# CQRS, event sourcing en Windows Azure

## Transacties en bevragingen scheiden en weer koppelen

**Tijmen van de Kamp**

CQRS (command query responsibility segregation) en event sourcing zijn op dit moment populaire patterns waar veel over geschreven wordt. Bij CQRS worden transacties in en bevragingen aan een systeem op een fundamenteel architectureel niveau van elkaar gescheiden, om vervolgens door events aan elkaar te worden gekoppeld. Event sourcing houdt in dat al deze events letterlijk worden opgeslagen.

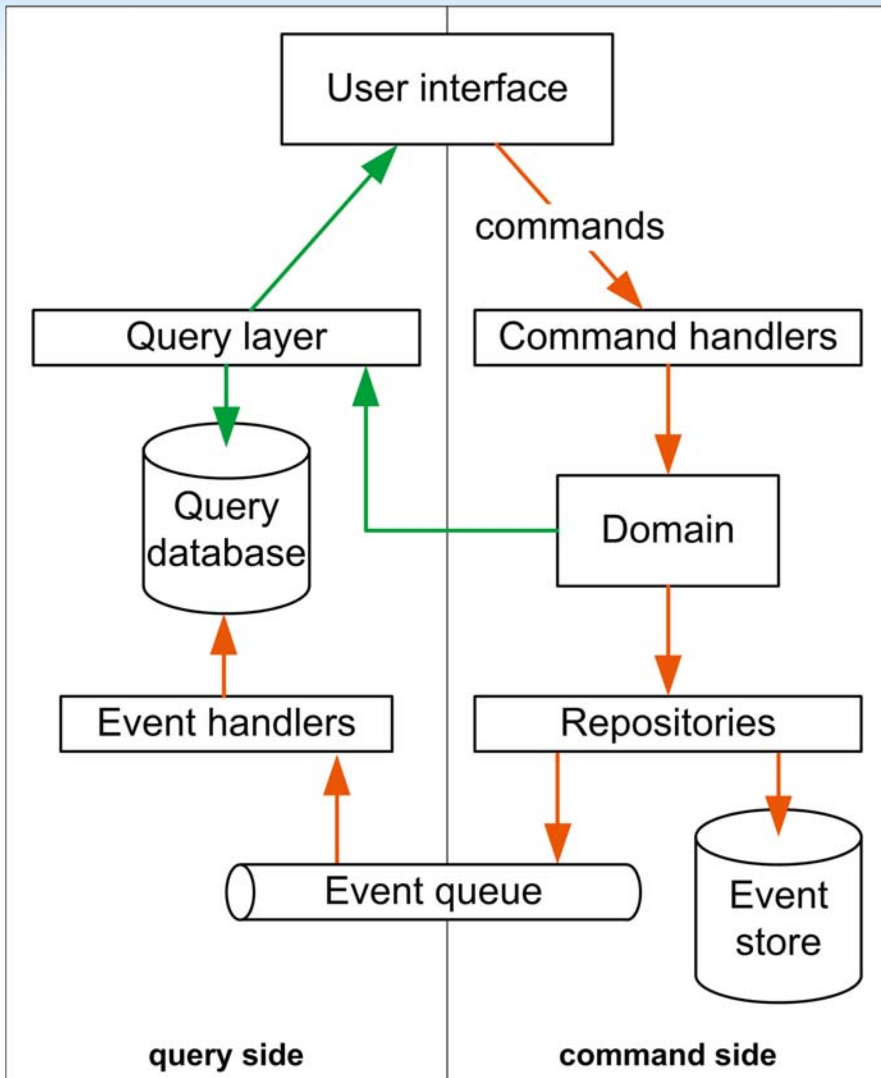
Het combineren van deze twee patterns leidt tot systemen met een hoge beschikbaarheid en schaalbaarheid, eigenschappen die voor een cloud oplossing zeer wenselijk zijn. De bouwstenen van het Windows Azure cloud computing platform van Microsoft lenen zich bij uitstek voor een implementatie van deze patterns. Dit artikel gaat in op deze patterns en laat zien hoe je eenvoudig met behulp van de Windows Azure Tools en SDK deze patterns kan implementeren.

De basis van CQRS is te vinden in het command-query separation principe van Betrand Meyer. Dit principe stelt dat elke methode of een command moet zijn die een wijziging uitvoert, of een query die gegevens teruggeeft aan de aanroeper, maar nooit beide. Een bevraging heeft op deze manier nooit ongewenste neveneffecten.
CQRS neemt dit principe als uitgangspunt, maar past het op macroniveau toe: niet op het niveau van methoden binnen klasses, maar de volledige softwarearchitectuur wordt opgedeeld in een lees- en een schrijfdeel. Zelfs de gegevens die worden gebruikt zijn gesplitst in een deel dat bestaat ten behoeve van bevragingen en een deel dat is toegewijd aan bewerkingen.
Dit is radicaal anders dan de klassieke 3-lagen architectuur waar de meeste ontwikkelaars mee groot zijn gebracht. In een dergelijke architectuur worden een presentatielaag, een laag voor de business logica en een laag gericht op de opslag van gegevens die meestal in contact staat met een database onderscheiden. In een 3-lagen architectuur wordt geen onderscheid gemaakt in commands en queries, waardoor het niet eenvoudig is de code of de gegevensopslag voor één van beide te optimaliseren.

In figuur 1 wordt een overzicht van de componenten in de CQRS architectuur gegeven. Er bestaat een aantal variaties op dit overzicht, maar op hoofdlijnen zijn dit de belangrijkste componenten. De command handlers ontvangen de commands vanuit de user interface, en passen deze toe op het domein. Om een command goed te kunnen uitvoeren, kan het zijn dat in het domein ook gegevens moeten worden opgevraagd. Dit kan worden gefaciliteerd door het domein informatie uit de datastore aan de commandzijde te laten ophalen, maar eventueel ook door het domein de relevante data uit de querykant op te laten vragen. Dit alternatief is een typisch voorbeeld van een variatie op CQRS waar niet iedereen het over eens is.

Uiteindelijk is het command goed verwerkt binnen het domein en moet de wijziging worden opgeslagen. Dit is het punt waarop het grootste verschil met een klassieke architectuur naar voren komt: afhankelijk van de precieze implementatie worden de wijzigingen vastgelegd in een opslagstructuur die het dichtst bij het transactionele domein ligt. Daarnaast worden er events afgevuurd met als expliciet doel om de queryzijde op de hoogte te brengen van de doorgevoerde wijzigingen. Dit gebeurt door events te publiceren, die de bevragingskant asynchroon zal oppakken en verwerken in een gegevensstructuur die in niets op die van de commandzijde hoeft te lijken.
De gegevensstructuur aan de queryzijde is namelijk volledig geoptimaliseerd voor bevragingen. Dit kan betekenen dat de informatie aan de queryzijde flink gedenormaliseerd wordt opgeslagen wanneer dat de uitvoering van de queries ten goede komt. Dat informatie aan deze kant redundant wordt opgeslagen is helemaal niet erg, omdat de command zijde uiteindelijk fungeert als een single point of truth. Sterker nog, informatie kan zelfs redundant voor verschillende doelgroepen met uiteenlopende informatiebehoeften worden vastgelegd.

Een simpel voorbeeld waar dit bruikbaar kan zijn: in veel grote webshops staat vaak op een artikelpagina ook een lijstje als "klanten die dit kochten, kochten ook …". Dat is een typisch voorbeeld van een bevraging die in een zuiver transactioneel systeem lastig is, omdat het continu raadplegen van alle ooit gedane orders een zware wissel op het systeem kan trekken. In een CQRS oplossing kan het event KlantHeeftArtikelGekocht eenvoudig ook een redundante tabel met gerelateerde artikelen bijwerken.




FIGUUR 1. DE COMPONENTEN IN DE CQRS ARCHITECTUUR.

Event sourcing wordt vaak in één adem met CQRS genoemd, maar is een op zichzelf staand concept. In een traditioneel systeem wordt veelal de huidige toestand van het systeem vastgelegd. In de beschrijving van CQRS hierboven wordt impliciet aangenomen dat de commandzijde een vastlegging van de huidige state van de objecten in het domeinmodel bevat, zoals men dat ook van traditionele systemen gewend is. Event sourcing houdt in dat deze state opslag wordt vervangen door een event store waarin alle gebeurtenissen worden opgeslagen die ooit op een object van toepassing zijn geweest. Deze events kunnen gebruikt worden om een object terug te halen naar zijn huidige toestand (state). Event sourcing past van nature goed bij CQRS omdat het een architectuur is waarbij events centraal staan. Bovendien bevat de architectuur al een vastlegging van de huidige stand van zaken in de vorm van de opslag aan de queryzijde.

Doordat een event store alle gebeurtenissen in het systeem vastlegt, ontstaat ineens ook de mogelijkheid om achteraf inzicht te verschaffen in het gedrag van het systeem. Zo is het kinderlijk eenvoudig om uit een event store uit te lezen hoe vaak gebruikers in een bepaalde periode hebben geprobeerd hun wachtwoord te resetten. In een klassiek systeem zou je daarvoor eerst historietabellen moeten toevoegen, en pas vanaf dat moment kan je dergelijke gegevens gaan verzamelen. Een ander groot voordeel is dat het met event sourcing mogelijk wordt om alle events die ooit in het systeem zijn afgegaan opnieuw af te spelen bij wijze van regressietest. Uiteraard stelt dat wel eisen aan het correct kunnen omgaan met eerdere versies van events.

## CQRS en Windows Azure

Vanuit een functioneel oogpunt is de match tussen CQRS en cloud logisch: CQRS biedt een architectuur voor schaalbare applicaties terwijl de cloud het platform is voor applicaties die grote hoeveelheden gebruikers en gegevens aan moeten kunnen. Ook in termen van bouwstenen past CQRS goed bij Windows Azure: zo is er behoefte aan de mogelijkheid om asynchroon events en mogelijk ook commands te kunnen verwerken. Door gebruik te maken van aparte worker roles in Windows Azure kan software om events te verwerken op een aparte groep machines worden uitgevoerd, los van de web roles die belast zijn met het tonen van de informatie uit de queryzijde. Om de worker roles van input te voorzien, kan gebruik gemaakt worden van Azure Storage Queues, die door web roles gevuld en door worker roles uitgelezen kunnen worden.

Om gegevens op te slaan, biedt Windows Azure in principe twee mogelijkheden: de relationele database SQL Azure en Azure Storage met daarin Tables, Blobs en – zoals genoemd – Queues. Aan de queryzijde maakt het in feite erg weinig uit hoe de gegevens worden opgeslagen. Om de query layer en de UI zo simpel mogelijk te kunnen ontwikkelen, is SQL Azure aan te bevelen

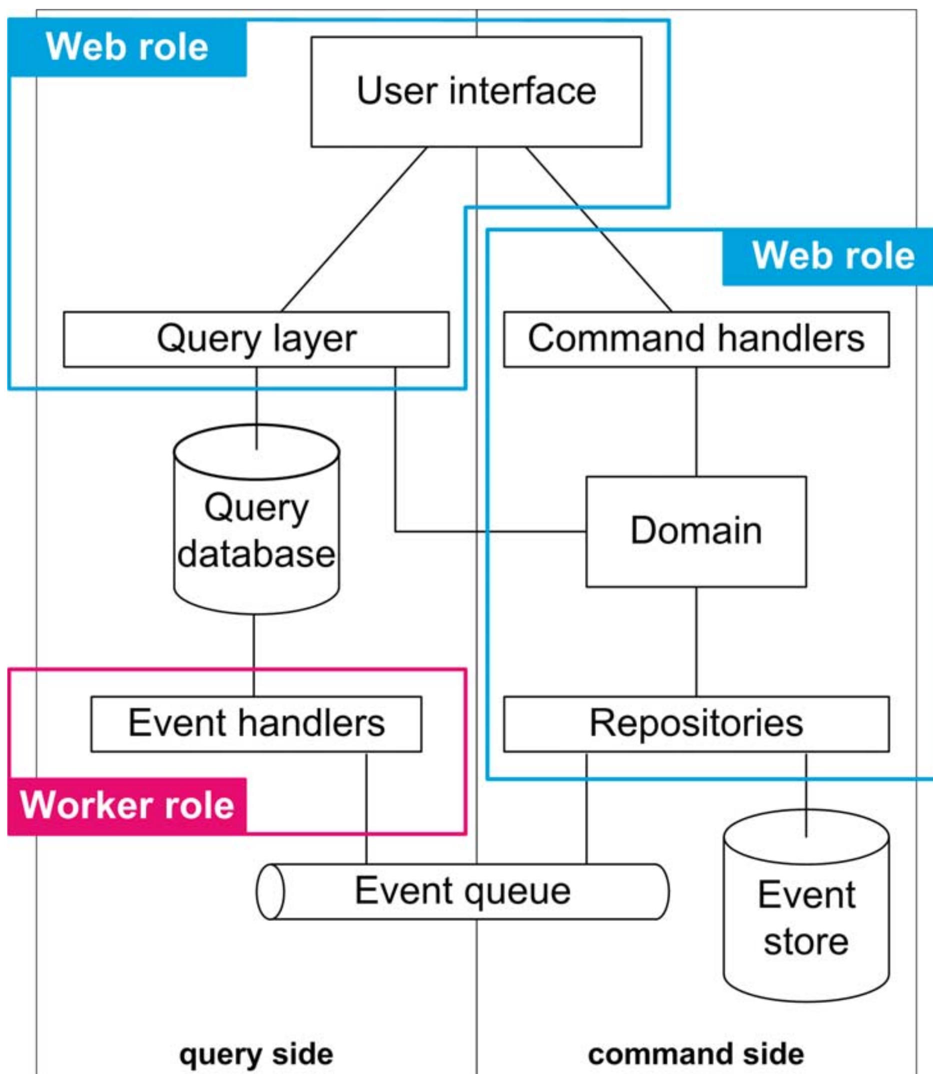
omdat veel standaard tools en O/RM oplossingen gemakkelijker met een relationele database overweg kunnen.
Voor het opslaan van events ligt een relationele database minder voor de hand, omdat hier vooral makkelijk partitioneerbare data zonder onderlinge relaties moet worden opgeslagen. Deze aspecten spreken in het voordeel van Azure Storage Tables. Aan de andere kant is het ook prima om dit wel SQL Azure te doen, dit hangt vooral af van voorkeur en framework.

## CQRS in vier stappen

Om de implementatie van CQRS op Windows Azure te bespreken, volgt hier eerst een beschrijving van de stappen die doorlopen worden bij regulier gebruik van het systeem. In figuur 3 is het verloop weergegeven van het opvragen van informatie (1), het versturen van het bijbehorende command (2), het verwerken van het command in het domein (3) en het daarna afhandelen van het resulterende event in de event store en aan de queryzijde (4).

Als eerste wordt informatie uit de querydatabase gelezen. Zoals eerder aangeven heeft het voordelen om hier gebruik te maken van een relationele database, omdat het daarmee relatief simpel is om een read layer en de bijbehorende UI op te bouwen.
Op basis van de geraadpleegde informatie wordt een command samengesteld. Bij het definiëren van commands is het belangrijk om de intentie van het command te vangen naast de te wijzigen gegevens. Deze intentie werkt vervolgens door in de later te creëren events, waardoor de kwaliteit van de beschikbare gegevens in de event store toeneemt. Deze gedachte staat haaks op wat in CRUD (create/read/update/delete) systemen – systemen die in feite niet veel meer zijn dan een uitgebreide editor op een set tabellen – gangbaar is. Omdat gebruikers inmiddels al heel vaak geconfronteerd zijn met dit type systemen, zijn zij zelf ook geneigd te denken in termen van tabellen. Daardoor gaat de intentie van de handeling verloren: de CRUD handeling BewerkKlant kan allerlei intenties hebben: de klant is verhuisd, hij heeft een hogere kortingsgroep bereikt, etc. Juist deze intentie is van groot belang, omdat daarmee aanvullende informatie over de wijziging op tafel komt.

Nadat het command binnen het domein verwerkt is, wordt via de betreffende repository een set aan events enerzijds aangeboden aan de event store en anderzijds klaargezet op de event bus. Het opslaan van de events in de event store is relatief rechttoe-rechtaan, maar het wegschrijven van de resultaten van een gebeurtenis aan de querykant hoeft dat niet te zijn. De verantwoordelijkheid van de repository eindigt met het opslaan van het event op de event bus, waarna de eerder genoemde worker role één voor één de events verwerkt in de querydatabase. Bij dit verwerken vindt ook de transformatie plaats waarbij de gegevens worden voorbereid op de leesbewerkingen aan de queryzijde.




FIGUUR 2. AZURE ROLES VULLEN DELEN VAN DE CQRS ARCHITECTUUR IN.

**Stap 1: gegevens lezen**
Voor het uitwisselen van gegevens tussen de querydatabase en de UI kan gebruik gemaakt worden van simpele data transfer objecten, objecten zonder noemenswaardig gedrag die louter bestaan om informatie over te brengen. Het verdient aanbeveling het query model (het model van de querydatabase) zo goed mogelijk aan te laten sluiten op de informatiebehoeftes in de UI, wat in het meest extreme geval kan leiden tot een table per view situatie: elke gewenste weergave heeft een eigen tabel die rechtstreeks bevraagd kan worden met een Select * from a where b.
Doordat de queryzijde van de applicatie veel rechtoe-rechtaan hetzelfde is, is werk aan dit deel van de applicatie ook een goede kandidaat om offshore te laten uitvoeren: er is geen sprake van complexe specificaties, sterker nog, het querymodel is al min of meer zelfbeschrijvend. Een database-first Entity Framework model kan hier de nodige versnelling bieden in de implementatie van de read layer.

**Stap 2: commands versturen**
Een commando is een simpel object dat alle informatie bevat die nodig is om de onderliggende actie uit te voeren. Zoals eerder aangegeven ligt in de naam een commando ook de intentie opgesloten, naast de waarden van alle attributen waar het commando betrekking op heeft. Elk commando is een specifieke klasse die aan interface en/of overerving in de volgende stap door de CommandHandler te herkennen is (zie codevoorbeeld 1).

```
public class FinalizeOrder : CommandBase
{
  public string Remarks { get; set; }
  public Guid OrderId { get; private set; }

  public FinalizeOrder(Guid orderId, string remarks)
  {
    OrderId = orderId;
    Remarks = remarks;
  }
}
```

CODEVOORBEELD 1. VOORBEELD VAN EEN COMMAND.

Bij het definiëren van commands is het van belang om dit zo veel mogelijk te doen in termen van eenvoudige common type system typen (integer, string, etc.) en de definitie van een command niet afhankelijk te maken van een read model, domein model of andere commands. Wanneer een command bijvoorbeeld verwijst naar een artikel, dan verdient het aanbeveling naar dit artikel te verwijzen met een simpele ID waarde en die later in het verwerken van het command te controleren.
Een belangrijke discussie die vaak gevoerd wordt bij het ontwerpen van een CQRS oplossing, is welke component nu precies verantwoordelijk is voor de validatie van ingestuurde gegevens en waar het eventueel samenstellen van een voor de gebruiker begrijpelijke foutboodschap dient plaats te vinden. Zaak is om zo veel mogelijk validatie in ieder geval door de UI te laten plaatsvinden, aangezien de impact van het (onderbouwd) moeten weigeren van een command relatief groot is.

**Stap 3: commands verwerken**
Nadat een command vanuit de user interface is verzonden, moet het verwerkt worden. Bij elk command hoort een command handler die dient als entrypoint voor het verwerken van het binnenkomende bericht. Om een handler te vinden bij een command, is een command processor nodig die de juiste handler aanroept. Een simpele command processor kan bijvoorbeeld op basis van reflectie eenmalig alle handlers bij de commands opzoeken en deze in een dictionary bewaren (zie codevoorbeeld 2), maar dit kan ook door een dependency injection container in te zetten.

```
public class CommandProcessor
 {
   private Dictionary<Type, CommandHandler> _commandHandlers;

   public CommandProcessor()
   {
     RegisterCommandHandlers();
   }

   public void ExecuteCommand(CommandBase command)
   {
     CommandHandler handler;
     if ( _commandHandlers.TryGetValue(command.GetType(),
        out handler))
     {
       handler.Execute(command);
     }
   }
   // ...
```

CODEVOORBEELD 2. EEN EENVOUDIGE COMMAND PROCESSOR.

In veel gevallen verdient het aanbeveling om een enkel commando te zien als eenheid van consistentie en daarmee als transactiescope. Er zijn echter ook gevallen denkbaar waarbij meerdere commands gezamenlijk als eenheid verwerkt dienen te worden; in dat geval is het verstandig een unit of work in het leven te roepen die bij het uitvoeren van de individuele commands als context wordt meegegeven. In codevoorbeeld 3 wordt een voorbeeld van een command handler getoond.

```
public class FinalizeOrderHandler :
  CommandHandler<FinalizeOrder>
  {
    public override void Execute(FinalizeOrder command)
    {
      var service = new OrderService();
      service.MarkOrderAsComplete(command.OrderId,
       command.Remarks);
    }
  }
```

CODEVOORBEELD 3. IMPLEMENTATIE VAN EEN COMMAND HANDLER.

De generic type parameter van de commandhandler legt de relatie tussen command en handler vast. Dit wordt door de command processor gebruikt om bij het juiste command de juiste handler te vinden Het verdient aanbeveling om in elk CQRS project een test op te nemen die controleert op commands zonder handlers en vice versa. Hiermee wordt voorkomen dat pas op runtime ontdekt wordt dat er koppelingen tussen commands en handlers ontbreken.
De enige verantwoordelijkheid van de command handler is het op elkaar laten aansluiten van binnenkomende commands en diensten in het domein. Dit blijkt ook uit de implementatie in figuur 6: het enige wat de command handler doet, is een service in het domeinmodel aanroepen. Deze service zorgt er vervolgens via de betreffende repository voor dat het resultaat van dit command als event in de event store wordt opgeslagen.

**Stap 4: events verwerken**
De repository die het event verwerkt in de event store, heeft daarnaast in deze oplossing de verantwoordelijkheid om de queryzijde

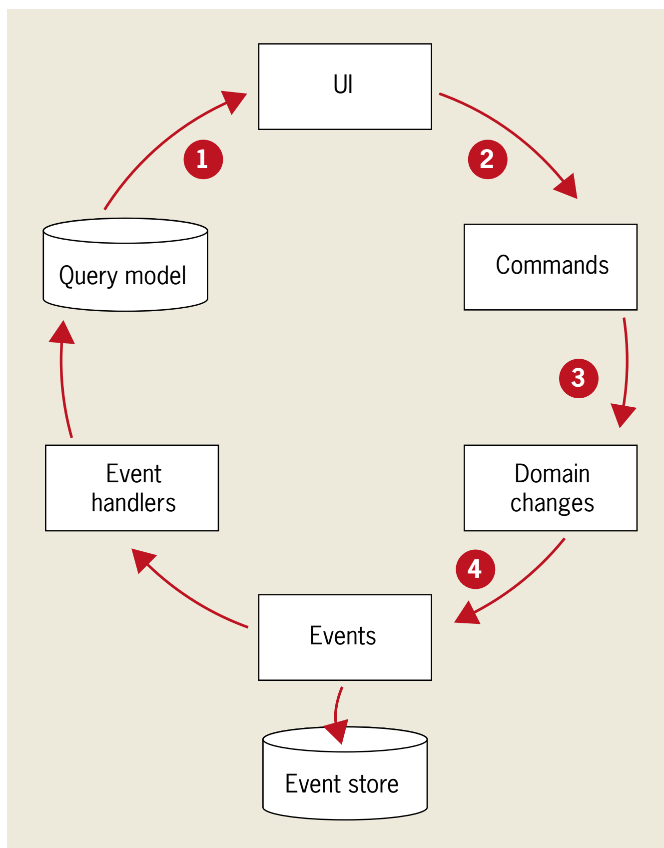
van de oplossing op de hoogte te brengen van het event. De definitie van een event lijkt veel op die van een command, maar de formulering is anders: een command beschrijft altijd iets wat nog moet gaan gebeuren, terwijl een event iets aanduidt wat heeft plaatsgevonden. In dit voorbeeld zou het event bijvoorbeeld OrderIsFinalizedEvent kunnen heten, zodat door de werkwoordsvorm duidelijk wordt dat dit gebeurd is.

Een basisimplementatie van het wegschrijven van events naar een queue is te vinden in codevoorbeeld 4.

```
public abstract class Repository
{
  public void PublishEventToBus(IDomainEvent domainEvent)
  {
    // gebruik een extension method om een byte array te maken
    var eventByteArray = domainEvent.ToByteArray();
    var message = new CloudQueueMessage(eventByteArray);
     var storageAccount = CloudStorageAccount.FromConfigurationS
     etting("DataConnectionString");
    var queueClient = storageAccount.CreateCloudQueueClient();
    CloudQueue queue = queueClient.GetQueueReference
     ("cqrs-event-queue");
    queue.CreateIfNotExist();
    queue.AddMessage(message);
  }
}
```

**CODEVOORBEELD 4. BASISIMPLEMENTATIE PUBLICEREN VAN EEN EVENT NAAR EEN AZURE QUEUE.**

Dit simpele voorbeeld houdt met heel veel zaken geen rekening, waaronder het feit dat een bericht op een Azure Storage Queue niet langer mag zijn dan 8 kB. Als het event groter is (meer data bevat), zou het in Blob storage opgeslagen kunnen worden en als URL naar die plek op de queue gezet kunnen worden.
Een worker role pollt de betreffende event queue en leest eventuele events uit. Op analoge wijze aan commands wordt bij een event uit de queue een event handler gevonden, die verantwoordelijk is voor het bijwerken van de read database (of databases). Eventuele denormalisatie van de gegevens vindt ook hier plaats, waarna de betreffende tabel of tabellen worden bijgewerkt. Met deze stap is het kringetje rond, en kunnen gewijzigde gegevens uitgelezen worden.




**FIGUUR 3. HET VERLOOP VAN QUERIES EN COMMANDS.**

## Conclusie

Het is lastig om binnen de grenzen van één artikel zowel de bredere concepten als CQRS en event sourcing te introduceren en tegelijkertijd de implementatieaspecten op het Windows Azure platform toe te lichten. Gelukkig is er voor de nieuwsgierig gemaakte lezer online veel achtergrondinformatie over beide onderwerpen te vinden. Naast achtergrondinformatie bestaat er ook een aantal frameworks die CQRS op Windows Azure mogelijk maken, zoals Ncqrs en Lokad CQRS. Aspecten die in dit artikel niet naar voren zijn gekomen zijn onder andere de structuur en implementatie van de event store, het correct afhandelen van foutieve commands en events en het afvuren van de commands vanuit de (ASP.NET) UI.

CQRS is zeer geschikt voor cloud toepassingen op Windows Azure vanwege de intrinsieke mogelijkheid om het lees- en schrijfgedeelte van de oplossing apart te schalen. Ook draagt het asynchrone karakter bij aan de schaalbaarheid van het systeem. Andersom biedt Windows Azure standaard componenten die het gemakkelijk maken CQRS te implementeren, zoals de Azure Storage Queues en de worker roles.

Ontegenzeggelijk bevat de CQRS architectuur meer bewegende delen dan een klassieke 3-lagen architectuur. Daarnaast brengt CQRS zijn eigen tekortkomingen mee: niet in alle scenario's is de eventual consistency van de queryzijde wenselijk. Ook is het uitvoeren van validaties waarbij gegevens geraadpleegd moeten worden lastiger; denk hierbij bijvoorbeeld aan de controle op een uniek veld zoals gebruikersnaam.

Omdat er tegenover deze toegevoegde complexiteit wel meerwaarde moet staan, verdient het aanbeveling CQRS alleen in te zetten als de intrinsieke voordelen benut worden: toegenomen schaalbaarheid, een eenvoudiger te implementeren domein en natuurlijk de functionele voordelen van event sourcing. CQRS is een verfrissende doorbreking van klassieke patronen, maar uiteraard moet de oplossing wel bij het probleem passen.

**Links**

- http://code.google.com/p/lokad-cqrs/
- http://ncqrs.org/

**Tijmen van de Kamp,** is solution architect bij Avanade, een joint venture tussen Accenture en Microsoft. De cloud in het algemeen en Windows Azure in het bijzonder hebben zijn interesse, naast een passie voor kwaliteit en craftsmanship.