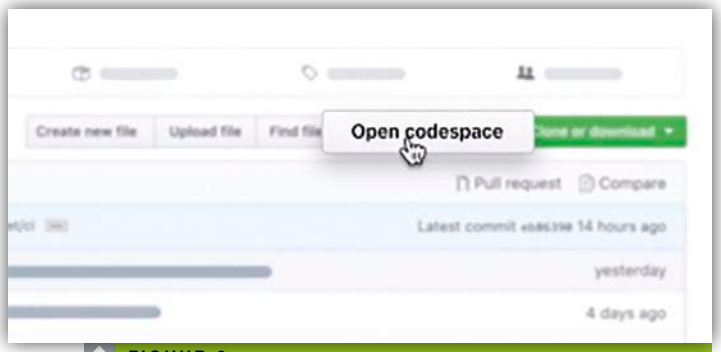
```
version: '3'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        VARIANT: 3.1
        INSTALL_NODE: "false"
        NODE_VERSION: "lts/*"
        INSTALL_AZURE_CLI: "false"
        USER_UID: 1000
        USER_GID: 1000

    volumes:
      - ..:/workspace:cached

    # Overrides default command so things don't shut down after the process ends.
    command: sleep infinity

    network_mode: service:db

  db:
    image: mcr.microsoft.com/mssql/server:2019-latest
    restart: unless-stopped
    environment:
      SA_PASSWORD: P@ssw0rd
      ACCEPT_EULA: Y
```

Dit voorbeeld zorgt voor een container waarin ontwikkeld kan worden (de service 'app') en een container met SQL Server.

De connectionstring van SQL Server zal de host ‘db’ gebruiken en zal voor iedere ontwikkelaar exact hetzelfde zijn. Nog meer voorbeelden zijn te vinden op een github repository van Microsoft zelf: https://github.com/microsoft/vscode-dev-containers/tree/master/containers

ONTWIKKELEN IN DE CLOUD

Tot nu toe heb ik het alleen maar gehad over het ontwikkelen met VS Code in een container op een lokale omgeving. Je hebt nog steeds VS Code op je machine nodig, maar wat nu als je helemaal geen software nodig hebt? Dit zijn mogelijkheden die Visual Studio/GitHub Codespaces bieden. Omdat de UI van VS Code ook in de browser kan draaien en de backend gescheiden kan draaien heeft Microsoft een dienst ontwikkeld die dit helemaal ontzorgt. Zij leveren de code editor die draait in de browser EN zorgen dat de devcontainer op de achtergrond draait. Deze dienst gebruikt ook een vergelijkbare devcontainer setup zoals bij een lokale VS Code.
Om te starten met ontwikkelen zou je in het geval van GitHub Codespaces naar de betreffende repository navigeren en op “Open codespace” moeten klikken. Vervolgens regelt GitHub de rest.

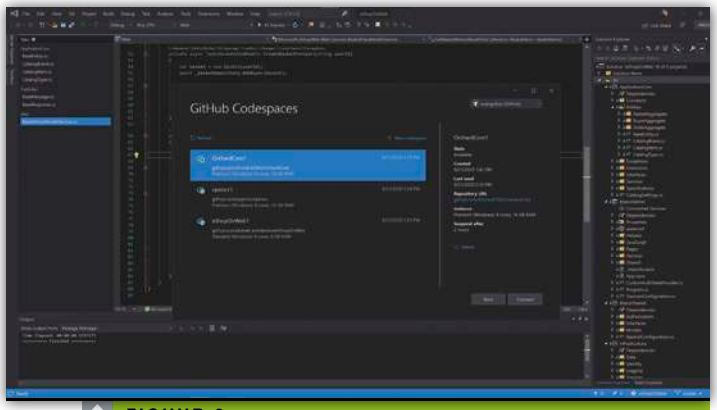


FIGUUR 2

In een tijd waar thuiswerken steeds vaker voorkomt kan een opstelling zoals deze erg fijn zijn. De broncode staat op 1 plek en dus niet op een laptop die bijvoorbeeld gestolen kan worden.

VISUAL STUDIO

VS Code is een geweldige tool, maar toch niet helemaal hetzelfde als Visual Studio. Mijn persoonlijke ervaring is dat het ontwikkelen van .NET applicaties in Visual Studio toch fijner werkt dan in VS Code. Gelukkig zijn ze ook bezig om Visual Studio te laten werken met een Github Codespace omgeving.



FIGUUR 3

SECURITY

Vragen die ik regelmatig krijg als ik over dit onderwerp spreek gaat over security. Ik kan hierover alleen maar voordelen benoemen. Zou er namelijk een kwaadwillende package zijn dan is dit sowieso geïsoleerd in een container en is er geen mogelijkheid om bij andere resources te komen zoals broncode van andere projecten.
Daarbij is het ook mogelijk om voorgebouwde container images te gebruiken die gescand zijn op bekende kwetsbaarheden.
Gebruik je geen voorgebouwde container images, maar een lokale Dockerfile? Zelfs dan kun je gebruik maken van scan functionaliteiten zoals docker scan (https://docs.docker.com/engine/scan/).

ONTWIKKELOMGEVINGEN MOETEN OOK WORDEN ONDERHOUDEN

CONCLUSIE

Ik geloof dat dit de toekomst gaat zijn voor veel ontwikkelaars. Het bied mogelijkheden om zeer snel een ontwikkelomgeving te creëren voor nieuwe ontwikkelaars met zo min mogelijk inspanning. Ook lost het problemen op m.b.t. afhankelijkheden van diverse tools en SDK’s die per project/code base anders kan zijn. Op het moment dat 1 ontwikkelaar met verschillende code bases werkt is het een kwestie van tijd dat hier een probleem ontstaat. Tegelijkertijd zorgt het ook voor veel minder documentatie behoefte t.b.v. de ontwikkelomgeving die vanaf nu altijd up-to-date is.
Vond je dit verhaal interessant en zou je meer willen weten?
Kom gerust met me in contact via: https://www.linkedin.com/in/vhendriks

LINKS

- https://code.visualstudio.com/docs/remote/containers
- https://github.com/microsoft/vscode-dev-containers/tree/master/containers
- https://docs.github.com/en/free-pro-team@latest/github developing-online-with-codespaces/using-codespaces-in-visual-studio

VINCENT HENDRIKS

Vincent Hendriks is een Software Engineer met meer dan 15 jaar ervaring en een enorme passie voor het vak. Vincent heeft een zeer brede ervaring in onderwerpen zoals C#, .NET/.NET Core, Software achitectuur, CQRS, Event Sourcing, SQL, Azure, Docker, Umbraco en nog veel meer. Vincent is regelmatig te vinden bij verschillende usergroups en is werkzaam bij Team Rockstars IT.

LinkedIn: https://www.linkedin.com/in/vhendriks

Tim van der Weijde

Workflows met Azure Functions

DURABLE FUNCTIONS

Steeds vaker hoor je de wens voor het gebruik van microservices. Het is een hot item en wordt al snel ingezet met als oplossing tot het schaalbaar maken van een groot technisch landschap. Daarbij kan er ook gekozen worden om volledig serverless te gaan met bijvoorbeeld Azure Functions. Data kan nog steeds worden opgeslagen in 1 of meerdere databases.
Een vraag die ik dikwijls terug hoor komen is hoe je complexe workflows implementeert binnen Azure Functions? Sla je dat op in je bestaande database(s) of zijn er andere oplossingen te bedenken?

AZURE FUNCTIONS & SERVERLESS

Tegenwoordig zijn we het allemaal gewend om code te schrijven in kleine testbare units. Vaak gaat het om een systeem waarbij gereageerd wordt op bepaalde events zoals een klik op een button, een timer die afgaat, een bericht op een queue of een HTTP request. Azure Functions is hierbij eigenlijk niks anders dan een “klein” stuk code, die uitgevoerd wordt na het afgaan van een te definiëren trigger. Het “Hello universe” kan je bijvoorbeeld eenvoudig implementeren met een functie met een HTTP request trigger, die vervolgens een HTTP Response teruggeeft.
Wat Azure Function krachtig maakt is dat het volledig serverless is. Dit houdt in dat jouw code gedeployed kan worden, zonder dat je je zorgen hoeft te maken om de onderliggende infrastructuur. Dus geen beheer van aantal servers, keuze in CPU en/of keuze in memory.
Azure Functions kan ook automatisch scalen door extra host-instanties in het leven te roepen wanneer de hoeveelheid van executies toeneemt. Als het aantal executies weer afneemt, zal ook de benodigde host-instanties weer afnemen. Bijvoorbeeld bij een functie die een queue trigger heeft; wanneer de queue is leeg, zal er geen host-instantie actief zijn. Komt er een bericht op de queue, dan zal Azure Functions een instantie opstarten om het bericht uit de queue te halen. Komen er veel meer berichten op de queue, dan zal Azure Functions meer host-instanties opstarten.

WORKFLOWS IN AZURE FUNCTIONS

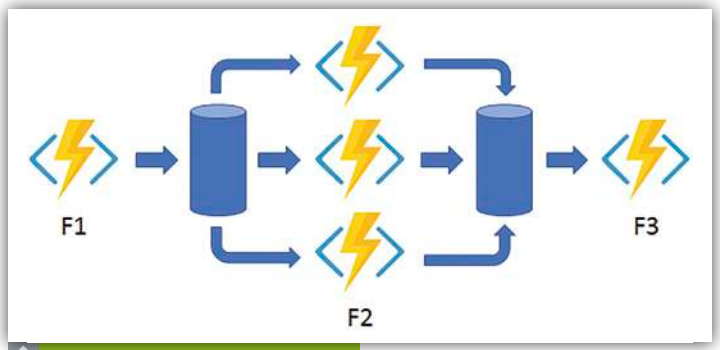
Het is dus eenvoudig om code te schrijven die eenmalig wordt uitgevoerd. Ook het uitvoeren van meerdere functies achter elkaar kan door middel van het gebruik van een message queue, ook wel chaining genoemd.
Het implementeren van een workflow met standaard Azure Functions wordt daarentegen al snel ingewikkeld. Bedenk hierbij ook dat een Azure function stateless moet zijn om automatisch te kunnen scalen. Een voorbeeld is het “Function chaining” pattern, waarbij de ene functie de ander opvolgt nadat die klaar is met het uitvoeren van zijn taak. Het toepassen van error-handling over alle verschillende functies heen is al best lastig om bij te houden. Je kan natuurlijk denken aan het apart opslaan in een database.
Een ander voorbeeld is het “Fan-out/Fan-in” pattern, waarbij 1 functie meerdere andere functies triggered. Vervolgens wil je wachten totdat al die functies klaar zijn om vervolgens een laatste eind functie uit te voeren. Om dit uit te voeren in Azure Functions moet je dus ergens een state bewaren.

DURABLE FUNCTIONS

Microsoft heeft voor de ondersteuning van workflows een uitbreiding toegevoegd genaamd Durable Functions.
Met de Durable Functions extension is het mogelijk om complexe workflows in te richten, waarbij het afhandelen van de state geheel uit handen wordt genomen.
Uiteindelijk is het een extra abstractie laag, die onder water wel degelijk Azure Storage gebruikt om de state bewaren. Maar het grote voordeel is dat je zelf daar niet mee bezig hoeft te zijn. Hierdoor kan je je geheel richten op de business logica.
Door middel van verschillende nieuwe functie types kan je eenvoudig een workflow definiëren en iedere stap in de workflow apart implementeren. Het volgende deel legt de verschillende functie types uit.

TYPES

Voor het maken van een workflow binnen Azure Functions zijn er meerdere functies nodig. De Durable Function extension bevat 4 verschillende types.
Orchestrator functies zorgen voor de regie van de workflow. Door middel van code bepaal je de manier en in welke volgorde bepaalde acties worden uitgevoerd. Vanuit de orchestrator functie worden weer andere functies getriggerd. De orchestrator functie hoort zelf geen logica te bevatten.
Activity functies bevatten de daadwerkelijke logica binnen de workflow. Bijvoorbeeld de credit check of het bestaan van de klant in een externe database. Deze functies lijken het meest op een gewone Azure functie.
Client functies zijn over het algemeen gewone Azure functies, maar met een durable client input binding. Door middel van deze binding is het mogelijk om orchestrator functies en/of entity functies te triggeren. Een orchestrator functie wordt altijd getriggerd door een client functie en zal nooit direct worden aangeroepen.waar de workflow zich bevindt.



FIGUUR 1: FAN-OUT/FAN-IN

Entity functies kan je optioneel gebruiken om een deel van de workflow state te bewerken. Waarbij de state van de workflow impliciet wordt beheerd, kan je met de entity functie de state expliciet bewerken.

In .NET is het mogelijk om een wrapper-class te maken om een entity functie heen. Hierdoor heb je properties en methods tot je beschikking en je kan deze op een type-safe manier aanroepen. Het voorbeeld later in het artikel zal hiervan gebruik maken.

PATTERNS

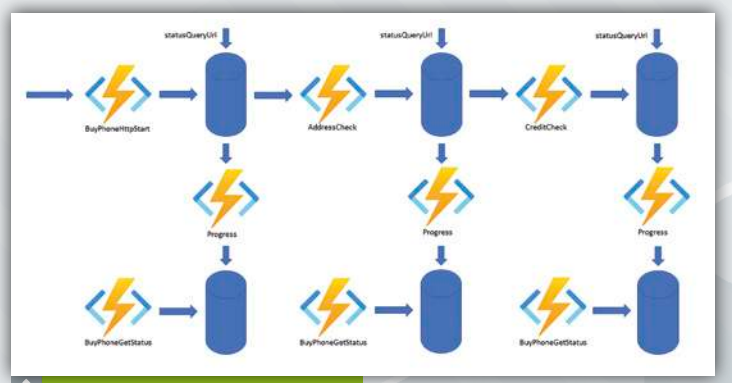
Om Durable Functions goed te begrijpen is het handig om naar voorbeelden te kijken. Het programmeren van workflows in code is natuurlijk niet nieuw en er bestaan hiervoor al meerdere patterns. Durable Functions helpt je bij het opzetten van verschillende soorten patterns.
De bekende “Function chaining” en “Fan-out/Fan-in” patterns zijn al eerder genoemd.
Een voorbeeld van een “Function chaining” pattern is het downloaden van een bestand en vervolgens het wegzetten op een export locatie.
Een voorbeeld van de “Fan-out/Fan-in” pattern is het downloaden van meerdere bestanden, en het mogelijk is om deze bestanden parallel weg te schrijven i.p.v. één voor één. Vervolgens wordt er gewacht op het resultaat.
Als voorbeeld zal ik het Async HTTP API pattern wat uitgebreider toelichten.
Bij het Async HTTP API pattern gaat het om het uitvoeren van een langdurig proces/workflow waarbij een gebruiker moet wachten. Dit is iets wat vaak voorkomt in een weboplossing. De gebruiker stuurt een HTTP request naar een webapplicatie en vervolgens gaat er een langdurig proces/workflow draaien.
Omdat bij de Durable Functions ook de state wordt bijgehouden, bevat het gebruik van deze functies al een standaard manier om de status van de workflow op te vragen, hierover straks meer.

VOORBEELD ASYNC HTTP API’S

Er zijn meerdere manieren om te beginnen met het ontwikkelen van Azure Functions, waaronder in het Azure Portal. Dit is een eenvoudige manier om inzicht te krijgen hoe het is opgezet binnen Azure.
Ander manieren (maar niet de enige) zijn met Visual Studio Code of via de AZ CLI.
De getoonde code-voorbeelden zijn door middel van Visual Code en Azure Function extension gemaakt. Hierbij worden delen van de functies automatisch gegenereerd. Doe je dit via het Azure Portal, dan krijg je weer andere voorbeelden/templates te zien.
In dit voorbeeld kan de gebruiker via het internet een telefoon kopen. Voordat de telefoon daadwerkelijk besteld kan worden zijn er een aantal validaties nodig, waaronder een credit check en een adrescontrole. Om het voorbeeld eenvoudig te houden, zal het alleen deze validaties bevatten.

1.	**BuyPhoneHttpStart**	Een HTTP trigger client functie die zorgt voor het starten van de orchestrator functie en het geven van informatie over de nieuwe orchestrator functie instantie. (b.v. instanceID)
2.	**BuyPhoneOrchestrator**	De orchestrator functie die de workflow uitvoert. Deze functie bevat geen logica.
3.	**Progress**	Een wrapper-class voor een entity functie, die de voortgang van de validatie bevat.
4.	**AddressCheck**	Een activity functie die zorgt voor de adres validatie
	Creditcheck	Een activity functie die zorgt voor de credit check validatie
5.	**BuyPhoneGetStatus**	Een HTTP trigger functie die de voortgang van de validaties ophaalt

TABEL 1: OVERZICHT FUNCTIES



FIGUUR 2: OVERVIEW

1. DE CLIENT FUNCTIE BUYPHONEHTTPSTART

Binnen de HTTP client functie wordt een nieuwe orchestrator functie instantie aangemaakt en wordt het resultaat weer teruggegeven.

```
[FunctionName("BuyPhoneHttpStart")]
0 references
public static async Task<HttpResponseMessage> BuyPhoneHttpStart(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    // Create new instance of the orchestrator
    string instanceId = await starter.StartNewAsync(nameof(BuyPhoneOrchestrator), null);
    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

    // return information of the orchestrator instance(e.g. instanceId, resource-url's
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

FIGUUR 3

Het resultaat bevat onder andere de instanceId en een status URI “statusQueryGetUri”. Zie figuur 3.

```
{
  "id": "4d0177bd1b264ed29aae0286b8a95721",
  "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/4d0177bd1b264ed29aae0286b8a95721
  "sendEventPostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/4d0177bd1b264ed29aae0286b8a95721/
  "terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/4d0177bd1b264ed29aae0286b8a95721/
  "purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/4d0177bd1b264ed29aae0286b8a9
}
```

FIGUUR 4

Met de status URI “statusQueryGetUri” kan de client (webbrowser) eenvoudig de huidige status van de orchestrator opvragen. Het veld “runtimeStatus” bevat de huidige status van de orchestrator en is al gelijk een eenvoudige manier om te weten of alles al is uitgevoerd. Zie figuur 4.

```
{
  "name": "BuyPhoneOrchestrator",
  "instanceId": "5bf693bef37b49eab06c86b09eba0563",
  "runtimeStatus": "Completed",
  "input": null,
  "customStatus": null,
  "output": null,
  "createdTime": "2020-12-07T20:21:43Z",
  "lastUpdatedTime": "2020-12-07T20:22:03Z"
}
```

FIGUUR 5

2. DE ORCHESTRATOR FUNCTIE BUYPHONEORCHESTRATOR

In de orchestrator functie wordt met code de workflow gedefinieerd. Het aanroepen van andere functies kan op een synchrone of asynschrone manier gebeuren. Wanneer de orchestrator functie

een andere functie aanroept, zal zijn uitvoering stoppen en wachten totdat de andere functie klaar is. De state van de orchestrator functie wordt tussendoor opgeslagen.
Wanneer de orchestrator functie weer opgestart, zal deze beginnen met een nieuwe context waarin het resultaat van de aangeroepen functie is opgenomen.
In het voorbeeld maakt de orchestrator functie eerst een nieuwe entity functie aan op basis van zijn unieke instanceId. Door middel van een proxy class van de entity functie, is het mogelijk om op een type-safe manier de Entity Functie aan te roepen.
In figuur 5 wordt twee keer de “entityProxy.Set” method aangeroepen. Dit is onderwater een zogenaamde “one-way” signal methode, waarbij de entity functie een signaal krijgt om iets uit te voeren.
De derde en laatste aanroep gaat niet via een proxy, maar via de orchestration context. Hierbij wordt direct de “context.SignalEntity” methode aangeroepen zonder proxy. Beide manier zijn mogelijk, waarbij de laatste alleen niet type-safe is.
Omdat Durable functions werken met een queueing mechanisme, kan het zijn dat de status een update krijgt, maar dat de workflow al klaar is. Dit noem je eventual consistency.
Binnen de orchestrator functie worden twee activity functies aangeroepen. Beide await functies worden hier nu synchroon aangeroepen, maar kan natuurlijk makkelijk ook parallel uitgevoerd door het gebruik van aparte Tasks. Het parallel uitvoeren van functies wordt gebruikt bij het Fan-out/Fan-in pattern.

```
[FunctionName("BuyPhoneOrchestrator")]
public static async Task BuyPhoneOrchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context, ILogger log)
{
    // Create Progress Entity
    var entityId = new EntityId(nameof(Progress),
                                $"progress-{context.InstanceId}");
    // Create a proxy of the Progress Entity.
    var entityProxy = context.CreateEntityProxy<IProgress>(entityId);

    // initial value
    var progress = new ProgressInput(0, "Validation started.");
    entityProxy.Set(progress);

    // First long-running call
    var result = await context.CallActivityAsync<bool>("AddressCheck",
            new Address("2661PB", 12));

    // Save result to the Progress Identity instance via the proxy.
    progress = new ProgressInput(50, $"AddressCheck done. Result: {result}");
    entityProxy.Set(progress);

    // Second long-running call
    result = await context.CallActivityAsync<bool>("CreditCheck", "NL33TW01234567890");

    // Save result to the Progress Identity instance via the orchestration context.
    progress = new ProgressInput(100, $"CreditCheck done.  Result: {result}");
    context.SignalEntity(entityId, nameof(IProgress.Set), progress);
}
```

FIGUUR 6: DE ORCHESTRATOR FUNCTIE

3. DE PROGRESS ENTITY

Met de Progress entity wordt de status van de voortgang expliciet opgeslagen en uitgelezen wanneer nodig. Het expliciet aanroepen zie je daarom ook terugkomen in de orchestrator functie en in de Http trigger functie BuyPhoneGetStatus.

Deze entity functie kan op twee manieren worden geïmplementeerd: een entity functie methode of een wrapper-class. Ik heb in dit voorbeeld gekozen voor de wrapper-class. Het voordeel hiervan is dat je type-safe kan werken, omdat je de class direct kan aanspreken via zijn properties en methods.
Entity functies werken onder water met een queueing mechanisme en zullen berichten ook één voor één achter elkaar uitvoeren. De methode Set wordt vertaald in een “one-way” signal message, die vervolgens op een queue komt, die door een entity wordt opgepakt.

```
[JsonObject(MemberSerialization.OptIn)]
public class Progress : IProgress
{
    [JsonProperty("progress")]
    public int ProgressValue { get; set; }

    [JsonProperty("validationresults")]
    public IList<string> ValidationResults { get; set; } = new List<string>();

    public void Set(ProgressInput input)
    {
        this.ProgressValue = input.Progress;
        this.ValidationResults.Add(input.ValidationResult);
    }

    public Task<int> GetProgress() => Task.FromResult(this.ProgressValue);
    public Task<IList<string>> GetValidationResults() => Task.FromResult(this.ValidationResults);

    [FunctionName(nameof(Progress))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
        => ctx.DispatchAsync<Progress>();
}
```

FIGUUR 7: WRAPPER CLASS ENTITY FUNCTIE

4. DE ACTIVITY FUNCTIES “ADDRESSCHECK” & “CREDITCHECK”

De twee activity functies zijn de functies die de daadwerkelijke business logica horen te bevatten. In dit voorbeeld zijn ze placeholders en imiteren een langdurig proces. De werkelijke implementatie kan verschillende langdurige processen bevatten naar externe systemen en/of derde partij diensten.

```
[FunctionName("AddressCheck")]
public static bool AddressCheck([ActivityTrigger] Address address, ILogger log)
{
    var entry = $"Checking address {address.HouseNumber} {address.ZipCode}";
    log.LogInformation(entry);

    // Don't this at home!
    // Imitate long-running process.
    Thread.Sleep(10000); // 10 seconds

    return true; // Just a boolean as a demo example.
}

[FunctionName("CreditCheck")]
public static bool CreditCheck([ActivityTrigger] IDurableActivityContext context, ILogger log)
{
    var bankAccountNumber = context.GetInput<string>();
    log.LogInformation($"Checking {bankAccountNumber}");

    // Don't this at home!
    // Imitate long-running process.
    Thread.Sleep(10000); // 10 seconds

    return true; // Just a boolean as a demo example.
}
```

FIGUUR 8: ACTIVITY FUNCTIES

5. DE HTTP TRIGGER FUNCTIE “BUYPHONEGETSTATUS”

De Http Trigger functie bevat een DurableEntityClient waarbij het mogelijk is om de state van een entity functie op te vragen door middel van het meegeven van een instanceId.
De client.ReadEntityStateAsync method is een “two-way” methode. De “two-way” zegt het eigenlijk al; Waarbij een “signal” methode één kant op gaat en géén resultaat zal teruggeven, zal je bij deze methode juist wel een resultaat terugkrijgen.

Het resultaat is een EntityStateResponse van het type Progress. Dit is de wrapper-class om de entity functie heen.
Uiteindelijk geven we de EntityState terug. Dit is een gedeserializeerde JSON object van het type Progress. Hierin staat onze uitgebreide status van de workflow, die binnen de orchestrator functie wordt gevuld.

```
[FunctionName("BuyPhoneGetStatus")]
public static async Task<HttpResponseMessage> BuyPhoneGetStatus(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableEntityClient client,
    ILogger log)
{
    // use the instanceID to build up the unique indentifier of the Progress Entity
    var queryString = req.RequestUri.ParseQueryString();
    string instanceId = queryString.Get("instanceId");

    var entityId = new EntityId(nameof(Progress), $"progress-{instanceId}");

    // Get the current state of the Progress Entity
    EntityStateResponse<Progress> stateResponse = await client.ReadEntityStateAsync<Progress>(entityId);

    if (stateResponse.EntityExists)
    {
        // return the current state.
        return req.CreateResponse(System.Net.HttpStatusCode.OK, stateResponse.EntityState);
    }
    else
    {
        // return a not found error.
        return req.CreateResponse(System.Net.HttpStatusCode.NotFound);
    }
}
```

FIGUUR 9: HTTP TRIGGER FUNCTIE MET DURABLEENTITYCLIENT

```
{
  "progress": 100,
  "validationresults": [
    "Validation started.",
    "AddressCheck done. Result: True",
    "CreditCheck done.  Result: True"
  ]
}
```

FIGUUR 10: GETSTATUS RESPONSE

TOT SLOT

Met de Durable Functions extensions heeft Microsoft het een stuk eenvoudiger gemaakt om workflows in een serverless omgeving te implementeren. In dit artikel wordt een voorbeeld gegeven om een indruk te krijgen. Er zijn nog vele andere concepten die niet besproken zijn om het maximale uit je oplossing te halen. Deze concepten zijn goed terug te vinden in de officiële documentatie.
Heb je nog vragen, laat het dan gerust weten. Neem contact op met Tim via tim@bluecape.nl

TIM VAN DER WEIJDE

Tim van der Weijde is een zeer ervaren developer die al meer dan 16 jaar actief is op het Microsoft platform. Hij is gedreven en nieuwsgierig naar bestaande en nieuwe technologieën en architecturen. Met zijn praktische aanpak en zijn focus op hoge kwaliteit en het continu verbeteren helpt hij teams in de ontwikkeling van hun werkwijze en vergroting van kennis.

Marcel Meijer

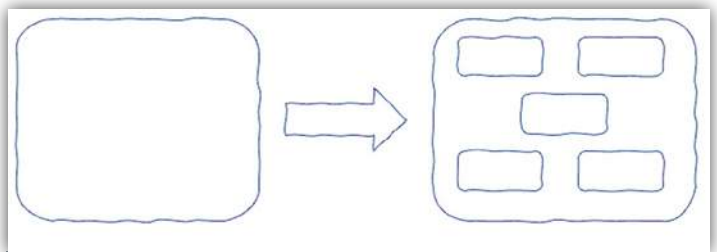
Monoliet naar Services

In dit magazine zie je veel auteurs vertellen over microservices, maar ook op onze events of andere conferenties zijn er veel sessies over. Ze vertellen dan altijd dat het belangrijk is om naar microservices over te stappen, want daarmee wordt het leven een stuk eenvoudiger. De voorbeelden die ze geven zijn dat altijd de standaardvoorbeelden van een product-order systeem of ze komen met voorbeelden vanuit een greenfield situatie. De presentaties gaan bijna altijd uit van het niet aanwezig zijn van legacy code of legacy systemen. Maar onze werkelijkheid is vaak veel weerbarstiger.

Veel bedrijven hebben een monoliet en een monoliet is niet iets waar je voor hoeft te schamen. Er zijn maar weinig omgevingen waar niet een vorm van monolithische code of omgeving is ontstaan. Zelfs bij startups begint het altijd met een redelijk puristische manier van ontwikkelen, maar na enkele weken/maanden wint de tijdsdruk van de schoonheid en raakt de software langzaamaan vervuild. Op zichzelf hoeft dat ook helemaal niet erg te zijn. Mits je tijdig refactorslagen uitvoert en bij iedere volgende aanpassing/verbetering/bugfix je code weer beter achterlaat (incheckt) dan dat je hem gevonden hebt: de Boyscout rule, laat het kampeerterrein schoon achter voor de volgende gebruiker en liefst nog iets schoner. Ook hier voer je dus verbeteringen uit stapje voor stapje, hapje voor hapje. Ik ben een groot voorstander van “Make-it-work-then-make-it-better-then-make-it-pretty”.

> MAKE IT WORK THEN MAKE IT BETTER THEN MAKE IT PRETTY

Want hoe migreer je vanuit een bestaande omgeving naar een service georiënteerde omgeving of microservices. Het antwoord is heel simpel zoals je ook een olifant eet, hapje voor hapje. “Haha wat leuk, neem je ons niet serieus of zo?”. Jazeker wel en ik zal jullie uitleggen hoe ik dat bedoel.



FIGUUR 1

Door de conferenties en artikelen (waar Microservice behoorlijk gehypet worden) hebben veel bedrijven de wens om over te stappen naar een service georiënteerde architectuur en bij voorkeur naar een Microservices architectuur. Maar natuurlijk moet de winkel wel open blijven en dat betekent dat support en bigfixes wel gewoon

moeten door blijven gaan. Tenslotte betalen de huidige klanten de rekeningen en organisaties hebben niet de mogelijkheid om voor een paar maanden of soms jaren onder een steen te kruipen om orde op zaken te stellen. Natuurlijk kan dat niet, tenslotte zijn klanten de reden waarom het bedrijf bestaat en zij betalen niet voor een product dat niet verbetert, wordt onderhouden en gewoon stilstaat. Ander risico is dat er een andere partij opstaat en jouw plekje inneemt.

Ik denk ook niet dat je dicht hoeft om tijdens het rijden een wiel te verwisselen. Alleen moet je dan als bedrijf wel een behoorlijke tijd uittrekken om te migreren. Je zult dat wel dubbel onderhoud en dubbele code moeten accepteren. Wat vaak in jaren is opgebouwd, is niet in een paar weken terug te draaien. En doordat je niet alles vanaf de grond opnieuw bouwt, zul je meerdere refactor slagen moeten maken en Projects/files/classes/code meerdere malen zult aanpassen. Het is pas af als jullie puntje op de horizon gehaald is.

Ja, dat is niet exclusief voor Service Oriëntatie! Klopt, want het begint bij het ontrafelen van de kluwen van code en zorgen dat alle onbedoelde maar ingeslopen afhankelijkheden weer ongedaan gemaakt worden. Pas als je de basis op orde hebt, dan kun je de stap voorwaarts maken en nieuwe dingen toevoegen. Ook microservices (hoe hip ook) zijn niet per se nodig. Wel wil je dat verantwoordelijkheden goed verdeeld zijn over de verschillende servers, computer systemen of code.

Eerst moeten we voor ons zelf bepalen, wat de reden is om te migreren naar Service Oriëntatie. Wat zijn de huidige uitdagingen met je applicatie, platform of service? Is het dat het testen niet meer te doen is? Of dat het aanpassen van code op de ene plek betekent dat er op een andere plek code / functionaliteit omvalt? Er te veel afhankelijkheden tussen verschillende onderdelen bestaan? Of dat het deployen een probleem is geworden en gewoon te lang duurt? Of wil de organisatie van het ASP (Application Service Provider) model af en omvormen naar overstappen naar een SAAS (Software As A Service) model? Is de applicatie, service of platform eigenlijk niet geschikt voor multi users omgeving? Of is je oplossing niet of nauwelijks in staat op te schalen bij meer gebruik? Is in het algemeen reageren op gewijzigde wensen van je klanten gewoon lastig, omdat er te vlug andere functionaliteit omvalt.

> TECHNOLOGY IS EEN MIDDEL
> EN NIET EEN DOEL.

Belangrijk is, dat de reden niet moet zijn omdat iedereen het doet of dat services of erger microservices zo belangrijk lijken in de business. Op zichtzelf hoeft er niks mis te zijn aan een monoliet. Er moeten andere redenen zijn om aan te passen. Ik heb liever een goede monoliet dan een slecht uitgevoerde Service Oriëntatie. Technology is een middel en moet nooit een doel op zichzelf zijn.

De formele beschrijving van een Service Orientation Architecture service: “A SOA service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online. SOA is also intended to be independent of vendors, products and technologies.”. Microservices zijn opzichzelf niet heel anders, maar waar we de definitie van een op zichzelf staande eenheid van functionaliteit wel iets strakker wordt neergezet. In een minder goed uitgevoerde Service Oriëntatie omgevingen zijn de services vaak te klein, waardoor de services elkaar aanroepen en eigenlijk niet meer los van elkaar gebruikt worden. Dat resulteert meestal in een Chatty systeem, waarbij services elkaar herhaaldelijk aanroepen en allesbehalve onafhankelijk van elkaar zijn. Iedere aanroep van een service levert weer vertraging op en als het een losse service is. Soms zijn de services dan ook nog eens zo gebouwd, dat ze niet verwachten dat de aanroepende service stuk of niet aanwezig is. En dat gebeurt juist als het zo losgekoppeld is. Waarmee de “updated independently” uit de definitie eigenlijk niet meer kan, andere services rekenen op elkaar en feitelijk heb je dan een monoliet in stukjes.

Belangrijkste in de definitie is volgens mij ook op zichzelf staande functionaliteit. Er staat nergens hoe groot de “op zichzelf staande functionaliteit” moet zijn. Wat er wel staat, alles wat de service nodig heeft om volledig autonoom te kunnen werken onafhankelijk van de rest van de systemen. Even heel gechargeerd, een monoliet is in deze beschrijving dus ook een goede service. Een monoliet herbergt misschien te veel verschillende functionaliteiten. Zoals een ordersysteem, een productsysteem, een klantsysteem etc. Ik schrijf expliciet systeem om aan te geven dat het losse onderdelen moeten zijn. Maar het ordersysteem heeft informatie nodig van de producten en de klanten! Ja, dat klopt, maar het ordersysteem zal niet het adres van de klant updaten in de database. Ieder systeem heeft zijn eigen verantwoordelijkheden. Het ordersysteem gebruikt de data van het klantsysteem. Verandert het adres in het ordersysteem? Dan maakt het ordersysteem een werkbon voor het klantsysteem om dit aan te passen. Voor hier gaat het te ver om hier dieper op in te gaan, lees het artikel van Dennis van der Stelt uit ons vorige magazine eens over het Starbucks model. Vuistregel is, maak je functionaliteit niet te klein en let goed op welke verantwoordelijkheden de functionaliteit heeft. Services zijn ook een middel en niet een doel. Keep it simple, maar om met Einstein te spreken; maak het niet simpeler dan dat.

Terug naar ons initiële probleem. We zien de monoliet als een samengeklonterde functionaliteit, maar feitelijk bestaat deze toch ook uit losse stukken en op zichzelf staande functionaliteit. Echter de functionaliteit is alleen te veel op de achtergrond geraakt door gebruik van technology. Vaak zijn databases te ver uitgenormaliseerd (8st normaalvorm) of is object Oriëntatie te strict doorgevoerd.

FIGUUR 2

Heel vaak is door technologische overload de functionaliteit teveel door elkaar heen gaan lopen. De technologie geeft de mogelijkheid en de druk om snel nieuwe functionaliteit er uit te sturen neemt dat ook toe.

Maar als je het goed doet, dan kun je het best weer naar zijn oorsprong ontrafelen. Je zult dan strict bezig moeten zijn met de functionaliteit en de technologie weer een middel laten zijn.

> MAAK EEN VERTICAL SLICE VAN JE FUNCTIONALITEIT

De volgende stappen kunnen dan gedaan worden. Bepaal een stuk functionalitieit dat op zichzelf staat of had moeten staan. Beschrijf het dan van de user interface tot aan de kern (de dataopslag of de servicebus of wat ook het eindpunt is). En zodra je een afhankelijkheid spot, bedenk dan steeds of deze afhankelijkheid noodzakelijk is en of er andere oplossingen zijn. Als er geen eindpunt is, dan is er misschien een uitdaging met de functionaliteit zelf.
Populair gezegd maak een verticale doorsnede (vertical slice) van je applicatie. Zie het als de taart met die vele gekleurde laagjes, beschrijf alles van dat ene stuk uitgesneden taart en alleen wat bij dat stuk hoort. Bouw dan dat stuk opnieuw en hergebruik zo min mogelijk eerdere code. Anders word je te vlug weer in de rabbithole getrokken. Bereid je voor op het herhaaldelijk refactoren van de code. Er lijkt geen einde aan te komen, maar uiteindelijk zal het beter worden.

Populair gezegd: maak een verticale doorsnede (vertical slice) van je applicatie. Zie het als de taart met die vele gekleurde laagjes, beschrijf alles van dat ene stuk uitgesneden taart en alleen wat bij dat stuk hoort.

Deze strategie maakt het ook mogelijk om de oude flow van de functionaliteit nog beschikbaar te houden en de nieuwe er echt los van te laten. Uiteindelijk is dat ook je doel in Service Oriëntatie. Als je dan volledig tevreden bent met de nieuwe implementatie, dan zet je de wissel om.

Soms kan de aanpassing ook aan de buitenkant gemaakt worden, waardoor bepaalde werkvoorraad niet eens je platform/applicatie of omgeving bereikt. Controle aan de deur. Beschrijf deze functionaliteit zo onafhankelijk mogelijk en als je of het team de behoefte voelt om andere functionaliteit toe te voegen of samen te voegen, stel dan de waarom vraag zo vaak mogelijk. Als je dan nog steeds niet zonder kunt, dan moet je het samenvoegen niet laten.

Maar iedere andere wijziging verstoort het proces. Dus probeer niet gelijk ook volledig nieuwe functionaliteit toe te voegen of bestaande functionaliteit volledig te veranderen. Niet omdat het niet zou kunnen, maar als we technologie of functionaliteit wijzigingen gaan doorvoeren dan worden we afgeleid en wij mensen kunnen helaas niet teveel multitasken. Het onafhankelijk en losgeweken van het stukje functionaliteit is al moeilijk genoeg. Daarom moet je de eenheid ook zo klein mogelijk maken. Dat voorkomt dat de rest van de applicatie/omgeving op slot zit voor aanpassingen of wijzigingen.

Het is te doen, maar je zult wel heel gericht door moeten gaan en je niet laten afleiden. En als je eenmaal begonnen ben, dan kun je eigenlijk niet meer stoppen. Meer dan eens zullen verschillende stukken software door je handen gaan en meer dan eens zal je code aanpassen om deze later weer ongedaan te maken. Dat geeft niet, je doel is op zichzelf staande stukken software/services.

Niets is onmogelijk. Maar laat technologie niet de reden zijn! En blijf refactoren.

MARCEL MEIJER

Marcel Meijer werkt als Software Developer / Solution Architect bij VX Company. In zijn vrije tijd is hij .NET track owner en bestuurslid van de SDN. Binnen de SDN is hij verantwoordelijk voor het regelen van sprekers voor de SDN Events (SDE), het regelen/redigeren van artikelen voor het SDN Magazine, mede verantwoordelijk voor de eindredactie van de hardcopy en digitale magazines en de inhoud van de SDN Conferences. Sinds 1 oktober 2010 heeft hij een MVP award gekregen.

Steef-Jan Wiggers

EVENT-GEDREVEN OPLOSSING BOUWEN MET

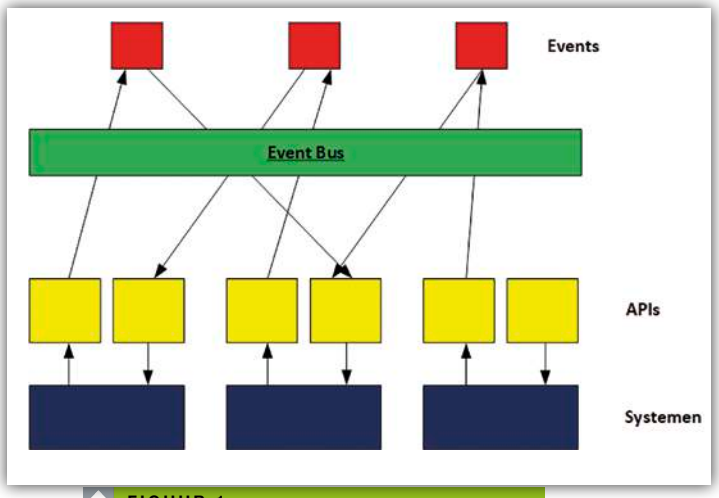
Azure Event Grid

Als solution-architect denk je in oplossingen voor je opdrachtgever. Wanneer dit een oplossing betreft voor een Cloud platform als Azure denk je na over welke bouwblokken lees Clouddiensten je wilt gebruiken. Daarnaast denk je ook over het toepassen van patronen en architectuur principes.

Stel je bevind je als architect in een scenario waarvoor je voor bij een opdrachtgever een oplossing moet bedenken over hoe systemen tijd en eenduidig informatie met elkaar moeten uitwisselen in Azure. Deze systemen zijn Software-as-a-Service (SaaS) diensten zoals ServiceNow, Salesforce, en Zendesk. Een verandering van gegevens in één systeem moet leiden tot uitwisseling van gegevens met een ander systeem of systemen. Een verandering kan in de vorm van generen van events leiden tot de gegevensuitwisseling. Je komt dan mogelijk tot de conclusie dat je een event-gedreven oplossing wilt ontwerpen en laten implementeren.

Het patroon voor het bovenstaande scenario kan event-gedreven architectuur principes bevatten als het gaat om genereren, detecteren, en consumeren van events. Een event kan de zogenaamde status bevatten van bijvoorbeeld een organisatie object in Salesforce, die veranderd is van 'prospect' naar 'customer'. Deze verandering leidt tot uitzenden van een event, wat moet leiden tot veranderingen in een ander systeem of systemen of te wel uitwisselingen van informatie van het betreffende organisatie object.

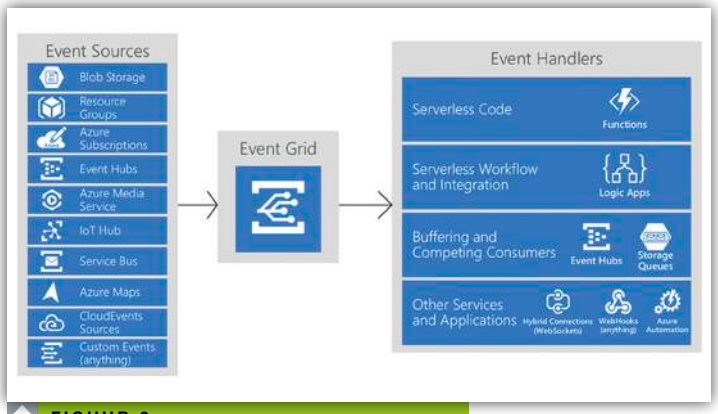
Het voordeel van een event-gedreven architectuur is dat informatiesystemen real-time kunnen reageren op events wat een natuurgetrouwe benadering oplevert van de werkelijkheid - en minder complex kan zijn dan traditionele procedurele benaderingen. Daarbij is het essentieel dat systemen niet direct met elkaar gekoppeld zijn en enkel met elkaar communiceren op basis van meldingen (events). Dus naast tijdig informatie uitwisselen biedt een dergelijke architectuur benadering ook loskoppelingen van systemen met zich mee met alle voordelen van dien. Systemen communiceren via APIs met elkaar door middel van berichten en meldingen. Verder zijn systemen op deze manier uitwisselbaar van elkaar.



FIGUUR 1

Microsoft biedt op haar Azure platform een dienst aan genaamd Event Grid (https://azure.microsoft.com/en-us/services/event-grid/). Met deze dienst kan je centraal events managen, wat inhoudt dat systemen of services een event kunnen versturen (produceren) naar een zogenaamde Event Grid Topic - waar vervolgens een of meerdere afnemers (abonneren) de events kunnen ontvangen. De dienst kan events routeren op basis van een specifieke filter, ingesteld voor een bepaalde afnemer. Met filtering in de dienst kan dus op een intelligente manier events gedistribueerd worden naar diverse afnemers.

De afnemers van events worden ook wel 'Event Handlers' genoemd en producenten van events 'Event Sources'. Het onderstaande diagram laat een overzicht zien van de producenten, Event Grid en de afnemers van events (handlers).



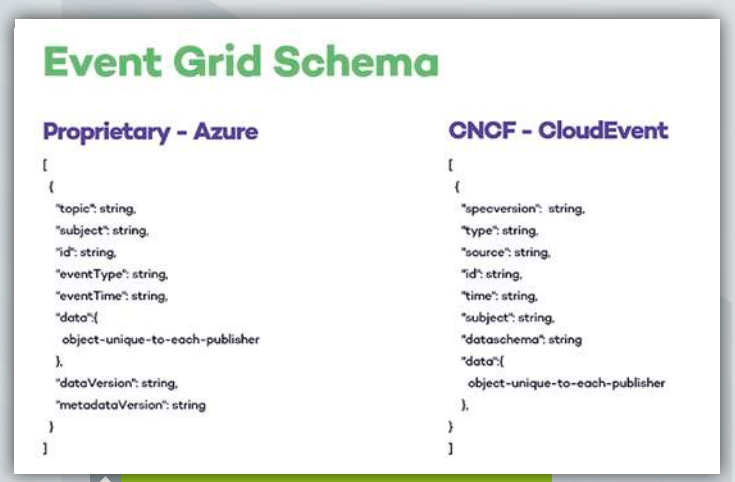
FIGUUR 2

Bron: https://docs.microsoft.com/en-us/azure/communication-services/concepts/event-handling

Binnen het Azure platform zijn er diverse diensten, die Event Grid ondersteunen en events uitsturen zoals bijvoorbeeld er is een blob aangemaakt (blobCreated) in een storage container. Daarnaast kan een systeem of applicatie een zogenaamde 'custom' event laten sturen. En als laatste kan een systeem of applicatie in de Cloud ook een zogenaamde 'Cloud Event' uitsturen, hetgeen later in dit artikel wordt beschreven. Al deze events komen dus van een 'Event Source', die events stuurt naar een Event Grid topic, waar een of meerdere afnemers een abonnement hebben.

De afnemers kunnen diverse soorten applicaties of services zijn zoals het diagram weergeeft. Een Azure Function kan door middel van een zogenaamde 'WebHook' events ontvangen en vervolgens verwerken. Hetzelfde geldt voor een Logic App, een workflow dienst in Azure. Verder kunnen events worden afgeleverd aan een Event Hub en Storage Queues voor buffering en latere verwerking. En tot slot kunnen andere diensten events ontvangen voor verwerking.

De events, die worden geproduceerd dienen zich te conformeren aan een Javascript Object Notation (JSON) schema. Dit schema kan zijn het schema, welke Microsoft heeft gedefinieerd voor de dienst Event Grid (https://docs.microsoft.com/en-us/azure/event-grid/event-schema) of het Cloud Native Computing Foundation (CNCF) schema of te wel CloudEvent schema (https://docs.microsoft.com/en-us/azure/event-grid/cloud-event-schema). Beide schema's worden door Azure Event Grid ondersteund. Inhoudelijk komen de schema's enigszins overeen, echter verschillen de namen van de eigenschappen (properties).



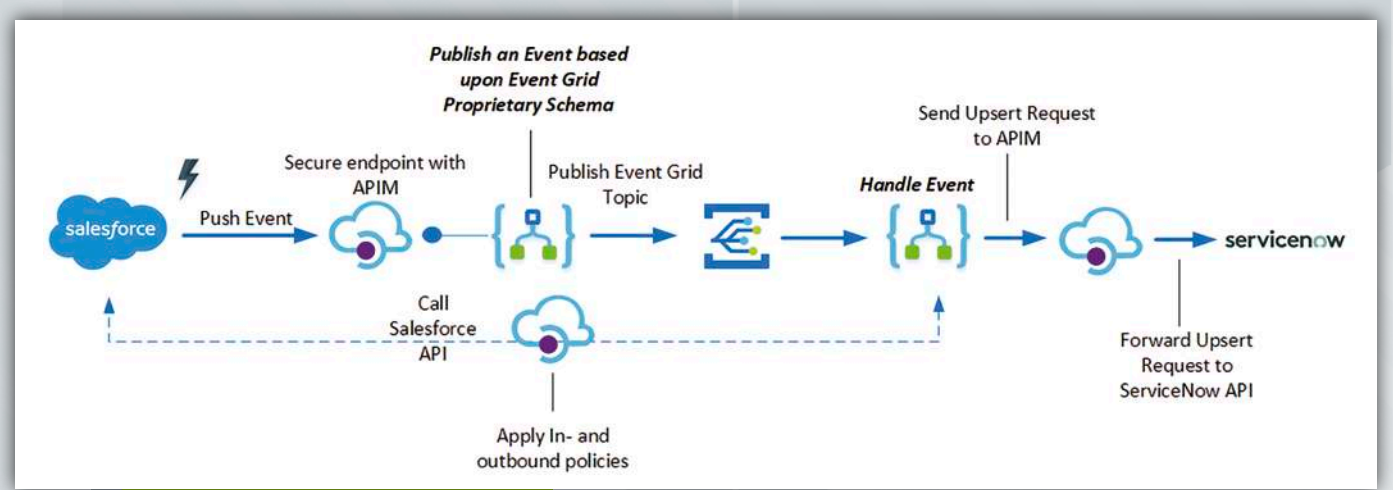
FIGUUR 3

In het Microsoft-schema vind je eigenschappen (properties) terug, die het een en ander over een event beschrijft – zoals een unieke identificatie (id), onderwerp (subject), type (eventType), datum en tijd (eventTime) van het event, en data van het event (data). Het laatste is een eigenschap van het event, waar meerdere eigenschappen in terug te vinden zijn – afhankelijk van de bron (Event Source).

In het data eigenschap (property) kan, bijvoorbeeld informatie bevatten van een blob, die is aangemaakt in een storage container. Verder heeft de versie (dataVersion) betrekking op de data in het data-property en meta-data versie (metaDataVersion) op het event schema eigenschappen in zijn geheel.

```
"data": {
    "api": "PutBlockList",
    "clientRequestId": "6d79dbfb-0e37-4fc4-981f-
442c9ca65760",
    "requestId": "831e1650-001e-001b-66ab-
eeb76e000000",
    "eTag": "\"0x8D4BCC2E4835CD0\"",
    "contentType": "text/plain",
    "contentLength": 524288,
    "blobType": "BlockBlob",
    "url": "https://my-storage-account.blob.core.
windows.net/testcontainer/new-file.txt",
    "sequencer": "00000000000044200000000000028963",
    "storageDiagnostics": {
      "batchId": "b68529f3-68cd-4744-baa4-
3c0498ec19f0"
    }
```

Wat betreft het CloudEvent schema kun je zien dat de eigenschappen vergelijkbaar zijn, echter is dit schema een initiatief van het CNCF waar meerdere partijen zich verbonden hebben om dit te ondersteunen. Je kunt CloudEvent schema, dus ook beschouwen als een industriestandaard voor events.
Met de eigenschappen in een event kunnen filters worden toegepast voor routering naar de gewenste afnemers. De filter opties, die Event Grid levert zijn op onderwerp (subject) en type event. Verder kunnen er meer specifieke filters worden aangemaakt op de eigenschappen binnen het data-property.



FIGUUR 4

Stel het volgende scenario voor waarbij Azure Event Grid een rol speelt in het tijdig uitwisselen van informatie tussen twee SaaS-systemen. Een object bijvoorbeeld het klant object X veranderd van status in het Salesforce systeem van bedrijf Y. Een klant X gaat meer diensten afnemen van bedrijf Y, waarbij de diensten uitlevering geregeld wordt via ServiceNow systeem van bedrijf Y. Dat betekent dat de status wijzing van het klant X object moeten worden doorgegeven aan

ServiceNow. In het onderstaande diagram staat de oplossing voor de gegevensuitwisseling tussen Salesforce en ServiceNow weergegeven.

Salesfore kan een event sturen via een https endpoint van een Logic App, die ontsloten is via een Azure API Management proxy (API-definitie). In dit scenario stuurt Salesforce een algemeen event uit wat niet voldoet aan een Azure Event Grid schema of CloudEvent schema. Dat zou wel kunnen als een Salesforce consultant het uitsturen van een event volgens een van deze schema's definieert, echter hoeft dat niet. Om implementatie van uitsturen simpel en generiek te houden kan een Salesforce consultant daarvoor kiezen (bijvoorbeeld enkel de object naam en identifier) en het Azure Platform de vertaling naar het schema en de routering naar de juiste afnemers te laten verzorgen (Event Grid).

De Logic App in deze ontvangt elke event, die generiek is opgezet en vertaald de inhoud naar een event volgens het Event Grid schema. De Logic App publiceert vervolgens het event op een Event Grid topic. Vervolgens is daar een Logic App op geabonneerd, die het event ontvangt en met de inhoud (object en de identifier) informatie op gaat halen bij Salesforce. In dit geval bevat het event informatie van klant X, waar de status wijziging heeft plaatsgevonden. Uitvragen gaat eveneens via een proxy (API-definitie) van de Salesforce API. Beide proxies (APIs) in API Management verzorgen de authenticatie/autorisatie tussen Salesforce en Azure componenten (Logic Apps). De informatie, die de Logic App vervolgens ontvangt wordt door gegeven aan de ServiceNow instantie via wederom een proxy (API) in API Management.
Door middel van Azure Event Grid worden status wijzingen in objecten in Salesforce direct doorgegeven aan geïnteresseerde afnemers, die daar direct op kunnen acteren zoals in het voorbeeld wat eerder is beschreven. Events worden via Event Grid meteen doorgeven (push) en niet zoals bij een Service Bus Queue of Topic bewaard tot dat een afnemer het komt halen (pull). Met Azure Event Grid heb je net als bij de Service Bus een 'publish-subscribe' model, echter bij de een is het 'push-push' en bij de andere 'push-pull'. Dus voor 'near real-time' verwerking van data is een event gedreven service een betere optie. Hetgeen één van de principes is van een event-gedreven architectuur.

Met Azure Event Grid ben je in staat op een event-gedreven oplossing te realiseren op het Azure platform. En meer dan dat zelfs kan je een volledige op event-gedreven architectuur principes een oplossing bouwen. Op moment dat tijdig informatie-uitwisselingen tussen systemen in de Cloud cruciaal is dan biedt een oplossing met Azure Event Grid een uitkomst. Je kunt meer informatie vinden over Azure Event Grid in de Microsoft documentatie (https://docs.microsoft.com/en-us/azure/event-grid/) en Microsoft Architecture center (https://docs.microsoft.com/en-us/azure/architecture/browse/?terms=Event%20Grid).

STEEF-JAN WIGGERS

Steef-Jan Wiggers is werkzaam als Azure Architect bij Wortell. Hij is zeer actief binnen de Azure en Integratie community, internationaal spreker en auteur van diverse White papers en (e)Books. Verder is hij Cloud journalist bij InfoQ en schrijft hij nieuwsartikelen rondom Azure, AWS en GCP.

Johan Smarius

gRPC-Web en Blazor

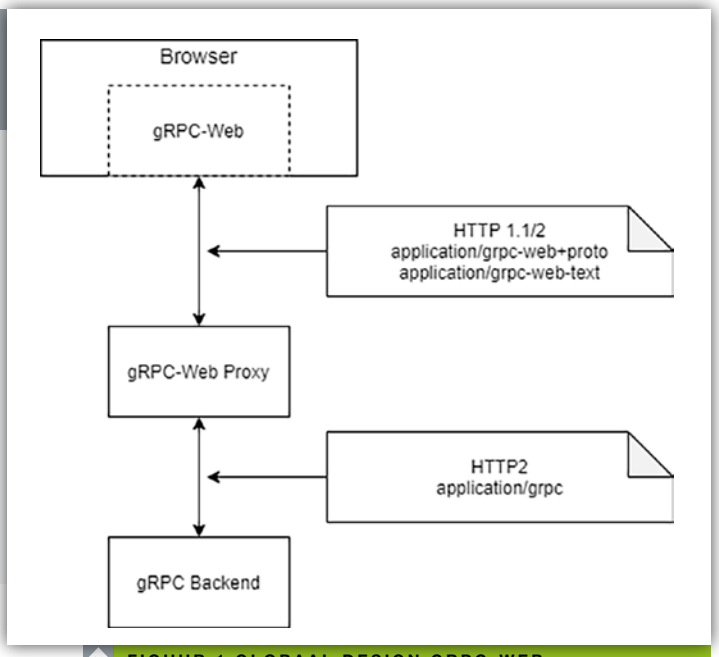
In de vorige editie van het magazine heb ik een artikel geschreven over hoe je gRPC kunt gebruiken binnen .NET. Binnen een webapplicatie is het nog niet eenvoudig om gebruik te maken van gRPC. Gelukkig is er nog een alternatief in de vorm van gRPC-Web. In dit artikel zal ik jullie laten zien hoe je kunt werken met gRPC-Web in combinatie met een andere Microsoft technologie: Blazor. Om dit artikel te kunnen begrijpen, heb je geen uitgebreide voorkennis van Blazor nodig.

WAT IS GRPC-WEB?

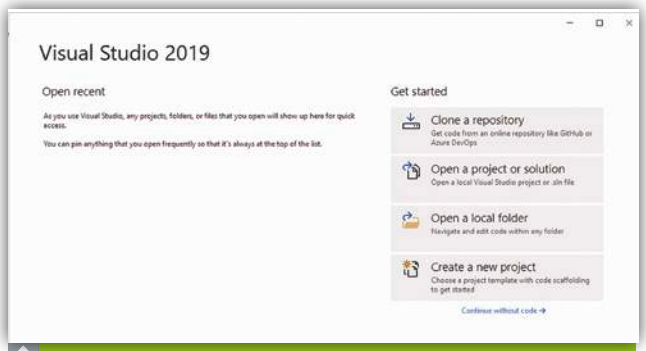
gRPC is gebaseerd op het HTTP/2 protocol. Om gRPC volledig te kunnen laten werken, is veel controle op laag niveau nodig over de berichten die tussen client en server gestuurd worden. Een moderne webbrowser heeft op dit moment nog niet de mogelijkheid om berichten op dit niveau volledig te ondersteunen. Om toch gRPC bruikbaar te maken binnen een webbrowser, kun je gebruik maken van gRPC-Web. gRPC-Web en gRPC zijn onder de motorkap wel verschillende protocollen. Ze zijn alleen ontworpen om met elkaar samen te kunnen werken. De voornaamste functie van gRPC-Web is te fungeren als proxy tussen de browser/client en de gRPC backend. Het globale design van gRPC-Web staat in Figuur 1. De content typen die typisch gebruikt worden voor het verkeer tussen de componenten staan ook in deze afbeelding.

OPZETTEN VAN EEN GRPC-WEB SERVER

Voor het opzetten van een gRPC-Web server ga ik gebruik maken van de standaard gegeneerde code voor een hosted Blazor WebAssembly applicatie. Een Blazor WebAssembly applicatie maakt gebruik van WebAssembly om .NET-code in de browser te draaien. WebAssembly wordt in de webbrowser uitgevoerd in een soort virtual machine die je een beetje kunt vergelijken met de JavaScript virtual machine. Je kunt dus een single page application (SPA) met C# maken, zonder (veel) JavaScript code te hoeven schrijven. Een SPA heeft voor de informatie die hij moet tonen een backend nodig. De hosted variant genereert een backend op basis van ASP.NET Core die voor het ophalen van de Blazor-code door de browser wordt gebruikt en die als WebService dient voor diezelfde Blazor applicatie.

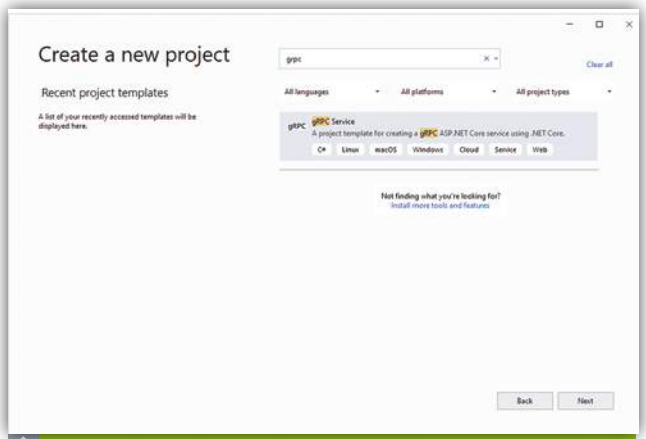


FIGUUR 1:GLOBAAL DESIGN GRPC-WEB



FIGUUR 2: STARTSCHERM VISUAL STUDIO 2019

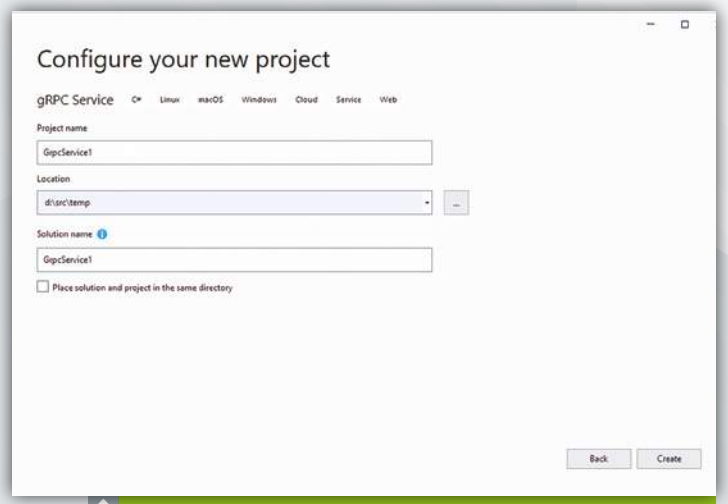
In het startscherm kun je kiezen voor “Create a new project” (Figuur 2).



FIGUUR 3: CREATE PROJECT PAGE

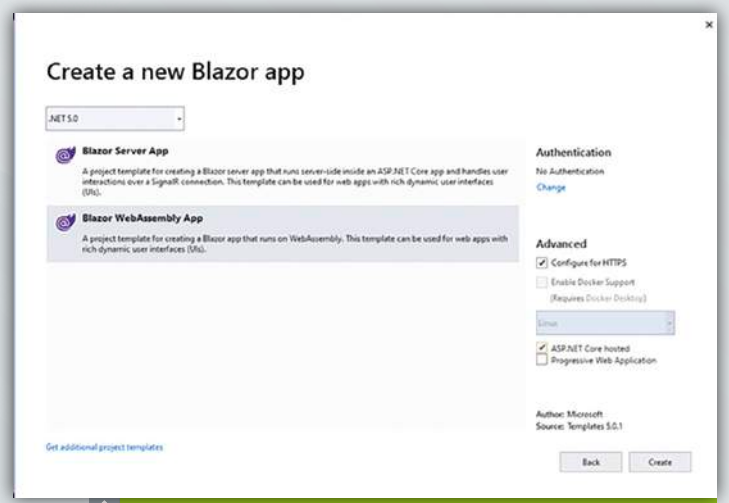
In de templatepagina (Figuur 3) kun je nu zoeken naar “Blazor App”. De verschillende varianten van Blazor vallen allemaal onder deze template. Als je deze template kiest en op “Next” drukt, dan kom je

terecht op de pagina die voor alle templates gebruikt wordt om de project naam, locatie en solution name in te stellen (Figuur 4).



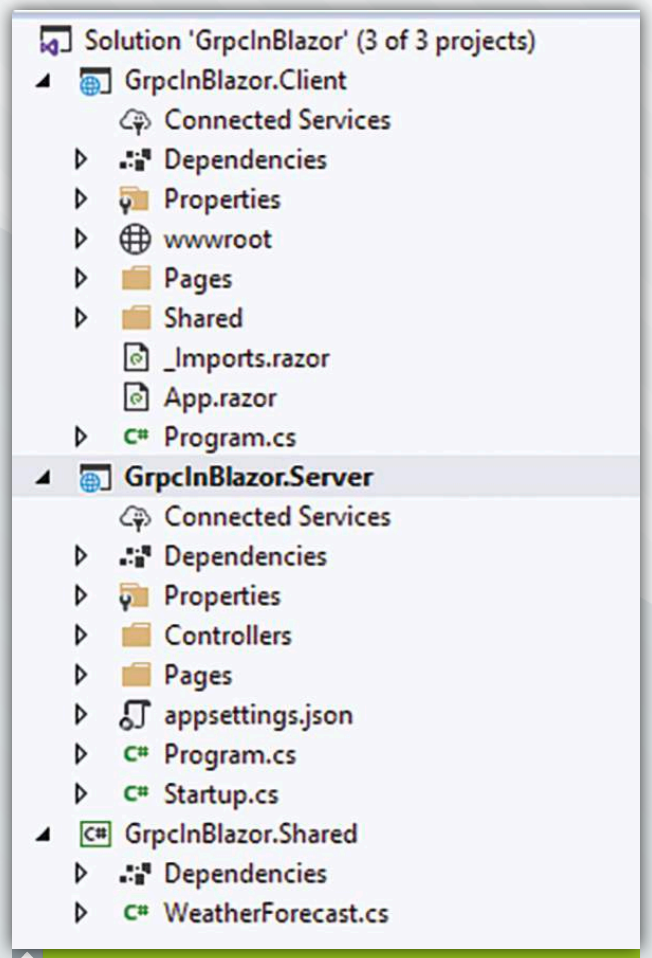
FIGUUR 4: PROJECTCONFIGURATIE

Als je nu op “Create” klikt, dan kun je aanvullende opties instellen voor je Blazor applicatie (Figuur 5). Belangrijk is om bij deze configuratie te kiezen voor .NET 5 als framework. Voor dit artikel gebruiken we de echte SPA-variant van Blazor die volledig in de browser draait, dus de “Blazor WebAssembly App” variant. Voor een webapplicatie kun je het beste HTTPS altijd aan laten staan. Om een standaard server te krijgen, is het belangrijk om de optie “ASP.NET Core hosted” aan te zetten. Als je nu weer op “Create” drukt, dan wordt de code voor je front-end en server gegenereerd.



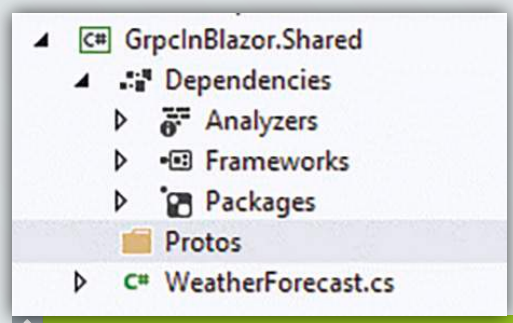
FIGUUR 5: BLAZOR CONFIGURATIE

In de gegeneerde code (Figuur 6) kun je zien dat er code is gegeneerd voor de client en voor de server. Bovendien is er een Shared-project aangemaakt waarmee code kan worden gedeeld tussen de client en de server.
Binnen .NET 5 kun je op twee manieren gebruik maken van gRPC-Web. Je kunt gebruik maken van de Envoy proxy om berichten te vertalen tussen gRPC-Web en gRPC of je kunt gebruik maken van de middleware optie in ASP.NET Core. In dit artikel zal ik de laatste optie behandelen, omdat die het beste aansluit bij de manier waarop we standaard in ASP.NET Core werken.
Voor het opzetten van een gRPC service moet ik eerst het interface definitie bestand maken in protobuf. Dit bestand moet worden gedeeld



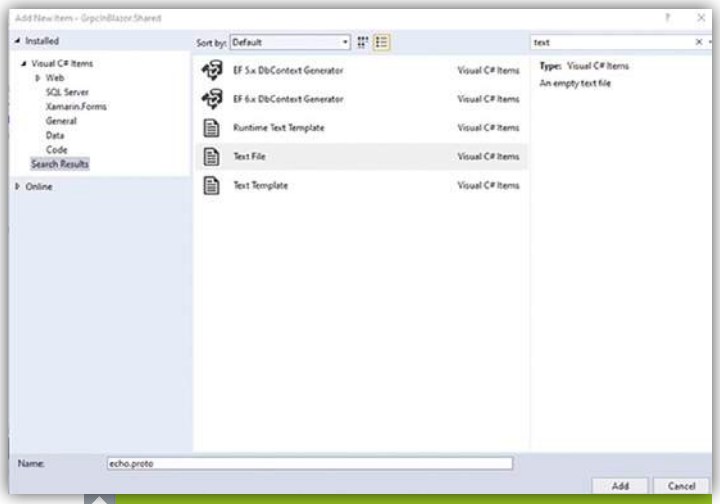
FIGUUR 6: INITIEEL GEGENEERDE CODE

tussen de server en de client, dus ik kies ervoor om dit bestand aan te maken in het Shared-project. Het Shared-project heeft natuurlijk nog geen ondersteuning voor gRPC, dus ik moet de NuGet-packages “Grpc.Net.Client”, “Grpc.Tools” en “Google.Protobuf” toevoegen aan het Shared-project.
De conventie binnen gRPC is om de protobuf-bestanden te plaatsen in een folder genaamd “Protos”. Ik heb deze dan ook aangemaakt binnen het Shared-project (Figuur 7).



FIGUUR 7: PROTOS-FOLDER IN SHARED PROJECT

In de Protos-folder kun je nu een protobuf-bestand plaatsen. Helaas kun je via de optie “Add New Item” niet direct een protobuf-bestand toevoegen. Gelukkig is een protobuf-bestand eigenlijk niets anders dan een tekstbestand, dus je kunt gewoon voor die optie kiezen en dan “.proto” als extensie voor het bestand gebruiken (Figuur 8).



FIGUUR 8: TOEVOEGEN PROTOBUF-BESTAND

Aan de bovenkant van het lege bestand moet je nu zelf wel de versie voor de syntax en de optie voor de namespace opgeven (Figuur 9).

```
1  syntax = "proto3";
2
3  option csharp_namespace = "BlazorGrpcWeb.Protos";
```

FIGUUR 9: STANDAARD CODEREGELS VOOR PROTOBUF-BESTAND

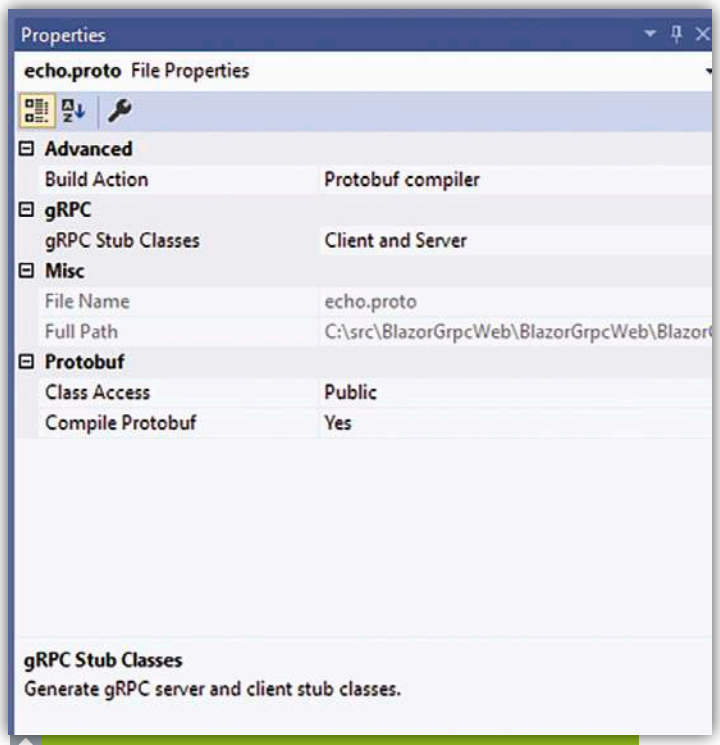
Voor de eerste gRPC-service ga ik gebruik maken van een simpele echo-service (Figuur 10).

```
service EchoService {
    rpc EchoMessage(EchoRequest) returns (EchoResponse);
}

message EchoRequest {
    string name = 1;
}

message EchoResponse {
    string name = 1;
}
```

FIGUUR 10: PROTOBUF DEFINITIE VOOR ECHO SERVICE

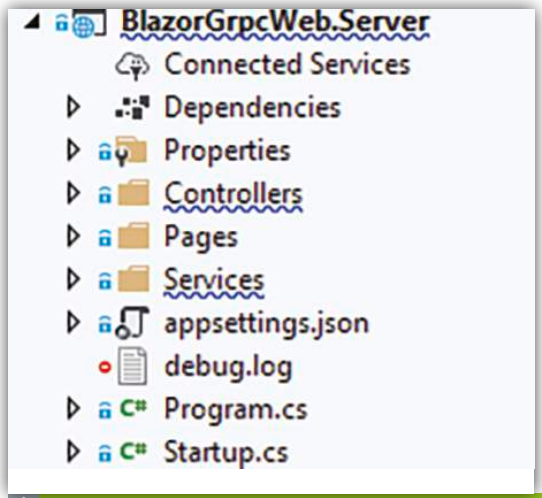
Het opzetten van de protobuf-definitie is exact gelijk aan gRPC. Je merkt hier dus nog helemaal niets van gRPC-Web.
Binnen het Shared-project wil ik graag de code laten genereren voor de server- en voor de clientkant (Figuur 11).

Belangrijk om hierbij als build actie natuurlijk “Protobuf compiler” te kiezen en voor de gRPC Stub classes nu te kiezen voor “Client and Server”.



FIGUUR 11: INSTELLEN PROTOBUF PROPERTIES

Binnen het Server-project kan ik nu de code gaan schrijven voor mijn Echo-service. Binnen gRPC is het gebruikelijk om de servicecode in de folder services te zetten. Deze standaard houd ik ook aan in deze opzet (Figuur 12).



FIGUUR 12: SERVICES FOLDER IN SERVER PROJECT

Om de servercode te kunnen schrijven, zijn een aantal NuGet-packages nodig: “Grpc.AspNetCore.Web”, “Grpc.AspnetCore” en natuurlijk “Google.Protobuf”.
De code voor de Echo-service is heel eenvoudig (Figuur 13).

Om gRPC te kunnen gebruiken in een service moet in de methode ConfigureServices in de Startup-klasse, de ondersteuning voor gRPC wel toegevoegd worden (Figuur 14).

gRPC-Web en Blazor

```
4 references | Johan Smarius, 34 days ago | 1 author, 1 change
public class EchoServiceImpl : EchoService.EchoServiceBase
{
    private readonly ILogger<EchoServiceImpl> _logger;

    0 references | Johan Smarius, 34 days ago | 1 author, 1 change
    public EchoServiceImpl(ILogger<EchoServiceImpl> logger)
    {
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    3 references | Johan Smarius, 34 days ago | 1 author, 1 change
    public override Task<EchoResponse> EchoMessage(EchoRequest request, ServerCallContext context)
    {
        var responseMessage = $"Echo: {request.Name}";

        var echoResponse = new EchoResponse { Name = responseMessage };

        return Task.FromResult<EchoResponse>(echoResponse);
    }
}
```

FIGUUR 13: IMPLEMENTATIE ECHOSERVICE

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddGrpc();
}
```

FIGUUR 14: TOEVOEGEN GRPC ONDERSTEUNING

In de Configure methode moet gRPC-Web ondersteuning aan de middleware-pipeline toe worden gevoegd door gebruik te maken van de methode UseGrpcWeb. De plaatsing van deze middleware is heel belangrijk. Deze moet namelijk tussen UseRouting en UseEndpoints toe worden gevoegd (Figuur 15).

```
app.UseRouting();

app.UseGrpcWeb();

app.UseEndpoints(endpoints =>
```

FIGUUR 15: INSTELLEN MIDDLEWARE

De service moet geregisteerd worden in de routing endpoints met behulp van de methode MapGrpcService. Om deze service ook beschikbaar te maken als gRPC-Web variant, moet EnableGrpcWeb op dit endpoint geconfigureerd worden (Figuur 16).

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGrpcService<EchoServiceImpl>().EnableGrpcWeb();
    endpoints.MapRazorPages();
```

FIGUUR 16: TOEVOEGEN ENDPOINT

Het opzetten van een gRPC-Web service is dus bijna gelijk aan het opzetten van een reguliere gRPC-Service. De enige wijzigingen zijn het toevoegen van de NuGet package voor “Grpc.AspNetCore.Web” en de toevoegingen aan de code in de Configure methode. Alleen door gebruik van de Blazor-template moet je een aantal dingen zelf instellen, die Visual Studio voor je had gedaan bij een keuze voor een gRPC-Service.
De code aan de serverkant is hiermee klaar. Aan de clientkant moet ook nog wat code worden geschreven om de ondersteuning in te regelen.
Om de clientcode te kunnen schrijven, zijn een aantal NuGet-packages nodig: “Grpc.Net.ClientFactory”, “Grpc.Net.Client” en “Grpc.Net.Client.Web”.

De standaard codegeneratie voor gRPC in het Shared-project heeft al een client proxy gegenereerd. Ik hoef in de Blazorclient dus geen gebruik meer te maken van “Add Service Reference”, zoals ik dat normaal bij een gRPC-client wel moet doen. De gegenereerde client kunnen we binnen onze Blazorcode gaan gebruiken. Voor het werken met de client zijn twee mogelijkheden: direct gebruik maken van de clientcode en gebruik maken van de clientcode via dependency injection. Ik zal beide mogelijkheden behandelen.

DIRECT GEBRUIK MAKEN VAN DE CLIENTCODE

Binnen de Blazorcode kan ik direct gebruik gaan maken van de client. Net als bij een normale client, moet ik eerst een channel aanmaken tussen de client en de server (Figuur 17).

```
var channel = GrpcChannel.ForAddress("https://localhost:44350/",
    new GrpcChannelOptions
    {
        HttpHandler = new GrpcWebHandler(new HttpClientHandler())
    });
```

FIGUUR 17: AANMAKEN CHANNEL IN CODE

Ik moet op deze channel instellen dat ik gebruik wil maken van Grpc-Web. Dit kan ik doen door de HttpHandler in de GrpcChannelOptions te configureren met de GrpcWebHandler.
Met behulp van deze channel kan ik nu de client aanmaken (Figuur 18).

```
var client = new EchoService.EchoServiceClient(channel);
```

FIGUUR 18: AANMAKEN INSTANTIE CLIENT

En op deze client kan ik nu de methodes aanroepen die ik op de service gedefinieerd heb. Voor de Echoservice is dit natuurlijk maar een methode: EchoMessage. De code die voor de EchoServiceClient gegenereerd wordt, bevat synchrone en asynchrone methodes. Voor Blazor kun je alleen maar gebruik maken van de asynchrone variant (Figuur 19). Dit heeft ook tot gevolg dat de methode die de code bevat zelf ook async moet worden (Figuur 20).

```
var response = await client.EchoMessageAsync(new EchoRequest() {Name = inputFromUser});
```

FIGUUR 19: ASYNCHRONE AANROEP VAN DE METHODE

```
private async Task OnSendMessage()
```

FIGUUR 20: ASYNC AANPASSING VAN DE METHODE

Het resultaat van de aanroep kan nu weer worden gebruikt om een variabele te vullen voor de databinding (Figuur 21).

```
receivedFromServer = response.Name;
```

FIGUUR 21: RESULTAAT TOEKENNEN VOOR DATABINDING

Binnen de markup voor het tonen van het ontvangen bericht is helemaal geen specifieke gPRC-code nodig (Figuur 22).
Als ik de server nu draai, dan wordt meteen ook de client opgestart. Ik kan nu de invoer in de Blazor component laten echoën door mijn EchoService (Figuur 23).

```
@if (!string.IsNullOrEmpty(receivedFromServer))
{
    <div class="alert alert-success">@receivedFromServer</div>
}
```

FIGUUR 22: MARKUP VOOR HET TONEN VAN HET RESULTAAT

FIGUUR 23: ECHO SERVICE AAN HET WERK

Gebruik maken van de clientcode via dependency injection

Standaard wordt voor een Blazor-applicatie een weersvoorspellingsservice op basis van een WebAPI-endpoint aangemaakt. Ik heb hier een gRPC-variant van gemaakt. De protobuf definitie hiervoor staat in Figuur 24. Voor de WeatherForecastService heb ik ook weer client- en servercode laten genereren door de gRPC-tooling in het Shared-project.

```
syntax = "proto3";

option csharp_namespace = "BlazorGrpcWeb.Protos";

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";

service WeatherForecastService {
    rpc GetForecasts(google.protobuf.Empty) returns (WeatherForecastResponse);
}

message WeatherForecastDTO
{
    google.protobuf.Timestamp Date = 1;
    int32 TemperatureC = 2;
    string Summary = 3;
}

message WeatherForecastResponse {
    repeated WeatherForecastDTO Forecasts = 1;
}
```

FIGUUR 24: PROTOBUF DEFINITIE VAN DE WEERVOORSPELLINGSSERVICE

Net als ASP.NET Core kan Blazor gebruik maken van dependency injection in de componenten, dus de visuele onderdelen van je applicatie. De code van de componenten hoeft dan geen weet te hebben van de exacte configuratie van de channel en van de client. Om dit mogelijk te maken, moet de client natuurlijk wel toegevoegd worden aan de services collection (Figuur 25).

```
builder.Services.AddGrpcClient<WeatherForecastService.WeatherForecastServiceClient>((services, options) =>
{
    options.Address = new Uri("https://localhost:44350/");
}).ConfigurePrimaryHttpMessageHandler(() => new GrpcWebHandler(GrpcWebMode.GrpcWebText, new HttpClientHandler()));
```

FIGUUR 25: TOEVOEGEN VAN DE CLIENT AAN SERVICES COLLECTION

Deze client moet ik natuurlijk ook van de benodigde instellingen voorzien. Het adres van de endpoint kan ik gewoon via de options instellen. Voor het instellen van de GrpcWebHandler moet ik gebruik maken van de extension method “ConfigurePrimaryHttpMessageHandler”. Met behulp van een lambda expressie kan ik de client nu zo configureren dat deze gebruik maakt van de “GrpcWebMode.GrpcWebText”. Dit content type is nodig voor gRPC-Web.

Binnen de component kan ik nu gewoon de client injecteren bovenaan in het bestand door gebruik te maken van het keyword inject (Figuur 26).

```
@inject WeatherForecastService.WeatherForecastServiceClient WeatherForecastClient
```

FIGUUR 26: INJECT IN MARKUP COMPONENT

Blazor biedt natuurlijk ook de mogelijkheid om de injectie te configureren in de codesectie (Figuur 27).

```
[Inject]
private WeatherForecastStreamingService.WeatherForecastStreamingServiceClient WeatherForecastClient { get; set; }
```

FIGUUR 27: INJECTEREN IN DE CODE

Welke variant je gebruikt, hangt erg af van je persoonlijke voorkeur. In de voorbeeldcode die bij dit artikel hoort, wordt bewust gebruik gemaakt van beide varianten voor de verschillende clients. In de praktijk zul je normaal voor een variant kiezen en deze consequent gebruiken in een project.

Nadat de client geïnjecteerd is, kan deze direct gebruikt worden in de eigen code (Figuur 28).

```
var result = await WeatherForecastClient.GetForecastsAsync(new Empty());

forecasts = result.Forecasts.ToArray();
```

FIGUUR 28: GEBRUIK CLIENT IN EIGEN CODE

Doordat ik in mijn forecasts nu gebruik maak van de code die door gRPC is gegenereerd, moet ik in mijn templatecode nog wel rekening houden met het bestaan van de Date property van het type TimeStamp door op deze property ToDateTime() aan te roepen voordat ik de formattering toe kan passen (Figuur 29).

```
<tr>
    <td>@forecast.Date.ToDateTime().ToShortDateString()</td>
    <td>@forecast.TemperatureC</td>
    <td>@forecast.Summary</td>
</tr>
```

FIGUUR 29: AANROEPEN TODATETIME() OP TIMESTAMP PROPERTY

De code binnen de component blijft met deze optie om met de client om te gaan een stuk schoner, dus in de praktijk gebruik ik zelf deze laatste variant om met de client om te gaan.

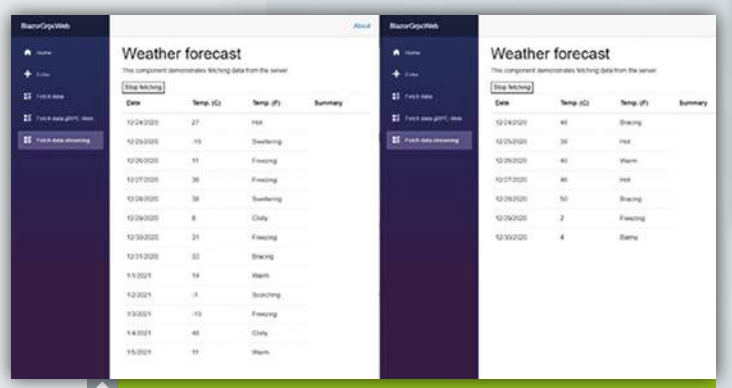
STREAMING IN GRPC-WEB

De complete implementatie van gRPC biedt ondersteuning voor server, client en directionele streaming. Helaas is de ondersteuning binnen gRPC-Web wat beperkter. De enige variant die hier praktisch beschikbaar is, is server side streaming. In het vorige artikel heb ik laten zien hoe client side streaming gebouwd kan worden. Ik zal nu de server side streaming mogelijkheid laten zien binnen een Blazor applicatie. Ik zal hiervoor als voorbeeld een server bouwen die continu weersvoorspellingen naar een client kan sturen. De client toont de ontvangen voorspellingen vervolgens telkens onder elkaar in een lijst. Server side streaming bouwt een unieke verbinding met de server op.

gRPC-Web en Blazor

Meerdere clients kunnen dus tegelijkertijd afzonderlijk gebruik maken van de server streaming. In Figuur 30 wordt getoond dat meerdere clients unieke antwoorden kunnen krijgen van de server.

Figuur 30 Ondersteuning voor meerdere simultane clients



FIGUUR 30: ONDERSTEUNING VOOR MEERDERE SIMULTANE CLIENTS

Server side streaming kan ik aangeven in mijn contract door gebruik te maken van het stream keyword in het returns deel (Figuur 31).

```
service WeatherForecastStreamingService {
    rpc GetWeatherStream(google.protobuf.Empty) returns (stream WeatherForecastStreamDTO);
}
```

FIGUUR 31: SERVER SIDE STREAMING

De codegenerator voor de servercode maakt een methode aan met de volgende signatuur (Figuur 32)

```
service WeatherForecastStreamingService {
    rpc GetWeatherStream(google.protobuf.Empty) returns (stream WeatherForecastStreamDTO);
}
```

FIGUUR 32: SERVER CODE

Ik kan een nieuwe weersvoorspelling vanuit de server sturen door een nieuwe forecast op de stream te zetten (Figuur 33). Belangrijk is hierbij wel om datums altijd in UTC-formaat op te nemen, want dat is een vereiste van gRPC-Web.

```
var forecast = new WeatherForecastStreamDTO()
{
    Date = Timestamp.FromDateTime(DateTime.UtcNow.AddDays(index++)),
    TemperatureC = randomGenerator.Next(-20, 55),
    Summary = Summaries[randomGenerator.Next(Summaries.Length)]
};

await responseStream.WriteAsync(forecast);
```

FIGUUR 33: POSTEN VAN NIEUWE WEERSVOORSPELLING

Ik wil de client de mogelijkheid geven om aan te geven dat de server moet stoppen met het sturen van nieuwe weersvoorspellingen. Hiervoor kan ik de context parameter van de methode gebruiken. Hierin zit namelijk een CancellationToken die ik kan gebruiken om te controleren of de client een verzoek tot stoppen heeft ingediend (Figuur 34).

```
while (!context.CancellationToken.IsCancellationRequested)
```

FIGUUR 34: WHILE LOOP DIE REKENING HOUDT MET HET CANCELLATION TOKEN

De server code kan dus nieuwe data blijven sturen totdat de aangesloten client een stopverzoek stuurt. De complete server side code is voor de volledigheid opgenomen in Figuur 35. Om verwerkingstijd aan de serverkant te simuleren, is per doorgang een kunstmatige vertraging toegepast.

```
public override async Task GetWeatherStream(Empty request, IServerStreamWriter<WeatherForecastStreamDTO> responseStream,
    ServerCallContext context)
{
    var randomGenerator = new Random();
    var index = 0;

    while (!context.CancellationToken.IsCancellationRequested)
    {
        await Task.Delay(500);

        var forecast = new WeatherForecastStreamDTO()
        {
            Date = Timestamp.FromDateTime(DateTime.UtcNow.AddDays(index++)),
            TemperatureC = randomGenerator.Next(-20, 55),
            Summary = Summaries[randomGenerator.Next(Summaries.Length)]
        };

        await responseStream.WriteAsync(forecast);
    }
```

FIGUUR 35: COMPLETE SERVER SIDE CODE

Vanuit de client kun je nu de berichten opvangen van de server door gebruik te maken van de AsyncServerStreamCall referentie die je kunt verkrijgen door de streaming methode aan te roepen (Figuur 36).

```
using var streamingCall = WeatherForecastClient.GetWeatherStream(new Empty(), cancellationToken: _cancellationTokenSource.Token);
```

FIGUUR 36: VERKRIJGEN REFERENTIE NAAR DE SERVER STREAMING CALL

Voor het mogelijk maken van de annulering is een CancellationTokenSource nodig. Deze kan gewoon aan worden gemaakt als field in de code van de component (Figuur 37). Ik heb in deze code gebruik gemaakt van de C# 9 ondersteuning van target-typed new. Het type hoef je dan niet meer expliciet achter de new-operator op te nemen. Het type wordt dan bepaald door het type van de variabele. In de token source zit de Token die je als parameter door moet geven.

```
private readonly CancellationTokenSource _cancellationTokenSource = new();
```

FIGUUR 37: AANMAKEN CANCELLATIONTOKENSOURCE

Het uitlezen van de verschillende waarden die door de server gestuurd worden, kun je het beste doen met de await foreach constructie (Figuur 38). De changetracking binnen Blazor herkent het ontvangen van nieuwe waarden via deze constructie niet. Je moet dus expliciet bij elke nieuwe waarde de methode StateHasChanged aanroepen. Als het CancellationToken geactiveerd wordt, dan wordt vanuit de server een RpcException gegooid. Het is dan ook belangrijk om deze exception op te vangen, maar dan alleen voor die excepties met de statuscode “StatusCode.Cancelled”. Gelukkig ondersteunt C# ook filter mogelijkheden bij catches door gebruik te maken van het when keyword.

```
try
{
    await foreach (var weatherData in streamingCall.ResponseStream.ReadAllAsync(cancellationToken: _cancellationTokenSource.Token))
    {
        _forecasts.Add(weatherData);

        StateHasChanged();
    }
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.Cancelled)
{
}
```

FIGUUR 38: CODE VOOR HET UITLEZEN VAN DE VERSCHILLENDE WAARDEN

Het daadwerkelijk annuleren van het verzoek wordt door een methode gedaan die gekoppeld is aan een stop-knop. De enige code die nodig is om het stoppen in gang te zetten, is het aanroepen van de Cancel-

methode op de CancellationTokenSource (Figuur 39). Binnen de CancellationToken van deze TokenSource komt hierdoor de property IsCancellationRequested op true te staan en stopt de server dus met zenden.

```
private void OnStopFetch()
{
    _cancellationTokenSource.Cancel();
}
```

FIGUUR 39: AANROEPEN VAN CANCEL OP DE CANCELLATIONTOKENSOURCE

De complete client side code is voor de volledigheid opgenomen in Figuur 40.

```
[Inject]
private WeatherForecastStreamingService.WeatherForecastStreamingServiceClient WeatherForecastClient { get; set; }

private readonly List<WeatherForecastStreamDTO> _forecasts = new();

private readonly CancellationTokenSource _cancellationTokenSource = new();

private async Task OnStartFetch()
{
    using var streamingCall = WeatherForecastClient.GetWeatherStream(new Empty(), cancellationToken: _cancellationTokenSource.Token);

    try
    {
        await foreach (var weatherData in streamingCall.ResponseStream.ReadAllAsync(cancellationToken: _cancellationTokenSource.Token))
        {
            _forecasts.Add(weatherData);

            StateHasChanged();
        }
    }
    catch (RpcException ex) when (ex.StatusCode == StatusCode.Cancelled)
    {
    }
}

private void OnStopFetch()
{
    _cancellationTokenSource.Cancel();
}
```

FIGUUR 40: CLIENT SIDE CODE

Server side streaming binnen gRPC-Web is dus met relatief weinig code makkelijk te implementeren.

> GRPC IS GEBASEERD OP HET HTTP/2 PROTOCOL VOOR VEEL CONTROLE OP LAAG NIVEAU TUSSEN CLIENT EN SERVER

WAAROM KIEZEN VOOR GRPC-WEB?

Heel veel webservices worden op dit moment geschreven in de vorm van een WebAPI. Waarom zou ik in de praktijk dan toch kiezen om gebruik te gaan maken van gRPC-Web?
In eerste instantie is het hebben van een contract natuurlijk heel erg prettig. Je weet exact wat je server aan methoden ondersteund en welke data door die methoden verwacht wordt en welke data weer retour komt. Bijna alle mogelijkheden die gRPC ondersteunt, zijn ook beschikbaar in gRPC-Web en dus in je SPA.
Een ander voordeel is dat gRPC-Web echt veel kleinere berichten kan versturen dan WebAPI. In de voorbeeldcode heb ik twee identieke services gemaakt die weerdata opleveren: een als WebAPI en een als gRPC-Web service. Beide implementaties geven 500 voorspellingen non-streaming door aan de Blazor-client. In de netwerk monitor van Chrome (Figuur 41) kun je duidelijk het verschil tussen WebAPI (WeatherForecast) en gRPC-Web (GetForecasts) zien. De berichtgrootte bij gRPC-Web is 44% kleiner en het bericht wordt 33% sneller verstuurd dan bij WebAPI. Voor een SPA is snelheid natuurlijk heel erg belangrijk om een goede gebruikersvriendelijkheid te kunnen realiseren. Daarom zou ik in de praktijk bij een Blazor SPA voor gRPC-Web kiezen in plaats van voor WebAPI. Ook Xamarin heeft op dit moment nog geen goede ondersteuning voor gRPC, maar gRPC-Web is wel bruikbaar binnen dit framework. Ook binnen een mobiel platform zijn kleinere berichten natuurlijk zeer welkom.

Name	Status	Type	Initiator	Size	Time
WeatherForecast	200	fetch	dotnet.5.0.1.js:1	41.0 kB	452 ms
GetForecasts	200	fetch	dotnet.5.0.1.js:1	18.2 kB	152 ms

FIGUUR 41: VERKEERANALYSE

In het vorige artikel heb ik besproken dat IIS en Azure App Services op dit moment nog niet goed overweg kunnen met de volledige gRPC-implementatie. De ondersteuning hiervan is in preview versies van Windows Server al wel beschikbaar, maar is nog niet in productie uitgerold. Als je gebruik maakt van gRPC-Web, kun je nu al wel gebruik maken van (bijna) alle voordelen van gRPC, zoals contracten en kleinere berichtgroottes.

SAMENVATTING

In dit artikel heb ik de belangrijkste aspecten voor het bouwen van een gRPC-Web service en een bijbehorende Blazorclient behandeld. Met gRPC-Web kan ik op een simpele manier een contract-first api maken die gebruikt kan worden binnen de context van een browser en die bovendien ook nog sneller is dan WebAPI en minder netwerkverkeer nodig heeft. Voor ontwikkelaars die al bekend zijn met gRPC binnen ASP.NET Core, is het toevoegen van gRPC-Web ondersteuning aan een gRPC service een hele kleine aanpassing in de code. Door de goede ondersteuning die standaard opgenomen is in de “Grpc.Net.Client.Web” NuGet package is het bouwen van een client ook niet heel lastig.

Voor backend services voor Blazor en Xamarin applicaties en voor gRPC services die op dit moment gehost moeten kunnen worden op Azure App Services of IIS, is gRPC-Web een goede oplossing die ik zelf zeker al in durf te zetten in productiesystemen.
Op https://github.com/JohanSmarius/BlazorGrpcWeb staat de code die gebruikt is in dit artikel.
Een aantal screenshots in dit artikel zijn afkomstig uit deze code.

JOHAN SMARIUS

Johan Smarius is een ervaren (lead) software developer, architect, spreker en trainer op het Microsoft .NET vlak met meer dan 25 jaar ervaring in de IT. Hij heeft ervaring met .NET sinds versie 1.0. Johan werkt als docent informatica bij Avans Hogeschool in Breda en als technical consultant bij zijn eigen bedrijf JMAC Software Solutions. In zijn vrije tijd leert Johan kinderen programmeren en is hij instructeur EHBO.
Twitter: @JohanSmarius
Website: https://www.johansmarius.com

Christiaan Nieuwlaat

UITROLLEN AWS INFRASTRUCTUUR MET

.NET 3.1 en AWS Cloud Development Kit (CDK)

Het is algemeen bekend, de IT-wereld veranderd snel en vaak. Steeds meer organisaties gaan over naar een Cloud gebaseerde infrastructuur, steeds vaker wordt DevOps als cultuur in een organisatie ingevoerd. Een van de veranderingen die hierbij zichtbaar aanwezig is, is het “codificeren” van het (Cloud) platform (Infrastructure as Code), of in Jip-en-Janneke taal, de blauwdruk van de (virtuele) infrastructuur uitgedrukt in software regels.

Dat is niets nieuws, we gebruiken al een paar jaar tools zoals Terraform, eventueel gecombineerd met Chef of Ansible, om onze virtuele infrastructuur uit te rollen en wijzigingen door te voeren. Maar, dit blijven externe tools, en hebben vaak een eigen specificatietaal. Om deze tools te kunnen gebruiken, moet je deze specificatietaal te leren kennen. Ondanks dat dit relatief eenvoudig is, zou het niet handiger zijn als je om met een programmeertaal (die je al kent) je infrastructuur te definiëren en uit te rollen?

AWS CLOUD DEVELOPMENT KIT (CDK) TO THE RESCUE

Precies dat concept heeft Amazon ter harte genomen en daarvoor onlangs de Cloud Development Kit (CDK) publiek toegankelijk gemaakt. Deze CDK bestaat uit een aantal bibliotheken voor verschillende programmeertalen, waarmee het erg eenvoudig wordt om met code je Cloud platform uit te rollen. De CDK is van origine een op TypeScript gebaseerde bibliotheek, maar door Cross Language Transpilatie is deze naast in TypeScript of Javascript ook te gebruiken in talen als C# (.Net Core 3.1 of .Net 5), Java en Python.

CLOUDFORMATION?

AWS heeft al jaren een toolkit voor IaC (infrastructuur als code) genaamd CloudFormation. Dit is een op templates gebaseerde oplossing om de gewenste infrastructuur te omschrijven, zodat deze met de CloudFormation applicatie geautomatiseerd kunt uitrollen. Hierbij speelt echter hetzelfde nadeel als bij de eerdergenoemde oplossingen als Terraform, je moet eerst de specificatie taal leren om dit efficiënt te kunnen gebruiken. Deze specificaties zijn vrij uitgebreid, waardoor je voor een simpele Cloud structuur al snel honderden regels code aan het schrijven bent.
De CDK is daaruit voortgekomen, en lift in principe mee met de CloudFormation structuur. Op de achtergrond genereert de CDK namelijk de uitgebreide CloudFormation templates, die voor het daadwerkelijk uitrollen van het platform worden gebruikt. Maar door de extra slimme logica in de CDK, is iets wat tientallen regels templatecode kost nu vaak in 1 regel code te vangen.

IN DE PRAKTIJK

Hoe de CDK werkt? Dat kan ik je het best laten zien in het volgende voorbeeld. We gaan een simpele omgeving opzetten op AWS met behulp van C# (.Net Core 3.1) en de AWS CDK.
Wat gaan we uitrollen?

We gaan een cloud platform uitrollen met de volgende componenten:

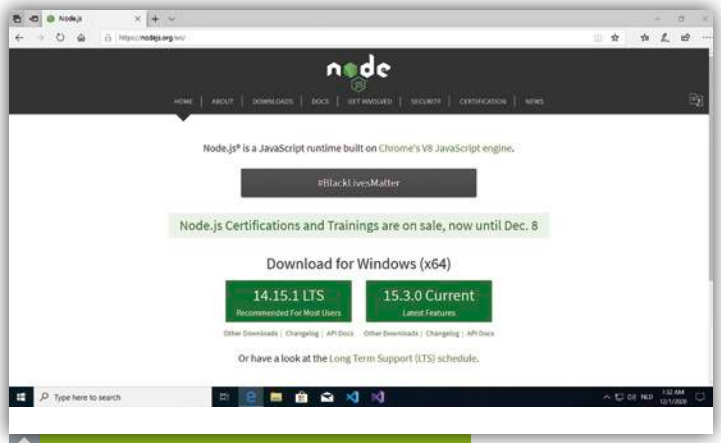
- Virtueel Private Cloud (VPC)
  - 1 Availability zone
  - IP Reeks (10.0.0.0/16)
  - 2 Subnets (1 public, 1 private)
- Routing tabellen
  - NAT gateway voor het private subnet
  - Bastion host in public subnet (t2.micro)
  - Private host in private subnet (t2.micro)

VOORBEREIDING

Wat hebben we nodig om de CDK te gebruiken?

- Visual Studio Code
- Node.js (voor de CDK CLI)
- AWS-account

Stap 1: Installeren Node.js en Visual studio code
Download van de node.js website (https://nodejs.org) de laatste LTS-versie van node (op het moment van schrijven is dat versie 14.15)



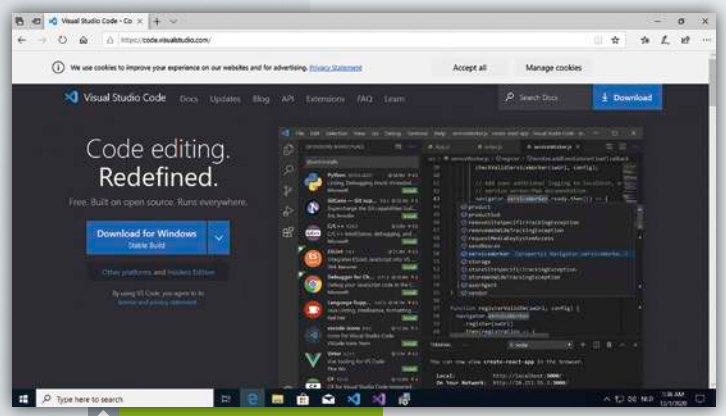
FIGUUR 1

Installeer deze versie van node.js.

Download de laatste versie van Visual Studio Code (https://code.visualstudio.com/)

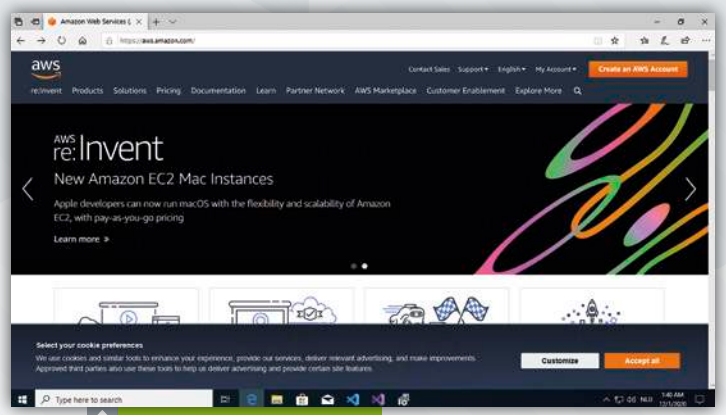
Installeer deze versie van VS Code.

[1] *Een t2.micro instantie is de goedkoopste EC2 instantie en uitermate geschikt voor de demo. Deze valt ook onder de 12-maanden free tier van nieuwe AWS-accounts, en is daarom ideaal om te gebruiken.*



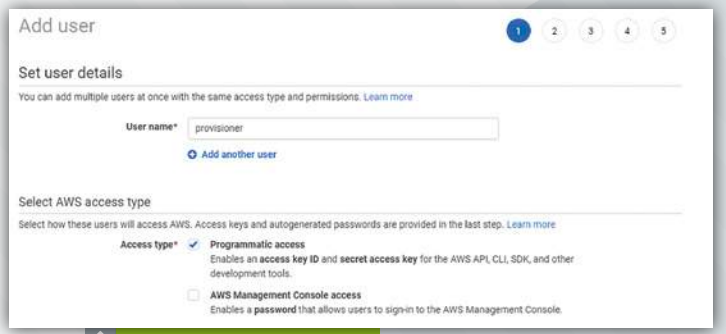
FIGUUR 2

Stap 2: Maak een account aan bij AWS
Indien je nog geen AWS-account hebt, kun je deze aanmaken door naar https://aws.amazon.com/ te browsen en daar op “Create an account” te klikken en het proces te doorlopen. Een account op AWS is gratis, zolang je geen services hebt afgenomen. De meeste services in deze demo vallen onder de zogenaamde free tier. Dit houdt in dat de eerste 12 maanden van een nieuw account deze services (al dan niet met gelimiteerde capaciteit) gratis te gebruiken zijn.



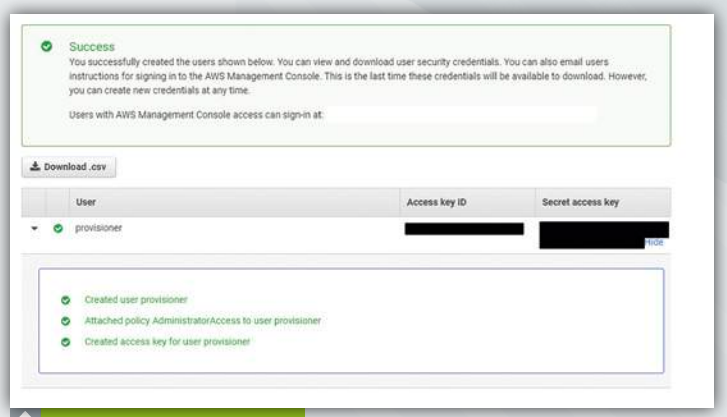
FIGUUR 3

Nadat je een account hebt aangemaakt is het verstandig om een admin user aan te maken specifiek voor het uitrollen van de omgeving. Het is een best practice om zo min mogelijk met je root account te doen. Belangrijk hierbij is dat je een gebruiker aanmaakt met programmatic access, want deze access keys hebben we nodig om geautomatiseerd te kunnen uitrollen.



FIGUUR 4

Sla vooral deze CSV lokaal op, want deze gegevens heb je nodig!

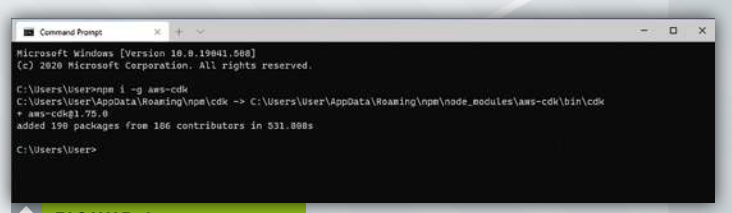


FIGUUR 5

Stap 3: Installeer de AWS CDK CLI

De AWS CDK cli is te verkrijgen uit de npm-repository, deze kun je installeren door het volgende commando uit te voeren

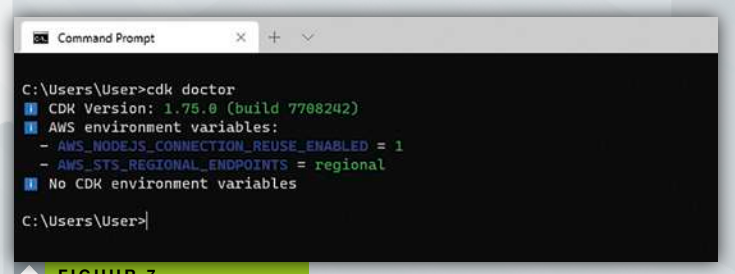
```
npm install -g aws-cdk
```



FIGUUR 6

Nu is de command line interface van de AWS CDK geïnstalleerd en beschikbaar. Dit kunnen we controleren door het volgende commando uit te voeren

```
cdk doctor
```



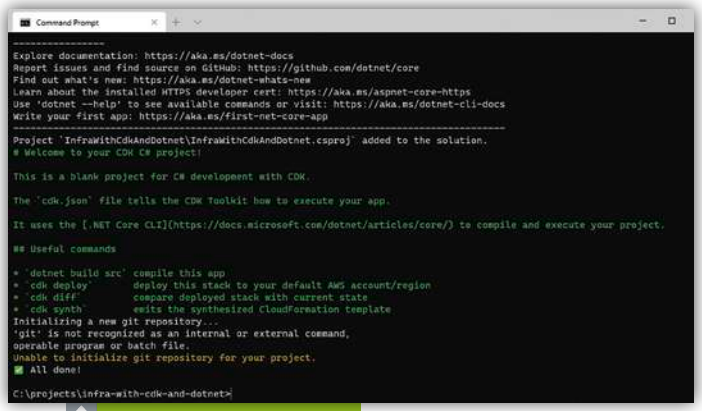
FIGUUR 7

Indien de cdk tooling goed geïnstalleerd is zal een scherm als hierboven getoond worden.
Nu staat alles klaar om het echte deployen te beginnen 😊

Lets get ready to rumble
Stap 1: Genereer een app skelet vanuit de aws-cdk cli
Maak een directory aan voor je infra project, bijvoorbeeld c:\projects\infra-with-cdk-and-dotnet, ga in deze directory staan en voer dan het volgende commando uit om een “app” skelet te laten genereren.

```
cdk init --language=csharp
```

Na dit commando krijg je onderstaand scherm te zien:



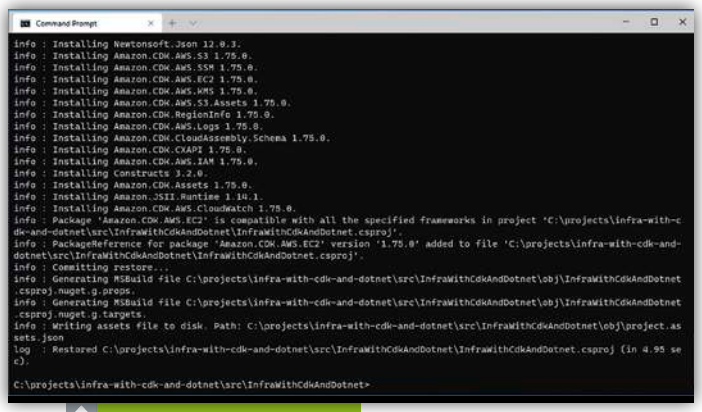
FIGUUR 8

We hebben nu een lege CDK-applicatie laten genereren. Nu is het zaak om deze applicatie te vullen met de logica die we nodig hebben om onze infra te laten uitrollen.

Stap 2: Installeer CDK-NuGet packages
De CDK is modulair opgebouwd en wel op een zodanige manier dat de verschillende AWS-infrastructuur componenten hun eigen module hebben. Dit betekent dat je voor te gebruiken componenten ook de benodigde modules moet toevoegen aan je project.
We moeten voor onze doel infrastructuur de package Amazon.CDK.AWS.EC2 toevoegen. Hierin zitten de vereiste componenten ("constructs") om onze doelinfrastructuur samen te stellen.
Dit doe je met de volgende commando's.

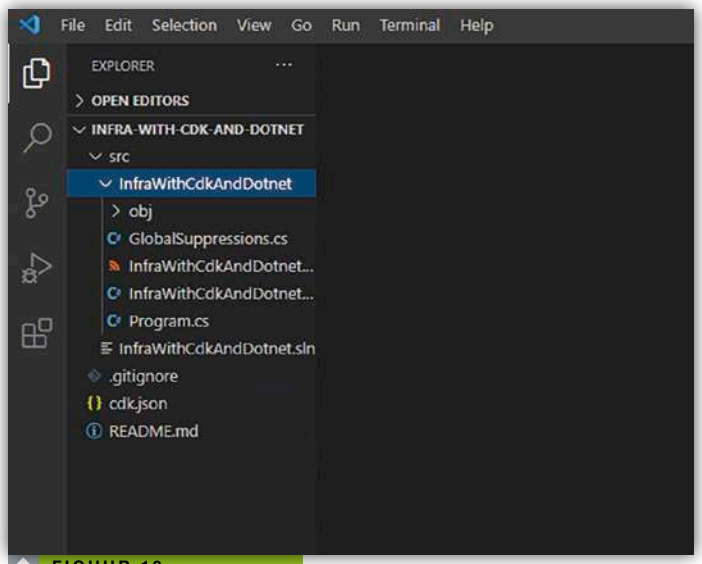
```
cd src/InfraWithCdkAndDotnet
dotnet add package Amazon.CDK.AWS.EC2
```

Hierdoor wordt de Amazon.CDK.AWS.EC2 package en al zijn afhankelijkheden geïnstalleerd in het project.



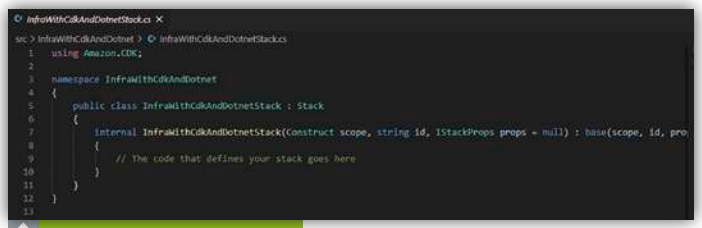
FIGUUR 9

Stap 3: Maak de "Stack"
Een stack is een serie componenten welke (eventueel herbruikbaar) gebruikt worden een samenhangend deel van je infrastructuur te beschrijven. Een stack is dus een container voor AWS-services, bijvoorbeeld een VPC, of een host of meerdere hosts.
In dit voorbeeld zullen we maar 1 stack maken die onze totale infra beschrijft, maar je kunt meerdere stacks maken als je platformstructuur in delen knipt. Bijvoorbeeld met een appStack, die je dan meerdere keren kan uitrollen, bijvoorbeeld 1x voor Productie en 1x voor een Staging omgeving.
Open Visual Studio Code, en open de solution folder.



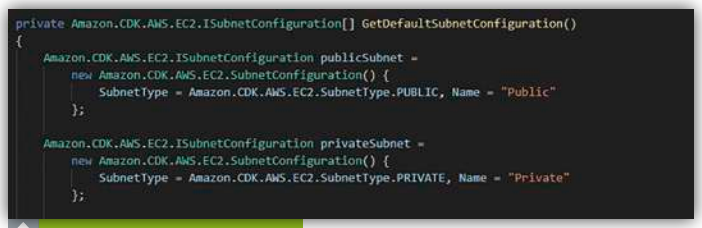
FIGUUR 10

Zoals je ziet is er een Program.cs en een InfraWithCdkAndDotnetStack.cs gegenereerd door de CDK cli. De Program.cs is het entrypoint van de gecompileerde executable, en bevat alleen een "App" class welke als container fungeert voor je infrastructuur.
De InfraWithCdkAndDotnetStack.cs is het bestand waar we onze infrastructuur gaan coderen.



FIGUUR 11

We moeten van buiten naar binnen denken, als we het hebben over infrastructuur. Dus in ons geval is de VPC de buitenste schil, met daarin de subnets, en daarin de hosts.
Dit zal in de code ook op deze manier worden opgezet, we beginnen met de VPC. Deze VPC heeft 2 subnets, die we moeten specificeren bij het creëren van een nieuwe instantie van de VPC Class.
We beginnen dus met een helper functie die de subnet specificatie teruggeeft:



FIGUUR 12

Nu kunnen we de VPC-instantie creëren door de constructor van onze stack als volgt te veranderen:

```
internal InfraWithCdkAndDotnetStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props)
{
    // The code that defines your stack goes here
    Amazon.CDK.AWS.EC2.Vpc myVpc =
        new Amazon.CDK.AWS.EC2.Vpc(this, $"VPC-DEMO",
        new Amazon.CDK.AWS.EC2.VpcProps() {
            Cidr = "10.0.0.0/16"
            MaxAzs = 1,
            NatGateways = 1,
            SubnetConfiguration = this.GetDefaultSubnetConfiguration(),
        });
    }
}
```

FIGUUR 13

Als we nu de applicatie builden dan kunnen we via de CDK CLI valideren dat er een VPC zal worden gemaakt met 1 Nat Gateway, 2 subnetten verdeeld over de CIDR 10.0.0.0/16. Dit kan je doen door op de commandline het volgende commando te gebruiken

```
cdk synth
```

Als er geen fouten in de code zitten is zie je een scherm zoals hieronder is weergegeven.

```
Command Prompt
C:\projects\infra-with-cdk-and-dotnet>cdk synth
Resources:
  VPCDEMO920D5CDF:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsHostnames: true
      EnableDnsSupport: true
      InstanceTenancy: default
      Tags:
        - Key: Name
          Value: InfraWithCdkAndDotnetStack/VPC-DEMO
    Metadata:
      aws:cdk:path: InfraWithCdkAndDotnetStack/VPC-DEMO/Resource
  VPCDEMOPublicSubnet1Subnet316CB5AB:
    Type: AWS::EC2::Subnet
    Properties:
      CidrBlock: 10.0.0.0/17
      VpcId:
        Ref: VPCDEMO920D5CDF
      AvailabilityZone:
        Fn::Select:
          - 0
          - Fn::GetAZs: ""
      MapPublicIpOnLaunch: true
      Tags:
        - Key: aws-cdk:subnet-name
          Value: Public
        - Key: aws-cdk:subnet-type
```

FIGUUR 14

Dit is de dump van de onderliggende CloudFormation template(s) die de CDK applicatie voor je genereert. In dit template zie je dat er een VPC resource wordt gemaakt (Type: AWS::EC2::VPC) met de gewenste CIDR.
De template en metadata welke door de CDK gebruikt worden zijn weggeschreven in de map cdk.out. Hierin staat een bestand genaamd InfraWithCdkAndDotnetStack.template.json, dit is de json versie van het CloudFormation template van onze stack.

Op dit moment, wordt er door het aanroepen van deze ene constructor een template gegenereerd van meer dan 400 regels. In dit template worden de volgende componenten gedefinieerd:

- VPC (CIDR 10.0.0.0/16)
  - Public subnet (CIDR 10.0.0.0/17)
    - Routetable public subnet
    - Subnet-Routetable association
    - Default Route
    - Nat Gateway
      - Elastic IP voor nat gateway
  - Private subnet (CIDR 10.0.128.0/17)
    - Routetable private subnet
    - Subnet-Routetable association
    - Default Route
  - Internet gateway
    - VPC Gateway Attachment (VPC->Internet Gateway)

Niet slecht he, voor nog geen 20 regels code 😊.
Maar…. We zijn er nog niet… we willen nog een bastion host[2] in het public subnet, en een private host in het private subnet. Gelukkig heeft de CDK een voorbedacht construct voor een Linux gebaseerde bastion host, genaamd BastionHostLinux. Hierdoor is het toevoegen van deze bastion host een fluitje van een cent.

Het enige wat we hoeven te doen is onze stack constructor uit te breiden met het volgende stukje code.

```
// Bastion host
Amazon.CDK.AWS.EC2.BastionHostLinux bastionHost =
    new Amazon.CDK.AWS.EC2.BastionHostLinux(this, "Bastionhost",
    new Amazon.CDK.AWS.EC2.BastionHostLinuxProps() {
    InstanceType = Amazon.CDK.AWS.EC2.InstanceType.Of(
        Amazon.CDK.AWS.EC2.InstanceClass.BURSTABLE2,
        Amazon.CDK.AWS.EC2.InstanceSize.MICRO),
    Vpc = myVpc
});
```

FIGUUR 15

Met dit kleine stukje code instrueren wij het platform om een bastion host draaiende op Amazon Linux en met de laatste SSM-agent te plaatsen in ons VPC. We hebben aangegeven dat wij voor deze bastion host ook een t2.micro EC2 instantie willen gebruiken. (Standaard zou er een t3.nano voor worden uitgerold, maar deze zit niet in de free tier, vandaar de keuze voor een t2.micro.)
En als laatste hebben we nog een host in het private subnet voorzien. Ook dit is een kwestie van een klein stukje code toevoegen aan onze constructor.

```
// Private server
Amazon.CDK.AWS.EC2.Instance_ privateServerInstance =
    new Amazon.CDK.AWS.EC2.Instance_(this, "Privateserver",
    new Amazon.CDK.AWS.EC2.InstanceProps() {
    AllowAllOutbound = true,
    InstanceType = Amazon.CDK.AWS.EC2.InstanceType.Of(
        Amazon.CDK.AWS.EC2.InstanceClass.BURSTABLE2,
        Amazon.CDK.AWS.EC2.InstanceSize.MICRO),
    MachineImage = Amazon.CDK.AWS.EC2.MachineImage.LatestAmazonLinux(),
    Vpc = myVpc
});
```

FIGUUR 16

En hiermee is onze doelinfrastructuur uitgeschreven.
Als je nu het template opvraagt met behulp van het cdk synth commando, dan zul je ook de bastion host en de private server in

[2] *Een zogenaamde bastion host is een machine in het publieke subnet, welke als "toegangspoort" fungeert naar het netwerk in de VPC. Met andere woorden, je kan alleen bij de machines in het private subnet via de bastion server.*

de templates zien staan. Ook de support templates zoals security groups enzovoorts worden meegenomen, hiervoor hebben we nog niets handmatig hoeven in te richten.

EN HOE NU NAAR LIVE?

Het enige wat je nu hoeft te doen om je stack live te zetten is de gegevens van je eerder gemaakte AWS-account in de environment variabelen AWS_SECRET_KEY en AWS_ACCESS_KEY_ID te zetten en dan het volgende commando uit te voeren

```
cdk deploy
```

De CDK gaat dan alle templates effectueren op het AWS-account behorende bij de gegevens die je in de environment variabelen hebt ingesteld.
Als alles goed gaat is een paar minuten later je omgeving in de AWS EC2 console beschikbaar.

Let op! De eerste keer dat je een omgeving uitrolt met behulp van de CDK dien je nog een extra commando te draaien voordat de deploy gedaan kan worden.

```
cdk bootstrap
```

Dit commando bereid het AWS account voor om deployments te kunnen uitvoeren.

OPRUIMEN

Op dit moment heb je resources actief, en gezien het pay-per-use model maak je nu kosten voor deze omgeving. Deze omgeving weer afbreken (en dus ook geen kosten meer maken), is net zo eenvoudig als het opzetten.
Je breekt de omgeving af met behulp van het commando

```
cdk destroy
```

Een paar minuten later is alles weer netjes opgeruimd.

SAMENGEVAT

In dit artikel heb ik jullie laten zien hoe eenvoudig het is om, met behulp van de AWS CDK (Cloud Development Kit) en een klein beetje C# code, een infrastructuur op AWS uit te rollen. We hebben het hier heel erg basaal gehouden, maar je kan je infrastructuur op deze manier ook als een applicatie beschouwen, en “Stacks” bouwen die door middel van parameters hergebruikt kunnen worden, analoog aan de object georiënteerde oplossingen die we in programmacode gebruiken. Hierdoor kunnen wij de kracht van de CDK volledig integreren in onze applicaties, en onze applicaties en de onderliggende infrastructuur verder verweven tot één geheel.

HOE HEBBEN WE DAT GEDAAN?

1. Installatie laatste versies van Node JS en Visual Studio Code, installatie AWS CDK CLI.
2. Genereer app skelet
3. Vul de logica van de app aan met de verschillende construct classes
4. Build de applicatie
5. Deploy via CDK CLI

CHRISTIAAN NIEUWLAAT

Al vanaf zijn 4e levensjaar is Christiaan Nieuwlaat bezig met programmeren, begonnen met MSX basic listings overtypen vanuit MSX Tips-en-trucs boekjes. Daarna via Assembler, Pascal, Delphi, C en C++, nu, 35 jaar ervaring verder, zowel bedreven op het gebied van Java als .NET. Op dit moment houdt hij zich voornamelijk bezig met AWS, Cloud, Software ontwikkeling, Architectuur en Security. Naast gepassioneerd ontwikkelaar / architect is Christiaan sinds 7 juni 2019 een Certified Ethical Hacker, wat zijn passie voor informatiebeveiliging en softwarebeveiliging alleen maar meer heeft aangewakkerd.

Christiaan werkt op dit moment bij Team Rockstars IT als software engineer / solution architect, en helpt klanten hun complexe IT vraagstukken op te lossen, dit in diverse rollen, variërend van technisch adviseur tot software architect.

Naast zijn werkzaamheden is Christiaan een liefhebbende vader, deejay, danser en houdt hij ervan om te “tinkeren” met micro-elektronica / IoT devices. Hij is tevens bestuurslid van de lokale EHBO vereniging en in opleiding tot instructeur EHBO.

WIJ KUNNEN DE KRACHT VAN DE CDK VOLLEDIG INTEGREREN IN ONZE APPLICATIES, EN ONZE APPLICATIES EN DE ONDERLIGGENDE INFRASTRUCTUUR VERDER VERWEVEN TOT ÉÉN GEHEEL.

Sander Hoogendoorn

A Glimmer of Hope

In September 2003 I visited the friendly city of Arhus in Denmark to deliver a talk at the JAOO conference. At that time, JAOO was an established software development conference, with well-known speakers such as Martin Fowler, Kent Beck, Linda Rising, and Erich Gamma. And me.

When I arrived at the venue, one of the organizers offered to give me a guided tour of the building. While we strolled through the conference center, she enthusiastically pointed out the major rooms where the conference would take place. Then all of a sudden, the organizer stopped at a wooden door in a small hallway. She opened the door respectfully and we looked in. An ill-lit room with furniture stacked to the walls. “Do you know what happened in this room?” she asked me almost whisperingly. I shook my head. Of course, I had no idea. She continued mysteriously. “This is where during JAOO last year, Kent Beck and Erich Gamma conceived JUnit.”

I guess that was my first introduction to unit testing.

The impact of JUnit on the industry of software development was huge. It was the first viable unit testing framework out there. Still, unit testing took a long time to gain traction. Even today in 2020, when I ask audiences at talks for a show of hands about who writes unit tests, most people do not raise their hands. And that’s a shame.

Recently, I was coaching a development team that had come to a complete standstill. They didn’t dare to change another line of code, afraid the codebase would break. Not a single feature had been delivered to their clients in over ten months. When we investigated their codebase, we found a few unit tests, extremely low code coverage, high cyclomatic complexity, and many untested paths in the code. And still, when I stressed the importance of solid unit testing for maintaining quality and speed of development, the developers stared at me with long faces and asked if writing all these tests was really, really necessary.

It takes time to learn.

It is.

To be honest, it took me quite some time to get convinced of the effects and benefits of unit testing and test-driven development. For authors Kent Beck and Erich Gamma, the number one goal for creating JUnit was “to write a framework within which we have some glimmer of hope that developers will actually write tests.” They built JUnit with already existing, familiar tools so that there was little new to learn. As they said: “It has to require no more work than absolutely necessary to write a new test.” Gamma and Beck made writing unit tests as simple as possible.

> WE NEVER SEEM TO HAVE TIME AVAILABLE TO WORK ON THE QUALITY OF OUR ARCHITECTURE AND CODE.

So why is it so hard to start writing unit tests?

From my own experience, despite the simplicity of the tooling, with Jest being my current poison to test my TypeScript and JavaScript code, one problem I think lies in the fact that writing unit tests still require skills and understanding. It takes time to learn. And time is something we are always short of in software development. There is continuous pressure from stakeholders, managers, or product owners to add and improve features in our products. We never seem to have time available to work on the quality of our architecture and code. Hence, we tend to skip writing unit tests.

Another challenge is that the benefits of having good unit tests are reaped in the long run. People are not good at preparing for long term benefits. Not unit testing is like smoking. We all know smoking is bad for us. Nonetheless, people find it extremely hard to stop smoking. This is because the drawbacks of smoking are usually not felt until years later. With unit testing, this is quite similar.

A Glimmer of Hope

Having unit tests for everything pays back in the long run. It pays back after six months when you notice you can change your code without pain. And it pays back after one year when you can still change the code. And even after multiple years, your code remains stable and has even become more robust than it ever was. Unfortunately, too many people postpone writing unit tests until it's late, and the consequences of not having tested code have become huge.

> UNIT TESTING IS ESSENTIAL

And it becomes worse when you work with modern software architectures, such as microservices or serverless, where you usually do not have a single codebase, but rather work on a vast collection of small codebases. With one of my recent clients, the total landscape consisted of around forty micro-applications, fifty microservices, and about the same number of serverless functions, each of which with its codebases, pipelines, and infrastructure. In environments like these, it becomes even more vital to keep your codebases in good health. Unit testing is essential.

As an example, while writing this post, my teams and I are looking at a small, but a rather delicate change in the core library of my client's landscape. This library contains around fifteen hundred lines of code. Around thirty applications and microservices running in production depend on it. Without even giving it a second thought, we made the change; we ran the three-hundred-eighty-five unit tests on the library; saw that these all turned green; we checked in the code. No pipelines broke in the process.

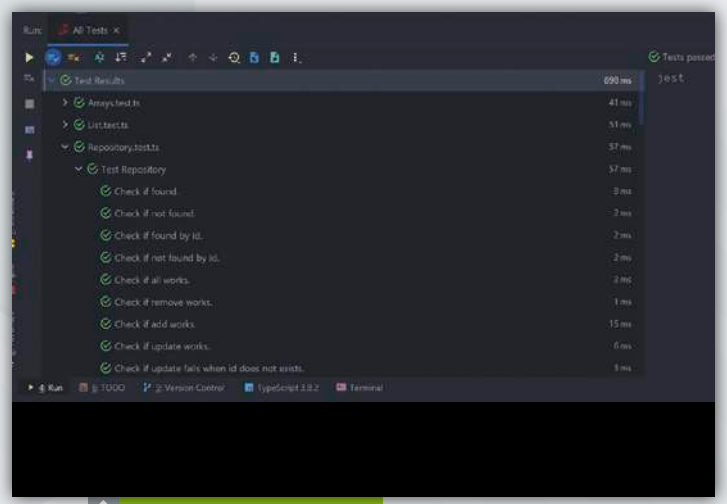


FIGURE 1

With proper unit testing, we would not have had the trust and guts to make this radical change. We would just let it slide. And we would just let it slide over and over again. Until we would not be able to guarantee the quality of our codebases anymore.

My advice? Stop whining about the effort and time it takes to start writing unit tests. Or even complain about it being boring. Even though unit testing might not seem to deliver on its promises immediately, it will in the long run. For sure. Don't wait until your code becomes unmaintainable. Just like you can better stop smoking today than to wait for your package to be empty, or until the first of January of next year. There is no excuse to postpone. It is always better to start writing unit tests today.

Actually? Stop reading now, start unit testing.

SANDER HOOGENDOORN

Sander is an independent dad, mentor, trainer, software architect, programmer, speaker, and writer. He is a highly appreciated catalyst in the innovation of software development at his many international clients.
Well known as the author of the best-selling book This Is Agile, Sander coaches organizations, projects and teams, has written books on UML and agile, and published over 250 articles in international magazines. He is an inspiring (keynote) speaker at many international conferences, and presents seminars and training courses on a variety of topics such as (beyond) agile, Scrum, Kanban, software estimation, software architecture, microservices, design patterns, modeling and UML, writing code, and testing.

Mail sander@ditisagile.nl
Twitter @aahoogendoorn
Web www.sanderhoogendoorn.com | www.ditisagile.nl

> MY ADVICE? STOP WHINING ABOUT THE EFFORT AND TIME IT TAKES TO START WRITING UNIT TESTS.

HELP JIJ ACHMEA #1STAPVOOR TE ZIJN?

Bij Achmea geloven we in een service waarbij wij 24 uur per dag, 7 dagen per week, klaar staan voor onze klanten. Daar hebben we de nieuwste technologie voor nodig. En IT'ers die **vooruitkijken**, **lef tonen** en **#1stapvoor lopen**.

Slimme mensen die kunnen bouwen – soms letterlijk – op onze huidige systemen en platformen, en mensen die ook verder kijken naar andere mogelijkheden. Collega's die samen in hoog tempo de technische ontwikkelingen aankunnen. Iets voor jou?

Zet jouw stap bij Achmea:

→ 1. Native Android Developer

→ 2. .Net Software Engineer Homies

→ 3. Sitecore Developer Centraal Beheer

Meewerken aan een gezonde en veilige toekomst?

werkenbijachmea.nl

Centraal Beheer · Zilveren Kruis · Interpolis · FBTO · achmea

SOFTWARE DEVELOPERS VAN TEAM ROCKSTARS IT ZIJN HET MEEST GELUKKIG IN HUN VAK EN MAKEN IMPACT BIJ BEDRIJVEN DOOR HEEL NEDERLAND.

KEUZEVRIJHEID & INSPRAAK

FAIRNESS & TRANSPARANTIE

COMMUNITY GEVOEL

TEAM ROCKSTARS IT

WWW.**TEAMROCKSTARS**.NL