Enterprise

De gelaagde architectuur biedt een manier om componenten overzichtelijk te structureren, maar schiet op een aantal andere gebieden ernstig tekort. Vooral wanneer schaalbaarheid en concurrency in een applicatie een rol spelen. CQRS is een architectuurpatroon dat zich richt op deze aspecten. In dit artikel benoem ik deze tekortkomingen en geef ik aan hoe CQRS deze adresseert.

Schaalbaarheid en concurrency met CQRS

Meer mogelijkheden door eenvoud

Met groeiende gebruikersgroepen en de toenemende online activiteit van gebruikers wordt van applicaties veel meer verwacht op het gebied van concurrency en schaalbaarheid. Niet alleen de voor de hand liggende voorbeelden als Facebook en Twitter hebben hiermee te maken. Ook applicaties met een selectieve groep gebruikers zullen uiteindelijk de problemen van gelijktijdig gebruik ondervinden wanneer zij met dezelfde data werken.

Tekortkomingen

De gelaagde architectuur is een veelgebruikt architectuurpatroon in hedendaagse (web)applicaties. Hierin worden verschillende verantwoordelijkheden verdeeld over verschillende componenten, waartussen duidelijke afhankelijkheden gedefinieerd zijn. Centraal hierin staat het domeinmodel, waarin in vele gevallen slechts de gegevens gemodelleerd zijn.

Alhoewel de gelaagde architectuur helpt de complexiteit van een applicatie te beheersen en de koppeling tussen componenten probeert te verzachten, heeft het een aantal fundamentele zwakheden. Deze kunnen in veeleisende applicaties tot vervelende problemen leiden. Er zijn vijf zwakheden die in meer of mindere mate een rol spelen. Dit hangt af van de manier waarop de architectuur is geïmplementeerd en de applicatie waarin het is gebruikt.

Suboptimaal domeinmodel

Aan de basis van veel applicaties staat het domeinmodel. In sommige gevallen wordt er veel aandacht aan besteed, in andere gevallen heel weinig. Het domeinmodel vormt het hart van de applicatie. Als dat hart niet naar behoren functioneert, is het een broeinest van problemen in de gehele applicatie. Het domeinmodel is in eerste instantie een 'model'. Nu bestaan er vele definities van het woord model, maar ze komen ongeveer op hetzelfde neer: een vereenvoudigde weergave van de werkelijkheid, met als doel bepaalde problemen op te lossen.

In de gelaagde architectuur moet het domeinmodel in nagenoeg alle applicaties meerdere problemen oplossen. Het krijgt te maken met verzoeken tot wijzigen van gegevens, maar het moet ook informatie verschaffen aan vaak meer dan één doelgroep. Zo heeft een klant andere informatiebehoeften dan backoffice medewerkers en hebben managers een onstilbare behoefte aan rapportages. Om in al deze informatiebehoeften te voorzien met slechts één domeinmodel betekent uiteindelijk dat het model suboptimaal is voor ieder van deze individuele toepassingen. Ieder van deze toepassingen vraagt om een eigen model, die vervolgens door een of ander mechanisme onderling consistent gehouden worden.

Groeiende complexiteit

Het principe van het lagenmodel is dat componenten van elkaar gebruik maken zonder op de hoogte te zijn van elkaars implementatiedetails. Maar alhoewel een servicelaag niet weet hoe een handeling als het opslaan van domeinobjecten wordt uitgevoerd, het weet wel dát het gebeurt. Wanneer in applicaties veel requirements als 'wanneer een order wordt bevestigd, moet er een email naar de klant worden verstuurd' voorkomen, heeft dit vaak if-then-else

Allard Buijze
is software architect bij JTeam en oprichter van het Axon Framework, een CQRS framework voor Java.

constructies in de servicelaag tot gevolg. Hiermee neemt de complexiteit gestaag toe. Uiteindelijk leiden dit soort requirements en uitzonderingsgevallen tot een onoverzichtelijk geheel, met als gevolg dat een ontwikkelaar het overzicht verliest. Bovendien heeft complexiteit (gedefinieerd als het aantal logische paden door een stuk code) een nadelige invloed op de testbaarheid van de applicatie.

Geen onderscheid

Wanneer in de applicatie-architectuur geen onderscheid wordt gemaakt tussen verschillende doelgroepen en verschillende soorten handelingen van deze doelgroepen, kan er ook nauwelijks onderscheid gemaakt worden in de daarbij behorende non-functionele eisen. Daar waar rapportages van een manager best een uur of een dag mogen achterlopen, moet de informatie voor de orderpickers in het magazijn in seconden worden bijgewerkt.
In de gelaagde architectuur wordt zelden aandacht besteed aan dit soort verschillende non-functionele eisen, doordat de verschillende verzoeken uiteindelijk door dezelfde componenten worden afgehandeld. Je zou voor rapportages bijvoorbeeld ieder uur een batch-job kunnen draaien, maar dat voegt wederom complexiteit toe aan de architectuur.

Logica ligt bij de eindgebruiker

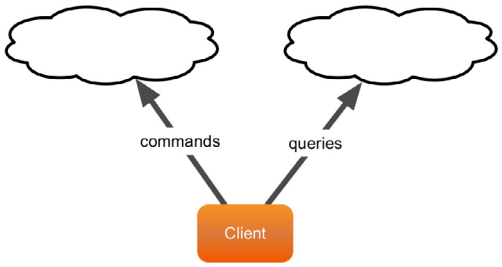
Niet zozeer een directe oorzaak van de gelaagde architectuur, maar wel van de manier waarop het vaak wordt toegepast, is dat veel van de logica bij de eindgebruiker terecht komt, in plaats van in de applicatie zelf. Zo ben ik meer dan eens een applicatie tegengekomen met de volgende flow:

1. De gebruikersinterface-laag haalt een entiteit op bij de service-laag;
2. De service-laag haalt deze entiteit op uit de data-laag;
3. De gebruikersinterface koppelt de entiteit aan een formulier en toont dit aan de gebruiker;
4. De gebruiker wijzigt velden in het formulier en stuurt het formulier op;
5. De informatie uit de formuliervelden wordt in de entiteit geplaatst;
6. De gebruikersinterface slaat de entiteit via de service-laag en data-laag op.

Je kunt je nu afvragen wat nog de functie van de servicelaag is, behalve een doorgeefluik van entiteiten tussen de gebruikersinterface en de datalaag. Vaak is het antwoord: de servicelaag doet validatie; het kijkt of de actie van de gebruiker wel is toegestaan. Maar wat is die actie? We weten niet beter dan dat de gebruiker 'wat velden' heeft aangepast, maar we weten niet precies welke, laat staan waarom. En als een andere gebruiker tegelijkertijd velden in hetzelfde object heeft aangepast zijn de rapen gaar. Dan wordt ofwel de wijziging van de eerste gebruiker overschreven, of krijgt de tweede gebruiker de niet-zo-vriendelijke melding: 'Sorry, uw collega heeft al gegevens voor deze klant aangepast. Probeert u het maar opnieuw.' Heel leuk wanneer die wijziging een verslag van een vervelend telefoongesprek bevat.

Zeer beperkte schaalbaarheid

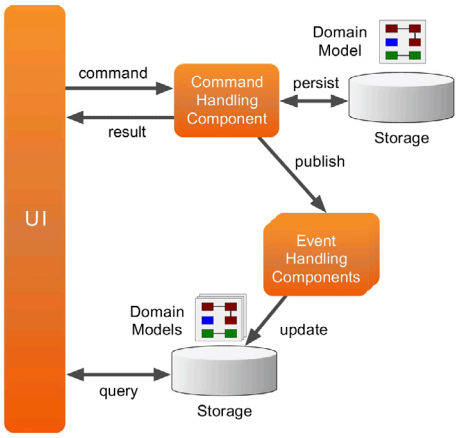
Schaalbaarheid is in webapplicaties een bijna onontkoombaar thema. Met het groeiend gebruik van webapplicaties en de hoger wordende eisen van eindgebruikers is de inzet van meerdere servers meer een zekerheid dan een mogelijkheid.
Aan de basis van schaalbaarheid staat de CAP-theorem [Eric Brewer, 2000] die in 2002 ook formeel is bewezen (door Seth Gilbert). De CAP-theorem beschrijft drie eigenschappen die onmogelijk alle drie door een gedistribueerd systeem kunnen worden gegarandeerd: Consistency, Availability en Partition tolerance. Consistency betekent dat twee systemen altijd over dezelfde data beschikken. Wanneer het ene systeem een wijziging ontvangt, wordt het door een ander ook direct verwerkt. Availability houdt in dat een systeem niet afhankelijk is van een ander system voor het beantwoorden van vragen of het doorvoeren van wijzigingen. Indirect heeft dit ook invloed op de beschikbaarheid van een systeem als geheel als een van de onderdelen niet beschikbaar is. Partition tolerance is de mate waarin een systeem kan omgaan met verlies van berichten tussen systemen.
Wanneer en systeem gedistribueerd wordt opgezet (het logische gevolg van schalen), kan een van deze drie eigenschappen dus niet gegarandeerd worden. De gelaagde architectuur stuurt uit zichzelf aan op Consistency, doordat een verzoek procedureel door de lagen heen wordt verwerkt. Wanneer deze lagen ook nog eens over meerdere machines worden verdeeld, komt de Availability van het systeem als geheel in het gedrang.
Het is niet onmogelijk de mate waarin een gelaagde architectuur de garantie van de drie componenten van de CAP-theorem garandeert te beïnvloeden, maar het voegt een hoop complexiteit toe. In een systeem waar de functionele complexiteit al hoog is, kan dit niet anders dan leiden tot een niet te onderhouden systeem.
Wantrouw systemen en schaalbaarheidsoplossingen die beweren 'geen last te hebben' van de CAP-the-



Afbeelding 1: Overzicht puur CQRS.

Wantrouw systemen die van het CAP-theorem geen last zeggen te hebben.

Afbeelding 2: Overzicht CQRS en EDA.



orem of 'er niet aan doen'. In de praktijk betekent dit dat ze alledrie niet worden gegarandeerd en dat het systeem 'spontaan gedrag' gaat vertonen. Er zijn reeds vele uren besteed aan het opnieuw optuigen van systemen die na een crash van een component onbereikbaar waren en daarna in een inconsistente staat weer zijn opgestart.

CQRS – Een alternatief

CQRS, Command and Query Responsibility Segregation, is een architectuurpatroon waarbij onderscheid wordt gemaakt tussen grofweg twee soorten componenten in een applicatie: een die verantwoordelijk is voor het wijzigen van data en een die vragen over data beantwoordt. Het patroon wordt al jaren toegepast, met name in de financiële sector, maar werd pas rond 2009 expliciet benoemd door Greg Young. Sindsdien wordt het steeds vaker in andere sectoren met succes toegepast.

CQRS is toe te passen in een gelaagde architectuur.

Een architectuurplaat van een CQRS-architectuur is bijzonder eenvoudig (zie afbeelding 1). Dat komt vooral doordat CQRS niet meer beschrijft dan het onderscheid tussen opdrachtverwerking en vraagbeantwoording. Het is zelfs mogelijk CQRS toe te passen in een gelaagde architectuur. Het zal duidelijk zijn dat de beperkende eigenschappen van de gelaagde architectuur zich dan nog steeds manifesteren.

CQRS + EDA

CQRS komt vooral tot z'n recht wanneer het wordt toegepast in een Event Driven Architectuur (EDA). In een EDA communiceren componenten indirect met elkaar door het gebruik van een Event: een notificatie van een gebeurtenis. Doordat een versturend component niet weet wie er in zijn events geïnteresseerd is, is er in grote mate sprake van ontkoppeling. Bovendien is het makkelijker om nieuwe componenten toe te voegen die op die events reageren.

Een typische op CQRS en EDA gebaseerde architectuur ziet er uit als in afbeelding 2. De gebruikersinterface stuurt opdrachten naar een opdracht verwerkend component. Dit component bevat de domeinobjecten die de opdrachten valideren en uitvoeren. De structuur van deze domeinobjecten wordt vooral bepaald door hun gedrag. Het modelleerproces bestaat dan voornamelijk uit het beschrijven van scenario's waarin deze objecten worden gebruikt.
Als gevolg van de uitvoering van de opdrachten vinden bepaalde gebeurtenissen plaats, bijvoorbeeld 'OrderBevestigdEvent'. Dit kan er één zijn, maar ook meerdere of zelfs geen enkele. Gebeurtenissen worden altijd uitgedrukt in de verleden tijd. Hiermee wordt benadrukt dat ze hebben plaatsgevonden en niet meer kunnen worden teruggedraaid, behalve wellicht met een corrigerende opdracht.
Deze Events worden op een Event Bus geplaatst en naar de diverse geïnteresseerde Event Handlers gestuurd. Sommige van deze Event Handlers zullen query tabellen bijwerken voor de publieke gebruikersinterface, terwijl andere bijvoorbeeld een email naar de gebruiker sturen of het backoffice systeem bijwerken (zie afbeelding 3).

Wanneer de gebruikersinterface een query doet, gebruikt het de data die in de query-tabellen is weggeschreven. Het datamodel dat hier is gebruikt, kan volledig op de behoefte van de interface worden afgestemd. Dit voorkomt dure queries met slecht onderhoudbare SQL-statements. Als er meerdere interfaces zijn, zijn er ook meerdere (groepen) tabellen, één voor iedere interface of pagina. Dit wordt ook wel table-per-view genoemd.

Schaalbaarheid door ontkoppeling

In een op CQRS en EDA gebaseerde architectuur is het Event het mechanisme waarmee de modellen up-to-date worden gehouden. Hierdoor wordt door middel van een infrastructuurkeuze bepaald of deze wijzigingen volledig synchroon en transactioneel worden uitgevoerd of dat een event middels een Message Broker (bijv. JMS of AMQP) asynchroon naar alle componenten wordt verzonden.
Deze scheiding geeft ook nieuwe mogelijkheden op het gebied van de CAP-theorem. Door events asynchroon te verzenden tussen de componenten kan voor het opdracht verwerkende component een ander CAP-duo worden gekozen dan voor het event verwerkende component. De CAP-theorem zegt namelijk dat een systeem niet alledrie de eigenschappen kan garanderen op hetzelfde moment.
Omdat opdrachtverwerking over het algemeen volledig consistent moet zijn, en partition tolerance vaak geen vereiste is op dit gebied, wordt hier standaard gekozen voor Consistency en Availability.
Bij de query-kant hangt het de CAP-keuze af van het type gegevens die er zijn opgeslagen. In sommige gevallen is het belangrijk dat de gegevens intern consistent zijn, terwijl in andere gevallen Availability en Partition tolerance belangrijk zijn.

Het enige dat hier wordt losgelaten is de consistentie tussen het opdracht verwerkende component (die gegarandeerd up-to-date is) en het query-component. Maar de informatie die het query component verschaft was 'op een bepaald moment in de tijd' een accurate weergave van de staat van de applicatie.

Eventual consistency

Wanneer het van absolute noodzaak is dat query-data volledig up-to-date is met de data in het opdracht verwerkende component, vraag je dan eens af of dat wel echt zo is. Vaak is dat een eis die niet blijft staan zodra er wat kritische vragen worden gesteld. Het uitgangspunt is dan dat een gebruiker accurate en actuele informatie op zijn scherm wil zien. Het slechte nieuws is: dat kan niet. Tegen de tijd dat de informatie op een scherm wordt weergegeven, kan geenszins worden gegarandeerd dat deze data nog accuraat is. Wellicht heeft iemand anders net een opdracht verzonden en is de data achterhaald. Bovendien blijft data enkele seconden op een scherm staan voordat de gebruiker een beslissing neemt.

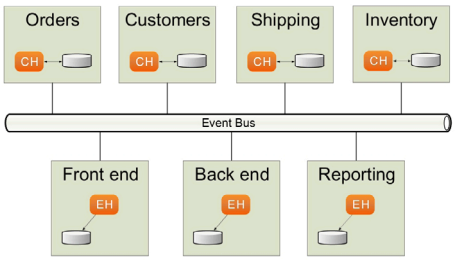
Dus hoe erg is het als deze informatie een vertraging heeft? Uiteraard hangt dat van de vertraging af. Wanneer een systeem niet hoog wordt belast, zal deze vertraging in de orde van grootte van enkele milliseconden liggen. Ter vergelijking, het versturen van een paginaverzoek, het ontvangen en het renderen van een webpagina is honderden malen groter. Het ophalen van www.google.nl, bijvoorbeeld, duurt al 300ms. Stel dat hierop informatie zou staan met een vertraging van 5ms. Dan hebben we het over een toevoeging van slechts 1,6% aan de ouderdom van de data. Niet echt een noemenswaardige toevoeging als dit betekent dat je lineair kunt schalen bij groeiende bezoekersaantallen.

Automatisch conflictoplossing

Een architectuur gebaseerd op CQRS en EDA biedt naast schaalbaarheid nog een groot voordeel. Het systeem kan door events te bewaren bepalen of een actie van een gebruiker conflicteert met concurrente acties van anderen. Aan de hand van deze informatie kan het systeem bepalen of de gebruiker van het conflict op de hoogte moet worden gebracht, of dat het zelf een oplossing vindt. Belangrijk hierbij is dat opdrachten en gebeurtenissen niet alleen een wijziging van gegevens uitdrukken, maar ook de intentie van de gebruiker. Juist deze intentie gaat met data versionering oplossingen als Hibernate Envers verloren.

Conclusie

De gelaagde architectuur maakt het lastig om te gaan met veel van de eisen die aan applicaties worden gesteld. Het gevolg hiervan is vaak uit de hand lopende complexiteit om alsnog aan deze eisen te kunnen voldoen.



Afbeelding 3: Schaalbaarheid.

CQRS geeft een fundamenteel andere kijk op architectuur, waardoor gemakkelijker kan worden voorzien in eisen die te maken hebben met schaalbaarheid, complexiteit van het domein en concurrent gebruik van data. «

CQRS in de praktijk

CQRS heeft z'n oorsprong in de financiële sector. Vooral in applicaties waar de verwerkingssnelheid van opdrachten cruciaal is, zoals aandelenmarkt-analyse applicaties (automated traders), wordt veel voordeel behaald door een apart model voor opdrachtverwerking te gebruiken. Maar ook de banken gebruiken op CQRS gebaseerde architecturen: het opvragen van het saldo op een rekening wordt door een ander component verwerkt dan binnenkomende betalingsopdrachten.

Inmiddels worden de voordelen van CQRS ook in andere sectoren ontdekt. In de medische sector levert de audit trail die event sourcing achterlaat veel waarde. Bovendien wordt de informatie gestructureerd naar de behoefte van de gebruiker, waardoor performance toeneemt.

In online spellen is CQRS gebruikt om zetten van spelers te valideren en de uitkomsten te beoordelen. Hierin worden de gegenereerde events soms direct naar de spelers gestuurd, zodat die realtime de gebeurtenissen kunnen weergeven.

Axon Framework

Bij het implementeren van een op CQRS en EDA gebaseerde architectuur worden veel generieke bouwblokken gebruikt. Zo is er altijd een Event Bus voor het transporteren van Events tussen componenten, en een Command Bus voor het versturen van opdrachten en het ontvangen van het resultaat.

Uit deze bouwblokken is het Axon Framework ontstaan, een open source framework dat Java ontwikkelaars helpt bij het opzetten van een CQRS en EDA architectuur. Zo biedt Axon bijvoorbeeld infrastructuur componenten die het makkelijk maken om automatische conflictdetectie en Event Sourcing te implementeren. Axon biedt veel standaardcomponenten die gemakkelijk kunnen worden aangepast of uitgebreid naar de specifieke wensen van iedere applicatie.

Axon Framework wordt inmiddels in meerdere landen en in meerdere domeinen in productie gebruikt.

De voordelen van CQRS worden ook in andere dan de financiële sector ontdekt.

Meer informatie

- www.cqrsinfo.com – Algemene informatie over CQRS
- www.axonframework.org – Java Framework voor CQRS-gebaseerde architecturen
- www.axonframework.org/training - Workshop over het toepassen van CQRS met Axon Framework.