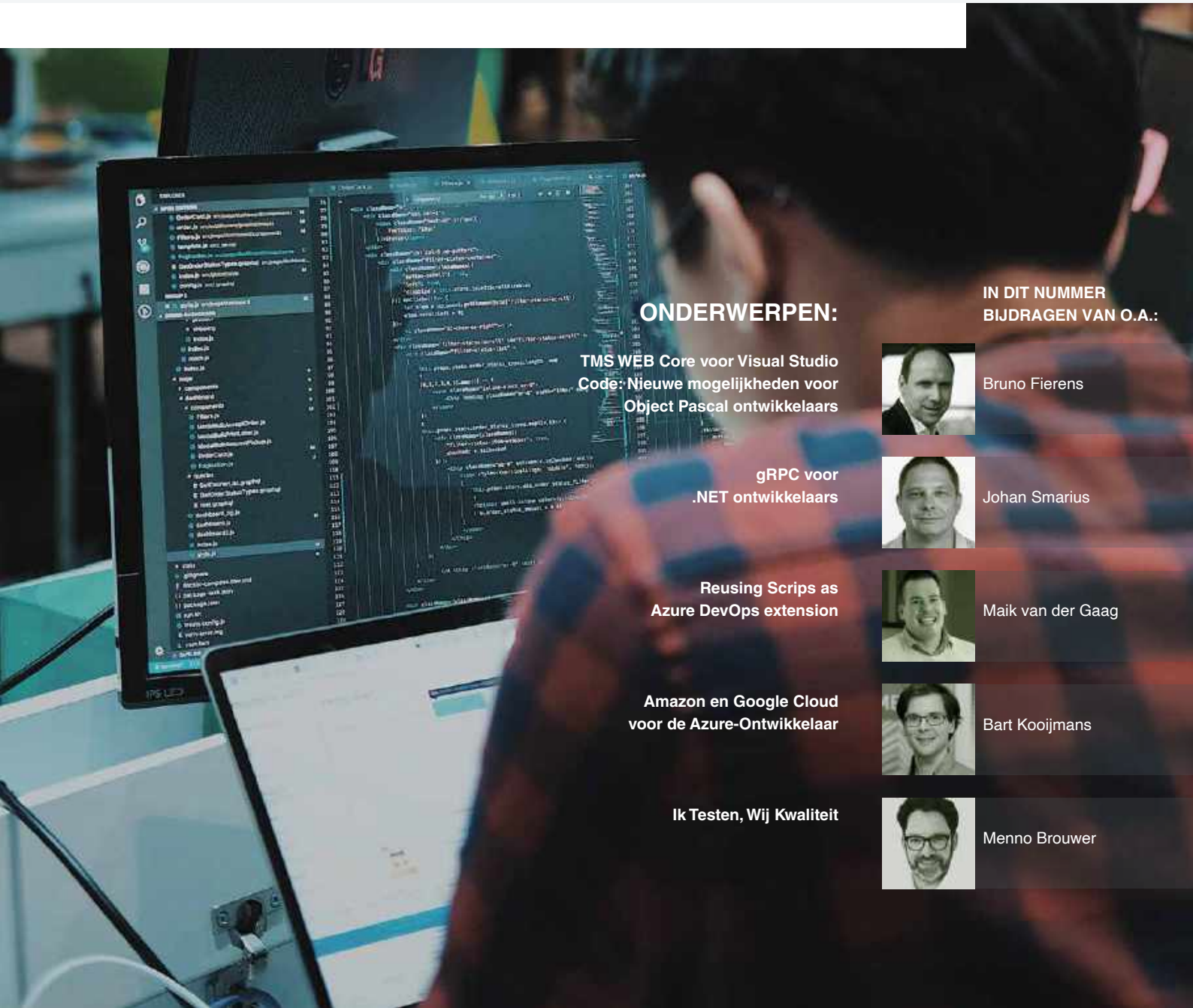
ISSN: 2211-6486

SDN MAGAZINE KNOW-LEDGE #139

SOFTWARE DEVELOPMENT NETWORK

Nummer 139 Oktober 2020 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.:

Bruno Fierens

Johan Smarius

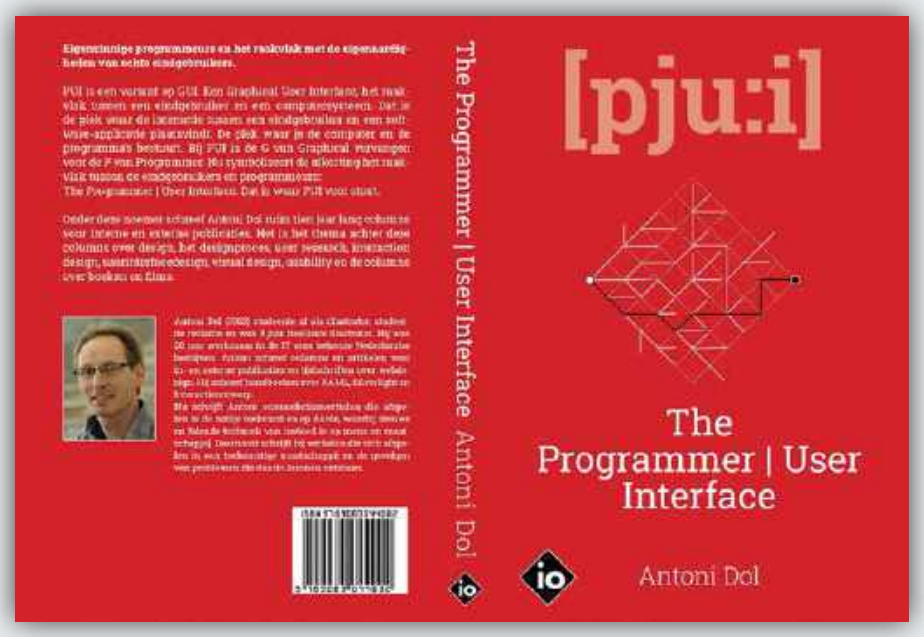
Maik van der Gaag

Bart Kooijmans

Menno Brouwer

PUI – The Programmer User Interface

Antoni Dol (boekbespreking)



Antoni Dol heeft in de afgelopen jaren veel columns geschreven over design, het designproces, user research, interaction design, userinterfacedesign, visual design, usability en de columns over boeken en films. Na bijna 18 jaar columns vond hij het tijd om een groot gedeelte te bundelen in een boek The Programmer User Interface.

Veel van de columns zijn tijdloos en zelfs met de huidige technologische vooruitgang nog steeds actueel. De columns zijn gegroepeerd volgens de hierboven genoemde onderwerpen. Er staat een hele leuk column in over Scrum: “In veel projecten zie ik een span paarden die allen ongeveer in de juiste richting trekken en met veel moeite bij elkaar worden gehouden foor de master op de bok..”. Hij beschrijft hier de 10 impediments van Scrum. In zijn columns haalt hij ook vaak weer andere boeken aan, referenties naar deze boeken staan er natuurlijk ook bij. Het is een hele interessante bonte verzameling van grappig geschreven interessante kleine stukjes. Aan de hand van dagelijkse dingen (bijv het toilet) neemt hij je mee in het design denken. Het boek nodigt uit om er een paar te lezen en het weer even weg te leggen om het later weer op te pakken. Erg leuk en verrassend tijdloos boekje en je steekt er nog wat van op ook.

Antoni heeft meerdere boeken geschreven, naast dit non-fictie boek heeft hij ook studieboeken geschreven over XAML, Interaction Design en Prototypes met online tools. Sinds kort is hij ook begonnen met Fictie. Zijn eerste boek Autonoom gaat over een programmeur met een onmogelijke opdracht en het decoderen van software. Ik denk dat ik deze binnenkort ga aanschaffen voor de koude winteravonden. Boeken geschreven door computersnerds zitten over het algemeen goed in elkaar.
Antoni zijn boeken kun je bekijken op https://antonidol.nl/ en je kunt ze hier ook bestellen. Uiteraard kun je ook bij andere boekhandels terecht. Veel leesplezier!

voorwoord

Beste SDN Magazine lezer,

De pandemie heeft ons allemaal nog in zijn greep. Het thuiswerken en online vergaderen via Zoom, Teams of andere middelen is een gewoonte geworden. Voor ons IT-ers was dat natuurlijk al een gewoonte, maar onze niet IT-vrienden zien nu ook de voordelen. Het laatste In-person event is inmiddels alweer 7 maanden geleden en ook online events beginnen een gewoonte te worden. Het voordeel hiervan is wel dat ze een groter bereik hebben. Binnenkort hebben wij weer een online SDN-event.

Weet je dat wij twee-wekelijks een online webcast hebben via Twitch en Youtube? Iedere twee weken hebben wij een interessant onderwerp door 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: TMS Web, gRPC, Testen & Kwaliteit, React, Public Clouds, Azure DevOps en Priority Queues. Onze collega's uit het community Bruno Fierens, Johan Smarius, Menno Brouwer, Maurice de Beijer, Bart Kooijmans, Maik van de Gaag, Dennis van der Stelt en Weronika Łabaj 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#139

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. 139 • oktober 2020

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
Postbus 506 7100 AM Winterswijk
Tel. (085) 21 01 310
info@sdn.nl

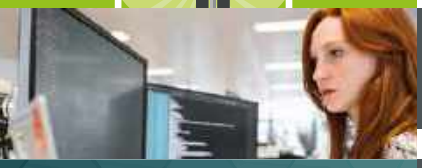
Vormgeving en opmaak
Reclamebureau Bij Dageraad
Winterswijk www.bijdageraad.nl

Adverteerders

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

SDN MAGAZINE KNOW-LEDGE #139

Inhoud



Bruno Fierens

TMS WEB CORE VOOR VISUAL STUDIO CODE:

Nieuwe mogelijkheden voor Object Pascal ontwikkelaars

Object Pascal ontwikkelaars zijn vooral gefocused op native Windows applicatie en server development met het VCL framework in Delphi alsook cross platform native applicatie development voor iOS, Android, macOS en Windows met het FireMonkey framework in Delphi en het LCL framework in Lazarus. Door de grote toename in kracht en mogelijkheden van de browsers sinds de introductie van HTML5 en CSS3 is de browser zelf als target voor rich client applicatie development zéér interessant geworden.

Al zo'n 2 jaar heeft tmssoftware.com daarom het TMS WEB Core framework ontwikkeld. Dit framework stelt Delphi en Lazarus developers in staat RAD component based rich web client development te doen vanuit het favoriete IDE. En nu introduceert tmssoftware.com ondersteuning om TMS WEB Core web client applicatie te bouwen vanuit de Visual Studio Code IDE.

WAT IS TMS WEB CORE?

TMS WEB Core is een framework en bijhorende toolchain ondersteuning om optimaal van dit framework gebruik te maken vanuit een IDE. Dit framework is geschreven in de Object Pascal programmeertaal. Met deze code kan een web client applicatie worden gemaakt dankzij de open-source pas2js compiler. Met de pas2js compiler komt tevens een run-time library die zeer nauw aansluit bij de RTL van Delphi of Lazarus. Object Pascal gebruikers kunnen hierdoor zeer veel bestaande code gewoon hergebruiken in een web client applicatie. Het TMS WEB Core framework bestaat uit een visuele en niet visuele component bibliotheek die toelaten met RAD component gebaseerd development snel web client applicaties te ontwikkelen. Dit framework is zo ontwikkeld dat het in hoge mate compatibel is met het VCL of LCL framework en Object Pascal developers dus onmiddellijk intuitief vertrouwd zijn met de omgeving.

WAT IS VISUAL STUDIO CODE?

Voor softwareontwikkelaars die niet bekend zijn met Visual Studio Code: dit is een gratis, open-source, uitbreidbare en platform-onafhankelijke IDE. Dit kan gedownload worden van https://code.visualstudio.com/ Dit betekent dat de Visual Studio Code IDE exact hetzelfde werkt op Windows, macOS en Linux en dit ook met volledige high-DPI ondersteuning. Visual Studio Code is een initiatief van Microsoft en richt zich op open en uitbreidbaar zijn. Het is open source en volledig gratis en is niet te verwarren met de reguliere Microsoft Visual Studio IDE. Er is ondertussen een zeer rijk aanbod aan extensies voor Visual Studio Code met talrijke ondersteunde programmeertalen en allerlei interessante IDE-tools, hulpprogramma's, handige extensies. Momenteel zijn meer dan 3000

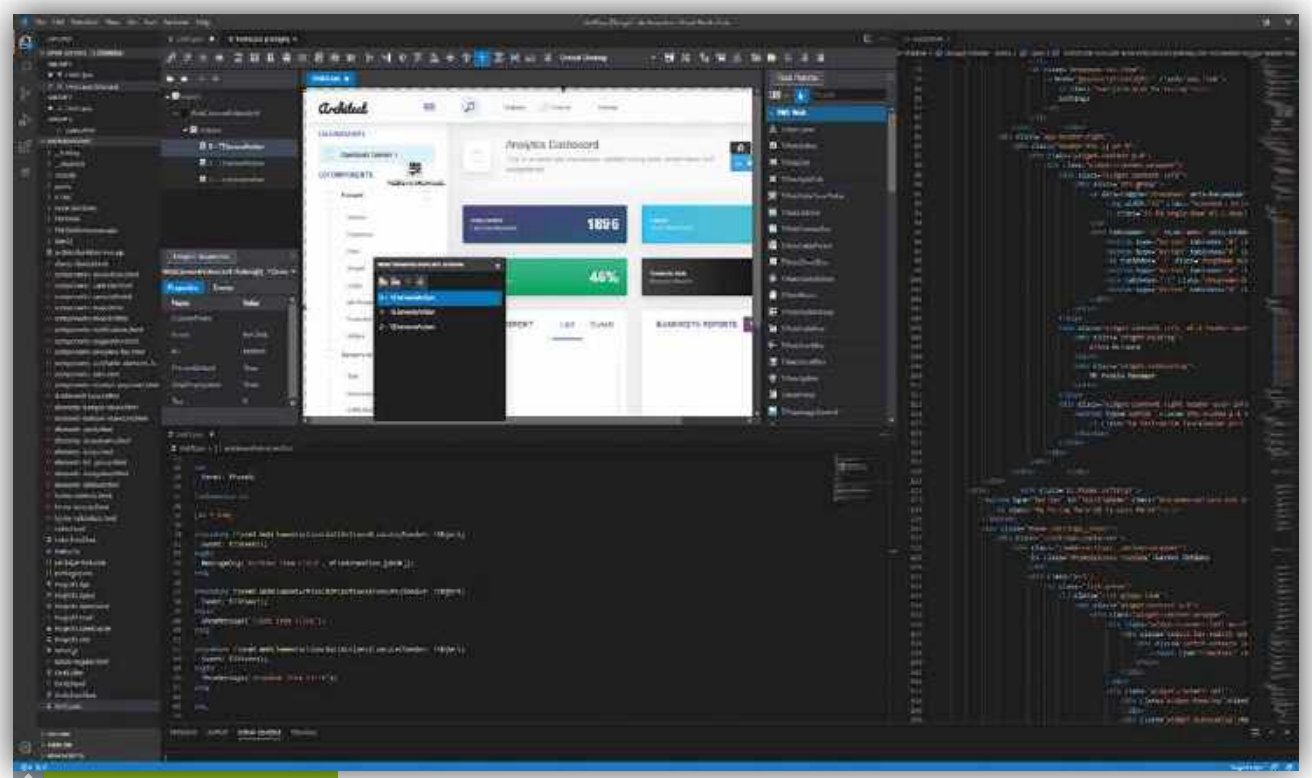


FIGURE 1

dergelijke extensies te vinden op https://marketplace.visualstudio.com/. De laatste ontwikkeling van Visual Studio Code is zelfs een online versie, dus je kan de IDE ook vanuit de cloud gebruiken.

WAT BIEDT TMS WEB CORE FOR VISUAL STUDIO CODE?

Voor zover we weten, biedt op dit moment noch Visual Studio Code noch één van de vele extensies een RAD, visueel component gebaseerd formulier ontwerp voor het bouwen van applicaties. Omdat RAD altijd de kern is geweest van wat Delphi of Lazarus betekent voor Object Pascal-ontwikkelaars, biedt TMS WEB Core voor Visual Studio Code een op RAD-componenten gebaseerde applicatie ontwikkelingstool. Dit houdt in: toolpalet, object inspector, structuur venster, form designer, code venster, debuggen. Met andere woorden, het stelt de Delphi- of Lazarus-ontwikkelaars in staat om op vrijwel dezelfde manier aan TMS WEB Core-web client applicaties te werken vanuit Visual Studio Code.

WAAROM DE KEUZE VOOR VISUAL STUDIO CODE?

De reden voor de inspanningen om een TMS WEB Core voor Visual Studio Code extensie te ontwikkelen, is puur technologiegedreven. Visual Studio Code zelf is gebouwd met behulp van webtechnologie. De gebruikersinterface van Visual Studio Code wordt volledig weergegeven met HTML / CSS / JavaScript via de Chrome-browser-engine. Dit opent de schitterende mogelijkheid om een formulier designer te bouwen op basis van webtechnologie, wat betekent dat we op de formulier designer de TMS WEB Core-webcomponenten kunnen zien zoals ze zijn, dus WYSIWYG. Waar in de Delphi IDE de formulier designer een VCL gebaseerde formulier designer is en tijdens het ontwerpen de webcomponenten worden nagebootst als VCL-besturingselementen, hebben we hier de echte webcomponent weergegeven in de formulier designer, net zoals deze in de browser wordt weergegeven wanneer de applicatie wordt gegenereerd. Zelfs tijdens het ontwerpen kan een CSS-bibliotheek zoals Bootstrap worden gebruikt en deze toont de user interface controls op de formulier designer met behulp van de geselecteerde Bootstrap-klasses & themes. Het feit dat de IDE zelf gratis, licht, snel, modern is en zowel op Windows, macOS en Linux draait, is natuurlijk een fantastisch en welkom bijkomend voordeel. Een ander niet over het hoofd te zien technisch voordeel is dat Visual Studio Code browser communicatie voorziet om het debuggen van Object Pascal-code vanuit de IDE eenvoudig te maken (in tegenstelling tot het debuggen van JavaScript-code vanuit de browserconsole).

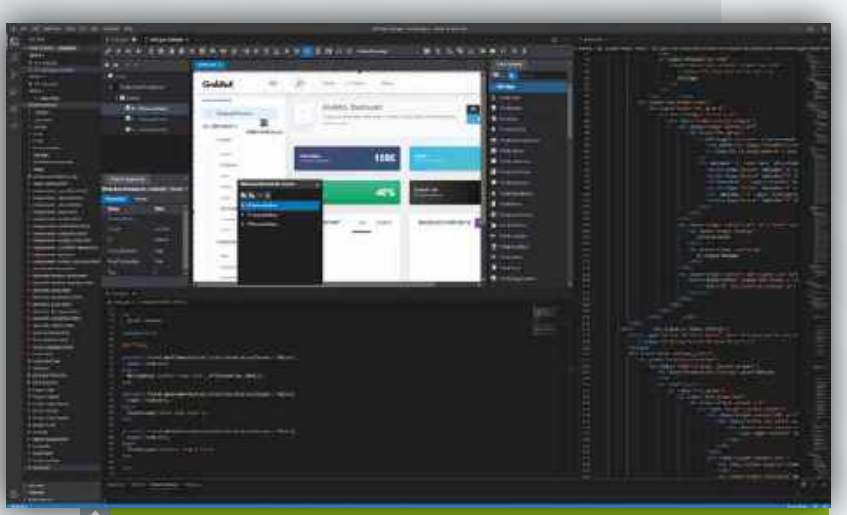


FIGURE 2: LIVE DATA VAN EEN SERVER GEVISUALISEERD IN DE FORMULIER DESIGNER

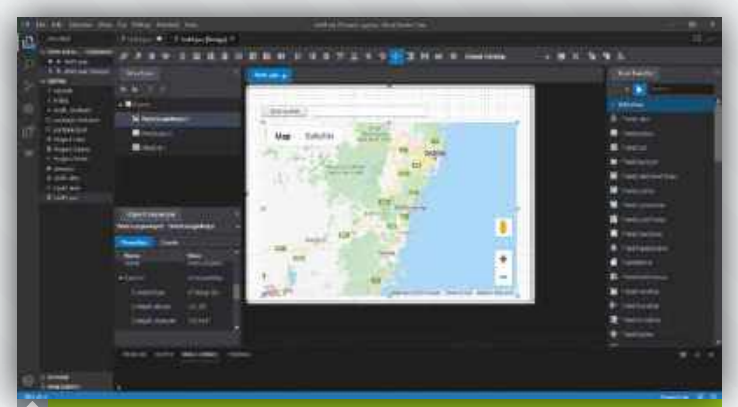


FIGURE 3: DE INGESLOTEN GOOGLE MAPS CONTROL LIVE IN DE FORMULIER DESIGNER

VEREISTEN?

De OmniPascal Visual Studio Code-extensie (https://www.omnipascal.com/) biedt al uitstekende Object Pascal-syntax highlighting, code-completion en verschillende andere functies om het coderen met Object Pascal te faciliteren. Een interessante bijkomstigheid is dat OmniPascal intern gebruik maakt van de Delphi AST-engine, geschreven door Roman Yankovski, productmanager van TMS FixInsight (https://www.tmssoftware.com/site/fixinsight.asp). Een andere vereiste is natuurlijk de pas2js-compiler (https://wiki.freepascal.org/pas2js). Dit is de compiler die de magie van het compileren (of transpileren als je de voorkeur geeft aan deze terminologie) van de Object Pascal-code naar JavaScript in de browser brengt. Dit betekent Delphi of Lazarus ontwikkelaars alles wat er in de browser gebeurt onder controle hebben met de strict getypeerde en object geörienteerde Object Pascal taal. Gelukkig is de extensie TMS WEB Core for Visual Studio Code zo ontwikkeld dat deze beide prerequisites volledig automatisch in de IDE worden geïnstalleerd.

HOE VERHOUDT HET ZICH TOT TMS WEB CORE VOOR DELPHI OF LAZARUS?

Het goede nieuws hier is dat het volledige TMS WEB Core-framework ongewijzigd werkt onder Visual Studio Code. Er is dus volledige pariteit van de framework-functies tussen Delphi, Lazarus en Visual Studio Code. We hebben het ook als een ontwikkelingsvereiste gesteld om de mogelijkheid te bieden om door Delphi gemaakte TMS WEB Core-projecten te openen vanuit Visual Studio Code en vice versa. Omdat TMS WEB Core voor Visual Studio Code exact dezelfde framework code gebruikt als Delphi, betekent dit dat wanneer functies of verbeteringen aan het Delphi-product worden toegevoegd, dit automatisch wordt overgenomen in de Visual Studio Code-versie.

Het omgekeerde is natuurlijk al waar. Een belangrijk verschilpunt echter is dat Visual Studio Code werkt volgens het concept : één project is één folder. Dit heeft als gevolgd dat je binnen een folder slechts één TMS WEB Core project kan beheren met Visual Studio Code. Dit is op zich een kleine beperking daar de meeste ontwikkelaars ook zo werken vanuit Delphi of Lazarus.

WAT BETEKENT DIT VOOR TMS WEB CORE FOR DELPHI?

TMS WEB Core voor Visual Studio Code is geen vervanging voor TMS WEB Core voor Delphi. Het is niets meer en niets minder dan nog een extra bijkomende IDE-keuzemogelijkheid. Elke ontwikkelaar kan dus voor zichzelf beslissen welke IDE de voorkeur heeft: Delphi, Lazarus of Visual Studio Code. Het feit dat Visual Studio Code & Lazarus platform-onafhankelijk zijn, d.w.z. direct vanuit macOS of Linux kunnen worden gebruikt, kan natuurlijk een doorslaggevende factor zijn. Verder wordt Delphi of Lazarus voor veel meer gebruikt dan alleen het ontwikkelen van TMS WEB Core webclientapplicaties, bijvoorbeeld de ontwikkeling van een TMS XData of Embarcadero RAD server REST back-end.

WAT ZIT STANDAARD IN TMS WEB CORE?

Het basis TMS WEB Core framework komt al met een ganse set visuele en niet visuele componenten. Alle standaard user interface controls zoals een edit, checkbox, memo, button, listbox, grid, treeview, menu etc… zijn allemaal aanwezig. De visuele controls zijn zo gebouwd dat hun programmeer interface zo nauw mogelijk aansluit bij de VCL equivalenten. Zo is er TWebEdit die het equivelent is voor de VCL TEdit en dus bijvoorbeeld ook een TWebEdit.Text: string property heeft. Of zo is er TWebStringGrid als equivalent voor TStringGrid waarbij er naar analogie een property TWebStringGrid. Cells[Col,Row: integer]: string is voor access naar de cell data.

WERKT TMS WEB CORE OOK MET VISUAL STUDIO CODE ONLINE?

Visual Studio Code Online is momenteel in bèta en belooft een IDE in de cloud te brengen. Een IDE die u eenvoudig vanuit de browser kunt gebruiken. Elke machine met een browser kan overal en altijd worden gebruikt om te ontwikkelen, ontwikkelen, ontwikkelen ... (ik hoor Steve Ballmer hier ergens).
TMS WEB Core is getest met Visual Studio Code Online en ook vanuit de browser is het mogelijk om webclient applicaties voor de browser te maken. Ook in dit gebied is de cirkel gesloten.

AAN DE SLAG MET TMS WEB CORE FOR VISUAL STUDIO CODE

Na installatie van TMS WEB Core in Visual Studio Code krijgt de gebruiker een extra icon in de activity bar (linker kant IDE). Dit opent de TMS Repository van waar gekozen kan worden een nieuw project te starten of een nieuwe form, datamodule of unit toe te voegen aan een project.

FIGURE 4

Na creatie van een project verschijnen de bestanden van dit project in de Explorer ook links (net zoals voor andere Visual Studio Code projecten). Voor een TMS WEB Core applicatie betekent dit het project .DPROJ, .DPR en .HTML bestand en daarnaast het .PAS, .DFM en .HTML bestand van een eerste formulier binnen dit project. Dit is een identieke project structuur als in de Delphi IDE wordt gebruikt.

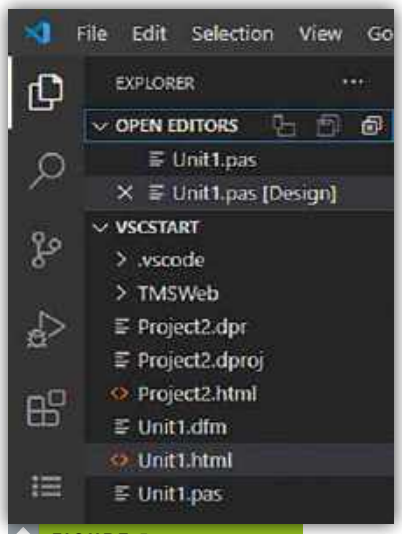


FIGURE 5

Bij het openen van de bron-code van het eerste formulier unit1.pas verschijnt de code. De formulier designer wordt geactiveerd met de sneltoets Ctrl+F12. Hiermee heeft U links van de formulier designer het structuur venster en object inspector en rechts de tool palette.
U kan vervolgens componenten slepen op het formulier en door middel van de object inspector eigenschappen wijzigen en events toevoegen.

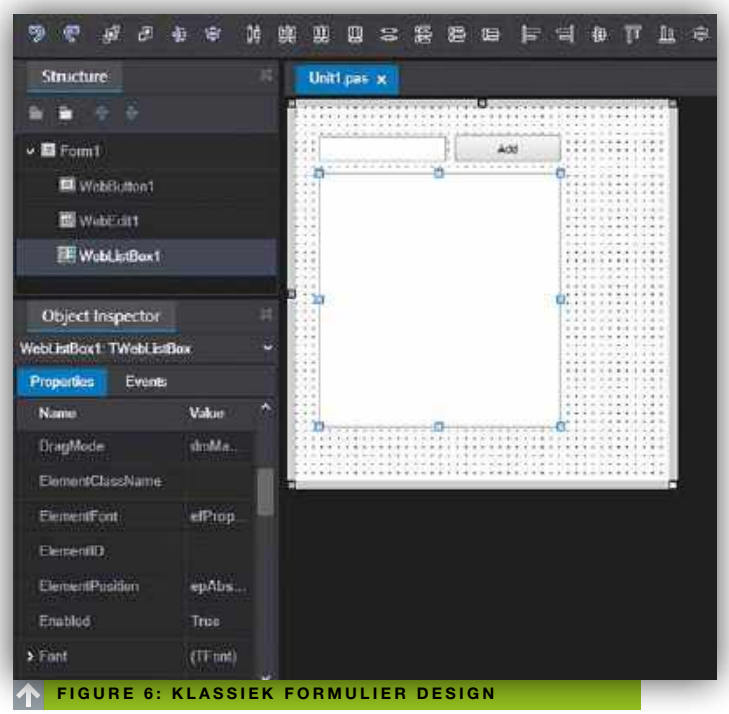
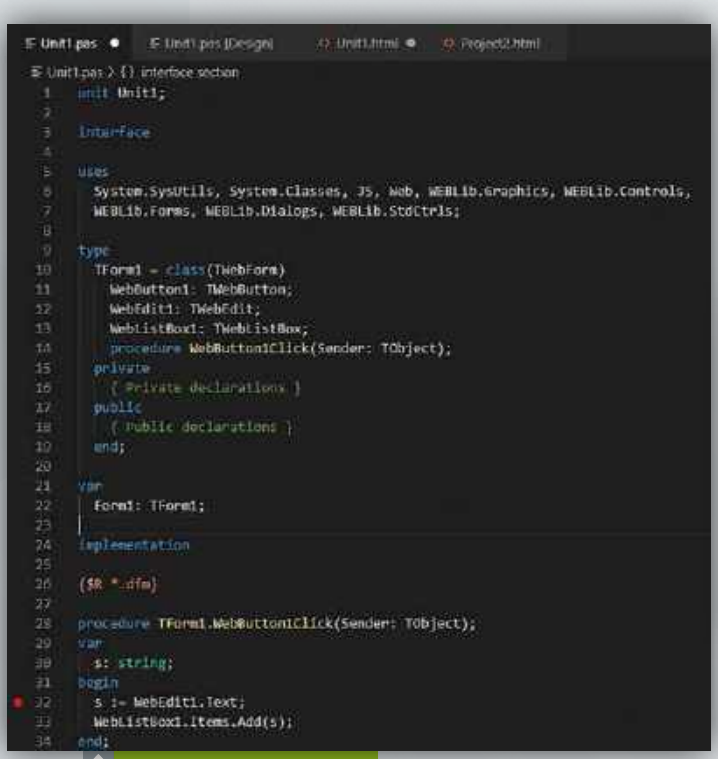


FIGURE 6: KLASSIEK FORMULIER DESIGN

Nieuwe mogelijkheden voor Object Pascal ontwikkelaars

Zoals U merkt, allemaal heel vertrouwd voor Object Pascal ontwikkelaars. Met de F5 toets wordt de compiler gestart en vervolgens de applicatie gedraaid in de browser. Vanuit de code window kunnen breakpoints worden aangebracht.



```
unit Unit1;

interface

uses
  System.SysUtils, System.Classes, JS, Web, WEBLib.Graphics, WEBLib.Controls,
  WEBLib.Forms, WEBLib.Dialogs, WEBLib.StdCtrls;

type
  TForm1 = class(TWebForm)
    WebButton1: TWebButton;
    WebEdit1: TWebEdit;
    WebListBox1: TWebListBox;
    procedure WebButton1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.WebButton1Click(Sender: TObject);
var
  s: string;
begin
  s := WebEdit1.Text;
  WebListBox1.Items.Add(s);
end;
```

FIGURE 7

Tot zover is dit alles heel gelijkaardig met het bouwen van een Windows applicatie met het VCL of LCL framework. Het moment dat gestart wordt met het gebruik van templates opent de kracht van het gebruik van web design (HTML/CSS) zich voor TMS WEB Core. Om dit te illustreren kunnen we in een eerste stap beginnen met het opnemen van de Bootstrap library voor het stylen van de applicatie.

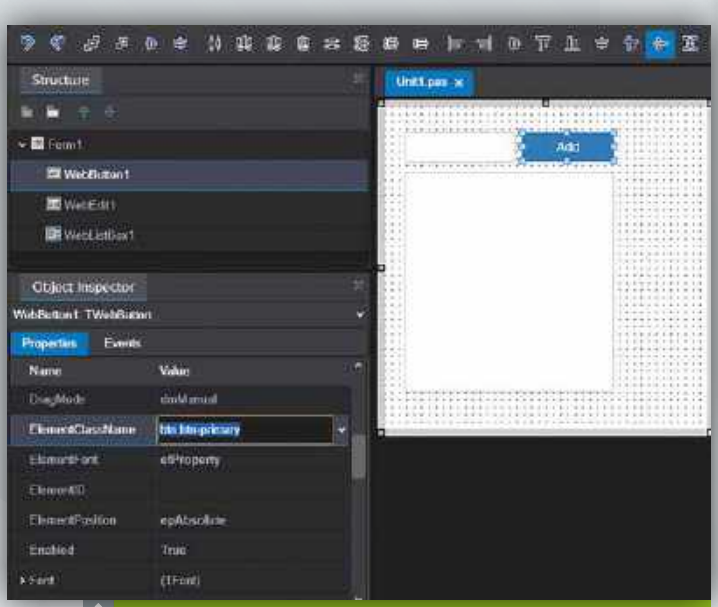
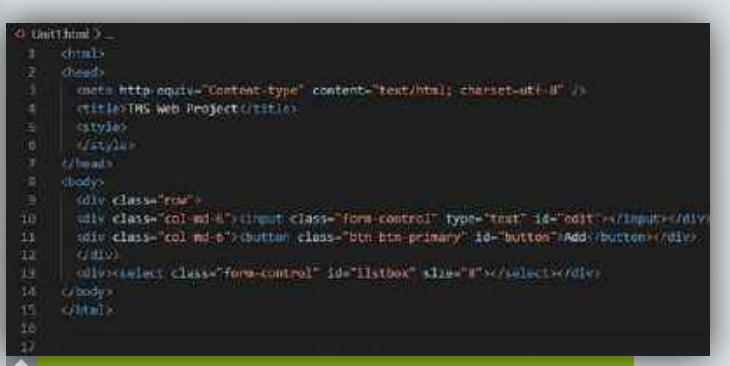


FIGURE 8: KLASSIEK FORMULIER DESIGN MAAR MET BOOTSTRAP STYLING

Dit gebeurt door een link toe te voegen in de project HTML en vervolgens op de controls de te gebruiken CSS class te specifiëren via de ElementClassName property. In dit eenvoudig voorbeeld plaatsen we de 'form-control' CSS class op TWebEdit en TwebListBox en de 'btn btn-primary' class op TWebButton.

In een volgende stap gaan we opnieuw verder weg van het klassieke VCL formulier design en specifiëren we de opmaak van het formulier in HTML. Om tenslotte de user-interface logica te verbinden met de HTML, linken we de user-interface controls met de HTML elementen door de control.ElementID property in te stellen op de ID van de HTML elementen.
De HTML voor het formulier wordt:



```
<!doctype html>
<html>
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <title>TMS Web Project</title>
  <style>
  </style>
</head>
<body>
  <div class="row">
    <div class="col-md-6"><input class="form-control" type="text" id="edit"></input></div>
    <div class="col-md-6"><button class="btn btn-primary" id="button">Add</button></div>
  </div>
  <div><select class="form-control" id="listBox" size="8"></select></div>
</body>
</html>
```

FIGURE 9: HTML TEMPLATE CODE VOOR HET FORMULIER

Verbonden met de control op de form designer ziet dit er in de IDE uit als:

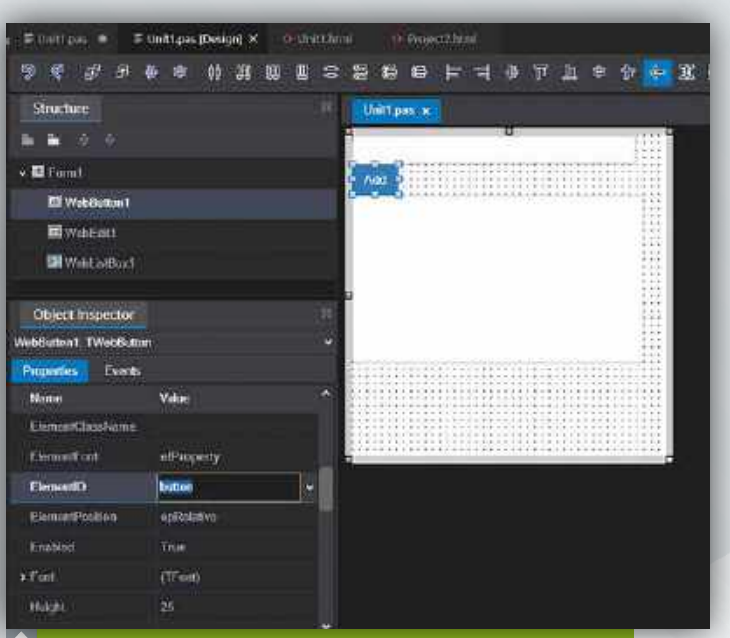


FIGURE 10: LAYOUT VOLLEDIG DOOR HTML/CSS GECONTROLEERD EN UI LOGIC VERBONDEN IN FORMULIER DESIGNER

Hier zien we WebButton1 via ElementID "button" gelinkt met het HTML button element. Er zijn Bootstrap classes voor de DIV elementen gebruikt die meteen dit formulier ook een basis responsive design geven. Als het browser venster breed genoeg is, zullen de DIV elementen voor de HTML INPUT contrl hier gelinkt aan TWebEdit1 en de HTML BUTTON op één rij staan.

Voor een kleiner scherm wordt automatisch overgeschakeld naar kolom weergave. Met andere woorden, er is een loskoppeling tussen de opmaak (via HTML/CSS) en de code in Object Pascal.
Uiteraard hebben we hier maar een fractie van de mogelijkheden van TMS WEB Core aangeraakt. TMS WEB Core komt intussen met al bijna 100 verschillende demos, tientallen videos en er is ook reeds een boek op de markt die het framework beschrijft:

https://www.amazon.com/TMS-WEB-Core-Application-Development/dp/B086G6XDGW

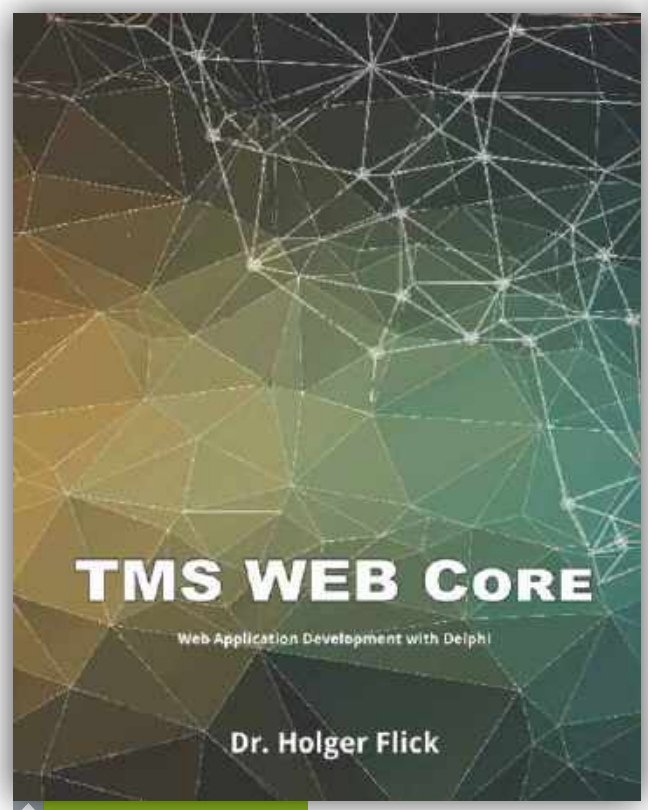


FIGURE 11

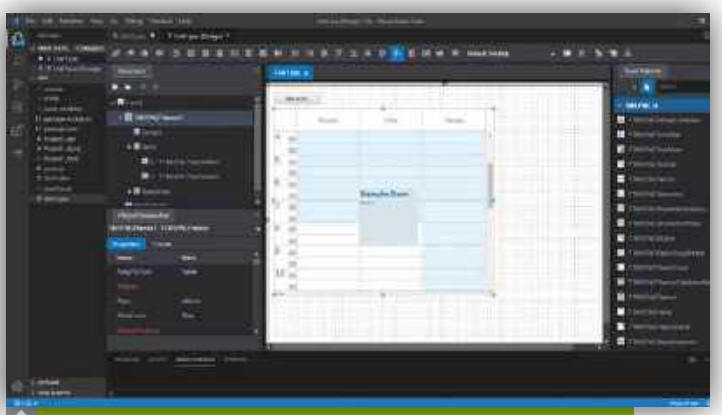


FIGURE 12: EEN BLIK OP DE TOEKOMST, TMS FNC PLANNER OP DE VISUAL STUDIO CODE FORMULIER DESIGNER

ROADMAP

v1.0 is duidelijk de eerste stap, het fundament om met het TMS WEB Core framework in Visual Studio Code web client applicaties te ontwikkelen. Er zitten al heel wat verdere evoluties in de pipeline. Enkele van de belangrijkste en reeds publiek aangekondigde features voor volgende versies van TMS WEB Core for Visual Studio Code zijn:

- Ingebouwde ondersteuning voor het maken van PWA projecten
- Ingebouwde ondersteuning voor het maken van cross-platform desktop appplicaties met het Electron framework
- Ondersteuning voor installeerbare 3rd party componenten waaronder de volledige TMS FNC libraries van cross-platform/ cross-framework componenten
- Ondersteuning voor nieuwe Object Pascal taal eigenschappen
- Templates en wizards voor automatische applicatie generatie

CONCLUSIE

Visual Studio Code is een relatief jonge en op web technologie gebaseerde IDE. Het was een technisch logische en voor de hand liggende keuze om hiervoor de nodige ondersteuning te bouwen om het TMS WEB Core framework voor RAD component gebaseerd web client ontwikkeling hierin te integreren. Met versie 1.0 is de eerste stap gezet en verdere ontwikkelingen worden gedaan om dit uit te bouwen tot een zeer productief en rijk platform.

BRUNO FIERENS

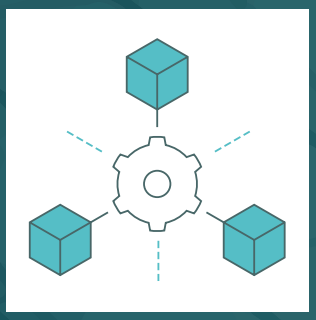
- Studied civil electronic engineering at university of Ghent, Belgium (1987-1992).
- Started a career as R&D digital hardware engineer at Barco Graphics Belgium designing with FPGA, VHDL, graphic processors, PCI, Silicon Graphics add-on boards, high-end printer controllers,...
- Began writing software in Turbo Pascal 3.0 since 1987 and used all Borland Pascal and all Delphi versions since then.
- Founded TMS software in 1996, developing VCL components starting with Delphi 1.
- TMS software became Borland Technology Partner in 1998 and developed Delphi Informant award-winning grid & scheduling components.
- From 2001 IntraWeb component development was started.
- From 2003 ASP.NET component development was started.
- From 2011 FireMonkey cross platform components,targetting Windows, macOS, Android, iOS.
- In 2016, TMS software launched FNC, a framework neutral component architecture enabling to use UI controls in VCL,FMX & LCL apps
- In 2018, TMS software launched TMS WEB Core, a framework for creating rich web clients using ObjectPascal
- Currently doing and managing VCL, FMX, Web, .NET, IoT, LCL, REST, node.js development.
- Is a regular speakers at conferences (Be-Delphi, DelphiTage, ITDevCon, CodeWay Tour, EKON, DevTracks, SDN, ..).
- Available for consulting & custom project development.
- Bruno Fierens was titled Embarcadero MVP since 2012.
- Special area of interest are user interfaces design, UX, RAD software development, hardware/electronics.

Johan Smarius

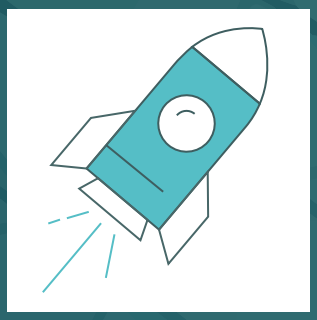
gRPC voor .NET ontwikkelaars

gRPC

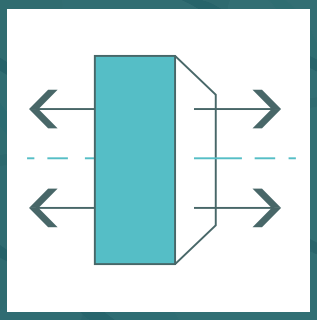
Simple service definition



Start quickly and scale



Works across languages and platforms



Bi-directional streaming and integrated auth

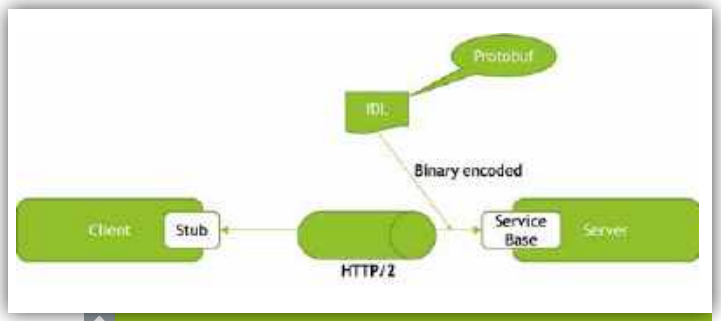
In de applicaties die we tegenwoordig schrijven is connectiviteit tussen applicaties en tussen applicatie-onderdelen heel erg belangrijk. Op dit moment wordt daar binnen het .NET framework eigenlijk met name WebApi voor gebruikt. Met de komst van .NET Core 3 hebben we hier nog een andere mogelijkheid bij gekregen namelijk gRPC.

In dit artikel zal ik jullie laten zien hoe je met gRPC kunt werken binnen een .NET omgeving. Om gRPC te kunnen gebruiken, heb je geen uitgebreide editie van Visual Studio 2019 nodig. De community editie is voldoende.

WAT IS GRPC?

gRPC is een variant van een RPC-stack gemaakt door Google en is sinds 2015 al open source. De techniek is dus zeker niet nieuw, maar was nog relatief onbekend bij de .NET ontwikkelaars. De uitgangspunten voor het ontwerp van gRPC zijn dat het lichtgewicht moet zijn, een hoge performance moeten bieden en platform en ontwikkeltaal onafhankelijk moet zijn. Google heeft als basis voor gRPC protocol buffers en HTTP/2 gebruikt.

In Figuur 1 staat de globale architectuur van een systeem dat via gRPC communiceert.



FIGUUR 1: GLOBALE ARCHITECTUUR GRPC

Een centrale rol in gRPC is er voor protobuf. Dit is de interface definition language voor gRPC. Een interface definition language is een taal die los staat van een specifieke ontwikkeltaal en die gebruikt wordt om de interface tussen systemen te beschrijven. Deze beschrijving kan dan weer gebruikt worden om taal specifieke code te genereren. In ons geval kan aan de hand van het protobuf bestand een klasse ServiceBase gegenereerd worden die als basisklasse dienst kan doen voor het implementeren van de code voor de service. Aan de client kant kan aan de hand van hetzelfde protobuf-bestand een stub gegenereerd worden die gebruikt kan worden als proxy om te communiceren met de service. De communicatie tussen client en server gebeurt aan de hand van een binair formaat.
Protobuf speelt hierin een dubbele rol, want protobuf wordt ook gebruikt voor het binair serialiseren en deserialiseren van de data die over de lijn gaat.

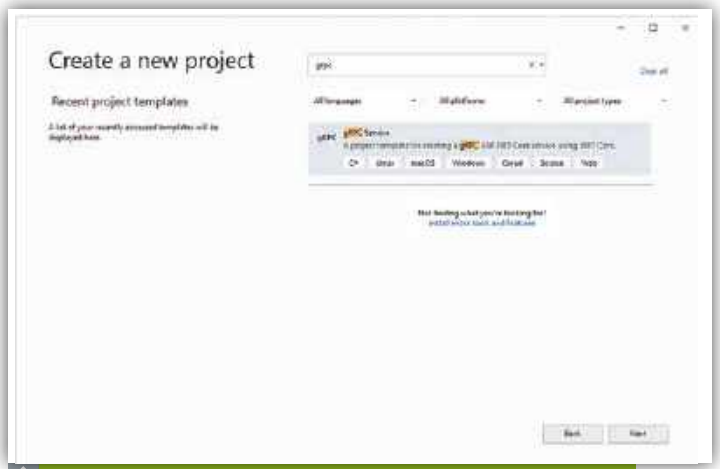
De client en de server hoeven niet per se in dezelfde ontwikkeltaal geschreven te worden. Met gRPC is het goed mogelijk om een C# server bijvoorbeeld te laten communiceren met een Java of Python client. De tooling van gRPC ondersteunt op dit moment 13 verschillende ontwikkeltalen.

OPZETTEN GRPC SERVER

Voor het opzetten van een gRPC server kun je gebruik maken van een template binnen Visual Studio 2019.

FIGUUR 2 STARTSCHERM VISUAL STUDIO 2019

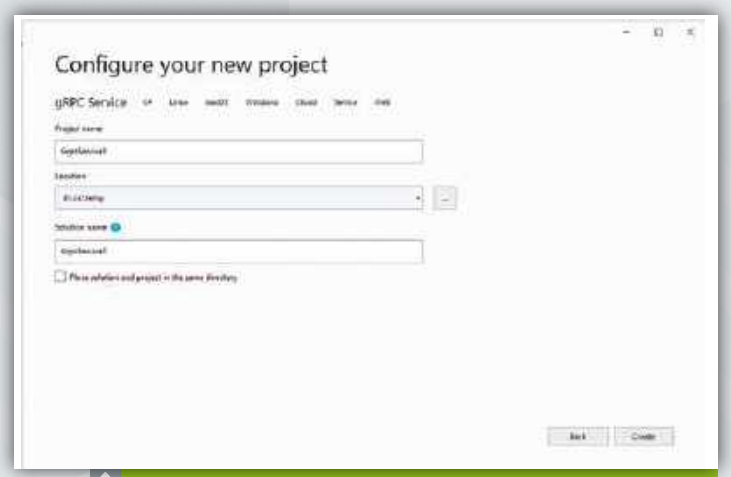
In het startscherm kun je kiezen voor “Create a new project” (Figuur 2).



FIGUUR 3: CREATE PROJECT PAGE

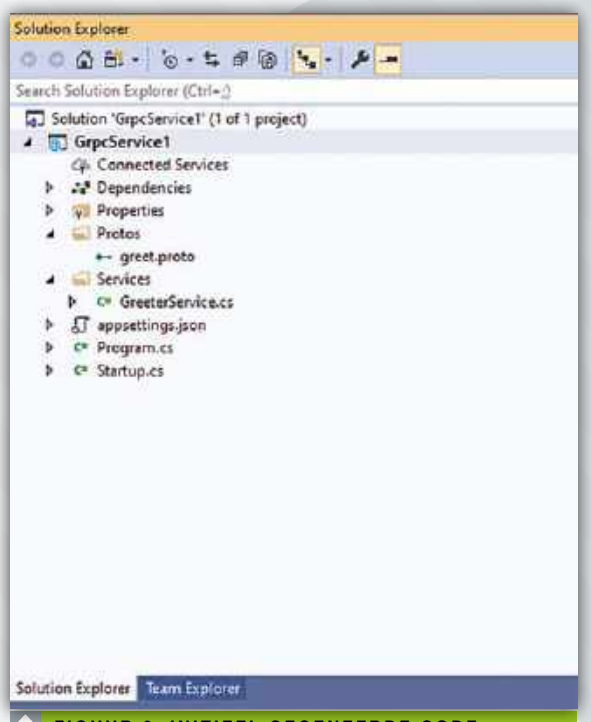
In de template pagina (Figuur 3) kun je nu zoeken naar gRPC. Voor gRPC is er op dit moment maar een template beschikbaar. Als je deze template kiest en op “Next” drukt, dan kom je terecht op de pagina die voor alle templates gebruikt wordt om de project naam, locatie en solution name in te stellen (Figuur 4).

Als je nu op “Create” klikt dan kun je nog wat aanvullende opties instellen (Figuur 5).



FIGUUR 4: PROJECTCONFIGURATIE

FIGUUR 5: SERVICE CONFIGURATIE



FIGUUR 6: INITIEEL GEGENEERDE CODE

Op dit moment is er nog niet veel te kiezen. Alleen Docker ondersteuning is beschikbaar als keuze. Als je nu weer op “Create” drukt, dan wordt de code voor je server gegenereerd.

REVIEW GEGENEREERDE CODE

In de genereerde code (Figuur 6) kun je een aantal gRPC specifieke bestanden terugvinden. Het “greet.proto” bestand is het protobuf-bestand en de GreeterService is de gegenereerde servicecode. Laten we eerst even kijken naar de inhoud van het proto-bestand.

```
1   syntax = "proto3";
```

FIGUUR 7: VERSIE PROTOBUF

Als eerste regel in het proto-bestand (Figuur 7) is de versie van de syntax opgenomen. Het is belangrijk om deze regel op te nemen, omdat er meerdere versies van protobuf bestaan en gRPC het beste werkt met versie 3.

```
3   option csharp_namespace = "GrpcService1";
```

FIGUUR 8: C# SPECIFIEKE TOEVOEGING

In een protobuf-bestand kun je ook taal specifieke opties opgeven. In Figuur 8 gebruiken we deze optie om de namespace op te geven die gebruikt zal worden voor het genereren van de code.

```
5   package greet;
```

FIGUUR 9: PACKAGE

Standaard wordt door Visual Studio ook een package statement in het bestand opgenomen. De package wordt gebruikt als een soort namespace binnen protobuf. Met behulp van packages kun je dus naamconflicten voorkomen binnen protobuf definities. Dit statement is optioneel en mag je weglaten als je hier geen behoefte aan hebt. Voor de code generatie wint de optie csharp_namespace het van de instelling van package. Als je dus net zoals in het voorbeeld beide opneemt, dan wordt de csharp_namespace gebruikt voor de codegeneratie.
gRPC werkt met request en response berichten (Figuur 10).

```
13  // The request message containing the user's name.
14  message HelloRequest {
15    string name = 1;
16  }
17
18  // The response message containing the greetings.
19  message HelloReply {
20    string message = 1;
21  }
22
```

FIGUUR 10: BERICHTEN

Een bericht moet beginnen met het keyword “message” gevolgd door de bericht naam. Binnen de body van de message kun je de velden definiëren. Per veld moet je het type en de naam van het veld opgeven en heel belangrijk het volgnummer van het veld. Omdat gebruik wordt gemaakt van binaire serialisatie is dit volgnummer heel belangrijk. Bij het aanpassen van berichtdefinities die al gebruikt worden in productie, is het heel belangrijk om deze volgorde te handhaven.

Berichtdefinities uitbreiden met nieuwe velden met hogere volgnummers is in principe wel mogelijk. Bestaande volgnummers hergebruiken voor een ander doel is niet toegestaan. Bij alle RPC-oplossingen kies ik er persoonlijk maar zelden voor om een berichtdefinitie aan te passen. Ik gebruik veel liever expliciete versionering van de berichtdefinities om het ook voor de aanroepende partij heel duidelijk te maken welke velden verplicht gevuld moeten worden, maar dat is een persoonlijke keuze.
De berichtdefinities kunnen weer worden gebruikt in service definities (Figuur 11).

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

FIGURE 11: SERVICE DEFINITIES

Een service bestaat uit het keyword “service” gevolgd door de naam van de service. Binnen de body van de service, kun je de individuele methoden opnemen die de service ondersteund. Elke methode moet beginnen met het keyword “rpc” gevolgd door de naam van de methode en het type van de request parameter tussen haakjes. Voor de parameter geef je geen naam op. Achter de naam staat het returns keyword gevolgd door het type van de response parameter tussen haakjes. Ook voor deze parameter hoef je geen naam op te geven. Voor C# ontwikkelaars is het vaak wel even wennen dat het keyword returns heet in plaats van return.
Aan de hand van deze definitie wordt een klasse gegenereerd door de protobuf compiler met low-level code (Greeter.GreeterBase) en een afgeleide klasse (GreeterService) die de abstracte methoden implementeert uit deze basisklasse (Figuur 12). Als ontwikkelaar heb je met name met deze laatste klasse te maken.

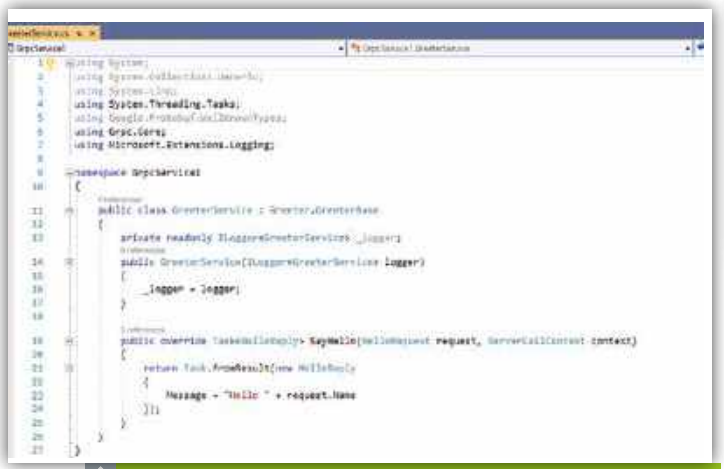


FIGURE 12: SERVICECODE

De standaard dependency injection opties binnen .NET Core zijn ook beschikbaar binnen gRPC, dus we kunnen ook gewoon gebruik maken van onder andere de logger.
Voor elke methode in het protobuf bestand wordt een aparte methode gegenereerd. Deze zijn standaard, zoals je ook wilt voor een server side call, asynchroon. Voor de messages uit het protobuf bestand, zijn klassen gegenereerd en deze kunnen worden gebruikt in de service.
Elke methode krijgt ook nog de ServerCallContext mee. Deze context bevat onder andere de header informatie die je kunt gebruiken om informatie tussen client en server uit te wisselen. In Figuur 13 kun je hier een versimpelt voorbeeld van zien.

FIGURE 13: REQUEST HEADER INFORMATIE IN DE CONTEXT

De standaard gegenereerde code voor SayHello doet niet veel meer dan een echo van de ontvangen naam.
Als je naar de instellingen (Figuur 14) kijkt van het proto-bestand, dan zijn een aantal instellingen heel belangrijk. De build actie moet staan op “Protobuf compiler”. Voor een server kun je de instelling voor “gRPC Stub Classes” op “Server only” laten staan. Later in het artikel zullen we nog verder ingaan op de andere opties voor deze instelling.
Als je zelf een protobuf bestand toevoegt aan je server dan is het belangrijk om te controleren of deze instellingen wel goed staan. Als deze instellingen niet goed staan, dan wordt er geen code voor je gegenereerd.

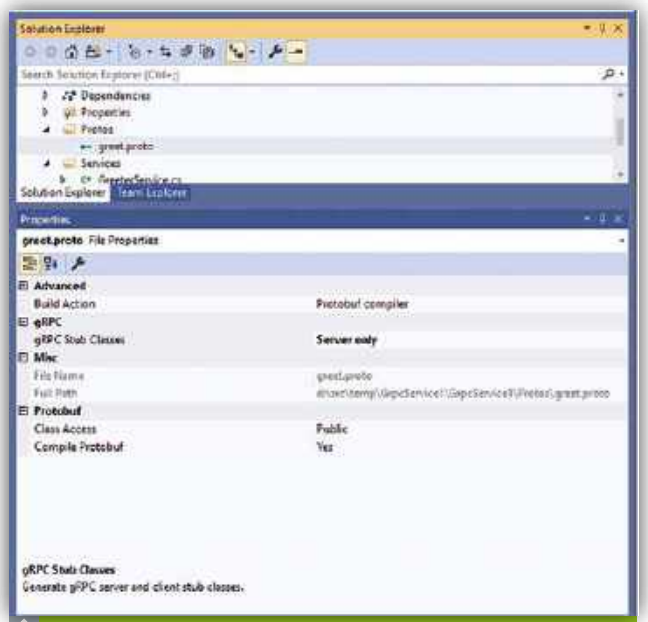


FIGURE 14: INSTELLINGEN PROTOBESTAND

De server heeft nog een paar specifieke instellingen nodig in de “Startup.cs”. In de ConfigureServices methode (Figuur 15) moet gRPC aan de service collectie toe worden gevoegd.

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

FIGURE 15: CONFIGURESERVICES

Binnen de Configure methode (Figuur 16) is het belangrijk dat de service als endpoint beschikbaar komt. Hiervoor wordt de MapGrpcService extension method gebruikt.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGrpcService<GreeterService>();

    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Communication with gRPC endpoints must be made through a gRPC client. To learn how
            to create a client, visit: https://go.microsoft.com/fwlink/?linkid=2086909");
    });
});
```

FIGURE 16: ROUTING CONFIG

Een gRPC service kan niet zomaar via het web aangeroepen worden. Daarom worden alle http verzoeken standaard afgevangen en met een foutmelding afgebroken.
De basis van gRPC is HTTP/2. Kestrel is in staat om met dit protocol te werken, maar in de appsettings (Figuur 17) moet dan wel opgenomen worden dat dit voor alle endpoints standaard gebruikt moet worden.

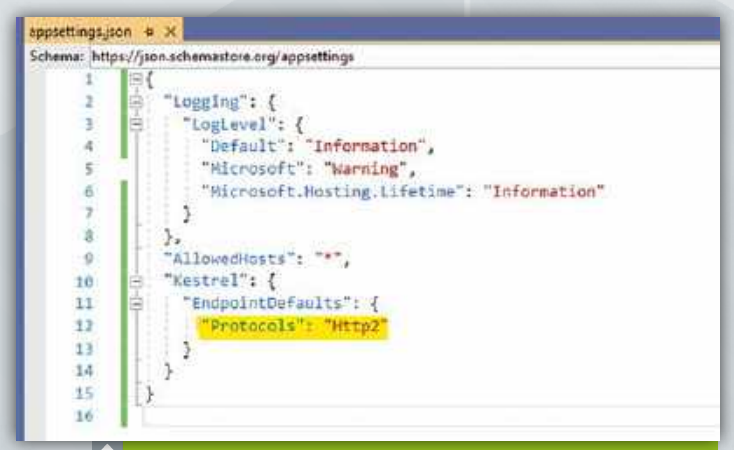


FIGURE 17: APPSETTINGS.JSON

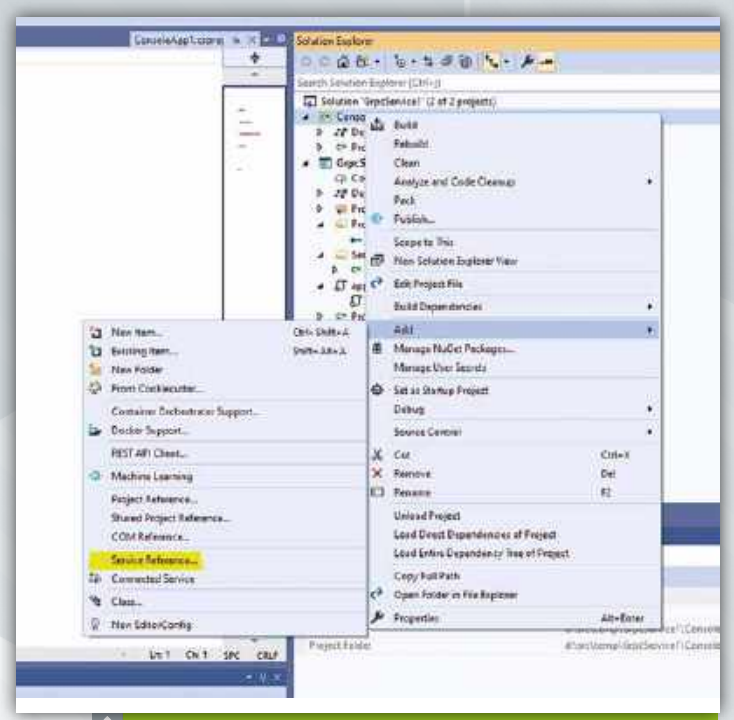


FIGURE 18: ADD SERVICE REFERENCE

GEBRUIK VAN EEN SERVICE

Een gRPC service kan niet zoals een WebAPI direct via een HTTP-endpoint aangeroepen worden in een client applicatie. Hiervoor hebben we een stub nodig. Een stub kan in Visual Studio heel makkelijk worden gemaakt. Hiervoor kan de optie Add Service Reference worden gebruikt (Figuur 18). Voor de .NET ontwikkelaars die ook met WCF hebben gewerkt, is deze optie geen onbekende.

In het scherm voor de Connected Services (Figuur 19) kan nu gekozen worden voor een service reference. De popup dialoog voor de keuze tussen OpenAPI en gRPC (Figuur 20) wordt nu getoond. Je moet hierbij kiezen voor gRPC natuurlijk.

FIGURE 19: CONNECTED SERVICES

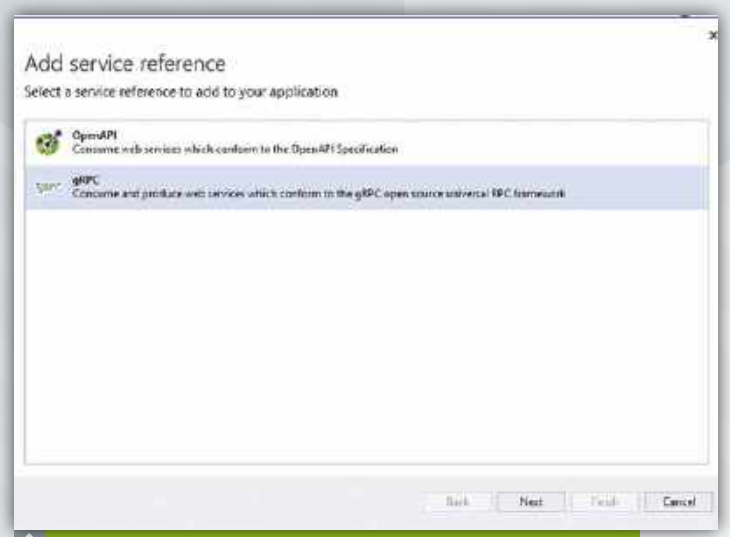


FIGURE 20: ADD SERVICE REFERENCE TYPE KEUZE

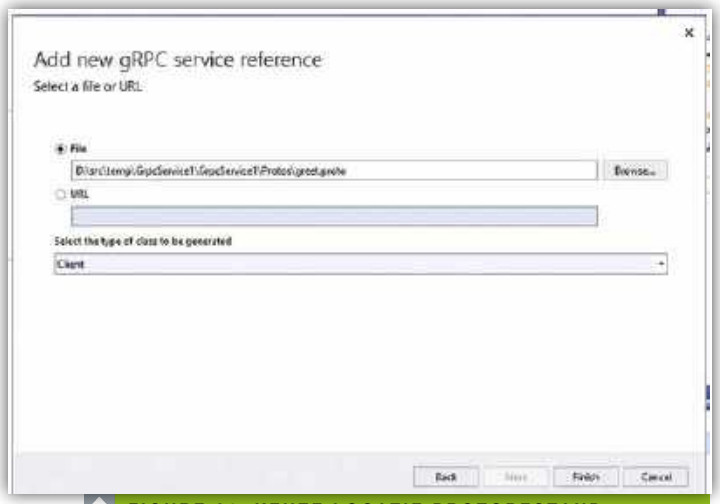


FIGURE 21: KEUZE LOCATIE PROTOBESTAND

In het vervolgscherm (Figuur 21) kun je nu kiezen waar je proto-bestand staat. Omdat we de client in dezelfde solution hebben als de service, is het voor nu het meest voor de hand liggend om voor bestand te kiezen. De keuze voor type klasse is wel belangrijk. Aan de serverkant hebben we deze instelling ook gezien in de instellingen van het proto-bestand. Aan de server wilden we graag de basisklasse laten genereren die we konden gebruiken als basis voor onze implementatie van de service. Voor een zuivere client willen we eigenlijk alleen maar de stub laten genereren. Met de optie “Client” kun je dit bereiken. Er bestaat ook nog de optie “Client en server” en “Messages only”. Bij “client en server” wordt zowel de basisklasse als de stub gemaakt. Persoonlijk houd ik van seperation of concerns, dus deze optie kies ik nooit. Met “Messages only” kan ik alleen code voor de berichtdefinities laten genereren. Voor een grotere applicatie waarin ik meer gelaagdheid heb, kan dit een zinvolle optie zijn om de berichten in een assembly los van de stub te laten genereren. Na het drukken op “Finish” kun je zien dat de nodige code gegenereerd wordt (Figuur 22).

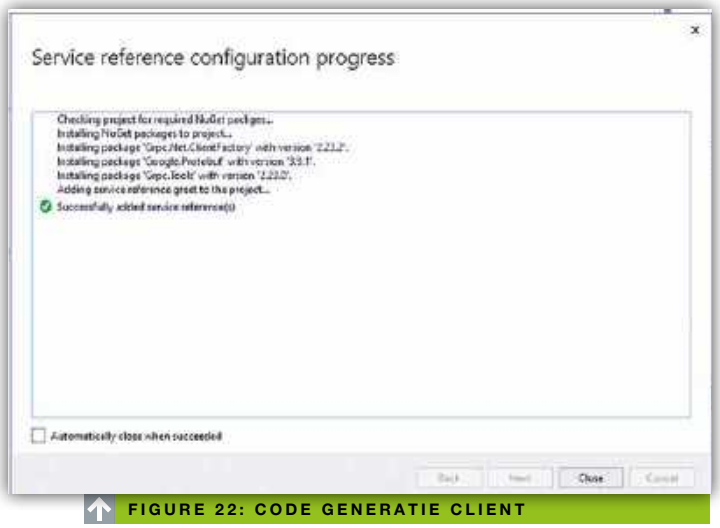


FIGURE 22: CODE GENERATIE CLIENT

In het overzicht van de gegenereerde code kun je zien dat er 3 packages nodig zijn aan de client kant: Grpc.Net.ClientFactory, Google.Protobuf en Grpc.Tools. De tool voegt deze automatisch aan je project toe. Het proto-bestand wordt ook automatisch toegevoegd aan het client project in de folder Protos.
In de client kan ik nu gebruik gaan maken van de gegenereerde stub. De code van de stub kan ik vinden in de obj/debug/netcoreapp3.1 folder (Figuur 23).

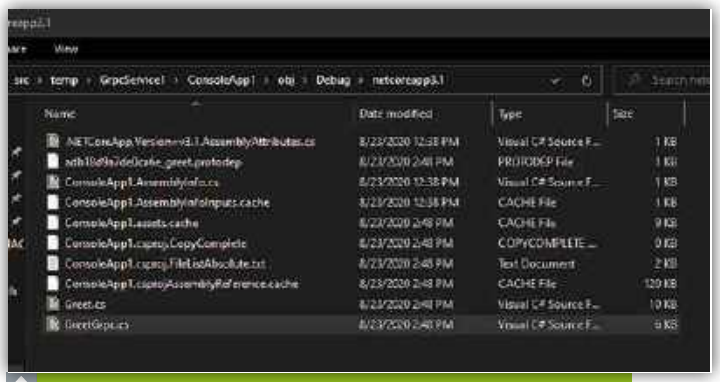


FIGURE 23: GEGENEREERDE STUB IN CLIENT

```
/// <summary>Client for Greeter</summary>
7 references
public partial class GreeterClient : grpc::ClientBase<GreeterClient>
{
```

FIGURE 24: CLIENT STUB

In het bestand GreetGrpc.cs kun je de code voor de client vinden (Figuur 24). Deze client klasse kunnen we gebruiken om te communiceren met de service.
Voordat we de client aan kunnen maken, moeten we eerst een zogenaamd channel aanmaken. Hiervoor kun je de code gebruiken in Figuur 25.

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
```

FIGURE 25: AANMAKEN CHANNEL

Dit channel kan dan weer gebruikt worden om de client te instantiëren (Figuur 26)

```
var client = new GreeterClient(channel);
```

FIGURE 26: INSTANTIËREN CLIENT

Voor alle methoden in de service is binnen de client een synchrone en asynchrone variant beschikbaar (Figuur 27).

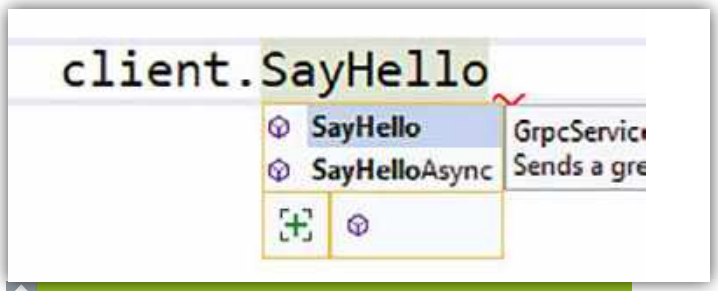


FIGURE 27:BESCHIKBARE METHODEN PER SERVER METHODE

Omdat ik hier met een simpele client te maken heb, kies ik nu voor een simpele synchrone variant (Figuur 28).

```
var response = client.SayHello(
    new GrpcService1.HelloRequest { Name = "SDN" });

Console.WriteLine(response.Message);
```

FIGURE 28: AANROEPEN SAYHELLO IN STUB

Als ik deze code nu uit wil voeren, moet ik er rekening mee houden dat mijn oplossing uit 2 uitvoerbare projecten bestaat. Om te kunnen debuggen, moet ik dus beide applicaties starten. Dit werkt het makkelijkst met de optie van Visual Studio voor Multiple startup projects binnen de optie “Set startup projects” voor een solution (Figuur 29). Je kunt nu gewoon op de start knop drukken om beide applicaties uit te voeren.

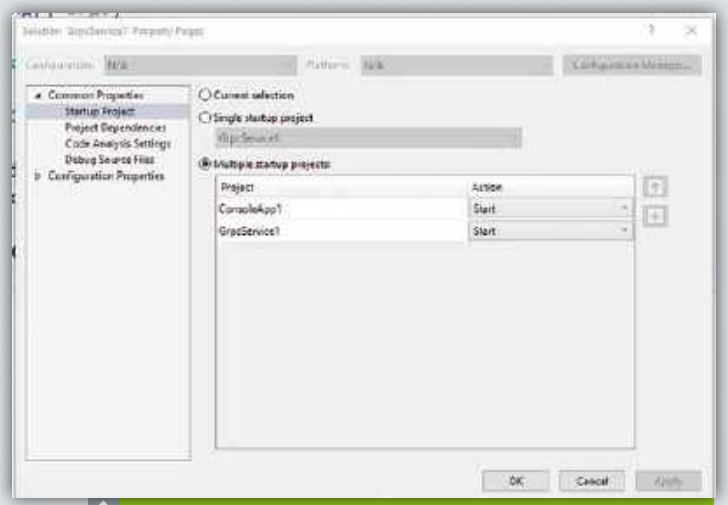


FIGURE 29: MULTIPLE STARTUP PROJECTS

FIGURE 30: SSL MELDING

FIGURE 31: INSTALLEREN CERTIFICAAT

Als je nog niet het SSL development certificaat van Visual Studio hebt vertrouwd, dan krijg je een melding (Figuur 30). Je kunt dit certificaat veilig vertrouwen en installeren (Figuur 31) voor je ontwikkelmachine.

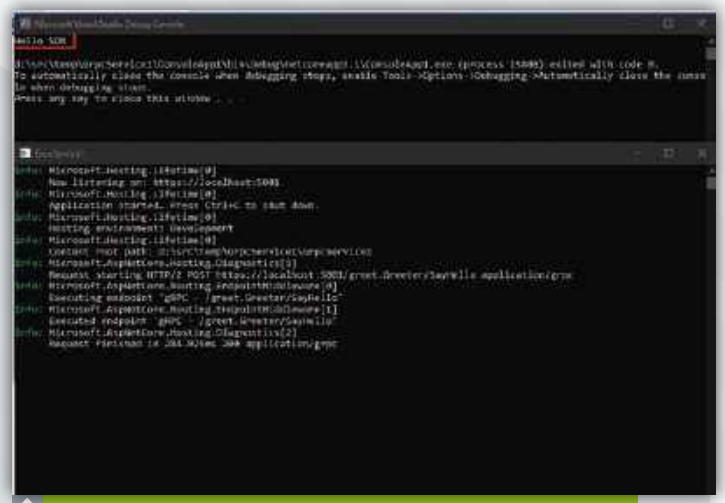


FIGURE 32: UITVOER VAN DE SERVICE

Bij het uitvoeren van je code, zul je twee console windows zien. In de Debug Console kun je de uitvoer van de client zien (Figuur 32). In de client kun je zien dat de code inderdaad werkt.

UITGEBREIDERE MOGELIJKHEDEN

In dit artikel hebben we tot nu toe eigenlijk alleen maar gewerkt met de beperkte service die standaard door Visual Studio gegenereerd wordt. Voor een sessie over gRPC tijdens DotNED Saturday 2020 heb ik een uitgebreider voorbeeld uitgewerkt van een gRPC service voor het domein van een webwinkel. We zullen nu wat uitgebreidere mogelijkheden van gRPC behandelen, zodat je ook een goede indruk krijgt wat je allemaal met gRPC kunt doen.

```
message AddInvoiceResponse {
}
```

FIGURE 33: LEEG BERICHT

Als een methode geen parameter of een return type nodig heeft, dan kun je ervoor kiezen om een leeg bericht (Figuur 33) of void in de vorm van Empty terug te geven (Figuur 34).

```
rpc LogMessage(HelloRequest) returns (google.protobuf.Empty);
```

FIGURE 34: EMPTY BERICHT

Om Empty te kunnen gebruiken, moet je deze wel importeren in je protobuf-bestand (Figuur 35) aan de bovenkant van het bestand.

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

FIGURE 35: IMPORT EMPTY

Binnen berichten kan ik ook gebruik maken enumeraties (Figuur 36)

Bij een enumeratie moet ik ook verplicht de volgorde van de opties vastleggen.

```
enum CustomerType {
    Error = 0;
    Private = 1;
    Corporate = 2;
}
```

FIGURE 36: INSTALLEREN CERTIFICAAT

Deze enumeratie kan dan weer direct gebruikt worden in een berichtdefinitie (Figuur 37)

```
message CreateOrder {
    int32 customer = 1;
    CustomerType type = 2;
    google.protobuf.Timestamp orderedAt = 3;
    repeated CreateOrderLine orderLines = 4;
}
```

FIGURE 37: UITGEBREIDER BERICHT

In deze berichtdefinitie kun je ook zien hoe je met collecties binnen protobuf om moet gaan. Hiervoor is het keyword repeated. Binnen C# wordt dit vertaald naar een speciale collectie van Google van het type RepeatedField<T>. De interface van deze klasse wijkt soms een beetje af van wat je standaard van de .NET collecties gewend bent, maar functioneel wordt hetzelfde ondersteund. Aan deze collectie kun je bijvoorbeeld CreateOrderLine-objecten toevoegen (Figuur 38).

```
var orderLine1 = new CreateOrderLine()
{
    ProductId = 100,
    Price = 10,
    Quantity = 2
};

var orderLine2 = new CreateOrderLine()
{
    ProductId = 200,
    Price = 20,
    Quantity = 3
};

var order = new CreateOrder()
{
    Customer = response.Id,
    OrderedAt = Timestamp.FromDateTime(DateTime.UtcNow),
};

order.OrderLines.Add(orderLine1);
order.OrderLines.Add(orderLine2);
```

FIGURE 38: TOEVOEGEN ORDERLINES

In de definitie van CreateOrder heb ik ook gebruik gemaakt van de mogelijkheid van protobuf om TimeStamps te gebruiken. Ook dit is een speciaal type voor gRPC. In Figuur 38 kun je zien dat je van een .NET DateTime een TimeStamp kunt maken door gebruik te maken van de methode FromDateTime. Dit is een methode van het TimeStamp object en niet zoals je misschien zou verwachten een extension method. Voor de conversie naar DateTime bestaat een methode binnen de TimeStamp klasse “ToDateTime”.
Om gebruik te kunnen maken van TimeStamp, moet je deze wel weer importeren, net zoals Empty (Figuur 39).

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

FIGURE 39: IMPORT TIMESTAMP

Protobuf ondersteunt standaard nog veel meer typen. De complete lijst met typen kun je vinden op: https://bit.ly/2QkeV8J.
Een belangrijke feature van gRPC is de eenvoudige ondersteuning van streaming. Streaming wordt ondersteund vanuit de client, vanuit de server of bidirectioneel. In Figuur 40 kun je een voorbeeld zien van een client side stream.

```
service SynchronisationService {
    rpc AddTemperatureReading(stream AddTemperatureReadingRequest) returns (google.protobuf.Empty);
}

message AddTemperatureReadingRequest {
    google.protobuf.Timestamp RegisteredAt = 1;
    double temperature = 2;
}
```

FIGURE 40: CLIENT SIDE STREAM

Het enige dat ik aan de protobuf kant aan hoef te geven is het keyword “stream”. Dit geldt voor ook voor de server stream. Als ik hier gebruik van zou willen maken, dan hoef ik alleen het keyword stream voor het returntype op te nemen. Voor bidirectionele streaming hoef ik dus alleen twee keer het keyword stream op te nemen.
De code voor de serverkant krijgt bij client streaming een IAsyncStreamReader binnen (Figuur 41)

```
public override async Task<Empty> AddTemperatureReading(IAsyncStreamReader<AddTemperatureReadingRequest> requestStream,
    ServerCallContext context)
{
    var callingClient = context.RequestHeaders.SingleOrDefault(header => header.Key.ToLower() == "clientid");

    await foreach (var reading in requestStream.ReadAllAsync())
    {
        Console.WriteLine($"Received @ {reading.RegisteredAt.ToDateTime()} temp {reading.Temperature} from {callingClient.Value}");
    }

    return new Empty();
}
```

FIGURE 41: CLIENT STREAM SERVER KANT

De requestStream die van de client komt, kan eenvoudig met de C# 8 code constructie await foreach verwerkt worden. De lus in deze methode blijft actief totdat de client aan de server doorgeeft dat de stream ten einde is.

Aan de client kant moet ik voor streaming nog wel wat speciale code opnemen (Figuur 42). De channel en de client kan ik aanmaken zoals we dat van te voren ook gedaan hebben. De call naar de methode moet je opvangen in een variabele. Je roept namelijk nu niet direct de methode aan, maar je krijgt een object van AsyncClientStreamingCall terug als je schijnbaar de methode aanroept. Dit object kun je nu weer gebruiken om via de RequestStream een nieuwe schrijfactie uit te voeren via een call naar WriteAsync. Aan het eind van de stream moet je dit ook expliciet aangeven door CompleteAsync aan te roepen. In ons voorbeeld weet de server dan dat hij geen nieuwe berichten meer hoeft te verwachten. Het werken met streaming is dus vrij simpel te implementeren.
Een gRPC service kan ook gebruik maken van een andere gRPC service om zo zijn werk uit te voeren. Hiervoor moet je net als in de console client uit dit artikel een service reference maken naar het

protobuf-bestand, maar je moet ook nog een extra stap uitvoeren. Om de client van de service netjes mee te laten draaien in de ..NET Core code, moet je de client toevoegen als GrpcClient aan de services collection in de ConfigureServices methode van de Startup klasse (Figuur 43).

```
var channel = GrpcChannel.ForAddress("https://localhost:5005");

var client = new SynchronisationServiceClient(channel);

var metaData = new Metadata();
metaData.Add("ClientID", clientId.ToString());

using (var call = client.AddTemperatureReading(metaData))
{
    var timer = new System.Timers.Timer();
    var randomizer = new Random(Int32.MaxValue);
    timer.AutoReset = true;
    timer.Elapsed += (o, args) =>
    {
        var request = new AddTemperatureReadingRequest()
        {
            RegisteredAt = Timestamp.FromDateTime(DateTime.UtcNow),
            Temperature = 20.00 * randomizer.NextDouble()

        };

        call.RequestStream.WriteAsync(request);
    };

    timer.Start();

    Console.WriteLine("Press <enter> to stop sending data");
    var shouldEnd = Console.ReadLine();

    timer.Stop();

    await call.RequestStream.CompleteAsync();
}
```

FIGURE 42: CLIENT STREAM CLIENT KANT

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
    services.AddGrpcClient<FinancialServiceClient>(options =>
        options.Address = new Uri("https://localhost:5003"));
}
```

FIGURE 43: ADDGRPCCLIENT

De FinancialServiceClient kan nu gewoon via dependency injection gebruikt worden in de servicecode (Figuur 44).
Met gRPC is dus heel veel mogelijk om communicatie tussen systemen en binnen systemen mogelijk te maken met relatief weinig eigen code.

```
public ShopService(ILogger<ShopService> logger, FinancialServiceClient financialClient)
{
    _logger = logger;
    _financialClient = financialClient;
}
```

FIGURE 44: GEBRUIK VAN DE SERVICE IN EEN ANDERE SERVICE

WAAROM KIEZEN VOOR GRPC?

Waarom zou ik nu kiezen voor gRPC in plaats van het meer gebruikte WebAPI. Ten eerste gRPC kent een vast contract. Met een WebAPI kan ik dit ook bereiken met OpenAPI, maar deze standaard wordt nog niet heel uitgebreid gebruikt. Ik heb heel veel koppelingen tussen systemen geschreven en het hebben van een goed contract tussen systemen helpt enorm tijdens de softwareontwikkeling.
Als je gebruik maakt van niet JavaScript talen, dan kost het omzetten van en naar JSON best wel wat tijd. Met een binair protocol als gRPC heb je hier geen last van. De berichten kunnen dus sneller uit worden gewisseld. Een bijkomend voordeel is dat de berichten binair van aard zijn en dus veel kleiner dan JSON. Als je te maken hebt met beperkte bandbreedte of moet betalen op basis van gebruik, kan dit heel belangrijk zijn. Ook bij communicatie tussen containers binnen je server is dit prettig, want daar geldt toch altijd nog minder is beter.
Een laatste voordeel vind ik persoonlijk dat er een proxy gegenereerd kan worden. Bij gebruik van een WebAPI moet ik die altijd zelf maken. Dit is niet zo'n heel spannende code om te schrijven en juist daardoor kom ik in de praktijk vaak fouten in dit soort code tegen.
Er zit op dit moment nog wel een nadeel met betrekking tot hosting. gRPC leunt heel sterk op HTTP/2 en IIS en Azure AppService ondersteunen HTTP/2 op dit moment nog niet in voldoende mate. Microsoft werkt aan een oplossing hiervoor, maar voorlopig is die oplossing nog niet beschikbaar in productie. Bij gebruik van de gRPC-service binnen bijvoorbeeld een Kubernetes cluster heb je hier niet zozeer last van, maar als je de service ook beschikbaar wilt stellen voor bijvoorbeeld single page applicaties, is dit wel een uitdaging. Een haalbaar alternatief is om de gRPC-service als docker container te hosten met Kestrel als webserver. Kestrel ondersteunt wel goed alle noodzakelijke functionaliteit van HTTP/2 en Kestrel is voortaan ook echt wel geschikt om in productie te worden gebruikt.
Voor gebruik in een web omgeving is er nog een ander alternatief in de vorm van gRPC-Web. Deze oplossing heeft wel als nadeel dat niet alle functionaliteit ondersteund wordt, maar het maakt het wel mogelijk om vanuit een JavaScript en/of Blazor single page applicatie gebruik te maken van een gRPC-service. In een volgend artikel zal ik specifiek ingaan op het gebruik van gRPC-Web in combinatie met Blazor WebAssembly.

SAMENVATTING

In dit artikel heb ik de belangrijkste aspecten voor het bouwen van een gRPC service en client behandeld. Met gRPC kan ik op een simpele manier een contract-first api maken die bovendien ook nog sneller is dan WebAPI en minder netwerkverkeer nodig heeft. Een bijkomend voordeel is dat gRPC echt cross-platform en ontwikkeltaalonafhankelijk werkt. Ik heb dus de vrijheid om zelf een keuze te maken voor de ontwikkeltaal die ik op de server en op de client wil gebruiken. Op dit moment is gRPC heel erg goed bruikbaar binnen de muren van je serverpark. Voor gebruik door webapplicaties zijn er nog wel wat uitdagingen, maar de belangrijkste daarvan kunnen weg worden genomen door gebruik te maken van gRPC-Web.
Op mijn github (https://github.com/JohanSmarius) heb ik een aantal gRPC demo projecten staan waarin de onderwerpen uit dit artikel plus nog wat meer features getoond worden. 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 / https://www.johansmarius.com

Menno Brouwer

Ik Testen, Wij Kwaliteit

Hulpvraag: zorg jij voor de kwaliteit in jouw werk, dan doe ik dat voor het mijne en wij samen voor het geheel. Herken je zulke nieuwsitems?

- Gemeente keert miljoenen te veel uit
- Storing met betaalautomaten banken
- Lekkende persoonsgegevens
- Forse budgetoverschrijdingen op ICT-projecten

Heb je zelf wel eens een stuk code geschreven, dat bij livegang (of eerder) niet het gewenste effect bleek te hebben? Was dit omdat je het niet goed gebouwd had, of dat het toch achteraf niet duidelijk genoeg was wat de vraag was…?

Wist je dat 75% van de fouten komen doordat de foute oplossing goed geïmplementeerd is?!!

De wet van Böhm geeft duidelijk weer dat hoe later je een (software) fout vindt in je proces, hoe meer het kost om de fout te herstellen. Om maar niet te spreken over imago schade, die je oploopt wanneer een systeem falen het nieuws haalt.

Is dit te allen tijde te voorkomen? Nee, ik denk het niet, maar in veel gevallen wel. Gestructureerd testen biedt doorgaans een goed beeld van de kwaliteit van de opgeleverde software. Is aan het eind nog even testen de oplossing voor alle potentiële problemen? Nee, ook niet!
Vaak is het zo, dat de ingeplande testtijd als sneeuw voor de zon verdwijnt door uitlopende ontwikkeltijd, en het niet navenant mee verschuiven van de go-live datum. De acceptatie tests zitten vaak in de kreukelzone van het ontwikkeltraject. Vandaar dat we risk-based testen. De belangrijkste tests eerst… Zelf ben ik acceptatietester en heb ik vaak gesprekken met belanghebbenden over wat we nog kunnen doen en welke risico's we dan niet hebben afgedekt.
Maar stel nu dat het gestructureerd acceptatie testen met voldoende

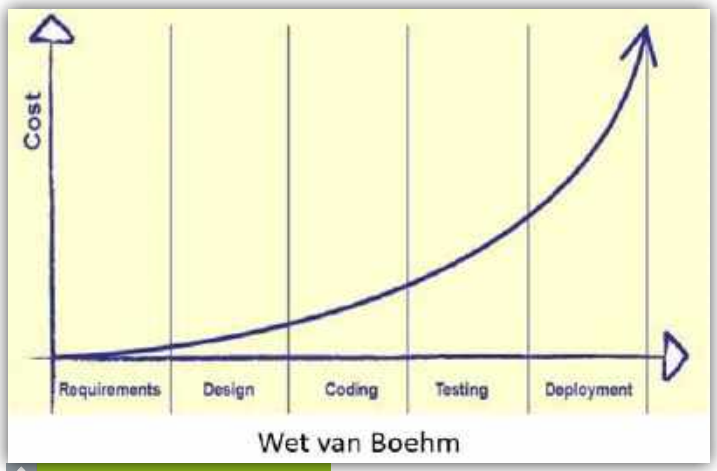


FIGURE 1

Ik Testen, Wij Kwaliteit

tijd een acceptabele dekking heeft plaatsgevonden, zijn we dan zeker van een goede kwaliteit? Op hoofdlijnen wel, wanneer alle technische tests zijn uitgevoerd. En dit is niet altijd het geval. Wanneer dit niet is gebeurd moeten we alle mogelijke varianten van het systeem met elkaar te testen, vanaf de voorkant van het systeem, dat kost veel tijd en is zelfs vaak onmogelijk.

WAT EN HOE DAN WEL?

Om een kwalitatief goed product neer te zetten hebben we elkaar nodig. Het klinkt evident, maar we willen de goede producten goed en tijdig ontwikkelen. Onder ontwikkelen versta ik het hele kwaliteitstraject van opstellen requirements, via bouwen en testen tot en met livegang. In dit simpele zinnetje zitten drie elementen in die van belang zijn voor een tevreden klant.

1 **Goede producten**
Zorg dat je vooraf helder hebt welke functionaliteit er verwacht wordt. Dat is een actieve fase waarin je scherp moet weten wat er gevraagd wordt en moet je met jouw expertise, beoordelen waar de moeilijkheden en mogelijkheden zitten, om zodoende een goede verwachting te scheppen.

2 **Goed ontwikkelen**
Zodra helder is wat de gevraagde functionaliteit is, kan het bouwen en de testvoorbereiding starten. Vroege feedback helpt om het juiste te bouwen en continue te verifiëren dat we de juiste dingen doen.

3 **Tijdig ontwikkelen**
Vroege feedback en onderhanden werk afmaken zorgen ervoor dat er het meest efficiënt wordt opgeleverd binnen tijd. Loopt er dan niets uit? Ja, vast wel, maar dan is er ook al heel wat wel af.

WELKE ACTIVITEITEN KUNNEN WE SAMEN INZETTEN OM DIT VOOR ELKAAR TE KRIJGEN.

Allereerst beginnen we aan de voorkant met het helder krijgen wat gevraagd wordt. Alvorens 1 regel code te schrijven moeten we weten wat er gevraagd wordt. De wat en waarom moet helder zijn voor zowel de vragende als de leverende partij. Aannames moeten uitgesloten worden. Het zou mooi zijn wanneer je in overleg met de opdrachtgever vast kunt stellen wanneer een item acceptabel. Wanneer je acceptatiecriteria in testscenario's kunt vatten, en daarmee volledig bent, helpt dit het hele ontwikkeltraject in doorlooptijd. De discussies worden dan gevoerd in de startfase en niet later in het traject (zie weer Wet van Böhm).

EARLY FEEDBACK

Om op het juiste pad te blijven is mijn motto om continu feedback te vragen op je werk of zelfs je ideeën. Een niet gecheckt idee, is natuurlijk gewoon een aanname. En aannames zijn in ons vak vaak de reden van bevindingen. Het goed bouwen van de verkeerde software. Zie eerdere quote.

In elke fase van het proces kunnen we nagaan of onze ideeën, prototypes of code overeenkomt met de functionele of technische verwachting. In elke fase zijn methoden voorhanden om elkaar hierop te challengen.

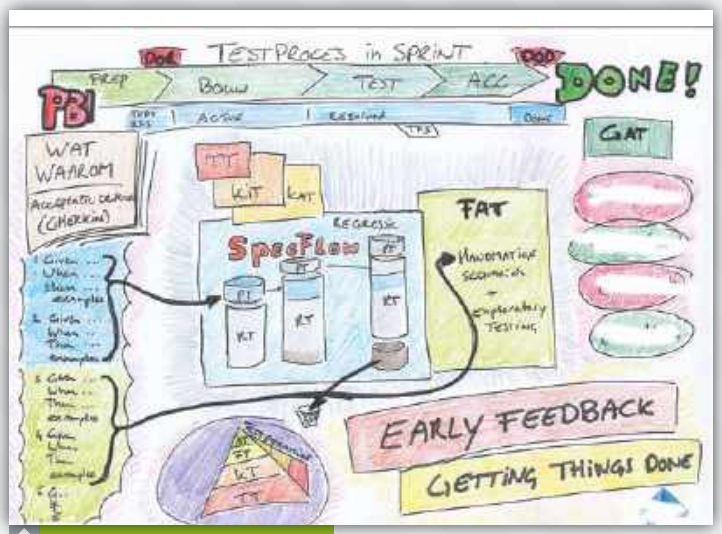


FIGURE 2

Zodra we gestart zijn een functionaliteit (item) op te pakken, is het zaak deze af te ronden. Afmaken…

GETTING THINGS DONE

Collectief een item op pakken en afronden. Afronden zorgt voor rust en ruimte. Het alternatief is namelijk dat er aan het einde nog vele items afgerond moeten worden. Ontwikkelen, testen, fixen en hertesten.
Zodra de software testbaar is, is het zaak zo snel mogelijk de tests uit te voeren na een warme overdracht. Wanneer je als ontwikkelaar feedback krijgt op software die je zojuist hebt gebouwd, dan kun je dat waarschijnlijk vrij snel herstellen, omdat je nog in de code zit. Heb je inmiddels alweer drie items gebouwd en komt er een bevinding op de code van vorige week (of eerder), dan kost het herstellen veel meer tijd. Inmiddels ben je niet meer daarmee bezig en de gemaakte keuzes blijken vaak niet zo helder en evident. De extra fix tijd kan oplopen tot meer dan 30 keer de tijd om het gelijk te fixen.
Dus door iets direct af te ronden wordt de doorlooptijd verkort en zijn er dus altijd items af om in productie te nemen. Zijn er externe factoren welke potentieel de voortgang hinderen? Pak de gerelateerde items dan nog niet op.
Hoe kunnen we elkaar helpen? Vanuit test en zeker kwaliteitsperspectief wil ik de test pyramide benoemen.

TEST PYRAMIDE

Idealiter is het testproces ingericht volgens de test pyramide. Dus de unit en integratietests zijn geautomatiseerd en voldoende dekkend. De E2E tests zijn ook geautomatiseerd en een aantal tests voeren we nog handmatig uit ter verificatie.

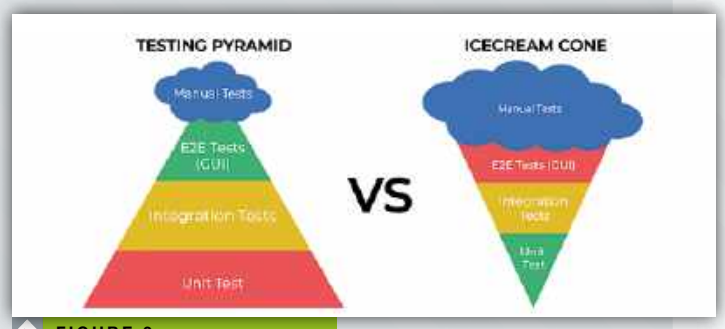


FIGURE 3

Vaker is het echter zo, dat het testproces meer weg heeft van een ijshoorn. Alle (deel)functionaliteiten en integraties moeten handmatig via de GUI getest worden naast de reguliere functionele tests. Dit levert een hoop extra en onnodig werk op. Ook omdat na een afwijking van de verwachting een ingewikkelde analyse volgt naar de oorzaak van de afwijking.
De technische tests (Unit en integratietests) moeten geautomatiseerd uitgevoerd worden en in de build proces worden opgenomen. Ik focus me nu vanaf hierop functioneel testen.
Wanneer items ge- of verbouwd worden, voeren we daarop een progressietest uit.

PROGRESSIETEST

De focus van de progressietest is de nieuwe of vernieuwde functionaliteit.
Testscenario's zijn per item idealiter vastgesteld in de opstartfase. Testtechnieken helpen bij het gestructureerd uitwerken van de scenario's. Naast alle gewijzigde functionaliteit is er meestal meer software welke ogenschijnlijk niet geraakt is. Ook van dit deel moet de ongewijzigde werking elke oplevering worden aangetoond. Dit noemen we regressietesten.

REGRESSIETEST

Zodra de progressietests zijn uitgevoerd en een item done is, worden deze tests potentieel onderdeel van de regressie testset. Deze testset stelt vast dat werking van de belangrijkste (risk based) onderdelen van de niet gewijzigde functionaliteiten niet veranderd is en kan geautomatiseerd uitgevoerd worden.
Idealiter wordt een progressietest geautomatiseerd opgeleverd en na validatie toegevoegd aan de regressie testset.

GEAUTOMATISEERD OF HANDMATIG TESTEN?

Wat in de volksmond geautomatiseerd testen wordt genoemd is eigenlijk, geautomatiseerd checken. Een gelimiteerd aantal checks worden automatisch uitgevoerd. Namelijk exact wat je het systeem verteld wat er gecheckt moet worden. Een tester ziet altijd meer dan dat er gecheckt wordt. Mensen nemen meer waar, zoals bijvoorbeeld samenhang en kleurgebruik.
Beide systemen hebben op functioneel niveau hun beperkingen. Handmatig regressietesten kost veel tijd en is steeds hetzelfde, dus heeft het risico in zich dat het met minder focus of zelfs niet uitgevoerd wordt. Bovendien is het wenselijker dat de tester zich richt op het ontwikkeltraject en dus progressietests uitvoert.
Automatische checks kunnen in kortere tijd veel meer checks doen, zeker wanneer je je test slim inricht. Non gui testen bijvoorbeeld. Echter meer dan de gelimiteerde checks worden niet uitgevoerd. Keer op keer… Ontstaan er geen fouten op niet gecheckte elementen?

Hoe dan?

Zorg voor een goed dekkende geautomatiseerde functionele regressietest en vul die per oplevering aan met een gefocuste handmatige test. Je kunt ervoor kiezen om met de testers elke oplevering op een ander gebied te focussen, zodat ook daar na verloop van tijd een grote dekking ontstaat.

NU NOG EVEN OVER HET TEAM

Om goed werk te kunnen leveren met het team moet iedereen kritisch zijn. En dat ook van elkaar accepteren. Er zijn altijd verschillende persoonlijkheden en kwaliteiten in je team. De een schreeuwt zich een ongeluk, de ander houdt zich dan maar gedeisd. De een ziet kansen, de andere alleen de bedreigingen. De een heeft verstand van interfaces, de ander van de back-end of front-end. De een heeft veel business proces kennis, de ander kan de boel aantonen. Per slot is iedereen op zijn persoonlijke kwaliteiten beoordeeld en aangenomen. Bij elkaar heb je dus een prachtige mix van kwaliteiten, die, wanneer iedereen zich kan uiten, alles in huis heeft om succesvol producten te leveren. De kunst is dus om een efficiënte manier te vinden om ieders mening op waarde te schatten en daarop als team te acteren. En wel zo vroeg mogelijk.

CONCLUSIE

Met je team creëer je mooie producten waar je trots op wil zijn. Dat kan alleen maar samen. Testen is een uitvoerend onderdeel van het collectieve kwaliteitsdenken.

DE TESTER KAN GEEN KWALITEIT IN HET SYSTEEM TESTEN

Neem je verantwoordelijkheid in dit samenspel. Onduidelijkheden komen voor en dienen zo snel mogelijk uitgesproken te worden. Help elkaar in dit mooie proces.
In onze bedrijfstak hebben we te maken met faalkosten tussen de 60 en 70% in tegenstelling tot het fabriceren van bijvoorbeeld tafels, waar faalkosten zich beperken tot zo'n 6%. Dat moeten we samen beter gaan doen.

IK ZORG VOOR ACCEPTATIETESTEN, WIJ SAMEN VOOR DE KWALITEIT

Dan zorgen we er samen voor dat onze projecten niet het nieuws halen.

MENNO BROUWER

Ik ben testspecialist en agilist in hart en nieren. Ik test sinds 2001 en sinds 2012 ben ik betrokken in agile scrum trajecten.
Mijn passie is om samen tot het optimale resultaat te komen.
Volg me op LinkedIn:
https://www.linkedin.com/in/menno-brouwer-00a1261/

Dennis van der Stelt

Weronika Łabaj

Wat Starbucks ons kan leren

OVER SCHAALBAARHEID VAN SOFTWARE

Meer en meer raken we in Nederland bekend met Starbucks. De eerste winkel werd geopend op Schiphol, maar tegenwoordig vind je ze veel vaker terug in steden en langs de weg. Starbucks heeft haar logistieke erg goed op orde, zelfs binnen de winkel. In 2004 publiceerde Gregor Hohpe zijn briljante artikel “Starbucks does not use Two-Phase Commit”* waarin hij uitlegt hoe Starbucks met consistency om gaat. Maar we kunnen nog veel meer leren van de populaire koffieketen.

Hoewel veel mensen schaalbare software willen bouwen, is het lastiger dan het op het eerste gezicht lijkt. Terwijl we de componenten van een systeem bouwen, kunnen we wellicht vergeten dat niet alle dingen even belangrijk zijn, dezelfde middelen nodig hebben en synchroon dienen te worden uitgevoerd in een vooraf gedefinieerde volgorde.

Veelal hoeft dat niet zo te zijn - althans niet in schaalbare systemen, en zeker niet bij Starbucks.

HOE KOFFIE TE ZETTEN

Koffie zetten bij Starbucks is een proces in vier stappen. Ten eerste staan klanten in de rij (een queue) aan het loket en plaatsen hun bestellingen volgens de first-in-first-served-regel. Ten tweede neemt de medewerker (barista) een bestelling van de klant aan en accepteert deze de betaling. Ten derde begint het bereiden van de drank. Ten vierde, als de bestelling klaar is, wordt het drankje op de balie gezet en wordt de naam van de klant geroepen.

FIGURE 1

Hoewel dit misschien als een redelijk model klinkt, kan het snel tot lange rijen leiden. Het is onmogelijk voor één persoon om meer dan één actie tegelijk uit te voeren, dus klanten gaan in de rij staan terwijl de barista elke bestelling opeenvolgend afhandelt. Als Starbucks meer klanten wil bedienen, moet er worden opgeschaald. Laten we eens kijken hoe ze dat kan doen.

BARISTA’S OPSCHALEN

Een manier waarop Starbucks kan opschalen, is door super-barista’s in te huren - zeer getalenteerde, snelwerkende, slimme mensen. Ze zouden zwaar moeten investeren in hun ontwikkeling, elk aspect van hun werk moeten optimaliseren en hun efficiëntie voortdurend moeten verbeteren. In software zou een dergelijke benadering scale-up (verticale schaalvergroting) worden genoemd.

Het probleem met deze strategie is dat er een limiet is aan hoe snel (en hoe lang) iemand kan werken. Op een gegeven moment kan zelfs de super-barista niet aan de vraag voldoen. Wanneer dit gebeurt, verlaten klanten gefrustreerd de winkel en komen ze mogelijk niet meer terug.

Evenzo is er een limiet aan de mate waarin we onze software kunnen optimaliseren als alles opeenvolgend wordt uitgevoerd. We kunnen gewoon geen 200 GHz CPU kopen. Zelfs de zwaarste CPU’s zijn multi-core, waarbij elke core op niet meer dan 3–4 GHz klokt.

Een andere manier waarop Starbucks kan opschalen, is door het werk zo te organiseren dat er meer normale werknemers kunnen worden toegevoegd, wat de essentie is van concurrent processing in software. Nadat de ene barista een bestelling heeft opgenomen, kan een andere beginnen met het voorbereiden ervan. De eerste barista kan dan een andere bestelling aannemen terwijl de eerste bestelling parallel wordt voorbereid.

Je zou kunnen denken dat het beste idee zou zijn om concurrent processing pas te overwegen nadat de vraag een bepaald niveau heeft bereikt. Helaas is het niet zo eenvoudig. Er is geen magische schakelaar waarmee we dit kunnen inschakelen op het moment dat we het nodig hebben. We moeten ons hier tevoren op voorbereiden.

Starbucks weet dat. Wanneer een nieuwe winkel wordt geopend, zelfs als ze maar één werknemer per ploeg hebben, is alles wat nodig is voor concurrent processing vanaf de eerste dag aanwezig. Ze kunnen op elk moment meer mensen toevoegen.

Wijze les: we kunnen concurrent processing niet gemakkelijk toepassen als we ons systeem niet zo bouwen dat dit vanaf het begin wordt ondersteund.

Laten we eens kijken hoe Starbucks dit bereikt.

HET BEGINT MET BERICHTEN

Als je ooit koffie bij Starbucks hebt besteld, heb je misschien kleine vierkantjes met symbolen op de beker gezien. Deze symbolen zijn een soort afkorting die door de barista’s wordt gebruikt om snel de drank voor de bestelling te identificeren, evenals eventuele extra’s (bijv. caramel, schuim, enz.).

** https://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html*

Wat Starbucks ons kan leren

OVER SCHAALBAARHEID VAN SOFTWARE

De beker, of het bericht, is essentieel voor de communicatie tussen medewerkers. De beker geeft bij de barista aan dat een bestelling moet worden gemaakt en de symbolen die erop zijn geschreven, geven details over wat voor soort drank deze moet bereiden. Zelfs als het in de coffeeshop niet druk is en er maar één persoon is die de klanten bedient, zullen er nog steeds symbolen aan de beker toegevoegd worden.

FIGURE 2

Op het eerste gezicht lijkt dit misschien extra werk. Maar als er plots een grote groep klanten de winkel binnenkomt, kunnen de andere medewerkers direct bijspringen om te helpen. Zonder dat er extra communicatie-overhead nodig is, kunnen ze drankjes gaan maken op basis van de berichten.

Wijze les: plotselinge pieken zijn niet problematisch als we op elk moment eenvoudig meer werknemers kunnen toevoegen en het werk over deze kunnen verdelen. Het gebruik van berichten, ofwel messages, is een manier om dat te doen.

VERDEEL EN HEERS

Zoals eerder beschreven, kan het hele process to koffie zetten worden verzorgd door één enkele medewerker: een barista. Maar standaard bij Starbucks zal één medewerker (een kassamedewerker) bestellingen en betalingen aannemen en een andere (de barista) de drankjes bereiden.

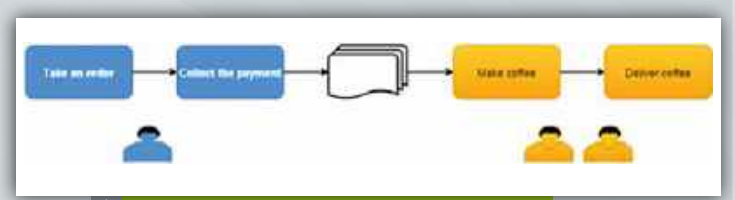


FIGURE 3

Meestal is het bereiden van koffie het meest langzame deel van het proces, daarom bereiden meerdere barista's drankjes als het druk is in de winkel. Vaak nemen ze bekers van dezelfde stapel en verdelen ze het werk gelijkmatig. Dit is een voorbeeld van het patroon van competing consumers.

Er kunnen echter scenario's zijn waarin deze aanpak problemen oplevert. Stel dat er drie barista's zijn die werken met één koffiemachine en één Frappuccino machine. Drie klanten bestellen koffie en de volgende klant bestelt een Frappuccino. De persoon die bestellingen opneemt, plaatst vier bekers in de rij en tekent de juiste symbolen op elk. Elke barista pakt een beker om koffie te zetten. De eerste begint met het maken van het drankje en de andere twee staan nu geblokkeerd te wachten op de koffiemachine.

We kunnen deze "strijd" bij de koffiemachine vermijden door het werk te verdelen. Een manier om dit te doen, is door berichten op te splitsen in meer fijnmazige typen, zodat deze anders kunnen worden behandeld. We hebben bijvoorbeeld gezien hoe Starbucks de beker gebruikt als een bericht om aan te geven dat er een drankje moet worden bereid. Maar het systeem maakt ook onderscheid tussen warme en koude dranken: warme dranken worden geserveerd in papieren bekers en koude in plastic bekers. Als we drie bestellingen krijgen voor warme koffie gevolgd door één voor een Frappuccino, hebben we nu drie papieren bekers en een plastic bekertje op twee verschillende stapels. De eerste barista pakt de kartonnen beker van de eerste stapel en begint de drank klaar te maken. De tweede barista, die ziet dat de koffiemachine bezet is, pakt de plastic beker van de tweede stapel en gebruikt in plaats daarvan de Frappuccino machine.

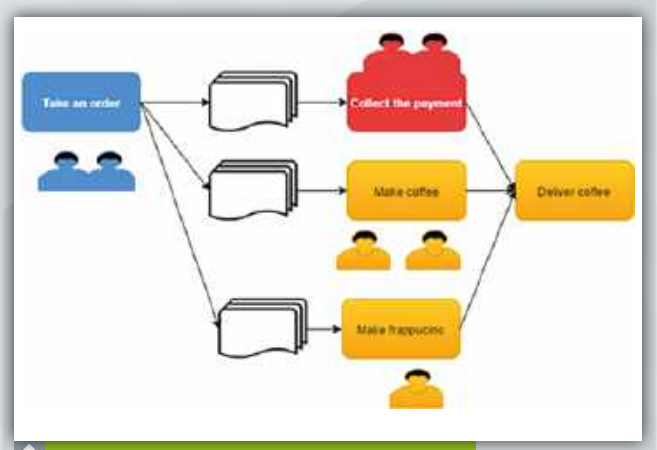


FIGURE 4

Deze werkverdeling, waarbij barista's taken verdelen en parallel werken, wordt partitionering genoemd.

Wijze les: Het blijkt dat partitionering een cruciaal element is van een effectieve strategie om te schalen. Niet al het werk dient hetzelfde te schalen. Kleine taken die snel worden uitgevoerd, kunnen door één werknemer worden uitgevoerd, terwijl meerdere werknemers de meer veeleisende, langzamere taken voor hun rekening nemen. Door partitionering te gebruiken, kunnen we elke activiteit onafhankelijk schalen.

NIET AL HET WERK IS EVEN BELANGRIJK

Een van de dingen die Starbucks succesvol maakt, is dat ze hun personeel hebben getraind in het belang van herkennen van de veelvuldig terugkerende klanten. Neem de man die elke ochtend binnenkomt om twee venti americanos en twee grande lattes te halen voor zijn team. Of de vrouw die elke woensdag een caramel macchiato bestelt en dan een uurtje in de winkel blijft om haar boek te lezen. Als een barista de "karamel-macchiato-vrouw" woensdag de winkel ziet binnenkomen, beginnen ze met het bereiden van haar favoriete drankje nog voordat ze naar de toonbank komt.

De klant krijgt een aangename verrassing als ze niet hoeft te zeggen wat ze wil. De kassamedewerker kent haar gebruikelijke drankje al, dus deze vraagt haar alleen hoe het met haar gaat en neemt de betaling aan. Voordat de betaling is afgerond, staat haar koffie al op haar te wachten aan de balie.

Het zal je misschien verbazen hoe hoog een percentage van de klanten van Starbucks reguliere klanten zijn. Deze de best mogelijke ervaring bieden, heeft hoge prioriteit. Vaak krijgen ze hun drankjes sneller dan andere klanten. Hierdoor voelen ze zich belangrijk en worden ze aangemoedigd om terug te komen, waardoor hun waarde voor het bedrijf toeneemt.

FIGURE 5

Wijze les: sommige taken zijn belangrijker dan andere. Door standaardactiviteiten te organiseren in herbruikbare, onafhankelijke bouwstenen, kunnen we het proces eenvoudig aanpassen om superieure service te bieden voor de meer waardevolle taken wanneer dat nodig is.

NIET ALLE FOUTEN MOET JE WILLEN VOORKOMEN

In alle bovenstaande voorbeelden moesten Starbucks-medewerkers controleren of klanten hebben betaald voordat ze hun koffie kregen. Om ervoor te zorgen dat dit gebeurt, kunnen barista 's klanten vragen hun bonnetjes te tonen voordat ze een drankje overhandigen. Maar zo werkt het niet.

Wat Starbucks ontdekte, is dat maar heel weinig mensen koffie meenemen zonder te betalen. Uit hun analyse bleek dat het winstgevender is om barista's gefocust te houden op het uitvoeren van bestellingen in plaats van af en toe een onjuist meegenomen koffie te voorkomen. Als iemand toevallig de door jou bestelde koffie meeneemt (wat meestal alleen per ongeluk gebeurt), zal de barista een nieuwe voor je bereiden, zonder vragen te stellen.

Wijze les: om schaalbare systemen te bouwen, dienen we het idee te omarmen dat sommige mislukkingen onvermijdelijk zijn. Het kost te veel om te proberen fouten volledig te voorkomen. In plaats daarvan zouden we ervoor moeten zorgen dat we problemen snel kunnen detecteren en ze kunnen compenseren zodra ze zich voordoen.

CONCLUSIE

Wat leek op een eenvoudig proces in vier stappen voor het maken van koffie, is uitgegroeid tot een interessant bedrijfsproces. Wat op het eerste gezicht uitzonderlijk en zeldzaam leek, bleek een essentieel aspect van het bedrijf te zijn.

Zaken zoals plotselinge pieken in de vraag of storingen kunnen meerdere keren per dag voorkomen. Om een systeem te ontwerpen dat deze goed afhandelt, moeten algemene aannames in twijfel worden getrokken. Vaak neemt een eerste architectuur of design dergelijke zorgen niet weg. Er zijn vaak nog veel meer uitzonderlijke situaties waarmee rekening moet worden gehouden. Het annuleren van bestellingen is bijvoorbeeld op zich al een interessant probleem.

Zoals het voorbeeld van Starbucks laat zien, zou het volgen van een eenvoudig aanpak ons bedrijf verhinderen uit te breiden om een groter aantal klanten te bedienen. Ons niveau om service te verlenen zou dalen naarmate we meer en meer klanten kregen, tot het punt waarop ze niet meer zouden komen. In plaats daarvan moeten we ons werk zo organiseren dat we aan de toenemende vraag kunnen voldoen. Bij het bouwen van schaalbare systemen gaat het uiteindelijk net zozeer om het herzien van onze bedrijfsprocessen als om technologie.

Dit artikel werd als eerste gepubliceerd op de Particular Software weblog en is geschreven door Weronika Łabaj en Dennis van der Stelt

ADDITIONELE BRONNEN:

- Horizontaal schalen met asynchroon messaging: http://bit.ly/sdn-async-messaging
- Competing Consumer pattern: http://bit.ly/sdn-competing-consumer
- Priority Queues heb je niet nodig http://bit.ly/sdn-priority-queues

DENNIS VAN DER STELT

Dennis van der Stelt innovative software architect with over 15 years of experience in development of distributed systems. Has a continuous drive to learn and improve knowledge in different architectural styles, including quality in software development. Highly motivated to share his knowledge via numerous articles, presentations and posts on his blog. If you want to exchange knowledge, ideas or have questions, don't hesitate to reach out for contact!
Specialties: Distributed Architectures • .NET Software Development • Code Quality

WERONICA ŁABAJ

Weronica is a passionate developer, equally interested in building the right product and in building it right. Greatly inspired by Software Craftsmanship movement she learned to appreciate the value of Agile practices, TDD, automated tests, writing clean, testable code and having a stable deployment process. She verifies their value in practice and she is looking forward to new experiences in this area. She continuously expands her knowledge outside of work by means of Coding Dojos, Code Retreats, workshops, conferences and community meetings. At work she strives towards finding win-win-win-win solutions in all circumstances – that all parties are satisfied.

Bart Kooijmans

Amazon en Google Cloud

VOOR DE AZURE-ONTWIKKELAAR

Ik ben een Microsoft .NET ontwikkelaar en werk met Microsoft Visual Studio en Microsoft MSSQL databases voor software op veelal Microsoft Windows systemen. Vijf jaar geleden was het dan ook een normale ontwikkeling in mijn carrièrepad om de cloud van Microsoft Azure te gaan gebruiken. Niet dat ik vergelijkend warenonderzoek had gedaan of een weloverwogen keuze had gemaakt. Nee, het leek vanuit mijn Microsoft tunnelvisie het juiste om te doen. En het is zeker geen slechte 'keuze'. Sterker nog, ik vind Microsoft Azure geweldig! Of je nu een kleine website, een stevige schaalbare pipeline of een serverless toepassing maakt, het werkt allemaal top en het onderhoud is minimaal.

De gang naar de cloud is voor mij al een zekerheidje. Van een gedeelde ftp-server en een paar servers in je kelder naar eigen servers in een datacenter in de buurt was al een hele stap. Maar de vervolgstap van je favoriete lokale datacenter naar de cloud brengt legio mogelijkheden: schaalbaarheid, redundantie, onderhoudsloos, wereldwijde dekking, etc.

Zoals je merkt, vind ik het leuk om mijn kennis en ervaring van de Azure cloud te delen met iedereen die het wil horen. Zodoende kwam ik dankzij een tip van mijn werkgever Team Rockstars IT uit bij de lokale hogeschool. Voor afstudeerders zochten zij een extern expert en ik had de opdracht met Azure gekregen. Ik vond het geweldig en bij de volgende lichting vroeg ik weer om 'iets met de cloud'. En ik kreeg... Amazon AWS. Oeps, daar had ik geen rekening mee gehouden. Dankzij mijn geweldige tunnelvisie wist ik er eigenlijk niks van af. Ja natuurlijk, ik weet dat ze de eerste waren met het verhuren van ongebruikte ruimte in hun datacenter. En het gerucht wil dat ze vaak eerder nieuwere functionaliteit dan Azure hadden. Maar toch... Azure is de beste... toch?

Ik was geïntrigeerd en ging op onderzoek uit: Amazon AWS had eigenlijk wel veel weg van mijn Azure ervaring: Azure Functions, Azure Container Instances, ARM templates... allemaal aanwezig. Als Microsoft ontwikkelaar had ik mijn Azure Software Developer certificaat al op zak. Door mijn onverwachte kennismaking met Amazon AWS ging ik me er steeds dieper in verdiepen en stelde mezelf de uitdaging om ook het Amazon AWS Software Developer certificaat te gaan behalen. Ook de lokale markt lijkt vaker Amazon AWS te vragen dan me eerder was opgevallen dus het zou niet slecht zijn er eens wat mee te gaan doen. Het examen heb ik vervolgens gehaald en om de horizon helemaal te verbreden heb ik toen de Google GCP Software Developer ook in mijn planning opgenomen.
Het leuke van dit alles is dat ik met een andere bril naar cloud ontwikkelen heb leren kijken. Door van meerdere partijen te leren, kun je beter uitzoomen en de praktijken als ontwikkelaar beter plaatsen. Deze ervaringen wil ik nu verder delen.

Laten we eens inzoomen op Amazon, Microsoft en Google. Amazon is begonnen met Amazon Web Services (AWS) in 2006. Het was destijds baanbrekend om een wereldwijde dienst aan te bieden voor datacenters. Daarbovenop kwamen kant-en-klare services waardoor je het onderliggende systeem niet hoeft te beheren.

FIGURE 1: AMSTERDAM AZURE DATACENTER

De eerste services waren File Storage, Virtual Machines en Message Queueing. Nu in 2020 telt AWS 77 availability zones verdeeld over 24 geographical regions en 175 services. We kunnen gerust zeggen dat het een daverend succes is geworden met een businessmodel dat miljarden euro's opbrengt. Verder is door het cloud model de softwareontwikkeling en architectuur een nieuw tijdperk in gegaan met ongekende mogelijkheden en een verschuiving van verantwoordelijkheden.

FIGURE 2: DE WERELDDEKKING VAN MICROSOFT AZURE

Microsoft volgde in 2010 met Azure en bood aanvankelijk File Storage, Virtual Machines, Message Queueing en Compute Service aan. Dit is nu uitgebreid naar een vergelijkbaar portfolio als Amazon AWS verdeeld over meer dan 60 geographical regions. Azure heeft met Microsoft het voordeel dat het met Windows, SQL Server,.NET en Visual Studio, vier populaire componenten van de ontwikkelaarsmarkt

in handen heeft. Door deze componenten naadloos met elkaar te integreren vormt het een ijzersterk marketingblok tegenover Amazon AWS.

FIGURE 3: DE WERELDDEKKING VAN AWS

Google doet ook al een tijdje mee en volgde in 2011 met Google Cloud Platform. In tegenstelling tot de Enterprise markt van Amazon en Microsoft, is Google vooral bekend van cloud diensten voor zaken als YouTube, GMail, Google Docs en Google Drive. Inmiddels weten zij echter perfect aan te sluiten bij de Enterprise markt en bieden zij ook een vergelijkbaar portfolio als AWS en Azure. Google Cloud Platform heeft uiteraard ook een wereldwijde dekking en bedient 24 regio's en innoveert stevig in de container markt met Kubernetes en multi cloud container oplossing Anthos.

Dit zijn de drie grootste spelers waarbij Google zelfs nog een buitenbeentje is. Alledrie worden ze in het beroemde Gartner Magic Quadrant genoemd als leaders. Amazon en Microsoft hebben samen

FIGURE 4: HET GARTNER MAGIC QUADRANT VOOR 'CLOUD INFRASTRUCTURE AS A SERVICE'

70% van het marktaandeel in handen, Google volgt op verre afstand met 4% marktaandeel. Gartner noemt ook nog de niche players IBM, Alibaba en Oracle.

AWS, Azure en Google spreken een zo breed mogelijk publiek aan: praktisch alles kun je er regelen in de cloud en tot de klanten behoren de grootste spelers van het internet. AWS schermt met high profile customers zoals Netflix en Formule1. Microsoft benadrukt dat 95% van de Fortune 500 bedrijven van Azure gebruik maken. Google doet er niet voor onder en spreekt van toonaangevende bedrijven als Nintendo en Toyota. Als ik verder kijk dan de marketing sheets en in mijn eigen omgeving rondkijk, dan zie ik als ontwikkelaar Amazon en Azure over de hele linie gebruikt worden en Google vooral in de markt van machine learning, big data en containers.

Qua grootte en voetafdruk doen ze dus niet voor elkaar onder. Hoe zit het dan met de domeinen en services onderling? Zoals eerder gezegd begonnen ze allemaal met Virtual Machines, File Storage en Message Queueing. Amazon heeft respectievelijk Elastic Compute Cloud (EC2), Simple Storage Services (S3) en Simple Queue Service (SQS) toegevoegd. Microsoft volgt met Virtual Machine, Blob Storage en Queue Storage (of Service Bus). Google kiest voor Compute Engine, Cloud Storage en Cloud Pub/Sub. Qua naamgeving zul je veel componenten terug zien komen in de services van elke cloud. Dit zijn de basics waar veel vervolgproducten op leunen.
Voor een beetje backend heb je al gauw een database nodig. Qua relation database bieden ze allemaal SQL Server, MySQL en PostgreSQL aan 'as a service'. De handigheid met de meeste platform as a service diensten is dat je geen onderhoud hebt maar wel de beschikking over bijvoorbeeld de database. Ook NoSQL is de afgelopen jaren aan een opmars bezig en wordt aangeboden als DynamoDB (AWS), CosmosDB (Azure) of Cloud Bigtable (GCP). Voor vluchtige memory cache kun je terecht bij een Redis oplossing (Elasticache (AWS), Azure Cache for Redis of Cloud Memorystore (GCP).
Het gehele compute spectrum breng je onder in een groot virtueel netwerk zodat ze onderling kunnen communiceren maar niet per se open staan voor iedereen van buitenaf. Azure noemt het een Virtual Network (Vnet) en AWS en GCP noemen het een Virtual Private Cloud.

FIGURE 5: EEN VOORBEELD ARCHITECTUUR VAN EEN AZURE SCALABLE WEB APP

Eigenlijk kun je het zo gek niet verzinnen of elke aanbieder heeft er een concurrerende service voor. Qua containers hebben ze alle drie een registry voor het hosten van je container images, een service voor het draaien van één instance van een docker image en een Kubernetes service voor het draaien van een hele compositie van docker images.

Op security-vlak beschikken zowel AWS, Azure als GCP over Identity & Access Management (IAM) en een Key Management Service (KMS) (Key Vault in Azure).

De web ontwikkelaar wordt op zijn wenken bediend met moderne API-gateways, traffic management en een web host as-a-service. Monitoring is diep geïntegreerd en serverless functions mogen uiteraard niet ontbreken. Lambda functions (AWS), Azure Functions (Azure) en Cloud Functions (GCP) zijn populaire serverless toepassingen waarin je met slechts een paar regels code een stuk functionaliteit kunt draaien op een enorm schaalbaar platform.

Wat ik als ontwikkelaar één van de mooiste ontwikkelingen vind is het platform voor het opslaan van je broncode en het build en release platform wat eraan vastzit. Toen ik kennis maakte met Azure DevOps als build en release platform was ik meteen verkocht. Geen onderhoud meer aan on-premise draaiende servers, maar gewoon een service met build machines in de cloud die door Microsoft worden onderhouden. Het verbaasde mij zelfs enigszins dat Amazon en Google dat precies ook aanbieden. Maar nu ik de clouds naast elkaar leg kijk ik er niet meer van op. Het is moeilijk te zoeken naar verschillen of wie er bovenuit steekt. Het punt blijft dat ze allemaal heel goed naar elkaar kijken en als de ene komt met een nieuw idee dan volgt de rest vrij snel daarna.

De eerste geluiden over verschillen die ik kreeg, hadden te maken met de portemonnee. “Amazon AWS is veel goedkoper met virtuele machines dan Azure” zei men. Nu is er niets zo moeilijk te vergelijken als schommelende prijsstellingen, maar ik ging me er eens in verdiepen. Een Windows of Linux VM met 2 virtuele CPU’s en 8 GB ram kost bij elk platform 8 á 16 cent per uur. Er zijn verschillende versies beschikbaar, van snelle CPU’s tot snelle disks, wat een exacte vergelijking eigenlijk onmogelijk maakt, maar ik zie geen schokkend grote prijsverschillen. De grootste VM die je kunt ‘krijgen’ bestaat al uit 100+ vcpu’s en terabytes aan ram geheugen. Als je dat soort machines nodig hebt of in het algemeen als je in dat soort orde van grootte zit dan is het normaal dat je onderhandelt over korting op prijzen. Dat maakt de prijsvergelijking nog onmogelijker!
Ik had graag nog een gefundeerde uitspraak gedaan over downtime/uptime en storingen, maar helaas is dat nergens openbaar makkelijk terug te vinden. Je ziet wel óf er storing is NU, maar een overzicht van een heel kalenderjaar met de behaalde service levels (SLA) of percentage uptime is informatie die de providers graag voor zich houden. Wat rest is opinies dat de ene beter is dan de andere, maar dat kan ik helaas niet staven.
Ten slotte is op de certificeringsladder ook nog wat gelijkheid te bekennen. Zowel Amazon, Microsoft als Google bieden certificeringen aan voor mensen met minder dan 6 maanden ervaring (fundamental/foundational). Daarna komen de paden voor 1 á 3 jaar ervaring voor ontwikkelaars, architecten en DevOps experts. Doordat de producten en best practices zoveel op elkaar lijken is het gemakkelijk om te switchen of meerdere examens te combineren. Ikzelf heb met minimale extra investering mijn Azure developer examen ook kunnen incasseren als een AWS developer examen.

CONCLUSIE:
Ik dacht dat Azure nog wel redelijk uniek was met de developer focus zoals met Azure Devops, maar guess what? Amazon en Google hebben dat ook. En bijvoorbeeld Windows of MSSQL licenties, dat hebben ze alle drie.

Zowel Amazon als Microsoft als Google doen niet voor elkaar onder als Cloud aanbieder. Het portfolio is nagenoeg hetzelfde, de prijzen zijn hetzelfde. Het onderscheid zit hem vooral in de niches, speciale toepassingen en koppelingen. Ik denk dat vooral de ontwikkelaar zijn favoriet kiest en daarom vind ik het zo belangrijk om het hele plaatje te overzien. Als .NET ontwikkelaar hoef je zeker niet blind naar Azure te kijken. Amazon en Google hebben ook interessante Clouds die ik van harte aanraad om eens te gaan bekijken.

BART KOOIJMANS

Bart Kooijmans has been working as a .NET developer for over 10 years and in those years he has seen many different facets of software development and the business. This experience combined with his solution architecture skills, help him in designing and implementing different projects and scenarios. He is now specialised as a backend developer in the Microsoft stack and works as solution architect and cloud specialist with Microsoft Azure, C#, Azure DevOps, etc. Moreover, he is always growing his knowledge of the larger Cloud world, i.e. AWS and Google Cloud. Sharing knowledge and experience is what he loves best.

“DE WEB ONTWIKKELAAR WORDT OP ZIJN WENKEN BEDIEND MET MODERNE API-GATEWAYS, TRAFFIC MANAGEMENT EN EEN WEB HOST AS-A-SERVICE”

Datacenter Quincy, WA

Maik van der Gaag

REUSING SCRIPTS AS

Azure DevOps extension

For automation purposes a lot of scripts are developed, in many of these situations the scripts are used over and over usually before and/ or after a release. Imagine a situation where the script is added to the deployment/ release pipeline as a task, and configurable like any other task.

Making an extension of the script also facilitates the reusability. Via this solution you only must maintain one code base. In this article the script in the below image is used as an example for an automation task.

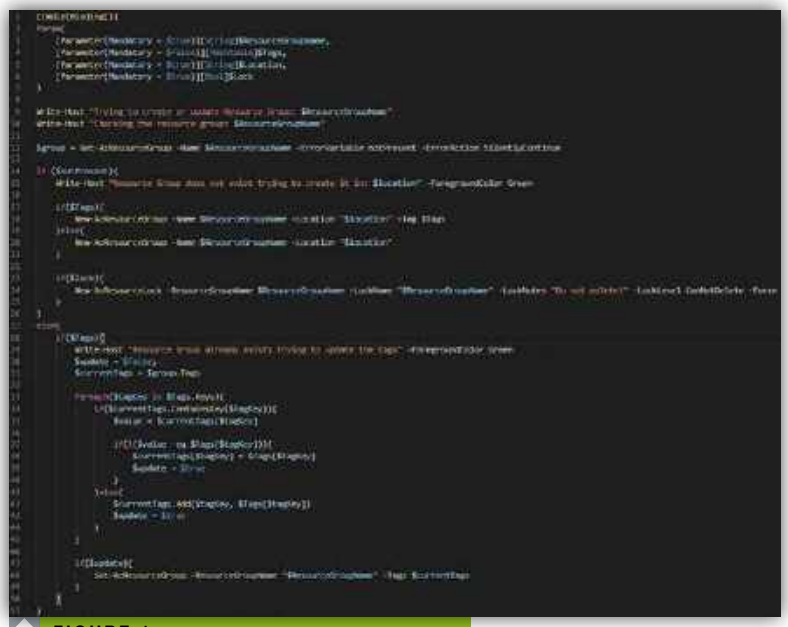


FIGURE 1

The script is used in automation scenarios were resource groups in Azure are created or updated with resources tags and resource locks. At several clients I have seen scripts like this copied into different repositories. When adjustments were needed on the script (for example an issue) it was done in one repository. As it is quite easy to incorporate the script within a pipeline using the default Azure PowerShell task.

When evolving your script to an Azure DevOps extensions a reusable solution is created for the organization in a matter of minutes.

AZURE DEVOPS EXTENSIONS

Extensions can be developed for different types of components within Azure DevOps. A few examples for extensions are:

- Azure Pipeline Tasks (Build and Release).
- Dashboards widgets.
- Work items form extensions.
- Create your own Hub (Pages)
- Actions for on the hubs.

All these extensions have a fixed format and set of files needed to operate correctly. The below image shows the required components to make an operational extension.

GETTING STARTED

To get started with Azure DevOps extensions it is best to make use of the following tools:

- Code editor, my preferred choice is Visual Studio Code
- Node CLI for Azure DevOps (tfx-cli), CLI tool for communicating with Azure DevOps.

Scaffolding the directory with the required files is the first step for the node/web extensions of Azure DevOps. There are specific scaffolding options, but these are not available for PowerShell pipeline extensions. That is why the structure needs to be created manually.

The “extension” folder in the image is the root folder for the extension where the other required files are saved.

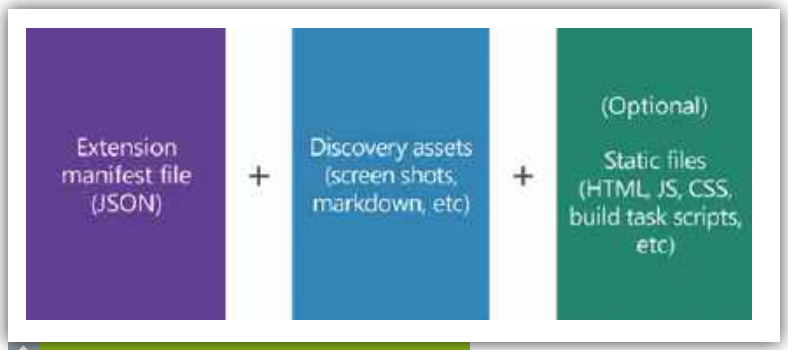


FIGURE 2

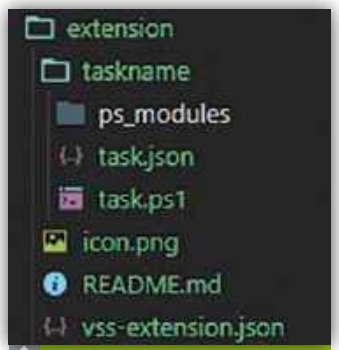


FIGURE 3

- *vss-extension.json*: The file that contains the specification of the extension. This file specifies where the extension can be used in Azure DevOps. Besides that, it contains basic information about the files that should be included in the extension.
- *README.md*: Information on the extension used for the Visual Studio Marketplace.
- *icon.png*: The icon for the extension.
- *Folder - taskname*: This folder contains the technical files for the extension. It is required to name it like the name specified in the vss-extension.json file.
- *task.json*: The specification of the task and fields for the Pipeline task.
- *task.ps1*: The script that will be executed. This script can have any name, it needs to be referenced in the task.json file.
- *Folder - ps_modules*: Folder for the PowerShell modules that are needed to run the task.

To be able to distribute an extension without sending the files to every user, a publisher within the Visual Studio Marketplace needs to be registered. Through the Marketplace it is possible to share public (available for everyone) and private (only allowed for Azure DevOps organizations with whom you share your extension) extensions.

FIGURE 4

HOW TO CREATE AND MANAGE A MARKETPLACE PUBLISHER WILL NOT BE DESCRIBED IN THIS ARTICLE BUT TO GET STARTED THE BELOW LINK CAN BE VERY HELPFUL.

HTTPS://DOCS.MICROSOFT.COM/EN-US/AZURE/DEVOPS/EXTEND/PUBLISH/OVERVIEW

SETTING UP THE FILES FOR THE EXTENSION

After scaffolding the directory, the files need to be filled with the correct information. Start with the vss-extension.json file as this is the basic file for the extension.

Documentation for this file can be found on the Microsoft documentation site - https://docs.microsoft.com/en-us/azure/devops/extend/develop/manifest?view=azure-devops. For a simple extension the manifest file can look like the one in the following image.

In this file a lot of information is specified. Let's take a look at the most important properties which are required in the file.

Property name	Description
manifestVersion	Version corresponding to the manifest version number. This number should always be 1. (There is no other version at the time of writing this article)
id	The Id of the extension. This must be unique among the extensions of the publisher.
name	Readable name for the extension (max 200 characters).
version	Version number of the extension in the form major.minor.patch
publisher	The identifier of the publisher within the Visual Studio Marketplace.
targets	The products for which this extension is supported: • Microsoft.VisualStudio.Services (extensions that works with Azure DevOps or TFS) • Microsoft.TeamFoundation.Server (extension that works with TFS) • Microsoft.VisualStudio.Services.Integration (integrations that works with Azure DevOps or TFS) • Microsoft.TeamFoundation.Server.Integration (integrations that work with TFS)
links	Collection of links that help users discover your extension.
description	Short description of the extension.
categories	Specifies for which product of Azure DevOps the extension is meant for. Possible values are: • Azure Repos • Azure Boards • Azure Pipelines • Azure Test Plans • Azure Artifacts
content	A dictionary of files that describe the extension. It is not required but I use this mainly for the readme file.
contributions	An array of items that specify which contributions you have for the platform. In the example we specify a task. The name of the task should be equal to the name of the "taskname" folder that is specified during the scaffolding. For the specification of the contribution model take a look at the following documentation: https://docs.microsoft.com/en-us/azure/devops/extend/develop/contributions-overview?view=azure-devops
files	A reference to the files that should be included in the extension.

TABLE 1

NOTE: FOR CLARITY NOT ALL PROPERTIES ARE INCLUDED IN THE EXAMPLE. FOR THE COMPLETE REFERENCE CHECK THE URL SUPPLIED EARLIER IN THIS ARTICLE.

TASK DEFINITION

As you may have noticed during the scaffolding of the folder each task has its own subfolder with the required files for that task. In this example we are creating a PowerShell extension. This type of extension also contains a sub folder called ps_modules.

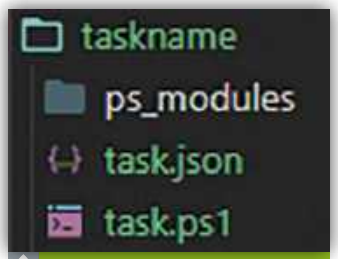


FIGURE 5

The task.json is the true specification of the task. The task manifest does have a published schema on github: https://github.com/microsoft/azure-pipelines-task-lib/blob/master/tasks.schema.json

As specified in the "task.json" file (later in the article) the "task.ps1" file is the script that will be executed within a pipeline.
The "task.json" file could look something like the following figure.

```
{
    "id": "567F7830-5655-5D17-B4C5-8ADA59A88B98",
    "name": "simpleadoextension",
    "friendlyName": "Simple Azure DevOps Extension",
    "description": "A simple build and release task for Azure DevOps",
    "author": "Maik van der Gaag",
    "helpMarkDown": "",
    "category": "Utility",
    "visibility": [
        "Build",
        "Release"
    ],
    "version": {
        "Major": "0",
        "Minor": "0",
        "Patch": "1"
    },
    "minimumAgentVersion": "2.115.0",
    "groups": [
        {
            "name": "GroupName",
            "displayName": "Group 1",
            "isExpanded": false
        }
    ],
    "instanceNameFormat": "Name for a task Instance $(stringValue)",
    "inputs": [
        {
            "name": "stringValue",
            "type": "string",
            "label": "Display value",
            "defaultValue": "",
            "required": true,
            "groupName": "GroupName",
            "helpMarkDown": "Additional information regarding the field"
        }
    ],
    "execution": {
        "PowerShell3": {
            "target": "task.ps1"
        }
    }
}
```

FIGURE 6

Let's take a look at the most important properties which are required in the file.

Property name	Description
id	A unique guid for the task.
name	The pipeline task name. I always keep this the same as the folder.
friendlyname	The display name for the task. This will be displayed in the interface of Azure DevOps.
description	Short description of the task, this will also be displayed in the UI.
author	The name of the author of the extension.
helpMarkDown	Help text that is displayed after clicking on the question marks. This field supports the markdown notation.
category	Category for the extension that is used in the classic interface when adding a task. In the new yaml pipeline interface this category isn't displayed. Possible values are: - Build - Utility - Test - Package - Deploy
description	Short description of the extension.
visibility	The visibility specifies where the pipeline tasks will be available. Possible values are: - Build - Release
version	The version number of the task. This version must be updated each time you upload the extension to Azure DevOps or the marketplace.
groups	The groups property gives the option to specify custom groups within the task interface.
inputs	In the input property, inputs field can be added to the interface for the task.
execution	The execution entry point for the extension. This is the process that should be started.

TABLE 2

As shown in the example there is a simple string input value within the "inputs" property. This value can be used within the extension and during the PowerShell execution. To use the input value, the PowerShell module "VstsTaskSdk" needs to be added to the "ps_modules" folder. Saving this module is very easy by using the following PowerShell command.

```
Save-Module -Name VstsTaskSdk -Path "[Path to ps_modules]" -Force
```

FIGURE 7

With the module included the values specified in the interface are retrieved in PowerShell. The next script shows the included PowerShell functions to use the specified input fields.

```
Trace-VstsEnteringInvocation $MyInvocation

$stringValue = Get-VstsInput -Name stringValue -Require

Write-Host $stringValue
```

FIGURE 8

MAKING USE OF EXISTING SERVICE CONNECTIONS

Many extensions in Azure DevOps make use of the Azure Resource Manager Service Connection to authenticate against the Azure Platform.

In custom extensions these service connections can also be used. To make use of these connections they need to be specified as a specific input value for the task.

```
{
    "name": "ConnectedServiceName",
    "type": "connectedService:AzureRM",
    "label": "Azure Subscription",
    "defaultValue": "",
    "required": true,
    "groupName": "AzureDetails",
    "helpMarkDown": "Select the Azure Resource Manager subscription for the deployment"
}
```

FIGURE 9

NOTE: FOR CLARITY NOT ALL PROPERTIES ARE INCLUDED IN THE EXAMPLE. FOR THE COMPLETE REFERENCE CHECK THE SCHEMA FOR THE TASK.JSON FILE (HTTPS://GITHUB.COM/MICROSOFT/AZURE-PIPELINES-TASK-LIB/BLOB/MASTER/TASKS.SCHEMA.JSON).

The input lets users select or create a service connection via the type property that is specified “ConnectedService:AzureRM”. Using input fields like this makes it easier to build up a connection in Azure DevOps Pipelines to, for example, an Azure subscription.
To make use of this input value the PowerShell script needs some additional functions as well.

```
Trace-VstsEnteringInvocation $MyInvocation

Import-Module $PSScriptRoot\ps_modules\VstsAzureHelpers_

$stringValue = Get-VstsInput -Name stringValue -Require
$serviceName = Get-VstsInput -Name ConnectedServiceName -Require

$endpoint = Get-VstsEndpoint -Name $serviceName -Require
$targetAzurePs = ""

Update-PSModulePathForHostedAgent -targetAzurePs $targetAzurePs
Initialize-AzModule -Endpoint $endpoint -azVersion $targetAzurePs

Write-Host $stringValue
```

FIGURE 10

To perform these functions an additional PowerShell module called “VSTSAzureHelper” is needed. The additional functions retrieve the chosen service connection and initializes the Azure connection via the “Initialize-AzModule” function.

PUTTING EVERYTHING TOGETHER

Evolving the scripts that is included at the beginning of the article will mean that the following values need to be placed within the inputs property of the “task.json” file.

FIGURE 11

With the specified input values the script can be executed with the appropriate arguments and the task interface will look like the image in figure 12.

Creating an execution script that will retrieve all the values specified in the Pipeline is the easiest approach. An example for a script like that is shown in figure 13.
In this script all values of the task are retrieved, it then initializes the connection to the Azure environment and executes the “New-AzureResourceGroup” script.

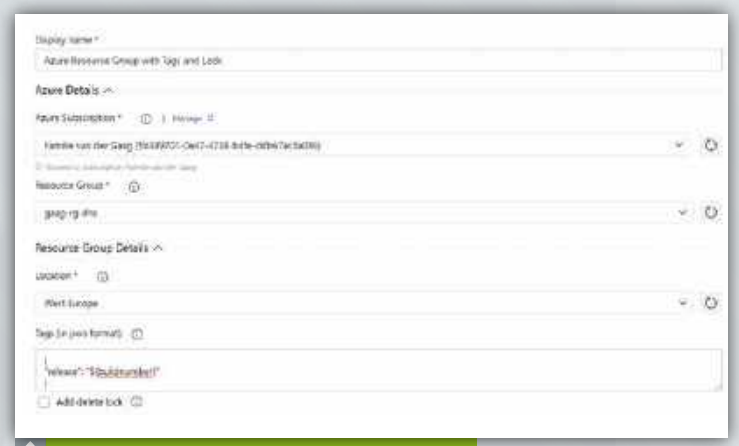


FIGURE 12

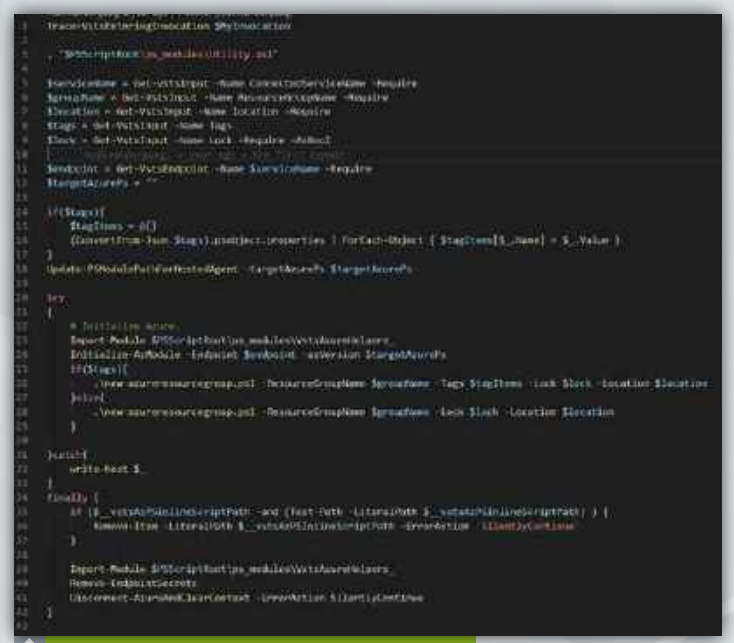


FIGURE 13

PACKAGING THE EXTENSION

When the task is finished and the script is ready, it needs to be packaged and uploaded to Azure DevOps / the Visual Studio Marketplace. To package the extension the Azure DevOps Node CLI also called “tfx-cli” is used:
The command will package the extension into a vsix file. This vsix file can be uploaded to the Publisher portal manually or the “tfx publish” command can be used to publish and directly share the extension with a Azure DevOps Organization.

```
tfx extension create --manifest-globs vss-extension.json
```

FIGURE 14

CONTINUOUS DELIVERY

These days everyone is into DevOps and continuous delivery is also an important subject. Azure DevOps extensions can be deployed continuously by using Azure DevOps or GitHub for example.
With the source code of the extension saved in a repository Azure Pipelines can be used to validate, package and install custom extensions.

To configure continuous delivery for your extensions a very handy extension is available in the marketplace called “Azure DevOps Extension Tasks”. With this extension you can easily configure continuously deliver for your own extensions.

WRAPPING UP

As you have read in this article it is relatively easy to evolve scripts to Azure DevOps extensions when knowing the required components and the possibilities of the extension platform.

With your published extensions the development and deployment teams within your organization are using the same tasks and the same settings. This will ensure that the results during your application lifecycle are more predictable and will increase the overall quality of your DevOps workflow.

For more examples of custom Azure DevOps extensions have a look at my GitHub repository:

https://github.com/maikvandergaag/msft-extensions

MAIK VAN DER GAAG

Maik van der Gaag is an Architect and Trainer at 3fifty, an experienced consultancy company with a strong focus on the Microsoft Cloud.
He has over 15 years of experience providing architecture, development, training and design expertise. He works in a variety of projects ranging from Cloud Transformations to DevOps implementations.
He loves to share his knowledge which was also one of the reasons why he founded the Dutch Cloud Meetup. Maik is a public speaker, writes blogs and organizes events.

BERGLER ICT

Zeg wat je doet. Doe wat je zegt.

No-nonsense full-service ICT-dienstverlening. Dat is waar Bergler voor staat.
In de volle breedte van het vakgebied en zonder compromis. Direct inzetbaar om de continuïteit van de bedrijfsvoering bij de opdrachtgever zeker te stellen.

Bergler Nederland B.V. Minervum 7210 4817 ZJ BREDA
info@bergler.nl T: 076 – 572 02 00

.NET Trainingen voor starters en specialisten
C# - XAML - ASP.NET Core - Visual Studio - SQL Server

Ons eigen lesmateriaal is altijd up-to-date en zeer praktijk gericht. Wij leveren maatwerk, ook in het Engels.

In-Company bij uw bedrijf.
Small-Group, 1-4 personen.
Online via Microsoft Teams.

Fons Sonnemans

MVP Microsoft® Most Valuable Professional

Reflectionit.nl

Maurice de Beijer

Ontwikkel React componenten met Storybook

De meeste ontwikkelaars maken hun React componenten samen met de applicatie waar ze aan werken. Op zich is dat niet fout want uiteindelijk werkt de applicatie wel. Maar is het de beste manier? Misschien niet. Probeer je een herbruikbare bibliotheek met componenten te maken? Moet je vaak veel stappen in de applicatie doen om te zien of de componenten goed werken? Hebben andere ontwikkelaars problemen met het juist gebruiken van de componenten? Of is het lastig om in de applicatie te zien wat de gevolgen zijn van een CSS wijziging? In die gevallen kan Storybook een uitkomst zijn.

WAT IS STORYBOOK?

Storybook is een stuk gereedschap dat het voor ontwikkelaars makkelijker maakt om componenten te ontwikkelen. De componenten worden dan interactief ontwikkeld. Denk hierbij een beetje aan Test Driven Development maar dan voor de visuele kenmerken van een component. In dit artikel gebruik ik React componenten maar dat hoeft niet. Storybook werkt net zo goed met andere web bibliotheken als Angular, Vue, Ember, Svelte en nog meer. Door de componenten binnen Storybook te ontwikkelen wordt het makkelijk om alle verschillende vormen van een component snel te bekijken. Je hoeft niet meer een applicatie te starten, een gegeven op te zoeken en een invoerfout te introduceren om te zien hoe dit aan de gebruiker getoond wordt. Nee, met Storybook maak je gewoon een story voor die mogelijkheid en je kan het gelijk zien.

Heb je documentatie nodig voor andere ontwikkelaars die jouw componenten gebruiken? Ook daar helpt Storybook. Je kan bij een story documentatie schrijven en die weer als een statische website publiceren, bijvoorbeeld gratis op GitHub pages.

Je zal zeker niet de eerste zijn die dat doet. Veel bekende en minder bekende ontwikkelaars zijn je al voor gegaan. Zo kan je op het web componenten bibliotheken vinden van bedrijven zoals AirBNB, Lonely Planet, Auth0 of SalesForce die hun opensource componenten bibliotheken met Storybook ontwikkelen en documenteren.

En over opensource gesproken, Storybook zelf is ook opensource. Dus je kan er zo mee aan de gang zonder het eerst te moeten kopen, je kan de broncode bekijken als iets niet duidelijk is en je kan zelf bijdrages leveren als je dat wil.

BEGINNEN MET STORYBOOK IN EEN REACT PROJECT

Als je Storybook toe wil voegen aan een bestaand React project gaat dit meestal zeer eenvoudig. Via een console venster voer je het commando npx sb init in de project folder uit. Deze kijkt dan naar het soort project en zal Storybook automatisch configureren. In het geval van een React applicatie die met Create React App gemaakt is zal dit met de standaard *@storybook/preset-create-react-app* bibliotheek gebeuren. Zo zijn er voor veel andere omgevingen vergelijkbare standaard bibliotheken.

Deze instelling kun je in de .storybook folder in main.js terugvinden. In dezelfde main.js kan je ook de diverse addons terugvinden die gebruikt worden, iets waar ik later op terugkom. Naast de main.js is er ook een preview.js in de .storybook folder. Deze preview.js wordt gebruikt om het canvas waar de component door een story getoond

wordt aan te passen.
Tegelijk zijn er in de package.json twee nieuwe scripts toegevoegd. De belangrijkste voor nu is storybook die je kan starten met npm run storybook om Storybook in een browser te starten. Als je dat commando uitvoert zie je gelijk een drietal standaard stories die als voorbeeld toegevoegd zijn.

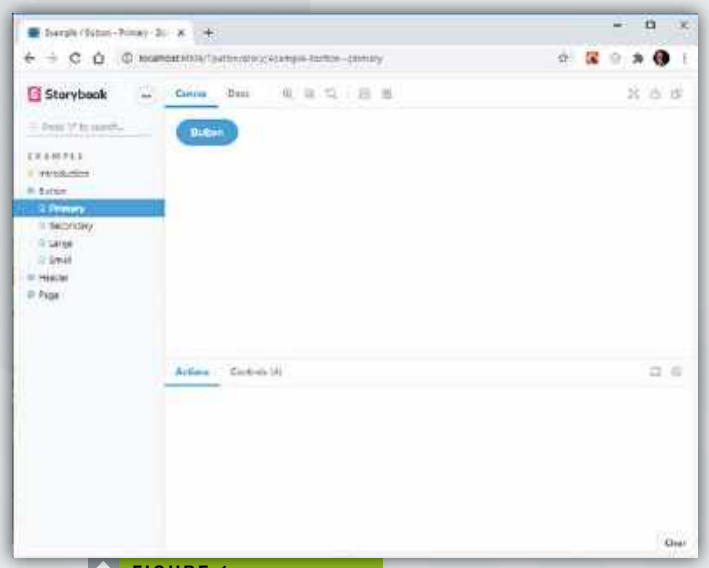


FIGURE 1

Zelf een eerste story maken is niet moeilijk. Er zijn verschillende opties om stories te maken maar de meest gebruikte, en beste om mee te beginnen, is de zogenaamde Component Story Format. Maar voor we dat gaan doen moeten we eerst de vraag beantwoorden wat een story nu eigenlijk is?

EEN STORY GEEFT DE VISUELE WEERGAVE VAN DE STATUS VAN EEN COMPONENT WEER.

Denk aan een input component. Die kan je op verschillende manieren op een scherm hebben. Zo kan de invoer leeg zijn. Of de invoer is niet leeg en valide. Maar de invoer kan ook gevuld en niet valide zijn. Dat zijn drie verschillende statussen van dezelfde input component met allemaal een net iets andere weergave. Dat zijn dus drie verschillende stories voor dit component.
Als demo heb ik een <LabeledInput /> component gemaakt dat in een invulformulier gebruikt kan worden. Dit component gebruikt Formik, een populaire helper voor React formulieren, achter de schermen om het iets uitdagender te maken. Zie hier voor de code.

EEN EERSTE STORY MAKEN

Om een eerste story te maken moeten we eerst een nieuw bestand met de naam LabeledInput.stories.tsx maken. Storybook zoekt namelijk naar bestanden die eindigen op .stories.tsx. Uiteraard kan een JavaScript versie hiervan ook maar ik heb een voorkeur voor TypeScript.

In dit bestand moeten we eerst React en het component importeren zodat we er mee kunnen werken. Daarna moeten we een default export met de metadata van de stories toevoegen. Hierbij geven we minimaal de titel en een referentie naar het component waar de stories voor gemaakt zijn aan.
Vervolgens gebruiken we een of meer named exports om de echte stories voor het component te maken. Deze stories zijn meestal heel eenvoudig en renderen het component met de benodigde props.

```
import type { Meta, Story }
  from "@storybook/react/types-6-0";
import React from "react";
import LabeledInput from "./LabeledInput";

export default {
  component: LabeledInput,
  title: "Elements/LabeledInput",
} as Meta;

export const Standard: Story = () => (
  <LabeledInput label="Firstname:" name="firstName" />
);
```

Als we Storybook nu starten met npm run storybook zal de Storybook UI zichtbaar worden in de browser. Helaas zijn er we nog niet helemaal omdat we bij deze story een runtime fout zullen zien. Deze wordt veroorzaakt doordat het component verwacht dat Formik aanwezig is. Iets wat nu niet het geval is. De beste manier om dit op te lossen is een decorator toe te voegen.
Een decorator wordt gebruikt op plaatsen waar je iets aan een component story toe wil voegen voor technische of cosmetische redenen. Decorators kunnen op verschillende punten toegevoegd worden. Ze kunnen globaal voor heel Storybook zijn, voor een set van stories in een bestand of voor een individuele story. In dit geval voeg ik de decorator op bestands niveau toe zodat die door meerdere stories gebruikt kan worden. Dit kan door de decorator in de metadata toe te voegen. Ik heb gelijk ook een tweede story toegevoegd om de invoer met een validatiefout weer te geven.

```
import type { Meta, Story }
  from "@storybook/react/types-6-0";
import React from "react";
import { Formik } from "formik";
import LabeledInput from "./LabeledInput";

export default {
  component: LabeledInput,
  title: "Elements/LabeledInput",
  decorators: [
    (storyFn) => {
      return (
        <Formik
          initialValues={{ firstName: "Maurice",
lastName: "" }}
          initialErrors={{
            lastName: "Lastname is required",
          }}
          onSubmit={() => {}}
        >
          {storyFn()}
```

```
          </Formik>
        );
      },
    ],
  } as Meta;

export const Standard: Story = () => (
  <LabeledInput label="Firstname:" name="firstName" />
);

export const WithError: Story = () => (
  <LabeledInput label="Lastname:" name="lastName" />
);
```

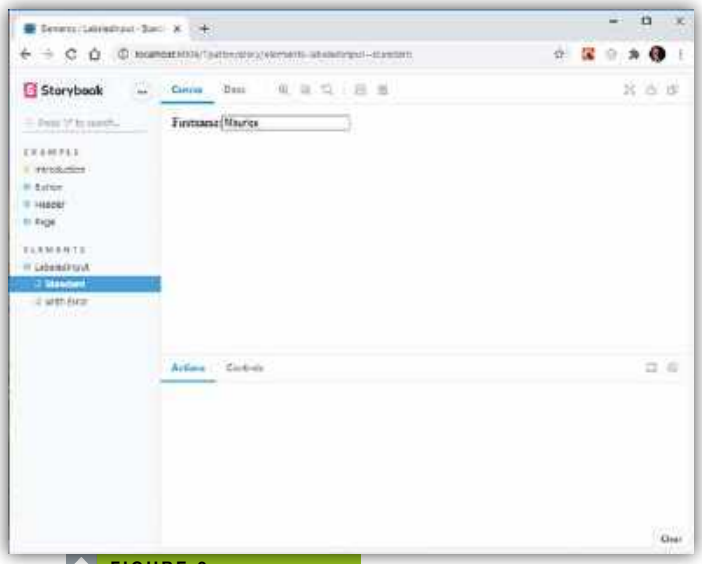


FIGURE 2

Als we de stories in de browser bekijken zien we de user interface wel maar ziet die er niet erg mooi uit. De reden hiervoor is dat er gebruik gemaakt wordt van Boostrap CSS styling maar dat dit niet beschikbaar is in het Storybook preview scherm. Dit is makkelijk te doen door een import van de CSS toe te voegen in .storybook\preview.js.

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Met deze wijziging ziet alles er uit zoals de bedoeling is. Nu is het makkelijk om wijzigingen toe te voegen in het component of de styling en snel te kunnen zien wat de gevolgen hiervan zijn.

GEBRUIKEN VAN STORYBOOK ADDONS

De standaard inrichting van Storybook bevat al een aantal nuttige addons. Een daarvan is de Controls addon. Deze vervangt de Knobs addon uit oudere versie van Storybook.

Het gebruik van de Controls addon is sterk aan te bevelen. De stories die we tot nu toe gemaakt hebben zijn hard gecodeerd op bepaalde waardes. Dat is prima maar het zou veel handiger zijn als een gebruiker die waardes ook aan kan passen om te zien wat er gebeurt. Op die manier is het makkelijk om andere dingen te testen en zien wat er gebeurt. Dit kan heel gemakkelijk door de Controls addon te gebruiken. Aangezien de Controls addon onderdeel is van de essentials addon is hij standaard al aanwezig en kunnen we er direct mee aan de slag.

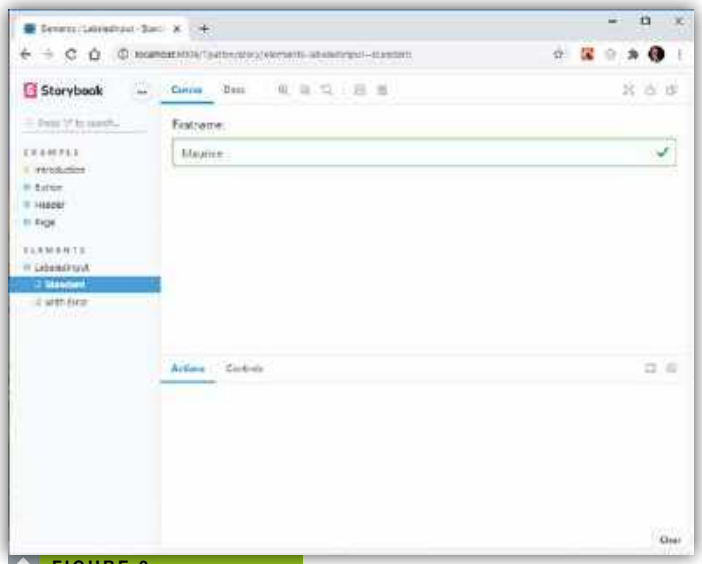


FIGURE 3

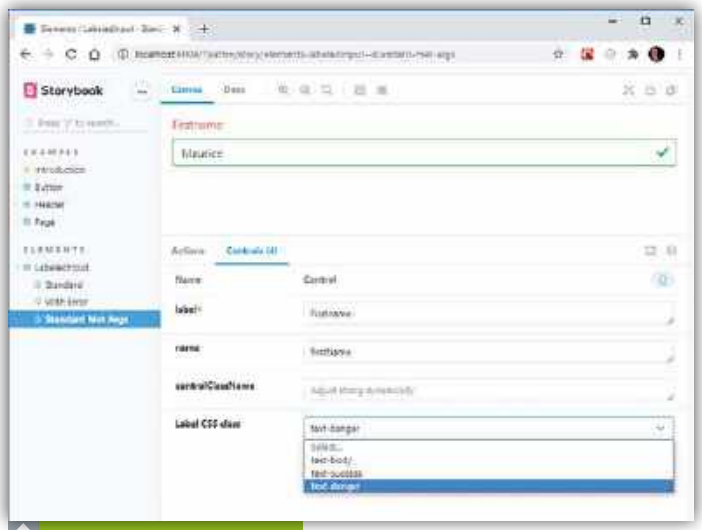


FIGURE 4

Het gebruik van de essentials addon in een story is makkelijk. Door het gebruik van TypeScript zal Storybook automatisch de gebruikte props van een component herkennen en in de browser tonen. De waardes hiervan worden dan als argument aan de story meegegeven. Door deze eenvoudigweg met de spread operator aan de component tot te voegen zullen deze gebruikt worden. En als de waardes in de UI veranderen dan wordt de story automatisch opnieuw gerenderd met de nieuwe waardes. Standaard zullen de waardes in de Controls addon leeg zijn maar door een args property aan de story toe te voegen kan een default waarde opgegeven worden.

```
export const StandardMetArgs:
  Story<ComponentProps<typeof LabeledInput>> = (
  args
) => <LabeledInput {...args} />;
```

```
StandardMetArgs.args = {
  label: "Firstname:",
  name: "firstName",
};
```

Soms is het ook handig om niet alle waardes toe staan. Dit is met de Controls addon ook makkelijk te regelen. Zo kan het soort invoer veranderd worden, bijvoorbeeld een range slider of een drop down select in plaats van een vrije invoer. Zie bijvoorbeeld de code hieronder waar ik een drop down gebruik om maar een paar CSS classes toe te staan voor het label.

```
StandardMetArgs.argTypes = {
  labelClassName: {
    name: "Label CSS class",
    defaultValue: "text-body",
    control: {
      type: "select",
        options: ["text-body", "text-success", "text-
danger"],
    },
  },
};
```

Zo zijn er nog veel meer nuttige Storybook addons. Kijk maar eens op https://storybook.js.org/addons/ voor een lijst van officiële en community addons. Sommige zijn heel makkelijk toe te voegen en heel nuttig. Zo is er een Accessibility addon die controleert op een hele lijst van toegankelijkheidsregels zoals bijvoorbeeld kleurcontrast en de aanwezigheid van aria attributen. Installeren is makkelijk door eerst *npm install @storybook/addon-a11y –dev* uit te voeren in een commando venster. Daarna de main.js openen en @storybook/addon-a11y toevoegen aan de addons definitie.

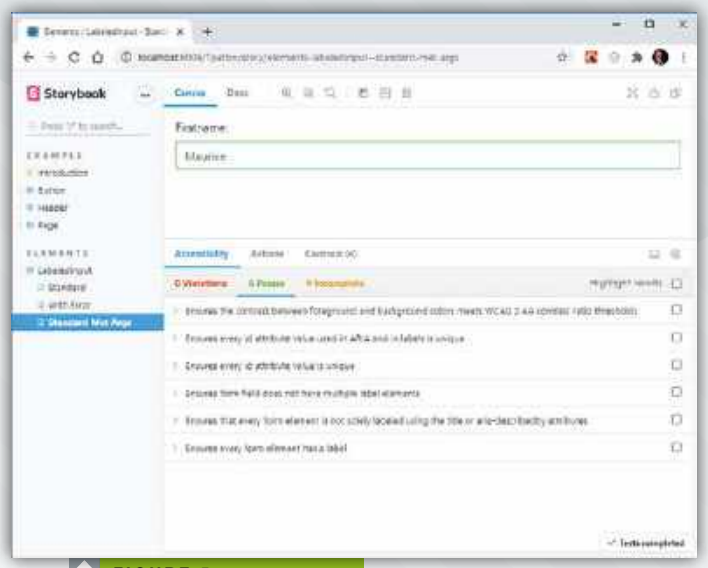


FIGURE 5

STORYBOOK ALS DOCUMENTATIE

Een andere handige optie van Storybook is om van de stories een statische site te bouwen en die als handleiding te hosten. Aangezien het resultaat een statische site is zijn er legio goedkope hosting opties zoals GitHub pages.

Het bouwen van de statische site gaat met het commando npm run build-storybook. Hierna is er een nieuwe folder storybook-static. Deze folder kan nu met een willekeurige web server gehost worden. Als je dat met GitHub pages of Amazon S3 wil gebruiken is de Storybook Deployer een handige tool van het Storybook team zelf.

CONCLUSIE

Het gebruik van Storybook is zo makkelijk en handig dat het bijna een standaard geworden is. De vele addons maken het makkelijk om precies te doen wat je wil. En het resultaat is makkelijk als documentatie of stijlgids toe te voegen.

MAURICE DE BEIJER

Maurice de Beijer is an independent software consultant and trainer. He specializes in TypeScript, ECMAScript, React and ASP.NET MVC. His work includes a large, global, safety application for the oil and gas industry. Maurice is also active in the open source community. He teaches ECMAScript, TypeScript, React, RxJS and ASP.NET MVC courses. Since 2005, he has received Microsoft's Yearly Most Valuable Professional Award. Further, Maurice is active in the Dutch dotNed user group and helps organize its meetings.

STORYBOOK IS OPEN SOURCE. DUS JE KAN ER ZO MEE AAN DE GANG ZONDER HET EERST TE MOETEN KOPEN, JE KAN DE BRONCODE BEKIJKEN ALS IETS NIET DUIDELIJK IS EN JE KAN ZELF BIJDRAGES LEVEREN ALS JE DAT WIL.

Visual Studio

Verbind. Synchroniseer. Transformeer de wereld van apps.

Welkom in het tijdperk van buitengewone verwachtingen. Want elke app moet altijd soepel op ieder apparaat kunnen werken. Zonder haperingen en problemen. Visual Studio helpt ontwikkelaars te presteren met de meest geavanceerde én geïntegreerde oplossing die er op de markt is: een ultramoderne set met tools en services die u helpen bij het ontwikkelen, testen en implementeren van servicegerichte apps. De gebruikservaring wordt moeiteloos getransformeerd op alle Windows-apparaten.

Ontdek Visual Studio
www.visualstudio.com

Het zijn mooie tijden voor ontwikkelaars

Microsoft

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