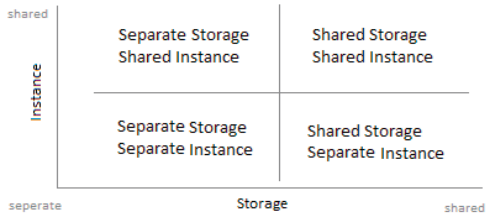
Multitenancy op Azure in de praktijk

Beschikbaarheid en schaalbaarhheid zijn sterke punten

Dennis Mijer en Matthijs van der Veer

Multitenancy is een principe dat al jaren bestaat; het bedienen van meerdere separate klanten vanuit een enkele applicatie. Ten opzichte van multi-instancing (waarbij dezelfde software meerdere malen wordt uitgerold) biedt dit onder andere voordelen in termen van kosten en beheer.

Deze twee principes leiden tot vier verschillende uitwerkingen. In figuur 1 is te zien op welke manieren de klanten bedient kunnen worden.



FIGUUR 1: VERSCHILLENDE MANIEREN VOOR HET BEDIENEN VAN KLANTEN.

Het voorbeeld dat we tonen is een uitwerking van de gedeelde instantie met de gescheiden opslag. We behandelen in dit artikel ook een aantal knelpunten en vraagstukken zoals het regelen van de dataopslag en het voorzien van voldoende beveiliging.

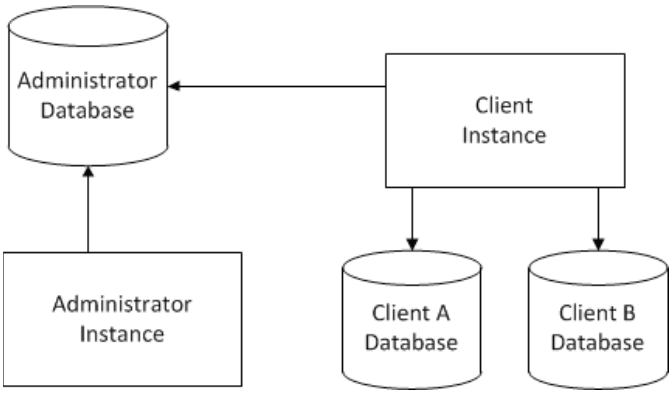
Het voorbeeld

De applicatie die in dit artikel wordt besproken, is een administratieve applicatie die ontworpen is om door meerdere klanten benaderd te worden. Elke klant heeft gebruikers die binnen de klantomgeving handelingen kunnen uitvoeren zoals het bewerken en toevoegen van data. Deze data moet alleen te zien zijn door medewerkers van de klant.

Name	Type	Status	Environment
[redacted]	Subscription	Active	
multitenant	Hosted Service	Created	
Certificates			
multitenant	Deployment	Ready	Production
UserWebApp	Role	Ready	Production
UserWebApp_IN_0	Instance	Ready	Production
UserWebApp_IN_1	Instance	Ready	Production
AdminWebApp	Role	Ready	Production
AdminWebApp_IN_	Instance	Ready	Production
AdminWebApp_IN_	Instance	Ready	Production

FIGUUR 2: DE GEDEPLOYDE ROLES.

Het voorbeeld maakt gebruik van een enkele instantie en meerdere databases. Daarnaast wordt er gebruik gemaakt van een algemene administrator front end waarin de klantomgevingen gemaakt en aangepast kan worden. Deze aparte instantie kan worden uitgeschakeld wanneer deze niet wordt gebruikt en zal op dat moment ook geen extra kosten genereren.



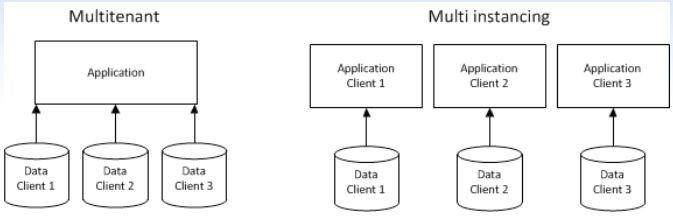
FIGUUR 3: MODEL VAN HET VOORBEELD.

Multitenancy vs. Multi instancing

Het voordeel van multitenancy is dat de applicatie op één plek wordt beheerd en dat de applicatie voor iedereen gelijk is, wat het opsporen van fouten makkelijker maakt. Daar ligt meteen ook het voordeel van multi-instancing, waarbij je aanpassingen in de code zou kunnen maken per klant en daarmee beter in kan spelen op hun wensen.

Uiteindelijk is multitenancy al snel aantrekkelijker wanneer er wordt gekeken naar de kosten; je maakt een eenmalige investering door het systeem iets meer generiek te maken en een aantal opties toe te voegen die de klant in staat stelt zijn omgeving binnen de applicatie zelf in te richten.

Dit voorkomt echter dat je voor elke klant een aparte code base moet onderhouden, je verdient de investering dus snel weer terug.



FIGUUR 4: HET VERSCHIL TUSSEN MULTITENANCY EN MULT-INSTANCING.

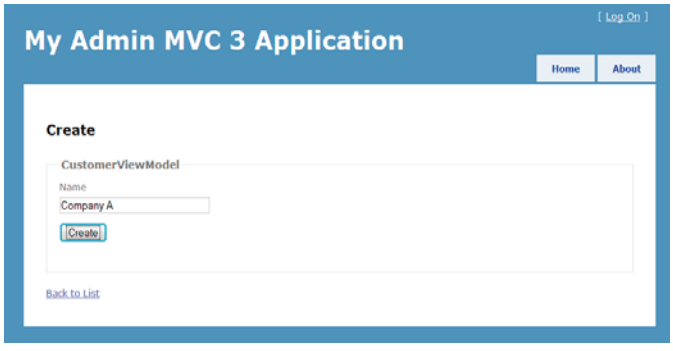
Wanneer men een applicatie wil deployen naar Azure komt er een waarschuwing te staan die meldt dat de applicatie minimaal twee instanties moet draaien om aan de Microsoft SLA te voldoen. Hierdoor lijkt het voorbeeld misschien multi-instancing, maar het voorbeeld dat wij gebruiken is wel degelijk multitenant, omdat beide instanties uit dezelfde sourcecode bestaat en op dezelfde omgeving draaien. Op het moment dat meerdere klanten tegelijk gebruik maken van de applicatie kan de workload worden verdeeld over de instanties. Ook zorgt dit ervoor dat wanneer één instantie kapot is, de tweede zijn plaats kan innemen.

Waarom in de Cloud?

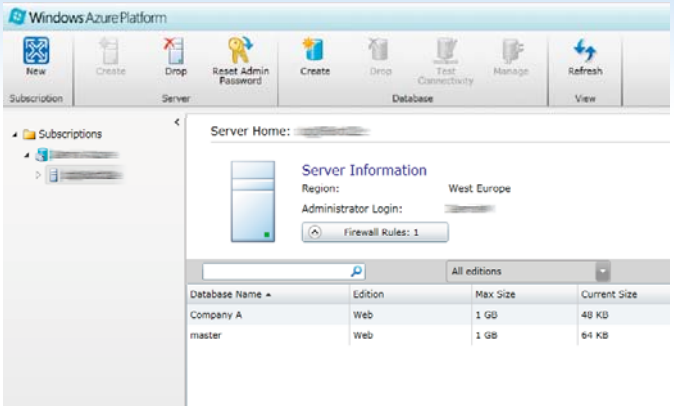
Met de komst van Windows Azure kan het multitenancy-principe in een nieuw jasje worden gestoken. Zo zijn er enkele nieuwe voordelen.
Een voordeel is de locatie waar de applicatie leeft. De applicatie kan direct worden gehost in datacenters verspreid over de wereld. Zo kan de vertraging voor klanten aan de andere kant van de wereld zoveel mogelijk worden beperkt.
Zoals al eerder besproken is een ander voordeel bereikbaarheid. Door meerdere instanties te gebruiken is het dus niet erg wanneer er eentje uitvalt. Mochten er meer dan twee instanties nodig zijn, is het ook een kleine moeite er meer te verkrijgen.
De grootste winst haal je echter met de kosten. Er is geen hardware in de vorm van server(farms), tenminste, niet op locatie bij het bedrijf of bij de klanten. Dat bespaart dure aankopen van servers, het onderhoud en reparatie van deze hardware. Dit zorgt ook voor een duidelijk prijsmodel voor je klanten, zij komen niet voor onverwachte verassingen te staan.

Gegevensopslag

Het grootste obstakel voor veel klanten om over te stappen naar een multitenant-omgeving is dataprivacy. Hoe leg je aan de klant uit dat het wel degelijk veilig is om zijn data op een onbekende plaats te bewaren zodat het overal beschikbaar is.
Om deze privacy te garanderen hebben we ervoor gekozen om voor elke klant een eigen database in te richten. Dus hoewel je de front end van de applicatie wel degelijk met andere klanten deelt, merk je niks van de data van andere klanten.



FIGUUR 5: HET AANMAKEN VAN EEN DATABASE.



FIGUUR 6: DE GEMAAKTE DATABASE IN DE AZURE PORTAL.

Zoals te zien in figuur 5 kan het aanmaken van een nieuwe klantomgeving met één druk op de knop. Dit wordt gerealiseerd door het nieuwe Entity Framework 4. Een belangrijke keuze was geen code of databasemodel te laten genereren. We maken onze database Code First. Dit houdt in dat we zelf onze klassen programmeren en deze dan omzetten naar een database. Op deze manier kunnen we ook terwijl de applicatie live is nieuwe databases aanmaken en houden we volledige controle over ons domeinmodel. Het maken van de klassen kost op deze manier in het begin van een project meer tijd dan wanneer je het zou laten genereren. Op den duur betaald het zich echter terug, omdat je de vrije hand hebt in het aanpassen van deze klassen naar je wensen. Je haalt zo het meeste uit dit framework.

Als voorbeeld nemen we de voorbeeldklasse Person (zie figuur 7). Deze simpele klasse heeft een aantal velden die we willen gebruiken.

```
namespace CodeSamplesAzureArticle.Domain
{
public class Person
{
public int Id { get; set; }
public string Email { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
}
}
```

FIGUUR 7: DE KLASSE PERSON.

Entity Framework 4 heeft wat uitleg bij deze klasse nodig om hem te vertalen naar een tabel, dat doe je door middel van de FluentAPI (zie figuur 8). In deze vertaling kun je de meta data, bijvoorbeeld de lengte van een veld aangeven.

```
namespace CodeSamplesAzureArticle.Configuration
{
public class PersonConfiguration : EntityTypeConfiguration<Person>
{
public PersonConfiguration()
{
HasKey(p => p.Id);
Property(p => p.Email).IsRequired().HasMaxLength(254);
Property(p => p.FirstName).IsRequired();
Property(p => p.LastName).IsRequired();
}
}
}
```

FIGUUR 8: DE CONFIGURATIEKLASSE.

Om de database dan te maken gebruik je de contextklasse. In deze contextklasse kun je allerlei configuratieklassen toevoegen om die

dan te combineren tot één database (zie figuur 9).

```
namespace CodeSamplesAzureArticle.Context
{
public class MyDatabaseContext : DbContext
{
         public DbSet<Person> Persons { get; set; }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            //Add tables here
            modelBuilder.Configurations.Add(new Person
Configuration());
            base.OnModelCreating(modelBuilder);
        }
     }
}
```

FIGUUR 9: DE CONTEXT KLASSE.

Het aanmaken van aparte databases heeft nog een voordeel, namelijk bij de facturering naar de klant. Bij Azure betaal je elke maand per database, zo zijn de kosten per klant altijd hetzelfde wat de database betreft. Omdat de databases niet worden gedeeld door meerdere klanten is de applicatie ook uiterst schaalbaar. SQL Azure stelt geen maximum aan databases.
Natuurlijk zal de applicatie wel ergens vast moeten leggen welke klantenomgevingen bestaan en welke database daarbij hoort, om deze reden zorgen we voor een aparte database voor deze gegevens, die beheert wordt door een administrator van je eigen bedrijf. De gegevens in deze database zijn werkelijk minimaal, maar desondanks erg belangrijk. De administratie front-end geeft ook toegang tot functies zoals het verwijderen van een klantomgeving.

Security

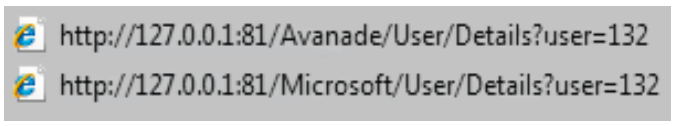
Naast aparte databases, krijgt elke klant ook een aparte URL. Aan de hand van deze URL weet de applicatie welke database hij mag benaderen. Het verzorgen van een aangepast URL is een koud kunstje in MVC3, je maakt simpelweg een nieuwe 'route' aan.

```
routes.MapRoute(
  "MultiTenant", // Route name
  "{tenant}/{controller}/{action}/{id}", // URL with parameters
  new

{ controller = "Home", action = "Index", id = UrlParameter.
Optional}
// Parameter defaults
                );
```

FIGUUR 10: VOORBEELD VAN DE ROUTE.

In de bovenstaande afbeelding zie je hoe een route aangemaakt wordt in MVC3, we introduceren daar een nieuwe paramater '{tenant}' waar later de naam van de klant in komt te staan. Figuur 11 toont de URLs zoals ze zouden zijn voor de verschillende bedrijven.



FIGUUR 11: VOORBEELD VAN DE RESULTERENDE URLS.

Bij elke pagina die de gebruiker bezoekt wordt allereerst gecontroleerd of hij nog wel binnen zijn eigen omgeving probeert te handelen. Op deze manier zal de gebruiker van de ene klant niet de omgeving van de andere kunnen bezoeken.

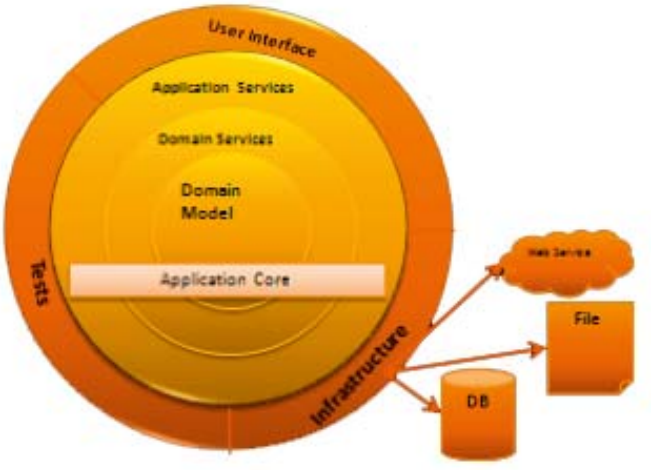
Bij een multitenant applicatie is het bijna een vereiste dat er een onderscheid tussen gebruikers wordt gemaakt. Als je te maken hebt met meerdere omgevingen wil je de klant ook een stukje controle geven, dit zorgt ervoor dat ze simpele beheertaken zoals het toevoegen van een gebruikersaccount zelf kunnen doen. Om dit te realiseren is er een administratoraccount beschikbaar. Dit houdt natuurlijk ook in dat je dit deel wilt afschermen van de rest van de gebruikers.
Dit hebben we geïmplementeerd door bepaalde taken te definiëren. Elke set aan acties heeft een eigen taak. Rollen bestaan dus uit een set van taken, op deze manier kun je voor elke klant andere rollen samenstellen, weer een stukje flexibiliteit voor de klant.

Onderhoud

Dergelijke software onderhouden is complexer dan het onderhoud voor een enkele klant. Om deze instanties te onderhouden hebben de beheerders een krachtig middel tot hun beschikking. De Windows Azure Portal. Hier kan de applicatie voor het eerst worden uitgerold naar de staging omgeving. Deze heeft een tijdelijke link waarmee de applicatie getest kan worden binnen Azure. Wanneer de applicatie goed draait is hij snel te verhuizen naar de productieomgeving.
Mocht zich het geval voordoen dat er veranderingen binnen de applicatie plaatsvinden kan de applicatie opnieuw gedeployed worden naar de staging omgeving. Terwijl de oudere versie blijft draaien in productie. Wanneer de wijzigingen zijn goedgekeurd kan de nieuwe versie naar productie verplaatst worden. Daarna kunnen de instanties opnieuw gestart worden met de nieuwe versie en is de applicatie (nagenoeg) zonder down time vernieuwd.

Onion Model

Voor dit project hebben we gekozen voor een implementatie van het Onion Model, deze architectuur is niet zo bekend, maar biedt wel veel voordelen. Er is voor deze architectuur gekozen, omdat elke laag makkelijk op een andere wijze opnieuw kan worden geïmplementeerd. Mocht het nodig zijn kun je op deze manier ook gemakkelijk schalen in de applicatie.



FIGUUR 12: THE ONION MODEL.

Om dit model uit te leggen beginnen we vanuit de binnenkant. Een belangrijk aspect van deze architectuur is 'Persistance Ignorance', letterlijk vertaald betekent dit onwetendheid over de persistentie. Dit betekent dat de binnenste laag (het domeinmodel) geen enkel idee heeft hoe het wordt opgeslagen of opgehaald. Je

wilt natuurlijk wel dat andere lagen weten hoe dit moet, om deze reden bestaat de tweede laag uit interfaces die de namen van functies biedt voor opslag die later geïmplementeerd wordt.
Door de implementatie van de dataopslag te verbergen voor de rest van de applicatie is het bijvoorbeeld mogelijk om te kiezen voor een opslag in Azure Storage van bepaalde onderdelen in plaats van SQL Azure zonder verdere aanpassingen in de rest van de applicatie.

De implementatie van dit voorbeeld wordt SQL Azure gebruikt, dit betekent wel dat voor elke user bijgehouden moet worden tot welke database hij toegang heeft. Deze data slaan we op bij het inloggen en worden bewaard in de ticket van de gebruikte Forms authenticatie. Door deze data niet in de session state op te slaan, scheelt dit ook weer in de kosten. De session state van Azure maakt namelijk gebruik van SQL Azure of Azure Storage, hiermee maak je dus ook kosten.

Om deze laag zit de applicatielogica, deze kent de interfaces wel, maar de uiteindelijke implementatie is onbekend. De buitenste laag bestaat uit meerdere onderdelen. Allereerst de infrastructuur, hier bevinden zich de implementaties van de interfaces voor de opslag. Door deze in de buitenste laag te plaatsen is hij naast de interface compleet onafhankelijk van andere lagen.

In de laatste laag bevindt zich ook de presentatiekant. Een klassiek probleem is echter de weergave van data. Deze data hebben we geplaatst in het domein, maar hoe geven we deze weer zonder de presentatielaag kennis te geven van het domein? Om dit probleem op te lossen gebruiken we View Models (zie figuur 12). Deze klassen bevatten daarmee ook alleen de data die we weer willen geven en niks meer.

```
namespace CodeSamplesAzureArticle.ViewModel
{
public class PersonViewModel
{
public string Email { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
}
}
```

FIGUUR 13: HET VIEWMODEL VOOR PERSON.

Customisation

Het oog wil natuurlijk ook wat, één applicatie voor meerdere klanten is erg rendabel, maar het kan een stukje 'look and feel' missen. Een aardige feature voor een multitenant applicatie is bijvoorbeeld een thema dat je kan aanpassen aan de huisstijl van je bedrijf. Dit kun je doen door een compleet andere stylesheet te gebruiken, of misschien te laten genereren vanuit de database. Wanneer je dit soort functies overweegt, onthoudt dan dat de aanpasbaarheid uiteindelijk een limiet heeft wanneer je gebruik maakt van multitenancy.

Zelfs bij simpele features als het aanpassen van een thema zul je enkele dingen in overweging moeten nemen. Als je bijvoorbeeld zou kiezen om een aparte stylesheet te gebruiken voor elke klant, onthoudt dan wel dat deze opgeslagen moeten worden in de cloud. Door het uit dezelfde database te laten generen, maak je geen extra kosten.
Doordat je op deze manier aanpassingen aan de klantomgeving erg toegankelijk maakt kan de klant deze opties ook zelf in gaan vullen, dit kan de klant weer geld besparen omdat de dure programmeur het nu niet hoeft te doen.

Kosten

Om deze applicatie in de cloud te hosten betaal je elke maand de kosten. Deze kosten kun je opdelen in vaste kosten voor bijvoorbeeld de databases en variabele kosten als je het hebt over gebruik van de applicatie. Vaste kosten zijn makkelijk door te berekenen aan de klant, omdat je weet hoeveel databases zij gebruiken. Maar hoe bepaal je de kosten voor variabel gebruik? Windows Azure heeft hier geen ingebouwde oplossing voor, dit stelt je dan uiteindelijk voor twee oplossingen.

De eerste oplossing is om simpelweg een vast bedrag af te spreken met de klant. Door vooraf informatie te verzamelen als het aantal gebruikers bij de klant kun je een aardige schatting maken van het gebruik van de applicatie.

Een andere mogelijkheid is om het gebruik zelf bij te houden. Zo zou je bijvoorbeeld de Azure Diagnostics kunnen gebruiken om elke login of zelfs elke request bij te houden. Hieruit kun je dan zelf weer rapporten genereren over het gebruik. Op deze manier kun je de kosten weer doorberekenen aan de klant.

Conclusie

Multitenancy en Windows Azure complementeren elkaar op veel vlakken. De beschikbaarheid en schaalbaarheid die Azure beloofd zijn juist de sterke punten waarom mensen multitenant applicaties gebruiken. De verschillende roles zorgen ervoor dat de klanten geen hinder ondervinden van de load die andere tenants met zich meebrengen. Bent u niet overtuigd genoeg om een nieuwe applicatie te ontwikkelen voor de cloud, bedenk dan dat elke goed ontworpen applicatie met weinig inspanning in de cloud neergezet kan worden.

Dennis Mijer en Matthijs van der Veer,
zijn solution developers bij Avanade Nederland. Zij zijn te bereiken via dennis.mijer@avanade.com en m.van.der.veer@avanade.com