# Een generieke opzet van smart resourcesystemen

Afstudeerverslag Falco IJben

Naam: Falco IJben
Studentnummer: 406761
Bedrijf: Recognize
Bedrijfsbegeleider: de heer J. Hobert
Afstudeerdocent: de heer R. Hommels

# 1. Inhoudsopgave

# 2. Samenvatting

Recognize heeft op dit moment drie soortgelijke smart resourceprojecten gerealiseerd en verwacht dat zij vaker smart resourceprojecten zal gaan realiseren. Smart resourcesystemen brengen vraag en aanbod zo efficiënt mogelijk bij elkaar. Deze systemen zullen op grote en kleine schaal moeten presteren, en tegelijkertijd wil Recognize een systeem dat een degelijke basis biedt voor het opzetten van een smart resourcesysteem.
Het afstudeerproject heeft een agile insteek door verschillende elementen uit agile toe te passen, echter heeft het ook veel weg van een waterval project doordat verschillende fases in het project van elkaar afhankelijk zijn. Naast werkmethodiek zijn er verschillende methodes toegepast om de kwaliteit te waarborgen, waaronder (automatische) testen, pull requests en continuous integration. Om de bruikbaarheid aan te tonen van het eindproduct is er ook een proof of concept die het eindproduct gebruikt om een smart resourcesysteem te representeren.
Om de requirements van het project duidelijk te krijgen en passende technieken te vinden, zijn er drie onderzoeken uitgevoerd: het overlappingsonderzoek, het onderzoek naar architectuur en technieken en het onderzoek naar dataopslag.
Aan de hand van deze onderzoeken zijn de requirements opgesteld van het eindproduct, is er een ontwerp gemaakt en zijn de technieken gekozen die gebruikt zijn. Uit de onderzoeken zijn twee belangrijke non-functionele eisen gekomen: het eindproduct moet schaalbaar en uitwisselbaar zijn.
Om deze eisen te behartigen is de microservice architectuur gekozen, zodat de microservices an sich uitwisselbaar zijn en het geheel horizontaal schaalt. De microservices zijn gemaakt in Kotlin, in het framework Spring en gebruiken Postgres in combinatie met PostGIS als dataopslag. De microservices worden gehost in een kubernetes cluster in combinatie met Docker, waardoor er gemakkelijk meerdere replica’s van een microservice naast elkaar kunnen worden gedraaid.
De microservices samen vormen de engine, die als het ware de motor is van toekomstige smart resourcesystemen. De engine bestaat uit twee microservices, genaamd: Match en Auth. De communicatie tussen de microservices en die met de buitenwereld gebeurt door middel van REST API’s, die JSON als datarepresentatie gebruiken. De microservices volgen meerdere design patterns om de uitwisselbaarheid te verhogen, zoals multilayer architecture en dependency injection.
De engine kent meerdere manieren van configuratie, waaronder het configureren van het aantal replica’s van een microservice, de logging en project specifieke constanten. Alle verschillende manieren van configuratie, een handleiding en verdere documentatie is te vinden in het systeemdossier.
Op basis van bevindingen tijdens het uitvoeren van het afstudeerproject en gesprekken met de bedrijfsbegeleider, worden er aanbevelingen gedaan voor toekomstige projecten die gebruikmaken van de engine, of die de engine uitbreiden.

# 3. Inleiding

In opdracht van Recognize en als afstudeeropdracht geboden aan mij, is er een project overeengekomen waarin het hoofddoel is een generiek stuk software te bouwen, die als basis dient voor toekomstige projecten waarin vraag en aanbod door slimme software bij elkaar worden gebracht.
Dit document geeft inzicht in het afstudeerprocess, wat het afstudeertraject heeft opgeleverd en hoe dat alles tot stand is gekomen.

## 3.1 Achtergrond

### 3.1.1 Bedrijfsbeschrijving

Recognize is een softwarebedrijf dat gevestigd is in Almelo en maakt maatwerkapplicaties voor onder andere: web, mobile applicatie, virtual reality, augmented reality, IoT en smart resourceprojecten.
Recognize is onderdeel van VolkerWessels Telecom.

# 4. Opdracht

De student is samen met Recognize in overeenstemming gekomen tot een passende en uitdagende afstudeeropdracht. De opdracht heeft betrekking op het verbeteren en generaliseren van systemen die vraag en aanbod met elkaar in contact brengen, zogeheten ‘smart resourcesystemen’.

## 4.1 Aanleiding

Recognize heeft op dit moment drie soortgelijke smart resourceprojecten gerealiseerd. Een voorbeeld hiervan is de situatie wanneer een consument iets geïnstalleerd moet hebben (vraag) en deze met een monteur (aanbod) in contact wordt gebracht; of als de consument een pakket snel wil verzenden (vraag) en deze met een koerier (aanbod) in contact wordt gebracht.
De drie reeds gerealiseerde systemen lijken veel overlap te hebben in de basiswerking en de wensen van het systeem. Echter zijn de gerealiseerde projecten afzonderlijk van elkaar vanaf de grond opgebouwd. Daardoor is de werking soortgelijk, maar is de implementatie een wereld van verschil.

## 4.2 Probleemstelling

Recognize verwacht dat ze vaker smart resourceprojecten zal gaan realiseren, mede omdat er reeds drie projecten succesvol zijn gerealiseerd. Echter is ze voor de realisatie daarvan telkens veel tijd kwijt. Ook zet ze vraagtekens bij de technieken die tot dusver zijn gebruikt in de smart resourcesystemen, met betrekking tot schaalbaarheid en uitwisselbaarheid.
Deze twee termen vormen tevens de belangrijkste non functionele eisen aan de smart resourcesystemen, omdat de systemen ook op grote schaal moeten presteren en flexibel moeten zijn.

## 4.3 Doelstelling

De uiteindelijke doelstelling is een basis bieden voor toekomstige smart resourceprojecten binnen Recognize, door middel van een bruikbaar stuk software. Deze software moet aan alle wensen van en eisen aan een smart resourceplatform overweg kunnen. Daarnaast zou in theorie deze software geïmplementeerd kunnen worden in de bestaande smart resource systemen.
Om deze doelstelling te halen, zijn er deeldoelstellingen opgesteld, die elk een onderdeel van deze doelstelling afbakenen. Deze deeldoelstellingen zijn:

- Onderzoek naar de eisen aan en wensen van een smart resource systeem en de technieken die daarvoor gebruikt kunnen worden.
- Het realiseren van een software-oplossing die kan fungeren als basis voor toekomstige smart resourcesystemen.
- Een proof of concept van een fictief smart resourcesysteem, die de basisonderdelen van een smart resourcesysteem bevat. Dit dient voor het aantonen van de bruikbaarheid van het stuk software.

Of en hoe deze doelstellingen zijn behaald in het verloop van het afstudeerproject, staat beschreven verder in dit verslag.

# 5. Process

Voor het afstudeertraject zijn er meerdere procesmatige maatregelen getroffen en methodes gebruikt om het onderzoek in goede banen te leiden. In dit hoofdstuk wordt er een overzicht gegeven van wat er allemaal is gedaan met betrekking tot het proces.

## 5.1 Werkmethodiek

In het plan van aanpak is er aangegeven dat er voor een agile aanpak van het project is gekozen, door de volgende elementen uit agile methodieken toe te passen:

- Een backlog. Dit is een vertrouwde manier om het werk in kaart te brengen en te prioriteren.
- Incrementeel de producten opleveren. Hiermee kunnen gevaren omtrent voortgang en planning op tijd worden opgemerkt en kan de planning worden bijgesteld.
- Iteratief te werk gaan. Door iteratief werk op te pakken, kan er reëel worden gepland. Het werk dat in de volgende increment wordt opgepakt hangt af van het voorgaande.

Dit heeft als voordeel dat de koers van het project bijgesteld kan worden wanneer er nieuwe inzichten zijn, zodat het project haalbaar blijft en daadwerkelijk wordt gerealiseerd wat de opdrachtgever nodig heeft. Hiervan is meerdere malen gebruik gemaakt, bijvoorbeeld in de ontwerpfase van het project of toen de implementatie van de engine in twee bestaande smart resourcesystemen niet meer realistisch bleek.

Hoewel de projectaanpak een agile insteek heeft, heeft het achteraf gezien ook veel weg van een waterval aanpak. Er waren duidelijke fases in het project die - hoewel deze soms overlapten - in een doorlopende reeks zijn afgerond, en niet op een iteratieve manier. Dit komt doordat er afhankelijkheden waren in fases uit de voorgaande fases. Bijvoorbeeld de afhankelijkheid van het onderzoek naar technieken die gebruikt worden in de engine. Zonder dit onderzoek afgerond te hebben, kon er geen doorslaggevende keuze worden gemaakt over de technieken, waardoor de daadwerkelijke realisatie van de codebase van de engine niet van start kon gaan.
De aanpak zoals hierboven beschreven is terug te zien in de de fasering die beschreven staat in hoofdstuk 7 (Eindproduct).

## 5.2 Begeleiding

Gedurende het afstudeerproject heb ik twee soorten begeleiding gehad, gericht op zowel het proces als het eindproduct van het afstudeertraject. Dit zijn de begeleiding vanuit Recognize en de begeleiding vanuit het Saxion.

### 5.2.1 Bedrijfsbegeleiding

De begeleiding vanuit Recognize is hoofdzakelijk door Juul Hobert verricht, wat tevens de bedrijfsbegeleider van mij is. Bijna elke week - op uitzondering van twee weken na - heeft er een voortgangsgesprek plaatsgevonden, waarbij de resultaten tot dan toe zijn besproken en er op resultaten is gereflecteerd. Naar aanleiding van deze gesprekken werden vervolgens verbeteringen aangebracht in de producten. Dit omvat verbeteringen in de codebase en in de documenten rondom het afstuderen.

Daarnaast was Juul aanwezig, wanneer ik vragen had over verschillende zaken rondom het eindproduct of de documentatie van het afstuderen.
Naast Juul stonden ook andere werknemers van Recognize open voor vragen. Echter beperkten zich deze vragen zich enkel tot programmeertechnische vragen.

### 5.2.2 Schoolbegeleiding

Vanuit het Saxion is Rogier Hommels als begeleider aangesteld. Rogier heeft hoofdzakelijk procesmatig advies gegeven en feedback op de documenten geleverd. Gedurende het afstudeerproject heeft er twee keer een voortgangsgesprek tussen de student en Rogier in persoon plaatsgevonden bij Recognize op locatie, waarin onder andere het plan van aanpak is doorgenomen.
Naarmate het einde van het afstudeertraject naderde, is de frequentie van het contact toegenomen, wat de vorm aannam van telefonische meetings. Deze meetings hadden als voornaamste doel feedback te leveren op de onderzoeken en afstudeerverslaglegging. Daarnaast heeft Rogier advies geleverd op de opzet van de verslaglegging.

## 5.3 Tijdsverloop

In het plan van aanpak is een globale planning gemaakt van het afstudeerproject. Hoewel deze globale planning grotendeels klopt, zijn er kleine wijzigingen geweest in de planning. Deze planning is dus up-to-date gehouden en kan daardoor worden gebruikt om inzicht te krijgen in het verloop van de afstudeeropdracht. De herziene globale planning is te vinden in de bijlage “Tijdsverloop-gantt-chart.pdf”.

## 5.4 Kwaliteit

Voor de waarborging van de kwaliteit van het eindproduct zijn een aantal methodes gebruikt. Deze methodes staan in dit hoofdstuk beschreven, en bij elke methode staat toegelicht hoe dit bijdraagt aan de waarborging van de kwaliteit. Ook wordt er beschreven wat deze methode inhoudt.

### 5.4.1 Testen

Om de kwaliteit van de codebase te waarborgen, is een combinatie van handmatige testen en automatische testen gebruikt.
Wat deze testen inhouden en hoe deze zijn geïmplementeerd is te vinden in hoofdstuk 7.2.5 (Testen).

### 5.3.1 Code Style

Binnen recognize zijn er automatische code style checks, met als doel de leesbaarheid en dus de kwaliteit van de codebase te garanderen.
Deze checks worden uitgevoerd wanneer er een codewijziging wordt gepushed naar bitbucket. Dit wordt gedaan door middel van bitbucket pipelines, zie 5.4.4 (Continuous integration).
Doordat kotlin nog weinig gebruikt wordt binnen recognize, zijn er nog geen code styles hiervoor. Dit houdt in dat er geen automatische check hierop worden gedaan. Er wordt echter wel op code style gelet in de pull requests.

### 5.4.3 Pull requests

Pull request is een methode om de kwaliteit van de codebase zeker te stellen. Een pull request is een voorlopige wijziging aan de codebase, die ter review wordt neergelegd bij werknemers van Recognize. Bij elke change in de programmatuur wordt een pull request gemaakt, waarop de reviewers de aanpassingen en toevoegingen in de code kunnen controleren en feedback kunnen geven waar zij dat nodig achten.
Pas wanneer de pull request is goedgekeurd, worden de wijzigen en toevoegingen doorgevoerd in de codebase van de engine.

Bij het beoordelen van een pull request wordt er gekeken naar de volgende aspecten:

- **Codestyling**. Is de code conform de de gangbare manier van de taal in kwestie? In het geval van kotlin is hier nog geen standaard voor binnen recognize. Wel wordt er op consistentie gelet van de codestyling;
- **Structuur**. De structuur van de code uit het pull request moet overzichtelijk zijn, met een duidelijke opsplitsing van functionaliteit en verantwoordelijkheid. Ook wordt er gekeken of er bijvoorbeeld geen dubbele code in staat;
- **Werking**. Er wordt gekeken naar wat de code zou moeten doen en wat het daadwerkelijk doet. Edgecases in de code worden nagelopen, om zo bugs te voorkomen;
- **Leesbaarheid**. De code uit de pull request moet leesbaar zijn. Dat wil zeggen: dat wat de code doet uit de code op te maken moet zijn, bijvoorbeeld door juiste naamgevingen van functies en variabelen.

Naast deze vier punten is het een vereiste dat er een ‘groen vinkje’ bij de pull request staat. Dit groene vinkje, of rode kruisje, is het resultaat van de pipelines. In deze pipeline kunnen bijvoorbeeld de automatische testen worden doorlopen. Wat er allemaal met de bitbucket pipelines kan worden gedaan is te lezen in paragraaf 5.4.4 (Continuous integration).

### 5.4.4 Continuous integration

Continuous integration is een methode om de kwaliteit van de codebase te waarborgen, door zo vroeg mogelijk inzichtelijk te maken dat een verandering in de codebase niet aan de eisen voldoet.
Continuous integration wordt toegepast door heel recognize, en hoewel dit afstudeerproject door een enkel persoon wordt uitgevoerd, ook toegepast bij dit afstudeerproject.
Hiervoor is Bitbucket gebruikt, met bitbucket pipelines[1]. De Repositories zijn gehost bij bitbucket, en door middel van de bitbucket pipelines kan de continuous integration worden bereikt. De pipeline van het project wordt bij elke code commit uitgevoerd. In deze pipelines kunnen de volgende zaken worden uitgevoerd:

- Testen. De automatische testen kunnen worden doorlopen en het rapport dat daar uitkomt kan worden gedownload;
- Builds. Er kunnen automatisch builds worden gemaakt van de codebase;
- Code style checks. Er kunnen automatisch code style checks worden uitgevoerd.
- Deployment. Er kan automatisch, of semi automatisch, worden gedeployed naar de cluster.

In het geval van de engine worden alle van de hierboven gestelde zaken toegepast, op de code style checks na.
Hoe dit alles te configureren is, is te lezen in paragraaf 7.2.4.5 (Configuratie).

### 5.4.5 Proof of concept

Om te bewijzen dat de engine daadwerkelijk gebruikt kan worden voor smart resourcesystemen, is er met behulp van de engine een proof of concept smart resourcesysteem gebouwd. Hierin zit de volledige basisfunctionaliteit van een typisch smart resourcesysteem.
Het smart resourcesyteem dat gebouwd is heet “Falonely” en is een systeem dat jongeren met te veel vrije tijd in contact brengt met eenzame ouderen, om samen tijd door te brengen.

# 6. Onderzoeken

Tijdens het afstudeertraject zijn de volgende drie onderzoeken, in de beschreven volgorde, uitgevoerd:

1. het overlappingsonderzoek;
2. het onderzoek naar architectuur en technieken;
3. het onderzoek naar data opslagmethodes.

Met behulp van deze onderzoeken wordt de deeldoelstelling uit paragraaf 4.3 (Doelstelling) behaald: “onderzoek naar de eisen aan en wensen van een smart resource systeem en de technieken die daarvoor gebruikt kunnen worden”.
Met het uitvoeren van deze onderzoeken wordt tevens aan de algemene eis voldaan die worden gesteld aan het afstudeertraject. Deze eis dicteert dat het afstuderen een onderzoekselement moet bevatten.

[1] Een continuous integration tool van bitbucket.

## 6.1 Overlappingsonderzoek

In de analysefase van het afstudeerproject is er onderzoek gedaan naar de overlapping van de reeds geïmplementeerde smart resourcesystemen, om een beeld te krijgen van wat een smart resourcesysteem inhoudt en vervolgens een advies over de requirements uit te kunnen brengen.

### 6.1.1 Probleemanalyse en doelstelling

Dit onderzoek dient ervoor om uit te zoeken wat de overlapping van de bestaande smart resourcesystemen Miles, Traffer en Utilio zijn en welke functionaliteit geabstraheerd kan worden en dus in het eindproduct zou moeten worden opgenomen.
Naast de functionaliteiten die aanwezig zijn in alle bestaande systemen, zijn er mogelijk ook functionaliteiten die gewenst zijn in toekomstige smart resourcesystemen, maar die niet in elk van de bestaande systemen aanwezig is.

De doelstelling van dit onderzoek is dan ook om te achterhalen wat de belangrijkste eisen en wensen zijn van een smart resourcesysteem. Op basis hiervan kunnen de prioriteiten van de requirements en de requirements an sich worden bepaald. De hoofdvraag van dit onderzoek luidt: **Welke functionaliteiten kunnen worden geabstraheerd uit de bestaande systemen, en zijn kandidaat om geïmplementeerd te worden in de software-oplossing?**

De hoofdvraag zal worden worden beantwoord aan de hand van de volgende drie deelvragen:

1. Hoe werkt elk van de smart resourcesystemen functioneel?
2. Welke van de functionaliteiten komen overeen in de verschillende systemen?
3. Welke van de andere functionaliteiten zijn verder gewenst in de software-oplossing?

### 6.1.2 Methodiek

Om het gestelde doel te bereiken, zijn er twee data verzamelmethodes gebruikt: literatuuronderzoek en interviewen.

#### 6.1.2.1 Literaruur onderzoek

Doordat de hoeveelheid documentatie van en over de systemen verschilt - van één systeem is zelfs helemaal geen documentatie - is alleen de documentatie gebruiken geen betrouwbare methode om de smart resourcesystemen te vergelijken.
Daarom zijn naast documentatie ook de systemen gebruikt en is de codebase vergeleken, om van elk systeem een beeld te krijgen hoe het werkt. Zowel vanuit het perspectief van de verschillende gebruikers, als op implementatieniveau van de business regels.

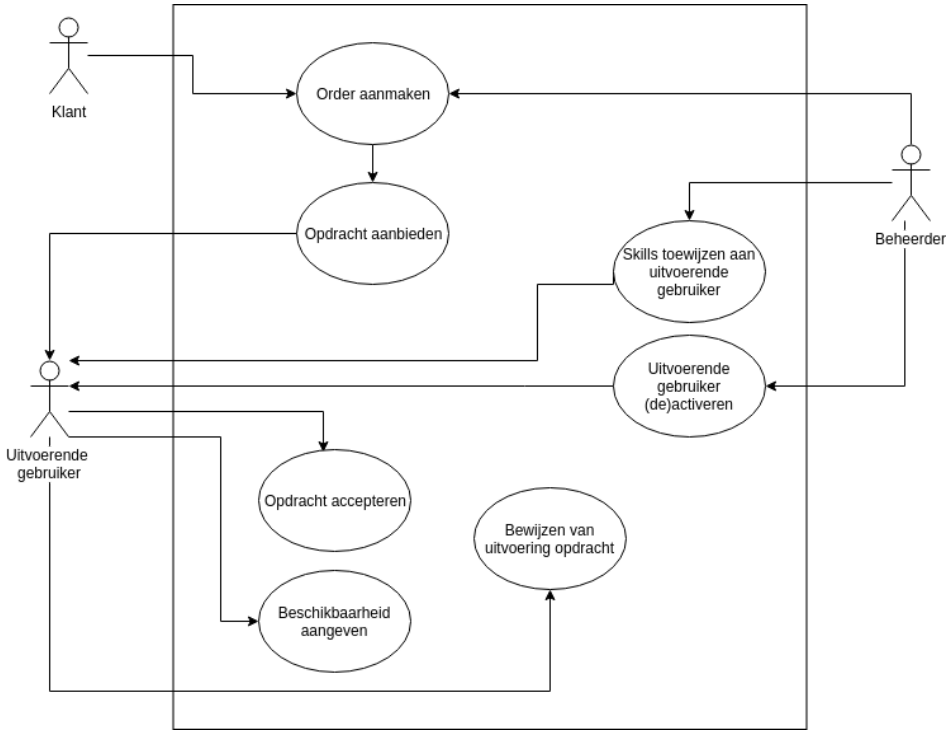
#### 6.1.2.2 Interview

Om een advies te vormen over de gewenste functionaliteiten van een smart resourcesysteem, is er een interview afgenomen met de product owner van Utilio, Timo van

Ginkel. Dit interview is een semi gestructureerd interview; dat wil zeggen dat er een aantal vragen zijn opgesteld om het gesprek te sturen, maar er ook ruimte is voor meer inbreng van de product owner. De uitwerking van dit interview is te vinden in de bijlage “Uitwerking-interview-Timo.pdf”.

### 6.1.3 Resultaten

De overeenkomsten in functionaliteit van de verschillende bestaande smart resourcesystemen vanuit het perspectief van de gebruikers, kan worden samengevat in een use case diagram. Dit diagram is gegeven in figuur 1.




*Figuur 1, Generiek use case diagram*

De complete resultaten van dit onderzoek, inclusief de gewenste maar niet overlappende functionaliteiten, zijn te vinden in bijlage 1 (Overlappingsonderzoek resultaten).

### 6.1.4 Conclusie

De hoofdvraag van dit onderzoek luidt: welke functionaliteiten kunnen worden geabstraheerd uit de bestaande systemen, en zijn kandidaat om geïmplementeerd te worden in de software-oplossing?
Hieronder zijn twee lijsten gegeven, die gezamenlijk alle functionaliteiten bevatten die kunnen worden geabstraheerd uit de bestaande smart resourcesystemen. De lijst is opgesplitst in twee delen: functionaliteiten die wel in de software-oplossing geïmplementeerd zouden moeten worden en functionaliteiten die niet in de software-oplossing zullen komen. Bij elke functionaliteit die niet zal worden meegenomen in de software-oplossing is aangegeven waarom deze niet in erin komt.
De indeling van de functionaliteiten in de twee lijsten is gedaan op basis van gesprekken met de productowner van het eindproduct van het afstudeerproject.

**Wel in de software oplossing**

- Klanten kunnen opdrachten aanbieden.
- Het aanbieden van opdrachten wordt gedaan op basis van locatie, beschikbaarheid en skills.
- Een beheerder kan skills toevoegen aan een uitvoerende gebruiker.
- Opdrachten worden automatisch aangeboden aan uitvoerende gebruikers.
- Uitvoerende gebruikers delen hun locatie met het systeem.
- Uitvoerende gebruikers kunnen hun beschikbaarheid aangeven.
- Uitvoerende gebruikers kunnen opdrachten afronden.
- Een beheerder kan een uitvoerende gebruiker activeren en deactiveren.
- Track en trace van een opdracht.

**Niet in de software oplossing:**

- Integratie met boekhoudpakket.
  Hoewel dit een zeer gewenste functionaliteit is van de software-oplossing, wordt het omwille van de tijd niet meegenomen in dit project. Daarnaast is elk van de koppelingen met een boekhoudpakket afhankelijk van de implementatiedetails van een smart resourceproject, dat wil zeggen, de velden en producten in kwestie.
- Een hiërarchische opzet van uitvoerende gebruikers.
  Dit is over het algemeen niet van toepassing in smart resourceprojecten. Deze feature wordt dan ook vaak niet gebruikt en wordt daarom niet meegenomen in de software-oplossing.
- Integratie met SMS-diensten. De integratie met SMS-diensten is bij elk smart resourceplatform anders, dit betekent dat de complete integratie niet mogelijk is.
- Push berichten in de app van de uitvoerende gebruiker. Omwille van de omvang van deze functionaliteit en het gegeven tijdsbestek, zal dit niet worden geïntegreerd in de software-oplossing.

## 6.2 Onderzoek naar architectuur en technieken

Om in de ontwerpfase van het afstudeerproject de technieken te bepalen die gebruikt zullen worden om de wensen van en eisen aan smart resourcesystemen te behartigen in de software-oplossing, is het onderzoek naar architectuur en technieken uitgevoerd. Dit onderzoek houdt zich echter niet bezig met de dataopslagmethodiek, daarvoor is een apart onderzoek uitgevoerd die te vinden is in hoofdstuk 6.3 (Onderzoek naar dataopslag).

### 6.2.1 Probleemanalyse en doelstelling

De drie smart resourcesystemen die reeds zijn ontwikkeld door Recognize, hebben elk een andere architectuur. Wel maken ze gebruik van dezelfde technieken: PHP met het Symfony framework. Deze draaien in een Docker container die weer draait in een Kubernetes cluster. Recognize heeft echter vraagtekens bij de schaalbaarheid van de huidige smart resourcesystemen, ze verwacht dat deze op grote schaal beter kunnen presteren.

Het doel van dit onderzoek is om uit te zoeken welke architectuur en technieken gebruikt kunnen worden om de schaalbaarheid en uitwisselbaarheid van de software-oplossing te ondersteunen, dan wel garanderen. Daarom luidt de hoofdvraag van dit onderzoek: **Met behulp van welke technieken kunnen de smart resourcesystemen binnen Recognize worden opgezet zodat deze schaalbaar en uitwisselbaar zijn?**
Met schaalbaarheid wordt in deze context bedoeld met of het systeem groter te maken is en of te zorgen is dat het systeem een grote workload aankan. Een systeem kan horizontaal schalen (dat wil zeggen dat het opschaalt door meer machines toe te voegen aan de clusteren) en verticaal schalen (dat het opschaalt door meer rekenkracht toe te voegen aan de machine).
Met uitwisselbaarheid wordt de mate waarin er gemakkelijk onderdelen van het systeem kunnen worden vervangen, toegevoegd of verwijderd bedoeld.

Deze hoofdvraag wordt beantwoord aan de hand van de volgende deelvragen:

1. Welke architecturen kunnen dienst doen voor de smart resourcesystemen?
2. Welke architectuur is het meest geschikt voor een smart resourcesysteem?
3. Welke technieken kunnen gebruikt worden voor de smart resourcesystemen?
4. Welke technieken zijn het meest geschikt voor een smart resourcesysteem?

De uitkomst van de eerste deelvragen kunnen invloed hebben op de laatste deelvragen, doordat er voor- of nadelen zijn met technieken in de context van een architectuur.

### 6.2.2 Methodiek

#### 6.2.2.1 onderzoeksopzet

In dit onderzoek worden een aantal opties voor zowel de architectuur en de technieken onderzocht. Om een keuze te kunnen maken uit deze opties, worden voor elk van de opties voor en nadelen benoemd die vervolgens tegen elkaar worden afgezet. Deze weging wordt

onderbouwd en herleidt naar de wensen voor een smart resourcesysteem of andere relevante criteria.
De technieken die onderzocht gaan worden zijn onderverdeeld in twee groepen: programmeertaal & framework en compartment managing.

#### 6.2.2.2 Literatuuronderzoek

Dit onderzoek baseert zich op literatuur, om een keuze te maken tussen de verschillende gekozen opties.

#### 6.2.2.3 Expertise

Als aanvulling op de literatuur worden ervaringen van de werknemers van Recognize gebruikt in dit onderzoek. Doordat de technieken die worden onderzocht bekend zijn bij hen, kan dit dienen als aanvulling.

### 6.2.3 Resultaten

De resultaten van dit onderzoek worden kort beschreven per deelvraag. Zo worden de voor- en nadelen niet benoemd voor elke architectuur of techniek, maar wordt alleen de conclusie die daaruit wordt getrokken gegeven. De volledige uitwerking van de resultaten, inclusief de voor- en nadelen met inbegrip van de redenatie, is te vinden in de bijlage: Onderzoek naar architectuur resultaten.

#### 6.2.3.1 Welke architecturen kunnen de dienst doen voor de smart resourcesystemen?

Doordat smart resourcesystemen alleen de basiswerking met elkaar gemeen hebben, zal er bij het opzetten van een nieuw systeem dus telkens moeten worden uitgebreid met nieuwe functionaliteiten. De architectuur moet hier mee om kunnen gaan en het het liefst ook ondersteunen.
Hiervoor zijn er een tweetal opties, die tevens de enige opties zijn die ooit ter sprake zijn gekomen:

- Framework. Een framework zal over basisfunctionaliteit van een smart resourcesysteem beschikken. Het framework is dan als het ware de eerste opzet van elk nieuw smart resourcesysteem;
- Microservice architecture. De smart resourcesystemen zullen komen te bestaan uit verschillende microservices die elk een deel van de verantwoordelijkheid van de algemene werking op zich nemen.

#### 6.2.3.2 Welke architectuur is het meest geschikt voor een smart resourcesysteem?

De belangrijkste uitgangspunten schaalbaarheid en uitwisselbaarheid zijn beide beter vertegenwoordigd in een microservice achitecture, doordat dit horizontaal schaalt en een microservice alleen hoeft te voldoen aan een de communicatie interface met andere microservices. Daarbij zijn er andere voordelen verbonden aan de keuze voor een microservice architecture, zoals dat er niet meerdere implementaties van een functionaliteit in de codebase komen. Er zullen dan meerdere microservices zijn waaruit men kan kiezen.

#### 6.2.3.3 Welke technieken kunnen gebruikt worden voor de smart resourcesystemen?

De technieken die gebruikt gaan worden voor smart resourcesystemen zullen bekend moeten zijn bij Recognize. Dit is om te zorgen dat de systemen uitbreidbaar en onderhoudbaar zijn door Recognize. De technieken die gebruikt gaan worden zijn onder te verdelen in twee groepen: programmeertaal & framework en compartment managing.

**Programmeertaal en framework**
De keuze van de programmeertalen vloeit voort uit de talen die op dit moment gebruikt worden binnen Recognize, en die dus bekend zijn bij haar programmeurs. Naast PHP, de taal waar alle huidige smart resourcesystemen in gerealiseerd zijn, worden ook Kotlin, C#, Java en Javascript gebruikt.

PHP is de meest gebruikte taal voor backend systemen binnen recognize en wordt in alle gevallen gebruikt met het framework Symfony. Het wordt gebruikt voor onder andere de bestaande smart resourcesystemen en zal daarom nader worden bekeken in dit onderzoek. Het framework dat samen met PHP nader bekeken gaat worden is Symfony. Dit framework is zeer bekend bij de programmeurs van Recognize en de huidige smart resourcesystemen zijn met dit framework gerealiseerd.

Kotlin is een taal op de JVM, maar met minder boilerplate en meer suger syntax dan Java en wordt binnen recognize beschouwd als strictly beter dan Java. Kotlin wordt nog niet zoveel gebruikt binnen Recognize als PHP, maar heeft bij nieuwe projecten de voorkeur over PHP vanwege de snelheid van de applicaties die erin gemaakt zijn. Kotlin zal nader worden onderzocht, in combinatie Spring boot. Spring boot is een framework bovenop Spring, die net als symfony het gemakkelijker maakt een web applicatie te realiseren.

Javascript kan serverside gebruikt worden door middel van NodeJS. Echter is het gebruik van NPM een probleem, veel libraries zijn constant in ontwikkeling. Daardoor moeten deze continu worden geüpdate of gedeprecated. Hierdoor biedt dit geen goede basis voor deze smart resourcesystemen die lang mee zouden moeten gaan

C# valt niet onder de opties die nader worden bekeken, omdat de teams binnen recognize die uiteindelijk de smart resourcesystemen gaan bouwen en die de bestaande beheren niet werken met C#. Daarvoor is er gekozen om het bij een taal te laten die al bekend is bij de teamleden.

De programmeertalen en frameworks die nader onderzocht worden in dit onderzoek zijn dus:

- Kotlin gebruik makend van Spring boot;
- PHP gebruik makend van Symfony.

**Compartment managing**
Voor compartment managing worden Docker en Kubernetes momenteel gebruikt binnen Recognize. Compartment managing is een onderdeel van het beheren van systemen wanneer deze uitgerold zijn. Docker wordt gebruikt om container applicaties te draaien in

een Kubernetes cluster. De verschillende projecten/systemen en de omgevingen (test/acc/pods) zijn gescheiden door middel van namespaces. Dit is echter geen vereiste, beide technieken kunnen afzonderlijk gebruikt worden. Docker kan multi-container applicaties draaien door middel van docker compose en docker swarm.
Docker compose is bedoeld voor lokale development en niet voor in een cluster en docker swarm is juist bedoeld voor in een cluster. Beide verbinden ze docker containers aan elkaar en aan de de buitenwereld.
Zowel Kubernetes als docker kunnen de dienst doen voor smart resourcesystemen.

#### 6.2.3.4 Welke technieken zijn het meest geschikt voor een smart resourcesysteem?

**Programmeertaal en framework**
Kotlin heeft aanzienlijk meer voordelen dan PHP in de context van de smart resourcesystemen en hun behoeftes. Kotlin heeft een voordeel op de schaalbaarheid, doordat het aantal I/O kleiner dan bij PHP.
De frameworks die gebruikt worden voor het opzetten van een webserver zijn zo goed als gelijkwaardig in functionaliteit.

**Compartment managing**
Voor het managen van de compartments kan er voor een combinatie van docker en kubernetes worden gekozen. Op deze manier zijn alle voordelen van beide technieken van toepassing.

### 6.2.4 Conclusie

Om smart resourcesystemen schaalbaar en uitwisselbaar te maken is er naar verschillende aspecten gekeken: de architectuur, de programmeertaal & frameworks en compartment managing.

Voor de architectuur geldt dat de belangrijkste uitgangspunten schaalbaarheid en uitwisselbaarheid beide beter zijn vertegenwoordigd in een microservice achitecture, doordat dit horizontaal schaalt en een microservice alleen hoeft te voldoen aan een de communicatie interface met andere microservices. Daardoor is een microservice architecture een betere keuze voor smart resourcesystemen.

De programmeertalen met hun frameworks waaruit gekozen is, zijn: Kotlin met Spring en PHP met Symfony. Uit dit onderzoek blijkt dat Kotlin beter aansluit op de wensen van een smart resourcesysteem, voornamelijk omdat Kotlin minder I/O heeft dan PHP waardoor het schaalbaarder is.
Daarentegen hoeven niet alle microservices in deze taal geschreven te worden, maar dit project beperkt zich alleen tot Kotlin. Doordat Kotlin gebruikt zal worden voor de microservices zal ook Spring gebruikt worden.

Voor het managen van de gehele microservice cluster adviseert dit onderzoek dat er een combinatie van docker en kubernetes gebruikt wordt. Dit omdat op deze manier alle voordelen van beide technieken van toepassing zijn. Er zal echter geen gebruik worden gemaakt van docker swarm of compose.

Door deze technieken toe te passen in de software-oplossing kan er gemakkelijk geschaald worden. Ook kan een microservice vervangen worden, bijvoorbeeld in het geval dat er een andere manier van matching wordt gekozen.
Smart resourcesystemen kunnen dus schaalbaar en uitwisselbaar worden opgezet door microservices architecture gebruikmakend van Kotlin Spring boot in kubernetes met docker. Eventueel kunnen er frameworks gemaakt worden voor een specifieke microservice.

## 6.3 Onderzoek naar dataopslag

In de ontwerpfase van het afstudeerproject is er onderzoek gedaan naar de dataopslag, om een advies uit te kunnen brengen over welke dataopslagmethode gebruikt zou moeten worden in de software-oplossing.

### 6.3.1 Probleemanalyse en doelstelling

Reeds zijn er drie smart resourcesystemen - die ontwikkeld zijn door Recognize - in gebruik, die elk gebruikmaken van PostgreSQL als data opslag. Echter zet Recognize vraagtekens bij de schaalbaarheid van de huidige systemen en ze vermoedt dat dit komt door het gebruik van PostgreSQL, of SQL in het algemeen.
Het doel van dit onderzoek is om een passende dataopslagmethode te vinden, die de wensen van smart resourcesystemen behartigt. Daarom luidt de hoofdvraag van dit onderzoek: **Welke manier of manieren van dataopslag sluit(en) het beste aan op de wensen van smart resourcesystemen?**

### 6.3.2 Methodiek

#### 6.3.2.1 Beoordeling

Om verschillende technieken en manieren van dataopslag met elkaar te kunnen vergelijken, worden deze getoetst aan de hand van een aantal criteria. Elk criterium heeft een weging, evenredig aan hoe belangrijk dat criterium wordt geacht aan het systeem. Vervolgens krijgt elke optie voor elk criterium een score. De eindscore van een dataopslag is de som van alle scores maal de weging van de bijbehorende criterium: eindscore = $\Sigma$ (score x weging) voor elk van de criteria.

#### 6.3.2.2 Criteria en weging

De criteria waaraan de technieken worden getoetst zijn:

- Mate van omgang van locatiedata
- Schaalbaarheid van de dataopslag
- Omgang met transacties en locking
- Complexiteit in gebruik van de dataopslag

**Omgang met locatie data**
Heeft een weging van 5, omdat een van de basisonderdelen van de software-oplossing is dat het matcht op basis van locatie. Een goede omgang met locatiedata versimpelt het systeem, maar is geen requirement. Maar het is wel van belang dat het systeem überhaupt kan omgaan met locatiedata. Daarom is er gekozen voor een weging van 5.

**Schaalbaarheid van de dataopslag**
Heeft een weging van 9, omdat de software-oplossing ook op grote schaal goed zal moeten presteren. Dit is een van de requirements en is het belangrijkste in de context van een smart resourcesysteem van deze criteria. Idealiter schaalt de dataopslag horizontaal. Daardoor weegt de score van dit criterium zwaar mee, en is een weging van 9 gekozen.

**Omgang met transacties en locking**
Heeft een weging van 8, omdat het smart resourcesysteem moet kunnen garanderen dat een gebruikersactie ook daadwerkelijk wordt uitgevoerd in de database of aan de gebruiker kenbaar wordt gemaakt dat een actie niet is uitgevoerd. Ook de problemen die gepaard gaan met concurrent schrijven naar de database moeten worden afgevangen. Daardoor weegt dit criterium zwaar mee met een weging van 8.

**Complexiteit in gebruik van de dataopslag**
Heeft een weging van 8, omdat de software die gebruik maakt van de dataopslag uitbreidbaar moet zijn. Bij de weging geldt: een hogere score betekent dat het minder complex is. Recognize zal de engine moeten kunnen gebruiken en ook kunnen uitbreiden. Doordat de complexiteit indirect invloed heeft op de realisatie van toekomstige smart resourcesystemen, is er gekozen voor een weging van 8.

#### 6.3.2.3 Opties

De verschillende manieren van dataopslagmethodes zijn onder te verdelen in drie hoofdgroepen: SQL, denormalized SQL en NoSQL. Uit elk van de drie categorieën is een techniek gekozen die zal worden onderzocht:

- **PostgreSQL**. Een SQL database. DIt wordt momenteel gebruikt voor de bestaande smart resourcesystemen.
- **Elasticsearch**. Een noSQL dataopslag. Dit wordt nader onderzocht naar aanleiding van het verzoek vanuit de opdrachtomschrijving van Recognize.
- **PostgreSQL Citus Data**. Een denormalized SQL dataopslag. Dit is een extensie op postgres, speciaal voor schaalbaarheid. Daardoor lijkt dit een goede kandidaat voor een systeem dat by design schaalbaar moet zijn.

### 6.3.3 Resultaten

In tabel 1 staan de resultaten van het onderzoek weergegeven.
Per rij staat per dataopslagmethode de score en tussen haakjes de eindscore die de dataopslagmethode heeft gekregen, voor het desbetreffende criterium van die rij. Welk criterium dat is, staat in de eerste kolom. In de laatste rij staat de totaalscore.

| Criteria | PostgreSQL | Elasticsearch | Citus data |
|---|---|---|---|
| location data (5) | 8 (40) | 6 (30) | 8 (40) |
| schaalbaarheid (9) | 3 (27) | 8 (72) | 7 (63) |
| transacties en locking (8) | 9 (72) | 4 (24) | 9 (72) |
| complexiteit (8) | 9 (72) | 6 (48) | 4 (32) |
| Totaalscore | **211** | **182** | **207** |

Tabel 1, Scores per optie

### 6.3.4 Conclusie

Aan de hand van de gestelde criteria - omgang met locationdata, schaalbaarheid, omgang met transacties & locking en complexiteit in gebruik van de dataopslag - blijkt dat een normalized PostgreSQL database het meest aan de wensen van een smart resourcesysteem voldoet. Hoewel deze niet het beste schaalt van de onderzochte dataopslagtechnieken, wegen de de voordelen van omgang met locatie, de mogelijkheid om transacties uit te voeren en het voordeel in complexiteit op tegen het verlies in schaalbaarheid.

Normalized PostgreSQL met behulp van Citus en Elasticsearch leggen het beide af tegen Citus Data. In het geval van Elasticsearch komt dit door de omgang met locatiedata, die beduidend minder goed is dan PostGIS (Mulders, 2018), en doordat er geen transacties mogelijk zijn. Citus data legt het voornamelijk af op complexiteit. Citus data geeft aan in hun documentatie (Citus, n.d.) dat het geen goed idee is om Citus data te gebruiken wanneer “single node PostgreSQL” het systeem ook goed kan dienen.

Tot dusver heeft PostgreSQL bewezen de bestaande systemen goed te dienen in de drie bestaande smart resourcesystemen. En door middel van Materialized views kunnen reads in PostgreSQL sneller worden gemaakt, zonder het gebruik van Citus data, wanneer dat nodig blijkt te zijn. Daarentegen is er geen reden om aan te nemen dat dit in de nabije toekomst een probleem gaat vormen.

# 7. Eindproduct

Het eindproduct, bestaande uit de engine en het proof of concept, is het resultaat van het doorlopen van meerdere fases. Deze fases zijn te herleiden tot AORTA (Analyse-, Ontwerp-, Realisatie-, Test-, Acceptatiefase). Dit hoofdstuk zal voor elke fase van AORTA toelichten wat er is gedaan, hoe dit tot stand is gekomen en inzoomen op interessante bevindingen en resultaten.

## 7.1 Analyse

In de analysefase worden de eisen voor het eindproduct opgesteld in de vorm van requirements en wordt er op basis van die requirements een ontwerp van het systeem gemaakt. Bij dit ontwerp horen ook de keuzes van de technieken die gebruikt worden en globale architectuur.

### 7.1.1 Requirements

In het plan van aanpak zijn al requirement opgesteld. Dit is gebeurd aan de hand van de opdrachtomschrijving. Aan de hand van het eerste onderzoek, paragraaf 6.1 (Overlappingsonderzoek), zijn de initiële requirements uit het plan van aanpak verbeterd. Deze verbeterde requirements zijn vervolgens geprioriteerd door middel van de MoSCoW-methode. Om de prioriteit aan te geven binnen elke categorie, is er ook een business-rating (B-rating) aan elke requirement gegeven. Daarbij geldt: hoe hoger de B-rating, hoe hoger de prioriteit. 100 is de hoogst mogelijke rating en 0 de laagste.

| Requirement | Must have | Should have | Could have | Won't have | B-rating |
|---|---|---|---|---|---|
| De engine matcht vraag en aanbod op basis van: locatie, skills en beschikbaarheid. | X | | | | 1000 |
| De engine heeft een API om orders aan te maken. | | X | | | 700 |
| De engine kan statusupdates naar de eindgebruiker sturen. | | | X | | 300 |
| De aanbodbiedende partij kan haar beschikbaarheid aangeven door middel van een API. | X | | | | 800 |
| De engine houdt de voortgang bij van de afhandeling van de vraag. | | X | | | 600 |
| De engine moet schaalbaar zijn. | X | | | | 950 |
| De engine past authenticatie toe op alle API's. | X | | | | 750 |
| De engine is modulair opgezet, zodat elke module vervangbaar is. | X | | | | 900 |
| De engine kan een bestaand smart resourceproject migreren. | | | | X | 100 |
| De interfaces van de engine zijn | X | | | | 850 |

| | | | | | |
|---|---|---|---|---|---|
| gedocumenteerd. | | | | | |
| De engine maakt de keuzes en errors van het systeem inzichtelijk. | | X | | | 500 |
| De engine past logging toe op essentiële onderdelen. | | X | | | 550 |
| Er zijn projectspecifieke modules. | | | | X | 150 |
| De engine faciliteert koppelingen met boekhoudpakketten. | | | X | | 200 |
| Gebruikers van het systeem zijn hiërarchisch opgezet. | | | | X | 50 |
| De engine faciliteert koppelingen met SMS-diensten. | | | X | | 250 |
| De engine stuurt push notificaties naar de uitvoerende gebruiker bij het aanbieden van een opdracht. | | | X | | 350 |

*Tabel 2, Requirements*

Zowel de categorisering van de requirements in de MoSCoW-categorieën als de prioritering binnen deze categorieën, is gedaan op basis van het interview met de productowner van Utilio en in samenwerking met de opdrachtgever, die tevens de begeleider van dit afstudeerproject is binnen Recognize.

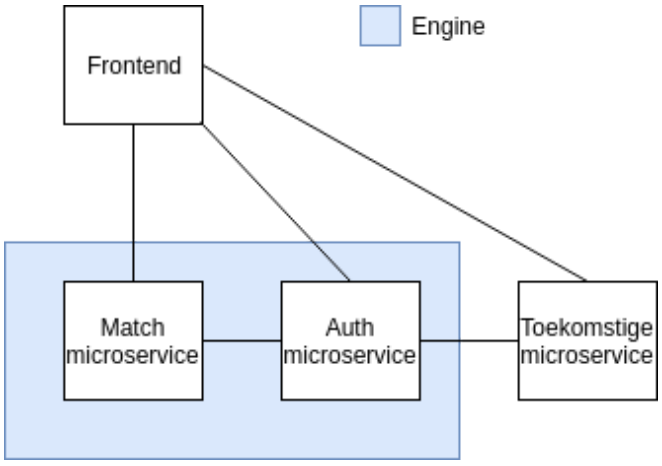
De requirements waarvan aangegeven is dat ze in de categorie “Must have” en “Should have” vallen, vormen samen de barebone van elk smart resourcesysteem. Dat is mede de reden dat deze requirements in deze categorieën vallen. Daarbij is de implementatie van deze requirements in een engine, hoewel een uitdaging, realistisch en haalbaar in het gegeven tijdsbestek van de afstudeeropdracht.

### 7.1.2 Ontwerp

Aan de hand van de requirements en het tweede onderzoek, paragraaf 6.2 (Onderzoek naar architectuur en technieken), is er een ontwerp gemaakt van hoe de engine globaal zou moeten werken, met de daarbij benodigde functionaliteit.
De engine zal uit microservices bestaan en daarmee de microservice architecture implementeren, met als voornaamste redenen dat deze architectuur horizontaal schaalt en uitbreidbaar is. Wanneer er extra functionaliteit toegevoegd moet worden, kan dat gedaan worden door een extra microservice toe te voegen aan het systeem.
De microservices zijn gemaakt in Kotlin, in het framework Spring en gebruiken Postgres in combinatie met PostGIS als dataopslag.

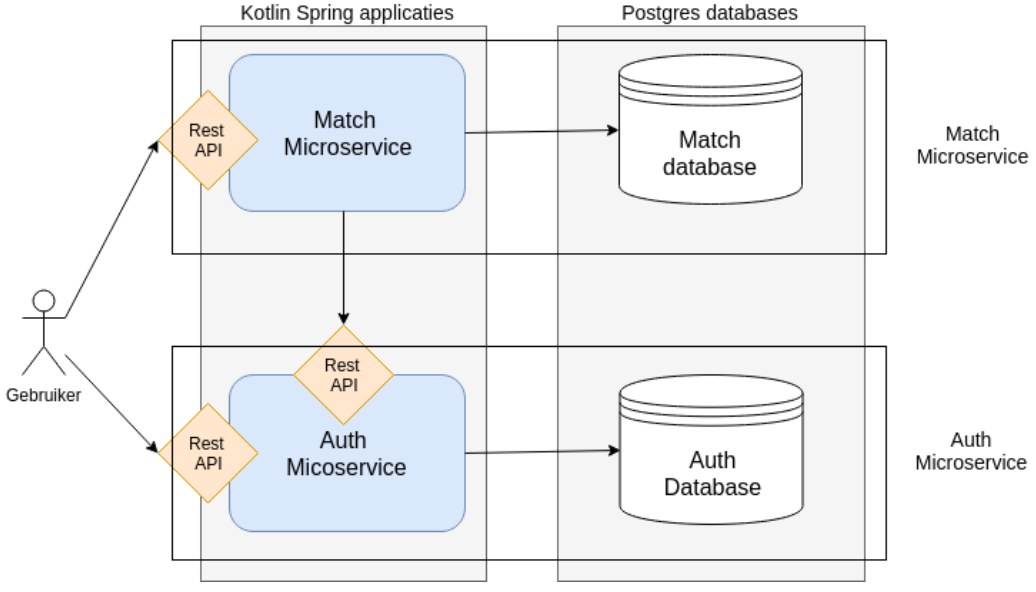
Doordat de microservice architecture by design een modulaire structuur heeft, zijn de services an sich naast schaalbaar, ook uitwisselbaar. Zolang de communicatie interfaces, de Rest API's, van een nieuwe microservice voldoen aan de specificaties van een bestaande microservice, kan de nieuwe service de bestaande vervangen.
Hoewel in dit project alle microservices in Kotlin en Spring zijn gemaakt en gebruikmaken van Postgres als dataopslag, kunnen in de toekomst ook andere programmeertalen, frameworks en databases gebruikt worden voor bijkomende of vervangende microservices.

Uiteindelijk is er tot het huidige ontwerp van de engine gekomen. In dit ontwerp bestaat de engine uit twee microservices, genaamd: Match en Auth. In figuur 2 is te zien hoe de engine in een compleet  smart resourcesysteem zou passen.




*Figuur 2, Schematische weergave van de engine binnen een smart resourcesysteem*

Deze services hebben elk hun eigen taak en zijn uitwisselbaar voor een andere service zolang deze dezelfde API's implementeren, zoals het hoort in een microservice systeem. De communicatie tussen de microservices en de buitenwereld gebeurt door middel van REST API's, die JSON als datarepresentatie gebruiken. JSON wordt standaard door Recognize gebruikt als datarepresentatie bij API's.
In figuur 3 is het ontwerp schematisch weergegeven in een diagram. De gebruiker links in het diagram is zowel de uitvoerende gebruiker, klant als beheerder.




*Figuur 3, High level ontwerp engine*

#### 7.1.2.1 Match

Match is de microservice waar de business logica van het smart resourceplatform zich bevindt. Dat wil zeggen: waar het daadwerkelijke bepalen van welke vraag met welk aanbod in contact wordt gebracht, en alle logica die daarbij komt kijken. Dit houdt onder andere de logica in die bepaalt wie van de users die de benodigde skills, locatie en beschikbaarheid heeft en de opdracht aangeboden krijgt. Ook handelt deze microservice het gros van de API calls af, op het authenticeren van gebruikers na.
Doordat elke microservice een eigen database heeft, is het niet mogelijk om deze microservice op te splitsen zonder behoorlijk in te leveren op de simpliciteit van de interactie tussen de microservices en dus ook op de algemene werking. Daarom is ervoor gekozen dit in één microservice te houden.
Een overzicht van alle taken die de Match microservice omvat:

- het beheren van skills van de uitvoerende gebruiker;
- de track en trace van een opdracht;
- het beheren van de beschikbaarheid van de uitvoerende gebruiker;
- het beheren van de locatie van de uitvoerende gebruiker;
- het maken van een opdracht;
- autorisatie van de gebruikers, niet te verwarren met authenticatie;
- het aanmaken van nieuwe gebruikers, samen met de Auth microservice.

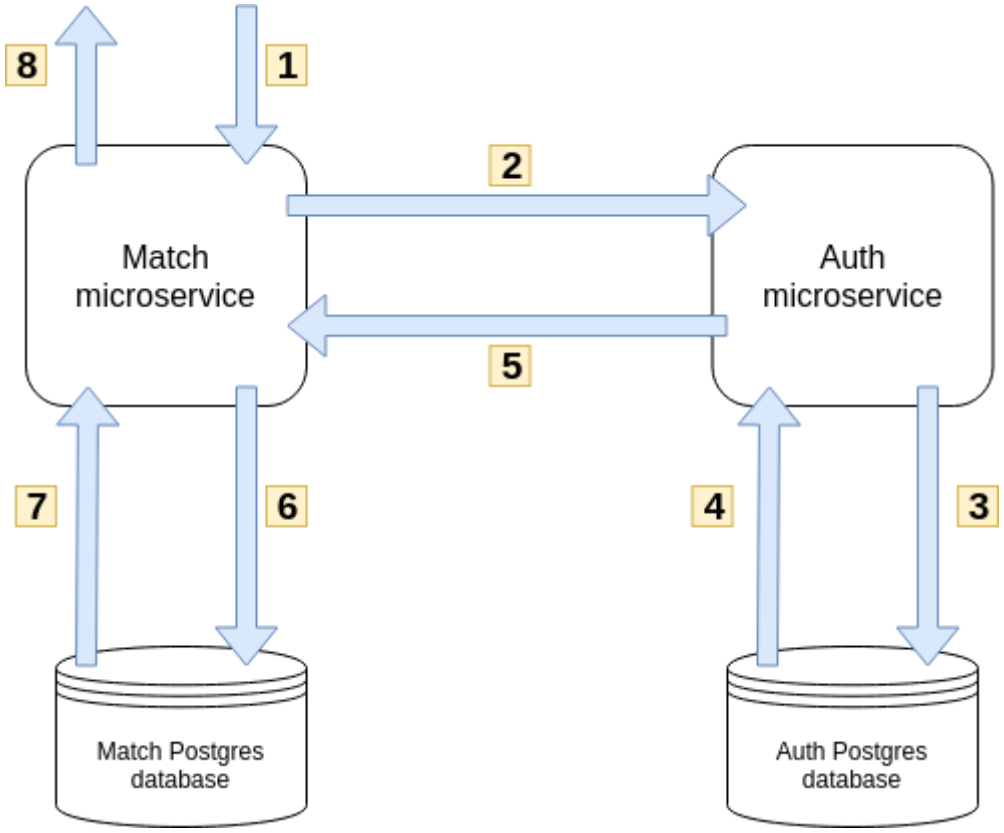
#### 7.1.2.2 Auth

De Auth microservice houdt zich alleen bezig met de authenticatie van de gebruikers die het smart resourcesysteem kent. De authenticatie van gebruikers wordt gedaan door middel van JSON web tokens (JWT) en refresh tokens. Om een JWT en refresh token te krijgen, kan een gebruiker zich ook authenticeren door middel van een gebruikersnaam en wachtwoord. Daarmee is dit dus een implementatie van Auth.
Er is voor een JWT-implementatie gekozen, omdat Recognize dit bij al haar applicaties gebruikt. Het maakt het mogelijk om informatie zoals rechten en rollen van een gebruiker uit te lezen vanuit de JWT zodat deze informatie overal gebruikt kan worden in het systeem, ook in de front end.
Een microservice is zelf verantwoordelijk voor de autorisatie, de rollen en rechten die een gebruiker heeft worden in de Auth microservice opgeslagen. Een microservice is zelf verantwoordelijk voor de interpretatie van deze rollen, de Auth microservice dicteert niet welke endpoints mogen worden benaderd met welke rollen.
De Auth microservice handelt de volgende taken af:

- login van gebruikers;
- het refreshen van tokens;
- het authenticeren van een request;
- het aanmaken van nieuwe gebruikers, samen met de Match microservice.

#### 7.1.2.3 Interactie tussen microservices

In figuur 4 is schematisch weergegeven hoe de twee microservices met elkaar communiceren bij een geauthenticeerde en geautoriseerde API call naar de Match microservice. Elke pijl representeert een actie, die onder het figuur is toegelicht.




*Figuur 4, Interactie tussen de microservices*

Toelichting op de acties uit figuur 4:

1. Gebruiker doet een request met JWT naar de Match service.
2. De Match service stuurt de JWT door naar de Auth service.
3. De Auth service decrypt de JWT en haalt de user op uit de database aan de hand van de username uit de JWT.
4. De database geeft de rollen en een reference ID van de user terug.
5. De Auth service stuurt de rollen en de reference ID naar de Match service.
6. De Match service checkt de autorisatie van de user op basis van zijn rollen en haalt de user informatie op uit zijn eigen database aan de hand van de reference ID van de user.
7. De database van Match geeft de user informatie, zoals skills, terug.
8. Match handelt de request af en geeft een response aan de gebruiker.

De authenticatie vindt dus altijd plaats in de Auth microservice, dat wil zeggen: het bepalen van wie de gebruiker is aan de hand van een JWT. De Auth microservice bepaalt de geldigheidsduur van de JWT, en deze kan worden veranderd door middel van configuratie. Dit is te lezen in paragraaf 7.2.4.5 (Configuratie).

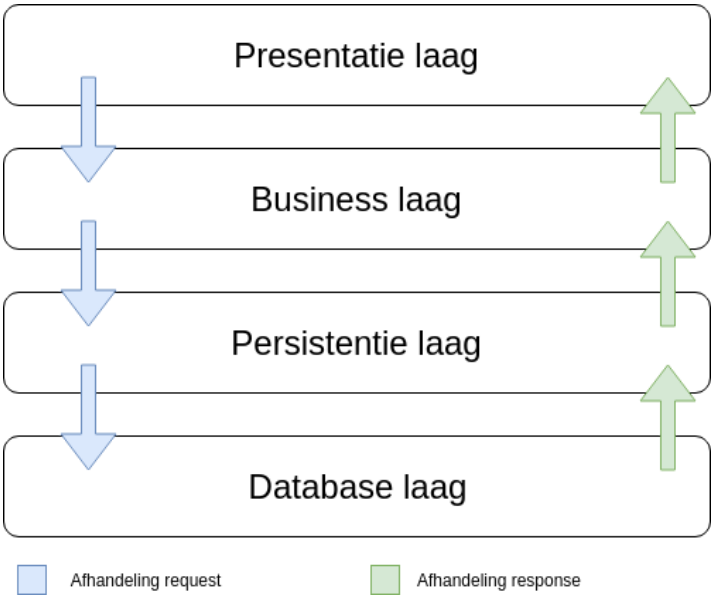
## 7.2 Realisatie

### 7.2.1 Werking van de engine

Het systeem brengt vraag en aanbod bij elkaar. De vraag is hierbij een opdracht die wordt gemaakt door een klant; wat deze opdracht inhoudt maakt voor het systeem niet uit. Een opdracht kan een aantal benodigde skills hebben. Om als uitvoerende gebruiker de opdracht aangeboden te krijgen, moet hij over deze skills beschikken. Naast de benodigde skills, moet de uitvoerende gebruiker beschikbaar zijn op het moment van de uitvoering van de opdracht. Het moment van uitvoering wordt aangegeven door de klant bij het maken van de opdracht. Ten slotte moet de uitvoerende gebruiker in de buurt zijn van de locatie waar de opdracht wordt uitgevoerd. Hoe ver 'in de buurt' is, is configureerbaar.
Wanneer een klant aangeeft dat hij de opdracht wil aanbieden aan geschikte uitvoerende gebruikers, kijkt de engine naar alle mogelijke opties en op basis van de hierboven gestelde eisen biedt de engine de opdracht aan, aan een aantal gebruikers. Het maximale aantal gebruikers dat dezelfde opdracht krijgt aangeboden is configureerbaar.
De uitvoerende gebruikers hebben dan de keuze om de opdracht te weigeren of te accepteren. Bij het accepteren geldt: wie het eerst komt, wie het eerst maalt.
Tijdens het uitvoeren van de opdracht, kan de uitvoerende gebruiker de voortgang aangeven. Bijvoorbeeld dat hij bezig is, of dat hij klaar is. Deze voortgang is op te vragen door de klant.
De beheerders kunnen skills maken, toewijzen aan en verwijderen van een uitvoerende gebruiker.

### 7.2.2 Architectuur van microservices

De twee microservices hebben dezelfde high level architectuur: een layered architecture, ook wel multitier architecture genoemd. In een dergelijke architectuur is een applicatie opgedeeld in verschillende lagen, die zich bezighouden met een globale taak. Een typische uitwerking van een multitier architectuur is weergegeven in onderstaand figuur, figuur 5.




*Figuur 5, Multitier architecture*

Deze gegeven layered architecture omvat zowel de client als server, echter bestaat de engine alleen uit een server. Daardoor is de presentatie layer - die normaliter aanwezig is in de architectuur - niet compleet aanwezig in de architectuur van de microservice, omdat de presentatielaag normaal gesproken refereert naar de client. Echter kan het presenteren van JSON in de API worden beschouwd als presentatielaag.
De onderdelen in de programmatuur van de microservices zijn in de structuur van Spring opgesteld. De belangrijkste elementen zijn:

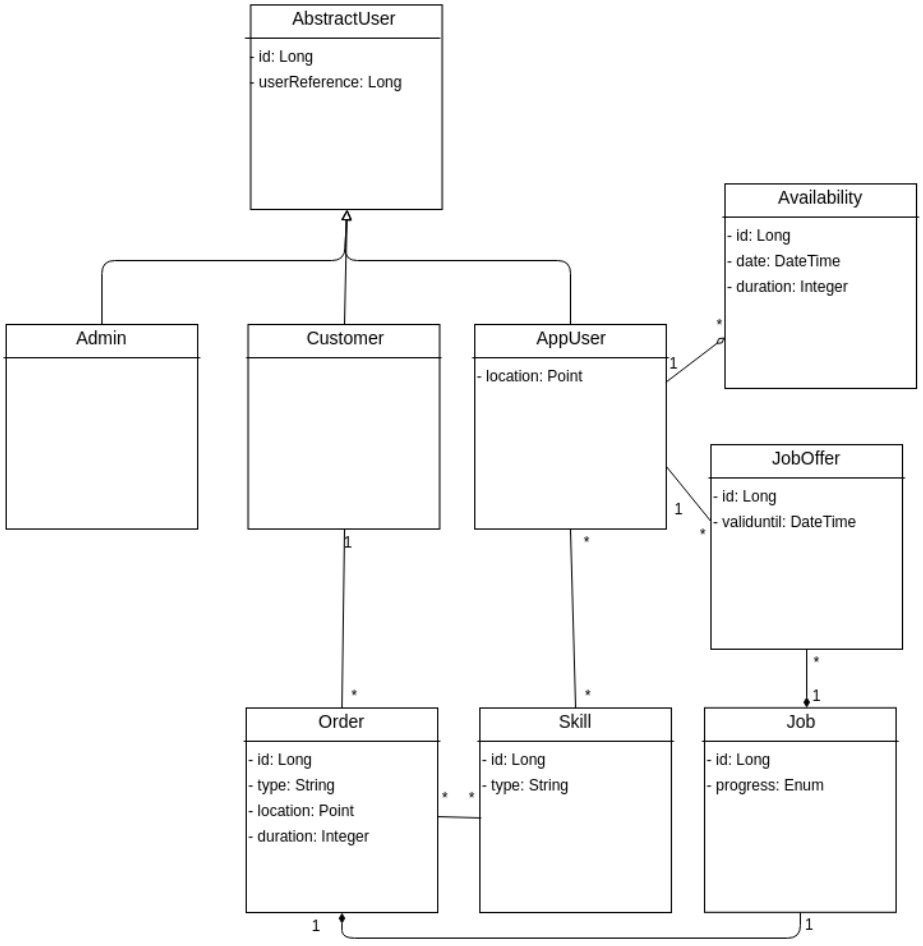
- **Controllers**. Controllers zijn onderdeel van de businesslaag, en zijn verantwoordelijk voor een deel van de logica en businessregels van de applicatie. Controllers handelen requests af. In de context van de engine zijn ze ook de presentatielaag, waarbij ruwe data wordt omgezet naar een gestandaardiseerde structuur en gepresenteerd wordt.
- **Services**. Services zijn, net als controllers, onderdeel van de businesslaag. Ze verschillen van controllers in dat ze niets van requests of responses afweten. Ze helpen echter wel indirect in de afhandeling daarvan. Services zijn mede verantwoordelijk voor de logica en businessregels van de applicatie.
  Soms worden services als aparte laag opgenomen in de layered architecture, maar dit is niet standaard. Doordat het gros van de requests geen gebruik maakt van een service is dit niet als aparte laag opgenomen in deze layered architecture.
- **Repositories**. Repositories vormen de persistentielaag en zijn daarmee verantwoordelijk voor de opslag van data in de database en het ophalen van data uit de database. De EntityManager (van hibernate ORM) is ook onderdeel van deze laag.
- **Entities**. Entities zijn de databaselaag in de programmatuur van de engine. Zij geven de structuur aan van de data die in de database staat, dat wil zeggen: de tabellen, kolommen en constraints op de kolommen.
  Hoewel Entities niks afweten van responses, zou er gesteld kunnen worden dat zij wel deel uitmaken van de presentatielaag in deze context, doordat entities geserialiseerd worden en gepresenteerd in de response. Hoe deze geserialiseerd worden, wordt bepaald door middel van annotations.

#### 7.2.2.1 Match

De Match microservice kent een aantal Entities, die elk een actie, gebruiker of object representeren van een smart resourcesysteem. Dit zijn:

- **AppUser**. Dit representeert een uitvoerende gebruiker.
- **Admin**. Dit representeert een beheerder.
- **Customer**. Dit representeert een klant.
- **Order**. Dit is een opdracht die een klant maakt.
- **Job**. Dit is een opdracht die uitvoerende gebruikers uitvoert.
- **Skill**. Uitvoerende gebruikers hebben skills, en opdrachten vereisen het hebben van skills voor het uitvoeren van de opdracht.
- **Availability**. Dit representeert de beschikbaarheid van een uitvoerende gebruiker.
- **JobOffer**. DIt is een opdracht die aangeboden wordt aan uitvoerende gebruikers. Wanneer deze wordt geaccepteerd wordt de uitvoerende gebruiker gekoppeld aan de opdracht.

Een overzicht van de entities en de relaties daartussen is gegeven in figuur 6.




*Figuur 6, Klassendiagram entities Match microservice*

#### 7.2.2.2 Auth

In tegenstelling to Match, kent de Auth microservice maar een enkele entity: User. Deze entity omvat alle drie de verschillende gebruikers van het systeem. Doordat Auth zich alleen bezighoudt met de authenticatie van de gebruikers, is het mogelijk om de verschillende gebruikers te abstraheren naar een enkele entity. De authenticatie gebeurt door middel van JWT tokens. Dit heeft als voordeel dat informatie zoals rechten en autorisatie uitgelezen kan worden vanuit de token zolang de JWT secret bekend is, zonder dat de Auth microservice daarbij gebruikt hoeft te worden.
De entity van de Auth microservice heeft de volgende eigenschappen:

- Username. De gebruikersnaam van de gebruiker, waarmee ingelogd kan worden.
- Password. Het wachtwoord van de gebruiker, waarmee ingelogd kan worden.
- Roles. Een lijst van rollen van de gebruiker, die wordt gebruikt voor autorisatie in andere microservices.
- RefreshToken. De refresh token waarmee de gebruiker zijn JWT token en refresh token kan verversen. Dit is onderdeel van de OAuth-implementatie.
- RefreshTokenValidUntil. Dit is de datum en tijd tot wanneer de refresh token gebruikt kan worden om de tokens te verversen. Dit is onderdeel van de OAuth-implementatie.

### 7.2.3 Systeemopzet

#### 7.2.3.1 Authenticatie

Voor de Authenticatie van de endpoints is Spring Security gebruikt. Om dit compatible te maken met de microservice architectuur is er een eigen implementatie gemaakt van de AuthenticationProvider: de FalconAuthenticationProvider. Deze class handelt in essentie de authenticatie van elk request af, door bij elk request de JWT door te sturen naar de Auth microservice om de gebruikersinformatie op te halen. De methode uit de provider die de communicatie met de Auth microservice afhandelt is gegeven in figuur 7.

```
private fun checkJWTToken(rawAccessJwtToken: RawAccessJwtToken): UserType {
    val response :Response  = khttp.post(
            headers = mapOf("Content-Type" to "application/json"),
            url = "http://$authHost/api/v1/authenticate",
            json = mapOf("token" to rawAccessJwtToken.token)
    )
    if (response.statusCode != 200) {
        if (response.statusCode > 403) {
            LOG.error("Auth service returned statuscode ${response.statusCode} with body: ${response.text}")
            throw HttpClientErrorException(HttpStatus.BAD_GATEWAY, "could not connect to Auth Service")
        } else {
            throw InsufficientAuthenticationException("Invalid authentication")
        }
    }

    return jacksonObjectMapper().readValue(response.text)
}
```

Figuur 7, FalconAuthenticationProvider checkJWTToken methode

#### 7.2.3.2 Autorisatie

De API van de match microservice is opgesplitst in vier delen: niet-geautoriseerde endpoints, uitvoerende gebruiker endpoints, klant endpoints en beheerder endpoint. Elk van deze endpoint-groepen hebben een andere autorisatie, die wordt gehandhaafd door rollen te vereisen aan de hand van het bijbehorende path. In figuur 8 is het deel van de “WebSecurityConfig” klasse gegeven die beschrijft welke autorisaties voor welke paden nodig zijn.

```
httpSecurity
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.NEVER).and()
    .csrf().disable() /// We don't need CSRF for JWT based authentication
    .cors().and()
    .addFilterBefore(CorsFilter(), SessionManagementFilter::class.java)
    .authorizeRequests()
    .antMatchers( ...antPatterns: "/api/v1/**/register").permitAll()
    .and()
    .authorizeRequests()
    .antMatchers( ...antPatterns: "/api/v1/appuser/**").hasAuthority( authority: "ROLE_APPUSER")
    .and()
    .authorizeRequests()
    .antMatchers( ...antPatterns: "/api/v1/customer/**").hasAuthority( authority: "ROLE_CUSTOMER")
    .and()
    .authorizeRequests()
    .antMatchers( ...antPatterns: "/api/v1/admin/**").hasAuthority( authority: "ROLE_ADMIN")
    .and()
    .authorizeRequests()
    .antMatchers( ...antPatterns: "/**").permitAll()
```

*Figuur 8, Authenticatie configuratie Match microservice*

### 7.2.4 implementatie

#### 7.2.4.1 Werking van het systeem

De engine heeft een aantal implementatiedetails die belangrijk zijn om een beeld te krijgen van de werking van het systeem. De drie voornaamste hiervan zijn: het matchen van een opdracht, de locatie-implementatie en het accepteren van de opdrachten.

Als er een opdracht wordt aangemaakt, dan wordt deze nog niet gelijk gematched. Dat wil zeggen dat de opdracht nog niet wordt aangeboden aan de uitvoerende gebruikers. Dit gebeurt pas nadat er een aparte API call wordt gedaan. Dit heeft als voornaamste reden dat verschillende smart resourcesystemen verschillende businessregels hierover kunnen hebben. Zo kan het bijvoorbeeld zijn dat de opdracht van een bepaalde dag pas op de dag zelf gematcht mag worden. Om die reden is het afgesplitst van het aanmaken van de opdracht, zodat dit flexibel blijft om later mee te werken.

De locatie van een uitvoerende gebruiker wordt gebruikt voor het matchen van een opdracht. In de engine is het geïmplementeerd als een standplaats. Dat wil zeggen dat er geen vervaldatum is en dat deze locatie niet vaak zal veranderen. Ook dit zal per smart resourcesysteem uiteindelijk verschillen, en daarom is er gekozen voor deze simpele oplossing. Standaard moet de standplaats binnen in een radius van 10 km van de locatie van de opdracht zijn.

Bij het accepteren van een opdracht door de uitvoerende gebruiker komen concurrencyproblemen kijken. Er moet gegarandeerd kunnen worden dat maar één enkele uitvoerende gebruiker een opdracht accepteert. Om dit te kunnen garanderen is er locking toegepast op rijniveau in de databasetabel. Dat betekent dat de database geen operaties toelaat op de rijen die gelockt zijn, totdat er een update heeft plaatsgevonden op die rijen. Omdat na het accepteren van een opdracht alle openstaande “JobOffers” worden verwijderd, fungeert dit als de update-actie op de rijen. In figuur 9 staat de implementatie inclusief de SQL queries gegeven.

```
@Query(value = "SELECT jo.id FROM job_offer jo WHERE jo.job_id = ?1 FOR UPDATE", nativeQuery = true)
fun lockJobOffersForJob(jobId: Long): MutableList<BigInteger>

// this is for preventing issues with locking and non existing jobOffers in the entity manager
// while simultaneously being an update to unlock locked rows
@Modifying
@Query(value = "DELETE FROM job_offer WHERE id = ?1", nativeQuery = true)
fun deleteJobOfferById(id: Long)
```

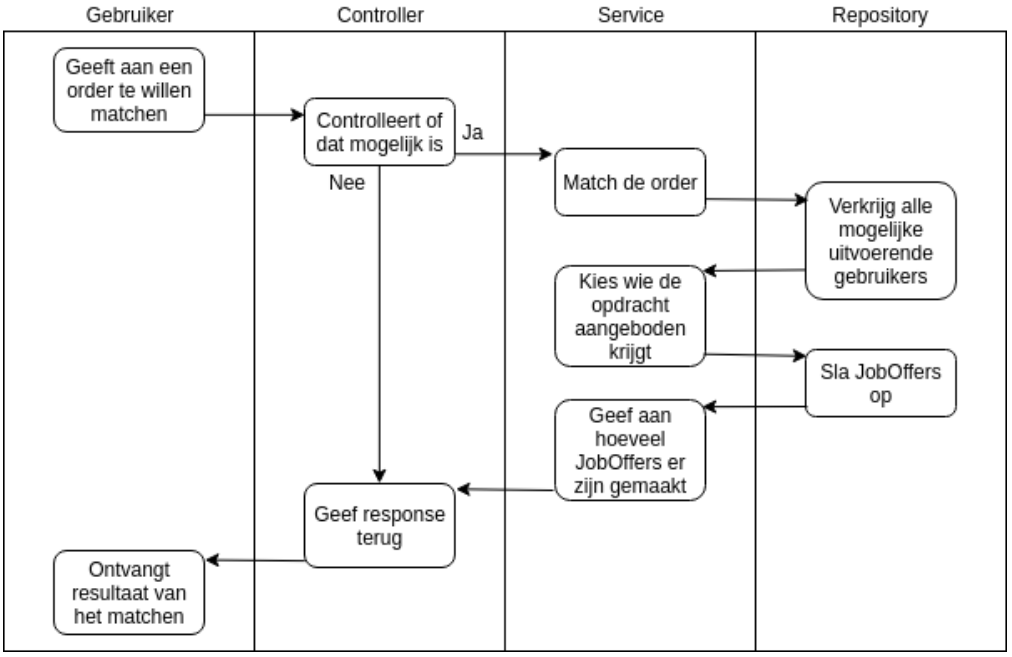
*Figuur 9, Locking en unlocking van JobOffers*

### 7.2.4.2 Design patterns

In de implementatie van de microservices, en dus de engine, zijn een aantal design patterns toegepast. Hieronder zijn er een aantal beschreven en is er toegelicht hoe de design pattern is toegepast.

**Multilayer architectuur**

De multilayer architecture design pattern is terug te vinden in beide microservices, en draagt bij aan de herbruikbaarheid van de code in de microservices. Een voorbeeld van hoe de multilayer architecture is toegepast in de engine is weergegeven in figuur 10; dit geeft een overzicht van de manier waarop het matchen van een opdracht in zijn werk gaat in relatie tot de verschillende lagen.




*Figuur 10, Swimlane diagram van het matchen van een opdracht*

Dit design pattern draagt bij aan de uitwisselbaarheid van de onderdelen van een individuele microservice. Door deze pattern toe te passen kunnen lagen veranderen zonder dat de andere lagen daar last van hebben of iets van af hoeven te weten. Er zou bijvoorbeeld een andere implementatie gemaakt kunnen worden om te bepalen welke uitvoerende gebruikers de opdracht aangeboden krijgen. De rest van de lagen hoeven daarvoor niet te veranderen. Meer over hoe multilayer architectuur is toegepast in de microservices, is te vinden in paragraaf 7.2.2 (Architectuur van microservices).

**Factory method**

Een voorbeeld van hoe de factory method design pattern terugkomt in de engine, is de logger. De logger library die de engine gebruikt, SLG4J, vereist dat de loggen door middel van factory method worden geïnstantieerd.
Naast de logging, is er ook een eigen implementatie van de factory method design pattern. Dit is de WithMockJWTUserSecurityContextFactory class, die wordt gebruikt voor de testen die authenticatie en autorisatie vereisen.
De factory method dient voor de uitwisselbaarheid van de implementatie van de applicaties. Door een interface te definiëren van zowel de factory als wat de factory bouwt, kan er een andere implementatie worden gegeven aan de factory en de uitkomst daarvan, zonder dat de rest van de codebase daar iets van merkt.

**Dependency Injection**

De dependency injection design pattern is een subset van inversion of control, waarbij objecten niet expliciet in de programmatuur worden geïnstantieerd. In plaats daarvan wordt er aangegeven aan welke interface een object moet voldoen en wordt dit als constructor parameter gedefinieerd. Dit object wordt vervolgens geïnstantieerd aan de hand van de configuratie. Voorbeelden van hoe dit terugkomt in de engine, zijn de repositories. Hiervan wordt alleen een interface gedefinieerd, zoals de order repository te zien in figuur 11.

```
@Repository
interface OrderRepository: JpaRepository<Order, Long> {
    fun getById(id: Long): Order?

    fun findByCustomer(job: Customer): List<Order>?
}
```

*Figuur 11, Definitie van de OrderRepository*

De order repository wordt vervolgens gedefinieerd als constructor parameter, zodat de repository wordt geïnstantieerd door Spring en kan worden gebruikt in de order controller, zoals te zien in figuur 12

```
class OrderController(
        private val entityManager: EntityManager,
        private val orderRepository: OrderRepository,
        private val matchRepository: MatchRepository
) : AbstractCustomerController() {
```

*Figuur 12, Gebruik van de OrderRepository*

Dependency injection draagt, net zoals de multilayer architecture en factory method, bij aan de uitwisselbaarheid van de applicatie. Het verschilt van de factory method doordat de implementatie van het instantiëren zelf moet worden gebouwd bij de factory methode. Bij dependency injection wordt dit gedaan door spring.

#### 7.2.4.3 Hosting

De engine wordt gehost in de kubernetes cluster van Recognize, in een aparte namespace. Deze namespace heeft een aantal pods, die die ieder een docker container bevatten. Elke microservice van de engine heeft meerdere replica's. Het aantal replica's is aanpasbaar, waardoor de schaalbaarheid wordt bereikt. De cluster route automatisch het verkeer op zo'n manier dat de workload van elke pod grofweg gelijk is.
Standaard heeft elke microservice twee replica's, om rolling deployment mogelijk te maken. Dit houdt in dat bij een update van een microservice de nieuwe versie eerst beschikbaar moet zijn, voordat de vorige versie wordt vervangen. Dit resulteert in geen downtime bij het updaten van een microservice.
De databases van de microservices worden niet in de cluster gehost, maar bij Azure. Dit is door VolkerWessels Telecom (waar Recognize onder valt) opgelegd en heeft enkel politieke redenen. Hiervan kan niet afgeweken worden.
Het deployen van microservices kan automatisch worden gedaan met continuous integration door een release branch aan te maken in de bitbucket repository.

#### 7.2.4.4 Logging

Om de keuzes die de engine maakt tijdens het matchen van vraag en aanbod inzichtelijk te maken, is logging toegepast. Bij elke keuze die het systeem maakt, wordt er gelogd wat de opties waren en welke het systeem vervolgens gekozen heeft. Ook worden de acties die de uitkomst van het matchen van vraag en aanbod beïnvloeden gelogd, omdat deze niet expliciet worden opgeslagen. Een aantal voorbeelden hiervan zijn:

- Welke uitvoerende gebruikers in aanmerking komen voor een opdracht.
- Welke uitvoerende gebruikers de opdracht krijgen aangeboden.
- Wanneer er een skill wordt toegewezen aan of verwijderd van een uitvoerende gebruiker.
- Wanneer er geen uitvoerende gebruikers gevonden zijn voor een opdracht.
- Wanneer een opdracht is geaccepteerd of afgewezen.
- Wanneer een opdracht van status veranderd.
- Wanneer een klant een opdracht annuleert.

Voor de logging is gebruik gemaakt van de SLF4J library, die standaard bij spring boot inbegrepen is. De logger kan door middel van factory method worden geïnstantieerd. Standaard logt spring naar de console. Dit kan veranderd worden door een custom logback-spring.xml. Een complete uitleg is te vinden in bijlage 5 (Systeemdossier) paragraaf 2.4 (Logging).

#### 7.2.4.5 Configuratie

De engine kan op verschillende onderdelen worden geconfigureerd. Deze niveaus zijn: hosting, bitbucket pipelines, logging en environment variabelen voor de microservices.

**Hosting**
De configuratie van de hosting van de microservices staan in de kubernetes map van de desbetreffende microservice codebase. Hierin bevinden zich drie files: de Dockerfile, de ingress.yaml en de engine.yaml. In de Dockerfile staat de configuratie van de docker container, zoals met welke JDK de applicatie wordt gecompileerd.
In de ingress.yaml staat alles dat te maken heeft met het verbinden van microservices en de manier waarop deze naar de buitenwereld worden getoond. Hieronder vallen bijvoorbeeld de routing naar de microservices en de security headers.
In de engine.yaml staat de configuratie met betrekking tot de microservice in de cluster. Hieronder valt onder andere het aantal replica's van een pod, of hoeveel memory deze mag gebruiken. Een complete uitleg over de configuratie van de hosting, staat in bijlage 5 (Systeemdossier) paragraaf 2.6.2 (Hosting).

**Bitbucket pipelines**
De bitbucket pipelines worden gebruikt voor continuous integration. In deze pipelines worden de volgende acties uitgevoerd bij elke codebase verandering:

- Automatische testen doorlopen;
- Builden van de microservice;
- Deployen naar de cluster;

Deze acties zijn te configureren in de bitbucket-pipelines.yaml, door aan verschillende situaties acties te hangen. Hoe dit te doen is, is te lezen in bijlage 5 (Systeemdossier) paragraaf 2.2.2 (Bitbucket).

**Logging**
De output van de logging is configureerbaar, wat wil zeggen dat de output in plaats van naar de console, ook naar een file weggeschreven kan worden. Deze file is configureerbaar door een custom logback-spring.xml te maken. Hoe dit te doen is en wat erin moet staan, is te lezen in bijlage 5 (Systeemdossier) paragraaf 2.4 (Logging).

**Environment variabelen**
De microservices behoeven een aantal parameters die variabel zijn per environment. Deze parameters zijn:

- database url, gebruikersnaam en wachtwoord;
- testdatabase url, gebruikersnaam en wachtwoord;
- JWT secret, de key waarmee de JWT's worden gesigned;
- JWT geldigheidsduur, hoe lang elke JWT geldig is;
- hoe lang een JobOffer geldig is;
- hoeveel uitvoerende gebruikers maximaal een opdracht aangeboden krijgen;
- de host van de Auth microservice;
- de 'tijdseenheid' die de engine hanteert in seconden;
- de maximale afstand die een uitvoerende gebruiker verwijderd mag zijn van de locatie van de opdracht in meters.

Deze parameters zijn opgenomen in de .env file in de codebase. Echter wordt deze file niet gebruikt in de kubernetes cluster, maar in plaats daarvan worden zogeheten secrets gebruikt. Deze secrets hebben dezelfde functie als de .env file, maar staan niet in de code

base omwille van security. Het is niet wenselijk dat bijvoorbeeld database wachtwoorden in de repository komen te staan. Daarbij verschillen deze parameters per environment en kunnen daarom niet in een bestand staan.

#### 7.2.4.6 Migrations

Voor het initieel opzetten van de databasestructuur, aan de hand van de entities, worden migrations gebruikt. Deze migrations kunnen worden gezien als een SQL script, die beschrijft hoe de database verschilt ten opzichte van de vorige versie. Bij het opstarten van de microservices wordt er gekeken of er nieuwe, nog niet uitgevoerde, migrations zijn. Wanneer dit het geval is, worden de migrations uitgevoerd. Op deze manier kan worden gegarandeerd dat de database altijd up to date is, mits de migrations zijn bijgewerkt.

De beheerder “default_admin” wordt aangemaakt in de migrations. Op deze manier is er altijd tenminste een beheerder. Dit is nodig doordat er beheerdersrechten nodig zijn om een beheerder aan te kunnen maken.

Voor de migrations maakt de engine gebruik van Liquibase. Liquibase maakt migrations aan de hand van de huidige database en de entities. Hoe de migrations te maken zijn en uit te voeren zijn, is te lezen in bijlage 5 (Systeemdossier) paragraaf 2.5 (Migrations).

### 7.2.5 Testen

In het plan van aanpak zijn drie soorten testen opgenomen: unittesten, systeemtesten en integratietesten.
Het systeem is op drie verschillende methodes getest: handmatig, automatisch en door middel van het integreren in een proof of concept.
Echter zijn er in het systeem geen unittesten gerealiseerd. Dit heeft als reden dat er geen ‘units’ zijn om in dergelijke testen te testen. De engine heeft geen onafhankelijke units die dit soort testen behoeven, de code wordt wel getest door API-testen. Deze verschillen van unit-testen doordat deze testen meerdere onderdelen van de engine tegelijk testen door een API call te doen en de response te vergelijken met wat het zou moeten zijn. Hoe dit in zijn werk gaat is te lezen in paragraaf 7.2.5.3 (Automatische testen).

#### 7.2.5.1 Integratietest

Integratietesten zoals in het plan van aanpak, is het proof of concept. Origineel was het plan om ten minste twee proof of concepts te maken, die gebruik maakten van de engine. Dit bleek echter te ambitieus, wat maakt dat er omwille van de tijd voor is gekozen om een proof of concept te maken.
Het proof of concept dat gebouwd is, Falonely, brengt jongeren met eenzame ouderen in contact. De jongeren zijn de uitvoerende gebruikers in dit smart resourcesysteem. De ouderen zijn de klanten en kunnen dus opdrachten aanmaken.

Wat het resultaat is van de integratietest, is te vinden in paragraaf 7.3.1 (Bruikbaarheid).

#### 7.2.5.2 Handmatige testen

Tijdens de ontwikkeling van de engine is er “handmatig” getest. Dat wil zeggen dat er niet-automatische API calls zijn gedaan, om de functionaliteit van de engine te vergelijken met wat het zou moeten zijn. Daarbij zijn zowel happy path als niet-happy pathscenario’s doorlopen. Dit is gedaan door middel van Postman; en de collectie die daarvoor is gebruikt is te vinden in de bijlage “Engine.postman_collection.json”. De requests in de collectie worden gedaan naar de lokale omgeving en zijn niet voorzien van een token om de request te authenticeren.
Hoewel elke functionaliteit van de engine handmatig is getest, zijn deze testen in het bijzonder nodig voor de authenticatie van de engine. Dit heeft als reden dat de automatische testen van de engine niet geschikt zijn om de authenticatie te testen, doordat het authenticeren van gebruikers van de enige in een aparte microservice plaatsvindt. Meer hierover is te lezen in paragraaf 7.1.5.2 (Automatische testen).

Een voorbeeld van de manier waarop de authenticatie is getest, is de login-actie. Hoewel dit een enkle API call is, is deze op meerdere manieren getest. De API verwacht een POST of GET request met daarin een JSON-object met een “username” en “password”. De server zou de access token en refresh token moeten teruggeven wanneer de combinatie van de gebruiker-credentials klopt. Deze API call is op de volgende manieren handmatig getest:

- Happy path. Een gebruiker logt in met een geldige gebruikersnaam en wachtwoord, zowel een POST als GET request.
- Null velden. Het username- en/of wachtwoordveld is null.
- Weggelaten velden. Het username- en/of wachtwoordveld mist in het JSON-object.
- Foute combinatie van credentials. Een geldige username en een geldig wachtwoord van een andere gebruiker.
- XML als content. In plaats van een JSON-object, wordt een login gedaan met een XML-object die geldige credentials bevalt.

#### 7.2.5.3 Automatische testen

Bij de Match microservice zitten een aantal automatische testen, dit zijn zogenoemde API-testen. Deze testen doen een API call en vergelijken de response daarvan met wat de uitkomst zou moeten zijn. Deze API-testen testen dus meerdere lagen, uit de layered architecture, van de engine tegelijk.
Deze automatische testen bestaan uit ‘good weather’- en ‘bad weather’-testen. Dat wil zeggen dat er zowel juiste requests worden getest als request die zouden moeten resulteren in een foutmelding. Alle API calls van de match microservice hebben tenminste één automatische test.
Een functionaliteit die niet getest kan worden met de integratietesten, is de authenticatie van gebruikers. Dit komt doordat de authenticatie in een andere microservice gebeurt dan waar de API calls verwerkt worden. De andere microservice weet niet wanneer er een test uitgevoerd wordt, en dus wanneer hij deze test data moet authenticeren. Het implementeren van workarounds hiervoor is geen optie. Daarom is er voor gekozen om in de autorisatielaag van de Match microservice users te authenticeren en daarmee ook te autoriseren door JWT tokens te mocken.

Tijdens het doorlopen van de integratietesten wordt er gebruik gemaakt van een testdatabase, die bij elke test opnieuw gevuld wordt met data door middel van fixtures. Na elke test wordt de database weer leeg gemaakt. Fixtures worden gemaakt door de Fixture interface te implementeren. Voor een test wordt de “up” methode aangeroepen, en na elke test de “down” methode. Dit wordt afgehandeld door de “FixtureLoader” service. De volgorde waarin de fixtures worden uitgevoerd hangt af van de “order” variable in de fixture.

```
interface Fixture {
    val order: Int

    fun up(context: FixtureContext)
    fun down()
}
```

*Figuur 13, De fixture interface.*

```
@Service
class FixtureLoader(private val fixtures: List<Fixture>) {
    fun up(): FixtureContext {
        val context = FixtureContext()
        for (fixture :Fixture in fixtures.sortedBy { it.order }) {
            fixture.up(context)
        }

        return context
    }

    fun down() {
        for (fixture :Fixture in fixtures.sortedByDescending { it.order }) {
            fixture.down()
        }
    }
}
```

*Figuur 14, FixtureLoader implementatie*

De API-testen zijn opgedeeld in API’s die ze testen. Dit zijn tevens de gebruikers van het systeem. De testen zijn gemaakt met behulp van de Spring boot starter test library, de standaard test library van Spring boot.

Een voorbeeld van een automatische test is het aanmaken van een opdracht door een klant, waarvan de code is gegeven in figuur 15. Met een annotatie is aangegeven welke eigenschappen de JWT moet hebben. De userReference in het bijzonder moet verwijzen naar een bestaande user in de database.

```
@Test
@WithMockJWTUser(username = "customer1", roles = [AbstractUser.ROLE_CUSTOMER], userReference = 1338001)
fun makeOrderTest() {
    val point :PointType = PointType().apply { this: PointType
        this.latitude = 1.337
        this.longtitude = 1.337
    }
    val biljartSkill :SkillType = SkillType().apply { this: SkillType
        type = "biljarten"
    }
    val order :OrderType = OrderType().apply { this: OrderType
        date = "2019-07-01 12:00:00"
        duration = 100
        location = point
        skills = listOf(biljartSkill)
        type = "potje 50 driebanders"
    }
    mockMvc
    .perform(MockMvcRequestBuilders.post( urlTemplate: "/api/v1/customer/order")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(order)))
    .andExpect(MockMvcResultMatchers.status().isOk)
    .andExpect(MockMvcResultMatchers.content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
    .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.job.progress", Matchers.equalTo(ProgressType.NEW.toString())))
    .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.skills.[0].type", Matchers.equalTo(biljartSkill.type)))
}
```

*Figuur 15, De automatische test voor het maken van een opdracht.*

Aan de hand van het doorlopen van de automatische testen, wordt een testrapport geregenereerd. De resultaten van het testrapport zijn gegeven in figuur 16. Hoe dit te doen is, is te vinden in bijlage 5 (Systeemdossier) paragraaf 2.2 (Testen runnen).


**Classes**

| Class | Tests | Failures | Ignored | Duration | Success rate |
|---|---|---|---|---|---|
| AdminIntegrationTest | 6 | 0 | 0 | 2.243s | 100% |
| AppUserIntegrationTest | 10 | 0 | 0 | 1.211s | 100% |
| CustomerIntegrationTest | 8 | 0 | 0 | 0.765s | 100% |

*Figuur 16, Automatische testresultaten*

### 7.2.6 Systeemdossier

Het systeem dossier is bedoeld voor developers die met de engine gaan werken. In dit document staat beschreven hoe het systeem werkt en hoe het gebruikt kan worden: van wat de verschillende microservices inhoudelijk doen, tot hoe de applicatie op te schalen is in de cluster. Het systeemdossier is gegeven in bijlage 5 (Systeemdossier).
Een overzicht van de onderwerpen die aan bod komen in het systeemdossier:

- functionaliteiten van de engine;
- werking van het systeem;
- development opzet;
- testen;
- deployment;
- logging;
- migrations;
- configuratie;
- gebruik van de applicatie;
- API-documentatie.

#### 7.2.6.1 API-documentatie

Voor het documenteren van de API's is gebruik gemaakt van apiary.io. Dit is een online tool speciaal voor het documenteren van API's. Er is hiervoor gekozen omdat er hierbij, in tegenstelling tot API-documentatie in een document, een duidelijke structuur is in de documentatie en het overzicht behouden blijft.

De complete API-documentatie is te vinden op https://engine9.docs.apiary.io/.
Een voorbeeld van hoe een API call gedaan kan worden, is het plaatsen van een opdracht door een klant.
**URL**: https://engine.test.recognize.hosting/api/v1/customer/order
**Method**: POST
**Headers**:

- Content-Type: application/json
- Authorization: Bearer <JWT>

**Request body**:

```
{
        “type”: <string>,
        “skills”: [
                {
                        “type”: <string>
                }
        ],
        “duration”: <int>,
        “location”: {
                “longtitude”: <double>
                “latitude”: <double>
        },
        “date”: <string>
}
```

**Responses**:

- 200. De order is geplaatst.
- 400. De order kon niet worden gemaakt op basis van de input, een veld ontbreekt of is niet juist.
- 401. Het request is niet geauthenticeerd.
- 403. De gebruiker waarmee de request is geauthenticeerd is mag geen order plaatsen.

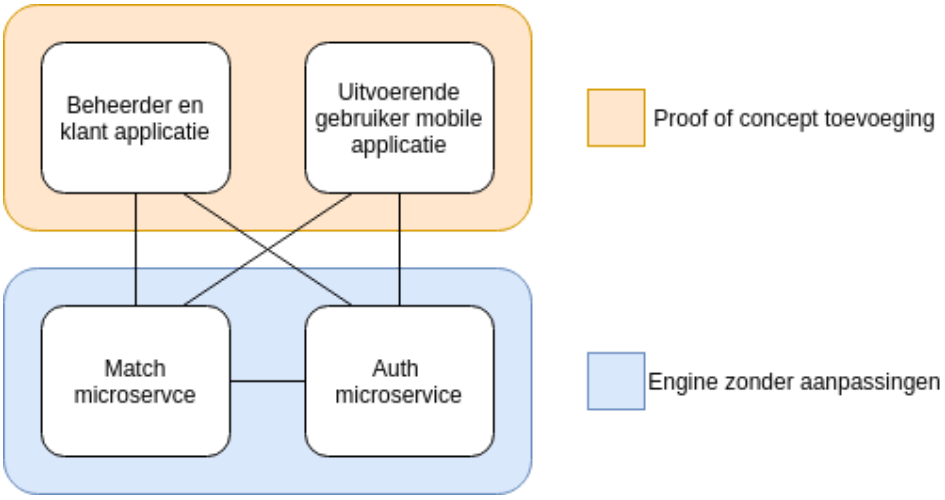
## 7.3 Acceptatie

De acceptatie van het eindproduct is onder te verdelen in drie categorieën: bruikbaarheid, schaalbaarheid en uitwisselbaarheid. Door deze drie onderdelen te behalen, wordt aangetoond dat het eindproduct acceptabel is.
Per onderdeel is aangegeven waarom hier wel of niet aan voldaan is.

### 7.3.1 Bruikbaarheid

Om de bruikbaarheid van de engine aan te tonen, is een proof of concept-project bedacht, waarin de functionaliteit van een typisch smart resourcesysteem aanwezig is.
Dit proof of concept is onder te verdelen in drie deelapplicaties, voor elk type gebruiker een. Wanneer de proof of concept aan de beschreven functionaliteiten voldoet, dan toont dit de bruikbaarheid van de engine aan en wordt dit geaccepteerd door Recognize.

Het proof of concept is een smart resourcesysteem dat eenzame ouderen in contact brengt met jongeren om samen tijd door te brengen. Het proof of concept gebruikt de engine zonder toevoeging van extra microservices; alleen de front end is specifiek voor dit smart resourcesysteem. Een schematisch overzicht met de interactie tussen de onderdelen van dit proof of concept is gegeven figuur 17.
Hiermee wordt aangetoont dat de basisfunctionaliteit aanwezig is in de engine. Echter zal de engine waarschijnlijk niet zonder aanpassingen of toevoegingen gebruikt worden in een echt smart resourcesysteem.




*Figuur 17, Proof of concept toevoegingen*

#### 7.3.1.1 Uitvoerendegebruikersapplicatie

De applicatie voor de uitvoerende gebruiker is een mobile app. Deze app is gemaakt in Angular en typescript, met behulp van Cordova en Ionic. Hierdoor is de applicatie een hybride applicatie, wat inhoudt dat het voor Android en IOS gebuild kan worden van dezelfde codebase. Echter is de IOS-applicatie niet getest, omdat deze alleen op een machine van Apple kan worden gebuild waarover ik niet beschik.
De mobile app implementeert alle functionaliteiten waarover de uitvoerende gebruiker beschikt in de engine. De gebruiker kan middels de app zijn locatie en beschikbaarheid aangeven, opdrachten accepteren en weigeren en de voortgang van opdrachten aanpassen.

#### 7.3.1.2 Beheerdersapplicatie

De beheerdersapplicatie is een typescript, een taal die compileert naar javascript, applicatie gemaakt in Angular en heeft als doel om skills te kunnen maken, toevoegen aan en verwijderen van een gebruiker. Maar voordat dat mogelijk is, moet de beheerder inloggen. Daarmee wordt alle functionaliteit geïmplementeert die de beheerder kan in de engine.

#### 7.3.1.3 Klantenapplicatie

De klantenapplicatie is, net als de beheerderapplicatie, een typescript applicatie gemaakt in Angular. Deze applicatie kan - naast het inloggen en registreren van gebruikers - orders aanmaken, orders matchen en de voortgang van een order opvragen. Dat is tevens alles wat een klant kan in de engine.

### 7.3.2 Schaalbaarheid

Een softwareprogramma is horizontaal schaalbaar wanneer het mogelijk is om het softwareprogramma te schalen door er extra replica's naast elkaar te draaien. Dat wil zeggen dat het mogelijk moet zijn om extra workload aan te kunnen, door een extra kopie van de applicatie tegelijkertijd met de originele te draaien zonder dat de gebruiker daar iets van merkt. De gebruiker, of andere systemen, voeren dezelfde acties uit als wanneer er een replica minder draait en krijgen ook hetzelfde resultaat.

De engine is verticaal schaalbaar, doordat er meerdere replica's naast elkaar kunnen draaien, die samen de workload dragen. Een extra replica van een microservice toevoegen kan worden gedaan door simpelweg de configuratie aan te passen. Deze microservices verwijzen naar dezelfde database. Dit levert geen performanceproblemen op, doordat de database niet de bottleneck is in het systeem. Hoe dit te doen is, is te lezen in bijlage 5 (Systeemdossier) paragraaf 2.6.2 (Hosting).

### 7.3.3 Uitwisselbaarheid

Doordat de verschillende interfaces van de microservices zijn gedocumenteerd, kan er een andere implementatie van een microservice worden gemaakt die een bestaande microservice kan vervangen. Zolang deze aan de interfaces voldoet van de bestaande microservice, zal de rest van de applicatie blijven werken en is er niets merken van de vervanging.
Een voorbeeld hiervan is de Auth microservice vervangen met een service die Azure AD authenticatie ondersteunt. Binnen recognize worden veel applicaties gebouwd die deze soort van authenticatie ondersteunen.
Daarnaast is de codebase van elke microservice uitwisselbaar opgezet, door middel van meerdere design patterns. Zo kan er een laag uit de multi layered architecture pattern, soortgelijk aan de microservice, worden vervangen door een laag met een andere implementatie zolang deze dezelfde functionaliteiten heeft als de originele laag.

Door dit alles toe te passen, heeft recognize bepaald dat daarmee de uitwisselbaarheid van de engine is aangetoond.

## 7.4 Aanbevelingen

Gedurende het afstudeerproject en in het bijzonder de realisatie van de engine, kwamen er zaken aan het licht die beter kunnen. Hieronder staan een aantal van deze zaken, onderverdeeld in categorieën. Bij elk punt is aangegeven wat er wordt verstaan onder 'beter' en waarom dit wordt aanbevolen.

### 7.4.1 Continuous integration

De continuous integration van de engine en de projecten die daar gebruik van maken, zou beter zijn door de toevoeging van een code style. Dit draagt bij aan de kwaliteit van de codebase, zoals beschreven in paragraaf 5.4 (Kwaliteit).

In het overgrote deel van de projecten bij Recognize wordt overigens wel een automatisch code style checking gedaan, echter is er voor Kotlin nog niks hiervoor aanwezig binnen het bedrijf. Dit zou een mooie toevoeging zijn voor de continuous integration.
Doordat dit invloed heeft op toekomstige projecten die gebruik gaan maken van de engine en dit buiten de scope van het project valt, is dit niet toegepast in het afstudeerproject.

### 7.4.2 Functionaliteit

Tijdens het maken van de engine kwamen er een aantal verbeterpunten aan het licht met betrekking tot de functionaliteit. Deze verbeterpunten zijn onder andere door gesprekken met de bedrijfsbegeleider aan het licht gekomen. Deze verbeterpunten staan los van de “Could haves” en de “Won’t haves” uit de requirements van dit project.
Van de hieronder genoemde verbeterpunten, wordt aanbevolen dat deze in een vervolgproject worden gerealiseerd. Dit zijn:

- Prijzen die verbonden zijn aan een opdracht. Momenteel wordt er niks met de kosten van het uitvoeren van een opdracht gedaan, of met de kosten van de uitvoerende gebruiker. Hoewel dit niet altijd van toepassing hoeft te zijn, denkt Recognize dat dit wel altijd van toepassing zal zijn.
- Meerdere Jobs bij een Order. Het zou kunnen voorkomen dat een opdracht in meerdere fases wordt uitgevoerd, door verschillende uitvoerende gebruikers. Hoewel dit niet in de huidige versie van de engine zit, wordt dit wel gefaciliteerd. De one-to-one relatie (zie figuur 6) tussen de Job en de Order entity kan gemakkelijk worden omgezet in een many-to-one relatie, zodat er later geen splitsing gemaakt hoeft te worden. Tevens leidt de functionaliteit niet onder deze one-to-one implementatie.
- Beschikbaarheid van een uitvoerende gebruiker op basis van de geaccepteerde opdrachten. Momenteel is de beschikbaarheid van de uitvoerende gebruiker zijn eigen verantwoordelijkheid. Echter zou dit ook op basis van de geaccepteerde opdrachten kunnen, bovenop de huidige functionaliteit rondom de beschikbaarheid.
- Opdrachten aanbieden aan de uitvoerende gebruiker op basis van een eigenschap. Momenteel werkt de engine zo dat op het moment dat er meer uitvoerende gebruikers in aanmerking komen voor een opdracht dan het configurabele maximum, er willekeurig bepaald wordt wie van deze gebruikers de opdracht daadwerkelijk krijgt aangeboden. Hoe dit anders bepaald moet worden, is per smart resourcesysteem verschillend en daarom niet meegenomen in de engine.
- Wachtwoord veranderen. In de huidige versie van de engine is er geen mogelijkheid om het wachtwoord van een gebruiker te veranderen. Dit is wel wenselijk in een smart resourcesysteem, omwille van de security. Ook zijn wachtwoordeisen mogelijk een goede functionaliteit, al kunnen deze eisen per systeem verschillen.
- Projectspecifieke regels omtrent de locatie van de uitvoerende gebruiker. In de engine wordt er enkel een locatie van de uitvoerende gebruiker opgeslagen. Deze wordt gebruikt als standplaats, wat wil zeggen dat deze niet de live locatie is van de gebruiker en dat deze ook geen vervaldatum heeft. Echter is het denkbaar dat er in een smart resourcesysteem andere businessregels hiervoor gelden.

### 7.4.3 Uitbreidingen

Naast de toevoeging van functionaliteit binnen de engine, kan er ook gebruik worden gemaakt van de engine, of van de microservices waaruit deze bestaat. Een voorbeeld van een uitbreiding op de gehele engine is een extra microservice die het maken van een opdracht grotendeels afhandelt. Op die manier is het bijvoorbeeld mogelijk om producten toe te voegen aan het smart resourcesysteem, of kunnen er standaard type opdrachten komen. Hoewel de match microservice niks van producten afweet, kan er een koppeling komen in de nieuwe microservice, die bijhoudt welke producten bij welke opdracht horen. Een soortgelijke koppeling wordt al toegepast op de gebruikers van het systeem.
Net als Match, kan ook de nieuwe microservice gebruik maken van de Auth microservice voor de authenticatie van de gebruikers.

Wat ook denkbaar is, is bijvoorbeeld een andere implementatie van de Auth microservice zodat de engine de authenticatie door middel van Azure AD laat lopen, zoals al gebeurt bij veel producten binnen Recognize.

# 8. Conclusie

Tijdens het afstudeerproject zijn er veel keuzes gemaakt, van implementatiekeuzes tot procesmatige keuzes. Deze keuzes hebben invloed gehad op de loop en resultaten van het project. In dit hoofdstuk worden de meest impactvolle keuzes uitgelicht en wordt er geconcludeerd wat het resultaat was van deze keuzes. Daarna worden op basis van het eindproduct een aantal aanbevelingen gedaan voor eventuele vervolgprojecten die te maken hebben met de engine.

## 8.1 Proces

In het proces van het afstudeerproject zijn er twee onderdelen die grote impact hebben gehad op het verloop van het project en de uitkomst daarvan. Dit zijn de projectaanpak en de kwaliteitsmaatregelen.

### 8.1.1 Projectaanpak

Als aanpak van het afstudeerproject is in het plan van aanpak aangegeven dat er voor een agile aanpak is gekozen. Deze aanpak zou dan worden gerealiseerd door de volgende elementen uit agile-methodes toe te passen:

- een backlog;
- incrementeel de producten opleveren;
- iteratief te werk gaan.

Hiervoor is gekozen om de koers van het project tijdens het project te kunnen bijstellen. Hiervan is bijvoorbeeld ook gebruik gemaakt toen de implementatie van de engine in twee bestaande smart resourcesystemen niet meer realistisch bleek.

Echter is er niet aan alle gestelde elementen compleet voldaan. Er waren duidelijke fases in het project die - hoewel deze soms overlapten - in een doorlopende reeks zijn afgerond, en niet op een iteratieve manier. Dit komt doordat er afhankelijkheden waren in fases uit de voorgaande fases. Bijvoorbeeld de afhankelijkheid van het onderzoek naar technieken die gebruikt worden in de engine. Zonder dit onderzoek afgerond te hebben, kon er geen doorslaggevende keuze worden gemaakt over de technieken, waardoor de daadwerkelijke realisatie van de codebase van de engine niet van start kon gaan.
Daardoor heeft de projectopzet dus ook in zekere zin wat weg van een watervalproject.

### 8.1.2 Kwaliteit

Om de uiteindelijke productkwaliteit te waarborgen zijn er een aantal procesmatige maatregelen getroffen. Van de vijf maatregelen zijn er hier drie beschreven, die het meest hebben bijgedragen aan de kwaliteit: automatische testen, continuous integration en pull requests.

#### 8.1.2.1 Automatiche testen

De Automatische testen van de enige dragen bij aan de huidige codebasekwaliteit, maar zorgen er ook voor dat toekomstige aanpassingen of toevoegingen niet de bestaande werking aantasten op onbedoelde manieren. Doordat de automatische testen bij elke codebase commit worden doorlopen met behulp van continuous integration, zijn fouten snel inzichtelijk en bespaart dit tijd op lange termijn.
De resultaten van de automatische testen zijn bijgevoegd in “automated-test-results.zip”.

#### 8.1.2.2 Continuous integration

De continuous integration die toegepast is tijden het project, heeft ervoor gezorgd dat er gemakkelijk getest, gebuild en gedeployed kan worden. Hoewel de opzet hiervan enige configuratie behoeft, heeft het wel bijgedragen aan de kwaliteit van het eindproduct. Daardoor is het een juiste keuze geweest om dit toe te passen.
Echter is er nog wel een verbeterpunt rondom de continuous integration, waarover te lezen is in paragraaf 8.3 (Aanbevelingen).

#### 8.1.2.3 Pull requests

De pull requests die zijn uitgevoerd hebben bijgedragen aan de codebasekwaliteit, maar ook aan de kwaliteit en validiteit van de implementatiekeuzes. Tijdens een pull request wordt er gekeken naar de codestyling, werking, structuur en leesbaarheid van de code. Daarnaast kunnen er vragen worden gesteld over de implementatiekeuzes, waarom iets wel of niet gebruikt is. Een voorbeeld hiervan zijn de migraties, die naar aanleiding van een pull request zijn geïmplementeerd.

## 8.2 Product

Aan de engine zijn een aantal eisen gesteld, onder andere in de vorm van requirements, te vinden in paragraaf 7.1.1 (Requirements). Alle requirements waarvan aangegeven staat dat deze behaald zouden moeten worden in het afstudeerproject, de must haves en should haves, zijn behaald. Hieronder worden de belangrijkste requirements uitgelicht en beschreven hoe deze zijn behaald in de engine.

### 8.2.1 Requirements

De drie belangrijkste requirements, de requirements met de hoogste business rating, van het eindproduct zijn:

- De engine matcht vraag en aanbod op basis van: locatie, skills en beschikbaarheid.
- De engine moet schaalbaar zijn.
- De engine is modulair opgezet zodat elke module vervangbaar is, verder in het document uitwisselbaar genoemd.

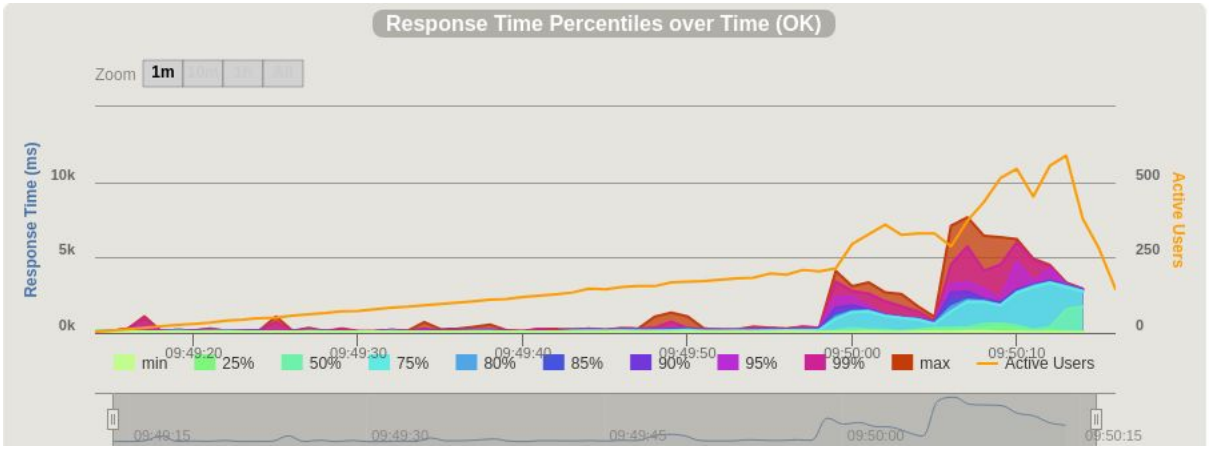
Het matchen van vraag en aanbod op basis van locatie, skills en beschikbaarheid is een functionele requirement en is daardoor ook te testen. Deze wordt getest in de automatische testen waarover meer te lezen is in paragraaf 7.2.5 (Testen).

De schaalbaarheid en uitwisselbaarheid van de engine zijn non-functional requirements. Om aan deze eisen te voldoen zijn er meerdere methodes en technieken toegepast, op meerdere niveaus van implementatie.
Op systeemarchitectuurniveau is er voor een microservice-architectuur gekozen. Hierdoor zijn de microservices zowel schaalbaar als uitwisselbaar. Schaalbaar omdat er voor elke microservice meerdere replica's gedraaid kunnen worden waardoor de workload wordt verdeeld onder de replica's, en uitwisselbaar omdat de microservices uitwisselbaar zijn voor andere microservices die dezelfde interfaces implementeren.
Op implementatieniveau zijn er design patterns toegepast, zoals multi layered architecture en dependency injection, om ook de onderdelen waaruit de microservices bestaan uitwisselbaar op te zetten.

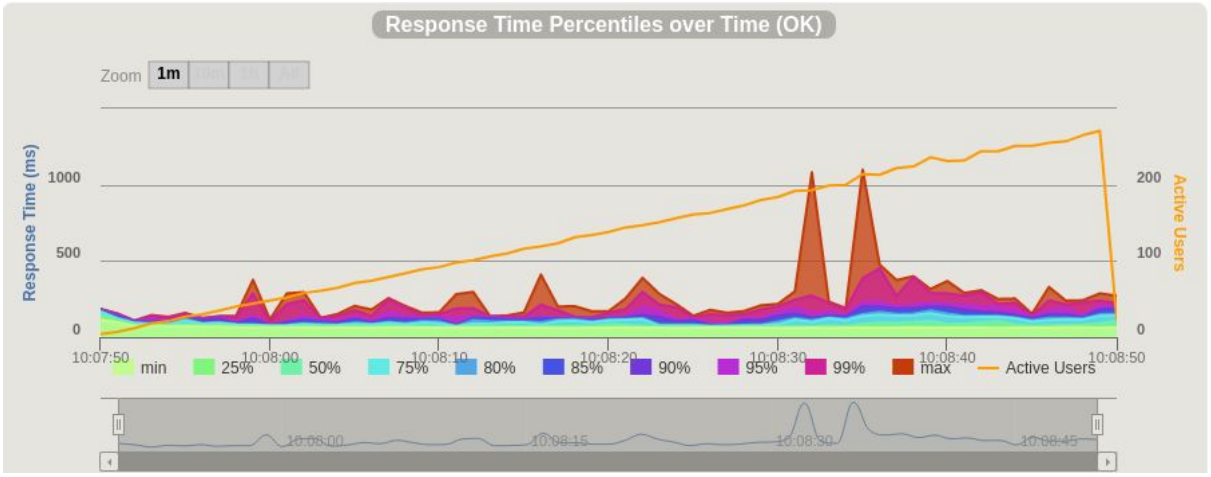
Daarnaast is het mogelijk om door middel van configuratie de applicatie te schalen. Doordat de engine in een kubernetes cluster wordt gedraaid, kan het aantal replica's van een microservice worden geconfigureerd. Doordat de engine gebruik maakt van de continuous integration van bitbucket, kan de configuratie worden gedaan door een bestand aan te passen. Hoe dit te precies te doen is, is te lezen in het systeemdossier paragraaf 2.6.2 (Hosting).

Dat de engine horizontaal schaalt is te zien aan de van de loadtesten die zijn uitgevoerd met behulp van Gatlin. Deze testen zijn performancetesten en bootsen een configurabel aantal gebruikers na. De loadtesten die zijn uitgevoerd duren elk grofweg een minuut, en gedurende die minuut wordt het aantal gebruikers die de engine gebruiker van 1 tot aan het maximum opgevoerd. Er zijn zes testen uitgevoerd die kunnen worden vergeleken: 100, 150 en 250 maximale gebruikers, bij een engine met 2 en 4 replica's. In figuur 18 en 19 is goed

te zien dat de 250 gelijktijdige gebruikers voor vertraging zorgt bij de engine met 2 replica's, terwijl 4 replica's van de engine er weinig last van hebben. In de grafiek worden de percentielen getoont van de responstijd van een request naar de engine, de "active users"-lijn bevat niet de daadwerkelijke gebruikers, maar dit zijn het aantal gebruikersrequesten die nog geen response hebben.




*Figuur 18, Responstijd tegenover gelijktijdige gebruikers bij een engine met 2 replica's.*




*Figuur 19, Responstijd tegenover gelijktijdige gebruikers bij een engine met 4 replica's.*

Alle resultaten van alle uitgevoerde testen zijn te vinden in de bijlage "load-test-results.zip". Om de resultaten te tonen, moet de index.html worden geopend in de desbetreffende directory. Er is tevens een zevende testresultaat gegeven: 450 gelijktijdige gebruikers bij een enige met 4 replica's.

### 8.2.2 Bruikbaarheid

Om de bruikbaarheid van de engine aan te tonen, is een proof of conceptproject bedacht, waarin de functionaliteit van een typisch smart resourcesysteem aanwezig is.
Het smart resourcesyteem, Falonely, is een systeem dat jongeren met te veel vrije tijd in contact brengt met eenzame ouderen, om samen tijd door te brengen.
Dit proof of concept is onder te verdelen in drie deelapplicaties, voor elk type gebruiker een.
Dit proof of concept beschikt over de beschreven functionele requirements en toont daarmee de bruikbaarheid van de engine aan.

## 8.3 aanbevelingen

Op basis van bevindingen tijdens het uitvoeren van het afstudeerproject en gesprekken met de bedrijfsbegeleider, worden er aanbevelingen gedaan voor toekomstige projecten die gebruik maken van de engine, of die de engine uitbreiden. Dit is onder te verdelen in drie categorieën: continuous integration, functionaliteit en uitbreidingen.

### 8.3.1 Continuous integration

De continuous integration van de engine en de projecten die daar gebruik van maken, zouden beter kunnen door de toevoeging van een code style check, zoals de meeste andere projecten binnen Recognize al hebben. Echter zal hiervoor wel eerst een techniek en een specifieke code style moeten worden bepaald. Dit draagt bij aan de kwaliteit van de codebase, zoals beschreven in paragraaf 5.4 (Kwaliteit).

### 8.3.2 Functionaliteit

Tijdens het maken van de engine, kwamen er een aantal verbeterpunten aan het licht met betrekking tot de functionaliteit. Deze verbeterpunten zijn onder andere door gesprekken met de bedrijfsbegeleider aan het licht gekomen. Deze verbeterpunten staan los van de “Could haves” en de “Won’t haves” uit de requirements van dit project.
Echter zijn de meeste functionaliteiten afhankelijk van de businessregels van het smart resourcesysteem. De enige twee functionaliteiten die geïmplementeerd zouden kunnen worden in de engine zijn dat de beschikbaarheid van een uitvoerende gebruiker ook op basis van de aangenomen opdrachten gaat en dat het wachtwoord van een gebruiker veranderd kan worden. De eerste functionaliteit wil zeggen dat een uitvoerende gebruiker niet twee opdrachten kan hebben met overlapping in de tijd.
De andere, businessregelafhankelijke, uitbreidingen in de functionaliteit zijn te vinden in paragraaf 7.4.2 (Functionaliteit).

### 8.3.1 Uitbreidingen

Naast de toevoeging van functionaliteit aan de microservices van de engine, kan er ook gebruik worden gemaakt van de engine of van de microservices waaruit deze bestaat. Een voorbeeld van een uitbreiding op de gehele engine is een extra microservice die het maken van een opdracht grotendeels afhandelt. Op die manier is het bijvoorbeeld mogelijk om producten toe te voegen aan het smart resourcesysteem, of kunnen er standaard type opdrachten komen. Hoewel de Match microservice niks van producten afweet, kan er een koppeling komen in de nieuwe microservice die bijhoudt welke producten bij welke opdracht 7.2.6 Systeemdossier horen. Een soortgelijke koppeling wordt al toegepast de gebruikers van het systeem, waarbij de Auth microservice alleen de benodigde informatie opslaat. Net als Match, kan ook de nieuwe microservice gebruik maken van de Auth microservice voor de authenticatie van de gebruikers.

Wat ook denkbaar is, is bijvoorbeeld een andere implementatie van de Auth microservice zodat de engine de authenticatie door middel van Azure AD laat lopen, zoals veel systemen binnen Recognize dat doen.

## 8.4 Reflectie

Gedurende ongeveer een half jaar heb ik gewerkt aan het project dat in dit document is beschreven. De reflectie op dit project is te vinden in bijlage 4 (Reflectie), waarin ik inga op mijn ervaringen en belevenissen die ik heb ondervonden tijdens het afstuderen. In dit document wordt er ook gereflecteerd op de keuzes die zijn gemaakt en het afstudeertraject in het algemeen.

# 9. Versiebericht

| Versie | Omschrijving | Datum |
|---|---|---|
| 0.1 | Initiële opzet van de hoofdstukken en korte beschrijving van wat erin moet staan. | 04-26 |
| 0.2 | Verbeterde indeling van de hoofdstukken een grove schets van de inhoud van elk kopje. Eerste versie van de opdrachtomschrijving gemaakt en onderzoeken deels geschreven. | 05-15 |
| 0.3 | Verdere uitwerking van onderzoeken en begin van de uitwerking van het process en eindproduct. | 05-19 |
| 0.4 | Verbeteringen aangebracht op basis van feedback. Verdere uitwerkingen van eindproduct en conclusie. | 05-24 |
| 1.0 | Verbeteringen aangebracht op basis van feedback. Afronding van hoofdstuk eindproduct. Reflectie en samenvatting geschreven. Concept versie. | 06-03 |
| 1.1 | Verbeteringen aangebracht op basis van feedback. Bijlagen opgenomen in het verslag. Meerdere figuren toegevoegd ter verduidelijking. Loadtestresultaten opgenomen. | 06-17 |

*Tabel 3, Versiebericht*

# 10. Literatuurlijst


Mulders, S. (2018, June 2). Performing geospatial queries at scale. Retrieved May 31, 2019, from https://medium.com/geophy-hq/performing-geospatial-queries-at-scale-7b64795d7704

Elastic. (n.d.). Limiting Memory Usage | Elasticsearch: The Definitive Guide [2.x] | Elastic. Retrieved May 31, 2019, from https://www.elastic.co/guide/en/elasticsearch/guide/current/_limiting_memory_usage.html

Elastic. (n.d.-a). Elasticsearch SQL: Query Elasticsearch Indices with SQL | Elastic. Retrieved May 31, 2019, from https://www.elastic.co/products/stack/elasticsearch-sql

Citus. (n.d.). What is Citus? — Citus Docs 8.2 documentation. Retrieved May 31, 2019, from https://docs.citusdata.com/en/stable/get_started/what_is_citus.html

# 11. Bijlagen

De bijlagen zijn opgesplitst in twee delen: de bijgevoegde bestanden en de in het document opgenomen bijlagen. De bijlagen die zijn opgenomen in dit document, zijn belangrijke documenten. De inhoud hiervan schetst de context van de software oplossing en geeft uitleg over het afstudeerproject, maar past niet meer in het verslag. Deze documenten zijn als paragrafen opgenomen in dit hoofdstuk.
De bijgevoegde bestanden zijn de minder belangrijke documenten, of kunnen niet in dit verslag worden weergegeven. Een voorbeeld van dit laatste is de codebase van de engine.
De bijgevoegde bestanden zijn:

- Uitwerking-interview-Timo.pdf
- Tijdsverloop-gantt-chart.pdf
- Plan van aanpak.pdf
- Engine.zip
- Engine.postman_collection.json
- load-test-results.zip
- automated-test-results.zip

# 11.1 Bijlage 1: Overlappingsonderzoek resultaten

## Probleemanalyse

Dit onderzoek dient uit te zoeken wat de overlapping van de bestaande smart resourcesystemen Miles, Traffer en Utilio zijn en welke functionaliteit hieruit geabstraheerd kan worden en dus in de de software-oplossing moet worden opgenomen.
Er zijn globale requirements opgesteld van overkoepelende functionaliteit, waarvan duidelijk is dat er overlap aanwezig is. Hoeveel en in hoeverre dat geabstraheerd kan worden zal in dit onderzoek worden uitgezocht.
Naast de functionaliteiten die aanwezig zijn bij alle bestaande systemen, zijn er mogelijk ook functionaliteiten die gewenst zijn in toekomstige smart resourcesystemen, maar die niet in elk van de bestaande systemen aanwezig is.

## Onderzoeksvragen

De kern van het onderzoek is om te bepalen welke functionaliteit in de software-oplossing zal worden opgenomen. Daarom luidt de hoofdvraag:
**Welke functionaliteiten kunnen worden geabstraheerd uit de bestaande systemen, en zijn kandidaat om geïmplementeerd te worden in de software-oplossing?**

De hoofdvraag zal worden worden beantwoord aan de hand van de volgende drie deelvragen:

1. Hoe werkt elk van de smart resourcesystemen functioneel?
2. Welke van de functionaliteiten komen overeen in tussen de verschillende systemen?
3. Welke van de andere functionaliteiten zijn tevens gewenst in de software-oplossing?

## Resultaten

### Werking van de systemen

Voor elk van de smart resourcesystemen is een algemene beschrijving gegeven, met daarbij wat het met het systeem kan worden gedaan. Alleen het happy path wordt beschreven, geen randgevallen of uitzonderingen, om de scope van het onderzoek klein en doelgericht te houden. Er is onderscheid gemaakt tussen de verschillende gebruikers van het systeem, zodat er een beeld wordt gevormd van alle perspectieven.

**Miles**

Miles is een platform waar pakketjes snel kunnen worden bezorgd. Iemand kan aangeven dat hij een pakketje wil bezorgen. Er wordt dan een order aangemaakt, waarna het systeem op basis van de huidige locatie of standplaats bepaalt welke koerier in aanmerking komt voor de opdracht.
De koerier krijgt een push melding op zijn telefoon dat hem een opdracht is aangeboden. Wanneer hij deze accepteert, kan hij het pakketje bezorgen en krijgt hij hier automatisch voor betaald. Dit gebeurt door middel van een integratie met een boekhoudpakket.

**Beheerders**

- Kunnen orders aanmaken in backoffice.
- Keuren koeriers en koeriersbedrijven goed.
- Kunnen skills aan koeriers toewijzen.
- Kunnen koeriers op non-actief zetten.
- Kunnen beheerders beheren.
- Kunnen wachtwoorden van regelaars veranderen.

**Koeriers**

- Kunnen beschikbaarheid aangeven.
- Kunnen zendingen accepteren of afwijzen.
- Ronden zendingen af.
- Hebben skills.
- Kunnen locatie aangeven.
- Krijgen een push notificatie wanneer een opdracht is aangeboden.

**Klanten**

- Moeten tekenen voor het ophalen en het afleveren van het pakketje.
- Kunnen zendingen aanmaken.

**Traffer**
Traffer is een platform waar opdrachtgevers en verkeersregelaars met elkaar in contact worden gebracht. Wanneer iemand een verkeersregelaar nodig heeft, dan kan diegene een opdracht aanmaken via de website, app of doormiddel van een beheerder te benaderen. Een opdracht heeft een tijdsvenster waarin regelaars nodig zijn, en welke verschillende regelaars nodig zijn (Coördinatoren, projectleiders, stewards, teamleiders, verkeersregelaars).
Voordat een regelaar mag regelen, moet de regelaar aantonen dat hij gekwalificeerd genoeg is door middel van certificaten. Een beheerder kan vervolgens bevestigen dat een regelaar over de benodigde kwalificaties beschikt.
Wanneer een opdracht in het systeem komt, komt het bij een planner (een van de beheerderlevels) terecht. Deze koppelt vervolgens regelaars aan de opdracht. Het systeem biedt handvatten hiervoor in de vorm van automatische filtering op regelaars die voldoen aan de eisen.

**Beheerders**
- Kunnen opdrachten aanmaken.
- Moeten opdrachten inplannen.
- Hebben verschillende levels in bevoegdheden.
- Maken projecten aan in de BackOffice.

**Verkeersregelaars**
- Kunnen beschikbaarheid in de week aangeven.
- Moeten aangeven wanneer ze starten en stoppen met de opdracht.
- Kunnen opdrachten accepteren of afwijzen.
- Hebben skills.
- Moeten locatie aangeven.

**Klanten**
- Kunnen opdrachten aanmaken.

**Utilio**
Utilio is een smart resourcesysteem voor monteurs. Bedrijven kunnen door middel van een API-koppeling opdrachten inschieten. Het systeem kijkt welke monteurs de benodigde “skills” hebben om de opdracht uit te voeren en wie beschikbaar is. Er wordt een selectie gemaakt van vijf monteurs die de opdracht aangeboden krijgen. De eerste monteur die de opdracht claimt, krijgt daadwerkelijk de opdracht.
Skills kunnen worden toegewezen aan een monteur door een beheerder. Dit wordt gedaan nadat de monteur de benodigde trainingen succesvol heeft afgerond.
Bij het uitvoeren van de opdracht kan de monteur foto’s maken van het verrichte werk, om aan te tonen dat het werk daadwerkelijk gedaan is.
Vervolgens krijgt de monteur per maand een overzicht van de verrichte werkzaamheden en krijgt hij daarvoor automatisch betaald. Dit gebeurt door middel van een integratie met een boekhoudpakket.

**Beheerders**
- Kunnen orders aanmaken in de backoffice.
- Keuren monteurs goed.
- Wijzen machtigingen en skills aan monteurs toe.
- Kunnen monteurs op non-actief zetten.
- Maken producten aan (in de backoffice), die geïnstalleerd kunnen worden.
- Maken klanten aan, die opdrachten kunnen inschieten.

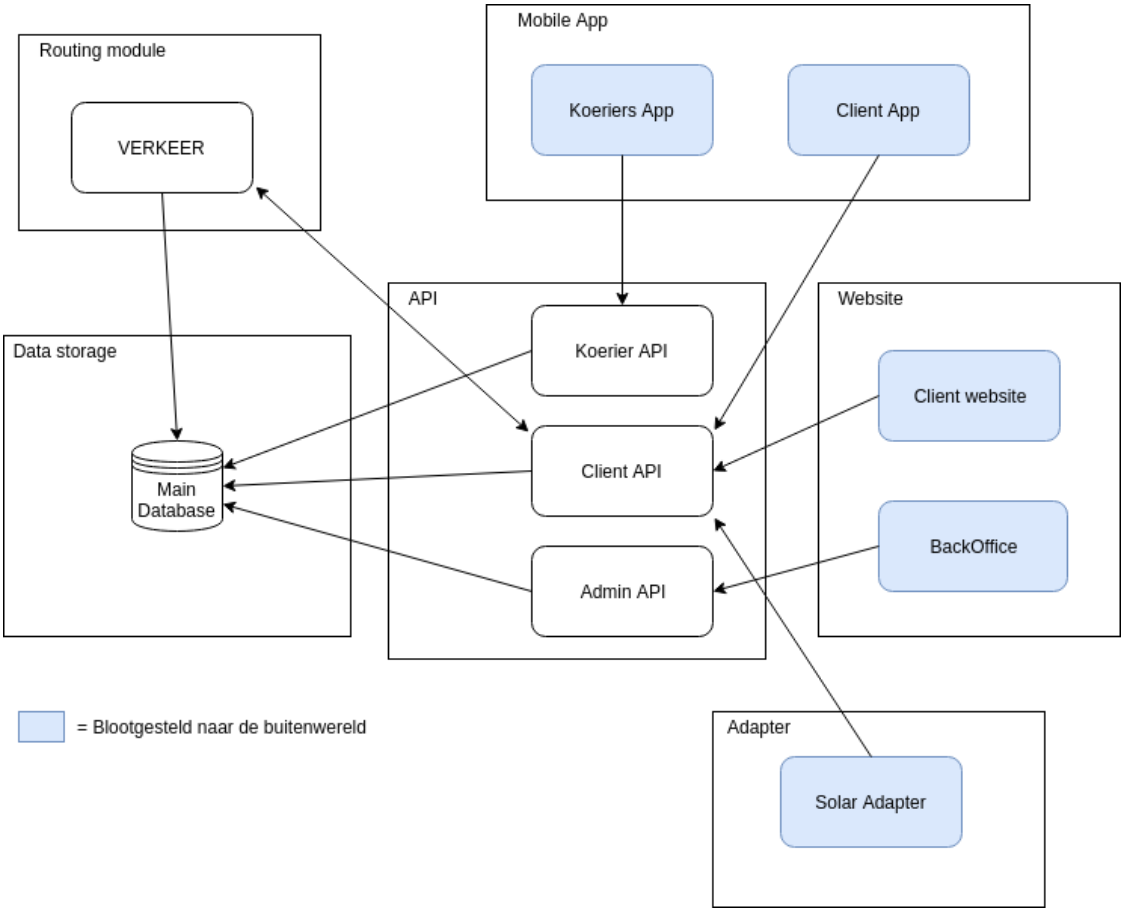
**Monteurs**
- Kunnen hun beschikbaarheid aangeven voor de komende 8 weken.
- Kunnen opdrachten accepteren of afwijzen.
- Geven locatie aan (ten alle tijden).
- Hebben skills en machtigingen.
- Moeten soms foto’s maken van het verrichte werk.
- Hebben een rating van stiptheid, kwaliteit en snelheid.
- Krijgen een push notificatie wanneer een opdracht is aangeboden.

**Klanten**
- Hebben toegang tot track en trace.
- Kunnen tevredenheid aangeven over verrichte werk.
- Moeten soms tekenen voor afronding opdracht.

### Overeenkomsten in functionaliteit

Bij elk van de systemen is een diagram gegeven, die een globaal overzicht geeft over de verschillende componenten van het systeem en de manier waarop deze met elkaar communiceren. In dit diagram zijn niet alle componenten weergegeven, zo is bijvoorbeeld niet de koppeling met de boekhoudpakketten weergegeven. Dit is om de diagrammen simpel en overzichtelijk te houden en tegelijkertijd alle informatie te tonen om een compleet genoeg beeld te krijgen van de systemen.
Ook worden de businessregels van elk systeem benoemd, samen met de systeemspecifieke implementaties. Vervolgens worden alle gelijkenissen tussen de drie systemen benoemd.

## Miles




*Figuur 20, Systeemdiagram Miles*

Businessregels

**Aanbieden van opdrachten**

Bij het aanbieden van de opdrachten in Miles:

- wordt een opdracht aan één iemand per keer aangeboden;
- wordt een opdracht maximaal twee keer aangeboden aan verschillende personen;
- moet er binnen vijf minuten worden gereageerd;
- verloopt volautomatisch;
- moet de koerier zijn locatie delen om in aanmerking te komen voor een opdracht;
- wordt er gebruik gemaakt van een aparte module ‘VERKEER’’, die de meest geschikte koerier kiest op basis van locatie, route en beschikbaarheid.

**Aangeven van beschikbaarheid**

Voor het aangeven van de beschikbaarheid in Miles:

- moet de koerier de app gebruiken;
- moet de koerier aangeven of hij op dit moment beschikbaar is.

**Aanmaken van opdrachten**

Een opdracht in Miles kan worden aangemaakt:

- door particulieren middels een app of de website;
- door bedrijven middels een app of de website;
- door beheerders middels de backoffice;
- door een specifiek bedrijf middels de op maat gemaakte adapter.

**Afronden van opdrachten**

Een opdracht in Miles wordt afgerond:

- wanneer de klant heeft getekend.

**Authenticatie en autorisatie**

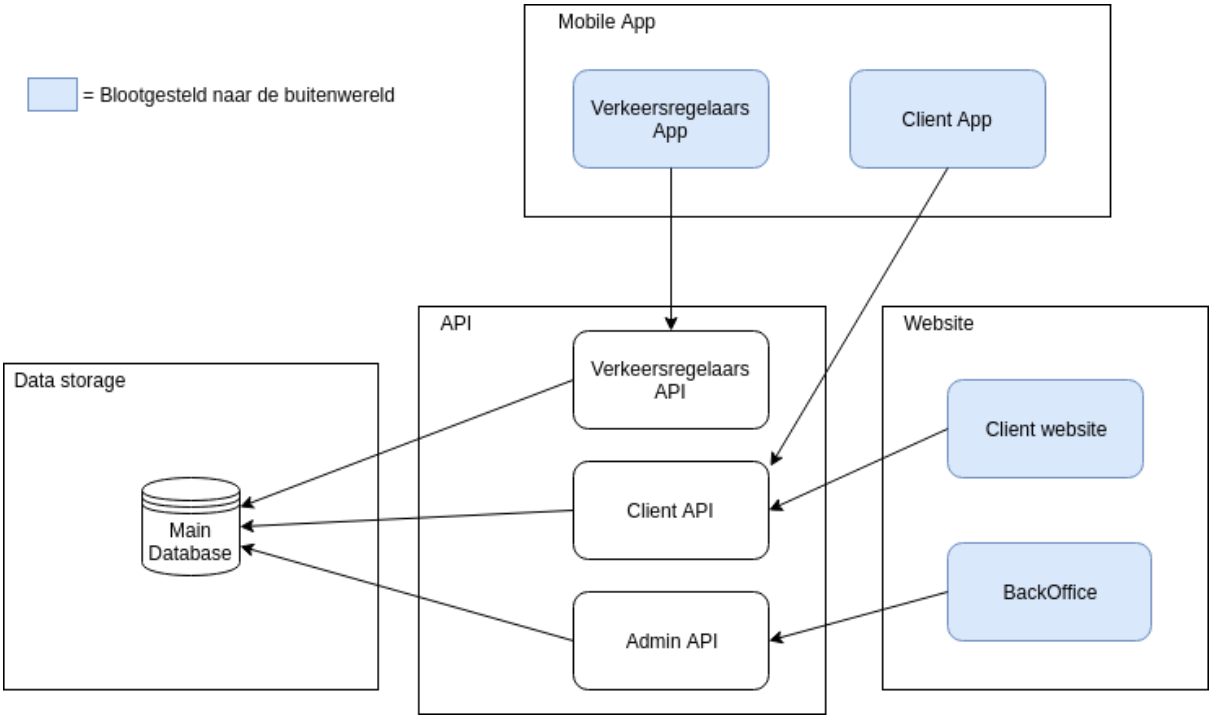
Beheerders in Miles:

- kunnen inloggen door middel van een e-mail- en wachtwoordcombinatie.

Koeriers in Miles:

- kunnen inloggen door middel van telefoon nummer;
- moeten verifiëren dat zij over het desbetreffende telefoonnummer beschikken door middel van een SMS-code;
- moeten worden goedgekeurd door een beheerder;
- kunnen speciale skills toegewezen krijgen door beheerders.

## Traffer




*Figuur 21, Systeemdiagram Traffer*

### Businessregels

**Aanbieden van opdrachten**

Het aanbieden van de opdrachten in Traffer:

- wordt gedaan door een subset van beheerders genaamd planners;
- wordt niet automatisch gedaan, het systeem biedt slechts handvatten hiervoor;

**Aangeven van beschikbaarheid**

Voor het aangeven van de beschikbaarheid in Traffer:

- moet de regelaar de app gebruiken;
- moet de regelaar aangeven wanneer hij in de week beschikbaar is;
- kan de regelaar op de minuut na nauwkeurig zijn.

**Aanmaken van opdrachten**

Een opdracht in Traffer kan worden aangemaakt:

- door bedrijven, nadat deze zijn goedgekeurd door een beheerder;
- door particulieren, nadat deze zijn goedgekeurd door een beheerder.

**Afronden van opdrachten**

Een opdracht in Traffer wordt afgerond:

- wanneer de regelaar aangeeft thuis aangekomen te zijn;
- als de regelaar zijn locatie aan laat staan gedurende de hele opdracht.

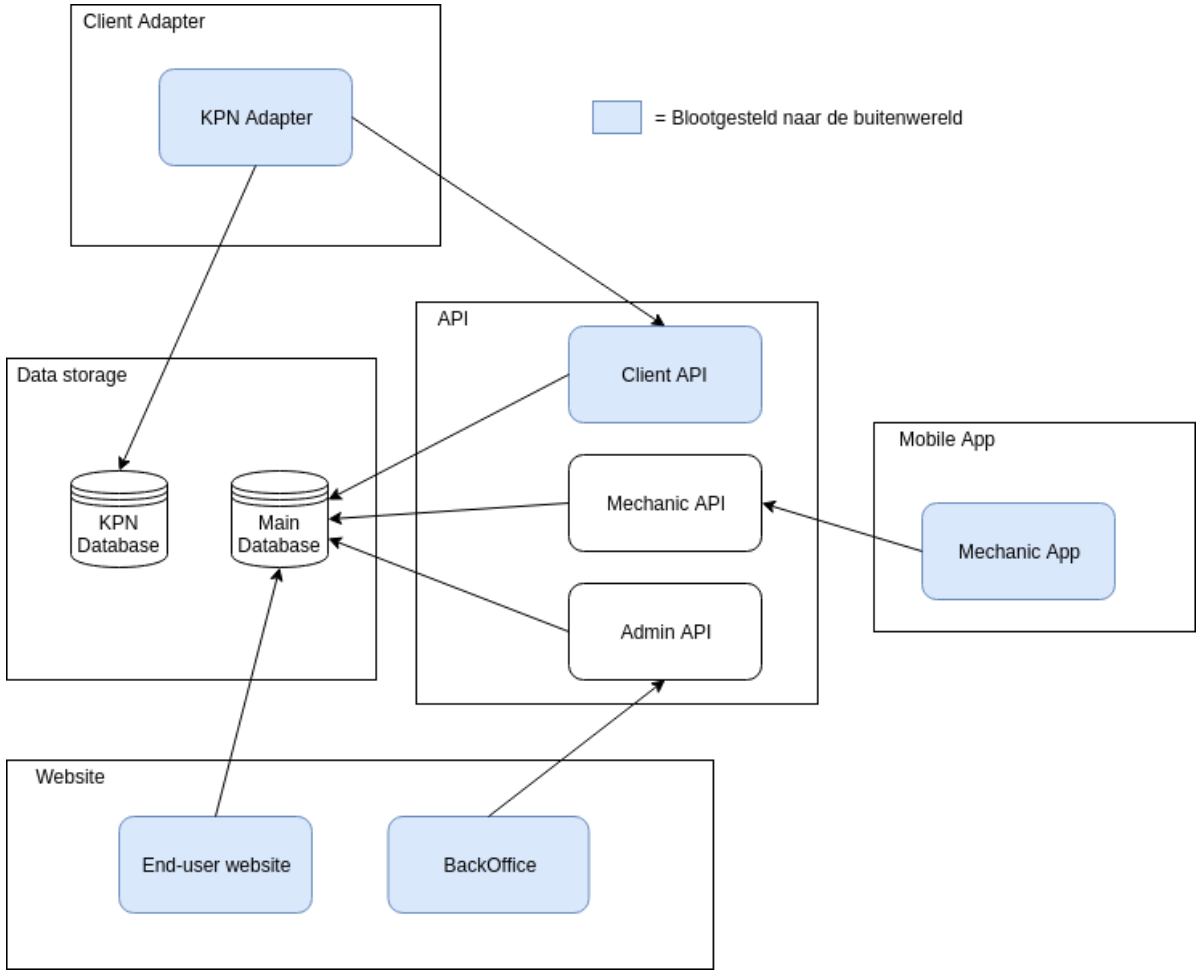
**Authenticatie en autorisatie**
Beheerders in Traffer:

- kunnen inloggen met een e-mail- en wachtwoordcombinatie;
- kunnen de wachtwoorden van regelaars veranderen wanneer die het wachtwoord zijn vergeten;
- hebben verschillende niveaus van rechten.

Regelaars in Traffer:

- kunnen inloggen met een e-mail- en wachtwoordcombinatie;
- hebben skills die toegewezen worden door beheerders.

## Utilio




*Figuur 22, Systeemdiagram Utilio*

Businessregels

**Aanbieden van opdrachten**

Bij het aanbieden van de opdrachten in Utilio:

- wordt een opdracht aan vijf monteurs per keer aangeboden;
- wordt een opdracht maximaal twee keer aangeboden aan verschillende monteurs;
- moet er binnen 10 minuten worden gereageerd;
- krijgt de eerste monteur die de opdracht accepteert de opdracht daadwerkelijk;
- verloopt volautomatisch.

**Aangeven van beschikbaarheid**

Voor het aangeven van de beschikbaarheid in Utilio:

- moet de monteur de app gebruiken;
- moet de monteur het per dagdeel aangeven;
- kan de monteur het invullen voor de komende 8 weken.

**Aanmaken van opdrachten**

Een opdracht in Utilio kan worden aangemaakt:

- door bedrijven middels de clinet API;
- door bedrijven middel de op maat gemaakte adapter.
- door beheerders in de backoffice.

**Afronden van opdrachten**

Een opdracht in Utilio wordt afgerond:

- wanneer de monteur dit aangeeft in de app;
- wanneer er getekend is als dat nodig is;
- wanneer er foto’s zijn gemaakt van het gedane werk, als dat nodig is.

**Authenticatie en autorisatie**

Beheerders in Utilio:

- kunnen inloggen door een e-mail- en wachtwoordcombinatie;

Monteurs in Utilio:

- kunnen inloggen door een e-mail- en wachtwoordcombinatie;
- moeten zich bij het inloggen verifiëren door een SMS-code;
- hebben skills die worden toegewezen door beheerders;
- hebben machtigingen die worden toegewezen door beheerders;
- kunnen hun wachtwoord resetten, middels een link die wordt gemaild.

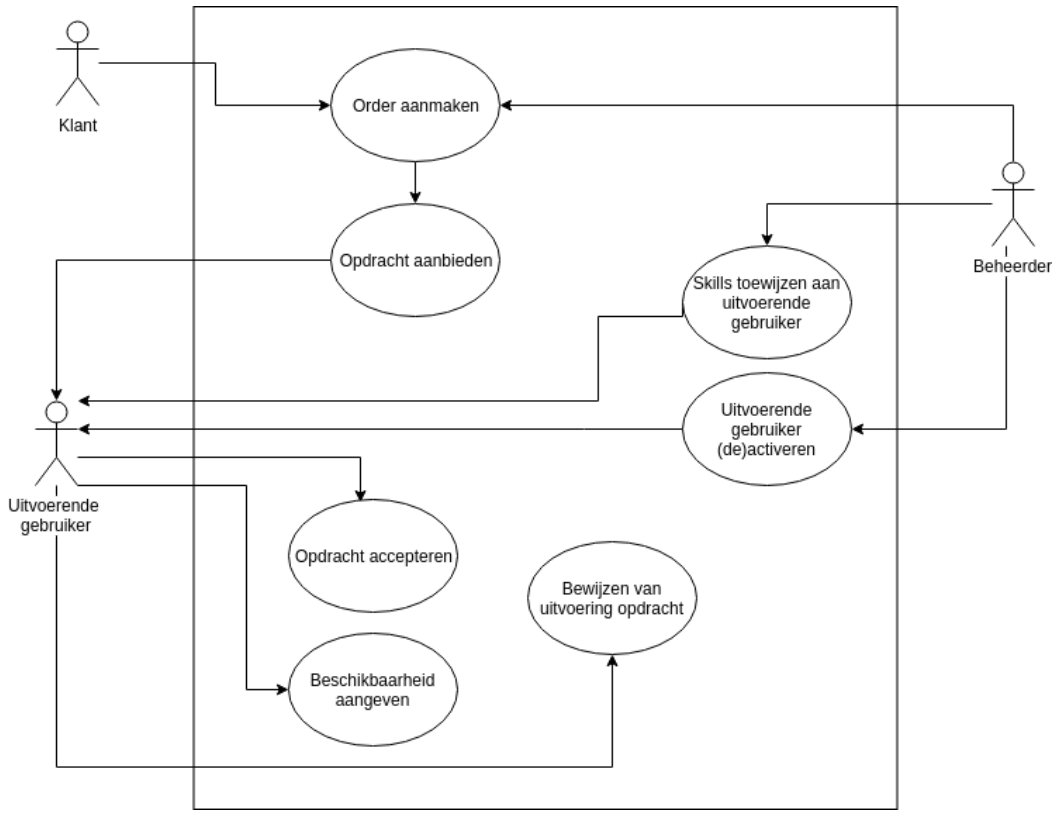
**Overeenkomsten**

Doordat de systemen andere gebruikers hebben en in een andere context gebruikt worden, kunnen de gelijkenissen abstract opgesteld zijn. Door deze abstractie zullen koeriers, regelaars en monteurs als een enkele gebruiker gezien worden: de uitvoerende gebruiker. Bij sommige gelijkenissen is een nuance gegeven.

De gelijkenissen zijn:

- De uitvoerende gebruiker kan zijn beschikbaarheid aangeven in de mobile app en alleen daar. Echter is de precisie waarmee dit gedaan wordt in elk van de systemen anders.
- Alle beheerders maken alleen gebruik van de backoffice.
- Alle uitvoerende gebruikers maken alleen gebruik van een app.
- Alle uitvoerende gebruikers moeten de opdracht afronden middels de mobile app, wat daarvoor nodig is kan verschillen per systeem en per opdracht.
- Een beheerder moet de uitvoerende gebruiker activeren. Ook kan de beheerder een uitvoerende gebruiker op non-actief zetten. Soms zijn er meer statussen dan alleen actief en non-actief mogelijk.
- Een uitvoerende gebruiker heeft skills nodig om een opdracht aangeboden te krijgen. Deze skills worden door een beheerder toegewezen. Soms zijn er ook machtigingen nodig voor opdrachten, die ook worden toegewezen door beheerders.
- In het aanbieden van opdrachten wordt locatie in acht genomen.

De overeenkomsten in functionaliteit van de verschillende bestaande smart resourcesystemen vanuit het perspectief van de gebruikers, kunnen worden samengevat in een use case diagram. Dit diagram is gegeven:




*Figuur 23, Generiek use case diagram*

### Niet overlappende maar wel gewenste functionaliteit

Naast de functionaliteit die in alle drie de smart resourcesystemen aanwezig is, zijn er ook functionaliteiten die gewenst zijn in toekomstige smart resourcesystemen. Bij elk van de functionaliteiten is aangegeven welk systeem dit reeds heeft geïmplementeerd, mits het van toepassing is. Deze functionaliteiten zijn:

- Een API waar klanten en andere systemen opdrachten kunnen inschieten. Mogelijk door middel van een aparte adapter. Deze functionaliteit is reeds aanwezig in Miles en Utilio.
- Volautomatisch aanbieden van opdrachten aan de uitvoerende gebruiker. Deze functionaliteit is reeds geïmplementeerd in Miles en Utilio.
- Integratie met boekhoudpakketten, voor automatische betalingen. Een dergelijke integratie is reeds geïmplementeerd in Miles en Utilio.
- Track en trace mogelijkheden voor de klant. Deze functionaliteit is reeds aanwezig in Utilio.
- Een hiërarchische opzet van uitvoerende gebruikers. Dit is reeds geïmplementeerd in Traffer.
- Een hiërarchische opzet van beheerders. Dit is reeds geïmplementeerd in Traffer.
- Klanten kunnen na afloop van het uitvoeren van de opdracht de tevredenheid invoeren. Deze functionaliteit is reeds geïmplementeerd in Utilio.
- Integratie met SMS-diensten, die gebruikt worden voor het verifiëren van uitvoerende gebruikers. Deze integratie is reeds geïmplementeerd in Miles en Utilio.
- Push berichten in de app van de uitvoerende gebruiker, wanneer er een opdracht is aangeboden. Deze functionaliteit is geïmplementeerd in Miles en Utilio.

## 11.2 Bijlage 2: Onderzoek naar architectuur resultaten

### Probleemanalyse

Deze systemen zullen op grote en kleine schaal moeten presteren, en tegelijkertijd moet er een systeem komen dat een degelijke basis biedt om een smart resourcesysteem op te zetten.
Recognize zet vraagtekens bij de schaalbaarheid van de huidige smart resourcesystemen; ze verwacht dat deze op grote schaal beter kunnen presteren.

### Onderzoeksvragen

Hoofdvraag: **Met behulp van welke technieken kunnen de smart resourcesystemen binnen Recognize worden opgezet zodat deze schaalbaar en uitwisselbaar zijn?**

Met schaalbaarheid wordt in deze context bedoeld of het systeem groter te maken is en of te zorgen is dat het systeem een grote workload aankan. Een systeem kan horizontaal schalen (dat wil zeggen dat het opschaalt door meer machines toe te voegen aan de clusteren) en verticaal schalen (dat wil zeggen dat het opschaalt door meer rekenkracht toe te voegen aan de machine).
Met uitwisselbaarheid wordt de mate waarin er gemakkelijk onderdelen van het systeem kunnen worden vervangen, toegevoegd of verwijderd bedoeld.

De hoofdvraag wordt beantwoord door te kijken naar de mogelijke architecturen en de technieken die deze kunnen implementeren. Daarom zijn de deelvragen:

1. Welke architecturen kunnen dienst doen voor de smart resourcesystemen?
2. Welke architectuur is het meest geschikt voor een smart resourcesysteem?
3. Welke technieken kunnen gebruikt worden voor de smart resourcesystemen?
4. Welke technieken zijn het meest geschikt voor een smart resourcesysteem?

# Resultaten

## Welke architecturen kunnen de dienst doen voor de smart resourcesystemen?

Doordat smart resourcesystemen alleen de basiswerking met elkaar gemeen hebben, zal er bij het opzetten van een nieuw systeem dus telkens een deel bij aangebouwd moeten worden. De architectuur moet hier mee om kunnen gaan, en het liefst ook ondersteunen. Hiervoor zijn er een tweetal opties, die tevens de enige opties zijn die ooit ter sprake zijn gekomen:

- Framework. De smart resourcesystemen zouden gebouwd kunnen worden met behulp van een framework. Dit framework zal over basisfunctionaliteit van een smart resourcesysteem beschikken, en wordt in elk project meegebakken. Het framework is dan als het ware de eerste opzet van elk nieuw smart resourcesysteem.
- Microservice architecture. De smart resourcesystemen zullen komen te bestaan uit verschillende microservices die elk een deel van de algemene werking op zich zal nemen.

## Welke architectuur is het meest geschikt voor een smart resourcesysteem?

Binnen de context van smart resourcesystemen zijn er een tweetal niet-functionele requirements die meegenomen worden tijdens het project, en die betrekking hebben op de techniek. Deze zijn dat de smart resourcesystemen schaalbaar moeten zijn en uitwisselbaar opgezet moet zijn. De voor- en nadelen hebben, naast dat ze betrekking hebben op de niet functionele requirements, ook betrekking op de algemene snelheid, de bruikbaarheid, het realisatieprocess en het onderhoud. Bij elk van de voor- of nadelen wordt hetgeen waar het betrekking op heeft expliciet genoemd achter de uitleg.

**Framework**

**Voordelen**

1. Minimale configuratie voordat een smart resourcesysteem de up en running is (realisatieproces).

**Nadelen**

1. De omvang van het framework neemt toe met elke nieuwe implementatie van een functionaliteit. Daarmee neemt ook de codebase van elk smart resourcesysteem toe, ook al wordt een deel niet gebruikt voor het systeem (onderhoudbaarheid).
2. Weinig uitwisselbaarheid. Doordat een framework een geheel is, is het niet zomaar mogelijk een deel te vervangen met iets anders. Het is wel mogelijk meerdere implementaties van een functionaliteit in het framework te bakken (uitwisselbaarheid).

**Microservice architecture**

**Voordelen**

1. De omvang van het project bedraagt alleen alle benodigde functionaliteiten (onderhoudbaarheid).
2. Dit schaalt horizontaal, dat wil zeggen dat het opschaalt door meer machines toe te voegen aan de cluster (schaalbaarheid).
3. Grote uitwisselbaarheid. Een microservice kan worden vervangen door een andere, zolang de communicatie onderling niet verandert (uitwisselbaarheid).

**Nadelen**

1. Overhead door communicatie tussen de verschillende services. Dit is echter minimaal wanneer het HTTP 2 bedraagt (snelheid).

De voordelen van een framework setup wegen niet op tegen de voordelen van een microservice architecture. Ook de nadelen van de microservice architecture zijn significant kleiner en minder relevant dan die van een framework.
De belangrijkste uitgangspunten schaalbaarheid en uitwisselbaarheid zijn beide beter vertegenwoordigd in een microservice achitecture, doordat dit horizontaal schaalt en een microservice alleen hoeft te voldoen aan een de communicatie interface met andere microservices. Daarbij zijn er andere voordelen verbonden aan de keuze voor een microservice architecture, zoals dat er niet meerdere implementaties van een functionaliteit in de codebase komen. Er zullen dan meerdere microservices zijn, waaruit men kan kiezen.

## Welke technieken kunnen gebruikt worden voor de smart resourcesystemen?

De technieken die gebruikt gaan worden voor smart resourcesystemen zullen bekend moeten zijn bij Recognize. Dit is om te zorgen dat de systemen uitbreidbaar zijn en onderhoudbaar zijn door Recognize. De technieken die gebruikt gaan worden zijn onder te verdelen in twee groepen: programmeertaal & framework en compartment managing.

**Programmeertaal en framework**

De keuze van de programmeertalen vloeit voort uit de talen die op dit moment gebruikt worden binnen Recognize, en die dus bekend zijn bij haar programmeurs. Naast PHP, de taal waar alle huidige smart resourcesystemen in gerealiseerd zijn, worden ook Kotlin, C#, Java en Javascript gebruikt.

PHP is de meest gebruikte taal voor backend systemen binnen recognize. Het wordt gebruikt voor onder andere de bestaande smart resourcesystemen en zal daarom nader worden bekeken in dit onderzoek. Het framework dat samen met PHP nader bekeken gaat worden is Symfony. Dit framework is zeer bekend bij de programmeurs van Recognize en de huidige smart resourcesystemen zijn met dit framework gerealiseerd.

Kotlin is een taal op de JVM, maar met minder boilerplate en meer suger syntax dan Java en wordt binnen Recognize beschouwd als strictly beter dan Java. Kotlin wordt nog niet zoveel gebruikt binnen Recognize als PHP, maar heeft bij nieuwe projecten de voorkeur over PHP vanwege de snelheid van de applicaties die erin gemaakt zijn. Kotlin zal nader worden onderzocht, in combinatie met Spring.

Javascript kan serverside gebruikt worden door middel van NodeJS. Echter is het gebruik van NPM een probleem, veel libraries zijn constant in ontwikkeling. Daardoor moeten deze continu geüpdatet worden of worden ze deprecated. Hierdoor biedt dit geen goede basis voor deze smart resourcesystemen die lang mee zouden moeten gaan.

C# valt niet onder de opties die nader worden bekeken, omdat de teams die binnen Recognize uiteindelijk de smart resourcesystemen gaan bouwen en de bestaande beheren,

niet werken met C#. Daarom is er gekozen om het bij een taal te laten die al bekend is bij de teamleden.

De programmeertalen en frameworks die nader onderzocht worden in dit onderzoek zijn dus:

- Kotlin gebruikmakend van Spring boot;
- PHP gebruikmakend van Symfony.

**Compartment managing**

Voor compartment managing worden Docker en Kubernetes momenteel gebruikt binnen Recognize. Compartment managing is een onderdeel van het beheren van systemen wanneer deze uitgerold zijn. Docker wordt gebruikt om container applicaties te draaien in een Kubernetes cluster. De verschillende projecten/systemen en de omgevingen (test/acc/pods) zijn gescheiden door middel van namespaces. Echter is dit niet een vereiste, beide technieken kunnen afzonderlijk gebruikt worden. Docker kan multi-container-applicaties draaien door middel van docker compose en docker swarm. Docker compose is bedoeld voor lokale development en niet voor in een cluster; docker swarm is juist bedoeld voor in een cluster. Beide verbinden docker containers aan elkaar en aan de de buitenwereld.
Zowel Kubernetes als docker kunnen dienst doen voor smart resourcesystemen.

Welke technieken zijn het meest geschikt voor een smart resourcesysteem?
Bij alle opties worden een aantal voor- en nadelen gegeven. Dit kan betrekking hebben op één van de twee of beide architecturen. Of dit kan betrekking hebben op de niet-functionele requirements, de algemene snelheid, de bruikbaarheid, het realisatieproces en het onderhoud. Net als in paragraaf 5.3 wordt achter elk voor- of nadeel expliciet hetgeen genoemd waar het betrekking op heeft.

## Programmeertaal en framework

### PHP (Symfony)

**Nadelen**

1. Maakt voor elke request een nieuw process aan, wat een grotere overhead heeft dan Kotlin (snelheid).
2. I/O is hoger dan bij talen op JVM. Daardoor is het op grote schaal merkbaar trager. Echter kunnen er extensies worden geschreven in C of C++ voor snellere performance, maar wat weer voor meer onderhoud zorgt (snelheid).

### Kotlin (Spring)

**Voordelen**

1. Draait in JVM in memory, daardoor sneller dan PHP (snelheid).
2. Spring heeft meerdere modules waaronder Spring Cloud met Netflix OSS speciaal voor microservices. Dit is alleen een voordeel voor microservices (realisatie process).

**Nadelen**

1. Veel syntactic sugar en daardoor een hoog instapniveau qua begrijpen van code met samenwerken (onderhoudbaarheid).

Kotlin heeft aanzienlijk meer voordelen dan PHP in de context van de smart resourcesystemen en hun behoeftes. Kotlin heeft een voordeel op de schaalbaarheid, doordat het aantal I/O kleiner dan bij PHP.
De frameworks die gebruikt worden voor het opzetten van een webserver zijn zo goed als gelijkwaardig in functionaliteit.

## compartment managing

### Docker

Wordt momenteel al gebruikt binnen Recognize. Er wordt Docker compose gebruikt om websites en webapplicaties te draaien in een container. Echter wordt dit niet gebruikt voor microservices.

**Voordelen**

1. Altijd hetzelfde basisbesturingssysteem tijdens development en productie, tussen verschillende host machines (realisatieprocess).

**Nadelen**

1. Docker swarm is erg complex in zowel opzet als gebruik. Dit blijkt uit ervaringen van deskundigen.
2. Beduidend minder monitoringmogelijkheden met swarm dan met Kubernetes (realisatieprocess).

**Kubernetes**
Wordt ook al gebruikt door Recognize, om de hosting van de verschillende applicaties te beheren en te regelen.
**Voordelen**

1. Is erg flexibel in inrichting van verschillende resources (schaalbaarheid).
2. Self healing nodes en health checks. Dit betekent dat wanneer er een node in de cluster omvalt, Kubernetes zelf zorgt dat deze wordt vervangen (onderhoudbaarheid).

**Nadelen**

1. Initiële opzet van kubernetes cluster is veel werk. Dit blijkt uit ervaringen van deskundigen (realisatieprocess).

Voor het managen van de compartments kan er voor een combinatie van docker en kubernetes worden gekozen. Op deze manier zijn alle voordelen van beide technieken van toepassing.

# 11.3 Bijlage 3: Onderzoek naar dataopslag resultaten

## Probleemanalyse en onderzoeksvraag

Bij het bepalen van de afstudeeropdracht is er aangegeven dat er zorgen zijn over de snelheid van de huidige systemen, wanneer deze op grote schaal worden uitgerold. De zorgen betrekken zich voornamelijk op de data-opslag en alles daaromheen. Recognize vraagt zich af of een SQL-database wel schaalbaar genoeg is voor een smart resourcesysteem.

Daarom luidt de hoofdvraag: **Welke manier of manieren van dataopslag sluit(en) het beste aan op de wensen van smart resourcesystemen?**

## Onderzoeksmethoden

### Beoordelen

Om de verschillende technieken en manieren van data-opslag met elkaar te kunnen vergelijken, zijn er een aantal criteria opgesteld. Elk van de mogelijke opties wordt getoetst aan deze criteria. Ieder criterium heeft een weging, evenredig aan hoe belangrijk dat criterium wordt geacht aan het systeem. Vervolgens krijgt iedere optie voor elk criterium een score tussen 1 en 10. De eindscore van een data-opslag is de som van alle scores maal de weging van de bijbehorende criterium: eindscore = $\Sigma$ (score x weging) voor elk van de criteria.

**Criteria**

Aan de hand van de wensen die een smart resourcesysteem heeft aan de data-opslag, zijn een drietal criteria opgesteld.
De criteria zijn:

- mate van omgang van locatiedata;
- schaalbaarheid van de dataopslag;
- omgang met transacties en locking;
- complexiteit in gebruik van de data-opslag.

### Wegingen

**Omgang met locatie-data**

Weging: 5
Een van de basisonderdelen van de engine is dat het matcht op basis van locatie. Dat de data-opslag daar goed mee kan omgaan is dus een must. Hoe goed, gemakkelijk en snel dit gaat, is van belang voor de werking van de engine. Dit staat los van de normale snelheid (read en write). Een goede omgang met locatie-data versimpelt het systeem, maar is geen requirement. Echter is het wel van belang dat het systeem überhaupt kan omgaan met locatie-data. Daarom is er gekozen voor een weging van 5.

**Schaalbaarheid van de data-opslag**

Dit houdt in dat de performance van de data-opslag niet lijdt onder de hoeveelheid data of de mate waarin het gebruikt wordt.

Weging: 9
De engine dient ook op grote schaal goed te presteren. Dit is een van de requirements. Dit is het belangrijkste in de context van een smart resourcesysteem. Idealiter schaalt de data-opslag horizontaal en presteert het systeem even goed, ongeacht het aantal gebruikers of hoeveelheid data. Daardoor weegt de score van dit criterium zwaar mee en is een weging van 9 gekozen.

**Omgang met transacties en locking**
Dit houdt in dat de database moet kunnen garanderen dat een query daadwerkelijk is uitgevoerd en dat er mogelijkheden zijn omtrent concurrency.
Weging: 8
Doordat de smart resourcesystemen veel gebruikers die tegelijkertijd acties op het systeem uitvoeren aan moet kunnen, moet het smart resourcesysteem kunnen garanderen dat een actie ook daadwerkelijk wordt uitgevoerd, of moet aan de gebruiker kenbaar worden gemaakt dat een actie niet is uitgevoerd. Ook de problemen die gepaard gaan met concurrent schrijven naar de database moeten worden afgevangen. Daardoor weegt dit criterium zwaar mee met een weging van 8. Het is erg belangrijk, maar niet zo belangrijk als de schaalbaarheid. De problemen zouden ook in de programmatuur afgevangen kunnen worden. Dit is echter niet wenselijk.

**Complexiteit in gebruik van de dataopslag**
Hierbij geldt: een hogere score betekent dat het minder complex is.
Weging: 8
De software-oplossing - die gebruik gaat maken van de data-opslag - moet uitbreidbaar zijn, omdat dit enkel een basis biedt voor smart resourcesystemen. Daardoor speelt de complexiteit van de implementatie van de engine ook mee in het kiezen van een geschikte data-opslag. Recognize zal de engine moeten kunnen gebruiken en ook kunnen uitbreiden. Doordat de complexiteit indirect invloed heeft op de realisatie van toekomstige smart resourcesystemen, is er gekozen voor een weging van 8.

### Dataopslagmethodes
De verschillende manieren van data-opslagmethodes zijn onder te verdelen in drie hoofdgroepen: SQL, denormalized SQL en NoSQL. File storage is ook een data-opslagmanier, echter is dit een geval apart en is dit geen realistische optie, omdat er geen files worden opgeslagen in een smart resourcesysteem.
UIt elk van de drie categorieën is een techniek gekozen die zal worden onderzocht:

- **PostgreSQL**. Een SQL database. DIt wordt momenteel gebruikt voor de bestaande smart resourcesystemen. Ook is dit de database die het meest gebruikt wordt binnen Recognize.
- **Elasticsearch**. Een noSQL data-opslag, vooral goed in full text search. Dit wordt nader onderzocht naar aanleiding van het verzoek vanuit de opdrachtomschrijving van Recognize.
- **PostgreSQL Citus Data**. Een denormalized SQL data-opslag. Dit is een extensie op postgres, speciaal voor schaalbaarheid. Daardoor lijkt dit een goede kandidaat voor smart resourcesystemen.

## Resultaten

### PostgreSQL (normalized) and PostGIS

| ID | Criterium | Weging | Score | Eindscore |
|---|---|---|---|---|
| **1** | location data | 5 | 8 | 40 |
| **2** | schaalbaarheid | 9 | 3 | 27 |
| **3** | transacties en locking | 8 | 9 | 72 |
| **4** | complexiteit | 8 | 9 | 72 |

*Tabel 4, Scores PostgreSQL*

**Totaal: 211**

| ID | Uitleg score |
|---|---|
| **1** | PostGIS is extensie speciaal voor geografische data. Het is geoptimaliseerd voor veel use cases, waaronder de afstand bepalen tussen twee punten en werken met hoge precisie. |
| **2** | PostgreSQL schaalt van zichzelf verticaal en is daardoor dus niet goed schaalbaar. Echter kan door het toepassen van materialized views (het redundant opslaan van data dat bij elkaar hoort) op tabellen de read snelheid worden verbeterd, maar wordt de write snelheid minder. |
| **3** | Zowel transacties als locking worden gesupport door PostgreSQL. Bij transacties is het het geval dat er wordt gegarandeerd dat de transactie wordt weggeschreven naar de write ahead log, wat erop neerkomt dat de transactie zelf ook wordt uitgevoerd. |
| **4** | PostgreSQL is bekend bij Recognize en is een SQL-taal. Doordat het op Engels lijkt, is het geen complexe taal om te gebruiken of te leren. |

*Tabel 5, uitleg scores PostgreSQL*

Elasticsearch

| ID | Criterium | Weging | Score | Eindscore |
|---|---|---|---|---|
| **1** | location data | 5 | 6 | 30 |
| **2** | schaalbaarheid | 9 | 8 | 72 |
| **3** | transacties en locking | 8 | 4 | 32 |
| **4** | complexiteit | 8 | 6 | 48 |

*Tabel 6, Scores Elasticsearch*

**Totaal: 182**

| ID | Uitleg score |
|---|---|
| **1** | Er zit functionaliteit voor geodata native in Elasticsearch, echter is dit niet zo snel en geoptimaliseerd als PostGIS, blijkt uit benchmarks (Mulders, 2018). |
| **2** | Elasticseach schaalt verticaal en is gemakkelijk te schalen. Echter komt dit tegen de prijs van veel werkgeheugengebruik. Naarmate er meer gebruikers of records zijn, groeit het geheugengebruik. Een bekend en erkend probleem is outofmemory error, waar elasticsearch niet goed mee om kan gaan (Elastic, n.d.). |
| **3** | Transacties worden niet ondersteund in Elasticsearch. Hetgeen dat het dichtstbij komt, is data in bulk of te inserten of te updaten, en vervolgens de response van de query te checken tegen de data die opgeslagen zou moeten worden.<br>Locking wordt wel ondersteund, in meerdere vormen. |
| **4** | Naast SQL kent Elasticsearch ook zijn eigen query taal (Elastic, n.d.-a). Echter, doordat Elasticsearch niet kan garanderen dat data daadwerkelijk opgeslagen is zonder het zelf te checken, zowel door een out-of-memory error als door het ontbreken van transacties, zal er extra complexiteit moeten worden aangebracht om dit te garanderen. |

*Tabel 7, Uitleg scores Elasticsearch*

PostgreSQL (denormalized citusdata) and PostGIS

| ID | Criterium | Weging | Score | Eindscore |
|---|---|---|---|---|
| **1** | location data | 5 | 8 | 40 |
| **2** | schaalbaarheid | 9 | 7 | 63 |
| **3** | transacties en locking | 8 | 9 | 72 |
| **4** | complexiteit | 8 | 4 | 32 |

*Tabel 8, Scores denormalized PostgreSQL*

**Totaal: 207**

| ID | Uitleg score |
|---|---|
| **1** | PostGIS is extensie speciaal voor geografische data. Het is geoptimaliseerd voor veel use cases, waaronder de afstand bepalen tussen twee punten en bepalen of een punt in een gebied ligt. Dit is net als de normale PostgreSQL van toepassing. |
| **2** | De extensie is gemaakt om schaalbaarheid te waarborgen. Het past sharding en replication toe, waardoor het verticaal schaalbaar is. Niet alle functionaliteit is direct beschikbaar, maar zijn er wel workarounds voor deze gevallen. |
| **3** | Transacties worden sinds Citus Data 7.1 ondersteund. Echter is dit iets complexer dan transacties in PostgreSQL, maar het heeft hetzelfde resultaat. Voor locking wordt de native locking van PostgreSQL gebruikt en wordt het dus ook ondersteund. |
| **4** | Hoewel citus data gebruik maakt van van PostgreSQL, is het beduidend complexer dan PostgreSQL. Doordat er gebruik wordt gemaakt van sharding en replicating, moet er vooraf goed nagedacht worden over de indeling van de datastructuur en de tabellen. Er moet goed nagegaan worden welke data veel reads krijgen en welke veel writes. Ook niet alle SQL features zijn beschikbaar in citus data, wat voor extra complexiteit kan zorgen in de applicatie die gebruik maakt van de data-opslag (Citus, n.d.). |

*Tabel 9, Uitleg scores denormalized PostgreSQL*

## 11.4 Bijlage 4: Reflectie

### Verloop

Aan het begin van de afstudeerperiode lijken vijf maanden een zee van tijd; ik dacht dat ik de opdracht wel even zou doen en dit werd ook nog eens versterkt nadat er zowaar bijna geen feedback op mijn plan van aanpak kwam. Niets minder dan het tegendeel bleek waar te zijn.
Ergens halverwege het project ben ik mezelf kwijt geraakt in de details. Ik was voornamelijk bezig met het schrijven van software, en was het overzicht vrijwel volledig kwijt. Wanneer Juul - mijn bedrijfsbegeleider - mij dan naar de voortgang vroeg in de wekelijkse gesprekken had ik wel een antwoord, maar uit zijn reactie bleek wel dat hij zich zorgen maakte over de voortgang van het project. Achteraf kan ik ook met zekerheid zeggen dat ik achter liep op schema.
Tijdens het tweede gesprek met Rogier - mijn schoolbegeleider - werd ik flink met de neus op de feiten gedrukt. Naar aanleiding daarvan heb ik het roer omgegooid en heb ik een stap terug gedaan om het overzicht terug te krijgen. In plaats van extra tickets in de backlog te zetten, werden er tickets afgerond. En in plaats van schrijven wat ik nog moet verslagleggen, werd er daadwerkelijk verslag gelegd.
Na een flinke inhaalslag heb ik, al zich ik het zelf, toch een mooi stuk software weten te maken en het redelijk verslag weten te leggen.

### Projectaanpak

Vanuit de opleiding heb ik altijd meegekregen dat een agile werkmethode, specifiek de een SCRUM-methodiek, het beste aansluit bij software-ontwikkeling. Hoewel ik denk dat dit geldt voor het merendeel van de softwareprojecten, denk ik dat een watervalaanpak, die uiteindelijk is toegepast voor dit afstudeerproject, op zijn plaats was. Hoewel de elementen uit agile die ik heb meegenomen in de projectaanpak mij goed hebben gediend, ben ik erachter gekomen dat een watervalaanpak voor een project dat wordt uitgevoerd door een enkel persoon prima kan werken.
Daarnaast ben ik tevreden met de keuze om geen SCRUM te doen, omdat dit voornamelijk is gericht op software-ontwikkeling in teams.

## Ontwerp- en implementatiekeuzes

In de loop van de afstudeerperiode heb ik behoorlijk wat implementatiekeuzes gemaakt. De grootste en meest impactvolle keuzes heb ik aan de hand van een onderzoek gemaakt.
Zo heb ik bijvoorbeeld de architectuur af laten hangen van een onderzoek. Ik had echter blijkbaar moeite met het implementeren daarvan. In eerste instantie had ik een distributed monolith gemaakt, wat niet de bedoeling is en wat mij veel te veel tijd heeft gekost.
Wat ook veel tijd heeft gekost is de implementatie van alles in Kotlin en Spring. Hoewel ik nog steeds achter deze keuze sta, zorgde dit wel voor impediments. Spring is vrij nieuw bij Recognize en daarom is ook nog niet alle kennis daarover aanwezig. Daardoor was het soms het geval dat ik degene was die dingen voor het eerst uit moest zoeken. Erg leerzaam, maar ook erg lastig.

Al met al ben ik van mening dat er een bruikbare engine is neergezet, waar goed over na is gedacht en die hopelijk vaak een basis zal bieden voor smart resourceprojecten.

# 11.5 Bijlage 5: Systeemdossier

## 1. Algemene informatie over het systeem

### 1.1 Functionaliteiten

De engine is een abstracte basis voor smart resourcesystemen. Het implementeert basisfunctionaliteiten voor de drie standaard type gebruikers van smart resourcesystemen: beheerders, klanten en uitvoerende gebruikers. Deze functionaliteiten, geordend op gebruiker, zijn:

**Beheerders:**

- Skills aanmaken.
- Het toewijzen van skills aan een uitvoerende gebruiker.
- Het verwijderen van een skill van een uitvoerende gebruiker.
- Het aanmaken van een beheerder.

**Klanten:**

- Een opdracht aanmaken.
- Een opdracht annuleren.
- Een opdracht aangeven waarbij uitvoerende gebruikers gezocht moeten worden voor het systeem.
- De voortgang van een opdracht opvragen.

**Uitvoerende gebruikers:**

- Een opdracht accepteren of afwijzen.
- De voortgang van een geaccepteerde opdracht wijzigen.
- De beschikbaarheid van zichzelf beheren.
- Zijn locatie doorgeven.

**Functionele requirements:**

- Uitvoerende gebruikers worden op basis van locatie, beschikbaarheid en skills gezocht bij een opdracht.
- De engine heeft een API om orders aan te maken.
- De aanbod biedende partij kan haar beschikbaarheid aangeven door middel van een API.
- De engine houdt de voortgang bij van de afhandeling van de vraag.
- De engine past authenticatie toe op alle API’s.
- De engine is modulair opgezet, zodat elke module vervangbaar is.
- De engine maakt de keuzes en errors van het systeem inzichtelijk.
- De engine past logging toe op essentiële onderdelen.

## 1.2 Werking van het systeem

Het systeem brengt vraag en aanbod bij elkaar. De vraag is hierbij een opdracht die wordt gemaakt door een klant, wat deze opdracht inhoudt maakt voor het systeem niet uit. Een opdracht kan een aantal benodigde skills hebben. Om als uitvoerende gebruiker de opdracht aangeboden te krijgen, moet hij over deze skills beschikken. Naast de benodigde skills, moet de uitvoerende gebruiker beschikbaar zijn op het moment van de uitvoering van de opdracht. Het moment van uitvoering wordt aangegeven door de klant bij het maken van de opdracht. Als laatste moet de uitvoerende gebruiker in de buurt zijn van de locatie waar de opdracht wordt uitgevoerd. Hoe ver ‘in de buurt’ is, is configurabel.

Wanneer een klant aangeeft dat hij de opdracht wil aanbieden aan geschikte uitvoerende gebruikers, kijkt de engine naar alle mogelijke opties en op basis van de hierboven gestelde eisen biedt de engine de opdracht aan, aan een aantal gebruikers. Het maximale aantal gebruikers dat dezelfde opdracht krijgt aangeboden is configurabel.

De uitvoerende gebruikers hebben dan de keuze om de opdracht te weigeren of te accepteren. Bij het accepteren geldt: wie het eerst komt, wie het eerst maalt.

Tijdens het uitvoeren van de opdracht, kan de uitvoerende gebruiker de voortgang aangeven. Bijvoorbeeld dat hij bezig is, of dat hij klaar is. Deze voortgang is op te vragen door de klant.

De beheerders kunnen skills maken, toewijzen aan en verwijderen van een uitvoerende gebruiker.

# 2. Hoe gebruik je het systeem

## 2.1 Development opzet

Benodigdheden om de engine lokaal te draaien:
Git. https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
Docker: https://docs.docker.com/install

De engine is opgesplitst in twee microservices, die ieder een eigen repository hebben.
Om de projecten de downloaden van bitbucket, open de volgende commando's uit in een terminal:
***git clone git@bitbucket.org:recognize/match.git; git clone git@bitbucket.org:recognize/auth.git***

Om vervolgens de engine te starten, kan docker worden gebruikt. Dit kan gedaan worden door de volgende commando's uit te voeren in de terminal:
***cd /path/to/microservices/***
***cd auth/***
***docker-compose up***

Wanneer de commando's klaar zijn, is zijn de microservices compiled en de databases en hun tabellen aangemaakt.

Let op: Bij het lokaal hosten van de engine, draaien de microservice op een eigen poort. Auth draait op poort 82 en match op poort 81.

## 2.2 Testen runnen

Bij de engine komt met een aantal integratietesten. Deze testen zijn volautomatisch en kunnen zowel lokaal als in bitbucket worden gerund.

### 2.2.1 lokaal

Benodigdheden om de automatische testen lokaal te kunnen doorlopen:
Docker: https://docs.docker.com/install
Java: https://www.java.com/en/download/
Match en Auth gedownload hebben. (zie 2.1 development opzet)

Om de testen lokaal te kunnen draaien moet de test database van de match microservice draaien. Dit kan het gemakkelijkste gedaan worden door de engine te starten
(zie 2.1 development opzet).
Om alleen de test database te starten, zonder de overige delen van de engine, voer de volgende commando's uit:
***cd /path/to/microservies/***
***cd auth/***
***docker-compose start postgres-match_tests***

En om de test database weer te stoppen:
***docker-compose stop postgres-match_tests***

De automatische testen kunnen gestart worden door de volgende commando's in een terminal uit te voeren:
***cd /path/to/microservices/***
***cd match/***
***set -o allexport; source .env; set +o allexport***
***./gradlew test***

De resultaten van de testen kunnen worden bezichtigd door in een browser, naar eigen voorkeur, de url te bezoeken:
file://</path/to/microservices>/match/build/reports/tests/test/index.html

### 2.2.2 Bitbucket
De testen worden automatisch gerund wanneer er een nieuwe commit naar bitbucket wordt gepushed. Dit is gedaan door middel van de continuous integration oplossing van bitbucket, de bitbucket pipelines. Out of the box wordt er bij elke code commit de testen doorlopen. Dit is te configureren door middel van de bitbucket-pipelines.yml.

In de bitbucket-pipelines.yml, te vinden in de root directory van de microservice, staat aangegeven welke acties er uitgevoerd moeten worden bij een codecommit. Wanneer er een tag op een commit wordt gedaan, wordt er een build gemaakt van de microservice. Wanneer er een release branch wordt gemaakt, wordt deze branch gedeployed naar de cluster. En bij elke code commit worden de automatische testen doorlopen.

Alle mogelijkheden die de bitbucket-pipelines bieden en hoe deze te configureren zijn, is te vinden in de documentatie van bitbucket hierover:
https://confluence.atlassian.com/bitbucket/build-test-and-deploy-with-pipelines-792496469.html

## 2.3 Deployen naar server
Het deployen naar de cluster van recognize wordt ook via de bitbucket pipelines gedaan. Dit is simpelweg te doen door een release/<version-number> branch naar de repository te pushen. De rest wordt vervolgens geregeld. Waaronder de certificaten voor https.

## 2.4 Logging

Out of the box logt de engine naar de console. Dit kan worden verandert naar dat de engine naar een bestand logt, door custom logback-spring.xml bestand te maken.
De inhoud van dit bestand wordt dan:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
	<include resource="org/springframework/boot/logging/logback/defaults.xml" />
	<property name="LOG_FILE" value="</path/to/log/file>"/>
	<include resource="org/springframework/boot/logging/logback/file-appender.xml" />
	<root level="INFO">
		<appender-ref ref="FILE" />
	</root>
</configuration>
```

daarnaast moet er in de application.properties een extra veld worden toegevoegd “logging.file” met als waarde de logging file path.

## 2.5 Migrations

Migraties zijn een aanpak om te zorgen dat de database altijd up-to-date is met de codebase. De migrations van de engine worden automatisch uitgevoerd bij het opstarten van de engine. Daardoor is de database altijd up-to-date, mits de migrations up-to-date zijn.

Wanneer er een verandering in het datamodel van de engine wordt aangebracht, moeten er migrations worden gemaakt. Dit wordt gedaan door de volgende commando’s uit te voeren. Let op: de database van de desbetreffende microservice moet draaien.

```
cd /path/to/microservices/
cd <microservice>/
./gradlew diffChangeLog -PrunList=main
```

## 2.6 Configuratie

### 2.6.1 Engine configuratie

De engine kan geconfigureerd worden door middel van environment variabelen.
In de environment variabelen moeten tenminste de volgende dingen staan:

- Database credentials. usernames, wachtwoorden en URLs van de database en de test database;
- Job offer timeouts. de tijd in seconden die een uitvoerende gebruiker heeft om op een job offer te reageren;
- Max job offers. Het maximum aantal uitvoerende gebruikers die een opdracht aangeboden krijgt;
- Auth host. De URL van de Auth microservice;
- Availability time unit. De unit die wordt gebruikt bij de beschikbaarheid van een uitvoerende gebruiker in seconden;
- Job offer radius. De radius in meters waarbinnen een uitvoerende gebruiker moet zijn om een opdracht aangeboden te krijgen.
- JWT secret.
- JWT expiration.

### 2.6.2 Hosting

De engine wordt gehost in een kubernetes cluster. Om een nieuwe versie van de engine daar naartoe te pushen, kan worden gedaan door middel van bitbucket. Wanneer er een release/<version-number> branch naar de repository wordt gepushed, wordt deze branche automatisch naar de cluster doorgezet.
Hoe een microservice wordt gehost in de cluster, kan worden geconfigureerd in de file /kubernetes/engine-<name-of-microservice>.yml . In deze file staan een aantal variabelen. De belangrijkste zijn:

- replicas. Het aantal pods van een de desbetreffende microservice die parallel aan elkaar worden gerund.
- requests. De RAM en CPU van een enkele pod dat wordt gereserveerd in de cluster.
- limits. De maximale RAM en CPU van een enkele pod dat mag worden toegekend aan een enkele pod.

## 3. Documentatie van de interfaces

De applicatie kent 3 users, die elk hun eigen subpath hebben:

- /api/v1/appuser voor appusers
- /api/v1/customer voor customers
- /api/v1/admin voor admins

Deze paden zijn allemaal afgeschermd met authenticatie door middel van een JWT token en autorisatie.

Daarnaast zijn er ook enkele endpoint in de API waarvoor geen authenticatie nodig is.  Dit zijn endpoints die als doel hebben authenticatie tot stand te brengen, zoals

- login
- token refreshen
- registreren

Een uitzondering op het registreren is het registreren van een admin. Hiervoor is een ander admin account nodig. Er zit standaard een admin account in de engine.

Wanneer de de engine lokaal wordt gedraaid hebben de microservices een eigen poort waarop deze draaien. De Match microservice draait op poort 81 en de Auth microservice op poort 82. Daardoor moeten de API calls login en refresh naar “localhost:82” worden gestuurd en de rest van de API calls hebben de host “localhost:81”.

De documentatie van de engine is te vinden op https://engine9.docs.apiary.io. Bij de API calls zijn zijn de responses met statuscode 403 en 401 weggelaten. Wanneer 401 als status code wordt teruggegeven, is de authenticatie niet gelukt. Wanneer 403 als statuscode is gegeven, heeft de gebruiker niet de juiste rol om de API call uit te voeren.