Relationele databases vs. Document databases in de R&R-web applicatie

Afstudeerverslag

Xavyr Rademaker, 11077581

Referaat

Rademaker X.T., afstudeerverslag - Relationele databases vs. Document databases in de R&R-webapplicatie, Veenendaal, Info Support BV, 19 mei 2016.

Dit afstudeerverslag is geschreven in het kader van de afstudeerstage voor de opleiding Informatica aan de Haagse Hogeschool te Den Haag. Hierin worden de werkzaamheden tijdens de afstudeerstage beschreven. Er komen onderwerpen aan bod als document databases, onderzoek, pakketselectie, RavenDB, C#, SpecFlow en UML.

Het verslag behandelt het gehele afstudeerproces vanaf de start van het proces tot en met de laatste sprint.

Descriptoren

- C#
- .NET
- Document databases
- RavenDB
- UML
- SpecFlow
- AngularJS
- Scrum
- Test Driven Development
- Pakketselectie
- Praktijkgericht onderzoek

Voorwoord

Voorafgaand aan mijn afstudeerverslag wil ik een aantal personen/instanties bedanken, namelijk:

- Meneer Mijnarends, voor de begeleiding vanuit school gedurende mijn afstudeerperiode
- Info Support B.V. en de Vries WFM, voor het verstrekken van een stageplaats en opdracht
- Erma van Alebeek, voor de inhoudelijk begeleiding gedurende mijn afstudeerperiode
- Rick Megens, voor het functioneren als opdrachtgever gedurende mijn afstudeerperiode
- Marieke Keurntjes, voor de proces begeleiding gedurende mijn afstudeerperiode
- Oscar Lubbers, voor het functioneren als businessunit manager
- De medewerkers van Info Support B.V. en de Vries WFM voor een gezellige afstudeerperiode en het geven van tips/adviezen van tijd tot tijd
- de mede afstudeerders van Info Support B.V. voor de gezelligheid

Inhoudsopgave

1. Inleiding

In 2004 bracht Google zijn niet relationele database genaamd Big Table uit. Deze database was echter alleen voor intern gebruik bij Google, daar het heel specifiek gebouwd was om een probleem van Google op te lossen. Kort hierna publiceerde Google twee onderzoeksrapporten waarin het Google File System en Google MapReduce werden uitgelegd.[1]

Met behulp van deze papers bracht Yahoo in 2008 Hadoop op de markt (het prototype was klaar in 2006). Het uitbrengen van Hadoop zorgde voor een hype. Al snel kwamen er meer en meer niet relationele databases op de markt, waaronder databases die data opslaan aan de hand van een key en een waarde (key-value stores), databases die data opslaan in kolommen in plaats van rijen (column stores), databases die data opslaan in de vorm van grafen (graph databases) en databases die data opslaan in de vorm van documenten (document databases). Al deze vormen van databases hebben zo hun voor- en nadelen ten opzichte van elkaar en ten opzichte van relationele databases.[1]

Het bedrijf de Vries WFM is geïnteresseerd in de laatste vorm (document) van databases voor hun R&R-webapplicatie. Zij vragen zich af of zij voordelen kunnen behalen als zij afstappen van hun relationele database en deze vervangen voor een document database. Ook willen zij weten welke document database dan het best bij hun past. Gedurende mijn afstudeertraject moest ik deze vragen beantwoorden.

Dit document beschrijft de werkzaamheden die ik heb uitgevoerd gedurende mijn afstudeerperiode. Allereerst wordt er verteld wat de context van mijn opdracht was. Onder de context valt het bedrijf waar ik mijn afstudeerstage uitvoerde, mijn werkomgeving en mijn opdracht.

Vervolgens beschrijf ik hoe ik van plan was het project aan te pakken. Hierbij ga ik in op welke mijlpaalproducten er waren, welke ontwikkelmethodiek ik heb gekozen, hoe ik het onderzoek en de pakketselectie van plan was aan te pakken en hoe ik uiteindelijk het Proof of Concept zou gaan bouwen. Per product wordt er ook beschreven waarom ik het op deze manier van plan was aan te pakken. Ook is er in de aanpak een globale planning te vinden, welke laat zien hoe ik vooraf dacht dat het project zou verlopen.

Na de aanpak komen de uitvoeringshoofdstukken. Deze hoofdstukken beschrijven hoe ik de mijlpaalproducten (onderzoek, pakketselectie en bouwen) heb uitgevoerd en waarom ik het op deze manier heb uitgevoerd. Verder zullen er ook relevante tussenresultaten genoemd worden.

Het eerste uitvoeringshoofdstuk gaat over het door mij uitgevoerde onderzoek. Over dit onderzoek vertel ik welke deelvragen er waren, waarom deze deelvragen er waren en hoe ik deze deelvragen heb uitgevoerd. Er zullen in dit document alleen tussenresultaten behandeld worden die nodig zijn om de rest van het document te begrijpen.

Het tweede uitvoeringshoofdstuk gaat over de pakketselectie. Hierover zal ik behandelen welke criteria er gesteld werden aan de database, en hoe de databases zijn gecontroleerd op deze criteria. Ook in dit hoofdstuk geld dat er dieper wordt ingegaan op de interessantere criteria.

Het laatste uitvoeringshoofdstuk behandelt het bouwen. Dit hoofdstuk zal delen van het testen, ontwerpen, de database en de implementatie bevatten. Er wordt per sprint gekeken waar dieper op ingegaan wordt.

Tot slot is er een evaluatie te vinden. Hier zal ik een aantal punten evalueren, namelijk:

- De eindproducten
- Het proces
- De uitgevoerde beroepstaken.

2. Context

Dit hoofdstuk zal de context van mijn afstudeeropdracht beschrijven. Hierin wordt eerst verteld over het bedrijf Info Support B.V. en het bedrijf de Vries WFM, dat onder Info Support international group valt. Vervolgens beschrijf ik de werkomgeving waarin ik mijn afstudeeropdracht heb uitgevoerd. Na de werkomgeving komt een stuk over de opdracht welke ik heb uitgevoerd gedurende mijn afstudeerperiode, de gebruikte tools en de vooraf bekende uitgangspunten/kwaliteitseisen.

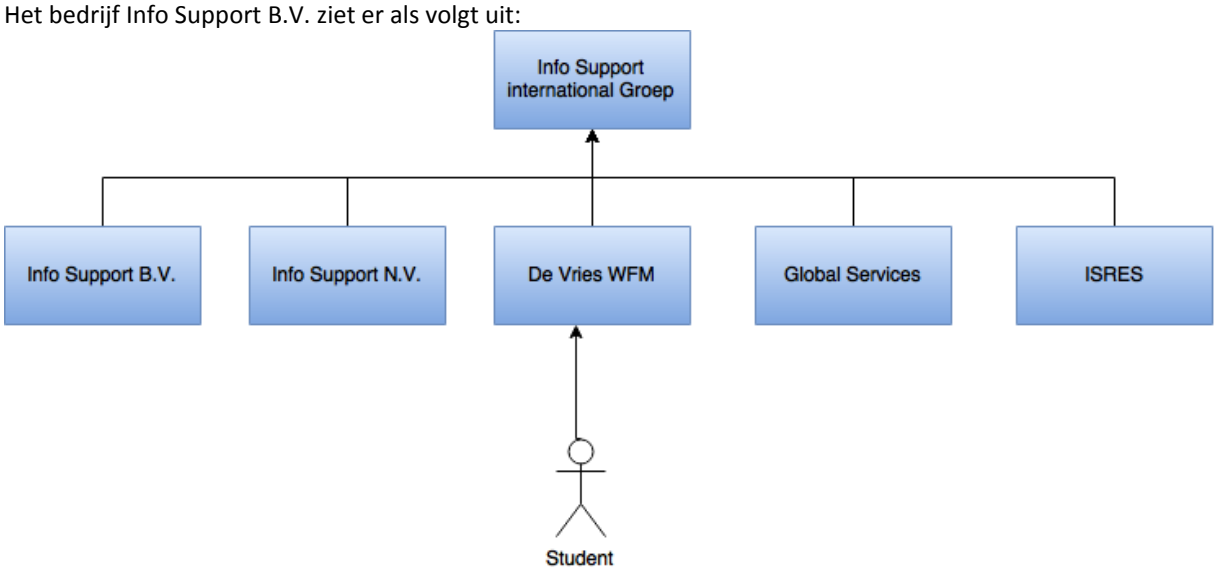
2.1 De organisatie

Info Support B.V. is een bedrijf wat zich onder andere bezighoudt met het ontwikkelen van op maat gemaakte software voor bedrijven in verschillende branches. Deze branches zijn:

- Financiële dienstverlening
- Woningcorporaties
- Overheid
- Zorg en verzekering
- Decentralisaties
- Pensioenfondsen
- Industrie en energie
- Vervoer

Naast het ontwikkelen van software, houdt Info Support B.V. zich ook bezig met het beheren van software en het opleiden van zowel eigen medewerkers als medewerkers van andere bedrijven. Bij het ontwikkelen van de software maakt het bedrijf gebruik van de nieuwste technieken.

Het bedrijf Info Support B.V. ziet er als volgt uit:

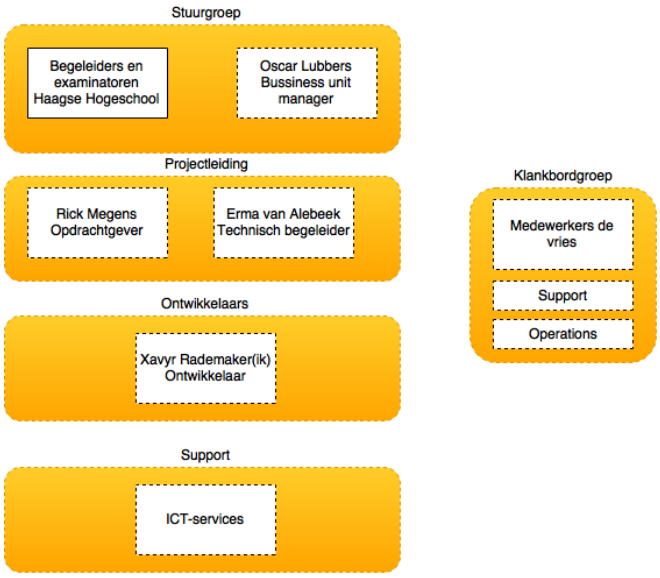


Figuur 1: Organogram Info Support international groep

Een paar jaar geleden heeft Info Support B.V. het bedrijf De Vries Workforce management (WFM) overgenomen. Dit bedrijf bestaat momenteel nog steeds, maar zit nu onder Info Support B.V. Het bedrijf houdt zich bezig met software ontwikkelen welke retailers helpt om hun werknemers kosteneffectief in te roosteren. Het vlaggenschip product van De Vries is de R&R-web (Retail & Resultaat) applicatie. Hierover meer in hoofdstuk 2.2.

Voor het uitvoeren van mijn opdracht is er een kleine organisatie om mij heen gezet. Het doel van deze organisatie is: begeleiden, sturen waar nodig, ondersteunen bij het project en de opdrachtgever die weet wat hij wilt.

Voor mijn opdracht is de volgende organisatie van toepassing:



Figuur 2: Organisatie van mijn project

2.2 R&R-web

De R&R-webapplicatie is een applicatie die winkelmanagers helpt hun medewerkers zo kosteneffectief mogelijk in te roosters. Met kosteneffectief wordt bedoeld: het inroosteren van de juiste mensen op de juiste plaats, tegen de juiste loonkosten op het juiste moment tegen de juiste kostprijs. De applicatie bestaat uit 4 grote modules, hierover meer in paragraaf 4.1.3, en maakt gebruik van een SQL-server 2012 database. Verder is de applicatie geschreven in Silverlight en C#, maar wordt deze sinds kort overgezet naar HTML5, CSS3 en AngularJS, met een C# back-end.

De applicatie wordt als SAAS geleverd bij supermarkt en bouwketens, zoals PLUS, Jumbo en GAMMA. Hierbij hebben alle winkels een eigen database gekregen. Deze databases staan bij de Vries WFM.

2.3 Mijn werkomgeving

Bij aanvang van mijn stage was er voor mij een desktop ingericht waarop Windows 10 draaide. Verder was er ook een bedrijfsemail gemaakt en een inlog account voor de desktop gemaakt. Deze desktop staat in een kamer van de Vries WFM, waar ook het ontwikkelteam van R&R-web zich bevindt. Mijn opdrachtgever bevindt zich ook in deze kamer.

Verder was er voor mij een virtual aangemaakt, waarop Visual studio 2015 geïnstalleerd was. Reden dat deze al geïnstalleerd was, was dat het al vast stond dat ik mijn Proof of Concept ga bouwen in .NET(C#).

2.4 De opdracht

De opdracht is het onderzoeken welke voordelen er te behalen vallen voor de R&R-webapplicatie bij het overstappen naar document databases en welke document database voldoet aan de eisen en wensen van de opdrachtgever. Dit moet allemaal aangetoond worden met een Proof of Concept van de huidige applicatie.

2.4.1 Aanleiding

Het bedrijf de Vries WFM gaat binnenkort hun klantenkring voor de R&R-webapplicatie uitbreiden naar Duitsland. Aangezien zij dan meer gebruikers zullen hebben en met meer data te maken krijgen, twijfelen zij of de huidige SQL-Server database de beste oplossing is. Om hunzelf te kunnen voorbereiden, willen zij weten of document databases wellicht een betere optie zijn, op het gebied van performance en schaalbaarheid.

De medewerkers van de Vries WFM merken dat steeds meer bedrijven/projecten gebruik maken van document databases. Uit de groei in de gebruikers van document databases concluderen zij dat deze bedrijven/projecten voordeel hadden bij het gebruiken van deze vorm van databases.

De medewerkers van de Vries weten echter zelf niet voldoende over deze vorm van databases om te bepalen of zij ook voordelen kunnen halen uit het gebruik van document databases bij de applicatie R&R-web. Het enige wat zij weten is dat deze databases makkelijker te schalen zijn, een hogere performance hebben en dat data in deze databases als losse stukken data wordt opgeslagen. Dit hebben zij echter alleen van horen zeggen.

2.4.2 Probleemstelling

Het probleem waar De Vries WFM tegenaan loopt is dat zij over onvoldoende kennis beschikken over document databases, om te kunnen bepalen of er voordelen te behalen zijn als zij overstappen naar deze vorm van databases. Ook weten zij niet wat de gevolgen van zo een overstap zouden zijn en welke document database het beste bij hun eisen en wensen past.

2.4.3 Doel

Het doel van de opdracht is het onderzoeken welke voordelen er te behalen vallen bij het overstappen naar document database binnen de R&R-webapplicatie. Verder moet er onderzocht worden welke document database het beste past bij de eisen en wensen van de Vries WFM en moet dit aangetoond worden door middel van een Proof of Concept.

2.4.4 Resultaat

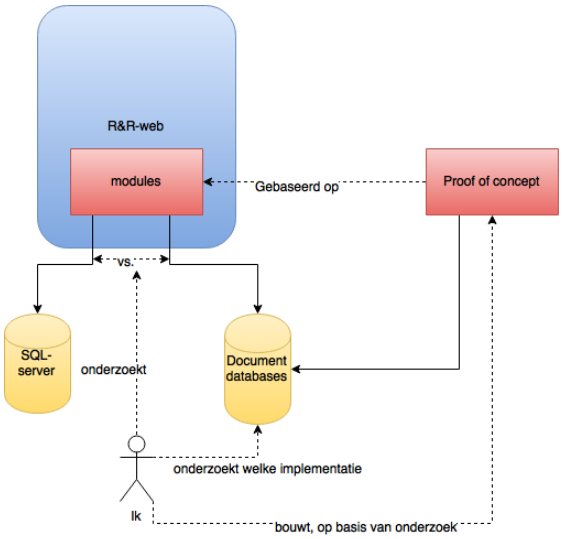
Aan het eind van het project heeft de opdrachtgever een document dat beschrijft in welke modules van de R&R-webapplicatie er voordelen te behalen zijn en wat de gevolgen van de overstap zullen zijn. Ook beschikt de opdrachtgever aan het eind van het project over een document waarin staat welke document database voldoet aan hun eisen en wensen en waarom. Verder beschikt de opdrachtgever aan het einde van het project over een Proof of Concept die de conclusies uit het onderzoek aantoont.

2.4.5 Effect

Aan de hand van de resultaten en het Proof of Concept kan de Vries WFM beslissen of zij wel of niet overstappen op de geselecteerde document database. Mocht de Vries WFM ervoor kiezen over te stappen, dan kunnen zij de overstap beginnen te realiseren. Mocht de Vries WFM ervoor kiezen om niet over te stappen aan de hand van de getrokken conclusies, weten zij dat zij verder moeten zoeken naar een andere oplossing.

2.4.6 Fases

De opdracht bestaat uit 3 fases waarbij elke fase een document/product oplevert. De eerste fase bestaat uit een onderzoek naar de voordelen die er te behalen vallen bij het overstappen naar document databases voor R&R-web. De tweede fase is het onderzoeken welk document database het beste bij de eisen en wensen van de opdrachtgever past en de laatste fase is het bouwen van het Proof of Concept. Deze fases worden weergegeven in figuur 3.



Figuur 3: Focus en fases van het project

2.5 Tools

Naast Visual studio 2015 zijn er nog een aantal andere ontwikkel tools die gebruikt gaan worden gedurende mijn opdracht, namelijk:

- Enterprise architect voor het ontwerpen van mijn Proof of Concept, inclusief het ontwerp van de database
- Visual Paradigm in het geval dat er iets mist in Enterprise architect en voor het maken van diagrammen gedurende de onderzoeksfase.
- SpecFlow voor het vertalen van Gherkin voorbeelden naar unit tests
    - Hierover meer in hoofdstuk 6
- MSTest voor het opstellen van unit tests
- Git voor source code beheer
- JIRA voor Workitem Tracking
- SQL-server voor het analyseren van de huidige applicatie en eventueel om verschillen tussen relationele en document databases aan te tonen
- Eén of meerdere later te bepalen document database(s)

2.6 Uitgangspunten en kwaliteitseisen

Aan het begin van het project zijn er kwaliteitseisen en uitgangspunten opgesteld in samenwerking met de opdrachtgever. Een aantal van deze kwaliteitseisen waren:

- Alle geschreven code wordt gedocumenteerd, inclusief commentaar waar nodig
- De code van de Proof of Concept is getest met een code coverage van minimaal 80%
- Er mag geen functionaliteit uit de R&R-webapplicatie verloren gaan

Een aantal van de opgestelde uitgangspunten waren:

- De nieuwe database moet draaien op Windows Server 2012 R2
- De applicatie is geschreven als een web-API
- De Proof of Concept is gebouwd in C#/.NET

3. Aanpak

Dit hoofdstuk beschrijft welke mijlpaalproducten het project had en hoe ik voorafgaand aan het project van plan was het tot stand komen van deze mijlpaalproducten aan te pakken. Hierbij komen ook de genomen beslissingen en toelichtingen bij deze beslissingen naar voren. Het volledige plan van aanpak is te vinden in bijlage A: Plan van Aanpak.

3.1 Mijlpaalproducten

Om tot het gewenste resultaat te komen, zijn er een aantal mijlpaalproducten/fases opgesteld. Deze zijn:

- **Onderzoek**: bij welke modules van R&R-web zijn er voordelen te behalen bij het overstappen naar document databases zonder dat dit ten koste gaat van de huidige functionaliteit, en welke gevolgen heeft deze overstap?
- **Pakketselectie**, achterhalen welke document database bij de eisen en wensen van de opdrachtgever past
- **Bouwen**
    - Volgens Scrum, sprints van 2 weken
    - Gebruik makend van Test Driven Development(TDD)
    - Meer hierover in paragraaf 3.4.1 en 3.4.2

De aanpak per mijlpaalproduct wordt beschreven in het vervolg van dit hoofdstuk.

3.2 Onderzoek

Het eerste mijlpaalproduct is een onderzoek. In eerste instantie zouden het 2 analyses zijn, 1 over de voor- en nadelen van document databases vs. relationele databases en 1 over de huidige applicatie. Deze analyses zouden achter elkaar uitgevoerd worden. Echter gaf de technisch begeleider aan dat het misschien slim is om de analyses gelijk aan elkaar uit te voeren.

Ik besloot dit advies te volgen, en deels aan te passen, door de analyses niet alleen parallel uit te voeren, maar ook te combineren tot 1 document. Reden dat ik besloot deze samen te voegen is dat de conclusie van dit document dan een stuk interessanter is voor de opdrachtgever. In plaats van de 2 type databases (relationeel en document) tegenover elkaar te zien in 1 document en in een ander document de R&R-webapplicatie in kaart te brengen, krijgt hij nu een document wat de 2 type databases in kaart brengt met betrekking tot R&R-web.

Bij het onderzoek ga ik gebruik maken van hoofd- en deelvragen. Het antwoord op deze vragen zal de opdrachtgever een beeld geven van wat de voor- en nadelen van document databases zijn en wat voor invloed deze voor- en nadelen op de R&R-webapplicatie hebben.

3.3 Pakketselectie

Het 2^{e} mijlpaalproduct is een pakketselectie. Deze selectie moet gaan bepalen welke document database het best past bij de eisen en wensen van de opdrachtgever. Om dit te kunnen bepalen zal een lijst met bestaande document

databases opgesteld worden. Vervolgens ga ik meerdere malen eisen en wensen bij de opdrachtgever ophalen. Door per criteria te kijken welke databases er wel of niet aan voldoen kan ik databases wegstrepen tot er uiteindelijk 1 database overblijft.

3.4 Bouwen

Het laatste mijlpaalproduct is de Proof of Concept (en de shippable products per sprint). Zoals eerder gezegd zal de Proof of Concept ontwikkeld worden volgens Scrum, TDD en getest worden met behulp van SpecFlow. De sprints zullen 2 weken duren. De productbacklog zal voor de eerste sprint gevuld en geprioriteerd worden voor zover de backlog items op dit moment bekend zijn. Zoals in Scrum mogelijk, kunnen de backlog items gedurende het project aangevuld/aangepast worden.

3.4.1 Scrum

Als iteratieve methodiek is er gekozen voor Scrum. Er zijn een aantal redenen dat hiervoor gekozen is. De twee belangrijkste redenen dat er voor Scrum gekozen is zijn:

- Het is iteratief, waardoor er kort cyclisch feedback gegeven kan worden door zowel de product owner (in het vervolg opdrachtgever genoemd) als de technisch begeleider.
    - Dit betekent dat de opdrachtgever direct kan ingrijpen als er niet gebouwd wordt wat hij wilt hebben en de technisch begeleider snel kan ingrijpen als de kwaliteit van de producten niet voldoende is
- De opdrachtgever en technisch begeleider werken al met Scrum.
    - Dit betekent dat er voor hun geen leertijd meer inzit, waardoor zij mij optimaal kunnen begeleiden

Aangezien het Scrum-team alleen uit mij bestaat, zal de uitvoering van deze ontwikkelmethodiek wel aangepast worden. Er zullen wel gewoon sprints, shippable products, een product backlog, sprint backlogs, sprint reviews en sprint retrospectives zijn (samen met de opdrachtgever en eventueel de technisch begeleider). De product/sprint backlogs worden opgesteld in overleg met de opdrachtgever.

Een andere afwijking van Scrum zijn de daily stand-ups. Aangezien deze geen nut hebben in mijn eentje, heb ik besloten daily logs te maken in plaats van de stand-ups. De logs worden aan het eind van elke dag gemaakt als een reflectie op de werkzaamheden van die dag en het plannen van de volgende dag. In deze logs komt in ieder geval het volgende naar voren: wat was ik van plan, wat heb ik gedaan, wat ging goed, wat ging fout, wat is er gedaan om het op te lossen, welke beslissingen zijn er genomen en wat ga ik morgen doen.

3.4.2 Test driven development

Het ontwikkelen van de applicatie zal gebeuren aan de hand van Test Driven Development(TDD). Dit houdt in dat zowel het ontwerpen als het bouwen zal gaan aan de hand van unit tests. De unit tests worden opgesteld op basis van de user story's en de voorbeelden die hierbij horen. Hierbij wordt eerst een kleine test opgesteld, vervolgens wordt er een stukje ontworpen en gebouwd. Het ontwerpen en bouwen wordt zo minimalistisch mogelijk gedaan om de test te laten slagen.

De redenen voor het gebruik van TDD zijn:

- De kwaliteit van de geschreven code wordt gewaarborgd, alle geschreven code is vooraf getest
    - Zo wordt er voldaan aan de minimale 80% code coverage eis.
- Het is een andere manier van ontwikkelen dan ik gewend ben (heb het enkel deels toegepast in een minor), wat het een mogelijk leerzame ervaring maakt.

Verder gaat TDD ook goed samen met iteratieve ontwikkelmethodieken. Omdat je in iteratieve methodieken het geheel opsplitst in kleinere delen (bijvoorbeeld een sprint backlog). Wat je in TDD doet, is een deeltje (een backlog item) nog verder opsplitsen in kleine unit tests. Hierdoor ontwikkel je 1 backlog item iteratief.

SpecFlow:
Om te helpen met TDD zal ik gebruik gaan maken van SpecFlow. Dit is een tool/NuGet package die specificaties (user story's en bijbehorende voorbeelden) omzet in unit tests. Om gebruik te maken van SpecFlow, dienen de user story's aangevuld te worden met scenario's in Gherkin Language. Meer hierover in hoofdstuk 6: Uitvoering bouwen.

3.4.3 Backlogs

Daar de Proof of Concept gebaseerd wordt op een bestaande applicatie, zullen er geen nieuwe backlogs opgesteld worden. Wel zal er vastgesteld worden welke backlogs er in mijn Proof of Concept gebouwd zullen worden en zullen deze backlogs aangevuld moeten worden met examples waar nodig, om deze te kunnen testen met SpecFlow.

3.4.4 Testen

Zoals eerder verteld zal er gebruik gemaakt worden van TDD. Hierdoor zullen de tests opgesteld worden nadat de sprintbacklog klaar is. Hierbij wordt als eerst de SpecFlow test opgesteld (dus 1 scenario). Mocht het scenario groot zijn, wordt er vervolgens een kleinere unit test opgesteld, die een deel van de scenario test. Meer hierover in hoofdstuk 6: uitvoering bouwen.

Voor de unit en integratietests zal gebruik gemaakt worden van MSTest, het test framework van Microsoft.

3.4.5 Ontwerpen

Het ontwerpen gebeurd na het opstellen van de tests, en wordt zo minimaal mogelijk gedaan om de test te laten slagen. Voor het ontwerpen zal gebruik gemaakt worden van UML. De volgende diagrammen zullen gemaakt worden in de sprints:

- Klasse diagram van de applicatie
    - Wordt mogelijk opgedeeld in kleinere diagrammen
- Database diagram
    - Wordt mogelijk opgedeeld in kleinere diagrammen
- Database implementatie model
    - Wordt mogelijk opgedeeld in kleinere modellen
- Sequentie diagrammen

3.4.6 Bouwen

Het bouwen gebeurt na het ontwerp en zal zo minimaal mogelijk uitgevoerd worden om de test te laten slagen (net als het ontwerp). Op het moment dat de test slaagt zal de code refactored worden.

De code van de .Net applicatie zal geschreven worden als een API, gebruik makend van Web Api2 en op de front-end zal AngularJS gebruikt worden, onder andere voor het doen van de requests naar de server.

3.5 Globale planning

Om de mijlpaalproducten/sprints uit te voeren is de volgende planning gemaakt:

Document	Verwachte start week	Verwachte start datum	Verwachte eind week	Verwachte einddatum	Totaal weken
Onderzoek	3	15-2-2016	4	4-3-2016	2
Pakketselectie	5	7-3-2016	8	25-3-2016	4
Bouwen(incl. alle documentatie)	9	28-3-2016	17	27-5-2016	9
Afstudeerverslag	1	5-2-2016	18	3-6-2016	
Afstudeerzitting	-	Eind juni	-	Begin juli	

Tabel 1: Globale planning afstudeerproject

Voor het bouwen zullen er 4 sprints van 2 weken en 1 sprint van 1 week uitgevoerd worden. De laatste sprint is 1 week vanwege de tijd die over is gedurende mijn afstudeerperiode.

4. Uitvoering onderzoek: Voordelen van document databases bij R&R-web

Het eerste mijlpaalproduct was het onderzoek. Zoals eerder gezegd, heb ik voor de uitvoering van het onderzoek gebruik gemaakt van hoofd- en deelvragen. In dit hoofdstuk worden deze hoofd- en deelvragen bekend gemaakt. Ook leg ik per deelvraag uit wat deze bijdraagt aan het beantwoorden van de hoofdvraag.

Nadat alle deelvragen in kaart zijn gebracht, zal ik beschrijven hoe ik de deelvragen heb uitgevoerd en wat de tussenresultaten waren. Al deze tussenresultaten worden vervolgens gecombineerd en verdeeld over verschillende categorieën. Nadat de resultaten in kaart zijn gebracht volgt er een conclusie. De conclusie zal antwoordt geven op de hoofdvraag. Tot slot worden er een aantal gevolgen voor R&R-web genoemd, welke van belang zijn als er wordt overgestapt naar document databases.

Het volledige onderzoek is terug te vinden in Bijlage A: Onderzoek.

4.1 Hoofd- en deelvragen

De hoofd- en deelvragen welke beantwoord moesten worden in het onderzoek staan beschreven in deze paragraaf.

Bij welke modules van R&R-web zijn er voordelen te behalen bij het overstappen naar document databases zonder dat dit ten koste gaat van de huidige functionaliteit, en welke gevolgen heeft deze overstap?

Om deze hoofdvraag te beantwoorden zijn er een aantal deelvragen opgesteld. Hieronder wordt per deelvraag beschreven waarom de deelvraag nodig is om de hoofdvraag te beantwoorden.

4.1.1 Deelvragen

In deze paragraaf wordt beschreven welke deelvragen er waren, en waarom deze deelvragen van belang zijn voor het beantwoorden van de hoofdvraag.

Wat zijn relationele databases? /Wat zijn document databases
Om de relationele databases en de document databases met elkaar te kunnen vergelijken, moet eerst duidelijk zijn wat deze 2 vormen van databases inhouden. Daar zijn de eerste 2 deelvragen voor bedoeld.

Welke modules zijn er in R&R-web? /Welke functionaliteiten zitten er in deze modules?
Omdat de hoofdvraag gaat over welke modules er kunnen overstappen, moeten de modules geïdentificeerd worden. Verder mogen er geen functionaliteiten verloren gaan. Hier kan alleen een uitspraak over gedaan worden als de functionaliteiten bekend zijn.

Welke niet functionele eisen zijn er aan het systeem?
Niet functionele eisen kunnen betrekking hebben op de database (denk aan performance of security). Het zou dus mogelijk zijn dat er voordelen te behalen vallen bij het gebruik van document databases, omdat niet functionele eisen makkelijker of beter gerealiseerd kunnen worden (betere performance of security bijvoorbeeld).

Welke data gebruiken de modules van de applicatie en wat is de structuur van deze data?
In eerste instantie dacht ik dat dit een belangrijke deelvraag zou zijn, omdat het gaat over de vorm van de gebruikte data in de database. Toen ik er eenmaal mee bezig was, kwam ik erachter dat ik de vraag anders had moeten

stellen, namelijk: “Welke tabellen uit de huidige applicatie worden vaker samen opgehaald dan apart?”. Dit is dan ook een nieuwe deelvraag geworden.

Welke tabellen uit de huidige applicatie worden vaker samen opgehaald dan apart?
De reden dat deze deelvraag is toegevoegd, is dat het gaat om genormaliseerde data welke eigenlijk ook embed (meer hierover in paragraaf 4.2) zou kunnen (of misschien zelfs moeten) worden. Daar het in document databases opgeslagen kan worden in 1 document, kan hier mogelijk een voordeel behaald worden.

Welke functionaliteiten zitten er in de database van de huidige applicatie?
Omdat het niet in alle document databases mogelijk is om functionaliteiten in de database te plaatsen (bijvoorbeeld constraints, stored procedures enz.), is het van belang om te weten welke functionaliteiten er in de database zitten. Dit zou namelijk betekenen dat deze functionaliteiten naar de applicatie verplaatst zouden moeten worden.

Welke krachten en zwakheden van relationele databases hebben betrekking op de R&R-webapplicatie? /Welke krachten en zwakheden van document databases hebben betrekking op de R&R-webapplicatie?
Om uiteindelijk te bepalen of er voordelen te behalen zijn, moeten de krachten en zwakheden van beide databases in kaart gebracht worden. Van deze krachten en zwakheden moet gekeken worden welke invloed hebben op de R&R-webapplicatie.

4.2 Uitvoering en resultaten deelvragen

In deze paragraaf leg ik uit hoe ik de deelvragen heb uitgevoerd en wat voor resultaten er hieruit kwamen. Sommige deelvragen worden samen behandeld.

4.2.1 Wat zijn relationele databases?

Voor het beantwoorden van deze vraag heb ik bronnen geraadpleegd en heb ik gebruik gemaakt van kennis die ik heb opgedaan op school.

Uit het bestuderen van de bronnen en de kennis die ik heb opgedaan op school kwamen onder andere de volgende kenmerken van relationele databases:

- Er wordt gebruik gemaakt van Primairy- en Foreign Keys om relaties tussen tabellen aan te tonen
- Data wordt genormaliseerd
    - Alle data uit een tabel die niet bij de primairy key hoort in een andere tabel zetten
    - Dit wordt onder andere gedaan om te vermijden dat data op meerdere plaatsen wordt opgeslagen
- Data uit meerdere tabellen wordt gecombineerd door middel van joins
- Er wordt vooraf een schema vastgesteld
- Er kunnen constraints op data gezet worden
- Er is een standaard query taal(SQL)

4.2.2 Wat zijn document databases?

Ook voor deze deelvraag gold dat ik bronnen heb geraadpleegd en gebruik heb gemaakt van kennis die ik heb opgedaan op school.

Uit het bestuderen van de bronnen haalde ik de volgende informatie over document databases:

- Document databases zijn een vorm van NoSQL-databases
- Er wordt vooraf geen schema vastgesteld

- Het is de bedoeling dat documenten op zichzelf alle data bevatten die zij nodig hebben
    - Denormalisatie
- Het is niet gebruikelijk om constraints op de data te zetten
    - De applicatie moet de data valideren voor het wordt opgeslagen
- Er is geen vaste query taal

Naast deze kenmerken zijn er in document databases ook 2 manieren op met relaties om te gaan. Een van deze manieren is het gebruik maken van references. Dit is sterk vergelijkbaar met een foreign key uit relationele databases, maar er zijn geen constraints of controles op deze reference (het is mogelijk dat een reference verwijst naar een document dat al verwijderd is).[2]

Een andere manier om met relaties om te gaan is het embedden van documenten (denormaliseren). Dit houdt in dat je objecten in elkaar opslaat in 1 document. Het voordeel hiervan is dat het document alles bevat wat nodig is, dus dat er zijn geen joins nodig. Het nadeel is dat data wel op meerdere plaatsen opgeslagen wordt, en dus ook op meerdere plaatsen gewijzigd moet worden op het moment dat er een wijziging plaatsvindt.[2]

Hieronder is een voorbeeld van een embedded document te vinden. Dit voorbeeld is in JSON-formaat en laat de relatie tussen een student (“Xavyr Rademaker”) en een docent (“W.F.M. Vries”) zien.

```
{
        Id: 1,
        Naam: “Xavyr Rademaker”,
        Talenten: [“Is heel snel aan de bal”, “Call of Duty: Black Ops 3”],
        Docent: {
                        Id: 4,
                        Naam: “W.F.M. Vries “,
                        In_dienst_sinds: 1-1-1995,
                        Hobby: “Voetbal”
        }
}
```

4.2.3 Welke modules zijn er in de huidige applicatie en wat doen deze? / Welke functionaliteiten zitten er in de modules van de applicatie?

Voor het beantwoorden van deze 2 deelvragen heb ik eerst de website van de Vries WFM bezocht om daar meer informatie over de modules te verkrijgen. Vervolgens ben ik op een testomgeving door de applicatie zelf gaan klikken en heb ik de use cases/user story’s van de Vries WFM erbij gehaald om de functionaliteiten te achterhalen. Tot slot heb ik in samenwerking met de opdrachtgever de gehele lijst met functionaliteiten langsgelopen om de kernfunctionaliteiten te identificeren.

De R&R-webapplicatie bestaat uit de volgende 4 grote modules:

- Prognose
    - Het doen van een voorspelling (hoeveel mensen zullen er waar nodig zijn) voor de geselecteerde week, op basis van historische weken, geplande omzet en de openingstijden van de winkel/afdelingen.
- Rooster
    - Het maken/invullen van de roosters van de afdelingen van een winkel voor een bepaalde week. Het invullen van de diensten wordt door de afdelingsmanager zelf gedaan, niet door het systeem.
    - Bij het invullen van de diensten kan de afdelingsmanager gebruik maken van de geprognotiseerde data om per uur te zien of hij genoeg mensen heeft ingeroosterd.
- Realisatie

    - Het invullen van de daadwerkelijk gemaakte uren. Dit kan door de afdelingsmanager gedaan worden, of kan automatisch gedaan worden door middel van een kloktijden systeem.
    - Hier wordt inzicht geboden wat er voorspeld was, wat er geroosterd was en de daadwerkelijk gemaakte uren waren
- Uitbetaling
    - Uitvoeren van eventuele bedrijfsregels op de uren, beheren van de verlofbalans van medewerkers en exporteren van de gerealiseerde data naar salarissystemen

4.2.4 Welke niet functionele eisen zijn er aan het systeem?

Deze deelvraag is beantwoord door met de opdrachtgever te gaan zitten. De opdrachtgever gaf aan, dat er niet echt harde niet functionele eisen zijn. Hij gaf alleen een voorbeeld over de performance. Dit voorbeeld was: Het is niet zo dat een scherm binnen 0.5 seconden moet inladen, maar als een klant aangeeft dat het sneller moet wordt het sneller gemaakt.

Hoewel er geen harde niet functionele eisen zijn aan de applicatie, zijn er wel een aantal proces gerelateerde eisen aan het systeem en de database. Deze kwamen naar voren in dit gesprek, en zijn meegenomen als criteria in de pakketselectie. Deze eisen zijn terug te vinden in hoofdstuk 5.

4.2.5 Welke data gebruiken de modules van de applicatie en wat is de structuur van deze data? / Welke tabellen uit de huidige applicatie worden vaker samen opgehaald dan dat ze individueel worden opgehaald?

Om deze deelvraag te beantwoorden heb ik in eerste instantie een testdatabase van R&R-web reverse engineered naar een ERD. Vervolgens heb ik deze ERD verdeeld over de verschillende modules. Op dit punt kwam ik erachter dat dit niet de juiste deelvraag was. Vandaar dat ik de nieuwe deelvraag “Welke tabellen uit de huidige applicatie worden vaker samen opgehaald dan dat ze individueel worden opgehaald?” heb opgesteld.

Om de nieuw gestelde deelvraag te beantwoorden, ben ik met de opdrachtgever gaan zitten. Hem stelde ik deze vraag, waarop zijn antwoord was: op de instellingen tabel na, worden alle tabellen met een of meer joins opgehaald. Zo zijn er meerdere tabellen die opgehaald worden aan de hand van een relatie naar organizationlink en periode. Deze 2 tabellen worden overigens ook nooit gewijzigd, maar dienen enkel als filter data (voor de where clauses).

Een ander voorbeeld is het tonen van een rooster. De deel van de data die in het rooster getoond wordt is onderverdeeld in de volgende tabellen:

- EmployeeSchedule
- QualificationSchedule
- RenumerationSchedule
- EmployeeSchoolShift
- EmployeeSchoolPeriod
- WorkPeriod
- Organization
- OrganizationLink
- Employment
- Employee
- Enz.

Omdat de data zo genormaliseerd is vereist het tonen van een rooster vrij veel joins.

4.2.6 Welke functionaliteiten (triggers, stored procedures enz.) zitten er in de database van de huidige applicatie?

Om de functionaliteiten in kaart te brengen heb ik gebruik gemaakt van dezelfde testdatabase als in de vorige deelvragen. In deze database heb ik van de tabellen en functionaliteiten in de 4 tabellen scripts laten genereren (de scripts werden gegenereerd door SQL-Server zelf). Vervolgens heb ik in deze scripts gezocht naar constraints, stored procedures, functies enz.

Hieruit kwamen de volgende resultaten:

- Stored procedures
  - Alle query's die uitgevoerd worden door de applicatie zijn in stored procedures gezet.
- Custom functions
  - Er zijn een aantal functions die gebruikt worden om bijvoorbeeld primairy keys op te halen aan de hand van andere kolommen
- Indexen
  - Er zijn een aantal soorten indexen gebruikt binnen de database, namelijk:
    - Unique
    - Clustered indexes
    - Nonclustered indexes
    - Primairy en foreign keys
- Constraints
  - Een aantal voorkomende constraints zijn:
    - Kolommen waarvan de waarde alleen 'Y' of 'N' mogen zijn
      - Yes en no
    - Kolommen waarvan de waarde beperkt is tot een aantal mogelijkheden
    - Foreign keys
    - Primairy keys
    - Default waardes (huidige datum, 0 enz.)
    - Waarde is groter dan x, kleiner dan x of ligt tussen x en y

Zoals in paragraaf 4.1.2 uitgelegd, is het niet gebruikelijk om constraints in de database te hebben bij document databases. Als de Vries WFM besluit over te stappen naar document databases moeten deze constraints verplaatst worden naar de applicatie.

4.2.7 Welke krachten en zwakheden van relationele databases hebben betrekking op de modules van de huidige applicatie?

Voor het beantwoorden van de laatste 2 deelvragen is weer gebruik gemaakt van bronnen. Bij het bepalen van de invloed op de applicatie is gekeken naar de resultaten van alle deelvragen die betrekking hadden op de applicatie.

Krachten:

- Data wordt 1-malig opgeslagen(normalisatie)[5]
  - Er is data in de applicatie, zoals medewerkersgegevens, die op veel plaatsen voorkomt. Op het moment dat er een fout in deze data zit hoeft dit maar op 1 plaats verbeterd te worden.
- Er kan complex gequeried worden[5]
  - Omdat elke module gebruik maakt van een bepaalde set data (sommige data wordt in meerdere modules gebruikt) is het belangrijk dat de data op verschillende manieren opgehaald kan worden. (De ene keer een join op tabel X en Y en in een andere module een join op tabel X en Z).
- De integriteit en consistentie van de database worden gewaarborgd door het hanteren van ACID

    - Er wordt gegarandeerd dat de data altijd in consistente staat is. Dit is belangrijk omdat de data genormaliseerd is. Neem bijvoorbeeld het voorbeeld over het rooster uit paragraaf 4.1.7. Op het moment dat er een rooster wijziging wordt gedaan welke meerdere tabellen raakt, wil je dat of de gehele wijziging aankomt, of de hele wijziging niet aankomt.

Zwakheden:
- Het is niet goed in staat horizontaal* te schalen
    - Met de uitbreiding naar Duitsland (zie paragraaf 2.4.1) zal er vroeg of laat geschaald moeten worden. Relationele databases schalen niet optimaal horizontaal.
- De data in de database heeft een andere structuur dan de data waar de applicatie mee werkt
    - Dit betekent dat de data in de data laag getransformeerd moet worden naar de objecten die het systeem gebruikt.
- Het joinen van tabellen en plaatsen van constraints op tabellen gaan ten koste van performance
    - Zie voorbeeld rooster in paragraaf 4.1.7
- Normalisatie
    - Veel data die gebruikt wordt in de R&R-webapplicatie hoeft niet genormaliseerd te worden (zie voorbeeld rooster module in paragraaf 4.1.7)

*horizontaal schalen houdt in dat de database(Let op niet de DBMS, maar echt 1 database in de DBMS) verdeeld kan worden over meerdere servers.

De reden dat document databases beter horizontaal schalen dan relationele databases is, dat documenten in document databases geen onderlinge relaties hebben. Omdat elk document een onafhankelijk stuk data is kan dit makkelijker verspreid worden over verschillende servers. Bij relationele databases gaat dit minder eenvoudig omdat het verspreiden van de database over meerdere servers betekent dat joins ook over meerdere servers uitgevoerd gaan moeten worden. Dit kan ten koste gaan van de performance en eventueel van de beschikbaarheid (als 1 server down is, kan de join niet uitgevoerd worden).

4.2.8 Welke krachten en zwakheden van document databases hebben betrekking op de modules van de huidige applicatie?

Krachten:
- Het is eenvoudig te schalen, ook horizontaal[4]
    - De uitbreiding naar Duitsland (zie paragraaf 2.4.1) kan gaan betekenen dat de database server(s) zullen moeten opschalen. Dit kan uiteindelijk in kosten schelen door horizontaal te schalen.
- Het is sneller in zowel het ophalen als opslaan van data[4]
    - Dit komt omdat constraints ontbreken en joins (bijna) niet nodig zijn
- De data wordt op dezelfde manier opgeslagen als hoe de applicatie met de data werkt(denormalisatie)[4]
    - De applicatie hoeft dus niet de data samen te voegen naar de objecten waar mee gewerkt wordt, dit scheelt in code in de data laag. Ook zorgt dit er grotendeels voor dat joinen niet nodig is.
- De meeste implementaties werken met JSON, BSON of XML[4]
    - Dit zijn web vriendelijke formaten en er zijn in .NET standaard serializers dit de formaten omzetten naar objecten.

Zwakheden:
- Niet alle document databases garanderen ACID-transacties[3]
    - Dit betekent dat de applicatie moet controleren of de transactie volledig is aangekomen, en zo niet moet de applicatie de transactie ook terugdraaien. Dit is echter niet altijd mogelijk.
- De applicatie wordt complexer omdat de database als het ware versimpeld is[4]
    - Contraints op data en controles op het type van de data moet op applicatie niveau gebeuren.
- Het kan voorkomen dat data meerdere malen is opgeslagen in de database[4]

- Op het moment dat een medewerker in elk rooster waar die op voorkomt embed wordt en deze medewerker wijzigt zijn naam, moet deze wijziging op alle roosters doorgevoerd worden.

4.3 Resultaten

Na het beantwoorden van de deelvragen heb ik de resultaten samengevoegd. Deze resultaten heb ik onderverdeeld in 3 categorieën, namelijk: R&R-web, relationele databases en document databases. In deze paragraaf is per categorie te zien wat de resultaten waren.

4.3.1 R&R-web

Binnen de modules van R&R-web wordt veel gebruik gemaakt van joins van het uitvoeren van handelingen. Dit komt doordat de database op sommige plaatsen (overbodig) genormaliseerd is. Verder is de output van de ene module (bijvoorbeeld Prognose) de input/een onderdeel van de volgende module (bijvoorbeeld Rooster). Binnen de database van de applicatie zitten onder andere constraints, stored procedures, functies en indexen.

4.3.2 Relationele databases

Binnen relationele databases is het gebruikelijk om data te normaliseren en op te halen door middel van joins. Hierdoor wordt data op 1 plek opgeslagen. In relationele databases wordt vooraf een schema gedefinieerd welke bepaald hoe de data opgeslagen moet worden.

Binnen relationele databases is het mogelijk constraints op data te plaatsen en is ACID gegarandeerd. Ook is het mogelijk complex te query'en op data, wat nodig is omdat data genormaliseerd is.

De zwakheden van relationele databases zijn afgeleid uit de karakteristieken uit deze paragraaf. Doordat tabellen zo afhankelijk van elkaar zijn kunnen relationele databases niet zo eenvoudig horizontaal schalen. Ook gaan joins ten kostte van de performance van de database. Verder kan het voorkomen dat data overbodig genormaliseerd wordt omdat de applicatie de data altijd als geheel gebruikt.

4.3.3 Document databases

In tegenstelling tot relationele databases is het in document databases juist de bedoeling data zoveel mogelijk te denormaliseren. Hierdoor zijn de documenten onafhankelijk van elkaar waardoor horizontaal schalen mogelijk is. Er wordt vooraf ook geen schema vastgelegd, de data wordt opgeslagen zoals het binnenkomt. Denormalisatie betekent echter wel dat data op meerdere malen wordt opgeslagen, en dus ook op meerdere plaatsen onderhouden moet worden.

Verder is het zo dat het in document databases niet mogelijk is constraints of functies op de data te plaatsen, wat ervoor zorgt dat opslaan van data sneller gaat. Het ontbreken van constraints/functies in de database betekent echter dat deze constraints en functies verplaatst moeten worden naar de applicatie.

Een aantal andere eigenschappen van document databases zijn dat er geen schema wordt vastgesteld, de applicatie bepaald hoe de data opgeslagen wordt (de database slaat het letterlijk op zoals het binnenkomt). Ook is ACID niet gegarandeerd binnen document databases.

4.4 Conclusie

Op basis van de resultaten kan ik concluderen dat er voordelen te behalen vallen in alle 4 de modules van R&R-web. De reden dat ik deze mening heb is dat de belangrijkste schermen uit de 4 modules (Vaststellen prognose, Beheren rooster, beheren realisatie en afronden uitbetaling) bestaan uit genormaliseerde data welke prima gedenormaliseerd kunnen worden. Ook is het zo, dat de output van de ene module embed kan worden in de volgende module. Aangezien deze data output is, en het dus onwaarschijnlijk is dat dit nog wijzigt, is het niet erg dat deze data op meerdere plaatsen is opgeslagen. Verder is er een performance te behalen op database niveau, en is het horizontaal schalen wellicht belangrijk voor de uitbreiding naar Duitsland.

4.5 Gevolgen

Het overstappen naar een document database gaat wel een aantal gevolgen met zich meebrengen. Een aantal van deze gevolgen zijn:

- De complexiteit van de applicatie zal omhooggaan
    - De functionaliteiten uit de database (paragraaf 4.2.2) zullen naar de applicatie verplaatst moeten worden.
    - Embedded data betekent dat data op meerdere plaatsen wordt opgeslagen, de applicatie moet er dus voor zorgen dat deze data ook op meerdere plaatsen moet worden geüpdatet (mocht een prognose toch wijzigen als het rooster is gemaakt, moet het ook in het rooster wijzigen).
- De structuur van models in de applicatie gaan bepalen hoe de database eruit zal zien
    - Hier moet rekening gehouden worden met welke data wijzigt vaak en op veel plaatsen, en welke data is vrij statisch (embedding vs. reference).
    - Bijvoorbeeld het referencen van medewerkers data en embedden van periodes.
- Het zal mogelijk worden horizontaal te schalen
    - Wellicht dat dit nodig zal zijn bij de uitbreiding naar Duitsland.
- De data laag gaat aangepast moeten worden
- ACID kan niet gegarandeerd worden
    - Hier zal op applicatieniveau iets voor geschreven moeten worden.

5. Uitvoering pakketselectie

In dit hoofdstuk wordt een deel van mijn werkzaamheden gedurende de pakketselectie beschreven. De volledige versie van de pakketselectie is te vinden in bijlage C: Pakketselectie.

Bij het uitvoeren van de pakketselectie is eerst een initiële lijst opgesteld. Deze lijst is gevormd door lijsten van document databases van Wikipedia en van nosql-databases.org samen te voegen tot 1 grote lijst van 35 document databases. Vervolgens zijn er criteria opgehaald bij de opdrachtgever. Per database moest er gekeken worden of het aan de criteria voldoet. Het meten van de criteria op de databases verliep in 2 fases.

In de eerste fase moest van de initiële lijst een shortlist gemaakt worden. Vanwege de omvang van de initiële lijst had ik besloten de criteria in deze fase te meten door de documentatie van de databases door te lezen. In deze fase viel een database af op het moment dat hij niet aan een criterium voldeed.

Op het moment dat ik nog maar 4 databases overhad startte de 2e fase. In deze fase heb ik de overgebleven databases geïnstalleerd. Het grootste deel van eisen die aan de shortlist gesteld werden heb ik gemeten door zelf te gaan testen met de verschillende databases.

In de 2e fase konden ook geen databases meer afvallen, in plaats daarvan werd er gewerkt met een punten systeem. De databases konden punten verdienen op het moment dat zij aan eisen voldeden. Deze punten zijn na de laatste test bij elkaar opgeteld. De database met de hoogste score past het beste bij de eisen en wensen van de Vries WFM.

In dit hoofdstuk zal ik eerst de initiële eisen in kaart brengen. Vervolgens leg ik per verwerkte criterium uit hoe ik deze heb uitgevoerd en waar nodig geef ik ook aan wat de resultaten hiervan waren.

5.1 Initiële eisen

Voor de uitvoering van de pakketselectie waren vooraf al een aantal eisen bekend (deze kwamen naar voren gedurende het plan van aanpak en het voorgaande onderzoek). Deze eisen en de bron zijn te vinden in de onderstaande tabel:

Criteria	Source	Datum
De database moet op Windows server 2012 R2 draaien.	Opdrachtgever → PVA	3-2-2016
Er moet documentatie beschikbaar zijn van de nieuwe database	Opdrachtgever → onderzoek	16-2-2016
De nieuwe database moet te monitoren zijn	Opdrachtgever → onderzoek	16-2-2016
De nieuwe database moet sneller zijn dan de huidige database	Opdrachtgever → onderzoek	16-2-2016
Het moet mogelijk zijn om op performance te sturen(indexen enz.)	Opdrachtgever → onderzoek	16-2-2016
Het moet mogelijk zijn back-ups te maken van de database	Opdrachtgever → onderzoek	16-2-2016
Het inladen van een back-up mag maximaal 3 uur duren	Opdrachtgever → onderzoek	16-2-2016

De kosten voor de database moeten niet te hoog zijn	Opdrachtgever → onderzoek	16-2-2016
Klanten mogen geen data van elkaar zien (momenteel hebben zij eigen databases)	Opdrachtgever → onderzoek	16-2-2016
Er moeten rechten en rollen toegekend kunnen worden aan gebruikers van de database	Opdrachtgever → onderzoek	16-2-2016
Het is wenselijk dat transacties of volledig, of helemaal niet aankomen in de database	Opdrachtgever → onderzoek	16-2-2016
Het moet mogelijk zijn om vanuit C#/.Net te communiceren met de database	Opdrachtnemer	01-03-2016
Er moeten stored procedures gemaakt kunnen worden in de database	Opdrachtgever → onderzoek	16-2-2016

Tabel 6: Initiële criteria uit plan van aanpak en onderzoek

Een van de criteria (het moet mogelijk zijn om vanuit C#/.Net te communiceren met de database) is door mij toegevoegd. Reden dat ik deze heb toegevoegd is, dat de applicatie die met de database praat is geschreven in C#/.NET. Als het niet mogelijk is vanuit C# te communiceren met de database, heeft de database dus geen meerwaarde. Ik heb deze eis voorgelegd aan de opdrachtgever en hij ging hiermee akkoord.

5.2 Eerste fase: Criteria meten op basis van documentatie van databases

Zoals eerder gezegd heb ik in de eerste fase de criteria gemeten door de documentatie van de databases door te nemen. In deze paragraaf leg ik uit hoe ik bij de criteria te werk ben gegaan.

5.2.1 Verwerking eerste 2 criteria

Na het vaststellen van de initiële lijst ben ik gestart met de volgende 2 eisen:

- De database moet op Windows server 2012 R2 draaien.
- Het moet mogelijk zijn om vanuit C#/.Net te communiceren met de database.

Ik startte met deze 2 eisen omdat de opdrachtgever niks zou hebben aan een database die niet op zijn server draait, of waar de applicatie niet mee zou kunnen communiceren.

Voor het meten van deze 2 eisen ben ik in de documentatie op zoek gegaan naar installatie handleidingen voor Windows, ondersteunde platformen en drivers/libraries om vanuit een C#/.NET applicatie met de database te communiceren.

Na het verwerken van deze eisen bleven er 21 databases over. Ik ben vervolgens met de opdrachtgever gaan zitten om te bepalen welke criteria op dit punt verwerkt zouden worden.

5.2.2 Achterhalen nieuwe criteria/welke criteria te verwerken op dit punt

Uit dit gesprek met de opdrachtgever kwam 1 nieuwe eis, namelijk load balancing. De opdrachtgever wist op dit moment echter nog niet wat hij hier precies mee wilde. Ook zou deze eis nog niet verwerkt worden op dit moment. Verder is er van de al bestaande criteria gekozen welke er verwerkt zullen worden op dit punt. De volgende criteria zouden worden uitgevoerd, in de desbetreffende volgorde:

- Klanten mogen geen data van elkaar zien (momenteel hebben zij een eigen databases)
- Het moet mogelijk zijn back-ups te maken van de database

- Er moeten rechten en rollen toegekend kunnen worden aan gebruikers van de database
- Het is wenselijk dat transacties of volledig, of helemaal niet aankomen in de database
- Er moet documentatie beschikbaar zijn van de nieuwe database

Voordat ik eisen ben gaan verwerken stelde ik voor om de laatste eis als eerst te doen. De reden dat ik dit voorstelde is dat ik merkte dat er veel databases waren overgebleven waarvan er bijna geen documentatie beschikbaar was. De opdrachtgever vond dit goed.

5.2.3 Er moet documentatie beschikbaar zijn van de nieuwe database

Om deze eis specifiek te krijgen heb ik de opdrachtgever gevraagd waar documentatie over beschikbaar moet zijn om aan deze eis te voldoen. Hierop gaf hij aan dat de volgende documentatie beschikbaar moet zijn:

- CRUD-acties
- Performance optimalisatie → wat te doen als de database langzaam wordt
- Hoe wordt de structuur van de database aangepast?
- Installatiehandleiding

Na het verwerken van deze eis bleven er 10 databases over waarvan gezegd kon worden dat het serieuze kandidaten waren. Deze databases waren:

1. BaseX
2. CouchBase Server
3. ElasticSearch
4. eXist
5. CouchDB
6. MarkLogic
7. OrientDB
8. RavenDB
9. ArangoDB
10. MongoDB

5.2.4 Klanten mogen geen data van elkaar zien (elke klant moet een eigen database kunnen krijgen)

Voor deze eis was het van belang dat alle klanten een eigen database moesten kunnen krijgen. De reden dat een eigen database per klant voldeed is dat de opdrachtgever niet wilt dat klant A data te zien krijgt van klant B door een verkeerde where-clausule. Alle overgebleven databases voldeden aan deze eis.

5.2.5 Het moet mogelijk zijn back-ups te maken van de database

Om te achterhalen hoe het back-up en restore proces van database in de verschillende DBMS-en gaat, is de documentatie nog een keer geraadpleegd. Mochten er restricties op het maken van back-ups zijn, bijvoorbeeld de database mag niet live staan tijdens het maken van de back-up, zijn deze restricties met de opdrachtgever besproken. Verder gaf de opdrachtgever het volgende aan:

- De back-ups moeten in te laden zijn op andere machines (de machines draaien wel op hetzelfde OS)
- Het moet mogelijk zijn back-ups te automatiseren. Dit mag eventueel ook via een command line script
- De database mag offline zijn bij het inladen van de back-up, maar dit mag geen uren duren.

De enige database die hier geen documentatie over had was CouchDB. Deze is dus ook afgevallen op dit punt.

5.2.6 Er moeten rechten en rollen toegekend kunnen worden aan gebruikers van de database

Om te controleren of de databases aan deze eis voldoen, heb ik gekeken naar wat voor security de databases bieden. Waar er onduidelijkheden/twijfels waren heb ik de opdrachtgever gevraagd wat zijn mening hierover was.

Er waren 2 twijfelgevallen bij deze eis, namelijk:

- ArangoDB
- Couchbase Server

In beide databases is het zo, dat er alleen admin rechten uitgedeeld kunnen worden. Dit betekent dat de gebruiker alles kan doen binnen de DBMS of geen toegang kan krijgen tot het DBMS. Na overleg met de opdrachtgever is besloten dat deze databases afvallen.

Ook ElasticSearch is afgevallen, omdat er op hun site staat dat zij niet aan security doen.

5.2.7 Het is wenselijk dat transacties of volledig, of helemaal niet aankomen in de database

De opdrachtgever gaf aan dat hij hiermee de volgende situatie bedoelde:

Als ik een document invoer in collection X en in collection Y, waarbij ik beide documenten in 1 transactie invoer. Het document uit collection X heeft een reference naar het document uit collection Y. Beide documenten moeten aankomen in de database, of beide moeten niet toegevoegd worden. Zodat er voorkomen wordt dat er een reference in de database staat die nergens naar verwijst.

Hiervoor is de documentatie van de databases geraadpleegd. Er is binnen deze documentatie gekeken naar de transactie support die de verschillende databases bieden.

Er zijn 2 databases afgevallen na deze eis, namelijk:

- MongoDB
- eXist

De reden dat zij afvielen was als volgt. Exist heeft geen documentatie beschikbaar over transactie support, en was daarmee vrij snel afgevallen. MongoDB garandeert van losse documenten dat zij volledig aankomen, of een rollback krijgen.

Op het moment dat er in 1 transactie 2 documenten worden opgeslagen, betekent dit dat MongoDB van de documenten apart garandeert dat ze aankomen of een rollback krijgen, maar niet van het geheel. Het kan dus voorkomen dat van de 2 documenten uit 1 transactie, 1 document aankomt en de ander een rollback krijgt.

Op dit punt waren er nog maar 4 databases over welke wel aan deze eis voldeden, namelijk BaseX, MarkLogic, OrientDB en RavenDB. Dit is uiteindelijk de shortlist geworden.

5.3 Tweede fase: Testen van databases op shortlist

Zoals eerder gezegd zouden er in de tweede fase van de pakketselectie getest worden met de overgebleven database. Ook vallen er in deze fase geen databases meer af. In plaats daarvan kunnen er punten verdiend worden

als een database aan een eis voldoet. De database met de meeste punten aan het einde is de winnaar en past dus het beste bij de eisen en wensen van de opdrachtgever.

In deze paragraaf bespreek ik welke eisen getest moesten worden op de shortlist, wat de wegingen van deze eisen waren, hoe databases punten konden krijgen en hoe ik de criteria getest heb.

5.3.1 Achterhalen criteria en weging hiervan voor de shortlist

Uit het gesprek met de opdrachtgever kwam naar voren dat hij in ieder geval de volgende eisen nog verwerkt wilde zien worden:

- De nieuwe database moet sneller zijn dan de huidige database
- Het moet mogelijk zijn om op performance te sturen (indexen enz.)
- Load balancing → het moet mogelijk zijn de DBMS op meerdere servers te installeren

Vervolgens zijn we de criteria een weging gaan geven. De volgende wegingen zijn aan de eisen toegekend:

Eis	Weging
De nieuwe database moet sneller zijn dan de huidige database	1
Het moet mogelijk zijn om op performance te sturen(indexen enz.)	3
Load balancing → het moet mogelijk zijn de DBMS op meerdere servers te installeren	2

Tabel 7: Te verwerken criteria shortlist

Per eis zijn er ook punten te verdienen door de databases. De manier van punten verdienen is ook met de opdrachtgever besproken. Hieruit kwam het volgende naar voren:

Het moet mogelijk zijn om op performance te sturen:

Punten	Eis
+1	Data moet terug te vinden zijn op basis van andere attributen dan het ID zonder dat alle documenten nagelopen worden (vergelijkbaar met secundaire indexen)

Tabel 8: Te behalen punten bij eerste criterium

Load balancing. Het moet mogelijk zijn de DBMS op verschillende machines te installeren:

Punten	Eis
+1	Als de DBMS op verschillende machines geïnstalleerd kan worden
+1	Als 1 DB/collection(gegevens van 1 klant) verplaatst kan worden van de DBMS uit de ene machine naar de DBMS op de andere machine
+1	Als replication van een DB/collection mogelijk is op meerdere servers → deze wordt bewaard tot het laatst

Tabel 9: Te behalen punten bij tweede criterium

De nieuwe database moet sneller zijn dan de huidige database:

Punten	Eis
+1	Sneller of even snel als SQL-Server(Select query)
+1	Sneller of even snel als SQL-Server(Insert query)
+1	Het invoeren van grote hoeveelheden documenten in 1x is sneller dan SQL-Server

Tabel 10: Te behalen punten bij derde criterium

5.3.2 Voorbereiding shortlist eisen

Voordat ik met de eisen aan de overgebleven databases aan de slag ging heb ik de 4 overgebleven document databases (BaseX, MarkLogic, OrientDB en RavenDB) op mijn werkstation geïnstalleerd. Hierbij zijn de installatiehandleidingen gebruikt die te vinden zijn in de documentatie van de databases.

5.3.3 Het moet mogelijk zijn om op performance te sturen (indexen enz.)

Zoals eerder verteld, was er 1 ding dat achterhaald/getest moesten worden voor deze eis, namelijk:

- Is data terug te vinden op basis van andere attributen dan het ID, zonder dat alle documenten nagelopen worden?

Om dit te achterhalen heb ik de databases eerst met testdata gevuld. Vervolgens heb ik query's uitgevoerd, indexen geplaatst waar mogelijk en de query's nog een keer uitgevoerd. Bij het uitvoeren van de query's is de cache uitgezet, en is voor en na het plaatsen van de index naar de query stats gekeken. Hierbij zijn onder andere het aantal reads in de gaten gehouden.

5.3.3.1 Vullen testdata

Zoals gezegd ben ik als eerst testdata in de databases gaan zetten. Dit ging om een kleine hoeveelheid data per database (20 personen per database). De reden dat ik zo een klein hoeveelheid data gebruikte is dat het ging om het aantal reads. Op het moment dat er in een query waar 1 resultaat uit moet komen, er ook maar 1 read gedaan wordt is dit dus optimaal.

De personen in de database hadden de volgende attributen: id, naam, hobby. De query's zochten een persoon op basis van de naam attribuut.

5.3.3.2 Testen criterium

Per database kwamen de volgende resultaten naar voren:

BaseX
Het was in BaseX niet duidelijk of alle documenten/elementen werden doorgenomen bij het zoeken naar elementen binnen een document. Dit wordt namelijk niet weergeven. Omdat dit niet duidelijk was heeft BaseX hier geen punten voor gekregen. Het is overigens niet mogelijk om op de XML-elementen indexen te plaatsen.

MarkLogic:
Bij MarkLogic liep ik tegen een probleem aan. Ik had MarkLogic geïnstalleerd, een database opgezet en probeerde via de REST API van MarkLogic te communiceren met de database. Om dit onder de knie te krijgen volgde ik de instructies van MarkLogic op, te vinden op: http://docs.marklogic.com/guide/rest-dev/service#id_20421.

De request die bovenaan de instructies staat deed echter niet wat hij moest doen. Ik kreeg namelijk een http-response 401(Unauthorized) terug, met daaropvolgend een 400(Bad Request). Dit vond ik vreemd aangezien ik de credentials van de admin user gebruikte in de request.

Vervolgens heb ik de volgende stappen ondernomen om het probleem op te lossen:

- Ik heb de admin user expliciet alle rechten toegekend
- Ik heb de security van digest naar Basic authentication gezet
- Ik heb de security op alle poorten uitgeschakeld
- Ik heb een nieuwe 'group' gemaakt (nieuwe instantie op een andere poort)
    - Ook bij deze group heb ik de security uitgeschakeld
- Ik heb de request gedaan via Postman
    - Een applicatie waarmee je requests kan versturen
    - Dit deed ik om te kijken of ik de request misschien verkeerd had opgebouwd
- Ik heb geprobeerd de requests vanuit .NET te verzenden
- Ik heb een mede afstudeerder gevraagd of hij een request door kon krijgen
- Ik heb de request gedaan zonder credentials

Alle bovengenoemde stappen resulteerde in hetzelfde resultaat. Dit heb ik aan de opdrachtgever uitgelegd. De opdrachtgever gaf aan dat hij van mening was dat de database niet mee hoefde, omdat het zo lastig is ermee te communiceren.

OrientDB:
Bij het uitvoeren van de 1^{e} query toonde de query-stats dat er 20 records gelezen waren. Vervolgens is er een property toegevoegd aan de class Person met de naam "name" (een property op het name attribuut). In OrientDB moet namelijk eerst een property gemaakt worden met dezelfde naam als het attribuut van de klasse voordat er een index op geplaatst kan worden. Na het maken van de property heb ik een index gemaakt op deze property.

Tot slot heb ik nog een keer gezocht op de naam van een persoon. Uit de stats bleek ditmaal dat er maar 1 record gelezen was. Hiermee is besloten dat OrientDB de punten voor deze eis heeft gekregen.

RavenDB:
In RavenDB is het niet mogelijk om te zoeken op attributen waar geen index op zit. Dit betekent dat er niet geoptimaliseerd kan worden, omdat de situatie altijd optimaal is. Om dit te bevestigen heb ik een database gemaakt, waarin ik een aantal documenten heb gezet met dezelfde attributen als de database in OrientDB.

Vervolgens heb ik in de database gezocht op de naam "person1". Nadat ik een index gemaakt had op de attribuut name, kreeg ik het juiste resultaat.

Ook RavenDB heeft hiervoor de punten gekregen.

5.3.4 Load balancing → het moet mogelijk zijn de DBMS op meerdere servers te installeren

5.3.4.1 Installeren databases

Voor het testen van de eisen over load balancing zijn de databases nog een keer geïnstalleerd. Ditmaal zijn ze ook op mijn laptop geïnstalleerd. Met het installeren van de databases op zowel mijn laptop als de desktop hebben de databases meteen een punt gekregen voor de eerste sub eis.

5.3.4.2 Overzetten data

Nadat de databases op 2 machines waren geïnstalleerd heb ik back-ups gemaakt van de databases (de databases gebruikt in paragraaf 5.10). Vervolgens heb ik de back-ups ingelezen op mijn laptop. Dit ging goed bij alle databases, wat betekent dat alle databases hier een punt voor hebben verdiend.

5.3.4.3 Replication

Voor de laatste sub eis over replication heb ik de documentatie van de databases doorgenomen. Hierin is gezocht naar instructies over het opzetten van replica's. Zowel OrientDB en RavenDB hebben hier documentatie over en hebben dus een punt gekregen, BaseX niet.

5.3.5 De nieuwe database moet sneller zijn dan de huidige database

Om te achterhalen of de document databases sneller zijn dan de huidige SQL-Server database heb ik besloten de performance te testen. Hiervoor heeft de opdrachtgever mij 2 stored procedures geleverd. Deze stored procedures worden door de applicatie gebruikt om diensten toe te voegen aan een medewerker in een afdelingsrooster, en om alle diensten van alle medewerkers van een afdeling op te halen. Vervolgens heb ik uit een test database in SQL-server alle tabellen, met bijbehorende testdata, gehaald die worden gebruikt in de stored procedures en deze in een aparte database geplaatst.

Nadat ik de SQL-server database had, heb ik alle data uit de gebruikte tabellen overgezet naar de 3 document databases. Deze data migratie is uitgevoerd door middel van een eigen geschreven console applicatie.

Toen de databases eenmaal klaar waren voor het testen, heb ik een console applicatie geschreven die hetzelfde deed als de insert stored procedure. De reden dat ik hier een applicatie voor heb geschreven is, omdat het niet mogelijk is stored procedures aan te maken in alle document databases.

Tot slot heb ik de stored procedures en select query's uitgevoerd en heb ik de applicatie de test meerdere malen laten draaien. Dit leverde per database een aantal resultaten op. De resultaten van de document databases zijn vergeleken met de resultaten van SQL-server. Op basis van deze vergelijking zijn punten toegekend aan de document databases. Let op dat voor elke uitvoering van een test, de cache van de database leeg is gemaakt (waar nodig).

In deze paragraaf leg ik eerst de getest functionaliteiten en de gebruikte tabellen/structuren uit. Vervolgens leg ik kort uit hoe ik de data migratie heb uitgevoerd en wat de uiteindelijke resultaten waren.

5.3.5.1 Geteste functionaliteiten

De opdrachtgever heeft, zoals te zien in hoofdstuk 4.2.9, besloten dat er 1 select, 1 insert en een grote insert getest moesten worden. Voor alle 3 de handelingen heeft de opdrachtgever stored procedures geleverd. De stored procedures deden het volgende:

Handeling	Functionaliteit
Select	Haal alle diensten van 1 medewerker van een afdeling van een winkel op, op basis van een periode, een afdeling, een winkel en een medewerker.
Insert	Voeg alle diensten van 1 medewerker van 1 afdeling toe aan een rooster.
Grote insert	Voeg alle diensten van alle medewerkers van een winkel toe aan een rooster.

Tabel 11: Uit te voeren functionaliteiten performance test

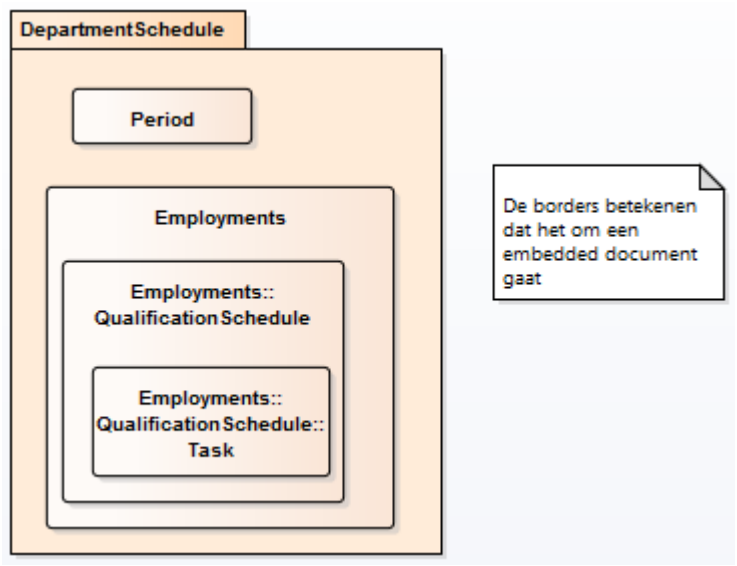
5.3.5.2 Gebruikte tabellen/structuren

Nadat de functionaliteit bekend was, is er bepaald welke tabellen er gebruikt gaan worden in de test. De tabellen en data komen rechtstreeks uit de testdatabase van R&R-web. Dit betekent dat de inhoud en structuur van de database gelijk is aan de daadwerkelijke applicatie. De volgende tabellen, met de volgende hoeveelheid data, zijn gebruikt in de test:

Tabel	Aantal rows
Schedule.EmployeeSchedule	356533
Schedule.QualificationSchedule	537094
Reference.Period	2831
Organization.WorkPeriod	18099
Organization.WorkVersion	84390
Organization.OrganizationLink	171
Organization.Organization	40
Organization.Employment	2963
Organization.Employee	2371
Organization.Qualification	7981
Reference.Task	87

Tabel 12: Te gebruiken tabellen/rows performance test

Voor de document databases is er een extra collection toegevoegd genaamd departmentschedule. Deze had de volgende structuur (alle associaties zijn embed):



Figuur 4: Structuur departmentschedule(alles wordt embed)

Bij het bepalen van deze structuur is rekening gehouden met de volgende punten:

- De gegevens die embed zijn wijzigen niet vaak
- Alle embedded documenten houden een referentie naar het volledige document bij, dit is gedaan om de documenten zo realistisch mogelijk te houden
- Alleen de gegevens die worden gebruikt in de query's worden bijgehouden

5.3.5.3 Migreren data

Nadat de datastructuur en gebruikte data vastgesteld was moest ik de data migreren van de SQL-Server database naar de 3 document databases. Hiervoor heb ik besloten zelf een console applicatie te schrijven die de data uitleest, eventueel transformeert en wegschrijft naar XML-bestanden (voor BaseX), RavenDB en OrientDB.

Voor het inlezen van de data is gebruik gemaakt van NHibernate. De reden dat ik besloot deze ORM te gebruiken is dat ik hier vaker mee gewerkt heb, en dit mij leertijd zou besparen.

Het wegschrijven van de data ging op verschillende manieren. Het wegschrijven naar XML-bestanden deed ik door de XMLSerializer van C# te gebruiken. Deze XML-bestanden laadde ik achteraf in BaseX in omdat dit de eenvoudigste manier van wegschrijven was.

Voor het wegschrijven naar OrientDB werd gebruik gemaakt van de http API van OrientDB. Deze werd aangesproken door middel van http requests via WebRequests. De reden dat er gebruik gemaakt werd van de http API en niet de C# driver van OrientDB, is dat de C# driver niet gebruiksklaar is. Voor het wegschrijven van de data naar RavenDB werd gebruik gemaakt van de C# cliënt driver van RavenDB.

5.3.5.4 Resultaten

In deze paragraaf laat er per database zien wat de resultaten van de performance test waren.

Resultaten SQL-Server:
Het uitvoeren van de stored procedures in SQL-Server gaf de volgende resultaten.

- Select: 2 seconden
- Insert: 26 seconden
- Grote insert: 37 seconden

De reden dat de stored procedures er zo lang over deden komt waarschijnlijk door de constraints die op de tabellen zit en de normalisatie van de database. Dit komt overeen met de theorie uit het onderzoek.

Sneller of even snel als SQL-Server (Select query):
Hieronder wordt beschreven hoe de select query's zijn uitgevoerd op de document databases en wat de resultaten hiervan waren.

BaseX:
BaseX maakt gebruik van de query taal XQuery. Dit is een query taal gemaakt voor het opzoeken van data in XML-documenten.

Uitvoeren van de query (zie paragraaf 5.12.5 voor de query) kostte BaseX 159.87 ms.

Dit betekende dat BaseX sneller is dan SQL-Server als het gaat om het ophalen van alle diensten van alle medewerkers van een afdeling.

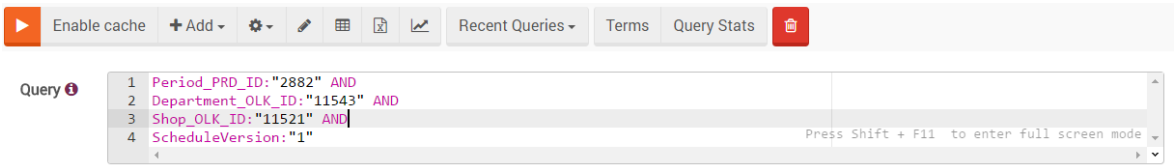
OrientDB:
In OrientDB kunnen query's uitgevoerd worden door middel van een aangepaste versie van SQL. De syntax ziet er hetzelfde uit als SQL, maar er zijn een aantal functionaliteiten toegevoegd. Een van de toegevoegde functionaliteiten is het zoeken op embedded documents.

Het uitvoeren van de select query in OrientDB kostte 599 ms. Dit betekent dat ook OrientDB sneller is dan SQL-Server.

RavenDB:
Zoals eerder gezegd kan er in RavenDB alleen gezocht worden op properties waar een index op zit. Dit betekende dat ik eerst een index moest maken die de volgende properties bevatte:

- Period.PRD_ID
- Shop_OLK_ID
- Department_OLK_ID
- ScheduleVersion

RavenDB maakt gebruikt van de Lucene query syntax, wat inhoud dat je in je query alleen aangeeft welke index je wilt gebruiken en welke combinatie van property en waarde je wilt hebben (vergelijkbaar met de WHERE in SQL). Dit zag er als volgt uit:



Figuur 5: RavenDB, Lucene query syntax

Deze query deed er 194 ms. over om uit te voeren en resultaten te tonen en is daarmee dus ook sneller dan SQL.

Sneller of even snel als SQL-Server (Insert query):

Zoals eerder beschreven is er een console applicatie gemaakt, welke alle diensten van 1 medewerker van 1 afdeling aan een rooster toe moest voegen.

Het uitvoeren van de applicatie leverde de volgende resultaten op:

Database	Resultaat
BaseX	0.74s
OrientDB	5.75s
RavenDB	1.12s

Tabel 13: Resultaten insert query

Dit betekent dat alle databases sneller waren dan SQL-server.

Het invoeren van grote hoeveelheden documenten in 1x is sneller dan SQL-Server:
Na het uitvoeren van de insert query's, moesten alleen de grote inserts nog. Dit leverde de volgende resultaten op:

Database	Resultaat
BaseX	1.19s
OrientDB	7.35s
RavenDB	1.84s

Tabel 14: Resultaten grote insert query

Dat de document databases op alle gebieden sneller zijn dan de SQL-Server database sluit aan op de theorie uit het onderzoek.

5.3.5.5 Leermoment XQuery

Zoals verteld in de vorige paragraaf, heb ik XQuery gebruikt voor het uitvoeren van de select query. In eerste instantie gebruikte ik de volgende query om alle diensten van alle medewerkers in afdeling in een periode op te halen:

```
1. for $DepartmentSchedule
2. in db:open('BaseXFiles')//DepartmentSchedule
3. where $DepartmentSchedule//Shop_OLK_ID=11521 and $DepartmentSchedule//Department_OLK_ID=11543
and
4. $DepartmentSchedule//ScheduleVersion=1
5. for $period in $DepartmentSchedule//Period[PRD_ID=2882]
6. return $DepartmentSchedule
```

Deze query deed er 6903.18 ms. over om deze query uit te voeren en de resultaten te tonen. Hieronder leg ik per line (de regels zijn genummerd) uit wat deze query doet:

1/2. Voor alle afdelingsroosters ($DepartmentSchedule is een variabele) in de database 'BaseXFiles' waar een XML-element 'DepartmentSchedule' voorkomt(//DepartmentSchedule).

3. waar (binnen het XML-element 'DepartmentSchedule' een element 'Shop_OLK_ID' voorkomt met de waarde 11521 en een element 'Department_OLK_ID' voorkomt met de waarde 11543.

4. en een element ScheduleVersion voorkomt met de waarde 1

5. en waar het element PRD_ID voorkomt met de waarde 2882 binnen het element Period

6. retourneer het afdelingsrooster (welke aan de bovenstaande eisen voldoet)

Later kwam ik erachter dat '//' niet geoptimaliseerd is. Wat '//' in query doet, is het nakijken van alle childelementen, dus ook de elementen in elementen (het gaat alle elementen binnen het hoofdelement langs, ook elementen in elementen).

Op het moment dat je de structuur van het XML-document weet, kan er beter gebruik gemaakt worden van '/'. Wat '/' doet, is alleen naar de eerstvolgende laag childelementen kijken. Dit betekent dat er veel minder elementen langsgegaan hoeft te worden.

Nadat ik hierachter kwam heb ik de select query anders opgesteld en uitgevoerd. De nieuwe select query zag er als volgt uit:

```
for $DepartmentSchedule
in db:open('BaseXFiles')/DepartmentSchedules/DepartmentSchedule
where $DepartmentSchedule/Shop_OLK_ID=11521 and $DepartmentSchedule/Department_OLK_ID=11543 and
$DepartmentSchedule/ScheduleVersion=1 and
$DepartmentSchedule/Period/PRD_ID=2882
return $DepartmentSchedule
```

Deze query leverde hetzelfde resultaat op, maar deed er, zoals eerder vermeld, maar 159.87 ms. over om uit te voeren en resultaten te tonen.

5.4 Conclusie

Uit de voorgaande tests zijn de volgende scores gekomen per database:

DBMS → Eis	BaseX	RavenDB	OrientDB
Sturen op performance	0	3	3
Load balancing	4	6	6
Performance vs. SQL Server	3	3	3
Totaal	7	12	12

Tabel 15: Behaalde punten per database

Na afwegen van alle criteria tegenover de databases is bleek dat RavenDB en OrientDB evenveel punten gescoord hadden. Om te bespreken hoe de winnaar bepaald gaat worden is er met de opdrachtgever gezeten. In dit gesprek wilde ik achterhalen hoe wij nu verder zouden gaan.

Naar mijn mening waren er 2 manieren om tot een winnaar te komen, namelijk:

1. We voegen meer criteria toe in de hoop dat 1 van de 2 databases hoger zou scoren dan de ander
2. Ik breng beide databases in kaart en laat de opdrachtgever kiezen welke database hij wilt

In het gesprek met de opdrachtgever gaf hij aan dat hij bij beide mogelijkheden niet zeker weet of er een winnaar uit zou komen. In plaats daarvan vroeg hij mij welke van de 2 databases eenvoudig te leren is voor een ontwikkelaar die er nog nooit mee gewerkt heeft.

Ik gaf aan dat RavenDB het makkelijkst is om mee te werken, en liet de opdrachtgever de code zien die wordt gebruikt om CRUD-acties uit te voeren in beide databases. Daar er een .NET driver beschikbaar is voor RavenDB is het slechts een kwestie van de juiste methodes uit het session object aanroepen om CRUD-acties uit te voeren. De driver van OrientDB daarentegen is nog niet klaar voor productie, en is ook niet goed gedocumenteerd.

Het feit dat de driver van OrientDB niet goed gedocumenteerd is, en dat er minder handelingen nodig zijn om met de database te praten in RavenDB, zorgde ervoor dat de opdrachtgever besloot dat RavenDB de winnaar wordt.

Uit dit gesprek kon ik dus concluderen dat de document database RavenDB het beste past bij de eisen en wensen van de opdrachtgever.

6. Uitvoering bouwen

Vanwege de overgebleven tijd zijn er voor het bouwen 4 sprints van 2 weken uitgevoerd. Dit komt omdat de pakketselectie een week langer duurde dan gepland. In dit hoofdstuk bespreek ik per sprint hoe de uitvoering verliep, wat voor (tussen)resultaten er waren en welke beslissingen er zijn genomen. De volledige functionele en technische ontwerpen zijn te vinden in bijlage D: Functioneel ontwerp en bijlage E: Technisch ontwerp.

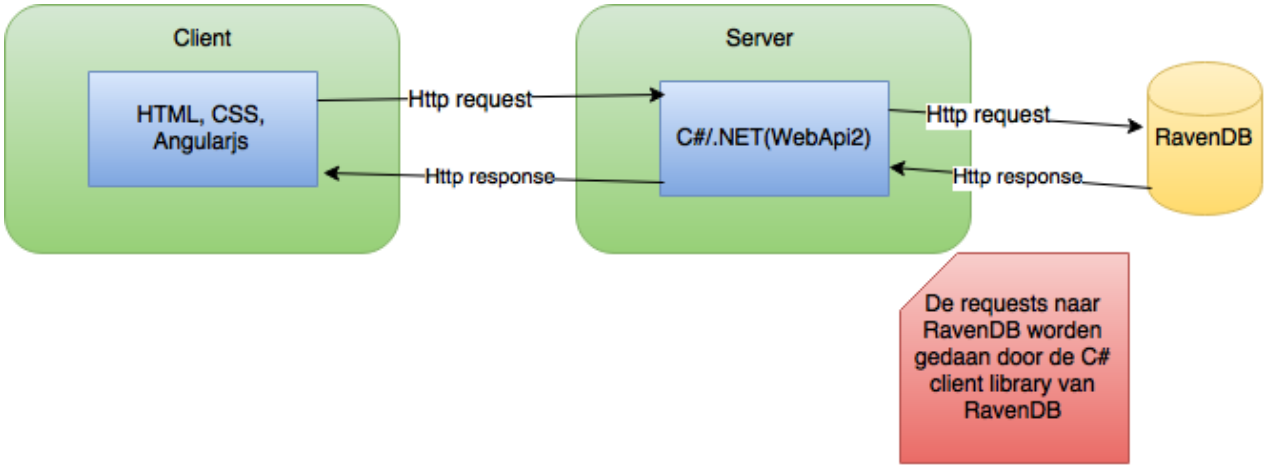
6.1 Voorbereiding bouwen

Voor het bouwen van de Proof of Concept is met de opdrachtgever afgesproken dat van de scratch een webapplicatie ga bouwen. In deze applicatie moeten functionaliteiten uit de R&R-webapplicatie nagebouwd worden, met RavenDB als database. De opdrachtgever zal gedurende de sprint voorbereidingen bepalen welke functionaliteiten er gebouwd moeten worden.

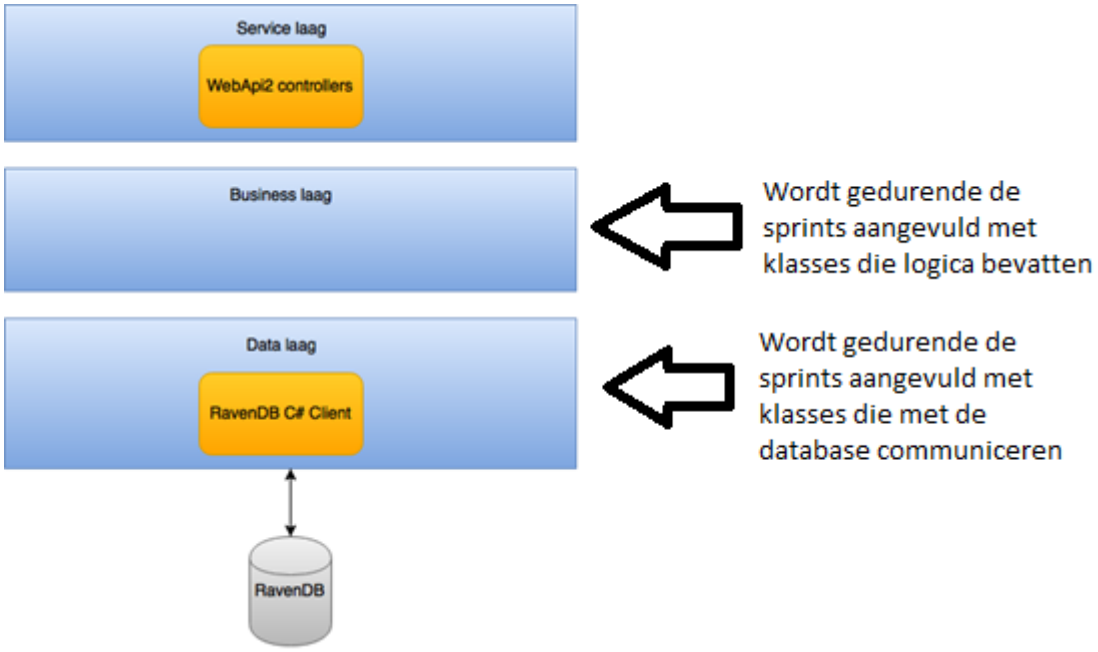
De applicatie wordt een single page application en zal gebruik maken van AngularJS, WebApi2(het wordt een API in C#/.NET) en RavenDB. AngularJS zal in combinatie met HTML en CSS ervoor zorgen dat de front-end dynamisch is en toont wat er getoond moet worden. Verder zullen de requests van de client naar de server gaan via http requests, welke worden uitgevoerd met behulp van de http-service van AngularJS.

Op de server worden de requests ontvangen en beantwoordt door middel van WebApi2 controllers. Verder wordt er vanuit de server gecommuniceerd met de database door gebruik te maken van de C# Library van RavenDB.

De samenwerking tussen AngularJS, WebApi2 en RavenDB is te zien is figuur 6. De structuur van de server is te zien in figuur 7.



Figuur 6: Samenwerking technieken Proof of Concept



Figuur 7: Lagen model van de .Net applicatie op de server van de Proof of Concept

De reden dat er voor deze technieken(.NET, HTML, CSS en AngularJS) is gekozen, is dat de R&R-webapplicatie met dezelfde technieken werkt.

6.2 Sprint 1: Zien en wijzigen van afdelingsroosters m.b.v. RavenDB

Alle backlog items voor de eerste sprint gingen over het tonen en wijzigen van afdelingsroosters. In deze paragraaf zal ik eerst de sprint backlog benoemen. Vervolgens ga ik verder in op het verwerken van de eerste backlog item.

Voor het realiseren van de backlog items heb ik eerst tests opgesteld (vanwege werken met TDD), hierover wordt meer verteld in paragraaf 6.2.2. Parallel aan het bedenken/opstellen van de tests werd de database structuur vastgelegd. In paragraaf 6.2.3 ga ik verder in op de structuur van de database, en de beslissingen die zijn genomen op database niveau.

Op het moment dat de database structuur en de eerste set tests bepaald waren werd er een ontwerp gemaakt. Ook werd er gedurende het ontwikkelen constant refactored. Bij het refactoren zijn een aantal ontwerp beslissingen gemaakt. In de paragrafen 6.2.4 en 6.2.7 komen ontwerp beslissingen naar voren.

Bij het ontwikkelen van de POC zijn ook een aantal C# technieken gebruikt. Een aantal van deze technieken zijn query's (om te communiceren met RavenDB) opstellen met LINQ (paragraaf 2.6.5_ en het gebruik van generics in de datalaag. De generics werden gebruikt om bepaalde query's te generaliseren, hierover meer in paragraaf 6.2.5.

Tot slot geef ik een voorbeeld van het verplaatsen van constraints uit de database naar de applicatie, zoals beschreven in het onderzoek.

6.2.1 Sprint backlog

De volgende backlog items stonden gepland in de eerste sprint.

User story	Module	Story points
Als een afdelingsmanager wil ik diensten kunnen inzien die ik eerder ingevuld heb, zodat de medewerkers weten wanneer ze moeten werken en ik weet dat ik genoeg mensen heb die mijn werk kunnen doen.	Ro	8
Als een afdelingsmanager wil ik diensten aan medewerkers in een afdelingsrooster kunnen toevoegen zodat ik de benodigde mensen op de juiste tijd kan inplannen	Ro	6
Als een afdelingsmanager wil ik medewerkers van andere winkels kunnen inplannen op het moment dat ik zelf medewerkers te kort kom zodat ik genoeg medewerkers kan inplannen	Ro	10
Als afdelingsmanager wil ik geïnformeerd worden van bijzonder dagen zodat ik daar rekening mee kan houden bij het inplannen	Ro/P	4

Tabel 16: Sprint backlog sprint 1

In de paragrafen 6.1.2 t/m 6.1.5 wordt ingegaan op de 1e backlog item (zien was afdelingsroosters).

6.2.2 Testen

De reden dat testen als eerst wordt behandeld, is dat er, zoals in hoofdstuk 3 verteld, gebruik gemaakt wordt van TDD. Dit betekent dat er eerst getest wordt, en daarna pas ontworpen/gecodeerd wordt.

6.2.2.1 SpecFlow

Zoals in de aanpak is genoemd wordt er gebruik gemaakt van SpecFlow voor het testen. De scenario's die getest gaan worden in SpecFlow worden opgesteld in Gherkin Language. Dit houdt in dat er eerst pre-condities worden aangegeven door Given stappen. Vervolgens wordt er een handeling opgesteld met het woord When en tot slot de controle stappen aangegeven door Then. De test in Gherkin Language wordt geschreven in een feature file.

Voor het zien van het afdelingsrooster is er een scenario opgesteld welke het ophalen van 1 enkele afdelingsrooster ophaalde. Hierbij werd het volgende getest: Gegeven(Given) dat er 3 afdelingsroosters in het systeem zitten. Als(When) ik een afdelingsrooster ophaal op basis van een winkel, afdeling en periode. Dan(Then) krijgt ik het juiste afdelingsrooster terug.

In de feature file zag dit er als volgt uit:

```
Feature: SeeDepartmentScheduleFeature
        Als een afdelingsmanager
        wil ik diensten kunnen inzien die ik eerder ingevuld heb,
        zodat de medewerkers weten wanneer ze moeten werken en ik weet dat ik genoeg mensen heb die mijn werk kunnen doen.

@seeDepartmentSchedules
Scenario: See shifts of employees of a single department
Given there are three departmentschedules in the system with the following data
| Id                          | ScheduleVersion | Period.Id     | Period.Week | Period.Year | DepartmentName | ShopName |
| DepartmentSchedules/test2 | 1               | Periods/test1 | Week 1      | 2016        | testVlees      | testPlus |
| DepartmentSchedules/test1 | 1               | Periods/test1 | Week 1      | 2016        | testBrood      | testPlus |
| DepartmentSchedules/test3 | 1               | Periods/test1 | Week 1      | 2016        | testZuivel     | testPlus |

When I retrieve the departmentschedule for department 'testVlees' of the shop 'testPlus' in the period of 'Week 1' 2016

Then the I get the following departmentschedule back:
| Id                          | ScheduleVersion | Period.Id     | Period.Week | Period.Year | DepartmentName | ShopName |
| DepartmentSchedules/test2 | 1               | Periods/test1 | Week 1      | 2016        | testVlees      | testPlus |
```

In de tabel zijn 3 afdelingsroosters te zien die voorafgaand de test bestaan

Na afloop van de test verwacht ik het afdelingsrooster uit laatste tabel terug te zien

Figuur 8: SpecFlow scenario

Vervolgens kan er aan de hand van dit scenario het skelet van een test opgesteld worden. Dit skelet is een .cs bestand, waarin er per stap (Given, When, Then) een methode is. De methodes zijn gebonden met de tekst in de feature file door middel van regular expressions. Ook worden de getallen, strings (tussen ‘’) en tabellen uit de stappen meegegeven als parameters aan de methodes.

De bovenstaande test genereerde het volgende skelet:

```
[Given(@"there are three departmentschedules in the system with the following  data")]
public void GivenThereAreThreeDepartmentschedulesInTheSystemWithTheFollowingData(Table table)
{
    ScenarioContext.Current.Pending();
}

[When(@"I retrieve the departmentschedule for department '(.*)' of the shop '(.*)' in  the period of '(.*)' (.*)")]
public void WhenIRetrieveTheDepartmentscheduleForDepartmentOfTheShopInThePeriodOf(string p0, string p1, string p2, int p3)
{
    ScenarioContext.Current.Pending();
}

[Then(@"the I get the following departmentschedule  back:")]
public void ThenTheIGetTheFollowingDepartmentscheduleBack(Table table)
{
    ScenarioContext.Current.Pending();
}
```

Regular expression

De 3 stappen(given, when, then) in code

Figuur 9: SpecFlow gegenereerde code

Tot slot heb ik het skelet ingevuld. Door alle methodes in te vullen komt er een volwaardige test uit. De uiteindelijke test zag er als volgt uit:

```
[Given(@"there are three departmentschedules in the system with the following data")]
public void GivenThereAreThreeDepartmentschedulesInTheSystemWithTheFollowingData(Table table)
{
        List<DepartmentSchedule> predefinedSchedules = this.TableToDepartmentSchedules(table);
        ScenarioContext.Current.Set<List<DepartmentSchedule>>(predefinedSchedules, "predefinedSchedules");
}

[When(@"I retrieve the departmentschedule for department '(.*)' of the shop '(.*)' in the period of '(.*)' (.*)")]
public void WhenIRetrieveTheDepartmentscheduleForDepartmentOfTheShopInThePeriodOf(string departmentNames, string shopName, string
week, int year)
{
        DepartmentScheduleController.Request = new HttpRequestMessage();
        DepartmentScheduleController.Configuration = new HttpConfiguration();
        List<DepartmentSchedule> depts;

        var response = DepartmentScheduleController.GetDepartmentSchedules(departmentNames, shopName, week, year, 1);
        response.TryGetContentValue<List<DepartmentSchedule>>(out depts);

        this.DepartmentSchedules = depts;
}

[Then(@"the I get the following departmentschedule back:")]
public void ThenTheIGetTheFollowingDepartmentscheduleBack(Table table)
{
        List<DepartmentSchedule> expectedSchedules = this.TableToDepartmentSchedules(table);
        DepartmentSchedule expected = expectedSchedules.First();

        Assert.AreEqual(expectedSchedules.Count, this.DepartmentSchedules.Count);
        Assert.IsTrue(this.DepartmentSchedules.Any(x => x.Id == expected.Id));
}
```

Definiëren preconditie(aanmaken 3 afdelingsroosters op basis van de tabel uit de given stap)

Uitvoeren van de actie(doen request)

Controleren of de juiste afdelingsrooster is opgehaald

Figuur 10: Uiteindelijke SpecFlow test

Aangezien er volgens TDD gewerkt wordt, was er op dit punt nog geen code geschreven. Oftewel, de code faalde op het moment van uitvoeren (omdat de code uit de WebApi2 Controller een NotImplementedExeption gooide).

6.2.2.2 Unit vs. Integratietests

Nu is het zo dat de test opgesteld in SpecFlow geen unit maar een integratie test is. Dit komt omdat het de volledige functionaliteit test, oftewel het test meerdere units aan code. Hierom is er besloten kleinere tests op te stellen welke units uit deze functionaliteit apart testen. Het idee hierachter was, dat als alle kleinere tests zouden slagen, deze scenario ook zou slagen.

6.2.2.3 Unit test

Om te beginnen aan de unit tests is begonnen aan het testen van de webapi2 controller. Deze test testte vrijwel hetzelfde als het scenario, maar dan focuste het alleen op de webapi2 controller, en niet op de overige klassen die betrekking hebben op de functionaliteit. Deze test zag er als volgt uit.

```
[TestMethod]
0 references | 0 changes | 0 authors, 0 changes
public void GivenIEnterAPeriodAShopAndADepartment_WhenIRetrieveDepartmentSchedule_ThenIGetTheDepartmentSchedule()
{
    //given
    string week = "Week 1";
    int year = 2016;
    string departmentName = "Brood";
    string shopName = "Plus";

    //when
    DepartmentScheduleController.Request = new HttpRequestMessage();
    DepartmentScheduleController.Configuration = new HttpConfiguration();
    DepartmentSchedule dept;

    var response = DepartmentScheduleController.GetDepartmentSchedule(departmentName, shopName, week, year);
    response.TryGetContentValue<DepartmentSchedule>(out dept);

    //then
    Assert.AreEqual(week, dept.Period.Week);
    Assert.AreEqual(year, dept.Period.Year);
    Assert.AreEqual(departmentName, dept.DepartmentName);
    Assert.AreEqual(shopName, dept.ShopName);
}
```

Figuur 11: Unit test

Daar er in TDD zo min mogelijk gedaan wordt om de test te laten slagen, hoefde de controller alleen het juiste afdelingsrooster voor deze test te lagen slagen. Dit zag er als volgt uit:

```
2 references | 1/1 passing | 0 changes | 0 authors, 0 changes
public HttpResponseMessage  GetDepartmentSchedule(string departmentName, string shopName, string week, int year)
{

    return Request.CreateResponse(new DepartmentSchedule()
    {
        DepartmentName = "Brood",
        ShopName = "Plus",
        Period = new Period()
        {
            Week = "Week  Period.Period()
            Year = 2016
        }
    });
}
```

Figuur 12: WebApi2 methode

Na het opstellen van de volgende unit test (dezelfde test, maar deze verwachte een andere afdelingsrooster terug) moest de code uitgebreid worden. Hierbij werd een nieuwe klasse genaamd DepartmentScheduleRepository toegevoegd. Deze klasse praatte met de database en haalde hier de juiste data uit op.

6.2.2.4 Mocking

Het probleem wat de uitbreiding van de code met zich meebracht was dat de bovenstaande test geen unit test meer was. Dit omdat de test zowel de controller als de databaserepository test. Om ervoor te zorgen dat de test alleen de controller test, moest de repository gemockt worden. Dit betekent dat de uitvoering en response van de methode uit de repository gesimuleerd wordt, zodat alleen de code uit de controller uitgevoerd wordt.

Voor het mocken is gebruik gemaakt van het mocking framework Moq. Met dit framework kunnen mock objecten dynamisch aangemaakt worden in de tests. Het probleem is echter dat methodes alleen gemockt kunnen worden als ze van een interface komen, of als de methodes virtual zijn. Als oplossing is er besloten gebruik te maken van dependancy injection over de hele applicatie. Hierover meer in paragraaf 6.1.4.

Het mocken van de databaserepository ging als volgt:

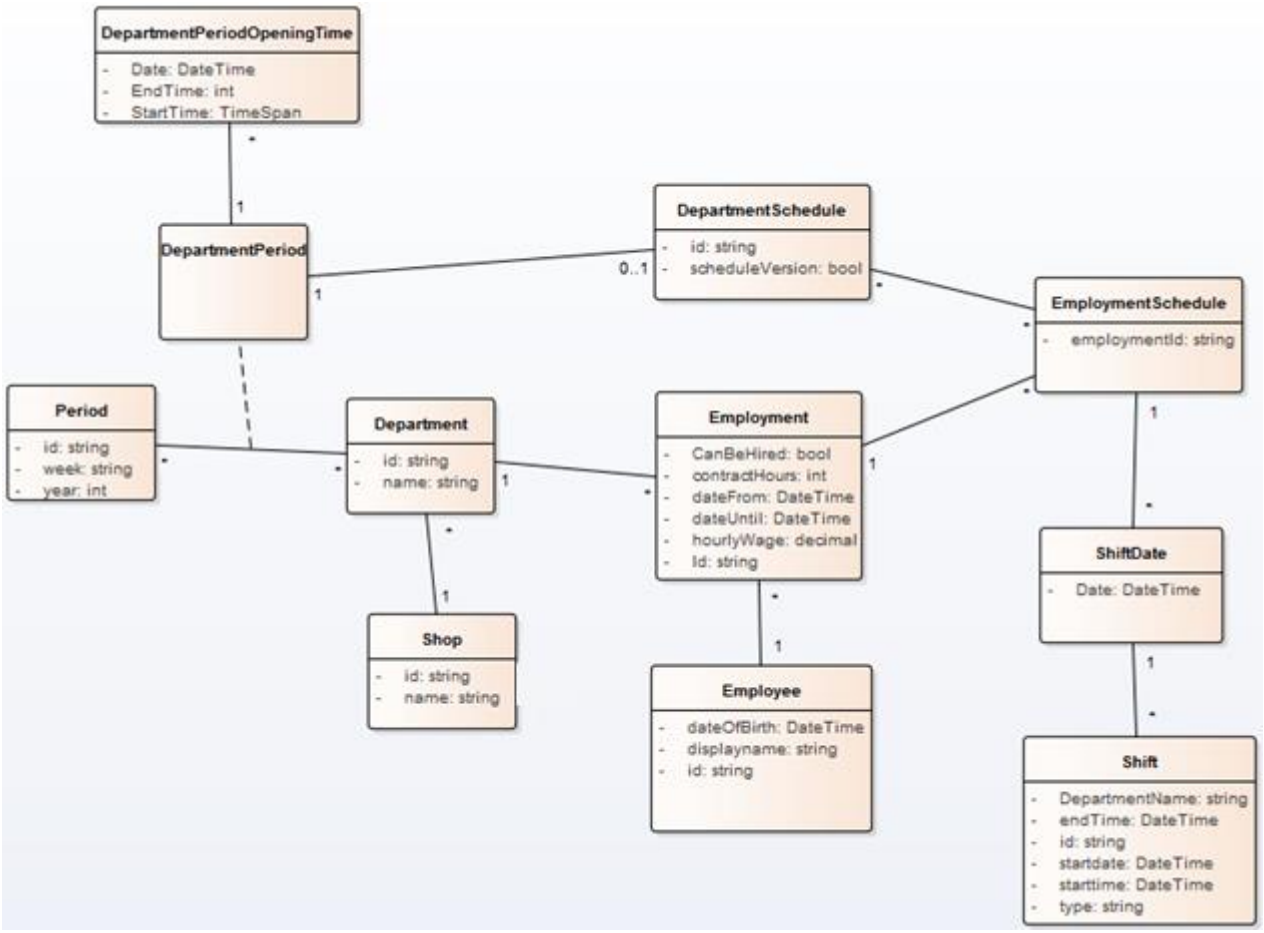
```
var mock = new Mock<IDepartmentScheduleDatabaseRepository>();

mock.Setup(departmentScheduleRepository => departmentScheduleRepository.GetDepartmentSchedules(new List<string> { "Brood" },
"Plus", "Week 1", 2016, 1))

.Returns(new List<DepartmentSchedule> { new DepartmentSchedule()
{
    DepartmentPeriod = new DepartmentPeriod() {
                    Department = new Department() {
                        Name = "Brood",
                        Shop = new Shop() {
                            Name = "Plus"
                        }
                    },
                    Period = new Period()
                    {
                          Description = "Week 1",
                          Year = 2016
                    }
            },
            Id = "DepartmentSchedules/1"
      }

});

this.DepartmentScheduleController = new DepartmentScheduleController(mock.Object);
```

Figuur 13: Mock van database repository

Wat deze code deed is het volgende: op het moment dat de methode GetDepartmentSchedules wordt aangeroepen met de opgegeven parameters, dan retourneert hij de opgegeven lijst, zonder daadwerkelijk de database aan te spreken.

6.2.3 Database

Voor het realiseren van de backlog item over het zien van afdelingsroosters is de volgende database diagram gemaakt:



Figuur 14: Database diagram sprint 1

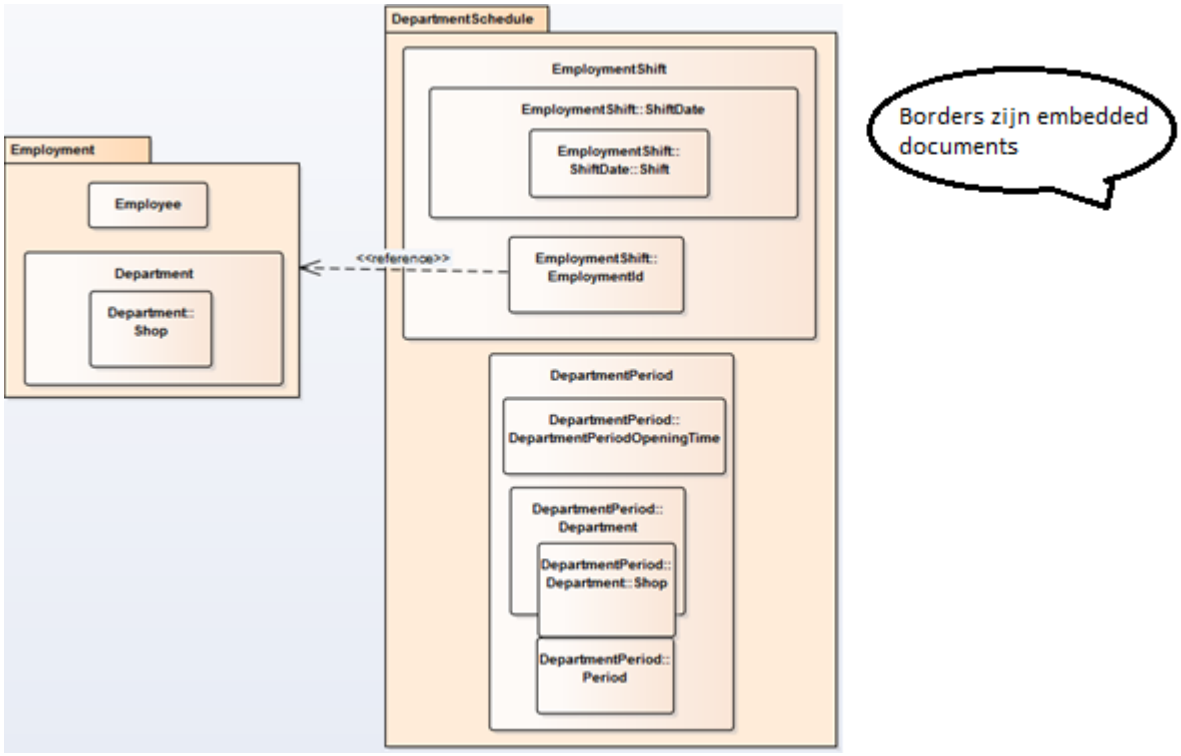
Voor het omgaan met relaties adviseert RavenDB 2 manieren, naast het embedden. Deze zijn te vinden op https://ravendb.net/docs/article-page/3.0/csharp/indexes/querying/handling-document-relationships en houden het volgende in:

- References
    a) Het bijhouden van een verwijzing naar het Id (dit is uitgelegd in het onderzoek)
- Denormalized references
    a) Dit is een combinatie van embedden en referencen, het houdt in dat document A alleen de gebruikte attributen uit document B bijhoudt samen met een reference naar het volledige document B.

Voor het implementeren van dit diagram in RavenDB zijn een aantal beslissingen gemaakt. De belangrijkste beslissing is gemaakt bij het implementeren van het afdelingsrooster. Er is namelijk besloten dat de departmentperiod en employmentshift embed (zoals beschreven in paragraaf 4.1.2 en paragraaf 4.1.7) kon worden in departmentschedule. Echter is er een reference gebruikt om de juiste employment in de employmentshift bij te houden.

De reden dat er een reference gebruikt is, is dat de gebruikte gegevens uit employment (allesbehalve Id) regelmatig wijzigen volgens de opdrachtgever. Ook is het zo dat een dienstverband op elk rooster van een afdeling voorkomt zolang de medewerker in dienst is, oftewel op minimaal 52 roosters per jaar. Dit betekent dat als een medewerker 2 jaar in dienst is, en zijn contract wijzigt, deze wijziging op minimaal 104 plaatsen doorgevoerd moet worden. Omdat dit niet praktisch is, is er gekozen voor een reference.

In figuur 15 is te zien hoe de departmentschedule en employment eruitzagen.



Figuur 15: Departmentschedule, embedded documenten en references

Alle overige klassen behalve EmploymentShift, ShiftDate en Shift hebben ook een eigen collection. De reden dat deze zowel een eigen collection hebben als embed worden, is dat deze data ook op andere plaatsen in het systeem gebruikt worden/geselecteerd kunnen worden. Het is echter vrij statische data die op meerdere plaatsen voorkomt. Vandaar dat er gekozen is om dit te embedden.

Zoals in de pakketselectie genoemd kan er in RavenDB alleen gezocht worden op attributen waar een index op zit. Omdat het afdelingsrooster opgehaald dient te worden op basis van: winkelnaam, afdelingsnaam, week en jaar diende hier een index voor gemaakt te worden. Deze index zag er als volgt uit:

Name DepartmentSchedules/ByPeriod_YearAndPeriod_DescriptionAndDepartmentNameAndShopName

Maps

```
from doc in docs.DepartmentSchedules
select new {
    DepartmentPeriod_Period_Year = doc.DepartmentPeriod.Period.Year,
    DepartmentPeriod_Period_Description = doc.DepartmentPeriod.Period.Description,
    DepartmentPeriod_Department_Name = doc.DepartmentPeriod.Department.Name,
    DepartmentPeriod_Department_Shop_Name = doc.DepartmentPeriod.Department.Shop.Name
}
```

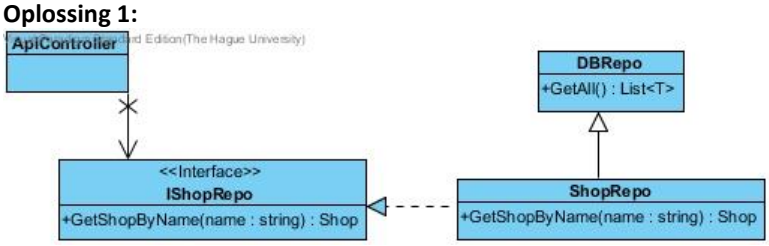
Figuur 16: Index in RavenDB

6.2.4 Ontwerpen

Zoals in paragraaf 6.1.2 aangegeven is er in vrijwel de gehele applicatie gebruik gemaakt van dependancy injection om de testbaarheid te verhogen. Bij het ontwerpen van de communicatie tussen de controller en de repository kon ik echter meerdere implementaties gebruiken.

Het probleem was namelijk dat de repository een aantal generieke methodes bevatte, namelijk: GetAll<T>(), SaveOne(ISaveable) enz. Deze methodes konden door alle repositories gebruikt worden, omdat er maar 1 implementatie was. Naast de generieke methodes waren er ook een aantal specifieke methodes, methodes die alleen voor afdelingsroosters gelde. De meest voor de hand liggende oplossing was het gebruiken van super en subklassen, maar dan bleef de vraag waar staan de generieke methodes in de interface. Voor dit probleem waren een aantal oplossingen bedacht. Deze worden hieronder besproken.

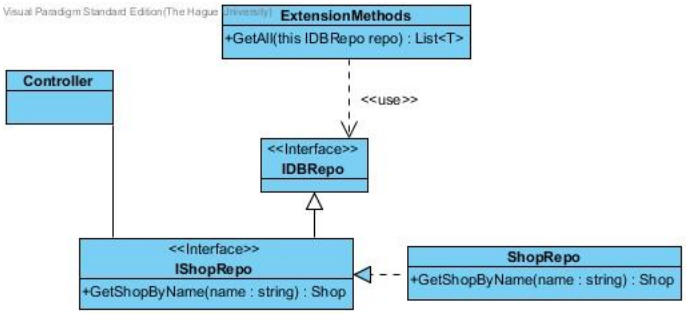
Oplossing 1:



Figuur 17: Eerste oplossing associatie controller en database repository

Met deze oplossing konden de specifieke methodes uit de repository aangeroepen worden door de controller, maar konden de generieke methodes alleen aangeroepen worden door te casten. Dit betekent dat er alsnog een afhankelijkheid tussen de apicontroller en de shoprepo bestaat.

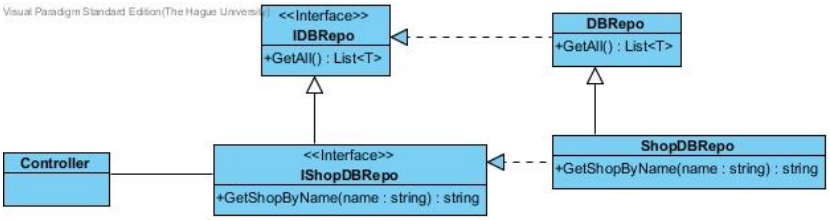
Oplossing 2:



Figuur 18: Tweede oplossing associatie controller en database repository

Deze oplossing loste het probleem grotendeels wel op, omdat de specifieke methodes van shoprepo aangeroepen konden worden, maar de generieke methodes ook (omdat er extension methods op de generieke interface zijn gemaakt). Het probleem hierbij is alleen dat de generieke(extension) methodes niet te mocken zijn, omdat ze statisch zijn en Moq geen statische methodes kan simuleren. Ook is het zo dat de applicatie gebonden is aan die 1e implementatie van de generieke methodes.

Uiteindelijke oplossing:



Figuur 19: Uiteindelijke oplossing associatie controller en database repository

De uiteindelijk gekozen oplossing is in de bovenstaande diagram te zien. Ook hier is er sprake van een super en sub interface, hebben alle repositories een eigen sub interface en houdt de controller een interface bij.

Dit betekend dat alle eerdergenoemde voordelen ook hier van toepassing zijn. Deze manier betekent echter, in tegenstelling tot de vorige oplossing, dat de generieke methodes ook kunnen wijzigen. Oftewel, het is mogelijk om een compleet nieuwe repository te maken, welke een hele andere implementatie heeft van alle methodes. Ook is het mogelijk om de generieke methodes te mocken met behulp van Moq.

Dit is dus de meest uitbreidbare/testbare oplossing.

6.2.4.1 Generics

Zoals in de figuren 17 t/m 19 te zien, is er in data laag een methode GetAll die List<T> returned. Deze methode haalt alle documenten op, op basis van het type wat wordt opgevraagd (bijvoorbeeld alle afdelingsroosters).

De T in de return List wordt ook wel een generic genoemd. Generics betekenen in deze situatie dat de type van de List niet vaststaat, en dus meegegeven moet worden. De methode werkt dus hetzelfde voor alle types en retourneert altijd een lijst van het type dat gevraagd wordt. Het voordeel hiervan is dat de code heel generiek gemaakt kan worden, waardoor je code duplication voorkomt.

Het nadeel hiervan is echter dat generics op zich geen restricties leggen op het type wat gevraagd wordt. In dit geval wil je bijvoorbeeld niet dat alle WebApi2 controllers uit de database worden opgehaald omdat deze zich niet in de database bevinden.

Gelukkig heeft .Net hier een oplossing voor. Het is namelijk mogelijk om in de signature van de methode aan te geven dat er restricties op de generic(T) zitten. Dit doe je door achter de signature het volgende te zetten: “where T : type wat is toegestaan”. Door dit te combineren met interfaces (of superklassen) kan afgedwongen worden dat alleen een bepaalde set types is toegestaan.

Het voordeel van het gebruik van generics met de where, ten opzichte van het retourneren van een object/list van het type interface/superklasse is dat er bij generics niet gecast hoeft te worden naar het specifieke type, als je het specifieke type nodig hebt. Een voorbeeld hiervan is als volgt (het voorbeeld gaat uit van de interface als returnwaarde):

Op het moment dat Interface A een property Id heeft, en klasse B een property Name heeft en de interface realiseert, zou de returnwaarde van de methode GetAll gecast moeten worden naar B om de Name property te gebruiken. Door generics te gebruiken (met de where om af te dwingen dat het type realiseert van interface A) is dit casten niet nodig, omdat je al aangeeft dat je het type B als returnwaarde wilt.

Een voorbeeld van de signature met generics en de where is te zien in Figuur 20.

```
public List<T> GetAll<T>() where T : ISaveable
```

Figuur 20: Generics met type constraint

6.2.5 LINQ-query’s

Om alle afdelingsroosters op te halen moest er vanuit de applicatie een query uitgevoerd worden. Het query’en vanuit de C# library van RavenDB kan op 2 manieren, namelijk:

- Met behulp van de load methode
  a) Gebruikt om documenten op te halen op basis van het id van een of meerdere document(s)
- Met behulp van de Query methode

a) Gebruikt om documenten op te halen op basis van attributen/properties die niet het id zijn.

In deze paragraaf wordt verder ingegaan op de Query methode. Op het moment dat je de query methode gebruikt, kan de query geschreven worden met behulp van LINQ. Verder is het zo, dat er een index gebruikt moet worden om deze query uit te voeren. Als de ontwikkelaar niet aangeeft welke index er gebruikt moet worden, maakt RavenDB zelf dynamisch een index aan welke een index plaatst op alle attributen die gebruikt worden in de query.

Het query-en werkt als volgt: je roept de query methode aan uit het session object. Deze geef je het type mee waarop je wilt query-en (op basis hiervan bepaald RavenDB welke collection hij moet hebben). Als parameter van de query methode geef je de naam van de te gebruiken index mee. In het voorbeeld in figuur 21 is dat de index welke is aangemaakt in paragraaf 6.1.3.

Vervolgens wordt achter de query methode de where methode aangeroepen(method-chaining). In de where methode geef je aan welke waarde de attributen moeten hebben (Vergelijkbaar met de where in SQL maar dan de syntax van LINQ).

In figuur 21 is de query te zien welke is gebruikt voor het ophalen van alle gevraagde afdelingsroosters. Deze query bevat ook een aantal bijzondere methoden, namelijk: Include() en In(). Deze methodes worden hieronder beschreven.

Include:
De include is een manier om references samen op te halen. Zoals in paragraaf 6.1.3 uitgelegd, maakt departmentschedule gebruik van references naar employments. In de applicatie moet echter het gehele object employment gebruikt worden, en niet alleen de reference. RavenDB heeft daar een oplossing op.

In plaatst van het uitvoeren van 2 of meer query's naar de database om de references op te halen, kan er gebruik gemaakt worden van de include methode. Wat deze methode doet is het haalt in de query waar de afdelingsroosters opgehaald worden, ook meteen de employments op, op basis van hun Id. Deze employments zet hij vervolgens in het session object.

Op het moment dat session.Load gebruikt wordt met de id's van de objecten die include waren, worden geen nieuwe requests meer gedaan naar de database. Dus hoewel het samenvoegen van de data nog steeds in de applicatie gedaan moet worden, hoeft er maar 1 query op de database uitgevoerd te worden voor het ophalen van alle data.

In:
De In methode is een vrij simpele methode die gebruikt wordt om te kijken of een waarde voorkomt in een array. In dit geval wordt het gebruikt om te kijken of de naam van een afdeling in een afdelingsrooster uit de database voorkomt in de array met geselecteerde afdelingsnamen.

```
using(var session = DatabaseConnection.OpenSession("RRWeb_POC"))
{
       List<DepartmentSchedule> departmentschedules = session
.Query<DepartmentSchedule>("DepartmentSchedules/ByPeriod_YearAndPeriod_DescriptionAndDepartmentNameAndShopNameAndScheduleVersion")
.Customize(schedule => schedule.Include<DepartmentSchedule>(deptSchedule => deptSchedule.EmploymentShifts.Select(empShift =>
empShift.EmploymentId)))
.Where(departmentSchedule => departmentSchedule.DepartmentPeriod.Department.Name.In(departmentNames)
&& (departmentSchedule.DepartmentPeriod.Department.Shop.Name == shopName)
&& (departmentSchedule.DepartmentPeriod.Period.Description == week)
&& (departmentSchedule.DepartmentPeriod.Period.Year == year))
.ToList();

foreach (var returnSchedule in departmentschedules) {
   foreach (EmploymentShift shift in returnSchedule.EmploymentShifts)
   {
              shift.Employment = session.Load<Employment>(shift.EmploymentId);
   }
}

return departmentschedules;
```

Figuur 21: LINQ query, ophalen afdelingsroosters met Include en In methodes

Wat belangrijk is om te weten, is dat de keuze voor embedding of reference (uit paragraaf 6.2.3) van invloed is op de query die opgesteld wordt. Op het moment dat er gekozen was om alle employments te embedden in departmentschedule in plaats van references te gebruiken, zou de query er ook anders uitzien. De query moet op de volgende manier aangepast worden om dezelfde resultaat te verkrijgen op het moment dat employment embed is:

- De regel die begint met .Customize (de customize methode) kan eruit gehaald worden.
    a) De medewerkers zitten al embed in het rooster, zij hoeven dus niet opgehaald te worden met includes
- De 2 loops aan het einde van de methode kunnen weggehaald worden.
    a) Ook hier is het niet nodig om de medewerkers te koppelen aan de roosters, omdat zij al in het rooster zitten

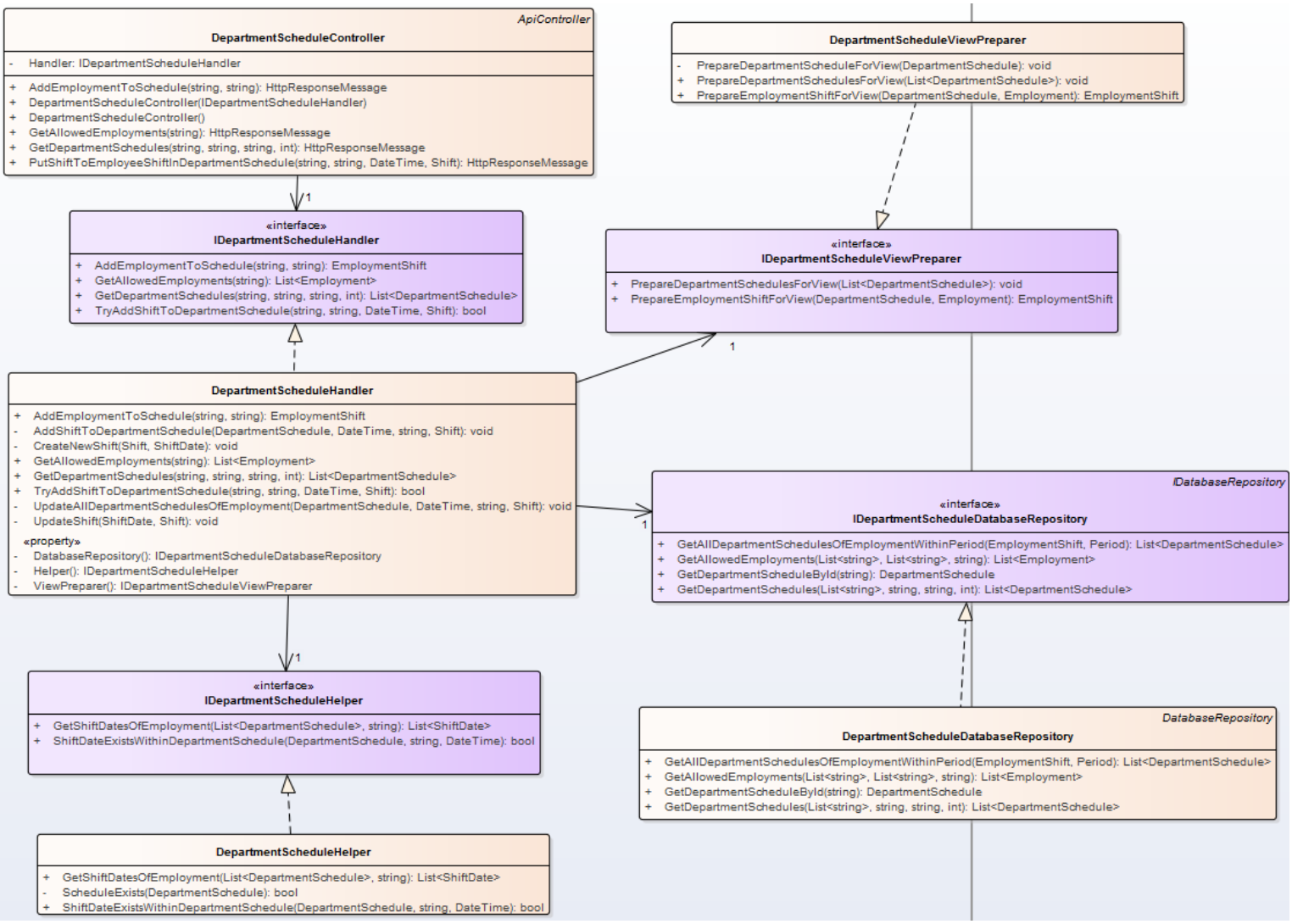
6.2.6 Beslissingen

Naast de beslissing over de data laag, is er in de eerste sprint nog een ontwerp beslissing genomen. Het probleem waar ik tegenaan liep, is dat de webapi2 controller te veel logica begon te bevatten. Dit hoort niet zo te zijn aangezien de controller als enige verantwoordelijkheid heeft het communiceren met de front-end (ontvangen requests en geven van de juiste responses).

Als oplossing hiervoor zijn een aantal klassen gemaakt, namelijk:

- Een handler
    a) Deze regelde het aansturen van de overige klassen
- Een helper
    a) Deze voert kleine methodes uit, als het ophalen van gegevens uit lijsten en controleren of alles in een lijst aanwezig is
- Een viewpreparer
    a) Het voorbereiden van afdelingsroosters (of delen ervan) voor views. Deze roept voornamelijk methodes uit de afdelingsrooster klasse zelf aan
- De databaserepository
    a) Deze is al eerder behandeld en praat met de database.

Om alles testbaar te houden is er ook per klasse een interface gemaakt. De controller hield de interface van de handler bij en de handler hield de interfaces van de overige klassen bij. Dit zag er als volgt uit:



Figuur 22: Design to interfaces

6.2.7 Dienst validatie

In het onderzoek kwam naar voren dat constraints en functionaliteiten uit de database naar de applicatie verplaatst moeten worden. Een van de constraints, genoemd in paragraaf 4.1.11, is de constraint op de start en eindtijd van een dienst. De starttijd moet namelijk voor de eindtijd zijn. Bij het implementeren van de backlog item over het toevoegen van diensten kwam deze constraint aan bod.

Hier is heel simpel een IsValid methode toegevoegd in de shift klasse. Deze controleert of de starttijd van het object voor de eindtijd komt. Dit zag er als volgt uit:

```
public bool IsValid()
{
        return this.StartTime < this.EndTime;
}
```

Figuur 23: Constraint op applicatie niveau

6.3 Sprint 2: Tonen prognosedata in afdelingsroosters m.b.v. RavenDB

De backlog items uit de 2e sprint gingen over het tonen van prognosedata in afdelingsroosters en het goedkeuren afdelingsroosters door de afdelingsmanager.

Om het tonen van de prognose data te realiseren is weer een beslissing op database niveau gemaakt. Deze wordt toegelicht in paragraaf 6.3.2. Ook is er gebruik gemaakt van een AngularJS plug-in voor het tonen van de grafieken. Hierover meer in paragraaf 6.3.3.

6.3.1 Sprint backlog

Voor de 2e sprint was de volgende sprint backlog opgesteld.

User story	Module	Story points
Verwerken feedback sprint 1	Ro	1
Als afdelingsmanager wil ik diensten kunnen verwijderen zodat ik optimaal kan roosteren	Ro	4
Als afdelingsmanager wil ik kunnen zien of dat ik genoeg mensen heb ingepland op een specifiek tijdstip op basis van prognose zodat ik nog mensen kan inhuren als ik tekort kom/minder mensen kan plannen als ik er genoeg heb	Ro/P	15
Als gebruiker wil ik kunnen inloggen en de juiste rechten krijgen zodat ik mijn functie kan vervullen.	Auth	2
Als afdelingsmanager wil ik een akkoord kunnen geven op het rooster zodat de winkelmanager weet dat ik klaar ben met roosteren	Ro	1

Tabel 17: Sprint backlog sprint 2

Halverwege de sprint bleek dat ik vroegtijdig klaar zou zijn met de backlog items. Hierover is met de opdrachtgever besproken dat ik een extra backlog item zou oppakken, namelijk:

User story	Module	Story points
Als winkelmanager wil alle afdelingsroosters van de winkel kunnen vaststellen zodat de staat van het rooster terug kan zien zoals ik er een akkoord op heb gegeven.	Ro	5

Tabel 18: Later toegevoegde backlog item

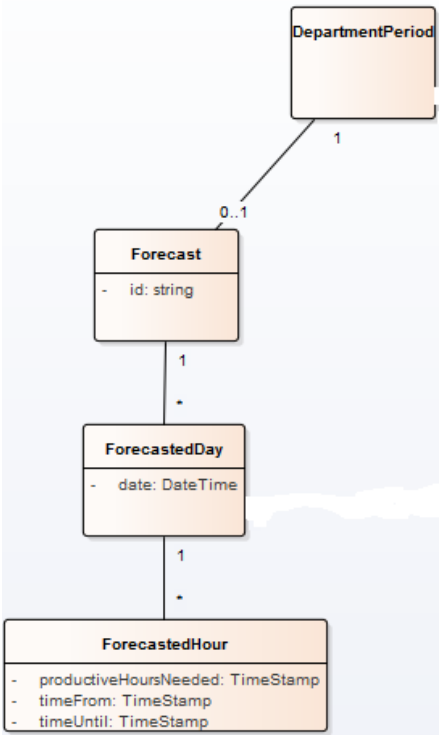
Het verwerken van de backlog items ging op dezelfde manier als beschreven in sprint 1, eerst tests opstellen vervolgens ontwerpen en ontwikkelen. In deze paragraaf ga ik alleen in op de database ontwerp beslissing welke is genomen bij de backlog item zien van prognose data en de AngularJS code die hierbij hoorde.

6.3.2 Database

Het tonen van prognose data ging over de volgende gegevens:

- Per afdeling in een periode, per uur de hoeveelheid productieve uren die nodig zijn.

De database diagram die hierbij hoorde zag er als volgt uit:

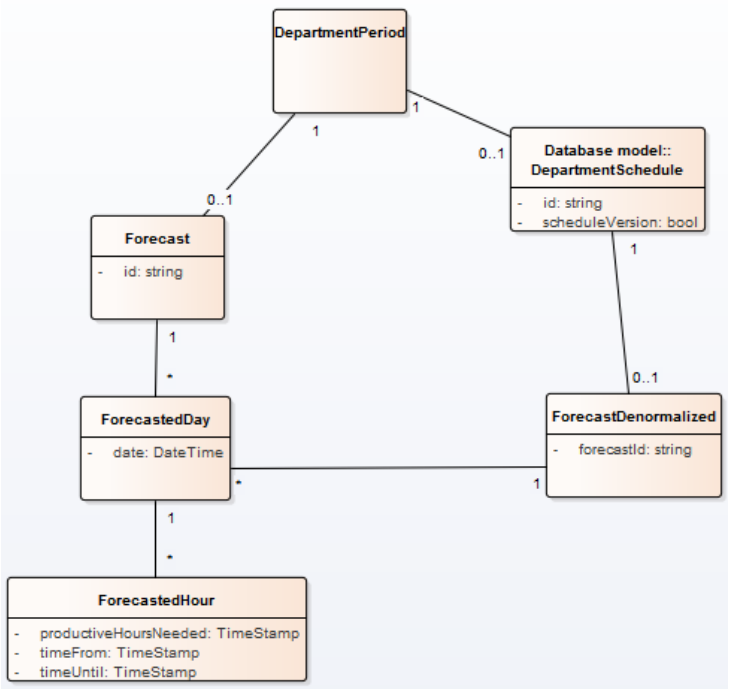


Figuur 24: Database diagram prognose data

Nu is het zo dat de prognose in de echte applicatie meer data bevat dan alleen de voorspelling per dag per uur. Oftewel, een prognose op zich zal veel data bevatten, welke niet allemaal zinvol is in het afdelingsrooster. Ook is het zo dat een prognose bijna nooit wijzigt, en bij maximaal 1 afdeling in 1 periode hoort (net als het rooster).

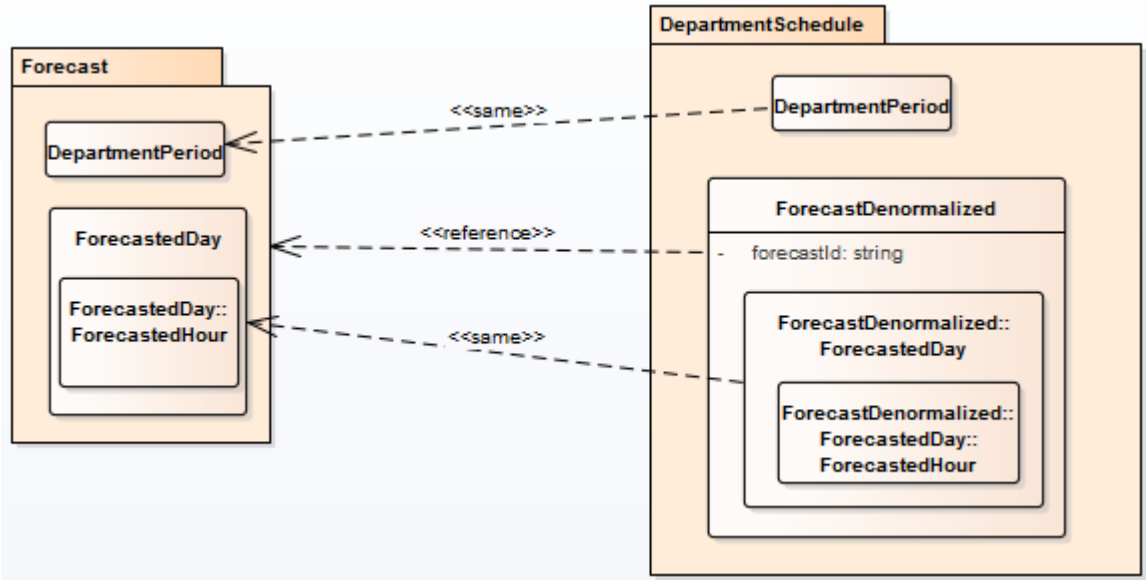
Met deze feiten in het achterhoofd is er besloten voor de techniek denormalized references te gaan (beschreven in paragraaf 6.1.3). Er is een klasse gemaakt genaamd ForecastDenormalized. Deze klasse is als het ware een kopie van de forecast klasse maar bevat alleen de data welke relevant is in het rooster. In dit geval is dit de relatie naar forecasted day. Verder bevat de denormalized variant het Id van de forecast als reference.

Dit zag er als volgt uit:



Figuur 25: Database diagram prognose data met afdelingsrooster

Let erop dat er maar 1 nieuwe collection bij is gekomen, namelijk forecast. De forecastedDays en Forecasted hours bestaan alleen binnen 1 forecast. De implementatie zag er als volgt uit:



Figuur 26: Implementatie relaties denormalized reference

De dependancy lijnen <<same>> zijn er om aan te geven dat dezelfde departmentperiod en dezelfde forecastedDays op 2 plaatsen worden opgeslagen. Dit betekent dat als 1 van de 2 wijzigt, dit altijd op 2 plaatsen gewijzigd moet worden.

6.3.3 AngularJS

Voor het tonen van de grafieken met prognose data is gebruik gemaakt van een open source AngularJS plug-in genaamd Angular Charts. De reden dat ik voor deze plug-in gekozen heb, is dat het precies de grafieken toonde die ik nodig had.

Voor vullen van de juiste data in de grafieken moesten er een aantal dingen gebeuren, namelijk:

- Kijken welke uren prognose data hebben (dus van hoelaat tot hoelaat is de prognose van de dag)
- Per uur kijken hoeveel productieve uren voorspeld zijn
- Per uur kijken hoeveel uur aan medewerkers zijn ingeroosterd

Het tonen van de grafieken per dag gebeurde door middel van de ng-repeat directive. In de repeat werd er een td gemaakt met daarin de directive van de plug-in. Dit zag er als volgt uit:

Figuur 27: Tonen grafieken, html en AngularJS

Wat ook in de bovenstaande afbeelding te zien is, is dat er een ng-click directive op de charts staat. De reden dat deze directive op de charts staat, is dat er een modal getoond moet worden als er geklikt wordt op een chart. De modal die getoond wordt, laat de grafiek in het groot zien. Voor het design van de modal is ook een AngularJS plug-in gebruikt, namelijk UI-bootstrap. Dit is een plug-in die bootstrap componenten gebruikt met AngularJS in plaats van JQuery.

Het openen van de modal gebeurde als volgt:

```
$scope.showModal = function (day) {
        $uibModal.open({
            animation: $scope.animationsEnabled,
            templateUrl: 'myModalContent.html',
            controller: 'departmentscheduleForecastModalController',
            size: "lg",
            resolve: {
                departmentSchedules: function () {
                    return $scope.DepartmentSchedules;
                },
                day: function () {
                    return day;
                }
            }
        });

        $scope.toggleAnimation = function () {
              $scope.animationsEnabled = !$scope.animationsEnabled;
        };
}
```

Figuur 28: Openen modal AngularJS code

De bovenstaande code opent de modal met de html code uit myModalContent.html. Hieraan geeft hij de AngularJS controller 'departmentscheduleForecastModalController mee. Deze controller verwacht 2 extra parameters, namelijk departmentschedules en day. Deze worden meegegeven in de resolve (met grijs aangegeven stuk).

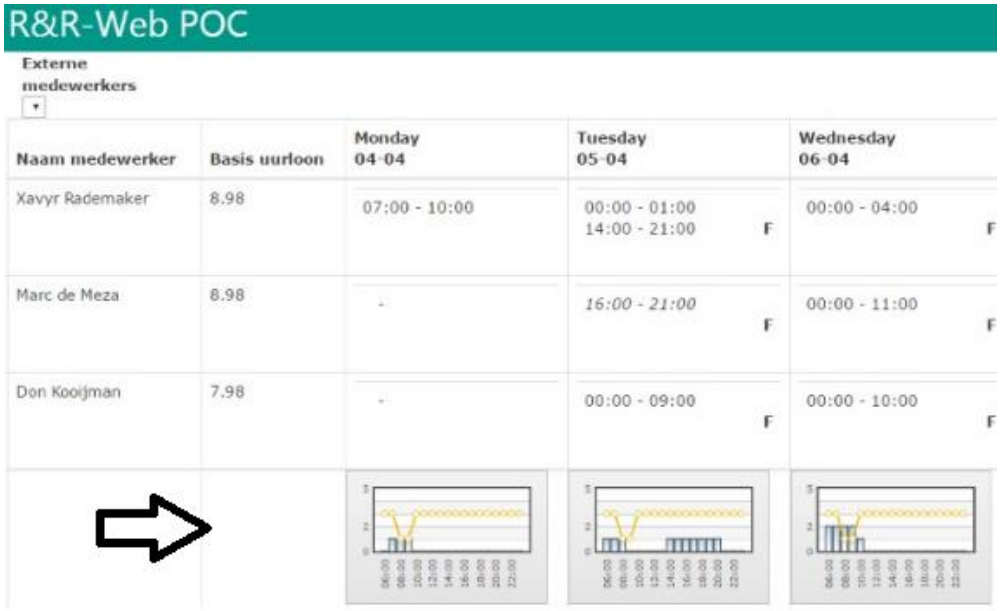
Het ontvangen van de parameters in de controller werkte op dezelfde manier als parameters als $scope en $http worden meegegeven aan controllers. Dit zag er als volgt uit:

```
app.controller('departmentscheduleForecastModalController', function ($scope, $uibModalInstance, departmentSchedules, day) {})
```

Figuur 29: ontvangen parameters in AngularJS controller

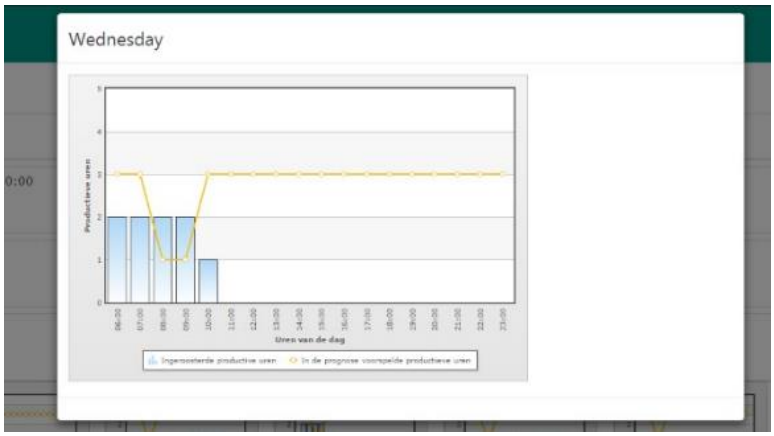
6.3.4 Resultaat

Het resultaat van deze backlog item zag er als volgt uit:



Figuur 30: Resultaat tonen prognose data

Als er vervolgens op een chart geklikt werd, opende er een modal welke de grafiek in het groot toonde. Dit zag er als volgt uit:



Figuur 31: Resultaat tonen prognose data modal

6.4 Sprint 3: Uitbreiden afdelingsrooster m.b.v. RavenDB

De backlog items uit de 3e sprint gingen over het initialiseren en afronden van afdelingsroosters, het tonen van tijdsbudgetten per taak en het tonen van beschikbaarheden van medewerkers.

Hier is weer een beslissing gemaakt op database niveau. Deze wordt besproken in deze paragraaf. Ook kwam er weer een constraint bij kijken, welke verplaatst moest worden naar de applicatie.

6.4.1 Sprint backlog

User story	Module	Story points
Als winkelmanager wil alle afdelingsroosters van de winkel kunnen vaststellen zodat de staat van het rooster terug kan zien zoals ik er een akkoord op heb gegeven.	Ro	2
Als afdelingsmanager wil ik kunnen zien hoeveel tijd dat mijn medewerkers krijgen om een taak uit te voeren zodat ik niet teveel of te weinig mensen inplan en of dat ik de goede personen inplan	Ro/p	6
Als afdelingsmanager wil ik kunnen zien of een medewerker beschikbaar is zodat ik geen niet-beschikbare mensen inrooster	Ro	10
Als afdelingsmanager wil ik een afdelingsrooster kunnen initialiseren zodat ik het roosterproces kan starten	Ro	6

Tabel 19: Sprint backlog sprint 3

In deze paragraaf wordt ingegaan op de volgende backlog items:

- Als afdelingsmanager wil ik kunnen zien hoeveel tijd dat mijn medewerkers krijgen om een taak uit te voeren zodat ik niet te veel of te weinig mensen inplan en of dat ik de goede personen inplan
- Als afdelingsmanager wil ik kunnen zien of een medewerker beschikbaar is zodat ik geen niet-beschikbare mensen inrooster

Deze backlog items hielden het volgende in:

Als afdelingsmanager wil ik kunnen zien hoeveel tijd dat mijn medewerkers krijgen om een taak uit te voeren zodat ik niet te veel of te weinig mensen inplan en of dat ik de goede personen inplan

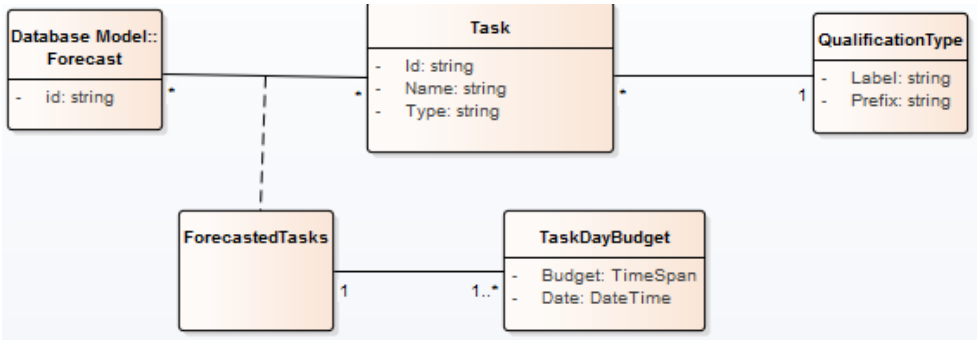
Er moet per afdelingsrooster getoond worden, wat het tijdsbudget van deze taak is. Dit budget moet per dag weergeven worden en is output van de prognose module. Verder kan een taak maar 1 keer voorkomen op de prognose van een afdeling in een periode. Een taak kan dus wel vaker voorkomen in verschillende afdelingsprognoses.

Als afdelingsmanager wil ik kunnen zien of een medewerker beschikbaar is zodat ik geen niet-beschikbare mensen inrooster
Deze backlog item gaat over het tonen van de beschikbaarheden van medewerkers. Deze beschikbaarheid geld per afdelingsrooster. Per dag kan een medewerker verschillende beschikbaarheden hebben, en elk beschikbaarheid heeft een type. Dit type kan zijn:

- Beschikbaar, de medewerker is beschikbaar
- Niet beschikbaar, de medewerker is niet beschikbaar
- Misschien, de medewerker is beschikbaar maar wordt liever niet ingeroosterd.

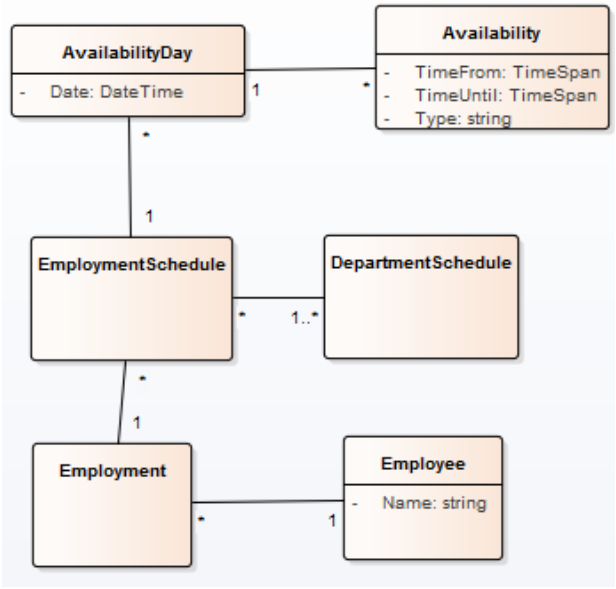
6.4.2 Database

Voor het tonen van het tijdsbudget per taak moest de prognose collectie uitgebreid. Dit betekent dat de denormalized variant van de prognose, welke staat beschreven in paragraaf 6.2.2, ook uitgebreid moest worden. Deze uitbreiding zag er als volgt uit:



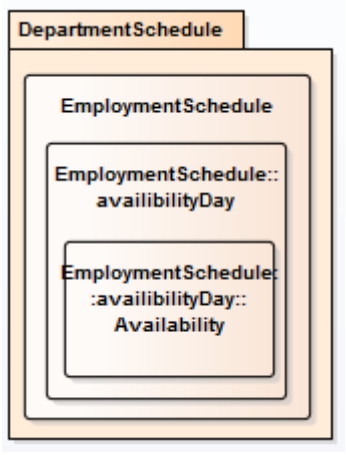
Figuur 32: Database diagram tonen taken

Het tonen van beschikbaarheden van medewerkers per dag zag er als volgt uit:

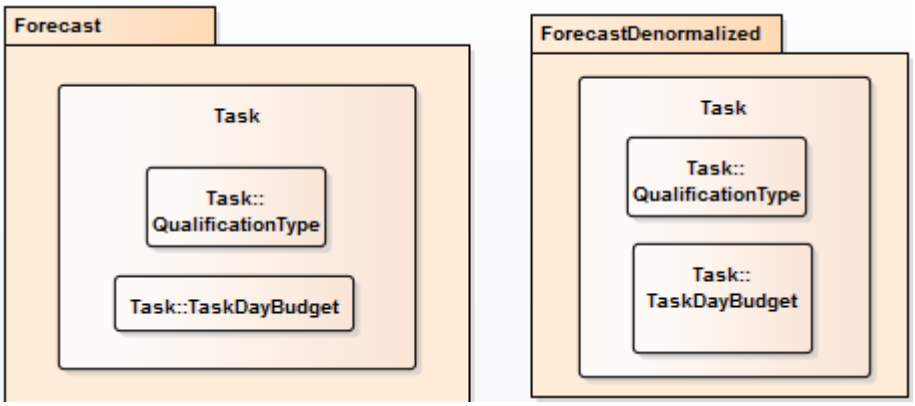


Figuur 33: Database diagram tonen beschikbaarheid

Alle nieuwe entiteiten worden embed in het afdelingsrooster. De beschikbaarheden worden embed via employmentschedule, en de taken via de ForecastDenormalized. Dit is te zien op de volgende afbeeldingen:



Figuur 34: Implementatie beschikbaarheden



Figuur 35: Implementatie tijdsbudget taken

De ForecastDenormalized is nog steeds embed in departmentschedule zoals is uitgelegd in paragraaf 6.2.2.

6.4.3 Grootte document departmentschedule

Zoals te zien in paragraaf 6.3.2 begon het een document uit de collection DepartmentSchedule behoorlijk groot te worden. Om te achterhalen of dit niet te groot werd ben ik gaan zoeken naar de maximaal aanbevolen grootte van een document in RavenDB.

In een forum (https://groups.google.com/forum/#!topic/ravendb/MvXvs6lNxWs) las ik dat Oren Eini(ook wel bekend als Ayende Rahien, Founder van Hibernating Rhino's het bedrijf achter RavenDB) aanraad documenten maximaal een paar honderd kilobyte groot te houden. Vervolgens heb ik in een document uit de collection DepartmentSchedule evenveel taken, werknemers, diensten en beschikbaarheden te plaatsen als in een rooster uit de testomgeving.

Nadat ik dit document had aangemaakt, heb ik de inhoud in een .txt bestand geplaatst, om te kijken hoeveel kilobyte dit document was. Het document was 114 KB groot en bevatte: 13 medewerkers, 56 diensten, 14 taken inclusief een budget voor elke dag voor elke taak en per medewerker 7 beschikbaarheden.

Op basis van deze cijfers heb ik besloten dat er op dit punt in mijn Proof of Concept niks hoeft te veranderen. Wel heb ik de opdrachtgever vermeld, en in het technisch ontwerp opgenomen, dat als het document in de applicatie veel groter wordt dan dit hij meer references gaat moeten gebruiken. Aangezien taken en beschikbaarheden in

aparte tabbladen zitten kunnen deze prima iets later ingeladen worden dan het afdelingsrooster (dus asynchroon opgehaald worden met $http.get nadat het rooster is opgehaald).

6.4.4 Constraint taak type

In de huidige SQL-server database zit er een constraint op de kolom type uit de tabel Task. Deze constraint ziet er als volgt uit:

```
([TAK_WindowType]='OT' OR
[TAK_WindowType]='WO' OR
[TAK_WindowType]='T' OR
[TAK_WindowType]='WE' OR
[TAK_WindowType]='WB' OR
[TAK_WindowType]='OE' OR
[TAK_WindowType]='OB' OR
[TAK_WindowType]='D')
```

Figuur 36: Constraint op taak type

Zoals in het onderzoek (hoofdstuk 4) is beschreven, moeten constraints naar de applicatie verplaatst worden bij het gebruik van document databases. Om deze constraint te realiseren is er gebruik gemaakt van een enumeration in de applicatie. Deze enumeration bevatte alle waardes die zijn toegestaan in de constraint uit figuur 36. Op deze manier dwingt de applicatie af dat een van deze waardes gebruikt worden bij het opslaan/ophalen van een taak.

6.5 Sprint 4: Consistentie in RavenDB

In de laatste sprint stonden geen nieuwe functionaliteiten meer gepland, in plaats daarvan wilde de opdrachtgever graag de conclusies van het project terugzien. Verder waren er een aantal design punten die verbeterd moesten worden in de Proof of Concept aan de hand van feedback uit de sprint review en moest er een data migratie plaatsvinden waar data uit een testdatabase van R&R-web naar de database(RavenDB) die ik gebruikte voor mijn Proof of Concept overgezet moest worden. De conclusies en aanbevelingen zijn te vinden in het volgende hoofdstuk, hoofdstuk 7.

Tijdens sprint 1(paragraaf 6.2) noemde ik al dat er 2 manieren zijn om data op te halen in RavenDB, de query en de Load methode. In de laatste weken kwam ik erachter dat er een belangrijk verschil zitten tussen deze twee methoden. In deze paragraaf leg ik uit wat het verschil is tussen deze methode en wat dit betekende voor zowel mijn Proof of Concept als voor de R&R-webapplicatie.

6.5.1 Load vs. Query methode

Zoals eerder gezegd, kunnen documenten in RavenDB opgehaald worden met behulp van de Load methode (ophalen op basis van id(s)) en de Query methode (ophalen op basis van andere attributen). Naast het verschil van zoekcriteria, is er nog een ander belangrijk verschil tussen deze twee methoden.

Het verschil tussen de twee methoden is namelijk de garantie die RavenDB biedt omtrent de consistentie van de opgehaalde documenten. Omdat de Query methode wordt uitgevoerd op een bepaalde index, garandeert RavenDB niet dat je altijd de meest recente versie van een document/lijst documenten ophaalt. Dit komt omdat het mogelijk is dat een net toegevoegd/gewijzigd document nog niet geïndexeerd is in de gebruikte index. Op het moment dat

dit voorkomt (dat een document nog niet geïndexeerd is) retourneert de query dus een verouderde versie van dit document, of een lijst waar het document onterecht wel of niet in voorkomt.

De Load methode daarentegen haalt altijd de meest recente versie van een document op, omdat het zoeken met deze methode gebeurd op basis van het Id. Het id van een document wordt toegevoegd op het moment dat het document wordt opgeslagen. Vanaf dit moment kan er dus gezocht worden op deze id aangezien het ook niet meer wijzigt.

6.5.1.1 Proof of Concept en R&R-web

Voor mijn Proof of Concept (en ook voor R&R-web als er overgestapt wordt naar RavenDB) betekend het verschil tussen de 2 methodes (Load en Query) dat er gekeken moet worden waar de meest recente versie belangrijk is.

Een voorbeeld van een actie waar de meest recente data heel belangrijk is, is het ophalen van afdelingsroosters. Op het moment dat iemand een dienst toevoegt aan een medewerker op een afdelingsrooster, moeten andere gebruikers deze toevoeging ook zien. De reden dat dit belangrijk is, is te zien in de volgende situatie:

Afdelingsmanager A van afdeling A plant een dienst in voor medewerker A in zijn rooster. Deze dienst is op maandag en begint om 12:00 uur en eindigt om 18:00 uur. Vlak nadat deze dienst is toegevoegd, opent afdelingsmanager B van afdeling B zijn afdelingsrooster, en huurt medewerker A in op zijn rooster. Bij het inhuren van medewerker A komen alle ingeplande diensten van deze medewerker ook mee. De onlangs toegevoegde dienst (maandag van 12 tot 18 uur) is nog niet verwerkt en wordt dus niet meegenomen. Afdelingsmanager B voegt een dienst toe aan medewerker A op maandag van 16:00 uur tot 20:00 uur.

In de bovenstaande situatie zou medewerker A op maandag van 16:00 uur tot 18:00 uur 2 diensten hebben op 2 verschillende afdelingen. Om dit te voorkomen moet in deze situaties de Load methode gebruikt worden. Dit betekent wel dat de client het Id van het rooster moet opsturen.

Een voorbeeld van een actie waar de meest recente data minder belangrijk is, is bijvoorbeeld het toevoegen van nieuwe Periodes. Op het moment dat we in oktober 2016 zitten en alle periodes/feestdagen van 2017 worden ingevoerd, hoeven deze periodes niet direct zichtbaar te zijn in de applicatie. De reden dat dit niet hoeft, is dat er nog 2 maanden zijn om de eerste weken van 2017 in te plannen en dit dus geen haast heeft.

7. Conclusies en aanbevelingen

Zoals in paragraaf 6.5 vermeld, wilde de opdrachtgever weten wat ik na afloop van het project concludeer omtrent het overstappen met (bepaalde modules van) R&R-web naar RavenDB. In dit hoofdstuk leg ik kort uit wat de conclusies en aanbevelingen waren. De conclusies zullen gaan over:

- Welke modules kunnen overstappen
- Wel of niet overstappen naar RavenDB op dit punt

Vervolgens komen de aanbevelingen. Deze zullen voortbouwen op de conclusie. In de aanbevelingen doe ik aanbevelingen over:

- Wat zijn naar mijn mening de volgende stappen die genomen moeten worden

7.1 Conclusie

Na te hebben gewerkt met RavenDB, en met functionaliteiten uit R&R-web ben ik tot een aantal conclusies gekomen. Deze conclusies zijn, zoals eerder dit hoofdstuk vermeld, onderverdeeld in 2 categorieën, namelijk: “Welke modules kunnen overstappen naar RavenDB” en “Kan er op dit moment overgestapt worden naar RavenDB”.

7.1.1 Welke modules kunnen overstappen naar RavenDB

Van de 4 grote modules in R&R-web, welke zijn benoemd in paragraaf 4.3.3(Prognose, Rooster, Realisatie en uitbetaling) kunnen de volgende modules/delen van de modules zeker overstappen naar RavenDB:

- Roostermodule (volledig)
- Realisatie (volledig)
- Prognose (in ieder geval de output die gebruikt wordt in de vervolgmodules)

Hieronder een korte toelichting per module.

7.1.1.1 Roostermodule

De roostermodule kan volledig overstappen, zoals ook te zien is in het Proof of Concept. De reden dat deze module volledig kan overstappen, is dat een afdelingsrooster geaggregeerde(gedenormaliseerde) data is. Er zijn verschillende componenten in het rooster, zoals diensten en beschikbaarheden van medewerkers, die nooit apart gebruikt worden. Verder is het zo dat diensten op meerdere afdelingsroosters voor kunnen komen (als een medewerker wordt ingehuurd), maar de hoeveelheid roosters waarop het voorkomt is beperkt.

Verder is het zo dat bij het toevoegen/wijzigen van 1 dienst alle diensten worden verwijderd uit de database, en de nieuwe situatie opnieuw wordt toegevoegd (in de huidige situatie). Dit is niet meer nodig op het moment dat RavenDB gebruikt wordt, omdat er dan maar 1 document geüpdatet hoeft te worden in plaats van meerdere losse rows.

7.1.1.2 Realisatie module

De realisatie module is niks meer dan de rooster module waarin de daadwerkelijk gemaakte uren worden toegevoegd. Dit betekent dat een document van het rooster kan worden gekopieerd en aangevuld met deze data in deze module.

7.1.1.3 Prognose output

De output van de prognose kan in ieder geval overgezet worden. Reden is, dat deze output gebruikt wordt in zowel de rooster als de realisatie module. Deze data kan dus embed worden in de gebruikte documenten uit de andere modules wat het uitvoeren van meerdere query's bespaard.

7.1.2 Kan er op dit moment overgestapt worden naar RavenDB

Momenteel ben ik niet van mening dat er overgestapt moet worden. Hoewel het vanuit een ontwikkel perspectief prima werkt, omdat de data is opgeslagen zoals het gebruikt wordt, is er nog te weinig bekend over de stabiliteit en betrouwbaarheid onder verschillende belastingen. Ook is er nog niet bekend hoe RavenDB omgaat met concurrency en sharding (wordt gebruikt om horizontaal schalen te realiseren). Deze aspecten moeten dus eerst onderzocht worden.

7.2 Aanbevelingen

Op basis van de conclusie heb ik een aantal aanbevelingen gedaan voor het vervolg van dit project. Deze aanbevelingen worden in deze paragraaf beschreven.

7.2.1 Wat zijn de volgende stappen

Zoals in paragraaf 7.1.2 vermeld, zijn er nog een aantal aspecten van RavenDB die niet behandeld zijn in dit project. Ik ben van mening dat deze aspecten eerst bekeken moeten worden voor er besloten kan worden of er wel of niet overgestapt kan worden. Hieronder is te zien wat er naar mijn mening nog moet gebeuren voor er een beslissing genomen kan worden:

- De betrouwbaarheid en stabiliteit van RavenDB moet gemeten worden onder verschillende belastingen
    a) Dit kan bijvoorbeeld worden gedaan door load, piek en stress testen uit te voeren
    b) Het liefst op een omgeving vergelijkbaar met de productie omgeving
- Er moet gekeken worden hoe RavenDB omgaat met meerdere gebruikers tegelijkertijd(concurrency)
- Er moet gekeken worden hoe RavenDB omgaat met sharding
    a) Om de horizontale schaalbaarheid te realiseren
    b) Alleen als de Vries WFM besluit hier gebruik van te maken

Afhankelijk van de resultaten van de bovenstaande tests, kan de Vries WFM besluiten wel of niet over te stappen naar document databases.

8. Evaluatie

In dit hoofdstuk wordt er teruggeblikt op mijn afstudeeropdracht. Zo zullen de opgeleverde producten, het verloop van het proces en de uitgevoerde beroepstaken geëvalueerd worden.

8.1 Productevaluatie

Hieronder is per product een korte evaluatie te vinden.

8.1.1 Onderzoek

Het onderzoek moest inzicht geven over waar document databases gebruikt konden worden in de R&R-webapplicatie. Het onderzoek heeft dit ook gedaan. Het toont aan waar document databases gebruikt kunnen worden, welke voordelen er te behalen vallen, welke nadelen dit met zich meebrengt en wat de gevolgen zullen zijn voor de applicatie, zoals te zien in paragraaf 4.3.10 en 4.3.11.

Zowel de voordelen als de gevolgen van het overstappen naar document databases heb ik kunnen gebruiken in het vervolg van het project, zoals bijvoorbeeld het verplaatsen van constraints naar de applicatie (paragraaf 6.2.8 en 6.4.4). Ook de kennis over de R&R-webapplicatie heeft bijgedragen aan onder andere het bouwen van de Proof of Concept. Een aantal beslissingen, zoals waar data te embedden en waar de data te reference in de Proof of Concept (paragraaf 6.2.3), zijn gebaseerd op eerder opgedane kennis uit het onderzoek.

Hoewel ik tevreden ben over het onderzoek, denk ik dat het nog beter had gekund. Namelijk door de pakketselectie als deel van het onderzoek op te nemen. Door de pakketselectie een onderdeel van het onderzoek te maken, konden de krachten –en zwakheden van relationele en document databases vervangen worden door de krachten –en zwakheden van SQL-Server 2012 en RavenDB. Ook de conclusie zou heel specifiek over RavenDB vs. SQL-server gaan.

De reden dat ik van mening ben dat dit een betere conclusie zou opleveren, is dat er een aantal punten zijn die RavenDB wel aanbiedt, die niet standaard zijn in document databases. Een voorbeeld hiervan is de garantie van ACID.

8.1.2 Pakketselectie

Over het algemeen ben ik tevreden over de pakketselectie. Er zijn echter wel punten waar ik rekening mee had moeten houden. Een van deze punten is het kijken of de SQL-database wel geoptimaliseerd was voor de performancetest. Een ander punt is dat ik MarkLogic niet aan de praat kreeg. Ik had deze database graag meegenomen in de performancetest.

Na het werken met RavenDB ben ik van mening dat de medewerkers van de Vries WFM er snel mee zullen leren werken zodra zij besluiten over te stappen. De reden dat ik dit denk is door de aanwezigheid van de C# library, de mogelijkheid tot query'en met LINQ (paragraaf 6.2.6) en omdat de opdrachtgever liet weten dat hij de standaard CRUD-acties met RavenDB begreep nadat ik het hem liet zien aan het einde van de pakketselectie (paragraaf 5.5).

Verder is RavenDB vanuit een ontwikkel perspectief ook een prima database voor de R&R-webapplicatie. Een aantal redenen hiervoor is de mogelijkheid om references samen met het geheel op te halen in 1 query, paragraaf 6.2.6,

en de garantie op consistente data bij het ophalen op basis van id, paragraaf 6.5.1, wat toch wel van belang is bij een aantal functionaliteiten (zoals het ophalen van afdelingsroosters).

Het is echter niet zo dat de Vries WFM direct kan overstappen op RavenDB. Hiervoor zouden zij eerst nog een aantal punten moeten onderzoeken, zoals de stabiliteit onder verschillende belastingen (paragraaf 7.2.1).

8.1.3 Proof of Concept

De Proof of Concept heeft zijn doel behaald. Alle opgestelde backlog items zijn gerealiseerd en werken zoals verwacht (alle tests slagen).

De Proof of Concept laat onder andere zien dat constraints (zoals de validatie op dienst en de type van taak in paragrafen 6.2.8 en 6.4.4) naar de applicatie verplaatst moeten worden, hoe er omgegaan moet worden met relaties en hoe document databases gebruikt kunnen worden in de R&R-webapplicatie. Binnen de Proof of Concept zijn manieren van relaties zoals references, embedded documenten en denormalized references gebruikt. Dit geeft de opdrachtgever een goed beeld van wanneer welke techniek te gebruiken.

Verder laat de Proof of Concept ook het verschil tussen de Load en de Query methode zien. Ook dit kan de opdrachtgever meenemen als er besloten wordt over te stappen naar RavenDB.

8.2 Procesevaluatie

Het grootste deel van het proces verliep goed. Zoals in de inleiding van hoofdstuk 6 vermeld heb ik over het gehele traject 1 week vertraging gehad. Dit had voorkomen kunnen worden als ik van tevoren rekening hield met de tijd die het kost om data over te zetten naar de 3 document databases. Naast de tijd die ik kwijt was aan de data migratie, had ik ook rekening moeten houden met de mogelijkheid dat het mis kon gaan, en de leercurve van het werken met verschillende databases. Oftewel ik had ruimer moeten inplannen.

Verder heb ik ook gebruik gemaakt van TDD en SpecFlow. Beide technieken zijn mij goed bevallen. Nu is het wel zo dat ik SpecFlow niet gebruikt heb voor de unit tests, omdat ik van mening was dat dit mij te veel tijd zou kosten. Het was een leerzame ervaring, waarbij de kwaliteit van de geschreven code te allen tijde gewaarborgd bleef. Ik zal deze technieken waarschijnlijk vaker gaan gebruiken.

Het werken volgens Scrum is mij ook goed bevallen. Het kort cyclisch werken en opleveren (2-wekelijkse sprints) zorgde ervoor dat de opdrachtgever feedback kon geven op het moment dat het product afweek van wat hij ervan verwachte. Het wijzigen van deze afwijkingen kostte ook niet veel tijd, omdat het om een klein deel van het systeem ging. Verder zorgde het testen, ontwerpen (applicatie en database) en bouwen per backlog item ervoor dat het werk niet eentonig werd, maar ook vlotter ging.

Wat mij ook erg heeft geholpen waren de daily logs (paragraaf 3.4.1) omdat ik mijn eigen fouten kon inzien, maar ook een oplossing ervoor kon bedenken op een korte termijn (omdat ik een moment nam om erbij stil te staan). Ook hielpen deze logs mij erbij om de sprint per dag in te plannen. Op het moment dat ik een dag meer of minder gedaan kreeg dan gepland, wist ik dus ook dat er gevolgen voor de rest van de sprint zullen zijn.

8.3 Evaluatie beroepstaken

Voorafgaand aan de afstudeerperiode heb ik een aantal uit te voeren beroepstaken in mijn afstudeerplan vermeld. In deze paragraaf wordt per beroepstaak besproken hoe/of ik hieraan heb voldaan.

8.3.1 Selecteren van standaardsoftware

Hoewel er een aantal verbeterpunten waren in de pakketselectie (de migratieproblemen) ben ik van mening dat de pakketselectie goed is uitgevoerd. De criteria zijn opgesteld met de opdrachtgever en waar nodig specifieker gemaakt op het moment van uitvoeren. Een voorbeeld hiervan is de eis over beschikbare documentatie (paragraaf 5.2.3).

8.3.2 Ontwerpen, bouwen en bevragen van een database

Bij het ontwerpen van de database voor de Proof of Concept is gedacht aan de situatie van de huidige applicatie. Met deze situatie in gedachte zijn structuur beslissingen gemaakt gebaseerd op de manier waarop RavenDB het adviseert (references, denormalized references en embedding). Dit is terug te vinden in paragrafen 6.2.3, 6.3.2 en 6.4.2 (database in sprint 1, 2 en 3).

Naast de Proof of Concept zijn er ook databases gebouwd en bevraagd tijdens de pakketselectie. Tijdens de pakketselectie zijn er performancetests uitgevoerd en is er een data migratie uitgevoerd. Voor het bevragen van de databases zijn zowel SQL, XQuery als Lucene query syntax gebruikt (paragraaf 5.3).

8.3.3 Bouwen applicatie

Zoals in hoofdstuk 6 te zien is er ontwikkeld in C#/.NET. De Proof of Concept is object georiënteerd gebouwd en maakt onder andere gebruik van WebApi2 voor de server en AngularJS op de front-end.

Er zijn in het Proof of Concept ook verschillende technieken gebruikt, zoals LINQ (voor het query'en, paragraaf 6.2.6) en Generics (paragraaf 6.2.5).

Gedurende de pakketselectie is er ook gebruik gemaakt van NHibernate, WebRequests en de XMLSerializer (Paragraaf 5.4.4.3).

8.3.4 Uitvoeren van en rapporteren over het testproces

Er is ontwikkeld volgens Test Driven Development en daarbij is gebruik gemaakt van SpecFlow, zoals te zien in sprint 1(paragraaf 6.2.2). Verder zijn er zowel unittests als integratietests uitgevoerd.

Bronnen en bijlagen

In dit hoofdstuk zijn de bronnen, gebruikt in het onderzoek (hoofdstuk 4), terug te vinden. Ook is er een tabel met de bijlages toegevoegd.

8.4 Gebruikte bronnen

In de volgende tabel is te zien welke bronnen er gebruikt zijn in het onderzoek.

Bron	Auteur	Link
1	Bill Vorhies	http://data-magnum.com/a-brief-history-of-big-data-technologies-from-sql-to-nosql-to-hadoop-and-beyond/
2	Kristina Chodorow	http://usuaris.tinet.cat/bertolin/pdfs/mongodb_%20the%20definitive%20guide%20-%20kristina%20chodorow_1401.pdf
3	Ameya Nayak, Anil Poriya, Dikshay Poojary	http://research.ijais.org/volume5/number4/ijais12-450888.pdf
4	Niels Nagle, Hylke Peek	https://www.youtube.com/watch?v=S3z5xnUHo-Q
5	-	http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_9/database_design/miniweb/pg8.htm

8.5 Bijlagen

In de volgende tabel is te zien welke bijlages er zijn.

Bijlage	Document
Bijlage A	Plan van Aanpak
Bijlage B	Onderzoek
Bijlage C	Pakketselectie
Bijlage D	Functioneel ontwerp
Bijlage E	Technisch ontwerp
Bijlage F	Conclusies en aanbevelingen
Bijlage G	Afstudeerplan