

HELMINK

BACKEND VOOR HET HABDESK PROJECT REALISEREN

Ruben van Gemeren - 0964326

HELMINK HOGESCHOOL ROTTERDAM

PARTIJEN

Onderwijsinstelling	Hogeschool Rotterdam Communicatie, Media en Informatietechnologie (CMI) – Informatica Wijnhaven 107, 3011 WN Rotterdam
Helmink	Helmink B.V. Job Bellekom (Algemeen Directeur) Wolweverstraat 4, 2984 CD Ridderkerk
Opdrachtgevers	Job Bellekom Algemeen Directeur en Development manager Helmink j.bellekom@helmink.nl Tim van der Waal Operationeel Manager Helmink t.vanderwaal@helmink.nl
Stagebegeleider	Stelian Paraschiv Docent HBO informatica Wijnhaven 107, 3011 WN Rotterdam i.s.paraschiv@hr.nl
Uitvoerende partij	Ruben van Gemeren Student Informatica Barendrecht, Zuid-Holland, Nederland 0964326@hr.nl

VOORWOORD

Dit document genaamd "*Backend voor het HABdesk project realiseren*" is opgesteld tijdens een periode van vier maanden tussen 7 februari en 20 juni. Dit onderzoek maakt deel uit van de hbo-opleiding Informatica van de Hogeschool Rotterdam en dient als afstudeerscriptie. Het onderzoek is uitgevoerd bij Helmink gevestigd in Ridderkerk.

Het rapport bevat de volgende hoofdstukken: Hoofdstuk 2 beschrijft Helmink als bedrijf, daarna wordt het project en de scope toegelicht in hoofdstuk 3. In hoofdstuk 4 worden de onderzoeksmethodes per deelvraag beschreven. De huidige situatie van Helmink en het HABdesk project staat in hoofdstuk 5 en de *stakeholders* analyse volgend door de software eisen komen aan bod in hoofdstuk 6. In hoofdstuk 7 staat het theoretisch onderzoek. Hoofdstuk 8 geeft adviezen op basis van de eerder beschreven onderzoeken. Hoofdstuk 9 licht de architectuur van de *backend* van het HABdesk project toe en laat het ontwerp voor het *Proof of Concept (PoC)* zien. In hoofdstuk 10 wordt de implementatie van het *PoC* uitgewerkt met daarbij de methodes die zijn gebruikt om de code te testen. De conclusie van het onderzoek staat beschreven in hoofdstuk 11 waarna de discussie plaatsvindt in hoofdstuk 12. Tot slot bevat hoofdstuk 13 het nawoord en de reflectie.

Mijn dank gaat naar alle medewerkers van Helmink voor de gastvrijheid en alle hulp die ik deze periode nodig had. Ik wil Job Bellekom bedanken voor alle adviezen en informatie die ik heb mogen ontvangen. Ik wil Tim van der Waal bedanken, als aanspreekpunt binnen Helmink heeft hij veel van zijn tijd besteed aan het helpen met alle complicaties en problemen. Het development team bestaande uit Thijs Treffers, Tim van Herwijnen en Wouter Pols wil ik bedanken voor de technische hulp, waarbij ik de heer Pols extra wil bedanken voor het helpen met het realiseren van het *Proof of Concept*.

Tot slot wil ik Stelian Paraschiv bedanken voor alle feedback en tips met betrekking tot de stage en de scriptie.

Ik wens u veel leesplezier toe,

Barendrecht, 2022

Ruben van Gemeren

SAMENVATTING

Helmink is een IT-communicatie bedrijf, het telt ongeveer 15 medewerkers en is vooral gericht op het verkopen en onderhouden van de communicatienetwerken bij hun klanten. Sinds 2015 is het bedrijf actief bezig met het opzetten en uitbreiden van een development afdeling. Eén van de grootste projecten binnen Helmink heet HABdesk, dit is een webapplicatie dat de klantgegevens vanuit verschillende systemen haalt en op een duidelijke manier centraal weergeeft. HABdesk komt voort uit het dagelijkse probleem waar Helmink last van heeft, de klantinformatie is verdeeld over meerdere systemen die niet met elkaar gekoppeld zijn.

De *backend* van het HABdesk project is een complexe samenvoeging van verschillende systemen en databronnen waarin koppelingen gevormd moeten worden. Uit deze stelling is de volgende onderzoeksvraag gekomen: "**Hoe kan de backend van het HABdesk project binnen Helmink worden ontwikkeld om de werkdruk van Helmink te verlichten?**"

Om de onderzoeksvraag te beantwoorden is er inzicht nodig in de huidige situatie van Helmink en het HABdesk project. Uit onderzoek naar de huidige situatie blijkt dat Helmink een klein development team heeft, dat het HABdesk project een samenvoeging is van databronnen die binnen een web portaal wordt weergegeven en dat dit volledig intern ontwikkeld wordt.

Ook blijkt dat het documenteren van werkzaamheden een organisatie breed probleem is dat voor veel onduidelijkheid zorgt.

Vanuit een *stakeholdersanalyse* en het onderzoeken van de huidige situatie is geconcludeerd welke volgende koppeling ontwikkeld moet worden, dit is de koppeling met het netwerk monitoringsysteem genaamd Auvik.

Uit onderzoek naar software architectuur en software integraties zijn een aantal conclusies getrokken:

- Binnen het veld software architectuur gaat alles om compromis, geen enkele oplossing is ooit perfect.
- De taken van een software architect zijn het beheren van het development proces en een goede communicatie met de *product owner(s)* onderhouden.
- Er zijn veel verschillende architecturen, deze kunnen opgedeeld worden in twee categorieën. centraal (*monolithic*) en decentraal (*distributed*).
 - Een centraal systeem is makkelijker te ontwikkelen maar mist een niveau van schaalbaarheid.
 - Een decentraal systeem heeft een complexere implementatie maar heeft potentieel een hogere schaalbaarheid en prestatie.
- Er zijn vier methodes voor het opzetten van integraties. *Point-to-Point*, *Hub-and-Spoke*, *Enterprise Service Bus (ESB)* en een *cloud* oplossing.
 - *Point-to-Point* is de simpelste versie van een integratie waarbij data van één punt naar een ander punt wordt gestuurd zonder een tussensysteem.
 - *Hub-and-Spoke* maakt gebruik van een centraal systeem dat alle data ontvangt en weer doorstuurt naar de juiste ontvanger.

- *Enterprise Service Bus (ESB)* is een zwaar pakket van losse systemen dat de datastromen kan overzien en beheren.
- *Cloud* oplossingen bestaan uit een combinatie van de bovenstaande punten binnen een *cloud* omgeving met alle voor- en nadelen die daarbij komen kijken.

Uit het onderzoek blijkt dat de *ESB* en de *Cloud* oplossingen niet geschikt zijn voor de integraties omdat deze te duur en te complex zijn voor Helmink.

Hoewel de andere integratie methodes voor het development team eenvoudiger zijn om te implementeren, hebben deze op een grote schaal meer nadelen dan voordelen. Het advies is om de huidige implementatie door te ontwikkelen en geen grote aanpassingen te maken binnen de architectuur. In plaats daarvan moet er gewerkt worden naar een decentrale architectuur door middel van kleine stappen binnen de huidige centrale architectuur.

Vanuit het onderzoek is een ontwerp gemaakt van het *Proof of Concept (PoC)* met de hulp van het C4 model dat een systeem opdeelt in vier lagen, de *context* laag, de *container* laag, de *component* laag en de *code* laag. Het *PoC* is een realisatie van de koppeling tussen het netwerkmonitoringssysteem Auvik en de server de *backend* van het HABdesk project. Het *PoC* bestaat uit drie delen, het ophalen van de data, het koppelen van de data en het visualiseren van de data. Het is gebouwd in *PHP* met het *framework Laravel* en maakt gebruik van API-verzoeken vanuit Auvik.

SUMMARY

Helmink is an IT communication company, it has about 15 employees and is mainly focused on selling and maintaining the communication networks from their customers. Since 2015, the company has been actively setting up and expanding a development department.

One of the largest projects within Helmink is called HABdesk, this is a web application that retrieves customer data from various systems and displays it centrally. HABdesk has arisen from a problem that Helmink suffers from daily, the customer information is spread over several systems that are not linked to each other, this means extra time needs to be spent finding the right data.

The backend of the HABdesk project is a complex combination of different systems and data sources in which links must be formed. The following research question emerged from this statement:

"How can the backend of the HABdesk project be developed to relieve Helmink's workload?"

To answer the research question, insight into the current situation of Helmink and the HABdesk project is needed. Research into the current situation shows that Helmink has a small development team, that the HABdesk project is a merge of data within a web portal and that it's completely internally developed. Also the documentation is an organization-wide challenge that causes a lot of problems.

Based on a stakeholder analysis and an investigation of the current situation, it was concluded which integration should be developed next, this is the network monitoring system in which all customer networks are managed.

A number of conclusions have been drawn from research into software architecture and software integrations:

- Software architecture is all about compromise, no solution is ever perfect.
- The tasks of a software architect are to manage the development process and maintain good communication with the product owner(s).
- There are many different architecture structures, that can be divided into two categories. centralized (monolithic) and decentralized (distributed).
 - A centralized system is easier to develop but lacks a level of scalability.
 - A decentralized system has a more complex implementation but potentially higher scalability and performance.
- There are four methods for setting up integrations. Point-to-Point, Hub-and-Spoke, Enterprise Service Bus (ESB) and a cloud solution.

- Point-to-Point is the simplest version of an integration method where data is sent from one point to another point without an intermediate system.
- Hub-and-Spoke uses a centralized system that receives and forwards all data to the correct recipient.
- Enterprise Service Bus (ESB) is a heavy package of separate systems that can oversee and manage data flows.
- Cloud solutions consist of a combination of points mentioned above within a cloud environment with all the advantages and disadvantages that come with a cloud environment.

The research shows that the ESB and the Cloud solutions are not suitable for the use of integrations because they are too expensive and complex for Helmink.

While the other integration methods are easier for the development team to implement. On a large scale, they have more disadvantages than advantages. The advice is to continue developing the current implementation and not to make major adjustments within the architecture. Instead, it is necessary to work towards a decentralized architecture through small steps within the current centralized architecture.

Based on the research mentioned above, a design was made of the Proof of Concept (PoC) with the help of the C4 model that divides a system into four layers, the context layer, the container layer, the component layer and the code layer. The PoC is a realization of the integration between the network monitoring system Auvik and the server, which is the backend of the HABdesk project. The PoC has three tasks, retrieving the data, linking the data and visualizing the data. It is built in PHP with the Laravel framework and uses API requests from Auvik.

INHOUDSOPGAVE

1. Inleiding	11
2. Bedrijfsbeschrijving.....	12
Bedrijfsstructuur.....	12
Diensten	13
Use Cases	14
ISO27001	15
Van telecom naar Development	15
3. Projectbeschrijving.....	16
Opdrachtomschrijving.....	16
Probleemstelling.....	16
Doelstelling.....	17
Vraagstelling	18
Gewenst resultaat.....	18
Scope	19
Kwaliteitseisen en randvoorwaarden	19
4. Onderzoeksmethode	20
5. Huidige situatie	22
Development team.....	22
Development proces.....	22
Het HABdesk project	24
Loader server / Wasmachine	24
Development Tools.....	25
Authenticatie.....	25
MVC Architectuur support	25
Intact Security.....	27
Artisan	27
Systemen voor Klantgegevens	28
6. Requirements	29
Stakeholderanalyse	29
Software Requirements Specification	31
Software requirements (software eisen)	31
Het opstellen van Requirements	31
Het controleren van Requirements	32
Requirements voor CRIS-X naar IT-Glue.....	32

Belangrijkste onderdelen.....	32
Requirements voor Auvik API naar IT-Glue.....	32
Belangrijkste onderdelen.....	32
7. Theoretisch kader	34
Software Architectuur	34
Software architect.....	35
Software architect vs Software developer.....	35
Cohesion en coupling.....	37
Architecture Charactaristics	38
Architecture styles	39
Monolithic vs Distributed Architectures	39
Overzicht verschillende structuren	40
Software Integraties.....	42
Onderdelen van software integraties	43
Soorten software integratie methodes	45
8. Advies	47
Welke soort integratie moet er gebruikt worden?.....	47
Point-to-Point.....	47
Hub-to-Spoke	49
ESB.....	49
Cloud oplossing	49
Waarom een eigen implementatie?	50
Welke soort architectuur moet er worden toegepast? (voor de Loader server)	51
Welke architectuur wordt er nu gebruikt?	52
Welke architectuur moet er gebruikt worden?	53
Waarom is het opstellen van requirements zo belangrijk voor Helmink?	54
Hoe moeten requirements worden verzameld?	55
Welke technieken kunnen Helmink helpen met het documenteren van de projecten?	55
Welke onderdelen van de Agile methode kunnen worden toegepast?	56
Waarom moet het C4 model gebruikt worden?	57
Het advies voor Helmink	60
Wat is er nodig om (de backend van) het HABdesk project te ontwikkelen?	60
9. Architectuur.....	61
Huidige architectuur backend HABdesk	61
Het Core systeem	62
<i>Plug-in Components</i>	65
Architectuur karakteristieken	69

Functionele volledigheid (Functional completeness)	69
Schaalbaarheid (Scalability).....	69
Herbruikbaarheid (Reusability)	70
Vervangbaarheid (Replaceability)	70
FOUTTOLERANTIE (Fault tolerance)	70
Herstelbaarheid (Recoverability)	70
Transitie Architectuur.....	71
C4 model.....	72
Het doel van C4.....	72
Het model.....	73
Niveau 1: System Context diagram	75
Niveau 2: Container diagram	75
Niveau 3: Component diagram.....	77
Niveau 4: Code: <i>Sequence diagram</i>	78
Testen	79
10. Implementatie	80
Versiebeheer	80
GitHub Desktop.....	81
<i>Best practices</i> binnen versiebeheer	82
Code Standaarden en conventies	85
Ontwerp principes.....	85
Case Styles.....	88
Documentatie	94
<i>Proof of Concept</i>	96
Functies.....	96
Auvik API.....	97
Auvik netwerk data opslag	101
Koppeling met klantgegevens.....	109
Visualisatie.....	113
Karakteristieken.....	122
Functionele volledigheid (Functional completeness)	123
Schaalbaarheid (Scalability).....	123
Herbruikbaarheid (Reusability)	123
Vervangbaarheid (Replaceability)	125
FOUTTOLERANTIE (Fault tolerance)	126
Herstelbaarheid (Recoverability)	127
Testen	127

Code Review	127
Testplan	129
API Unit test.....	129
Database testing	132
11. Conclusie	135
12. Discussie.....	137
13. Nwoord.....	138
14. Verwijzingen	139
5. Huidige situatie.....	139
6. Requirements	140
7. Theoretisch kader.....	140
8. ADVIES	142
8. Architectuur	144
10. Implementatie.....	145
Bijlagen	150

1. INLEIDING

Helmink is een IT-communicatie bedrijf gevestigd in Ridderkerk, het telt ongeveer 15 medewerkers en is vooral gericht op het verkopen en onderhouden van de communicatienetwerken bij hun klanten. Helmink is een tussenpersoon die services van KPN, T-Mobile maar ook Microsoft levert en onderhoudt. Sinds 2015 is het bedrijf actief bezig met het opzetten en uitbreiden van een development afdeling, met deze afdeling hoopt Helmink meer projecten te kunnen starten waar een combinatie van hardware en software bij komt kijken.

Eén van de grootste projecten binnen Helmink heeft HABdesk, dit is een webapplicatie die de klantgegevens vanuit verschillende systemen haalt en op een duidelijke manier centraal weergeeft. Dit project loopt sinds 2020. Er is vanuit het bedrijf moeite om concrete stappen en keuzes te maken.

De onderzoeksvraag luidt: "***Hoe kan de backend van het HABdesk project binnen Helmink worden ontwikkeld om de werkdruk van Helmink te verlichten?***"

Dit rapport bestaat uit drie delen. Het eerste deel is een onderzoek naar de huidige situatie van Helmink en het HABdesk project, wie is Helmink? Wat doen ze? Hoe ziet het development team eruit en welke technieken worden er gebruikt? Vanuit deze vragen wordt er bepaald welke onderdelen van het HABdesk project eerst ontwikkeld moeten worden.

Het tweede deel van het onderzoek gaat over software architectuur en software integraties. Dit zijn twee velden binnen softwareontwikkeling die de structuur binnen een systeem beschrijven en visualiseren. Dit literatuuronderzoek wordt samen met het onderzoek naar de huidige situatie gebruikt om advies te geven over de structuur van het HABdesk project en de werkwijze van het development team.

Ten slotte worden de keuzes die gemaakt zijn, gebruikt om een ontwerp en implementatie te realiseren van een deel van het HABdesk project. Deze realisatie wordt onderbouwd met afbeeldingen en diagrammen.

De opbouw van het rapport is als volgt: Hoofdstuk 2 beschrijft Helmink als bedrijf, daarna wordt het project en de scope toegelicht in hoofdstuk 3. In hoofdstuk 4 worden de onderzoeksmethodes per deelvraag beschreven. De huidige situatie van Helmink en het HABdesk project staat in hoofdstuk 5 en de *stakeholders analyse* volgend door de software eisen komen aan bod in hoofdstuk 6. In hoofdstuk 7 staat het theoretisch onderzoek. Hoofdstuk 8 geeft adviezen op basis van de eerder beschreven onderzoeken. Hoofdstuk 9 licht de architectuur van de *backend* van het HABdesk project toe en laat het ontwerp voor het *Proof of Concept (PoC)* zien. In hoofdstuk 10 wordt de implementatie van het *PoC* uitgewerkt met daarbij de methodes die zijn gebruikt om de code te testen. De conclusie van het onderzoek staat beschreven in hoofdstuk 11 waarnaar de discussie plaatsvindt in hoofdstuk 12. Tot slot bevat hoofdstuk 13 het nawoord en de reflectie.

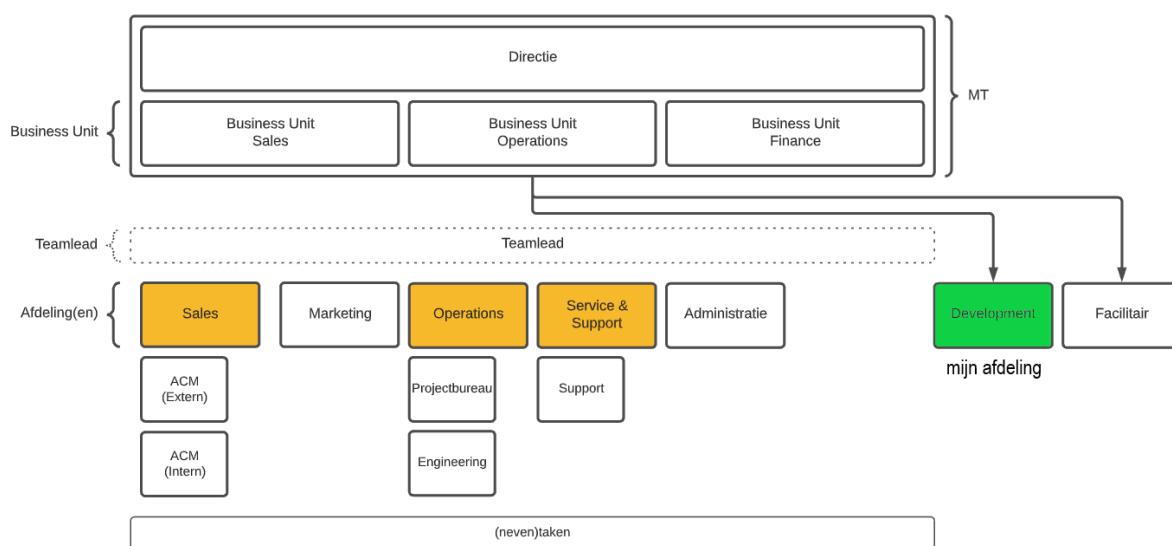
2. BEDRIJFSBESCHRIJVING

Dit hoofdstuk beschrijft de omgeving waarin het onderzoek is uitgevoerd. Allereerst wordt de structuur van het bedrijf beschreven, vragen zoals "Wat is de omvang van het bedrijf?" en "Wat zijn de taken van het bedrijf" worden beantwoord. Vervolgens worden de diensten en de *core-business* principes van het bedrijf beschreven. Met de hulp van een paar voorbeelden wordt het duidelijk waar het bedrijf zich bevindt in de IT-Communicatie branche en welke visie het heeft op het gebied van innovatie en techniek.

BEDRIJFSSTRUCTUUR

Helmink is een full-service communicatie IT-specialist gevestigd in Ridderkerk. Het bedrijf telt 15 medewerkers en wordt opgedeeld in zeven afdelingen. De bedrijfscultuur binnen Helmink is er één waarbij er geen vaste leiding is en waar de verschillende afdelingen dicht naast elkaar werken. Door de vele activiteiten die Helmink uitvoert, is het noodzakelijk dat er een strakke interne communicatie aanwezig is.

Elke afdeling heeft een 'verantwoordelijke' die aanspreekbaar is voor zijn of haar afdeling, daarnaast wordt er van elke medewerker verwacht dat deze zelf initiatief neemt als het gaat om het melden van problemen of veranderingen binnen de projecten. Na een interview met de directeur is het duidelijk dat Helmink actief op zoek is naar nieuwe ideeën en een ander perspectief binnen een branche die constant in verandering is. Door de relatief kleine omvang kan Helmink niet met mankracht boven de concurrentie uitsteken, hierdoor is een proactieve houding naar verantwoordelijkheid en zelfontwikkeling essentieel. In *Figuur 2.1 organogram* is een organogram te zien van de zeven afdelingen, waarbij de gele vlakken aangeven welke afdelingen verantwoordelijk zijn voor het grootste percentage van de werkzaamheden. De development afdeling is aangegeven met groen.



Figuur 2.1 organogram

DIENSTEN

Hoewel Helmink een multidisciplinaire aanpak heeft, is Managed Service Provider (MSP) op het gebied van telecommunicatie de *core-business*. Een MSP levert services zoals netwerken, applicaties, infrastructuur en beveiliging met doorlopende support en actieve administratie bij de klanten. Deze services worden door de MSP zelf gehost of aan een externe hosting maatschappij overgelaten.

Helmink beheert in totaal bijna 300 klanten en levert dagelijks support voor bedrijven met honderden medewerkers.

Het doel van Helmink is om kleine en grote bedrijven te helpen met de interne én externe communicatie, een betere samenwerking en meer inzicht in de datastromen binnen en buiten het bedrijf.

Oplossingen

Helmink biedt meerdere oplossingen die per klant gecombineerd kunnen worden voor een volledig gepersonaliseerde set van diensten. Dit omvat:

Managed KPN en T-Mobile:

Helmink is een gecertificeerd business partner van zowel KPN als T-Mobile. Dit houdt in dat ze toegang hebben tot de services die KPN en T-Mobile aanbieden en advies kunnen geven over de zakelijke communicatie en oplossingen van deze providers.

Omnichannel contactcenter:

Een contactcenter behandelt niet alleen de telefoongesprekken binnen een bedrijf maar ook de e-mails, chats en tickets. Helmink is in staat om een volledig center in te richten waarbij de benodigde data altijd beschikbaar is. Bij omnichannel ervaart de consument geen kanalen van communicatie meer, maar één bedrijf dat altijd bereikbaar is. Er is één uniforme ervaring, één prijs, één levertijd etc.

Omnichannel is een strategie, waarbij meerdere onderdelen samenwerken om één gestroomlijnd kanaal te creëren, waardoor de *customer journey* wordt verbeterd. Omnichannel kan gebruikt worden in verschillende branches van een organisatie.

Multisite location manager:

Helmink gebruikt Multiprotocol Label Switching (MPLS) en Software Defined Wide Area Network (SD-WAN) om verschillende locaties met elkaar te verbinden. Onderdelen zoals beveiliging en monitoring kunnen nu voor alle vestigingen worden gebruikt.

Ook is dit te combineren met de andere diensten die Helmink levert om als klant nog meer controle te hebben over de verschillende datastromen.

USE CASES

Door de jaren heen heeft Helmink met veel grote klanten mogen werken. Elke klant brengt eigen uitdagingen met zich mee waar rekening mee moet worden gehouden. Voor Helmink is de klantbeleving van hoge prioriteit, daarnaast is het één van hun specialisaties.

“De meeste Softwarebedrijven zijn op technisch gebied verder maar hebben geen idee hoe ze de klant daadwerkelijk enthousiast kunnen maken. Hier zijn wij dan wel weer goed in.”

- *Bellekom, J – Algemeen Directeur Helmink*

Case - DAKA Sport

DAKA is een expert als het gaat om sportartikelen en sport-casual gerelateerde producten. Tijdens de samenwerking tussen DAKA en Helmink is er na een onderzoek de Multisite Location Manager geïmplementeerd, hiermee hoopt DAKA een optimale klantbeleving te creëren.

“Voor ons is het zeer belangrijk dat de geïmplementeerde oplossingen bedrijfszekerheid bieden, zodat onze dienstverlening ongestoord door kan gaan. Helmink is hierin voor ons de juiste partner. De kwaliteit is van een hoog niveau.”

- *Stuivenberg, P.*

Naast het verbinden van de vele vestigingen die DAKA heeft, zijn er ook back-up systemen en interne communicaties opgezet die de gebruikerservaring zal verhogen.

Case - VORM

VORM is een landelijk opererende ontwikkelaar en bouwer. Ze ontwikkelen en bouwen op locaties waar mensen graag willen wonen, zowel binnen als buiten de stad.

VORM is de afgelopen jaren flink gegroeid, dit zorgde voor problemen bij de interne communicatie. Na een overleg met Helmink is een *Mobile First* strategie toegepast. Deze strategie zorgt ervoor dat VORM ten alle tijden bereikbaar is op diverse platformen.

“Het kennis en kunde niveau van de medewerkers van Helmink is op een hoog niveau. Hierdoor verliep het contact op een zeer aangename manier. Ook was er tijdens de implementatie nooit onduidelijkheid.”

- *Orgelist, P*

Binnen de bouwsector gaat Internet of Things (IoT) een steeds grotere rol spelen. VORM is met onder andere Helmink bezig met het onderzoeken naar nieuwe integratiemogelijkheden.

Case - overig

Naast deze cases levert Helmink oplossingen en diensten aan grote en kleine klanten. Per klant wordt er gekeken wat de beste combinatie van diensten kan zijn. Bij veel nieuwe klanten zijn er al werk- en communicatieprocessen aanwezig, Helmink moet flexibel kunnen omgaan met de bestaande communicatieomgeving. Dit vereist goed contact tussen Helmink en de klant.

ISO27001

De ISO27001 is een internationale standaard op het gebied van informatieveiligheid. Het wordt gebruikt als keurmerk om aan te tonen dat een bedrijf veilig omgaat met zowel de gebruikers- als personeelsdata. Binnen de development van nieuwe oplossingen moet er rekening worden gehouden met de manier waarop data wordt opgeslagen en projecten worden gedocumenteerd. Helmink is in maart 2022 opnieuw gecertificeerd. Tijdens de certificering is de kwaliteit van de softwareprojectdocumentatie geprijsd. Deze documentatie is voortgekomen uit dit onderzoek. Zie *Bijlage 5. – Software Requirements voor de documentatie*.

VAN TELECOM NAAR DEVELOPMENT

Helmink is ooit begonnen als puur telecombedrijf, waar telefoons en abonnementen werden verkocht. Sinds 2015 zijn ze actief bezig met het opbouwen van een eigen Development afdeling, met deze investering wil Helmink zelf oplossingen kunnen bieden op het gebied van IT-Communicatie. Helmink heeft verschillende interne systemen ontwikkeld om de werknemers te helpen met het leveren van diensten. Er is een eigen klantrelatie en opslag applicatie ontwikkeld genaamd CRIS-X, hierin worden alle klanten verwerkt en beheerd.

De afgelopen twee jaar is Helmink bezig geweest met het ontwikkelen van nieuw product genaamd HABdesk, dit is een zusterbedrijf van Helmink dat bezig is met het ontwikkelen van een web portaal waar klanten van een Managed Service Provider (MSP) hun eigen data kunnen inzien en voor een deel kunnen bewerken. Dit is een groot project dat veel impact zal hebben op de werkprocessen die nu binnen Helmink plaatsvinden.

3. PROJECTBESCHRIJVING

Om een beeld te krijgen van de scope van het project wordt in het hoofdstuk 'Projectbeschrijving' het project beschreven. Aan de hand van een voorbeeld wordt het probleem van Helmink toegelicht, daarna wordt de probleemstelling uitgelegd en beschreven wat de rol is van dit onderzoek. De kwaliteitseisen van Helmink en de Hogeschool Rotterdam worden ook genoemd.

OPDRACHTOMSCHRIJVING

Helmink is sinds 2020 bezig met ontwikkelen van een eigen *self service* portaal genaamd HABdesk. Dit project wordt ontworpen als een centrale locatie waar serviceproviders en klanten inzicht hebben in hun eigen data, daarnaast is het mogelijkheid om bepaalde acties uit te voeren. De omvang van het HABdesk project is erg groot voor de drie full-time developers die met dit onderzoek werkzaam zijn binnen Helmink.

In de huidige situatie is de ontwikkeling van het HABdesk project ongestructureerd en zijn geen duidelijke besluiten gevormd over de architectuur van de *backend*.

PROBLEEMSTELLING

Helmink beheert veel klanten. Het leveren van 24/7 support is één van de meest tijdsintensieve diensten die Helmink aanbiedt. Bij grote storingen moet heel het bedrijf actief meewerken aan het verbinden van klanten en het beantwoorden van vragen over de status van apparaten. Om de diensten te kunnen leveren, maakt Helmink gebruik van vele externe services om data te kunnen opslaan, apparaten te kunnen monitoren en klantcontact te kunnen beheren. Deze partijen hebben allemaal hun eigen architectuur en gebruikersomgeving.

Als een probleem is geconstateerd, wordt deze aangemaakt binnen het ticket systeem Zendesk dat Helmink gebruikt. Dit is een pakket waarin tickets zichtbaar zijn voor de klant.

Het probleem bij het afhandelen van tickets is het handmatige zoekproces dat plaats moet vinden om de juiste informatie te achterhalen. Er zijn weinig datakoppelingen aanwezig waardoor een medewerker zelf moet bedenken welke data bij elkaar hoort. Binnen Helmink zijn al minimaal 12 systemen aanwezig waar klantdata in is opgeslagen.

De huidige situatie kost Helmink veel tijd en geld. De tijd die nu verloren gaat door het verouderde systeem had gebruikt kunnen worden om betere oplossingen te verzinnen en meer klanten te kunnen helpen. Helmink is bereid om een grote investering te maken om een schaalbare oplossing te ontwikkelen.

De reden waarom het probleem zich nog steeds voordoet, ligt bij de complexiteit van de oplossing. Er moeten veel beslissingen worden genomen voordat een daadwerkelijk product kan worden ontwikkeld.

DOELSTELLING

De doelstellingen voor het onderzoek zijn bepaald aan de hand van de visie die Helmink heeft voor het HABdesk project. Na het opzetten van de doelstellingen zijn deze gecontroleerd door de Hogeschool Rotterdam.

Voor de doelstellingen is gebruik gemaakt van drie verschillende niveaus die het probleem van Helmink algemeen en specifiek beschrijven. De niveaus die worden gebruikt zijn:

- **Strategische doelstellingen:** Leggen de focus op het doel, niet (of zo weinig mogelijk) op de middelen. Dit zijn besluiten die worden genomen met een uitkijk op de toekomst, en gaan over de visie van het bedrijf op een langer termijn, voor Helmink is dit ongeveer 1-2 jaar.
- **Tactische doelstellingen:** Gaan over de middellange termijnen. De tactische doelstellingen zijn bedoeld om de kloof tussen de strategische en operationele doelstellingen te dichten. Er is hier geen vaste structuur voor en binnen sommige branches wordt er geen onderscheid gemaakt tussen een tactisch en een operationele doelstelling. Voor Helmink bestaat het tactische niveau uit een termijn van 1 jaar waarbinnen doelen gehaald moeten worden om dicht op de markt te blijven.
- **Operationele doelstellingen:** Gaan over de korte termijn, binnen Helmink gaat dit over een periode van een maand tot maximaal een half jaar. Deze doelstellingen worden gebruikt om de activiteiten van de huidige kwartalen te kunnen bepalen. Het betreft werkzaamheden die vaak op een korte termijn kunnen worden gerealiseerd. De doelstellingen gaan over het verbeteren van bestaande systemen of het ontwikkelen van kleine onderdelen van grotere projecten.

Voor het formuleren van de operationele doelen is er gebruik gemaakt van de SMART (Specifiek, Meetbaar, Acceptabel, Realistisch en Tijdsgebonden) methode om de duidelijkheid van de doelen te garanderen. De SMART methode helpt door de gebruikers ervan te迫eren om specifieke doelen vast te stellen, in plaats van vage beschrijvingen die in de praktijk niet gebruikt kunnen worden.

In *Tabel 3.1 Doelstellingen* hieronder worden de doelstellingen van het onderzoek benoemd, hoewel alle doelstellingen aansluiten op elkaar, wordt binnen de onderzoeksperiode gewerkt aan de tactische doelstellingen.

Niveau	Doelstelling
Strategisch	Het HABdesk portaal zorgt ervoor dat Helmink effectief en consistent service kan leveren wat zorgt voor een commerciële en interne groei van het bedrijf.
Tactische	Het HABdesk product speelt een grote rol binnen de interne en externe werkprocessen van Helmink en zijn klanten.
Operationeel	De backend van Helmink wordt ingericht aan de hand van de Enterprise Service Bus architectuur.

Tabel 3.1 Doelstellingen

VRAAGSTELLING

Op basis van de competenties vanuit de Hogeschool Rotterdam en de wensen van Helmink als opdrachtgever zijn zes vraagstellingen uitgewerkt, die het gehele project omschrijven. Het onderzoeken van deze vragen levert conclusies op die gebruikt kunnen worden door Helmink als *stakeholder*.

Mijn hoofdvraag luidt:

Hoe kan de backend van het HABdesk project binnen Helmink worden ontwikkeld om de werkdruk van Helmink te verlichten?

Om deze hoofdvraag te kunnen beantwoorden zijn vijf deelvragen opgesteld die samen antwoord geven op de hoofdvraag:

- Wat is de huidige situatie van het HABdesk project binnen Helmink?
- Wat zijn de systeemeisen voor de backend van het HABdesk project vanuit de behoeftes binnen Helmink?
- Wat is het ontwerp van de Loader server voor het HABdesk project?
- Hoe kunnen de integraties voor de backend binnen het HABdesk project worden gebouwd en beschikbaar worden gesteld?
- Hoe moet Helmink de backend van het HABdesk project testen?

Deze vragen zijn opgesteld aan de hand van besprekingen met het development team van Helmink en gecontroleerd door de stagebegeleiders vanuit de Hogeschool Rotterdam.

GEWENST RESULTAAT

Hier wordt het gewenste resultaat en de scope van het project aangegeven, daarbij wordt beschreven wat de verschillende onderdelen met elkaar te maken hebben.

Aan het einde van de stageperiode wordt een werkend Proof of Concept (PoC) opgeleverd, dat gebruik maakt van de bestaande processen binnen het bedrijf. Voordat er iets kan worden ontwikkeld, moet de huidige situatie worden geanalyseerd aan de hand van een SWOT-Analyse en een Hosin Kanri model. Met de kennis van het bedrijf en de development omgeving worden de eisen bepaald van de belangrijkste koppeling binnen de backend van het HABdesk project. Er wordt een onderzoek uitgevoerd, waarin de architectuur van software integraties wordt onderzocht en advies wordt gegeven welke architectuur het beste past bij het bedrijf en wat de mogelijke valkuilen zijn. De eisen voor het PoC worden uitgewerkt waarna er een ontwerp wordt gemaakt. Dit alles komt samen in een scriptie waarin de hoofd- en deelvragen worden beantwoord.

SCOPE

Het HABdesk project heeft een grote omvang en is binnen de aangegeven stageperiode van 20 weken niet afgerond of zelfs volledig uitgedacht, hierdoor is het belangrijk voor alle partijen om duidelijk te maken welke onderdelen in dit onderzoek worden beschreven.

Tijdens dit onderzoek wordt alleen onderzoek gedaan naar de *backend* van het HABdesk project, de *frontend* wordt niet onderzocht of meegenomen in het PoC. De *backend* is een aparte omgeving waar de interne en externe pakketten van Helmink worden gekoppeld door middel van een centrale server. Het onderzoek wordt gedaan naar de architectuur van deze *backend* en hoe deze kan worden ontwikkeld om uiteindelijk zuivere data te kunnen leveren aan het HABdesk project.

Voor het PoC wordt gewerkt met de huidige tools en technieken die binnen Helmink aanwezig zijn. Er worden geen nieuwe tools gebruikt, hoewel deze terug kunnen komen in het advies. De ontwerpen voor het PoC zullen de relevante context van het systeem beschrijven. Het advies dat wordt gegeven valt binnen een termijn van twee jaar en wordt gegeven in de veronderstelling dat Helmink minimale groei voorziet tot HABdesk als product in productie wordt gebracht.

Welke doelstellingen?

Welke beperkingen?

KWALITEITSEISEN EN RANDVOORWAARDEN

In de vorige paragraaf is vermeld dat er een PoC wordt ontwikkeld. Dit moet voldoen aan de eisen vanuit Helmink. De software moet de structuur volgen van de bestaande oplossingen en gebruik maken van de juiste benamingen, daarbij moet er wekelijks contact zijn tussen de developer en de leidinggevende. Er moet *API unit testing* worden toegepast voordat er gebruik kan worden gemaakt van de interne databases. De software moet aansluiten op de bestaande oplossingen van Helmink en werkbaar zijn voor de rest van het development team.

Binnen Helmink wordt alles voor het development team gedocumenteerd binnen Slite. De code wordt beheerd binnen Github en moet in je juiste *Repository* worden opgeslagen. Voordat code kan worden samengevoegd binnen Github moet het worden besproken met de leidinggevenden binnen het team.

4. ONDERZOEKSMETHODE

In dit deel van het onderzoek worden de onderzoeksmethodes beschreven die gebruikt zijn bij het beantwoorden van de deelvragen. Onderzoeksmethoden zijn specifieke benaderingen om data te verzamelen en te analyseren. Het onderscheid tussen kwalitatief en kwantitatief onderzoek wordt gemaakt, waarna wordt beschreven welke acties worden uitgevoerd om de onderzoeksraag te kunnen beantwoorden.

Kwalitatief onderzoek is onderzoek waarbij conclusies worden uitgedrukt in woorden. Interviews, observaties en literatuurstudies worden gebruikt om informatie te kunnen verzamelen over een bepaald onderwerp. Door de juiste mensen te spreken krijg je een beeld van de wensen en behoeftes van medewerkers en experts, deze data kan gebruikt worden om conclusies te trekken. Een valkuil bij kwalitatief onderzoek is het verifiëren van de bron, of het nu een persoon is of een website. Het is ook belangrijk om verschillende perspectieven mee te nemen binnen conclusie die worden getrokken.

Kwantitatief onderzoek is gericht op cijfers, met gebruikmaking van grafieken, tabellen en verzamelde data om conclusies te vormen. Kwantitatief onderzoek wordt veel gebruikt om theorieën en hypotheses te kunnen valideren. Door metingen uit te voeren en op grote schaal enquêtes af te nemen kan er zelf data verzameld worden voor het verdedigen van de probleemstelling.

Per onderzoeksraag wordt besproken welke methodes worden gebruikt om de onderzoeksraag te kunnen beantwoorden. Verdere toelichting van deze methodes staat in de **Bijlage**

Wat is de huidige situatie van het HABdesk project binnen Helmink?

Voor deze onderzoeksraag wordt een kwalitatief onderzoek gehouden, binnen dit onderzoek wordt de huidige situatie van Helmink beschreven aan de hand van interviews en observaties.

De interviews bestaan uit zowel één op één gesprekken, als teambesprekingen. De geïnterviewden zijn zowel technische als niet technische medewerkers van het bedrijf.

De observaties worden gebruikt om de kwaliteiten van het bedrijf in kaart te brengen. De observaties worden verwerkt in een SWOT-Analyse, dit is een manier om de sterke en zwakke punten van een organisatie samen te vatten. Om de doelen van Helmink te visualiseren wordt er gebruik gemaakt van een Hosin Kanri model zie *Bijlage 2. Hosin Kanri*, waar binnen vijf lagen de plannen en wensen van een organisatie worden samengevat.

Wat zijn de systeemeisen voor de backend van het HABdesk project vanuit de behoeftes binnen Helmink?

Om de systeemeisen te kunnen formuleren, wordt wederom een kwalitatief onderzoek uitgevoerd. Door de huidige situatie te onderzoeken en deze data te gebruiken als primaire bron wordt bepaald welke onderdelen prioriteit hebben. Elke week wordt gesproken met de *product owner* en het development team. Tijdens deze besprekingen worden via discussies en voorbeelden eisen bepaald. De eisen worden geformuleerd binnen een *System Requirements Specification*.

Wat is het ontwerp van de Loader server voor het HABdesk project?

Voor deze onderzoeksvraag is gebruik gemaakt van een kwalitatief onderzoek. Aan de hand van de besprekingen en uitwerkingen van de systeemeisen worden ontwerpen gemaakt voor de backend van het HABdesk project. Dit wordt de Loader Wasmachine genoemd. Een ERD voor de database, het C4 model voor een beter overzicht van de backend en meerdere UML-diagrammen met verschillende niveaus van abstractie worden gebruikt. Ontwerpen worden besproken tijdens wekelijkse meetings.

Hoe kunnen de integraties voor de backend binnen het HABdesk project worden gebouwd en beschikbaar worden gesteld?

Voor deze onderzoeksvraag wordt er zowel een kwalitatief, als een kwantitatief onderzoek gehouden. Om een overwogen conclusie te kunnen verwoorden, wordt *deskresearch* uitgevoerd over software integraties en de valkuilen die zich voordoen. Hierbij wordt gebruik gemaakt van onderzoeken en interviews met de directeur en het development team. Vanuit het onderzoek wordt een PoC ontwikkeld dat aansluit bij de conclusies en adviezen die zijn gegeven. Het PoC wordt ontwikkeld binnen de bestaande development omgeving van Helmink

Hoe moet Helmink de backend van het HABdesk project testen?

Voor het testen van het PoC en andere onderdelen van het HABdesk project wordt een testplan en testcase opgezet. Deze worden uitgewerkt aan de hand van onderzoek naar verschillende testmethodes. Er wordt gebruik gemaakt van *API unit testing* om het PoC te testen. Tests worden besproken met het development team en aangepast op basis van feedback.

5. HUIDIGE SITUATIE

In dit hoofdstuk wordt de huidige situatie van Helmink en het HABdesk project beschreven. Om HABdesk te ontwikkelen, is het van belang om in kaart te brengen wat de huidige situatie is en welke problemen zich voordoen binnen het bedrijf. Door de situatie te analyseren, is het mogelijk om de valkuilen en mogelijkheden in kaart te brengen.

De *backend* van het HABdesk project wordt onderzocht en hoe deze op de juiste manier kan worden ontworpen. Hiervoor is het belangrijk om te kijken naar andere soortgelijke producten en architecturen.

Het onderzoek naar de huidige situatie bestaat uit twee delen. Het eerste deel is een beschrijving van Helmink en de doelen die zij hebben opgesteld voor het bedrijf en de klanten. Deze doelen worden visueel gemaakt met de hulp van een Hosin Kanri model. Om de werkervaring binnen Helmink te kunnen peilen, is aan elke medewerker van Helmink gevraagd wat zij vinden van de huidige situatie binnen het bedrijf, daarnaast is een interview gedaan om dieper in te gaan op de technische kant van Helmink en het HABdesk project. Het interview is uitgewerkt in *Bijlage 12 – Interview Directeur Helmink*

Het tweede deel van dit onderzoek bestaat uit een beschrijving van de technische situatie waarin Helmink zich bevindt. Het doel hiervan is om in kaart te brengen wat de sterke en zwakke punten zijn van het development team, daarbij wordt beschreven wat voor effect dat heeft op de projecten die worden ontwikkeld. Met de hulp van een SWOT-Analyse worden de positieve en negatieve aspecten van het HABdesk project onderzocht.

DEVELOPMENT TEAM

Het development team bestaat uit een combinatie van *front-* en *backend* developers. Een daadwerkelijke leider/manager is niet aanwezig, wel is er een natuurlijke hiërarchie, die is ontstaan op basis van de kennis die developers bezitten. De *backend* developer heeft van nature meer controle over de beslissingen bij de *backend* van de projecten. In praktijk is de directeur van Helmink de manager van het development team. Elke week zijn er meerdere gesprekken over de lopende projecten en veel van de visies komen direct van hem. Bij afwezigheid van één van de teamleden is er niemand die zijn of haar taken over kan nemen, dit werk ligt stil totdat het lid terug is van verlof.

De kennis wordt niet expliciet binnen het team gedeeld. Tijdens meetings worden de werkzaamheden beschreven voor de andere teamleden. Deze beschrijvingen geven een globaal idee van de werkzaamheden, maar worden zelden in detail toegelicht.

DEVELOPMENT PROCES

Binnen Helmink ontbreekt een vast ontwikkelproces, in plaats daarvan worden verschillende losse elementen gebruikt van de Agile Scrum methodiek. Zo

vindt elke ochtend een stand-up (intern Dagstart genoemd) plaats, waarin elk teamlid drie onderdelen bespreekt.

- **Wat heb je bereikt sinds de vorige dagstart?**
 - Met de hulp van de vorige dagstart wordt aangegeven welke taken zijn afgerond met eventuele toelichting.
- **Wat ga je bereiken tot de volgende dagstart?**
 - Elk teamlid geeft met een toelichting aan welke taken hij hoopt af te ronden voor het eind van de dag.
- **Zijn er problemen waar je tegenaan loopt?**
 - Als er problemen zijn die de voortgang van het werk hinderen, worden deze hier besproken en samen nagedacht over een oplossing.

Naast de dagstart wordt gebruik gemaakt van Scrum meetings, waarin de werkzaamheden van het team worden besproken. Tijdens deze meetings worden de onderdelen van alle dagstarts van die week besproken. Een volledig uitgewerkte dagstart is beschreven in *Bijlage 10 – Development Dagstart 20 juni 2022*

Aan een paar onderdelen binnen Agile Scrum wordt een stuk minder aandacht aanbesteed. Zo wordt geen gebruik gemaakt van een *product backlog*. Het principe van *Sprints* wordt toegepast, maar hier is geen gestructureerde documentatie voor. Meetings worden genoteerd, maar dit kan niet worden geklassificeerd als een *Sprintreview* (*sprintreview*).

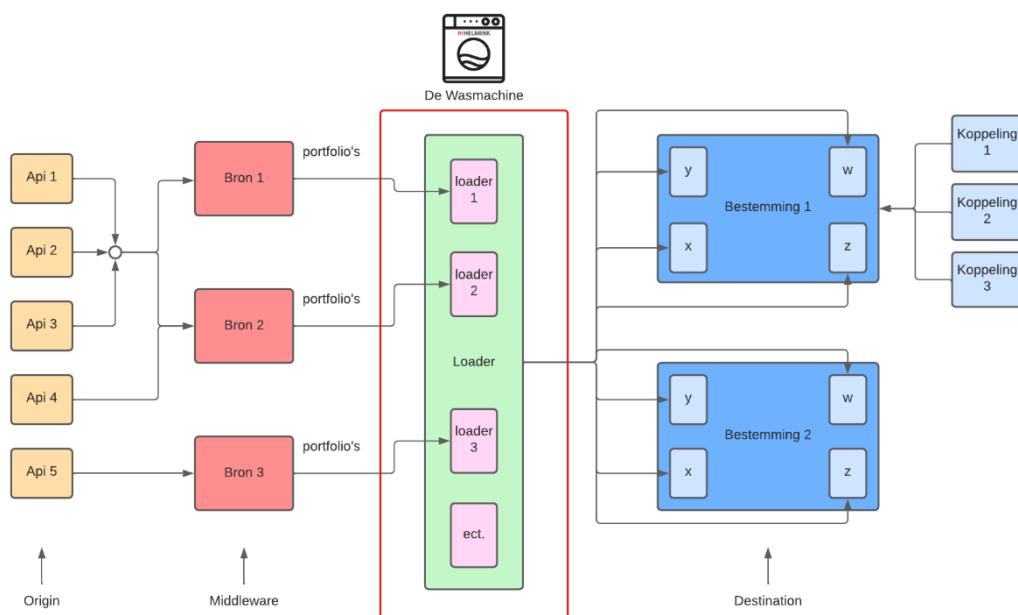
Documentatie

De documentatie van de projecten is gevarieerd en niet consistent. Schematische ontwerpen worden gemaakt en handleidingen geschreven maar dit gebeurt nauwelijks voor producten die nog in ontwikkeling zijn. Het gebruik van standaarden als het gaat om documentatie mist.

Uit één op één gesprekken met medewerkers van Helmink wordt het duidelijk dat de afwezigheid van gestructureerde documentatie een algemeen probleem is waar elke afdeling binnen de organisatie last van heeft.

HET HABDESK PROJECT

Het HABdesk (Handy Assistant Buddy Desk) project is een *selfservice* portaal dat wordt ontwikkeld voor intern en extern gebruik. Het project heet HABdesk maar bestaat niet alleen uit de applicatie zelf. Het portaal is de visualisatie van een uitgebreid *backend* systeem dat verschillende databronnen en portalen gebruikt om data te koppelen en *up to date* te houden (in sommige gevallen in *real time*). Dit systeem wordt binnen Helmink de "Wasmachine" genoemd door de gelijkenis met een echte wasmachine waarin vieze was (rauwe brondata) in de wasmachine (een server met verschillende Loaders die data verwerken en verrijken) gaat en wordt schoongemaakt tot schone was die je weer kunt opruimen (de verwerkte data dat gebruikt kan worden binnen onder andere HABdesk). *Figuur 5.1 – Overzicht Wasmachine* laat de structuur van de *backend* van het HABdesk project zien, het gedeelte omcirkeld met rood represeneert de wasmachine.



Figuur 5.1 Overzicht Wasmachine

LOADER SERVER / WASMACHINE

De Helmink "Wasmachine" bestaat uit meerdere losse systemen. Dit worden "Loaders" genoemd. Een "Loader" is een systeem binnen de "Wasmachine" dat data van een databron ophaalt en verwerkt voor een eindbestemming. Elke "Loader" bestaat uit dezelfde stappen maar moet apart worden ontwikkeld. Omdat er geen uniformiteit is tussen de verschillende databronnen moet per "Loader" worden vastgesteld welke data gestuurd moet worden.

Matching

Data kan niet zomaar worden overgezet van een bron naar een eindbestemming. Dit zou betekenen dat de databron leidend is in welke data er wordt opgeslagen binnen het portaal (de eindbestemming van de data), wat niet gewenst is.

Helmink wil zelf de controle houden over welke elementen worden opgeslagen en welke niet. Om dit te bereiken moet er een match worden gemaakt tussen een element van de bron en de eindbestemming. De match wordt gemaakt op een veld waarmee de data kan worden herkend, dit wordt ook wel de *Unique Identifier* (*primary key* of primaire sleutel) genoemd.

Door de vele verschillende systemen zijn er verschillen in categorisatie waardoor een computer niet kan weten welke data bij elkaar hoort. Door te kijken naar de *Unique Identifier* (*primary key*) worden verbindingen opgesteld die gebruikt worden om data vanuit een bron te verwerken en op te slaan. Na het maken van een match wordt deze opgeslagen binnen de Loader server, hierdoor hoeft de match niet opnieuw gemaakt te worden.

DEVELOPMENT TOOLS

Binnen het development team wordt gebruik gemaakt van Laravel, dit is een *opensource-PHP-webframework* dat is ontwikkeld voor het realiseren van webapplicaties volgens het Model-View-Controller-model software-architectuur

Laravel is het populairste PHP *framework* voor webdevelopment, dit komt door de grote hoeveelheid functionaliteiten die Laravel aanbiedt in vergelijking met standaard PHP. Hier zijn een paar van deze functionaliteiten

AUTHENTICATIE

Laravel biedt een eigen authenticatie applicatie aan die zonder veel werk gebruikt kan worden. Dit kan veel tijd besparen voor developers die authenticatie nodig hebben. Er zijn naast Laravel ook andere *frameworks* die uitgebreide authenticatie mogelijkheden bieden.

MVC ARCHITECTUUR SUPPORT

Laravel ondersteunt de MVC-architectuur.

"Het Model-view-controller is een ontwerppatroon dat het ontwerp van complexe toepassingen opdeelt in drie eenheden met verschillende verantwoordelijkheden: datamodel, datapresentatie en applicatielogica. Het scheiden van deze verantwoordelijkheden bevordert de leesbaarheid en herbruikbaarheid van code"

- (Wikipedia-bijdragers, 2021)

Door gebruik te maken van de MVC-architectuur worden de webapplicaties die gebouwd zijn met Laravel overzichtelijker en makkelijker te onderhouden.

De MVC-architectuur bestaat uit drie lagen die de logica, prestatie en databasetransacties verdeelt om het development proces overzichtelijker te maken. De splitsing van de code betekent dat meerdere developers zonder moeite tegelijkertijd aan hetzelfde systeem kunnen werken.

"MVC is a way to organize your code's core functions into their own, neatly organized boxes. This makes thinking about your app, revisiting your app, and sharing your app with others much easier and cleaner"

- (Codecademy, z.d.)

Het Model

Binnen een *model* worden de componenten van de applicatie beschreven. Deze componenten worden later in de code gebruikt om daadwerkelijk functionaliteit toe te voegen aan de applicatie.

De View

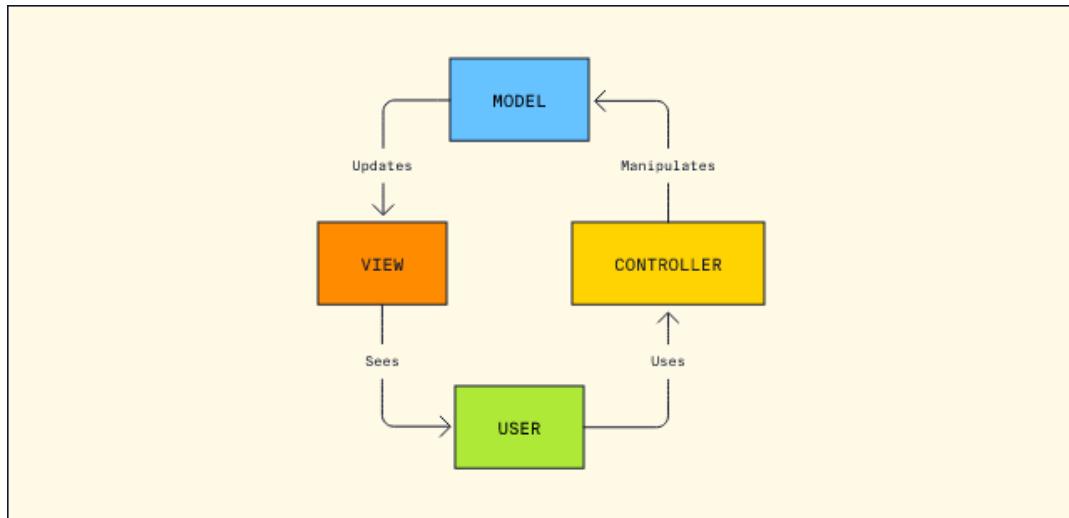
Binnen de *view* worden alle functies beschreven waar de eindgebruiker meer in aanraking komt. Het zorgt voor de opmaak van de applicatie en wat de gebruiker allemaal wel of niet kan.

De Controller

De code binnen de *controller* wordt gebruikt als een verbinding tussen het *model* en de *view*. Het krijgt gebruikersinput binnen en bepaald wat er moet gebeuren. De *controller* kan worden vergeleken met het brein van de applicatie, waarbij het *model* en de *view* worden verbonden.

Voordelen van Model View Controller (MVC)

Het grootste voordeel van de MVC-architectuur is de duidelijkheid van de code. Als alles is opgedeeld in drie duidelijke componenten is het zelfs bij grote complexe webapplicaties relatief eenvoudig om te begrijpen waar alles zich bevindt. Een developer die ervaring heeft met de MVC-structuur kan in één oogopslag zien hoe de applicatie is opgedeeld, dit scheelt veel tijd bij het oplossen van problemen. *Figuur 5.2 – MVC* laat een overzicht zien van de verschillende delen.



Figuur 5.2 MVC (Codecademy, z.d)

INTACT SECURITY

Laravel heeft zijn eigen webapplicatie beveiliging. Zo maakt het gebruik van Bcrypt Hashing Algoritme (Provost, 1999) om wachtwoorden te versleutelen voordat ze worden opgeslagen, waardoor het niet mogelijk is om de wachtwoorden direct vanuit de data te kunnen lezen.

ARTISAN

Artisan is een CLI, dit staat voor Command Line Interface (Ellis, 2019), dit is tekstuele *User interface (UI)* (online marketing agency, 2021) dat gebruikt wordt om met een applicatie of besturingssysteem te communiceren. Binnen Laravel kan er gebruik worden gemaakt van Artisan, waarmee je verschillende nuttige *commands* kan uitvoeren die kunnen helpen bij het onder andere managen de database.

SYSTEMEN VOOR KLANTGEGEVENS

CRIS-X

Dit is het Customer Relation Management Systeem (CRM) dat volledig is ontwikkeld door het development team van Helmink. Het bevat informatie over klanten, mogelijke klanten, offertes en locaties.

ITGLUE

IT-Glue is een framework dat wordt gebruikt voor alle IT-documentatie, wachtwoorden, en gebruikers van klanten te organiseren, structureren en op te slaan. Hierdoor wordt het oplossen van opkomende problemen een stuk eenvoudiger.

AUVIK

Auvik is een netwerkmonitoringsoftware dat wordt gebruikt voor netwerkcontrole en beheer van de klantnetwerken. Dit resulteert in betere zichtbaarheid, documentatie en monitoring van de klantnetwerken en automatiseert veel tijdrovende netwerktaken.

6. REQUIREMENTS

In dit hoofdstuk wordt beschreven wat de *requirements* zijn en hoe deze gebruikt worden voor het HABdesk project. De *requirements* die zijn opgesteld voor dit onderzoek worden beschreven aan de hand van voorbeelden. Om de systeemeisen op te zetten is er een stakeholderanalyse uitgevoerd binnen het bedrijf. Op basis van deze stakeholderanalyse zijn er eisen uitgewerkt.

STAKEHOLDERANALYSE

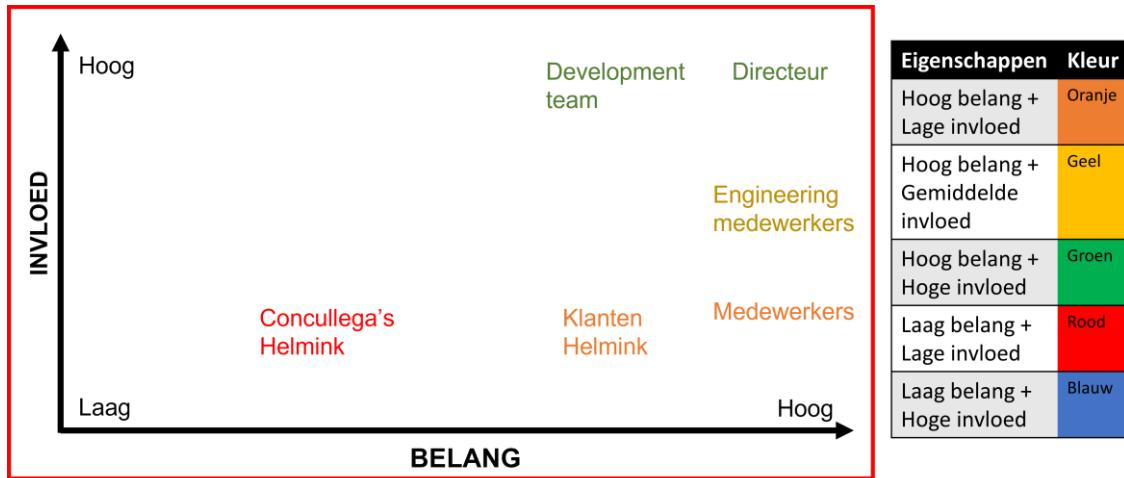
Een stakeholderanalyse richt zich op het evalueren en begrijpen van de stakeholders vanuit het perspectief van een organisatie. Ook wordt het gebruikt om de relevantie en invloed op een project te meten. (Bijvank, z.d.)

De eerste stap bij het opstellen van een stakeholderanalyse is het noteren van alle belanghebbende voor het project, dit zorgt voor overzicht en duidelijkheid van de verschillende partijen die direct of indirect betrokken zijn bij het project. In de *Tabel 6.1 – Belanghebbenden en hun invloed* hieronder is een overzicht te zien van alle partijen die belang hebben bij het project, er is een verdeling gemaakt tussen interne (Helmink) en commerciële belanghebbende.

Belanghebbenden	Belang
	Intern belanghebbenden (Helmink)
Medewerkers	Geoptimaliseerde support leveren
Directeur	Efficiënte workflows creëren
Engineering medewerkers	Geoptimaliseerde technische support leveren
Development team	Betere inzicht in datastromen creëren
	Commerciële belanghebbende
Klanten van Helmink	Meer overzicht in data, sneller support ontvangen
Concullega's van Helmink	Betere workflows opzetten, portfolio uitbreiden

Tabel 6.1 Belanghebbenden en hun invloed

Niet alle belanghebbende hebben evenveel invloed op de besluiten die worden genomen tijdens het ontwikkelproces, hierdoor is het belangrijk om de stakeholders te groeperen op basis van twee schalen, hoeveel belang er is en hoeveel invloed deze partij heeft. In *Figuur 6.1 - Grafiek Belang - Invloed* is de verdeling visueel gemaakt aan de hand van twee assen. Op de x-as is het belang weer gegeven, op de y-as staat de invloed die een belanghebbende heeft. De kleuren representeren verschillende eigenschappen.



Figuur 6.1 Grafiek Belang - Invloed

Het HABdesk project wordt vooralsnog alleen intern gebruikt, dit betekent dat de focus van de ontwikkeling ligt bij de onderdelen waar Helmink als bedrijf de meeste belangen bij heeft. De visie van Helmink is om het product verkrijgbaar te stellen voor concullega's die dezelfde problemen ervaren maar dit is tijdens het onderzoek niet de prioriteit.

De directeur van het bedrijf wordt de *product owner* genoemd en heeft het project gestart, dagelijks is er contact met het development team over de laatste veranderingen. De directeur heeft een grote invloed en veel belang bij het slagen van het project sinds hij tijd en geld investeert. De andere medewerkers van Helmink hebben behoefte aan het product maar minder invloed in het ontwikkelen ervan.

Het development team heeft veel invloed omdat zij het grootste deel van de technische kennis bezitten. Alle technische keuzes worden gemaakt met het team en de directeur, die zelf ook technische kennis heeft. In de *Tabel 6.2 Stakeholders en belangen* hieronder staat een samenvatting van alle partijen en bijbehorende belangen.

Stakeholder	Belang	Invloed
<i>Product owner</i> (Directeur Helmink)	Het product intern gebruiken	Wensen en eisen bespreken met development team.
Medewerkers Helmink	Geoptimaliseerde support leveren	Ervaringen uiten naar projectteam.
Development team	Het opleveren van het product	Het realiseren van wensen <i>product owner</i> .
Klanten van Helmink	Minder tijd besteden aan support	Assisteren met testen van product.
Uitvoerende partij	Afronden afstudeerstage	Onderzoekende, advies naar <i>product owner</i> en development team.

Tabel 6.2 Stakeholders en belangen

Het grootste knelpunt is de beschikbare tijd van de directeur van Helmink, de wensen en behoeftes voor het project komen vanuit één bron. Dit betekent dat het gehele project afhankelijk is van de aanwezigheid van de *product owner*, iets wat niet altijd mogelijk is.

SOFTWARE REQUIREMENTS SPECIFICATION

Aan de hand van de stakeholderanalyse worden er *requirements* (eisen) opgesteld voor verschillende delen van het project. Binnen de scope van dit onderzoek is het niet mogelijk om de eisen van het gehele project uit te werken, het project is hiervoor te groot en tijdens het onderzoek zijn deze eisen nog niet vastgelegd. Pogingen ervoor zijn wel gemaakt.

De *requirements* worden toegelicht aan de hand van voorbeelden. De volledige uitwerkingen zijn terug te vinden in *Bijlage 5 – Requirements CRIS-X Loader* en *Bijlage 6 – Requirements AUVIK API Loader*.

SOFTWARE REQUIREMENTS (SOFTWARE EISEN)

De termen *system requirements* en *software requirements* worden vaak door elkaar gebruikt, toch is er een duidelijk verschil. Het doel van *system requirements* is om een duidelijk beeld te geven van de behoeftes en wensen die zijn opgesteld door de stakeholders. *Software requirements* (software eisen) is een verzameling van al het gedrag en karakteristieken die een softwaresysteem (of deel van een systeem) bezit. (Davis et al., 1993)

Na verschillende pogingen vanuit de om de systeemeisen van de *backend* van het project op te stellen is er besloten om in plaats daarvan de *software requirements* op te stellen van de eerste twee koppelingen die gerealiseerd moeten worden, De koppeling tussen CRIS-X en IT-Glue en de koppeling tussen Auvik en CRIS-X.

HET OPSTELLEN VAN REQUIREMENTS

Het opstellen van de *software requirements* gebeurt met een *Software Requirement Specification (RSR)* document dat is opgesteld met de volgende hoofdstukken:

- Het doel van de software die wordt ontwikkeld.
- Een algemene beschrijving van de software.
- De functionaliteit van de software.
- De prestatie die de software moet leveren.
- De karakteristieken van de software.
- Externe *interfaces* en hoe deze communiceren met de software.
- Limitaties van de omgeving waarin de software zich bevindt.

Het document is compleet als de volgende doelen zijn bereikt:

- Alle functionaliteiten (en karakteristieken) van de software vermeld staat in het SRS-bestand.
- Alle pagina's en figuren zijn genummerd, alle termen zijn gedefinieerd en alle bronnen zijn vermeld.
- Alle 'To Be Determined' weg zijn. (Pataki, 2001)

HET CONTROLEREN VAN REQUIREMENTS

Het SRS-document wordt gecontroleerd aan de hand van besprekingen met het development team en de *product owner*. Verder moet het document voldoen aan de ISO27001 certificering waar Helmink aan voldoet.

REQUIREMENTS VOOR CRIS-X NAAR IT-GLUE

Eén van de belangrijkste onderdelen van de *backend* van het HABdesk project is de koppeling tussen het interne CRM-systeem CRIS-X en het externe Configuration management databasesysteem (CMDB) IT-Glue.

Voor het project moeten de 'organisaties' en de 'locaties' vanuit het CRM-systeem naar het CMDB-systeem worden gestuurd, om dit te doen is er een koppeling nodig.

BELANGRIJKSTE ONDERDELEN.

Binnen de software eisen wordt een verwijzing gemaakt naar de term '*Matching*'. Dit is de logica waarbij er op basis van een gekozen veld (of velden) een één-op-één relatie wordt aangemaakt binnen de database. Later in het proces is het mogelijk om data tussen de twee systemen te sturen.

Een ander punt dat van belang is bij deze koppeling zijn de condities die bepalen of informatie kan worden aangepast, dit is complex als het niet duidelijk is welke bron leidend is. Voor deze koppeling wordt informatie gestuurd als er een bepaald veld wordt aangepast en opgeslagen.

REQUIREMENTS VOOR AUVIK API NAAR IT-GLUE

Een centraal onderdeel waar veel systemen gebruik van maken is de netwerkmonitoring software genaamd Auvik, dit is een externe clouddienst die automatisch netwerken kan scannen en monitoren. Voor Helmink is het van belang dat data vanuit Auvik beschikbaar is voor IT-Glue maar ook andere systemen zoals CRIS-X. Door een koppeling met de centrale server te maken kan data worden gebruikt voor verschillende doeleinden.

BELANGRIJKSTE ONDERDELEN.

Het '*Matching*' is nogmaals van toepassing maar deze keer op een andere manier. Door de jaren heen is de data binnen de systemen van Helmink vervuild geraakt, dit

zorgt ervoor dat het niet duidelijk is welke data correct is. Binnen de logica van de software worden er extra condities toegevoegd om deze velden te controleren.

Voor deze software wordt er ook een visueel element ontwikkeld. Binnen dit element moet duidelijk zijn welke kunnen zien welke velden niet kloppen zodat deze handmatig kunnen worden aangepast. Dit element moet passen binnen de bestaande *User interface* (UI) van de server.

7. THEORETISCH KADER

In het hoofdstuk 'Theoretisch kader' worden de belangrijkste technische en theoretische onderdelen toegelicht. In het eerste deel wordt er beschreven wat software architectuur is en worden de belangrijkste aspecten toegelicht, daarna worden de populairste structuren samengevat met een overzicht. Het tweede deel bestaat uit een uitwerking van software integraties met een overzicht van vier integratie methodes. Dit hoofdstuk komt voort uit twee literatuur onderzoeken, deze onderzoeken zijn terug te vinden in *Bijlage 3 – Onderzoek Software Architectuur* en *Bijlage 4 – Systeemintegratie methodes*

SOFTWARE ARCHITECTUUR

De architectuur van een softwaresysteem is een grondlegging dat alle verdere beslissingen over dat systeem beïnvloed. Net als de grondlegging van een huis is het een basis dat correct moet worden uitgewerkt voordat er daadwerkelijk gebouwd kan worden. (Shrivastava & Srivastav, 2022)

Software architectuur is moeilijk te definiëren, het bestaat uit veel verschillende onderdelen en verantwoordelijkheden die alle aspecten van een softwareproject raken. Als er binnen een organisatie over een software architect en zijn taken wordt gesproken zijn er vier onderdelen die samen het vakgebied beschrijven. (Richards & Ford, 2020)

Structuur:

Bij de structuur wordt er verwezen naar de soort architectuur die wordt toegepast. Voorbeelden van structuren zijn *Layered Architecture*, *Event-Driven Architecture*, *Microkernel Architecture* en *Microservices Architecture*.

Hoewel de structuur van een architectuur een duidelijk beeld geeft van de globale ontwerpkeuzes die door het team gemaakt zijn, is dit niet het enige waar architectuur over gaat.

Karakteristieken

De *characteristics* (karakteristieken) van een architectuur beschrijven de succesfactoren die niet direct een functionaliteit beschrijven. De termen die ook vaak worden gebruikt zijn *non-functional requirements* of *Quality attributes*.

Beslissingen

Bij de *decisions* (beslissingen) worden de regels en beperkingen van een systeem geformuleerd. Deze worden gebruikt door het development team om de stroom van data uit te werken aan de hand van de opgezette systeemeisen.

Ontwerpprincipes

De termen *Design principles* (ontwerpprincipes) en *Architecture decisions* lijken veel op elkaar, toch hebben ze een ander doel. Waar een *Architecture decision* een vaste regel is waaraan voldaan moet worden, is een *design principle* meer een richtlijn. *Design principles* hebben vaak een grotere scope en een hogere abstractie.

Als voorbeeld: 'Waar mogelijk, maak gebruik van nested API-calls om relaties tussen klantdata zuiver te houden en het aantal API-calls te verminderen.'

SOFTWARE ARCHITECT

Zoals eerder genoemd zijn er verschillende activiteiten waar een software architect te maken mee krijgt. De belangrijkste taken van een architect zijn als volgt (Altexsoft, 2020):

- Evalueren van software systemen
- Het ontwerpen van globale software architecturen
- Onderzoek en noteren van risico's en valkuilen
- Ontwerpen van prototypes om levensvatbaarheid te bepalen

Deze taken omvatten het gehele development proces en worden geëvalueerd met het development team en *product owners*.

Naast de functies zijn er ook een aantal verantwoordelijkheden. Deze gaan over de taken die een architect heeft binnen het development team (*Software Architect Job Description [Updated for 2022]*, z.d.).

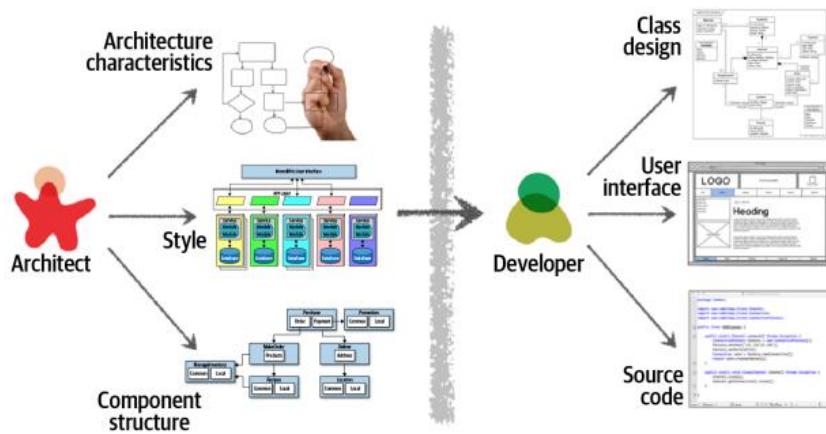
- Documenteren van softwareapplicaties en systemen
- Opstellen van planningen en schema's voor het opleveren van softwareproducten
- Het zoeken naar structurele problemen binnen softwaresystemen
- Het onderhouden van relaties met andere afdelingen zoals marketing, managers en operationele afdelingen.

Deze verantwoordelijkheden zijn breed en hebben te maken het onderhouden van projecten op organisatieniveau.

SOFTWARE ARCHITECT VS SOFTWARE DEVELOPER

Het verschil tussen een software architect en een software developer is niet direct zichtbaar, binnen veel kleinere bedrijven wordt er vaak geen onderscheid gemaakt sinds het team te weinig medewerkers heeft. Toch is er zeker een verschil op gebied van taken en verantwoordelijkheden.

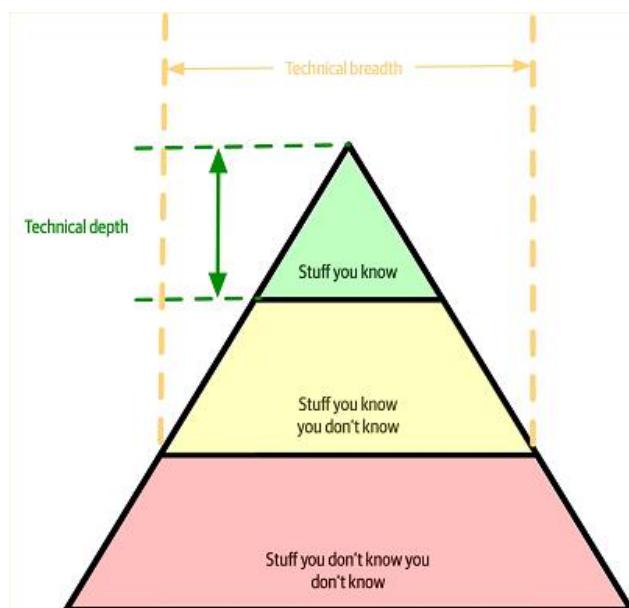
Binnen grotere organisaties met veel personeel worden de taken voor software architecten en software developers opgesplitst in taken die in *Figuur 7.1 - Architect vs Developer* te zien zijn. De architect werkt de structuur en kenmerken van een systeem uit, ondertussen werkt de developer aan de daadwerkelijke code en structuur van de database.



Figuur 7.1 Architect vs Developer (Richards & Ford, 2020)

Technische breedte

Een andere vergelijking die moet worden gemaakt met developers is het verschil binnen de technische kennis van het individu, dit verschil wordt weergegeven binnen *Figuur 7.2 - Technische breedte en diepte*. Een ontwikkelaar haalt zijn waarde uit de grote hoeveelheid kennis die hij of zij heeft over bepaalde technologieën, dit wordt gerepresenteerd door de groene lijn. Er moet veel tijd worden besteed aan het onderhouden van deze kennis, zeker in de wereld van de IT.



Figuur 7.2 Technische breedte en diepte (Richards & Ford, 2020)

Een software architect heeft ook technische kennis nodig, maar daarnaast is een technische breedte erg waardevol. Technische breedte (Technical breadth in *Figuur 7.2 - Technische breedte en diepte*) is bekend zijn met veel verschillende onderdelen, in plaats van veel kennis hebben van één. (Richards & Ford, 2020)

COHESION EN COUPLING

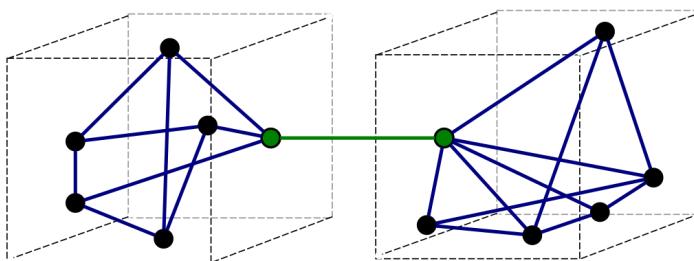
Twee termen die veel worden gebruikt bij het beschrijven van architecturen zijn *cohesion* en *coupling*. Deze termen vallen onder de overkoepelende term Modulariteit.

Modulariteit binnen software architectuur wordt gebruikt om de verdeling van een systeem te beschrijven. Hoe kan het systeem worden opgedeeld in kleinere 'Modules' en hoe worden deze opgebouwd en met elkaar verbonden.

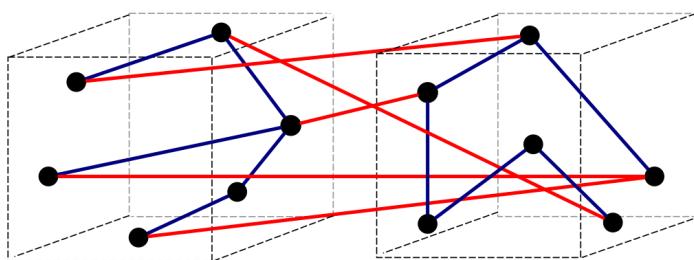
De *cohesion* (samenhang) van een systeem meet hoe de modules binnen een systeem op functioneel gebied zijn gekoppeld aan elkaar. Er wordt bepaald hoeveel taken een module heeft en of deze allemaal thuis horen daar. Er bestaan zeven soorten *cohesion* (Richards & Ford, 2020):

- Functional cohesion
- Sequential cohesion
- Communicational cohesion
- Procedural cohesion
- Temporal cohesion
- Logical cohesion
- Coincidental cohesion

Met *coupling* (koppeling) wordt de afhankelijkheid tussen twee modules gemeten, het wordt vaak samen met *cohesion* gebruikt. Een hoge *coupling* betekent dat er veel afhankelijkheid is tussen twee modules wat weer betekent dat er een lage cohesion is, het tegenovergestelde is waar bij een lage coupling. In *Figuur 7.3 Coupling vs Cohesion* hieronder is een visuele representatie te zien waarbij twee modules worden weergegeven als doorzichtige kubussen. De blauwe lijnen geven de *cohesion* weer van elke module de groen/rode lijn laat de *coupling* zien.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Figuur 7.3 Coupling vs Cohesion (Wikipedia contributors, 2022)

ARCHTECTURE CHARACTERISTICS

Eerder in dit hoofdstuk werd het concept van *architecture characteristics* beschreven als "succesfactoren die niet direct een functionaliteit beschrijven". Eén van de taken van een software architect is om de aspecten van het systeem te definiëren die niet direct te maken hebben met de bedrijfseisen.

Er bestaan *characteristics* voor elke soort software dat gebouwd kan worden, daarbij is er geen definitieve lijst van alle eisen. Voor de meeste projecten is er de officiële standaard opgezet door de *International Organization for Standards* (ISO - 2500), daarnaast is het mogelijk voor project om eigen eisen op te stellen op basis van de behoeftes vanuit de stakeholders. In *Tabel 7.1 Architecture characteristics* hieronder staan de *characteristics* beschreven, een verdere uitwerking van de *characteristics* zijn te vinden in *Bijlage 3 – Software Architectuur Hoofdstuk 'Lijst van architecture characteristics'*.

Functional Suitability - Functionele geschiktheid	Beschrijft hoe correct het product de systeem eisen naleeft en uitvoert, deze systeemeisen worden gedocumenteerd binnen documenten zoals <i>System Requirements Specifications</i> (SRS).
Performance efficiency - Efficiëntie van prestatie	Representeert de efficiëntie van het systeem bij vooraf bepaalde onderdelen en condities.
Compatibility - Compatibiliteit	Beschrijft in hoeverre een product, systeem of onderdeel kan communiceren met andere systemen of in hoeverre een systeem de nodige functies kan uitvoeren onder gelijke omstandigheden.
Usability - Bruikbaarheid	Beschrijft of het product of systeem kan worden gebruikt door gespecificeerde gebruikers om gespecificeerde taken effectief en naar verwachting uit te voeren.
Reliability - Betrouwbaarheid	Specificeert of het product of systeem gebruikt kan worden binnen een bepaalde context, met een bepaalde prestatie binnen een bepaald tijdslot.
Security - Beveiliging	Specificeert hoe goed het product of systeem is in het beveiligen en schermen van informatie voor gebruikers en waar deze gebruikers alleen informatie kunnen zien die voor hen van toepassing zijn.
Maintainability - Onderhoudbaarheid	Deze karakteristiek beschrijft in hoeverre het mogelijk is om het product of systeem aan te passen of te verbeteren en in welke mate dit effectief kan zijn voor nieuwe omgevingen en eisen.
Portability - Draagbaarheid	Beschrijft de mate waarin een product of systeem kan worden overgezet van de ene hardware naar de andere met een bepaald niveau van effectiviteit en efficiëntie.

Tabel 7.1 Architecture characteristics (*CaixaBank Obtains the ISO/IEC 25000 Functional Suitability Certificate for Their App CaixaBankNow, z.d.*)

ARCHITECTURE STYLES

MONOLITHIC VS DISTRIBUTED ARCHITECTURES

Een *monolithic architecture* is een op zichzelf staande structuur waarbij alle onderdelen en functionaliteiten van een systeem in één codebasis. Een *distributed architecture* wordt ook wel *microservices* genoemd en is in vergelijking tot een *monolithic* architectuur een structuur gebaseerd op meerdere kleinere componenten die met elkaar verbonden zijn via een communicatieprotocollen zoals HTTP (HyperText Transfer Protocol) of REST (Representational State Transfer).

Een *monolithic* architectuur heeft een paar voordelen, omdat alles op één plek staat is het een relatief eenvoudige structuur om te begrijpen en te implementeren, daarnaast is de prestatie van het systeem groter als dit wordt vergeleken met een *distributed* systeem.

De grootste nadelen van een opzichzelfstaande architectuur is de schaalbaarheid en de betrouwbaarheid van het systeem, fouten in de code kan het hele systeem onstabiel maken, daarbij is het uitbreiden en toevoegen van complexiteit een langdurig proces. (Richards & Ford, 2020)

De voordelen van een *distributed architecture* zijn de schaalbaarheid, beschikbaarheid en prestatie. Door het verdelen van het systeem in kleinere onderdelen kunnen de verschillende onderdelen los van elkaar worden ontwikkeld en is het toevoegen van nieuwe componenten eenvoudig. Een *distributed architecture* heeft een aantal valkuilen waar een *monolithic* systeem geen last van heeft. Een architect moet deze valkuilen meenemen bij het maken van beslissingen. De valkuilen van een *distributed architecture* zijn (Richards & Ford, 2020):

Het netwerk is stabiel

Bij een *distributed architecture* kan netwerkproblemen een groot deel van het systeem onbereikbaar maken, hiermee moet actief rekening worden gehouden tijdens het ontwerpen.

Bandbreedte is oneindig

Naast de betrouwbaarheid van het netwerk is de prestatie ook een onderdeel dat vaak wordt genegeerd. Bij het opsplitsen van een systeem is de communicatie tussen de systemen cruciaal.

Het netwerk is veilig

Het is belangrijk om duidelijk te maken dat een netwerk van nature niet veilig is, maar veilig gemaakt moet worden. Elk communicatiepunt moet beveiligd worden en er kan niet zomaar worden gewerkt met persoonlijke data. Veel bedrijven sluiten zich aan bij de ISO/IEC 27001 certificering over informatieveiligheid. Een certificering voor het omgaan met persoonlijke data en het beveiligen door middel van standaarden. (ISO 27001, z.d.)

De topologie verandert niet

De topologie van een netwerk is constant aan het veranderen, hardware en software moet vaak worden aangepast of opnieuw worden geconfigureerd, dit kan impact hebben op het netwerk. Als er geen rekening wordt gehouden met deze veranderingen kan het veel tijd kosten voordat een probleem bij een groot netwerk wordt herkend.

Er is maar één administrator

Binnen een bedrijf zijn er meerdere teams die allemaal eigen belangen en niveaus van kennis bezitten. Voor een projectteam is het soms moeilijk om goed contact te houden met de verschillende afdelingen van een organisatie. Het onderhouden van relaties en het bespreken van aanpassingen is een essentieel onderdeel dat veel impact heeft op de prestatie van het gehele systeem.

OVERZICHT VERSCHILLENDEN STRUCTUREN

Naam	Classificatie	Kenmerken
Layered Architecture	Monolithic	<ul style="list-style-type: none">✓ Eenvoudige implementatie✓ Eenvoudig testen✓ Lage kosten✗ Slechte schaalbaarheid✗ Moeilijk te onderhouden
Pipeline Architecture	Monolithic	<ul style="list-style-type: none">✓ Eenvoudige implementatie✓ Lage kosten✓ Eenvoudige aanpassingen✗ Slechte elasticiteit✗ Slechte schaalbaarheid
Microkernel Architecture	Monolithic	<ul style="list-style-type: none">✓ Eenvoudige implementatie✓ Lage kosten

		<ul style="list-style-type: none"> ✓ Hoge modulariteit ✗ Slechte schaalbaarheid ✗ Fout gevoelig
Service-Based Architecture	Distributed	<ul style="list-style-type: none"> ✓ Eenvoudige aanpassingen ✓ Veel flexibiliteit ✓ Lage kosten (voor <i>Distributed</i> systemen) ✓ Hoge betrouwbaarheid ✗ Slechte schaalbaarheid (voor <i>Distributed</i> systemen) ✗ Slechte elasticiteit (voor <i>Distributed</i> systemen)
Event-Driven Architecture (Broker topologie)	Distributed	<ul style="list-style-type: none"> ✓ Hoge schaalbaarheid ✓ Goede prestatie ✗ Geen controle over <i>workflow</i> ✗ Minder inzicht in problemen
Event-Driven Architecture (Mediator topologie)	Distributed	<ul style="list-style-type: none"> ✓ Wel controle over <i>workflow</i> ✓ Inzicht in problemen ✗ Lage schaalbaarheid ✗ Slechte prestatie
Space-Based Architecture	Distributed	<ul style="list-style-type: none"> ✓ Veel flexibiliteit ✓ Goede prestatie ✓ Hoge schaalbaarheid

		✗ Complexe implementatie ✗ Hoge kosten ✗ Slecht testen
Orchestration-Driven Service Oriented	Distributed	✓ Hergebruiken mogelijk ✗ Complexe implementatie ✗ Knelpunt
Microservices Architecture	Distributed	✓ Hoge schaalbaarheid ✓ Veel flexibiliteit ✓ Hoge modulariteit ✗ Onbetrouwbare prestatie ✗ Complexe implementatie

Tabel 7.2 Verschillende Structuren (Richards & Ford, 2020)

SOFTWARE INTEGRATIES

Software integration (integratie) is een tak van software development waar verschillende applicaties aan elkaar worden gekoppeld. Gartner definieert het als: Specifieke ontwerpen en implementaties die functionaliteiten of data binnen applicaties aan elkaar verbinden. (Gartner, z.d.)

Het concept van integraties is erg breed en kan op veel verschillende manieren worden geïnterpreteerd, daarbij zijn er ook veel verschillende modellen die worden gebruikt bij het koppelen van systemen.

Volgens een onderzoek gedaan door *Ai Ain* en de *SZABIST-universiteit* zijn er zes valkuilen bij het ontwikkelen van software integraties (RahimSoomro & Hasnain Awan, 2012):

1. Te weinig technische kennis
2. Niet inzien dat integratie software geen product is, maar een architectuur
3. Verwaarlozen van beveiliging, prestatie en monitoring
4. Het combineren van integraties met andere projecten
5. Integraties ontwikkelen zonder strategie
6. Slechte communicatie binnen het team

Omdat het bouwen van eigen software duur is, moet er rekening worden gehouden met deze valkuilen. Het negeren ervan kan leiden tot een architectuur die fundamenteel foutief is en resulteert in een verslechterde datastroom.

ONDERDELEN VAN SOFTWARE INTEGRATIES

Om een integratie te implementeren of te ontwikkelen zijn er een paar stappen die nodig zijn om de nodige context van een systeem te krijgen (Henderson, 2020).

- **Opstellen van systeemeisen**

Voor een project kan beginnen moeten de systeemeisen worden verzameld. Het doel van *system requirements* is om een duidelijk beeld te geven van de behoeftes en wensen die zijn opgesteld door de *stakeholders*. (Davis et al., 1993)

Het opstellen van deze eisen gebeurt aan de hand van observaties en interviews met medewerkers binnen de organisatie.

- **Situatieanalyse**

Na het opstellen van de verschillende systeemeisen moet de huidige situatie van het bedrijf in kaart worden gebracht. De reden hiervoor is omdat de zwakke en sterke punten van een organisatie veel invloed kunnen hebben op de scope van het project.

Deze punten kunnen op verschillende manieren worden samengevat. Een populaire methode is de SWOT-Analyse (Strengths, Weaknesses, Opportunities en Threats) of Sterkte-zwakteanalyse (Van der Linde, z.d.).

- **Ontwerpen systeem layout architectuur**

De ontwerpen hebben als doel om het systeem te visualiseren en context te geven voor de developer en software architecten die het systeem gaan realiseren. Een uitgebreid ontwerprinciple dat gebruikt wordt is het C4 model, dit is een model dat bestaat uit vier lagen die steeds meer abstractie aan het ontwerp toevoegen. (Brown, z.d.)

- **Opstellen van systeem management documentatie**

Voordat er daadwerkelijk kan worden gebouwd aan de oplossing is het belangrijk om het management op orde te hebben. Dit houdt in dat de ontwikkeling van het systeem goed verloopt en het voor iedereen duidelijk is wie wat doet en welke problemen zich voordoen.

Het gebruiken van de Scrum methode is een populaire manier om op korte termijn producten op te leveren, bij Scrum worden sprints van tussen 1-4 weken gebruikt waarin een deel van een product volledig afgerond moet zijn. (Petrova, 2019)

Het duidelijk documenteren van de voortgang is essentieel. Binnen het team moeten er keuzes worden gemaakt over welke *tools* er gebruikt kunnen worden. Github, Slite, Confluence, Trello, Slack, Discord, MS Teams ect.

- **Implementeren van systeem integraties**

Het meest tijdsintensieve onderdeel van het project is het implementeren van de integraties en alle code die daarbij hoort. Door een goede voorbereiding is het duidelijk welke onderdelen en er en niet moeten worden gebouwd.

- **Testen van systeem integraties**

Afhankelijk van het soort systeem dat wordt ontwikkeld, moet er gebruik worden gemaakt van bepaalde testmethodes. Bij integraties is het gebruik van Application Program Interfaces (API) populair. Met een API test kan de functies en limitaties van een API duidelijk worden. Als een onderdeel van het systeem is afgerond is het belangrijk om dit te testen met nep data om de verschillende aspecten te controleren.

SOORTEN SOFTWARE INTEGRATIE METHODES

Naam	Schaalbaarheid	Overige kenmerken
Point-to-point (P2P)	✗ Slechte schaalbaarheid	<ul style="list-style-type: none"> ✓ Eenvoudige implementatie ✓ Eenvoudig testen ✓ Lage kosten ✗ Fragiele structuur
Hub-and-Spoke (Enterprise Application Integration (EAI))	✓ Goede schaalbaarheid	<ul style="list-style-type: none"> ✓ Eenvoudige implementatie (<i>in de cloud</i>) ✗ Knelpunt ✗ Slechte prestatie door knelpunt
Enterprise Service Bus (ESB)	✓ Hoge schaalbaarheid (door SOA-structuur)	<ul style="list-style-type: none"> ✓ Goede prestatie (<i>door load balancing</i>) ✓ Hoge modulariteit ✗ Hoge kosten ✗ Complexe implementatie ✗ Veel extra componenten
iPaaS	✓ Goede schaalbaarheid	<ul style="list-style-type: none"> ✓ Kan overal werken (<i>volledig in de cloud</i>) ✓ Veel flexibiliteit ✓ Hoge betrouwbaarheid ✗ Afhankelijk van iPaaS provider.

		✗ Hoge kosten (bij maatwerk)
--	--	------------------------------

Tabel 7.3 Integratie methodes

8. ADVIES

Aan de hand van het onderzoek worden bepaalde adviezen gegeven over de manier waarop de onderzoeksraag kan worden beantwoord, deze wordt toegelicht in het hoofdstuk 'Projectomschrijving'. Eerst wordt beschreven welke soorten integratie structuren gebruikt moeten worden met een toelichting en verdere voorbeelden. De architectuur van de *backend* wordt besproken met een advies naar de verschillende soorten structuren en karakteristieken die gebruikt moeten worden voor het project. Na het technisch advies zijn er op basis van *fieldresearch* een aantal observaties gemaakt die de werking van het team en het project kunnen bevorderen. Het hoofdstuk wordt afgesloten met een algemeen advies richting Helmink met betrekking tot de onderzoeksraag.

WELKE SOORT INTEGRATIE MOET ER GEBRUIKT WORDEN?

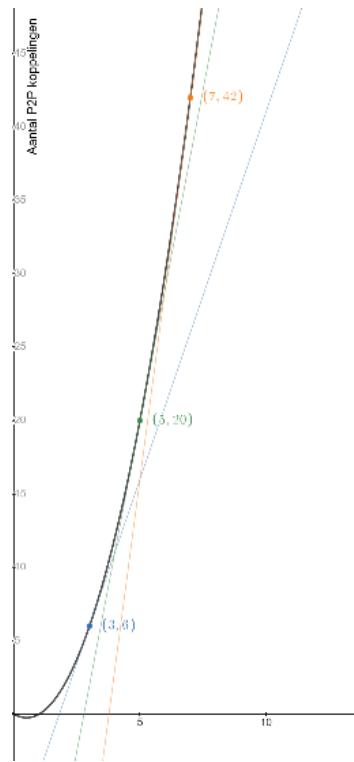
Bijlage 4 – Systeem integratie methodes bevat een onderzoek naar verschillende integratie structuren, in dit onderzoek worden een populairste manieren beschreven waarop integraties uitgevoerd kunnen worden, daarnaast wordt er aangegeven welke stappen doorlopen moeten worden om integraties te kunnen implementeren of ontwikkelen. Voor het kiezen van een integratie methode zijn er veel aspecten die invloed hebben op de structuur van de oplossing, per onderzochte methode wordt aangegeven waarom deze wel of niet geschikt is.

POINT-TO-POINT

Point-to-point (ook wel *P2P* genoemd) is de simpelste manier van een integratie, en bestaat uit een directe koppeling tussen twee systemen, ook wel een 1 op 1 relatie genoemd. Data wordt van systeem A naar systeem B gestuurd met daartussen eventuele mutaties.

Het is niet aangeraden om alleen P2P koppelingen te gebruiken binnen de *backend* van het HABdesk project, hier zijn twee redenen voor:

1. P2P koppelingen zijn van nature niet schaalbaar, voor elke nieuwe koppeling is een aparte verbinding nodig. Dit zorgt voor een exponentiële groei waarbij de hoeveelheid koppelingen $n(n-1)$ keer groeit in vergelijking tot de hoeveelheid systemen. In *Figuur 8.1 - P2P koppeling complexiteit* is deze groei te zien, waarbij de hoeveelheid P2P koppelingen (y-as) snel uit het zicht verdwijnt. Dit is niet gewenst voor Helmink omdat de *backend* van het HABdesk project schaalbaar moet zijn.



Figuur 8.1 P2P koppeling complexiteit

1. P2P koppelingen zijn erg fragiel door de directe afhankelijkheid die ze hebben met de systemen, als één van de systemen niet correct functioneert is direct de hele koppeling defect. De systemen die Helmink gebruikt zijn regelmatig defect en dus is het niet wenselijk om veel P2P koppelingen toe te passen.

Voor Helmink zijn P2P koppelingen bruikbaar, het zijn simpele verbindingen die niet veel computingkracht nodig hebben en een relatief eenvoudige implementatie en structuur bezitten. Voor de *backend* van het HABdesk project kan P2P gebruikt worden om de verschillende systemen te verbinden met de Loader server die al deze verbindingen verwerkt en een lokale database bezit. Mochten bepaalde koppelingen wegvallen is er nog data beschikbaar op de database van de server.

HUB-TO-SPOKE

Een natuurlijke evolutie van P2P is het *hub-and-spoke* model dat gebruikt maakt van een *message hub* die de verschillende P2P koppelingen samenbrengt tot één punt. Het lost een aantal problemen van de P2P structuur op maar past niet bij de *backend* van Helmink.

Het grote probleem van het *hub-and-spoke* model is het knelpunt dat ontstaat door gebruik te maken van één centraal punt waar alle bericht door communiceren. Het zorgt voor een *single point of failure* en is ook een knelpunt voor de prestatie van het systeem.

Het is niet gewenst voor Helmink om gebruik te maken van deze structuur door het *single point of failure*. Bij een slechte verbinding met een API zou het gehele systeem breken.

ESB

ESB staat voor *Enterprise Service Bus* en is opgemaakt uit vele verschillende services die samen communiceren om een groot pakket aan functionaliteiten aan te bieden, het kan worden gebruikt op locatie of in de *cloud*. Voor Helmink is een ESB in de huidige staat geen goede optie, hier zijn drie redenen voor:

1. De kosten van een ESB liggen hoog, per *provider* is het anders maar een *cloud hosted* ESB ligt rond de 20 tot 100 duizend euro per jaar. (Johnston, 2017) De prijs kan verschillende afhankelijk van de omvang van het bedrijf, nog steeds is het niet gewenst. Deze investering moet worden gebruikt om het team uit te breiden voor een verlichting in de werkdruk.
2. Een commerciële ESB bestaat uit veel componenten die allemaal een belangrijke rol spelen binnen het proces, bij het gebruik van een ESB moet het development team kennis hebben van het systeem om de juiste keuzes te maken.
3. Het zelf implementeren van een ESB is een grote taak voor de drie *full-time* developers binnen het bedrijf. Volgens Luuk Abels van Yenlo bestaat een goed ESB-development team uit de volgende leden: Enterprise architect, Solution architect, Information architect, Developers. (Abels, 2021) Het overnemen van deze taken is moeilijk voor een team dat deze expertise niet heeft. Uit een interview met de directeur van Helmink is dit duidelijk geworden, de volledige uitwerking van het interview is terug te vinden in *Bijlage 12 - Interview*.

CLOUD OPLOSSING

Het gebruik van de *cloud* heeft steeds meer invloed op de structuur binnen bedrijven, het gebruik van SaaS-applicaties groeit steeds (SaaS Market Size, 2021–2028, z.d.) meer en *cloud hosting* en *computing* wordt steeds goedkoper en is beschikbaar voor alle soorten bedrijven. (Supernor, 2018)

De overstep naar een *cloud* oplossing een logische stap is voor Helmink, en dit is al deels gebeurt. Uit het interview met de directeur van Helmink (te vinden in *Bijlage 12 - Interview*) is naar voren gekomen dat het HABdesk project en de bijbehorende Loader server allebei worden gehost in de *cloud*, niet bij Azure of AWS maar een lokale partij uit Amsterdam. Tijdens dit onderzoek is er gevraagd of een iPaaS oplossing een goede keuze kan zijn voor Helmink.

Uit dit interview zijn er drie redenen gekomen waarom Helmink geen *cloud* oplossing wil implementeren

1. De afhankelijkheid van een bepaald pakket, als een *service* wordt gebruikt is het moeilijk om later van *service* te wisselen. Voor een bedrijf als Helmink dat constant aan het veranderen en innoveren is het geen goed idee om van vele pakketten afhankelijk te zijn.
2. *Cloud* oplossingen kunnen niet alles bieden dat Helmink nodig heeft, er moet maatwerk worden verricht dat de prijs van de *service* omhoog brengt.
3. De prijs van vele *services* is volgens meneer Bellekom is nog steeds te hoog voor de functionaliteiten die het biedt, hij is van mening dat het beter is om een eigen oplossing te ontwikkelen.

De problemen die meneer Bellekom schets, zijn inderdaad de negatieve kanten van het toepassen van een *cloud service*. Toch is het ontwikkelen van een eigen implementatie ook niet zonder problemen en compromis.

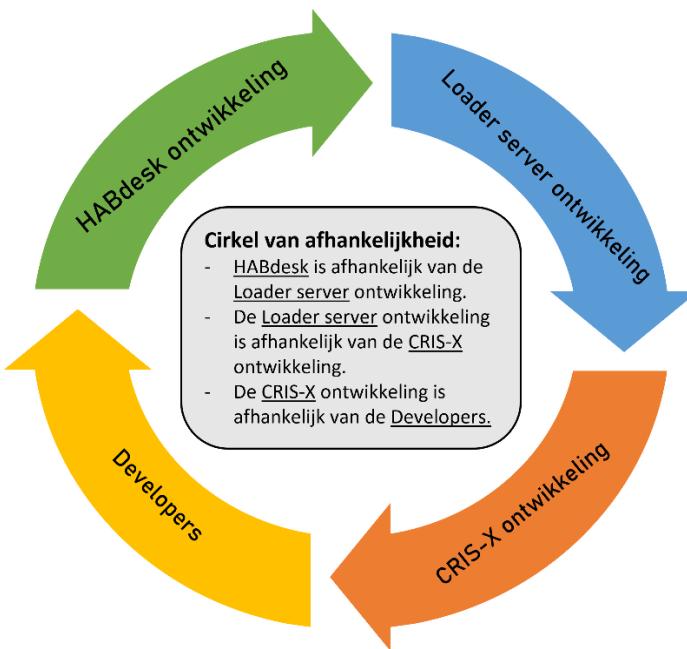
WAAROM EEN EIGEN IMPLEMENTATIE?

Voor het project wordt er nu een eigen implementatie ontwikkeld voor het HABdesk project en de bijbehorende Loader server, dit gebeurt volledig intern en neemt veel van de dagelijkse werkzaamheden in beslag. De ontwikkeling loopt sinds 2020 maar is in vanaf 2021 en het eerste kwartaal van 2022 echt aan het versnellen.

Hoewel de progressie zichtbaar is zijn er een drie problemen geobserveerd tijdens de eerste twee kwartalen van 2022:

1. Het development team bestaat uit drie full-time developers, er is kennis van *frontend* en *backend* maar deze kennis wordt niet actief gedeeld. Dit resulteert in een situatie waarbij één developer als enige kennis heeft over een bepaald onderwerp. Tijdens afwezigheid zorgt dit voor een vertraging in werkzaamheden, de zin "Als x terug is kan hier verder aan worden gewerkt" is een vaak terugkeerde opmerking.
2. De dagelijkse werkzaamheden binnen het team zijn erg verdeeld, het development team beheert het interne CRM-systeem genaamd CRIS-X. Dit systeem is nog niet afgerond en er komen wekelijks nieuwe ideeën over toevoegingen en aanpassingen. Tijdens een werkdag moet er een keuze worden gemaakt tussen het verbeteren van het bestaande systeem en het ontwikkelen van een nieuw systeem.
3. Het HABdesk project maakt gebruik van de Loader server, deze server beheert de interne systemen die Helmink gebruikt. Omdat niet alle systemen zijn afgerond zoals in het vorige punt wordt beschreven wordt de ontwikkeling

van het project ook vertraagt. Het creëert een cirkel van afhankelijkheid dat wordt gevisualiseerd in *Figuur 8.2 Cirkel van afhankelijkheid*.



Figuur 8.2 Cirkel van afhankelijkheid

Om het project in tijdig af te ronden moet er een oplossing worden bedacht voor de cirkel van afhankelijkheid. De ideale optie is om de ontwikkeling van het project uit te besteden aan een derde partij of nieuwe (wellicht tijdelijke) werknemers aannemen die focus kunnen leggen op de ontwikkeling van de Loader server parallel aan de ontwikkelingen van het CRIS-X systeem. Dit is erg afhankelijk van de huidige arbeidsmarkt en bedrijven als Helmink hebben veel moeite met het vinden van het juiste technische personeel.

De realistische optie is om de CRIS-X ontwikkeling tijdelijk stop te zetten, hierdoor kan er meer focus worden gelegd op de voortgang van de Loader server en HABdesk. CRIS-X als systeem moet eerst stabiel werkend zijn, zodat de intern gebruikt kan worden zonder verdere werkzaamheden.

WELKE SOORT ARCHITECTUUR MOET ER WORDEN TOEGEPAST? (VOOR DE LOADER SERVER)

In *Bijlage 3 - Architectuur* wordt er onderzoek verricht naar software architectuur en de stappen binnen het ontwerpen en ontwikkelen van een systeem op basis van een architectuur, daarnaast worden de taken en verantwoordelijkheden van een software architect benoemd. Om het beste advies te kunnen geven over de architectuur die gebruikt moet worden binnen de Loader server moet er ook buiten het project worden gekeken. Zo heeft de hoeveelheid tijd en de technieken die Helmink tot zijn beschikking heeft invloed om de mogelijkheden.

WELKE ARCHITECTUUR WORDT ER NU GEBRUIKT?

De architectuur die nu wordt gebruikt kan worden beschreven als een *Microkernel Architecture Style* met een *Modular core* dat gebruik maakt van *domain partitioned componenets*. hierdoor worde structuur toegelicht:

- ***Microkernel Architecture Style:***

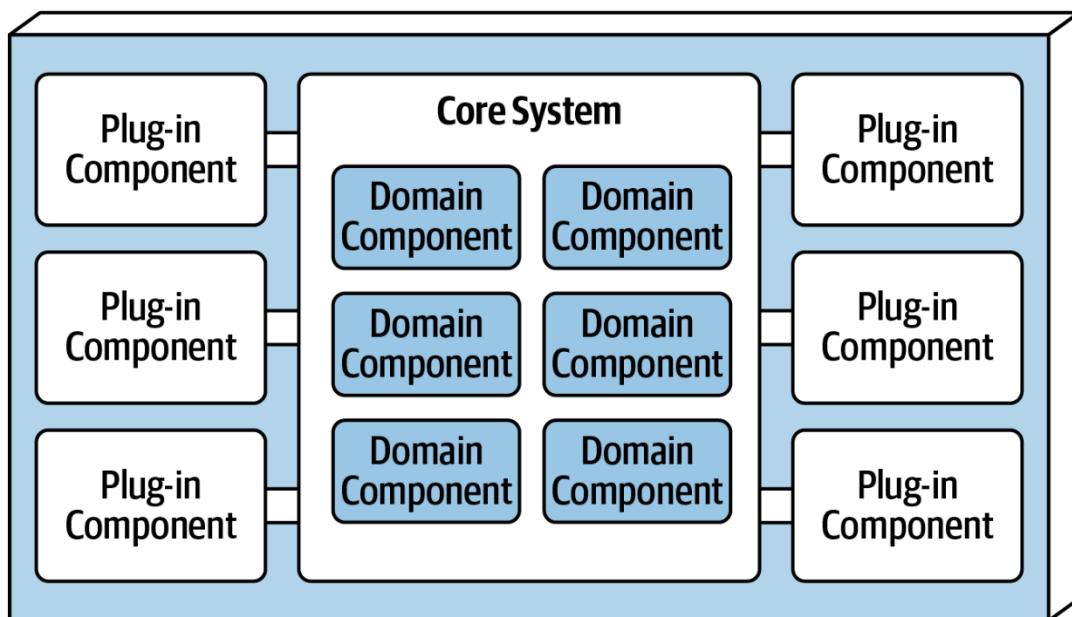
Dit is een stijl dat gebruik maakt van één *core system* dat algemene code bevat. De functionaliteiten worden toegevoegd aan de hand van verschillende *plug-in components* die specifieke functionaliteiten bieden en vaak worden gebruikt om integraties met andere systemen te faciliteren. In *Figuur 8.3 – Microkernel* is een schematisch overzicht te zien van de structuur. Deze structuur is *monolithic*, dit betekend dat alle code centraal beheerd wordt. (Richards & Ford, 2020)

- ***Modular core (domain partitioned)***

Modular core verwijst naar een *core system* dat is opgedeeld in meerdere delen, vaak om de verschillende *plug-in components* te kunnen beheren.

Een verdeling van het *core system* kan op twee manieren.

1. De eerste manier is om te verdelen op basis van de techniek, alle *frontend* onderdelen apart van de *backend* onderdelen.
2. Een andere manier is een verdeling op basis van het domein (*domain*), hierbij wordt er niet gelet op de techniek maar de taak die het onderdeel heeft binnen het project. Als voorbeeld, alle code die te maken heeft met het koppelen van een extern pakket wordt samengevoegd. Het koppelen van het externe pakket wordt het 'domein' genoemd.



Modular Core System (Domain Partitioned)

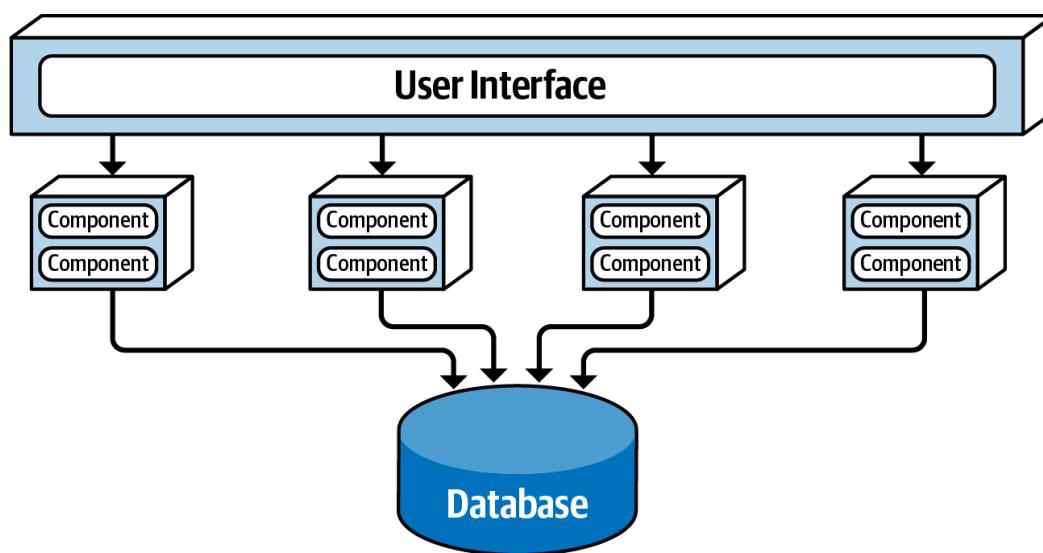
Figuur 8.3 Microkernel Architectuur (Richards & Ford, 2020)

Deze structuur heeft een paar voor- en nadelen. De grootste voordelen zijn dat het een relatief eenvoudige implementatie heeft, dit houdt de kosten ook laag, dit komt door de structuur die wordt gebruikt genaamd *monolithic architecture*. Een *monolithic architecture* is een op zichzelf staande structuur waarbij alle onderdelen en functionaliteiten van een systeem in één codebasis. Een *distributed architecture* is het tegenovergestelde omdat het een structuur is gebaseerd op meerdere kleinere componenten die met elkaar verbonden zijn via een communicatieprotocollen. (Richards & Ford, 2020)

Een *monolithic* structuur is makkelijker te ontwikkelen maar mist een niveau van schaalbaarheid. Ook is er een knelpunt aanwezig sinds het gebruik wordt gemaakt van één server, als deze server niet werkt is het gehele systeem buiten gebruik.

WELKE ARCHITECTUUR MOET ER GEBRUIKT WORDEN?

Om de problemen met de huidige structuur te verhelpen moet er gebruik worden gemaakt van een *service-based architecture style*. Dit is een structuur dat een *distributed architecture* toepast met één database. In *Figuur 8.4 – Service based* hieronder wordt een overzicht gegeven van de structuur.



Figuur 8.4 Service based architectuur (Richards & Ford, 2020)

Het gebruik van deze structuur in plaats van de *microkernel* structuur heeft een aantal voor- en nadelen, deze worden in *Tabel 8.1 – voor- en nadelen* beschreven.

Voordelen	Nadelen
Maakt gebruik van een <i>distributed</i> architectuur in plaats van een <i>monolithic</i> architectuur, dit zorgt voor een betere schaalbaarheid.	Door het gebruik van een <i>distributed</i> architectuur is de complexiteit van het systeem hoger, omdat de communicatie tussen de componenten beheert moet worden.

Net als bij de huidige structuur wordt er gebruik gemaakt van één database, hier heeft het development team ervaring mee.	De server moet anders worden ingericht, dit kan door meerdere servers op te zetten of verschillende virtuele machines op één server te zetten.
Er is geen afhankelijkheid meer tussen de onderdelen, hierdoor is het testen en aanpassen minder tijdsintensief.	

Tabel 8.1 Voor- en nadelen

In de periode waarin dit onderzoek plaatsvindt is er al een structuur bepaald en wordt deze ontwikkeld. Als de huidige situatie wordt meegerekend is het niet verstandig om de structuur aan te passen. De reden hiervoor is de extra complexiteit dat een *distributed architecture* met zich meebrengt, het development team is te klein om dit naast alle werkzaamheden te realiseren.

Als Helmink blijft groeien is het aangeraden om de eerder ontwikkelde producten (niet alleen het HABdesk project) op te schonen en te verbeteren voordat er nadacht wordt over nieuwe projecten. Dit probleem doet zich nu voor, zie *Figuur 8.2 Cirkel van afhankelijkheid*.

WAAROM IS HET OPSTELLEN VAN REQUIREMENTS ZO BELANGRIJK VOOR HELMINK?

Tijdens het onderzoek is er voor een periode van vijf maanden dagelijks meegelopen met het development team binnen Helmink. In deze tijd is er veel gewerkt aan het HABdesk project en de verschillende systemen die met HABdesk werken. Elke week is er een besprekking waarin de teamleden zijn of haar werkzaamheden bespreekt, verder zijn er dagelijks korte besprekkingen over de activiteiten van de dag. Bij het ontwikkelen van een nieuw systeem wordt er geen gebruik gemaakt van een *Software Requirements Specification* bestand. De *requirements* zijn alleen terug te vinden binnen notulen van de meetings waarin ze besproken zijn. Deze werkwijze zorgt voor veel verwarring binnen het team over de daadwerkelijke behoeftes van de *product owner*.

Op dit probleem op te lossen worden er 1-op-1 gesprekken gehouden tussen de *product owner* en de medewerker verantwoordelijk, deze gesprekken worden structureel niet gedocumenteerd. Het gebruiken van een vaste structuur in de vorm van een SRS-bestand kan helpen bij de onduidelijkheid die zich voordoet.

In de huidige situatie is er maar één *product owner*, dit is de directeur van Helmink, hierdoor ontstaat er een knelpunt sinds hij de enige is die duidelijkheid heeft over eisen van het project.

HOE MOETEN REQUIREMENTS WORDEN VERZAMELD?

De *system requirements* en *software requirements* moeten worden verzameld door middel van gesprekken met de *product owner*, deze gesprekken worden niet alleen genoteerd maar ook verwerkt in een SRS-bestand. Het product wordt ook gebruikt door andere medewerkers van Helmink, deze moeten ook worden geïnformeerd over de vorderingen van het project.

WELKE TECHNIEKEN KUNNEN HELMINK HELPEN MET HET DOCUMENTEREN VAN DE PROJECTEN?

In *Bijlage 1 – SWOT-Analyse* is de SWOT-Analyse te vinden die is uitgewerkt aan de hand van het analyseren van de huidige situatie. Eén van de zwaktes die is herkend is het niet consistent zijn als het gaat om het documenteren van alle projecten en meetings.

Anekdotisch is dit terug te zien binnen de interne documentatiesoftware, waar de uitvoerende partij in een periode van vijf maanden op nummer twee staat als het gaat om het schrijven en lezen van documenten. Elke functie binnen het bedrijf is anders en tijdens een actief onderzoek is het gebruikelijk om binnen een kort termijn veel documenten te maken en informatie op te nemen, toch is het ongebruikelijk om zo'n verschil te zien in activiteit.

Om het documenteren van onder andere het HABdesk project te bevorderen zijn er een paar mogelijkheden uitgewerkt en getest die kunnen helpen met het documenteren van projecten.

Begrippenlijst

Een probleem dat meerdere keren voor onduidelijkheden heeft gezorgd is de verschillende termen die worden gebruikt voor onderdelen en bronnen. Als voorbeeld, de term 'app' betekent twee compleet verschillende dingen binnen het HABdesk project, de ene term beschrijft een onderdeel binnen het product terwijl het andere binnen het authenticatie *framework* FusionAuth wordt gebruikt. Dit is één van meerdere instanties waarbij er een discussie ontstond tussen medewerkers. De oplossing hiervoor is het gebruik van een begrippenlijst.

In *Bijlage 11 - Definitielijst* is de begrippenlijst van het HABdesk project uitgewerkt waar veel gebruikte termen worden toegelicht.

Software Requirements Specification

Zoals eerder benoemd is het gebruik van een SRS-bestand en structuur erg van belang bij een project met de omvang van HABdesk. Dit project bestaat uit veel verschillende onderdelen die allemaal eigen functionaliteiten en karakteristieken bezitten. Een SRS-bestand dat is opgesteld met de hulp van de *product owner* is noodzakelijk.

Noteren problemen

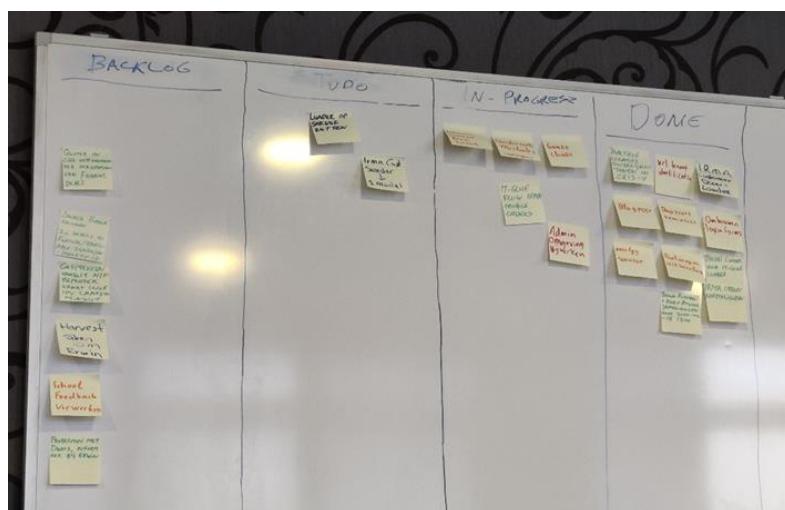
Vaak worden problemen alleen verbaal behandeld. Door problemen uit te schrijven is het voor alle betrokken medewerkers duidelijk en hoeft het niet herhaald te worden.

WELKE ONDERDELEN VAN DE AGILE METHODE KUNNEN WORDEN TOEGEPAST?

Voor het managen van project bestaan er voor methodes en technieken die het ontwikkelproces makkelijker maken. In het hoofdstuk 'Huidige situatie' wordt beschreven hoe er gebruik werd gemaakt van een 'dagstart'. Tijdens het onderzoek is het opgemerkt door de uitvoerende partij en een collega dat niet ieder lid actief meedoet met de dagstart, dit verslechtert de sfeer en maakt de besprekingen nutteloos, een actieve houding van alle medewerkers is gewenst.

Kanban

Een Kanban bord geeft de activiteiten van het team weer, door drie banen (To do, In progress, Done) te gebruiken is het duidelijk aan welke taken er wordt gewerkt. Om de projecten te managen is het Kanban bord opnieuw gebruikt met de hulp van kleuren om verschillende medewerkers aan te geven, in *Figuur 8.5 Kanban bord* is dit terug te zien.



Figuur 8.5 Kanban bord

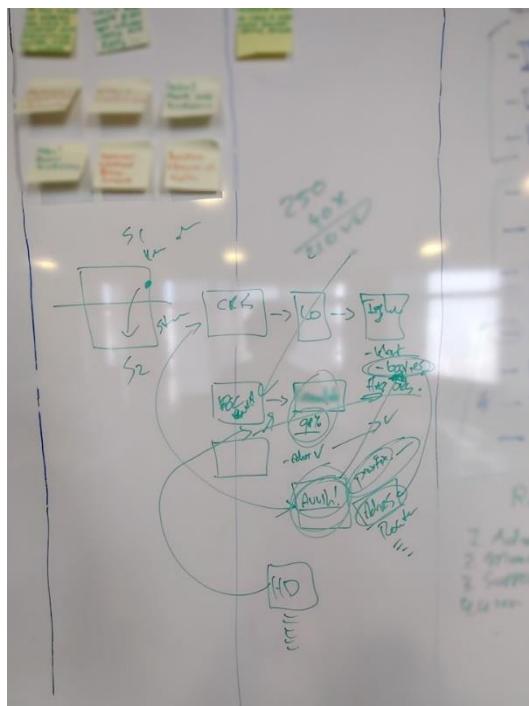
De Kanban methode is niet nieuw voor Helmink maar is net als de dagstart niet een structureel onderdeel van de dagelijkse werkzaamheden, waardoor de motivatie hiervoor daalt.

WAAROM MOET HET C4 MODEL GEBRUIKT WORDEN?

Tijdens de onderzoeksperiode hebben veel besprekingen plaatsgevonden waarbij toelichting werd gegeven door middel van een tekening. Deze diagrammen hebben verschillende niveaus van abstractie en representeren relaties maar ook datastromen of afhankelijkheden. *Figuur 8.6 Schema Loader server* en *Figuur 8.7 Schema HABdesk* laten twee diagrammen zien die allebei hetzelfde systeem en level van abstractie visualiseren. Het onderwerp van deze figuren zijn de verschillende *loaders* binnen de *backend* van het HABdesk project met daarbij velden die gebruikt worden om de koppelingen te maken.



Figuur 8.6 Schema Loader server



Figuur 8.7 Schema HABdesk

Deze tekeningen zijn gemaakt tijdens besprekingen tussen het development team en de *product owner* en vinden plaats binnen de werkplek van het development team of één van de conferentiekamers.

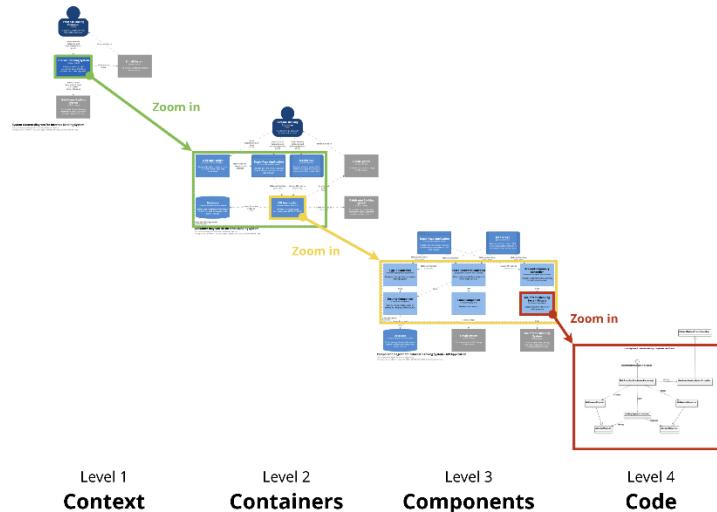
Voor medewerkers die geen verstand hebben van de verschillende systemen die gebruikt worden zijn deze figuren bijna onmogelijk te volgen. De lijnen lopen door elkaar heen, er worden cirkels getekend rond bepaalde elementen zonder duidelijke reden en er is geen toelichting gegeven over wat de elementen in het figuur betekenen binnen project zelf.

Hoewel een *Whiteboard* geen ideale plek is om een gedetailleerd ontwerp te maken is het een belangrijk deel van het ontwerpproces. Een onderzoek uit 2007 genaamd "Let's Go to the Whiteboard: How and Why Software Developers Use Drawings" concludeert uit een onderzoek dat het doel van deze figuren is om de ideeën die besproken worden tijdens een meeting te visualiseren. (Cherubini et al., 2007)

Het niet de bedoeling om deze tekeningen direct te gebruiken als startpunt voor verdere werkzaamheden, toch is dit meestal wel het geval binnen Helmink. Voor extra duidelijkheid wordt er gebruik gemaakt van een diagrammen *tool* genaamd LucidChart, hiermee kunnen verschillende UML-diagrammen worden gemaakt. Binnen de LucidChart omgeving van Helmink zijn er meerdere uitgebreide ontwerpen gemaakt, mede door de uitvoerende partij.

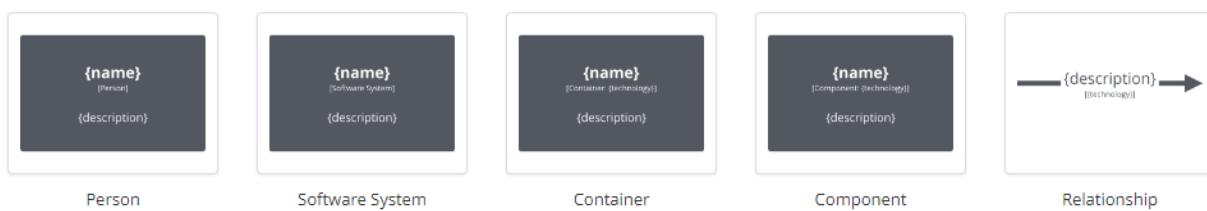
Het probleem wat zich voor doet is dat er niet genoeg duidelijke diagrammen worden gemaakt van het de *backend* en *frontend* van het HABdesk project, dit komt mede omdat er vooral tijd wordt besteed aan visualisaties zoals in *Figuur 8.6 Schema Loader server* en *Figuur 8.7 Schema HABdesk*.

Om dit op te lossen moet er gebruik worden gemaakt van het C4 model en de principes waarop het is gebaseerd. Het C4 model is een techniek waarin er wordt gewerkt met vier verschillende lagen van abstractie binnen hetzelfde project, elke laag zoomt dieper in op een specifiek deel van het systeem. (Brown, z.d.) De vier lagen zijn: Context, Container, Component, Code. In het hoofdstuk Architectuur wordt het C4 model verder uitgewerkt. Zie *Figuur 8.8 – C4 model*



Figuur 8.8 – C4 model

Dit model heeft als doel om context te geven aan de veel gebruikte code diagram, deze heeft vaak weinig context en is voor medewerkers buiten het development team vaak moeilijk te begrijpen. Het model maakt niet alleen gebruik van de vier lagen maar heeft ook een bepaalde notatie die is ontworpen om zoveel mogelijk informatie te geven met zo min mogelijk tekst. In *Figuur 8.9 Notatie C4 model* zijn de vijf onderdelen van een C4 model te zien. Bij elk component wordt er een naam en een type en aan beschrijving ingevuld, dit is niet gebruikelijk bij standaard UML-diagrammen. Het type en de beschrijving helpen bij het onderscheiden van verschillende type onderdelen die gebruikt worden en geven context over het exacte doel van de component.



Figuur 8.9 Notatie C4 model

De hierboven aangegeven notaties van *Figuur 8.9 Notatie C4 model* zijn niet de enige die gebruikt kunnen worden, binnen het C4 model is veel personalisatie mogelijk en kan er per project andere onderdelen worden gebruikt.

Voor besprekingen kan het C4 model ook worden toegepast, door de principes van het model over te nemen is het mogelijk om toch duidelijke figuren te maken op een *Whiteboard* of digitaal bord. Bij het maken van een tekening op het *Whiteboard* of scherm moeten de volgende regels worden aangehouden:

- Zorg ervoor dat alle onderdelen van hetzelfde type dezelfde vorm en grote hebben.
- Verschillende vormen kunnen worden gebruikt om andere type componenten te visualiseren, daarbij is het belangrijk om de hoeveelheid soorten componenten te zo laag mogelijk te houden.
- Gebruik maximaal twee soorten lijnen, dit houdt de verbindingen tussen de onderdelen duidelijk. Het is aangeraden om een dichte en een gestreepte lijn te gebruiken.
- Zorg ervoor dat de tekst binnen de onderdelen leesbaar is, gebruikt desnoods opsommingstekens om concreet te zijn.
- Probeer overlappende lijnen of componenten zo veel mogelijk te beperken.

HET ADVIES VOOR HELMINK

WAT IS ER NODIG OM (DE BACKEND VAN) HET HABDESK PROJECT TE ONTWIKKELEN?

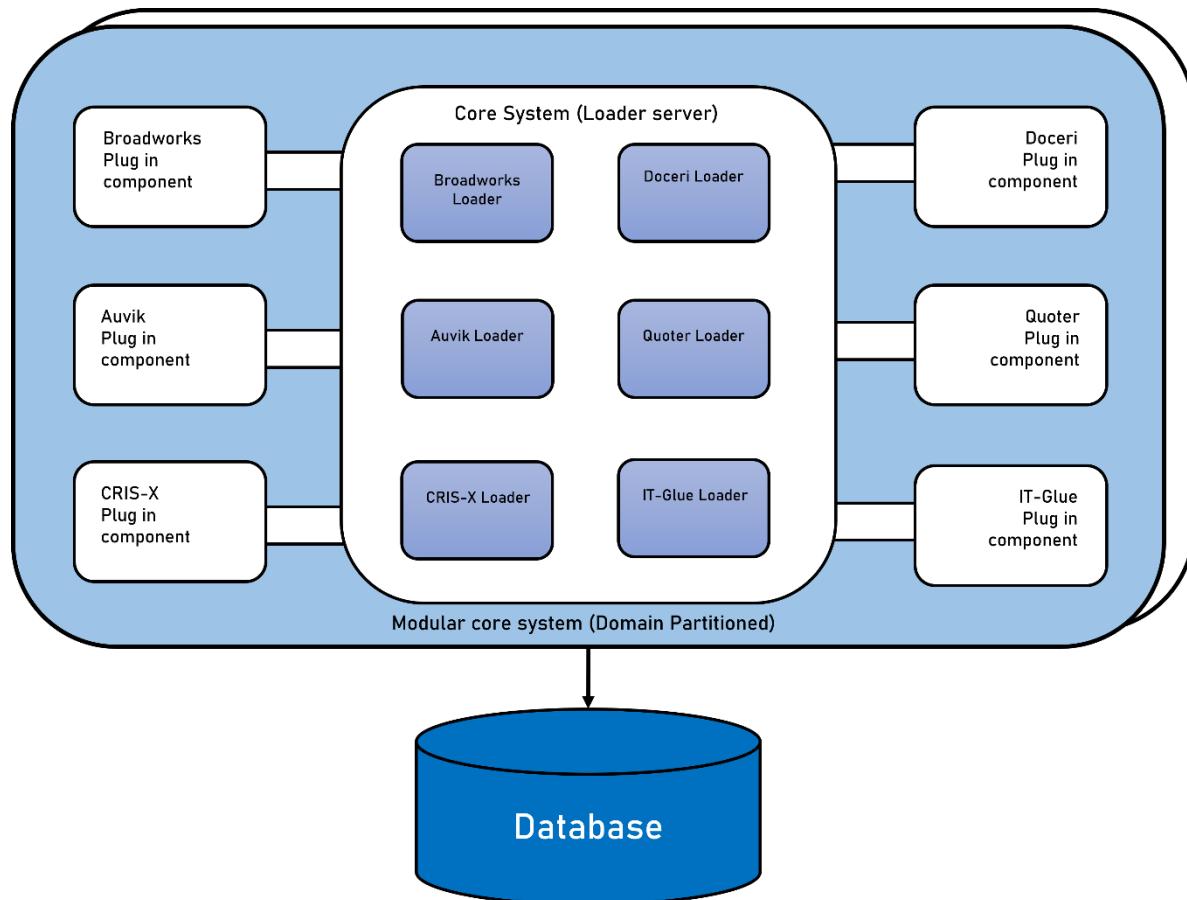
Om de *backend* van het HABdesk project te kunnen ontwikkelen....

9. ARCHITECTUUR

Het hoofdstuk Architectuur beschrijft de huidige structuur van de *backend* van het HABdesk project aan de hand van UML-diagrammen en de opbouw van de code. In de volgende *paragraaf – C4 model* wordt de structuur van het PoC toegelicht door middel van het C4 model dat in het hoofdstuk Advies is beschreven en een *activity diagram*. Het testplan voor het PoC wordt toegelicht in *paragraaf - Tester*.

HUIDIGE ARCHITECTUUR BACKEND HABDESK

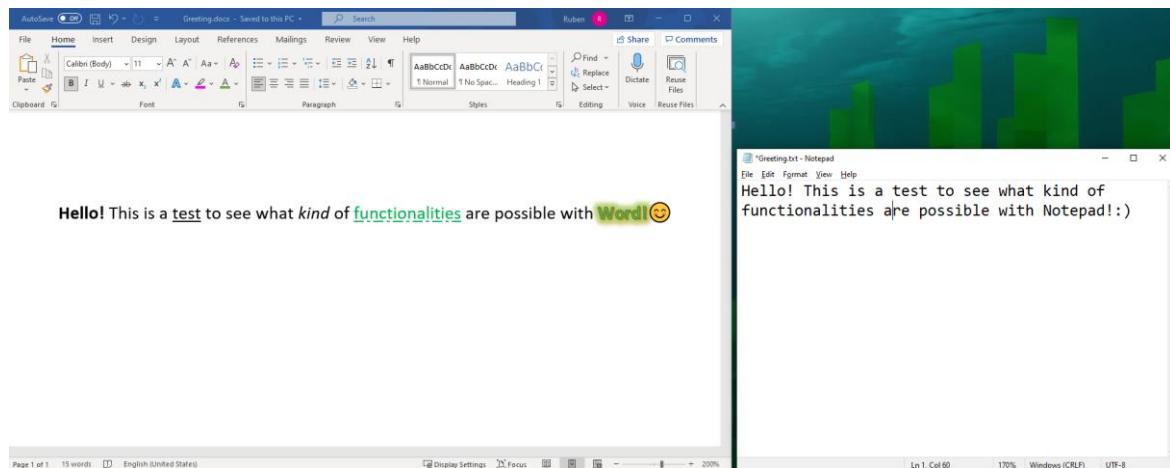
De huidige architectuur wordt toegelicht in het *hoofdstuk 8. Advies*. *Figuur 9.1 Microkernel architectuu Loader server* wordt gebruikt om een abstracte weergave te bieden van de denkwijze waarop de *backend* wordt ontwikkeld. In deze architectuur (genaamd *Microkernel Architecture*) wordt er gebruik gemaakt van een *core systeem* en *plug-in components*, deze zijn zichtbaar in *Figuur 9.1 Microkernel architectuu Loader server*.



Figuur 9.1 Microkernel architectuur Loader server

De beste manier om deze architectuur te beschrijven is het te vergelijken met een bestaand project zoals Word. Microsoft Word is een complexe applicatie vol met allerlei functionaliteiten. Als deze functies worden verwijderd blijft er een applicatie over dat erg lijkt op een kale *text editor* zoals de *notepad* app die wordt meegeleverd bij een nieuwe installatie van Windows. Om het verschil te visualiseren is in *Figuur*

9.2 Word vs Notepad dezelfde tekst geschreven met een de verschillende functies die beschikbaar zijn in beide applicaties.



Figuur 9.2 Word vs Notepad

Deze drastische versimpelde versie van Word kan worden vergeleken met een *core system* binnen de *microkernel* structuur, waarbinnen alleen onderdelen worden ontwikkeld die essentieel zijn voor het functioneren van het systeem. Alle andere mogelijkheden worden los ontwikkeld binnen de *plug-in components* en zijn beschikbaar binnen *core systeem*.

HET CORE SYSTEEM

De *microkernel* architectuur is te herkennen aan het *core system* dat de essentiële onderdelen van het systeem bevat, zonder deze onderdelen kan het systeem de belangrijkste taken niet verrichten.

Het *core systeem* van de *backend* van het HABdesk project is opgedeeld op basis van het domein, dit betekent dat de onderdelen van de code worden gegroepeerd op basis van het bedrijfsdomein waarin deze zich bevinden. Als voorbeeld, alle code die te maken heeft met een koppeling met Auvik wordt samen opgeslagen onder een gedeelde naam. De term 'groeperen' betekent niet direct de code van een onderdeel ook fysiek in dezelfde map aanwezig is. Binnen de code van Helmink wordt er gebruik gemaakt van het *Model View Controller (MVC)* patroon zoals beschreven in het hoofdstuk 5. *Huidige situatie*, dit is een manier om de code op te delen in drie delen, Het *model* die alle definities bepaald en data kan bevatten, de *view* die de *frontend* van de applicatie beheert en de *controller*, die de grootste hoeveelheid van de logica uitvoert.

Het MVC-patroon verdeelt de code op basis van de techniek in plaats van het domein, toch kan de architectuur worden beschreven als *domain partitioned* (verdeeld op domeinniveau) omdat het systeem is opgebouwd meerdere instanties van het MVC-model, deze kunnen onafhankelijk van elkaar worden aangepast.

Mappenstructuur

Het *core* systeem van de *backend* van het HABdesk project bestaat uit de project code van Laravel. Het *framework* Laravel dat de basis vormt voor het gehele project bevat veel elementen om een webapplicatie te kunnen bouwen. Zonder deze onderdelen is het niet mogelijk om de webapplicatie te starten. De belangrijkste componenten van een Laravel project zijn (Laravel, z.d.):

De App Directory

Deze *directory* ook wel *map* genoemd bevat kern code van het project, alle code om een webapplicatie op te zetten bevindt zich in deze map, de meest gebruikte sub mappen zijn:

- De *Http Directory*

Deze sub map bevat alle nodige elementen om inkomende verzoeken voor binnen de applicatie worden hier beheert.

- De *Jobs Directory*

Deze map bestaat wordt aangemaakt zodra er een *job* aangemaakt is. *Jobs* zijn taken die uitgevoerd worden door middel van een *queue*. Een *queue* is een rij waarin de verschillende taken van het project in de achtergrond uitgevoerd kunnen worden. Net als in een normale rij zijn er verschillende mogelijkheden. Zo kunnen sommige taken synchroon (één na de ander) worden afgehandeld of een asynchrone rij (*asynchronous queue*) wordt gebruikt waarbij taken niet hoeven te wachten op elkaar.

- De *Models Directory*

Zoals de naam suggereert bevat de *models* map alle *MVC-models* die gebruikt worden. Zoals beschreven in het hoofdstuk *Huidige situatie* bevatten en *models* de definities en data die door het gehele project gebruikt worden.

- De *Console Directory*

In deze map zitten alle zelf opgebouwde *commands*. Dit zijn de onderdelen die de eerdergenoemde *jobs* uitvoert door ze in de *queue* te plaatsen, dit wordt ook wel *dispatch* genoemd.

- De *Routes Directory*

In de *routes* map binnen de Laravel app beheert alle routes die binnen de applicatie gebruikt worden, deze routes verwijzen vaak naar *controllers* die logica van een bepaald domein beheren. Meerdere *route* bestanden zijn beschikbaar maar de meest gebruikte is de *web.php*. Binnen dit bestand worden de routes van de verschillende *plug-in components* gedefinieerd.

De Database Directory

Hier bevinden zich alle commando's die de database structuur beïnvloeden. De bestanden die dit uitvoeren worden *migrations* genoemd. Door migraties te gebruiken is het mogelijk om aanpassingen binnen de database terug te draaien, dit wordt *rollback* genoemd.

De Public Directory

De *public directory* is het startpunt van alle communicatie met de applicatie en verzoeken vanuit buitenaf. Als het project start is de *index.php* pagina de eerste die wordt uitgevoerd.

De Resources Directory

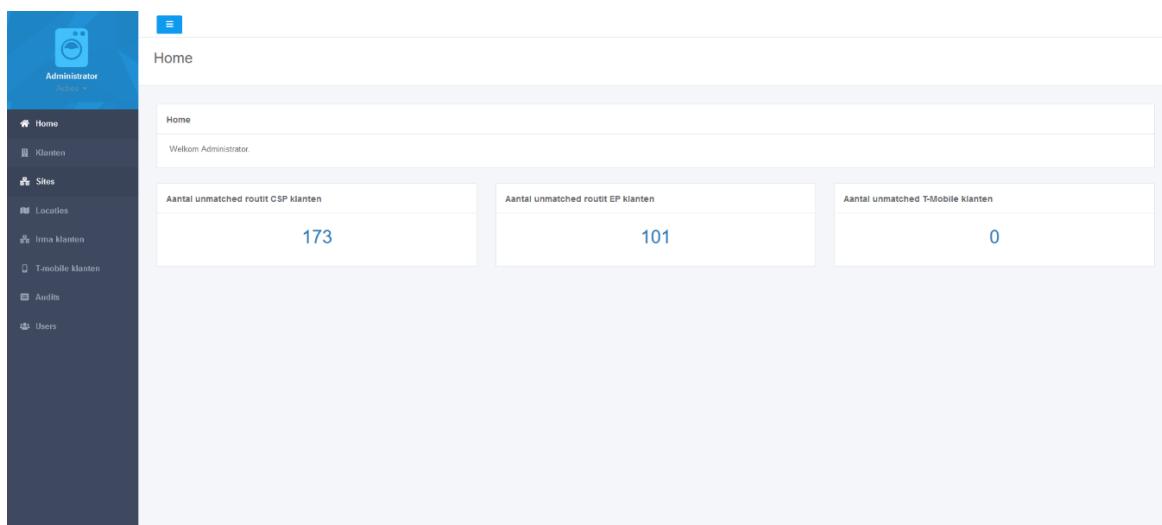
Bevat alle *views*. Dit zijn de elementen die samen de *frontend* van het systeem representeren. De bestanden zijn geschreven in *blade*. *Blade* is een manier om data simpel te kunnen verwerken binnen HTML-code, het wordt standaard meegeleverd met Laravel en werkt daarom goed samen. Als de *Blade* pagina's naar de browser worden gestuurd vindt er een vertaling plaats naar JavaScript.

De Tests Directory

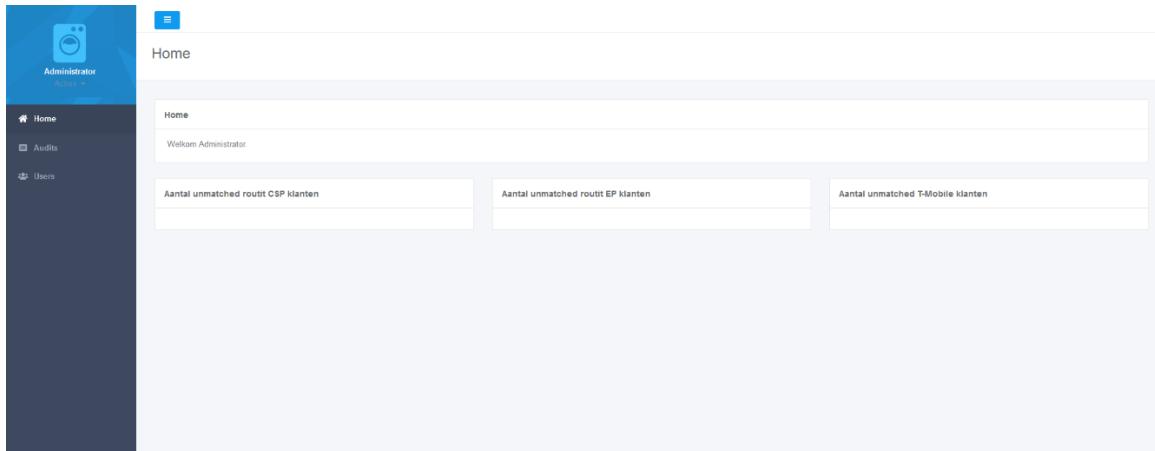
Zoals de naam suggereert bevat de *tests* map de verschillende geautomatiseerde tests die uitgevoerd kunnen worden. Deze test bestanden kunnen bestaan uit verschillende test *classes* en worden uitgevoerd binnen de *console* van het project.

Loader server core systeem

In *Figuur 9.3 Hoofdpagina Loader server* en *Figuur 9.4 Hoofdpagina Loader server* hieronder te zien hoe de applicatie zou functioneren als alle *plug-in components* weg worden gehaald. Links is de huidige applicatie en rechts is alleen het kern systeem zichtbaar. Een aantal aspecten vallen op:



Figuur 9.3 Hoofdpagina Loader server

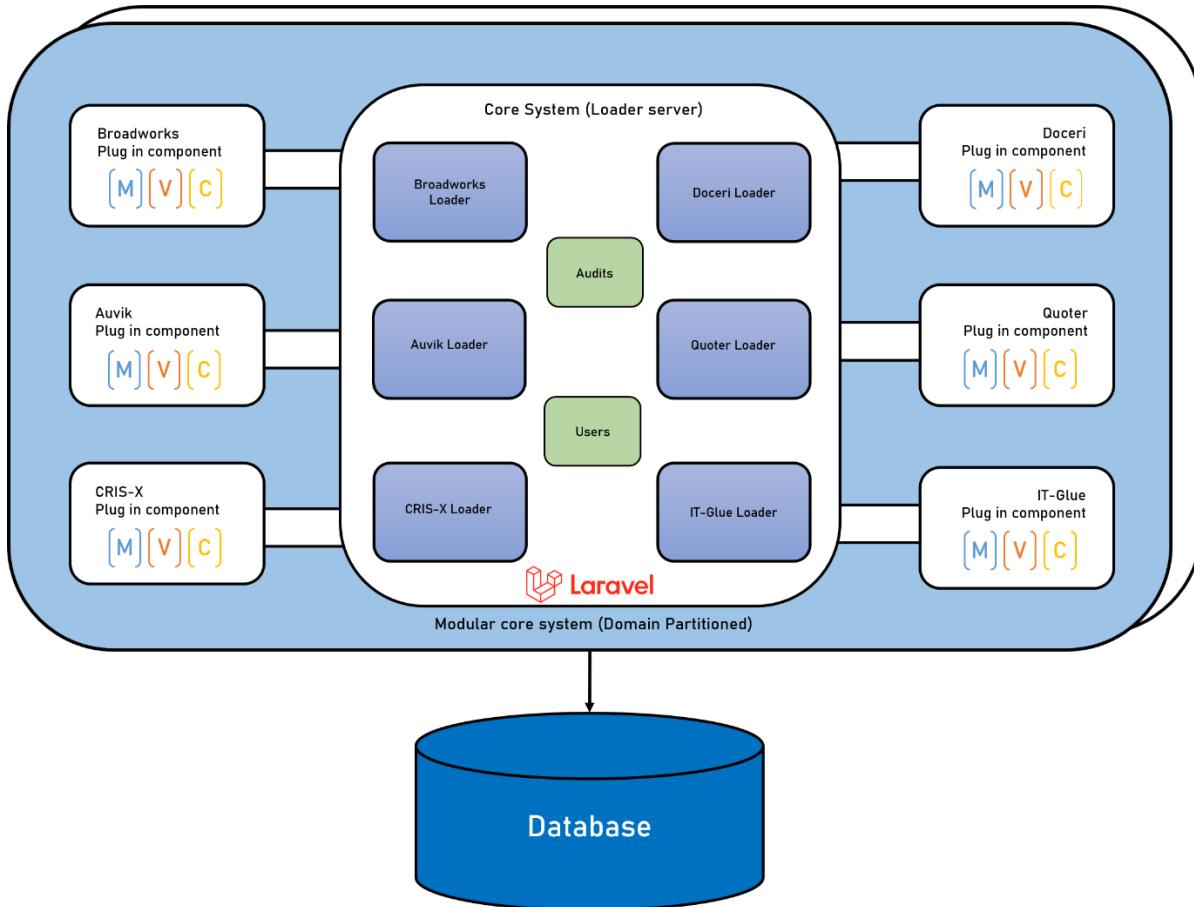


Figuur 9.4 Core system Loader server

- Technisch gezien zou de navigatie links niet aangepast worden, omdat deze niet direct afhankelijk is van de *plug-in components*. Om aan te geven over welke componenten het gaat zijn ze verwijderd uit de navigatie
- De data over de hoeveelheid klanten die nog niet zijn gekoppeld is leeg, deze data komt normaal gesproken vanuit de bijbehorende component
- De algemene navigatie en een aantal monitoringscomponenten zoals de *users* en de *audits* zijn wel aanwezig. Mocht dit wenselijk zijn kunnen deze ook worden verwijderd uit het kern systeem, dit is geen verplichting.

PLUG-IN COMPONENTS

In *Figuur 9.5 Extra detail* is te zien hoe het systeem is opgedeeld als de code wordt geanalyseerd. De groene blokken zijn de componenten die niet direct te maken hebben met de *plug-in components*. Bij elk component is het MVC-model toegevoegd, wat aantoont dat ieder een onafhankelijke instantie heeft van de *models*, *views* en *controllers*.



Figuur 9.5 Extra detail

In de *paragraaf Het core systeem* wordt duidelijk gemaakt welke onderdelen van het project worden geklassificeerd als de kern. De verschillende *plug-in components* worden geïntegreerd met het kern systeem en bieden gespecialiseerde functionaliteit. De benaming van deze *components* binnen de Loader server is 'De {bron} Loader', hierbij refereert de bron naar een service of extern pakket dat data aan kan bieden. Een groep van deze *components* worden 'Loaders' genoemd.

De Auvik API Loader

De loader die in het onderzoek wordt gerealiseerd heeft de officiële naam 'De Auvik API Loader', dit referent naar de officiële API die wordt gebruikt als communicatiemiddel tussen de Loader server en Auvik. Het MVC-model van de Auvik API Loader bestaat uit één *model*, twee *views* en twee *controllers*. Naast deze onderdelen zijn er een aantal *jobs* en *commands* voor het beheren van de functionaliteiten, binnen het hoofdstuk Proof of Concept worden deze onderdelen nader uitgewerkt.

Model: AuvikTenant.php

Het *model* bestaat uit op maat gemaakte definities die gebruikt worden binnen de rest van de applicatie. Deze definities worden ook wel *data logic* of *data logica*, het doel van het model is om de ruwe ongezuiverde data te verwerken zodat het kan worden toegepast binnen de applicatie. (Hernandez, 2021)

Binnen de code wordt *eloquent* gebruikt, dit is een Object-Relational Mapper (ORM). Het doel van een ORM is om de interacties tussen de applicatie en de database te versimpelen (Laravel, z.d.), dit gebeurt door de soms complexe SQL-commando's samen te voegen tot simpele functies die aangeroepen worden. Een verdere uitleg over ORM bevindt zich in het hoofdstuk 5. Huidige situatie. In *9.6 AuvikTenant Model* is een voorbeeld uit de *AuvikTenant model* weer gegeven.

```

78  /**
79   * Get a Collection of all clients assigned to this model.
80   *
81   * @return \Illuminate\Database\Eloquent\Relations\HasMany
82   */
83  public function clients()
84  {
85      return $this->children()->where('type', '=', 'client');
86  }
87

```

Figuur 9.6 AuvikTenant Model

In het voorbeeld is te zien dat de functie *clients* data uit de database ophaalt met een bepaalde conditie. Binnen de code is het mogelijk om op een object met het juiste model de functie op te roepen, waarnaar de data verder gebruikt kan worden.

Views: index.blade.php en create.blade.php

Voor de *views* zijn er twee bestanden waarin de data vanuit de Auvik API direct wordt gebruikt, dit zijn de *index.blade.php* en de *create.blade.php*. Verder wordt de data gekoppeld aan de klantgegevens, dit is verder toegelicht in het hoofdstuk *Implementatie*.

De code maakt gebruik van *blade*, een *templating engine*. Dit is een techniek waarbij onder andere data vanuit een *model* wordt samengevoegd binnen een *view*. (*What Is a Template Engine?*, z.d.) Dit betekent dat het eenvoudig is om de data vanuit een *model* op de webpagina weer te geven.

```

~~~
29
30    <tbody>
31        @foreach ($auvikTenants as $auvikTenant)
32            <tr>
33                <td>{{ $auvikTenant->domain_prefix }}</td>
34                <td>{{ $auvikTenant->type }}</td>
35                <td>{{ $auvikTenant->parent_id }}</td>
36                <td>{{ $auvikTenant->updated_at }}</td>
37            </tr>
38        @endforeach
</tbody>

```

Figuur 9.7 Index.blade foreach loop

Figuur 9.8 Auvik Tenant pagina

In *Figuur 9.7- Index.blade foreach loop* hierboven is een deel van de *index.blade.php* zichtbaar, binnen de *foreach* loop wordt een JSON-object opgedeeld in verschillende onderdelen die worden getoond in een tabel op de webpagina. De data binnen *\$auvikTenants* komt vanuit de *controller* die de *view* aanroeft zie *Figuur 9.9 AuvikTenantController*. Het resultaat van de code is zichtbaar in *Figuur 9.8 Auvik Tenant pagina*.

De *create.blade.php* maakt ook gebruik van een *foreach* loop om de verschillende netwerken (binnen Auvik *Tenants*) te verdelen over de webpagina.

Controllers: `AuvikTenantController.php` en `AuvikTestController.php`

De controller wordt gezien als het brein van de applicatie. Een *controller* verbindt de *model* en de *views* door data tussen de twee onderdelen te overzien. (MVC - MDN Web Docs Glossary: Definitions of Web-Related Terms | MDN, 2022) Afhankelijk van de functionaliteiten binnen de applicatie kan een *controller* veel of weinig taken hebben. Binnen de Auvik API Loader is de *controller* gebruikt om de *model* en de *view* te verbinden. In *Figuur 9.9 AuvikTenantController* wordt eerst de data opgehaald aan de hand van de *model* *AuvikTenant*. De *view* wordt daarna opgesteld met de data vanuit *auvikTenants*.

```
11 <  /**
12  * Display a listing of the resource.
13  *
14  * @return \Illuminate\Http\Response
15  */
16 public function index()
17 {
18     // Get all tenants.
19     $auvikTenants = AuvikTenant::get();
20
21     // Return the index page.
22     return view('auvik.tenants.index', compact('auvikTenants'));
23 }
```

Figuur 9.9 AuvikTenantController

Naast de *controller* hierboven is er ook een *AuvikTestController.php* aanwezig, hierin worden tests uitgevoerd die direct met de rest van de applicatie kunnen communiceren. Code wordt binnen de test *controller* uitgewerkt en later binnen daadwerkelijke applicatie geplaatst.

ARCHITECTUUR KARAKTERISTIEKEN

In *Bijlage 3 - Architectuur* wordt een onderzoek beschreven over software architectuur, één van de onderdelen die wordt beschreven is de term *architecture characteristics* (Ook wel Architectuur karakteristieken). De karakteristieken van een architectuur beschrijven de succesfactoren die niet direct een functionaliteit beschrijven. De termen die ook vaak worden gebruikt zijn *non-functional requirements* of *Quality attributes*.

Eén van de taken van een software architect is om de aspecten van het systeem te definiëren die niet direct te maken hebben met de bedrijfseisen. Het opstellen van deze karakteristieken onderscheidt de architectuur van het ontwerp en de code zelf. De karakteristieken worden opgesteld aan de hand van de wensen van de *product owner* en *stakeholders*.

Het verzamelen van de zorgen van de *stakeholders* lijkt op een relatief eenvoudige taak, de complexiteit ligt niet bij de wensen zelf maar bij het definiëren van een definitieve lijst waar alle *stakeholders* mee instemmen. Het doel is om een balans te vinden tussen de hoeveelheid karakteristieken en de waarde die elke karakteristiek toevoegt aan de kwaliteit van het product. Hoe meer *characteristics* er worden toegevoegd hoe hoger de complexiteit van het systeem is, dit is onwenselijk. (Richards & Ford, 2020)

Vanuit observaties en een interview met de directeur zijn een aantal karakteristieken opgesteld die de behoeftes van de *backend* van het HABdesk project beschrijven.

FUNCTIONELE VOLLEDIGHEID (FUNCTIONAL COMPLETENESS)

Het product moet alle eisen van de *stakeholders* behartigen. Het systeem is afhankelijk van de juiste data en logica om een impact te hebben op de gebruikers ervan. Omdat het systeem verschillende services koppelt kan het niet zijn dat data wordt gemanipuleerd op de manier die niet vooraf is gedefinieerd. Het updaten van de systeemeisen zijn essentieel.

SCHAALBAARHEID (SCALABILITY)

De schaalbaarheid van het systeem is een karakteristiek wat voortkomt uit de onduidelijkheid die Helmink heeft over de hoeveelheid gebruikers van het systeem. Uit verschillende observaties en gesprekken is de visie van Helmink bepaald, deze visie is toegelicht in het *hoofdstuk 5. Huidige situatie* en gevisualiseerd in het *Hosin Kanri* model dat in *Bijlage 2- Hosin Kanri* is uitgewerkt.

Het HABdesk product (inclusief de *backend*) moet uiteindelijk kunnen worden verkocht aan externe partijen. Om dit te realiseren moet het product de verschillende hoeveelheden gebruikers aankunnen. Bij de schaalbaarheid van de *backend* heeft gaan het om de hoeveelheid koppelingen die mogelijk ontwikkeld kunnen worden en de bijbehorende data die daarbij komt kijken. Het moet mogelijk zijn om meer of minder systemen te integreren zonder dat het kern systeem fundamenteel aangepast moet worden.

HERBRUIKBAARHEID (REUSABILITY)

Veel van de componenten die zich in de huidige configuratie bevinden bevatten dezelfde onderdelen, met name de MVC-structuur en soortgelijke code conventies. Om het ontwikkelproces en de uiteindelijke schaalbaarheid van de *backend* te bevorderen is er een niveau van herbruikbaarheid vereist. Het mag niet voorkomen dat elke nieuwe koppeling volledig vanaf niks ontwikkeld moet worden.

VERVANGBAARHEID (REPLACEABILITY)

Een niveau van vervangbaarheid valt samen met de schaalbaarheid. Een schaalbaar systeem heeft van nature een hoge vervangbaarheid, sinds de verschillende componenten los van elkaar worden geïmplementeerd. Een voorbeeld van een schaalbare structuur is vrijwel elk van de *distributed* architecturen. Door de separatie van het systeem is het eenvoudiger om delen te vervangen met minimale consequenties. Het mag niet zijn dat een vervanging van één koppeling het gehele systeem buiten werking stelt.

FOUTTOLERANTIE (FAULT TOLERANCE)

Omdat er met zoveel verschillende soorten bronnen communiceren met het systeem zijn fouten onvermijdelijk. Het systeem moet deze fouten kunnen incasseren en opslaan voor nader onderzoek. Het systeem mag bij onzuivere data niet buiten werking worden gesteld.

HERSTELBAARHEID (RECOVERABILITY)

Bij het geval van een storing moet het systeem zo spoedig mogelijk weer in productie zijn. Zonder het systeem kan er geen data worden uitgewisseld en is er het risico dat data verloren gaat. Het mag niet gebeuren dat bij een crash alle data binnen het systeem niet is opgeslagen.

TRANSITIE ARCHITECTUUR

Als er binnen een organisatie naar een andere structuur wordt gezocht is het niet zomaar mogelijk om de huidige architectuur om te bouwen, dit zijn complexe systemen die gigantische hoeveelheden data verwerken. Om organisaties te helpen bij het maken van architecturale keuzes is de TOGAF (staat voor The Open Group Architecture Framework) methode ontwikkeld. Dit is een open standaard waarin technieken en tips over het beheren van organisatiearchitectuur. (Done, 2021)

De TOGAF valt voor het grootste gedeelte buiten de scope van dit onderzoek. Wel wordt er gebruik gemaakt van de drie algemene stappen die de transitie naar een nieuwe architectuur beschrijven.

TOGAF noemt drie soorten architecturen die ontwikkeld worden om uiteindelijk de zogenoemde *vision* (ook *target*) *architecture* te bereiken. (Done, 2021)

- ***Baseline* architectuur**

De *Baseline* architectuur is de structuur die op het moment van het onderzoek in productie wordt gebruikt.

- ***Transition* architectuur**

Dit is de manier waarop een transitie naar een nieuwe architectuur in gang kan worden gezet. Een transitie architectuur is een aangepaste versie van de *baseline* en bevat een deel van de ideeen en wensen die bij de *vision* architecture worden beschreven.

Om deze wensen te bereiken moeten er op korte termijn meerdere *transition* architecturen worden ontwikkeld die steeds meer richting de uiteindelijke visie werken. De organisatie past zich met kleine stappen aan aan de hand van de transitie architecturen.

- ***Target/ Vision* architectuur**

Dit is het uiteindelijke doel, maar kan tijdens het proces worden aangepast. TOGAF beschrijft dat de visie die aan het begin van de transitie is ingezet ook evalueert om de nieuwe technische en organisatorische veranderingen te adapteren. Het is onwaarschijnlijk dat de originele *vision* architectuur ooit wordt bereikt.

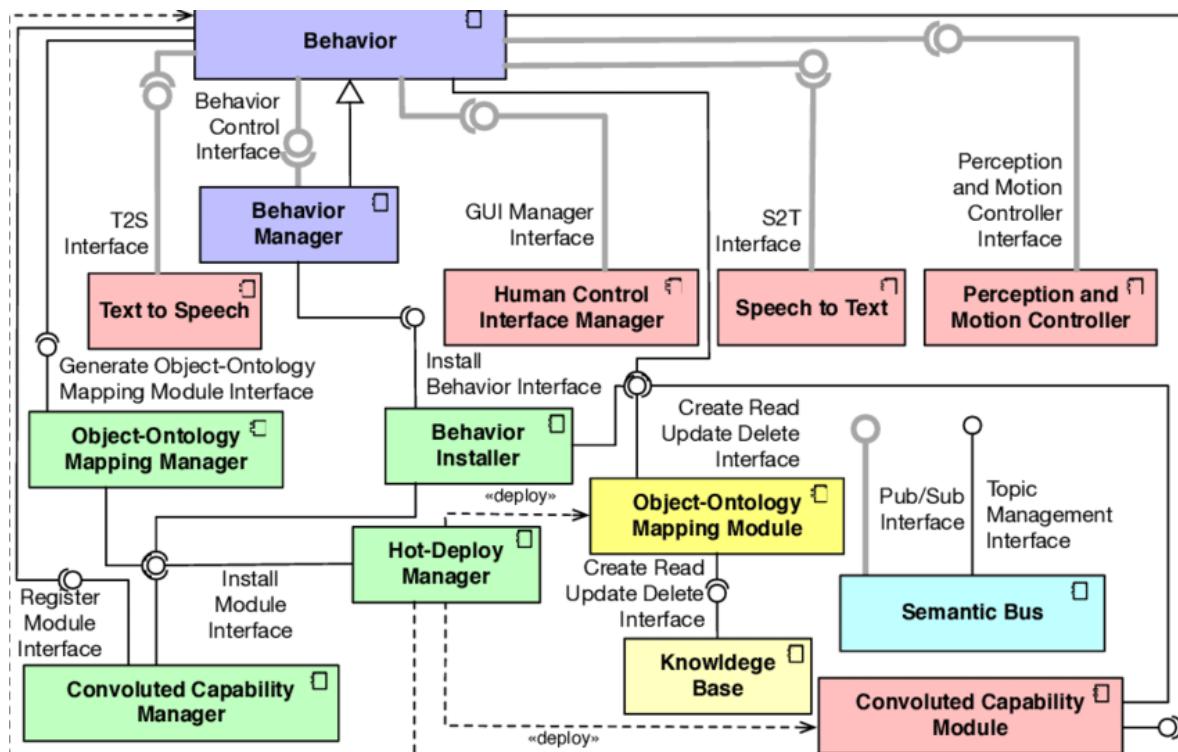
Dit proces hierboven is constant in flux en vindt plaats zolang de organisatie door groeit. Het onderzoek is gericht op het implementeren van de *baseline* architectuur maar kijkt ook naar de toekomst. TOGAF toont aan dat een architectuur nooit stil staat en constant in een staat van transitie bevindt, met of zonder vastgestelde visie.

C4 MODEL

Het C4 model wordt in *Hoofdstuk 8. Advies* toegelicht aan de hand van een voorbeeld waarbij de tekeningen op een *Whiteboard* erg onduidelijk overkomen omdat er geen standaard wordt gebruikt. Het advies is om de principes van het C4 model toe te passen binnen het formuleren van projecten.

HET DOEL VAN C4

Het model is ontstaan door dezelfde frustratie die beschreven is in het *Hoofdstuk 8. Advies* Simon Brown heeft rond 2006 de principes van het C4 model opgesteld door te kijken naar de huidige implementatie van UML-diagrammen. In die tijd was het gebruik van UML bekritiseert door de abstracte aard van vele methodes die systemen onduidelijk weergaven, hoewel een ontwerptechnisch gezien correct kan zijn is dit geen garantie dat het ook een toevoeging biedt aan het project. Het *Figuur 9.10 Voorbeeld UML diagram* hieronder is een voorbeeld van een UML-diagram die een systeem wellicht correct beschrijft maar verdere toelichting of uitleg biedt.



Figuur 9.10 Voorbeeld UML diagram (Asprino et al., 2022)

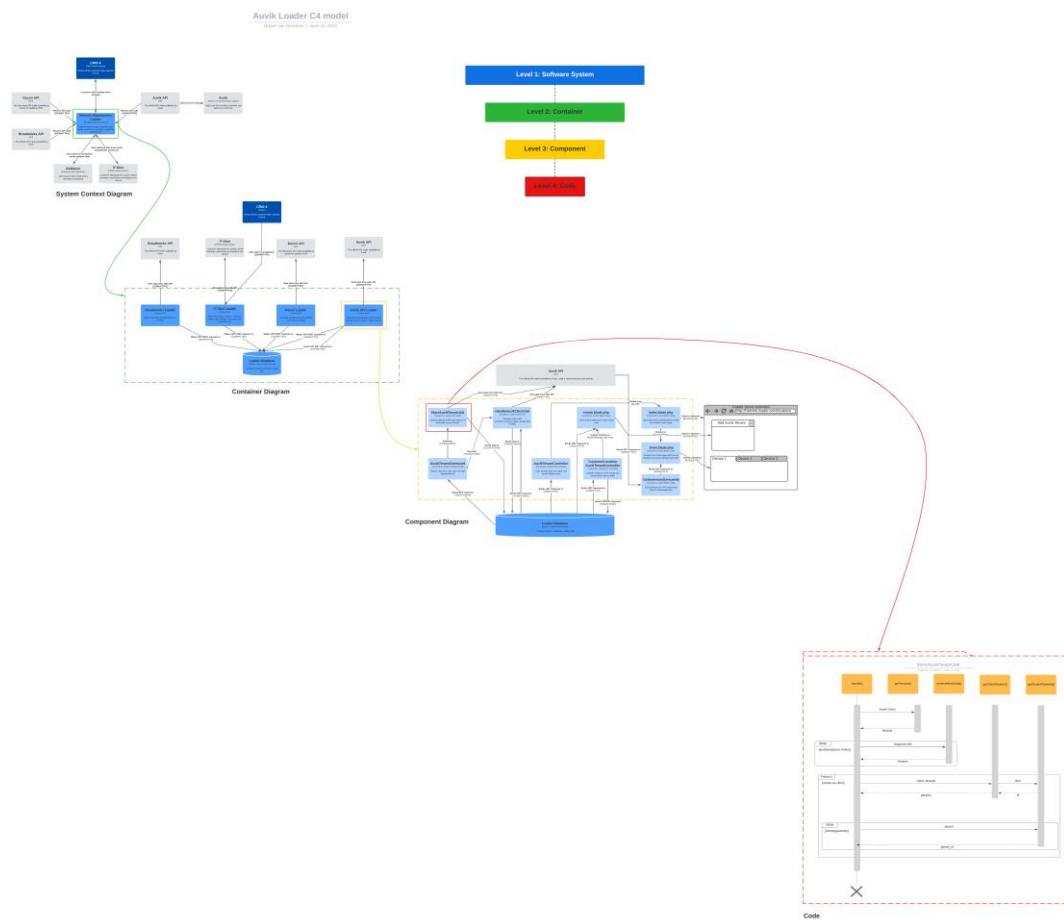
De architect die dit ontwerp heeft opgesteld snapt wat alle kleuren en lijnen betekenen, maar zonder een duidelijke notatie is het moeilijk voor andere medewerkers om te achterhalen wat voor proces er wordt geschatst.

Het doel van het C4 model is tweeledig:

- Het C4 model hoopt het ontwerp van een systeem leesbaarder en overzichtelijker te maken.
- Het C4 model wil het gat tussen een ontwerp en de code zo klein mogelijk houden, dit voegt meer waarde toe voor teamleden om het ontwerp te begrijpen. (Brown, z.d.)

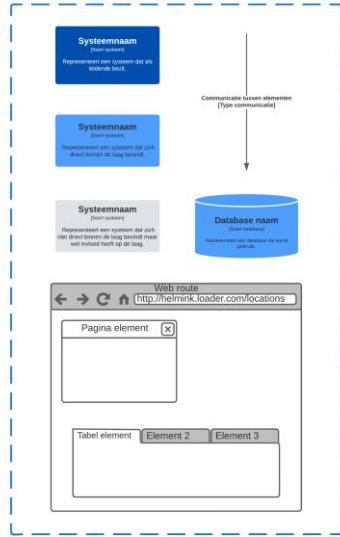
HET MODEL

Voor het *Proof of Concept* is het C4 model gebruikt om de context rondom het systeem duidelijk te definiëren. De Auvik API Loader is één onderdeel, dit model laat goed zien op welke manier de Loader server is ingericht. Het model is opgesteld met het diagrammen tool LucidChart. Het volledige model is zichtbaar in *Figuur 9.11 C4 Model Auvik Loader*. De vier lagen worden nader toegelicht.



Figuur 9.11 C4 Model Auvik Loader

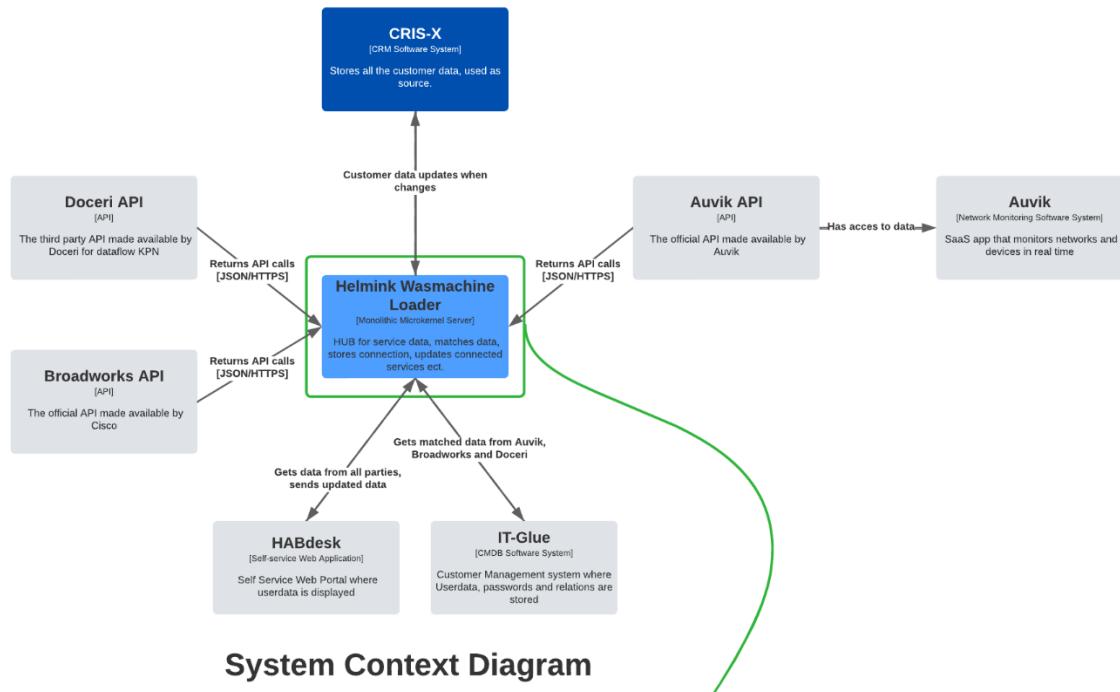
In *Figuur 9.12 C4 Notaties* is te zien welke notaties zijn gebruikt tijdens het bouwen van het model. Er wordt een onderscheid gemaakt tussen donkerblauw en twee lichte varianten van blauw. Donkerblauw represeneert het systeem dat wordt gezien als leidend, dit houdt is dat dit systeem wordt gebruikt om alle andere systemen te controleren. Binnen de *backend* van het HABdesk project is dit het CRM-systeem CRIS-X. De andere kleuren blauw representeren de systemen die zich binnen een bepaalde laag bevinden. De grijze blokken zijn de systemen die wel invloed hebben op een laag maar er geen direct deel van uitmaken.



Figuur 9.12 C4 Notaties

NIVEAU 1: SYSTEM CONTEXT DIAGRAM

Het eerste niveau is de *context diagram*. Het doel van dit niveau is om een startpunt te zijn voor het ontwerp. (Brown, z.d.) Het systeem dat wordt ontworpen is binnen *Figuur 9.13 C4 Context diagram* omringt met groen, de andere systemen daaromheen representeren de externe services en applicaties die communiceren met het zogenoemde *target systeem*.



Figuur 9.13 C4 Context diagram

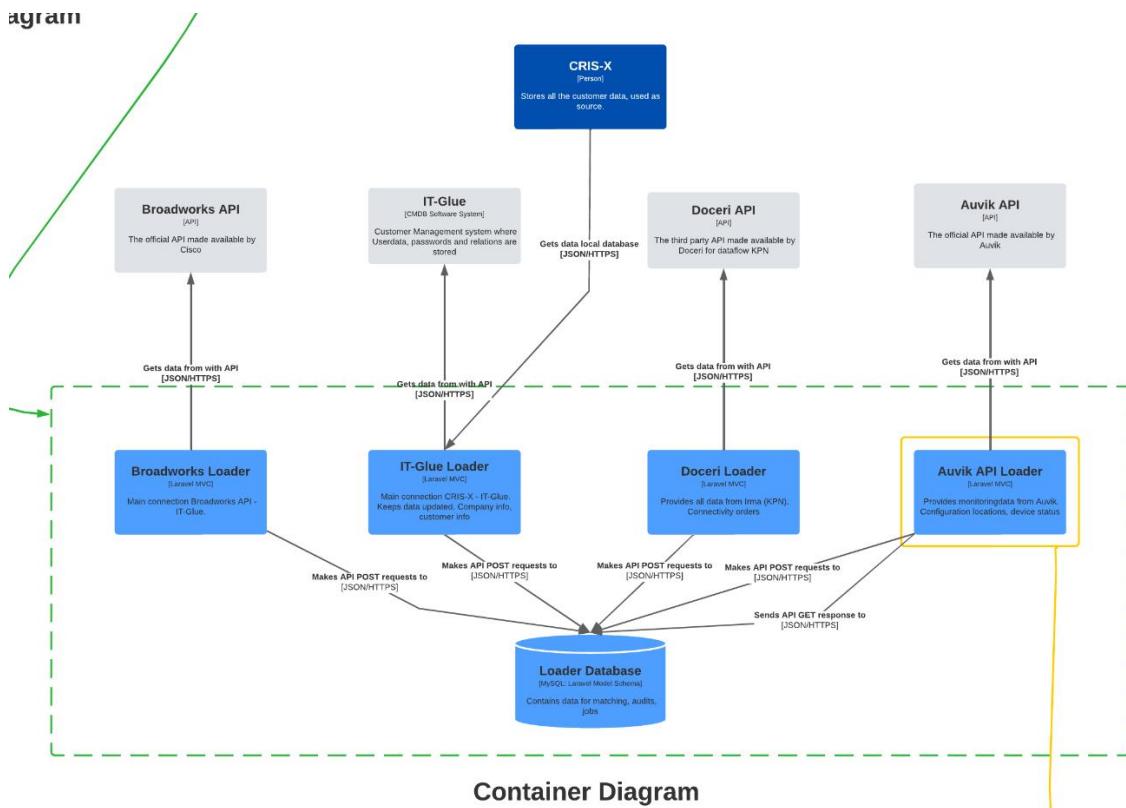
Het *target systeem* is de 'Helmink Wasmachine Loader', dit is de interne naam voor de Loader server en verwerkt de verschillende databronnen die nodig zijn voor HABdesk en andere systemen.

In het bovenstaande figuur is te zien dat HABdesk niet het enige systeem is dat data krijgt vanuit de 'Wasmachine', het CMDB-systeem IT-Glue maakt ook gebruik van de Loader server en krijgt data van onder andere Auvik. Het valt op dat er naast de Loader server geen ander systeem een koppeling heeft met zowel HABdesk als IT-Glue, de reden hiervoor is om de logica te forceren binnen de Loader server. Om alle logica te kunnen beheren is het wenselijk om alle logica binnen één systeem te houden.

NIVEAU 2: CONTAINER DIAGRAM

De groene lijn vouwt de Loader Wasmachine open en op het volgende niveau wordt dit systeem toegelicht. De term *container* kan voor verwarring zorgen sinds het in andere contexten ook wordt gebruikt, met name binnen *docker*, een programma waarbij de term *containers* wordt gebruikt om aan te geven dat een systeem alle nodige referenties en code bevat om ingezet te worden. Binnen het C4 model

refereert een *container* naar een soort systeem dat binnen verschillende omgevingen kan worden gebruikt. In *Figuur 9.14 C4 Container diagram* wordt de *conainter diagram* van de Loader server weer gegeven. (Brown, z.d.)



Figuur 9.14 C4 Container diagram

In de *container* van de Loader server zijn de verschillende Loaders te zien die ontwikkeld zijn, allemaal communiceren ze met één database. De structuur van de Loaders lijkt veel op elkaar en bestaat uit drie stappen:

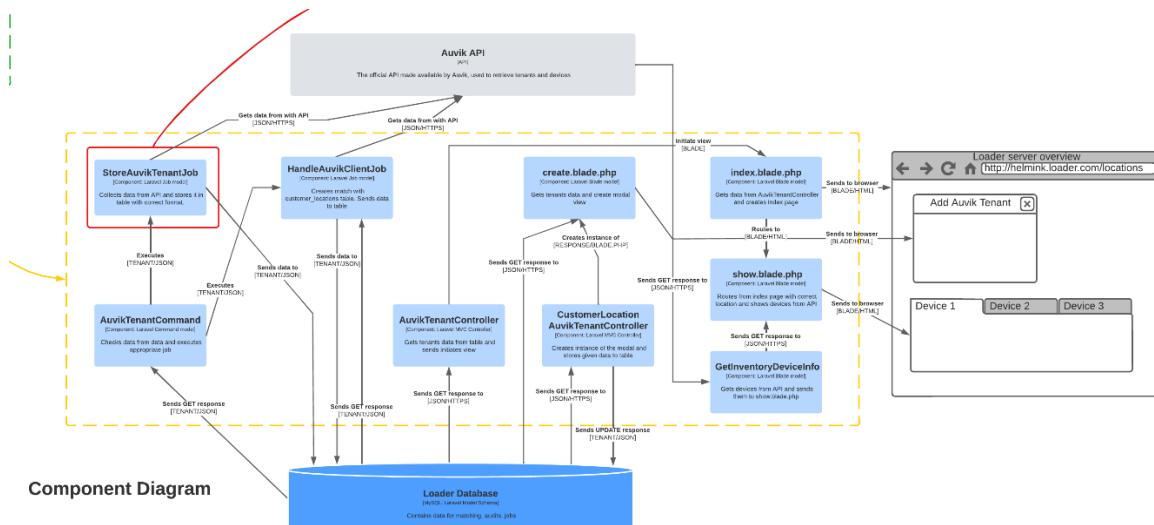
- De eerste stap is het verkrijgen van data vanuit een extern pakket dat benaderbaar is via een API of ander communicatiemiddel. In het geval van de Auvik API Loader is dit de API van Auvik zelf.
- De tweede stap is om deze data binnen de Loaders (in *Figuur 9.14 C4 Container diagram* aangegeven met lichtblauw) te verwerken, hiervoor kan extra data nodig zijn vanuit de database.
- De laatste stap is om de verwerkte en/of verrijkte data op te slaan binnen de database.

Loader database

In *Figuur 9.14 C4 Container diagram* is te zien hoe alle mogelijke Loaders die worden ontwikkeld met dezelfde database communiceren, hiernaast bevat de database de verwerkte data die binnen systemen zoals HABdesk gebruikt worden. Om schaalbaarheid en herbruikbaarheid te bevorderen is de database opgedeeld per Loader, hierdoor heeft elke nieuwe Loader aparte tabellen.

NIVEAU 3: COMPONENT DIAGRAM

Het derde niveau is een verder vergrootglas op een specifieke *container*, de onderdelen binnen de *container* worden *components* genoemd. Het doel van dit niveau is om op een technisch level dieper te kijken naar de onderdelen die binnen een *container* gebruikt worden. Niet elk ontwerp heeft dit niveau nodig, omdat de extra context binnen het niveau verschillend is per project. (Brown, z.d.)



Figuur 9.15 C4 Component diagram

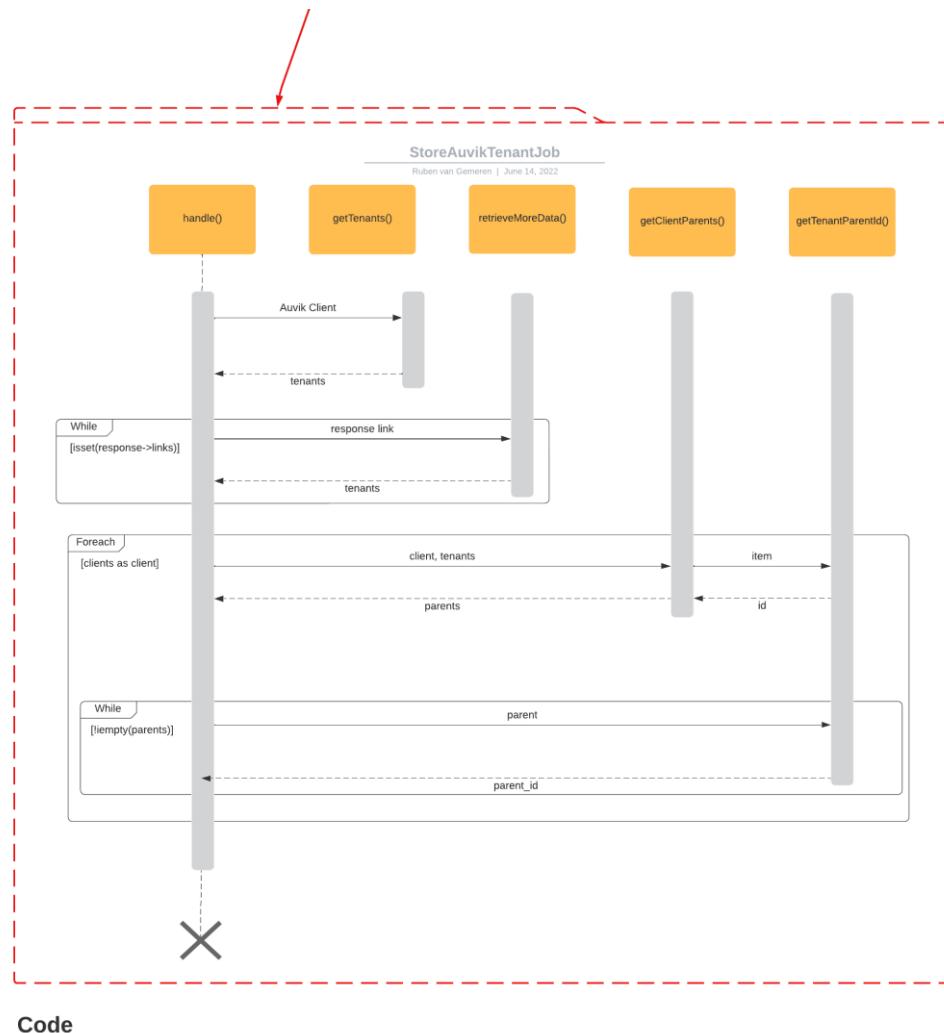
Figuur 9.15 C4 Component diagram laat de componenten zien die binnen de Auvik API Loader zijn ontwikkeld, hierbij wordt een onderscheid gemaakt tussen *backend* en *frontend components*. De *backend* is links weergegeven en bevat de twee *controllers* toegelicht in *paragraaf - het core systeem*. Deze *controllers* communiceren met de API en verwerken de data voor de database. Rechts is de *frontend* uitgewerkt met de *views* toegelicht in *paragraaf - het core systeem*. Deze *views* vormen samen de webpagina's en tabellen die binnen de Loader server zichtbaar zijn, deze pagina's zijn toegelicht in *het hoofdstuk 10. Implementatie*

Net als in niveau 2 is te zien dat vrijwel alle communicatie tussen de componenten via de database verloopt. Wel wordt er gebruik gemaakt van een paar koppelingscomponenten, zoals de *AuvikTenantCommand* die de *jobs* aanstuurt. Deze hebben als doel om data uit de database eerst te vervormen voordat het naar de juiste component wordt gestuurd.

Deze duidelijke afzonderlijkheid bevorderd de vervangbaarheid en herbruikbaarheid van het systeem. Als een deel van de *frontend* moet worden aangepast heeft dit geen impact op de *jobs*, andersom geldt dit ook.

NIVEAU 4: CODE: SEQUENCE DIAGRAM

Het laatste niveau bevat data over één van de *components* binnen het niveau 3. Er is geen vaste structuur die gebruikt moet worden binnen dit niveau. Net als de *component diagram* is het niet verplicht om dit niveau uit te werken. (Brown, z.d.)



Figuur 9.16 C4 Code diagram

*Figuur 9.16 C4 Code diagram laat een sequence diagram van de StoreAuvikTenantJob zien. Een sequence diagram is een UML-diagram waarmee de samenwerking binnen een proces wordt gevisualiseerd. Door gebruik te maken van pijlen en blokken is de flow van een systeem weer te geven. Een sequence diagram kan worden geclasificeerd als een *behavior diagram*, dit is een diagram die niet de structuur maar het gedrag van een proces weergeeft. Dit is anders van een *structural diagram*, die juist het tegenovergestelde doet. (What Is Sequence Diagram?, z.d.)*

De *sequence diagram* is toegepast omdat er binnen de `StoreAuvikTenantJob` meerdere functies worden aangeroepen die hoewel niet complex zijn wel het proces onoverzichtelijk maken. Een *sequence diagram* kan beter dan andere UML-methodes zoals de *State machine diagram* de stappen vanuit de code vertalen naar een simpele visualisatie.

TESTEN

Tijdens de implementatie van het *Proof of Concept* wordt er gebruik gemaakt van verschillende testmethodes. Het doel is om niet alleen de code te testen maar ook de karakteristieken van de huidige architectuur toegelicht in *paragraaf Architectuur karakteristieken* te controleren. Het testplan en *testcase* zijn terug te vinden in *Bijlage 7, 8 en 9*. In het *hoofdstuk implementatie* wordt het testen toegelicht aan de hand van voorbeelden en verbeteringen.

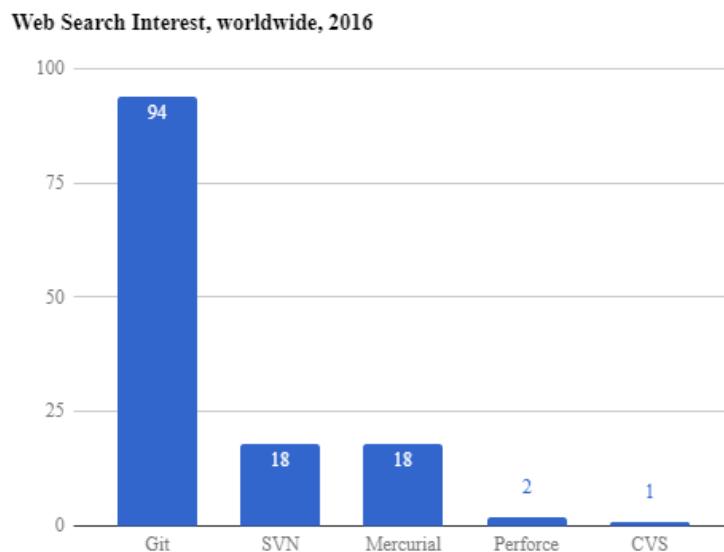
10. IMPLEMENTATIE

Tijdens de implementatie worden de conclusies en observaties samengebracht om een deel van de oplossing te ontwikkelen en beschikbaar te stellen. In het *Hoofdstuk Implementatie* wordt de implementatie van de Auvik API Loader uitgewerkt en toegelicht. De architectuur van de Loader is beschreven in *Hoofdstuk Architectuur*.

Binnen dit hoofdstuk wordt eerst het versiebeheer van het project besproken aan de hand van de systemen die binnen Helmink worden toegepast. In *Paragraaf - Conventies* worden de standaarden en conventies van de code toegelicht, deze standaarden zijn opgesteld vanuit het development team van Helmink. Het *Proof of Concept* wordt uitgewerkt de testmethodes en resultaten worden besproken.

VERSIEBEHEER

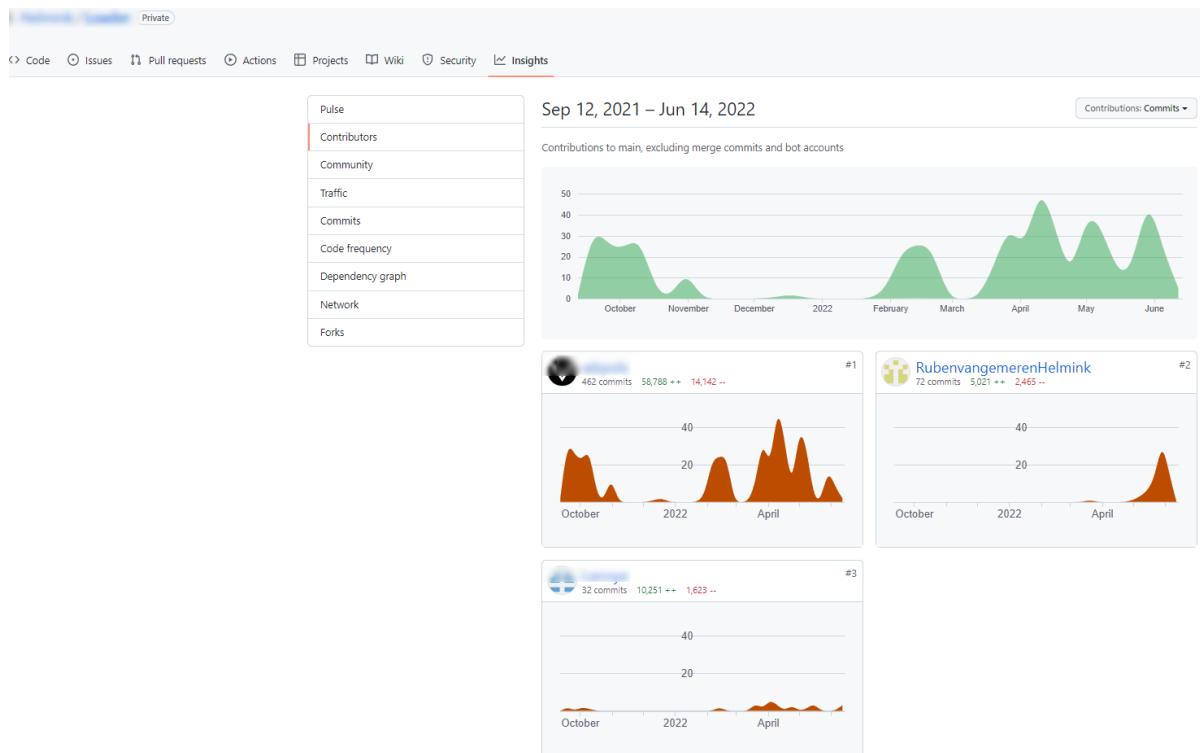
Voor het beheren van het *PoC* wordt er gebruikt gemaakt van *Git*, dit is software waarmee de constante aanpassingen en toevoegingen binnen een softwareproject beheert kunnen worden. Software die de verschillende versies van een project beheren wordt een *Version Control System (VCS)* genoemd. Er zijn verschillende systemen die versie beheer leveren maar uit een onderzoek van 2016 gehouden door het *code management* bedrijf RhodeCode blijkt dat *Git* verre weg het populairste systeem is. Bij *10.1 Verschillende versiebeheer opties* is het verschil duidelijk zichtbaar. (Version Control Systems Popularity in 2016, 2016)



Figuur 10.1 Verschillende versiebeheer opties

Binnen Helmink wordt er gebruik gemaakt van *Git* in de vorm van *Github* en *Github desktop*. *Github* is een online platform dat *Git* toepast zonder daar zelf iets voor hoeven te doen. Helmink beheert alle interne projecten binnen *Github*, deze projecten worden *repositories* genoemd. (An Introduction to GitHub, 2020) De implementatie vindt plaats binnen de *repository* van de Loader server. In *Figuur 10.2*

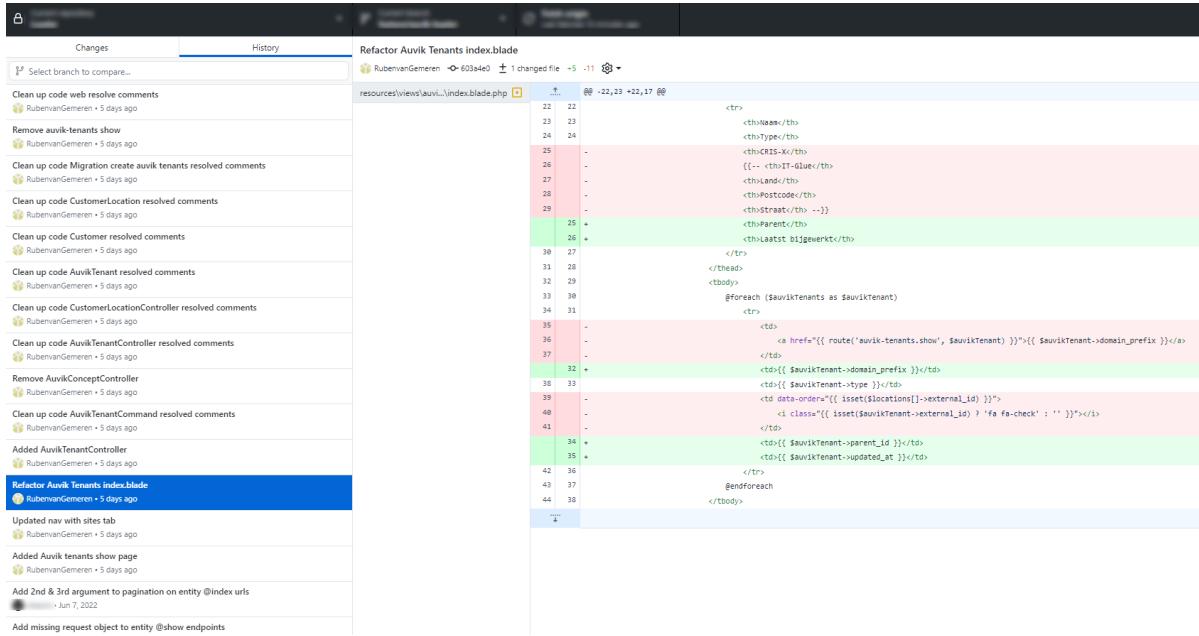
Overzicht Github is een globaal overzicht te zien wie er binnen de *repository* heeft gewerkt.



Figuur 10.2 Overzicht Github

GITHUB DESKTOP

Voor nog meer functionaliteiten gebruikt Helmink *Github desktop*. Een desktopapplicatie die de gebruiker direct toegang geeft tot de code en de aanpassingen die zijn opgeslagen. Als er binnen de code veranderingen worden gemaakt komen deze automatisch binnen de applicatie te staan. Zonder enige commands uit te voeren is het mogelijk om deze nieuwe code op te slaan, aanpassingen die worden opgeslagen binnen een *repository* worden *commits* genoemd.



The screenshot shows the GitHub desktop application interface. On the left, a sidebar lists a series of commits made by 'RubenvanGemen' over the past 5 days. These commits are organized into several branches, indicated by colored labels: 'Clean up code', 'Remove auvik-tenants', 'Clean up code Migration', 'Clean up code CustomerLocation', 'Clean up code Customer', 'Clean up code AuvikTenant', 'Clean up code CustomerlocationController', 'Remove AuvikConceptController', 'Clean up code AuvikTenantCommand', 'Added AuvikTenantController', and 'Refactor Auvik Tenants index.blade'. Each commit includes a brief description and the date it was made.

On the right, the main window displays a code diff for a file named 'resources/views/auvik.../index.blade.php'. The diff shows changes made to the code, with additions in green and deletions in red. The code itself is written in Blade templating language, used for Laravel applications.

Figuur 10.2 User interface Github app

Figuur 10.2 User interface Github app laat zien hoe de *user interface* van *Github Desktop* eruitziet. Links is een geschiedenis van alle *commits* die binnen de *repository* zijn opgeslagen. Rechts is een overzicht van een *commit* te zien met daarbij de code die is aangepast. Voor de verschillende Loaders zijn er *branches* aangemaakt, dit zijn afsplitsingen die later door middel van een *pull request* samengevoegd kunnen worden met de hoofd *branche*. Dit proces wordt *merging* genoemd.

BEST PRACTICES BINNEN VERSIEBEHEER

Helmink maakt gebruik van een aantal *best practices*, dit zijn richtlijnen en gedragsregels die gevolgd worden om het development proces overzichtelijk te houden. Voor het beheren van de code zijn er een aantal regels die zijn gevolgd tijdens de implementatie:

Kleine *commits*

Eén van de regels die wordt toegepast binnen het development team is het gebruik van kleine *commits*, waarbij elke *commit* bestaat uit een kleine hoeveelheid aanpassingen die duidelijk wordt beschreven. Dit gedrag heeft een paar voor- en nadelen. (Kamalizade, 2021)

○ Voordelen

- Meer overzicht. In **Figuur** is te zien dat er veel verschillende *commits* zijn opgeslagen. Het zoeken naar specifieke *commits* is minder gedoe.
- Eerder feedback. Dit hangt er ook vanaf hoe frequent de aanpassingen van de lokale omgeving naar de externe (ook wel *remote*) omgeving worden gestuurd. Als de code vaak wordt

aangepast is het mogelijk om meer feedback te ontvangen vanuit het development team.

- Het terugdraaien van *commits* heeft geen grote impact op de progressie. De developers gebruiken de *revert commit* optie vaak om aanpassingen terug te zetten.
- De *commit* beschrijving kan meer detail bevatten omdat maar een paar aanpassingen worden opgeslagen.

- **Nadelen**

- Bij een *pull request* zijn er veel individuele *commits* die nagekeken moeten worden, dit kan ervoor zorgen dat er minder *pull requests* worden uitgevoerd.
- Bij een grote hoeveelheid kleine aanpassingen moet er aandacht worden besteed aan de manier waarop de code wordt verdeeld. Een kleine *commit* betekent niet altijd dat er weinig aanpassingen worden opgeslagen. In het geval waar veel van dezelfde soort aanpassingen zijn verricht is het wenselijk om één *commit* op te stellen.

Continous improvement

Binnen het development team wordt *continuous improvement* (ook *kaizen*) toegepast, dit is een techniek dat gebruikt maakt van frequente kleinschalige aanpassingen om grote veranderingen door te voeren. Het idee hierachter is dat systemen altijd verbeterd kunnen worden.

De techniek bevat zeven stappen die het proces dat per onderdeel wordt herhaald, verschillende cyclussen kunnen parallel aan elkaar worden uitgevoerd. (Daniel, 2021)

- **Vraag hulp van medewerkers**

Het vragen om hulp van andere medewerkers zorgt voor een gedeelde interesse in het oplossen van het probleem. Binnen Helmink wordt dagelijks om hulp gevraagd bij de verschillende projecten die lopen. Niet alle medewerkers zijn even actief met het vragen om hulp, toch wordt het aangemoedigd. In *Bijlage 10 - Dagstart* is een dagstart uitgewerkt, in deze dagelijkse meetings worden problemen met iedereen besproken en samen gekeken naar oplossingen.

- **Formuleer problemen**

Formuleer de verschillende problemen en oplossingen. Dit gebeurt bij een dagstart of een wekelijkse meeting. Zie *Bijlage 10 - Dagstart* voor een uitwerking van de dagstart meeting.

- **Verzamel oplossingen**

Bedenk verschillende oplossingen voor de uitgezochte problemen, kijk welke oplossing het beste past en kies één oplossing die gekozen wordt. Het development team van Helmink doet dit grotendeels verbaal. Tijdens de dagstart worden problemen en mogelijke oplossingen besproken.

- **Test beste oplossing**

Test de oplossing die is geformuleerd. De andere medewerkers kunnen assisteren. Deze stap gebeurt nadat de dagelijkse meeting is afgerond.

- **Analyseer de oplossing**

Analyseer of de oplossing het probleem op de juiste manier oplost. Bepaal de kwaliteit van de oplossing samen met de andere medewerkers. Bij grote problemen wordt de *product owner* van het project geïnformeerd over de oplossing.

- **Succes? Adopteer de oplossing**

Als de oplossing is gecontroleerd door meerdere medewerkers kan het worden geadopteerd. Aanpassingen worden opgeslagen binnen de *repository*. Verdere feedback en testen wordt gedaan door andere medewerkers als de oplossing in productie wordt gebracht.

- **Herhaal proces**

Deze zes stappen zijn een integraal proces binnen de ontwikkeling van systemen binnen Helmink. De dagelijkse dagstart heeft als doel om de eerste drie stappen van het proces uit te voeren, de week meetings worden gebruikt om de verdere stappen te bespreken.



Figuur 10.3 Kaizen methode

Figuur 10.3 Kaizen methode laat de zeven stappen zien in de juiste volgorde.

CODE STANDAARDEN EN CONVENTIES

De implementatie van de Auvik API Loader vindt plaats binnen de code van de Loader server, hoewel een testomgeving is gebruikt om functionaliteiten te testen is het van belang dat de code voldoet aan de standaarden en conventies die binnen Helmink worden toegepast. In deze *paragraaf* worden de principes en code standaarden toegelicht aan de hand van voorbeelden.

ONTWERP PRINCIPES

De programmeertaal *PHP* is de hoofdtaal binnen alle projecten van Helmink. In het *Hoofdstuk 5. Huidige situatie* wordt aangegeven dat *PHP* een *Object-Oriented programming (OOP)* taal is. Dit is een programmeertechniek waar de focus wordt gelegd op het aanmaken en gebruiken van *objects*. De objecten zijn de verschillende componenten die gebruikt worden binnen een project, het zijn waardes en stukken code. Een object komt voort uit een *class*. Een *class* is een blauwdruk wat kan worden geïnstantieerd, deze geïnstantieerde *classes* zijn de *objects* en worden gebruikt om de processen te voltooien. (Doherty, z.d.)

Er zijn een aantal ontwerprincipes bij het volgen van OOP, deze kunnen worden samengevat in een afkorting SOLID en staat voor: *Single-responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion*. (Oloruntoba, 2021) In dit onderzoek wordt alleen de S toegepast, voor een verdere toelichting kan er worden verwezen naar het boek 'SOLID principles' van meneer Kumar Arora. (Arora, 2017)

De principes zijn bedacht door Robert C. Martin (ook Uncle Bob genoemd), een Amerikaanse *software engineer* met verschillende boeken en artikelen over OOP, UML en de Agile werk methodes. (Martin, z.d.)

Single-responsibility Principle (SRP)

Single-responsibility principle (SRP) of enkele verantwoordelijkheid. Het beschrijft het idee dat elke *class*, functie of programma één verantwoordelijkheid mag hebben. Om aan te tonen waarom het belangrijk is om de code op te delen is hieronder een voorbeeld direct afkomstig uit de code. *Figuur 10.4 getTenants* bevat een functie genaamd *getTenants* dat zich bevindt binnen de *StoreAuvikTenantJob*, dat als doel heeft om data (bevat de netwerken die *tenants* worden genoemd) vanuit de API op te halen en binnen de database op te slaan.

```

/**
 * Get a response from '/tenants'.
 *
 * @return array With tenant and parent id if available
 * @throws \GuzzleHttp\Exception\GuzzleException
 */
public function getTenants(string $id = null){
    $tenants = $this->client->get([
        'tenants',
        [
            'query' => [
            ]
        ]
    ]);

    $data = [];
    foreach ($tenants as $tenant) {
        if (isset($tenant->relationships)
            && isset($tenant->relationships->parent)
            && isset($tenant->relationships->parent->data)
            && isset($tenant->relationships->parent->data->id))
            ? array_push($data, [$tenant->relationships->parent->data->id, $tenant])
            : null;
    }
    return $data;
}

```

Figuur 10.4 `getTenants`

De functie heeft twee onderdelen. Het eerste onderdeel haalt data op vanuit de API door een `get()` functie aan te roepen. Het tweede deel van de functie gebruikt een `foreach` loop om van alle datapunten die zijn opgehaald de *parent id* op te halen. De data die is opgehaald zijn de netwerken binnen Auvik. Het *parent id* representeert een ander netwerk dat gekoppeld is aan het netwerk dat wordt gebruikt.

Er wordt een *array* gemaakt met de data van het netwerk en welke *parent id* eraan vast zit. Het probleem van deze functie is tweeledig.

1. De functienaam is `getTenants`, maar wat de functie teruggeeft zijn de *tenants* met een *parent id*. De naam is niet volledig en geeft niet goed aan welke data wordt teruggegeven.
2. Binnen de code zijn er functies waar het *parent id* moet worden opgehaald, als dit nodig is moet de hele functie worden uitgevoerd, daarna moet er door de *array* worden gezocht om de juiste waarde te vinden. Deze methode is omslachtig en vereist het ophalen van veel data vanuit de API, dit kost bandbreedte die wellicht ergens anders nodig is.

Volgens het *single-responsibility principle* moet de functie één doel hebben. Als de functienaam wordt aangehouden moet de functie alleen de *tenants* van de API ophalen. De andere functionaliteit moet in een aparte functie worden gezet, deze tweede functie krijgt een *tenant* als *input* en stuurt het *parent id* terug of een *null*

waarde als er geen *parent id* is gevonden. Het resultaat is te zien in *Figuur 10.5 getTenants met SRP*.

```
public function getTenants(string $id = null){  
    return $this->client->get(  
        'tenants',  
        [  
            'query' => [  
            ]  
        ]  
    );  
}  
  
public function getTenantParentId(\stdClass $tenant)  
{  
    return isset($tenant->relationships)  
        && isset($tenant->relationships->parent)  
        && isset($tenant->relationships->parent->data)  
        && isset($tenant->relationships->parent->data->id)  
        ? $tenant->relationships->parent->data->id  
        : null;  
}
```

Figuur 10.5 getTenants met SRP

De twee functies bevatten maar een enkele verantwoordelijkheid en kunnen afzondelijk van elkaar gebruikt worden. Het toepassen van het *single-responsibility principle* heeft een aantal voordelen die in het voorbeeld niet getoond kunnen worden.

Voordelen

- Bij het veranderen van eisen kan de code eenvoudig worden aangepast. De functies die verouderd zijn kunnen worden verwijderd terwijl nieuwe functies kunnen worden toegevoegd en getest. De impact of de andere functies binnen het systeem zijn minimaal.
- Door de enkele verantwoordelijkheid zijn de functies kleiner en geven de functienamen een beter beeld bij de inhoud van de functie. Dit maakt het lezen van de code eenvoudiger. (Janssen, 2021)

Tijdens de implementatie zijn er een aantal obstakels herkent die het gebruikt van het principe kunnen belemmeren.

Obstakels

- Het kost meer tijd om aparte functies en *classes* te maken in vergelijking met alle code in één functie te schrijven.
- Voor een *junior developer* is het moeilijk om functies die ander functies aanroepen onder de knie te krijgen. Tijdens de implementatie van het *PoC* is er veel tijd besteed aan het begrijpen van alle functies die nodig zijn voor één proces. (Janssen, 2021)

CASE STYLES

Spaties zijn in de meeste programmeertalen niet toegelaten en dus moeten woorden die normaal los staan op een manier aan elkaar worden gezet. In de code wordt er gebruik gemaakt van vier type benamingen, elk type wordt toegepast binnen bepaalde situaties. De verschillende technieken worden *case styles* genoemd.

camelCase

De *camelCase* maakt gebruik van hoofdletters op alle woorden behalve het eerste woord. Het wordt veel gebruikt om variabelen te benoemen. (Divine, 2019) In *Figuur 10.6 camelCase* is te zien hoe *camelCase* wordt gebruikt, de namen van functies en variabelen volgen de structuur. Als variabelen niet uit twee of meer delen bestaat is er geen hoofdletter aanwezig.

```
public function retrieveMoreData(string $url, array $options = [])
{
    try {
        // Get a response.
        $response = $this->client->get($url, $options);
    } catch (\GuzzleHttp\Exception\RequestException $exception) {
        // Set the error response.
        $response = $exception->getResponse();
    } catch (\Exception $exception) {
        // Throw a regular exception.
        throw $exception;
    }
}
```

Figuur 10.6 camelCase

Het gebruik van *camelCase* maakt het het lezen van functies makkelijker.

PascalCase

PascalCase heeft veel gelijkenissen met *camelCase* behalve dat hier ook het eerste woord een hoofdletter krijgt. *PascalCase* wordt gebruikt om *classes* te benoemen. (Divine, 2019)

```
3  namespace App\Models\Auvik;
4
5  use App\Models\Customer\CustomerLocation;
6  use GuzzleHttp\Client;
7  use Illuminate\Database\Eloquent\Model;
8
9  class AuvikTenant extends Model
10 {
11     protected $guarded = [
12         //
13     ];
14
15     public $incrementing = false;
16 }
```

Figuur 10.7 PascalCase

In *Figuur 10.7 PascalCase* is te zien dat niet alleen *classes* maar ook *namespaces* gebruik maken van deze benaming.

snake_case

Namen met een lage streep ertussen worden gebruikt binnen de database en om PHP-functies aan te roepen. Er is ook een versie waar alleen hoofdletters worden gebruikt

```
while (
|   ! is_null($this->getTenantParentId($item))
) {
    // Get the parent.
    $items = array_filter(
        $tenants,
        function ($tenant) use ($item) {
            return $tenant->id == $this->getTenantParentId($item);
        }
    );
}
```

Figuur 10.8 snake_case

In *Figuur 10.8 snake_case* is te zien dat de PHP-functies 'is_null' en 'array_filter' gebruik maken van de benaming.

kebab-case

Kebab-case is terug te vinden binnen de URL's van de verschillende webpagina's binnen de applicatie. (Divine, 2019) De benaming gebruikt een streepje tussen de verschillende termen.

```
var url = "{{ route('locations.auvik-tenants.create', ':id') }}";
url = url.replace(':id', id);
```

Figuur 10.9 kebab-case

Figuur 10.9 kebab-case laat zien hoe de naam van de URL-gebruik maakt van kebab-case.

Don't Repeat Yourself (DRY) en Keep It Simple, Stupid (KISS)

Twee algemene regels die worden gevuld tijdens de implementatie van de Auvik API Loader zijn *Don't Repeat Yourself (DRY)* en *Keep It Simple, Stupid (KISS)*. (Baghel, 2019) Deze regels beschrijven geen specifieke onderdelen, daarentegen zijn het regels die binnen elk deel van het systeem een rol spelen.

Don't Repeat Yourself (DRY):

Jezelf niet herhalen, binnen het schrijven van code is dit een deel waar veel developers vanaf weten maar wat moeilijk is om daadwerkelijk toe te passen. Het concept achter DRY vertelt ons dat elk stuk code zich maar op één plek in het systeem mag bevinden. Dit kan worden gedaan door de code op te delen in kleine stukken die los gebruikt kunnen worden, dit idee is vergelijkbaar met de *Single-responsibility principle* uit de paragraaf *Single-responsibility principle*.

```

207  v   while (
208      ! is_null($this->getTenantParentId($item))
209  v   ) {
210      // Get the parent.
211      $items = array_filter(
212          $tenants,
213          function ($tenant) use ($item) {
214              return $tenant->id == $this->getTenantParentId($item);
215          }
216      );
217  ....
218
219  }
220
221 /**
222 * Gets the parent id from the Auvik tenant
223 *
224 * @param \stdClass $tenant The tenant used.
225 * @return string|null The id of the parent or null
226 * @throws \Exception
227 */
228 public function getTenantParentId(\stdClass $tenant)
229 {
230     return isset($tenant->relationships)
231         && isset($tenant->relationships->parent)
232         && isset($tenant->relationships->parent->data)
233         && isset($tenant->relationships->parent->data->id)
234             ? $tenant->relationships->parent->data->id
235             : null;
236
237 }
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
233
```

```
207     while (
208         ! is_null(isset($item->relationships)
209             && isset($item->relationships->parent)
210             && isset($item->relationships->parent->data)
211             && isset($item->relationships->parent->data->id)
212             ? $item->relationships->parent->data->id
213             : null;
214     ) {
215         // Get the parent.
216         $items = array_filter(
217             $tenants,
218             function ($tenant) use ($item) {
219                 return $tenant->id == isset($item->relationships)
220                     && isset($item->relationships->parent)
221                     && isset($item->relationships->parent->data)
222                     && isset($item->relationships->parent->data->id)
223                     ? $item->relationships->parent->data->id
224                     : null;
225             }
226         );
227     }
```

Figuur 10.11 Slecht voorbeeld DRY

De code van de functie moet in het tweede voorbeeld twee keer gebruikt worden om hetzelfde effect te creëren. Het negeren van de DRY regel maakt de code onnodig lang en minder flexibel, als er een aanpassing moet plaats vinden moet dit nu op meerdere plekken worden aangepast.

Keep It Simple, Stupid (KISS):

Houd het simpel, dit idee valt samen met de regels binnen DRY. Het doel is om de code duidelijker en stabieler te maken. Onnodige complexiteit maakt het moeilijker om fouten te herkennen en systemen aan te passen. In de volgende twee figuren wordt een voorbeeld gegeven waarin dezelfde functie op twee verschillende manieren is ontwikkeld. *Figuur 10.11 Voorbeeld KISS* laat maakt gebruik van een *if statement* om verschillende aspecten van een waarde te controleren, de controles die uitgevoerd moeten worden zijn:

1. Bestaat de waarde
2. Is de waarde een getal
3. Komt de waarde overeen met hetzelfde veld binnen een ander tabel

Als deze controles zijn verricht kan er een *update* plaatsvinden binnen de tabel.

```

57  /**
58  * Store a newly created resource in storage.
59  *
60  * @param \Illuminate\Http\Request $request
61  * @param \App\Models\Customer\CustomerLocation $location
62  * @return \Illuminate\Http\Response
63  */
64 public function store(Request $request, CustomerLocation $location)
65 {
66   if (isset($auvik_id)
67     && preg_match('/^[0-9]+$/ ', $auvik_id) != false
68     && ! is_null(AuvikTenant::doesntHave('location')->where('id', '=', "$auvik_id")))
69   {
70     // does the ID exist in the database
71     $location = CustomerLocation::find($location);
72     if (! is_null($location)){
73       CustomerLocation::update([
74         'auvik_id' => $auvik_id,
75       ]);
76     }
77
78     return;
79   }
80 }
81 }
82 }
```

Figuur 10.11 Voorbeeld KISS

De code in *Figuur 10.11 Voorbeeld KISS* is niet per se slecht, en ook niet te uitgebreid. Het probleem is dat de manier die is gekozen omslachtig over komt, hoewel het in dit ene geval werkt kan dit anders zijn bij een complexer probleem. In *Figuur 10.12 Voorbeeld KISS simpeler* wordt hetzelfde probleem op een veel simpelere manier opgelost, de code heeft hetzelfde doel maar maakt gebruik van de ingebouwde Laravel functies om een leesbaardere oplossing te creëren.

```
30 <  /**
31  * Store a newly created resource in storage.
32  *
33  * @param \Illuminate\Http\Request $request
34  * @param \App\Models\Customer\CustomerLocation $location
35  * @return \Illuminate\Http\Response
36  */
37 public function store(Request $request, CustomerLocation $location)
38 {
39     // Validate request with auvik id and check if it exists
40     $attributes = $request->validate([
41         'auvik_id' => [
42             'required',
43             'numeric',
44             Rule::exists(AuvikTenant::class, 'id')->where('type', 'client'),
45         ]
46     ]);
47
48     // Update database with value
49     $location->update($attributes);
50
51     // Return correct response
52     return response(null, 200);
53
54 }
55 }
```

Figuur 10.12 Voorbeeld KISS simpeler

DOCUMENTATIE

De Loader server beheert veel systemen en wordt parallel ontwikkeld door verschillende developers. Het is essentieel dat de code die wordt geschreven niet alleen werkt maar ook gedocumenteerd wordt. Deze documentatie gebeurt in de vorm van de *commits* die worden uitgevoerd en de beschrijvingen binnen de code zelf, deze worden *comments* genoemd. Binnen *PHP* is het mogelijk om extra commentaar te geven binnen de code met de hulp van *docblocks*. Een *docblock* is een stuk code dat toegepast kan worden om functies te beschrijven, door een korte beschrijving toe te voegen. (PHPDocumenter, z.d.) Het gebruik van *docblocks* geeft de gebruiker ook de mogelijkheid om aan te geven welke types er worden gebruikt binnen de functie, als een functie ergens anders wordt aangeroepen is deze beschrijving te zien.

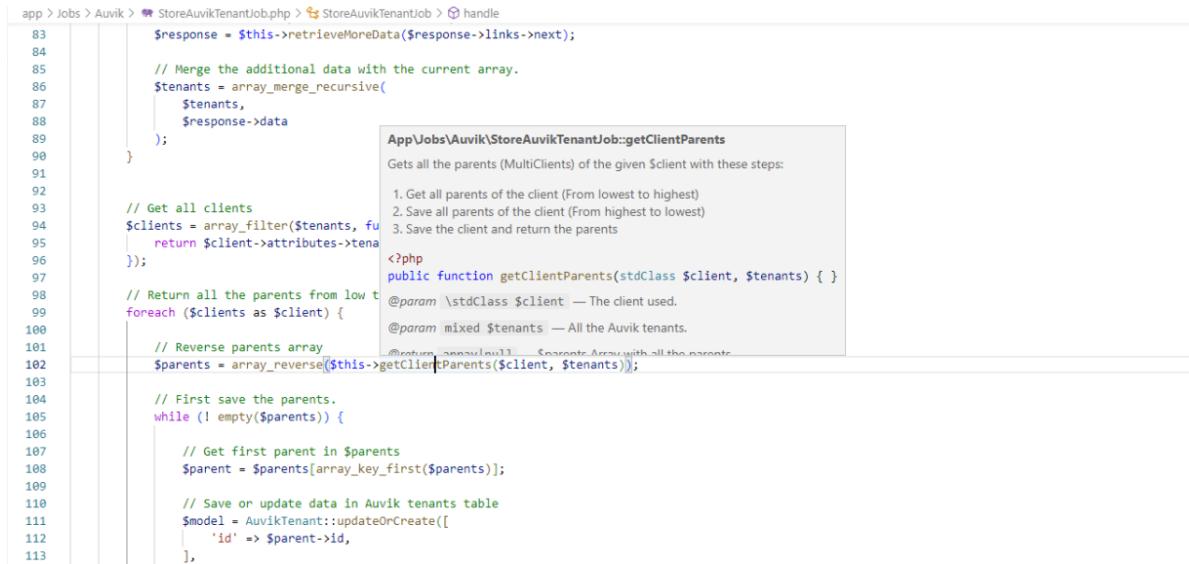
```

187     /**
188      * Gets all the parents (MultiClients) of the given $client with these steps:
189      * 1. Get all parents of the client (From lowest to highest)
190      * 2. Save all parents of the client (From highest to lowest)
191      * 3. Save the client and return the parents
192      *
193      * @param \stdClass $client The client used.
194      * @param mixed $tenants All the Auvik tenants.
195      * @return array|null $parents Array with all the parents
196      * @throws \Exception
197      */
198     public function getClientParents(\stdClass $client, $tenants)
199     {
200
201         $item = $client;
202
203         // Initialized failsafe to stop limit nested parents.
204         $failsafe = 0;

```

Figuur 10.13 DocBlocks voorbeeld

In *Figuur 10.13 DocBlocks voorbeeld* is één van de vele *docblocks* binnen de code weer gegeven. Het eerste deel op lijn 188 bestaat uit een korte beschrijving over wat de functie inhoudt, daarna worden vanaf lijn 193 tot 196 de verschillende parameters beschreven met een korte uitleg. Als de functie *getClientParents* ergens anders wordt aangeroepen komt deze beschrijving in beeld als de muis over de functie wordt geplaatst.



The screenshot shows a tooltip for the `getClientParents` method in a PHP file. The tooltip contains the following information:

App\Jobs\Auvik\StoreAuvikTenantJob::getClientParents

Gets all the parents (MultiClients) of the given \$client with these steps:

1. Get all parents of the client (From lowest to highest)
2. Save all parents of the client (From highest to lowest)
3. Save the client and return the parents

<?php
 public function getClientParents(stdClass \$client, \$tenants) { }
 @param \stdClass \$client — The client used.
 @param mixed \$tenants — All the Auvik tenants.

Figuur 10.14 DocBlocks extra informatie

Figuur 10.14 DocBlocks extra informatie laat dit zien. Deze optie is alleen zichtbaar als de optie beschikbaar is binnen het programma waarbij de code wordt geschreven, de voorbeelden hierboven maken gebruik van *Visual Studio Code*.

Naast *docblocks* worden reguliere *comments* ook gebruikt, deze hebben geen verdere functie en worden geschreven door de developer om toelichting te bieden. *Comments* worden gebruikt bij elke stuk code waar het niet direct duidelijk is welke handelingen er worden verricht.

```
89 // Show a message what is being handled.  
90 $this->output->text('Handling: ' . basename(AuvikTenant::class) . 's');  
91  
92 // Start a progress bar.  
93 $progress = $this->output->createProgressBar($this->tenants->count());  
94  
95 // Loop through all specified tenants.  
96 foreach ($this->tenants as $tenant) {  
97     // Execute the job.  
98     try {  
99         HandleAuvikClientJob::dispatch($tenant);  
100    } catch (\Throwable $th) {  
101        $this->output->newLine();  
102        $this->output->error("Failed to assign location to tenant '{$tenant->domain_prefix}'!");  
103    }  
104  
105    // Advance the progress bar.  
106    $progress->advance();  
107 }  
108  
109 // Finish the progress bar.  
110 $progress->finish();  
111
```

Figuur 10.15 DocBlocks 2de voorbeeld

Een voorbeeld van het gebruik van *comments* is te zien in *Figuur 10.15 DocBlocks 2de voorbeeld*, de code hierboven laat wordt gebruikt om een *job* wordt uitgevoerd en welke andere activiteiten daaromheen gebeuren. Met alle verschillende functies die gebruikt worden is het op het eerste gezicht lastig om te weten wat er gebeurt. de *comments* helpen hierbij en door ze één voor één langs te gaan worden onduidelijkheden verminderd.

PROOF OF CONCEPT

Deze paragraaf beschrijft het *Proof of Concept (PoC)*. Aan de hand van de software eisen die worden beschreven in *Hoofdstuk 6. Requirements* en de software karakteristieken gedefinieerd in *Hoofdstuk 9. Architectuur*. De verschillende functies zijn toegelicht met voorbeelden en stukken code.

FUNCTIONES

Het PoC bestaat uit drie belangrijke functies die ieder opgedeeld zijn in specifieker onderdelen. Samen vormen deze functies de datastromen en eigenschappen van het systeem. In het *Hoofdstuk 6. Requirements* staat beschreven welke functionaliteiten de Auvik API Loader bevat. De functies zijn:

1. **Auvik netwerk dataopslag**
 1. Beschrijft de manier waarop data van de Auvik API wordt opgehaald en opgeslagen binnen de database van de Loader server.
2. **Koppeling met klantgegevens**
 1. Hoe de netwerkgegevens van Auvik kunnen worden gekoppeld met de locatie gegevens van de klanten, deze data staat binnen de Loader server en is oorspronkelijk afkomstig van het CRIS-X systeem.
3. **Visualisatie**
 1. Beschrijft hoe de verrijkte data is gevisualiseerd binnen de *user interface* van de Loader server.

AUVIK API

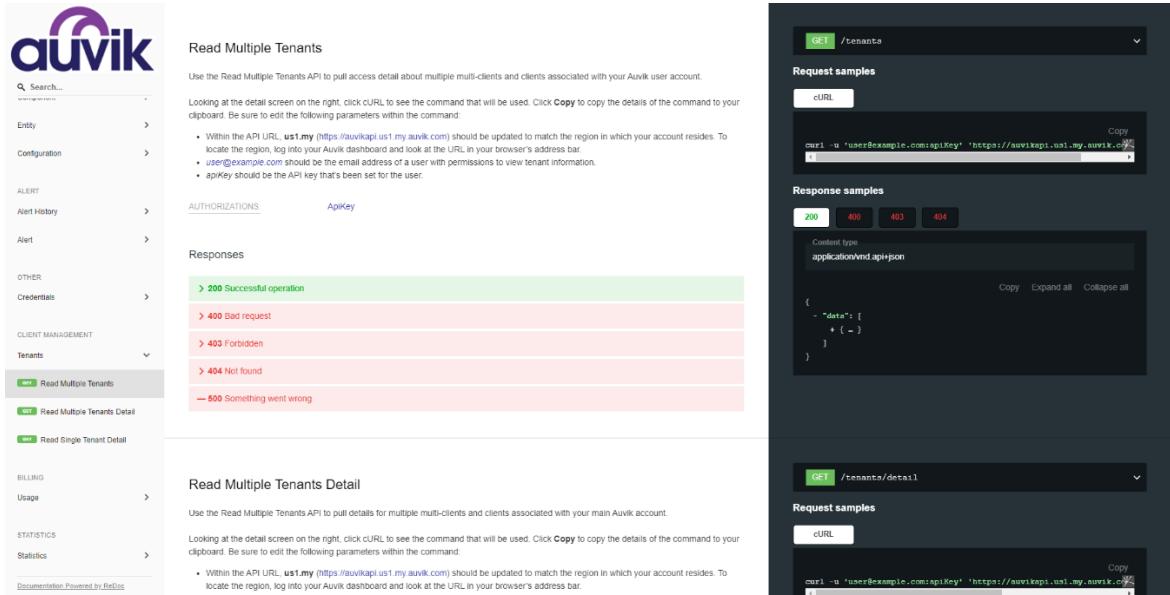
Data van Auvik kan worden verkregen door middel van een API. Deze API heeft verschillende *endpoints*, dit zijn eindbestemmingen van onder andere API's. De verschillende *endpoints* bieden data aan van de netwerken die de klant in beheer heeft. Om de Auvik API te kunnen benaderen zijn de juiste inloggegevens nodig. Deze inloggegevens worden samen met een *API Key* meegestuurd naar Auvik. In *Figuur 10.16 Auvik API Client* is het verzoek weergegeven (vanaf lijn 44) dat naar Auvik wordt gestuurd, het antwoord kan worden gebruikt als 'toegang' tot dat API en daarbij de data van Helmink.

```
35  /**
36   * Handle the incoming request.
37   *
38   * @param  \Illuminate\Http\Request  $request
39   * @return \Illuminate\Http\Response
40   */
41  public function __invoke(Request $request)
42  {
43
44      $this->client = new \GuzzleHttp\Client([
45          'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
46          'base_uri' => config('services.auvik.base_url'),
47          'headers' => [
48              'Accept' => 'application/json',
49              'Content-Type' => 'application/json',
50              'Encode' => 'UTF-8',
51          ],
52      ]);
53  }
```

Figuur 10.16 Auvik API Client

API Mogenlijkheden

Auvik heeft eigen documentatie beschikbaar voor het gebruik van de API, hierin staat welke verschillende *endpoints* benaderd kunnen worden en wat voor antwoord terug wordt gestuurd. (Auvik, z.d.) *Figuur 10.17 Auvik API Documentatie* laat één van de *endpoint* zien, naast een beschrijving van de data wordt er ook een voorbeeld gegeven van een antwoord. Als dit wordt uitgefouwlen is te zien welke data terug wordt gestuurd, dit is in een JSON-object.



The screenshot displays the Auvik API documentation for the 'Tenants' endpoint. It is divided into two main sections: 'Read Multiple Tenants' and 'Read Multiple Tenants Detail'. Each section contains a brief description, a 'Request samples' section with a curl command, and a 'Response samples' section showing a JSON response. The 'Read Multiple Tenants' section also includes a 'Responses' table with status codes and their descriptions.

| Responses |
|--------------------------|
| 200 Successful operation |
| 400 Bad request |
| 403 Forbidden |
| 404 Not found |
| 500 Something went wrong |

Figuur 10.17 Auvik API Documentatie

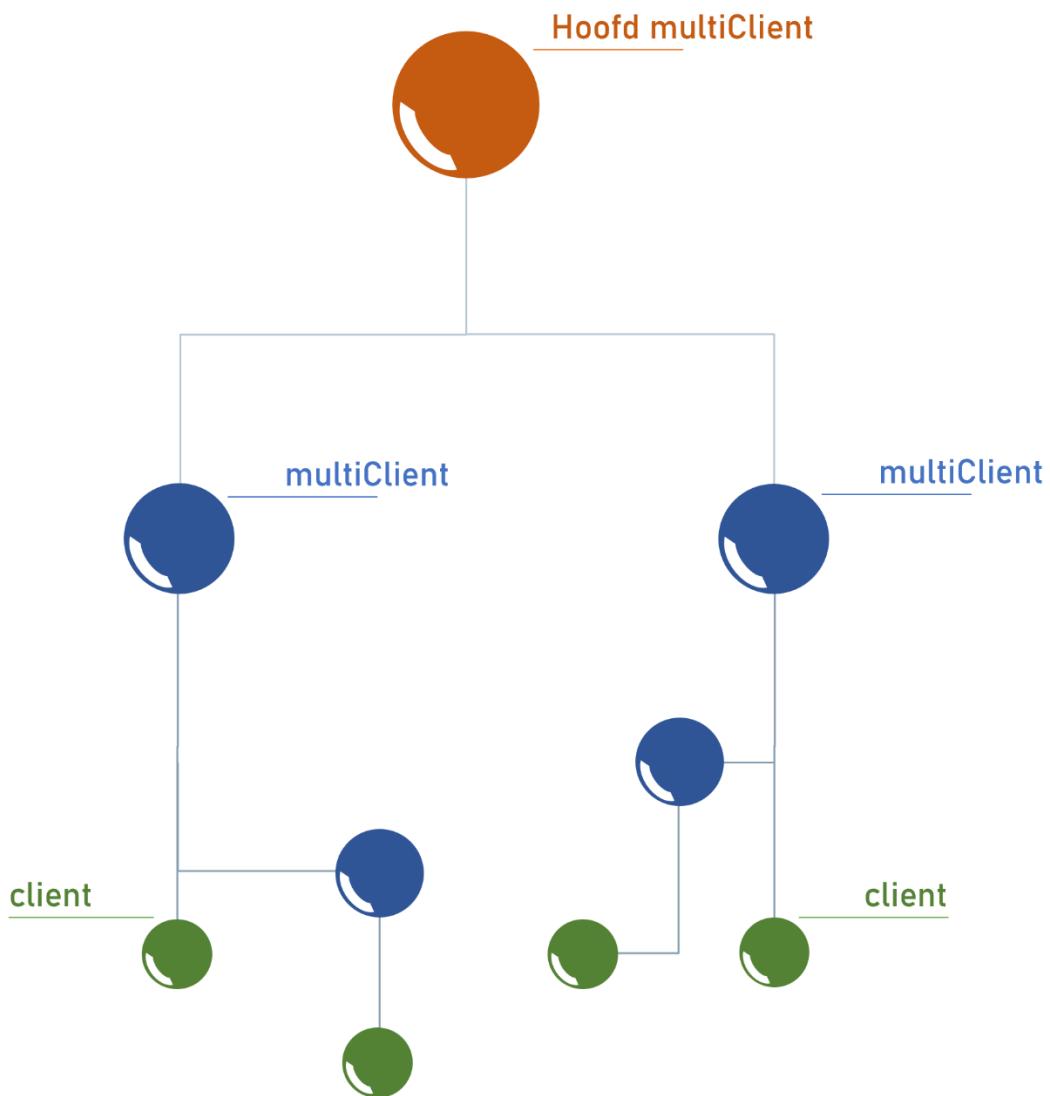
Voor het opslaan van de Auvik netwerkgegevens van Helmink zijn de volgende API endpoints gebruikt:

Tenants

Tenants is de term binnen Auvik die wordt gebruikt voor een specifiek netwerk. Binnen Helmink wordt de term *Sites* gebruikt. Er wordt onderscheid gemaakt tussen twee soorten *tenants*, *clients* en *multiClients*.

MultiClients zijn de *tenants* die andere netwerken onder zich hebben, zogeheten *children*. Elke *multiClient* heeft minimaal één *child*, dit kan een *client* zijn of een nog een *multiClient*.

Clients zijn de netwerken die geen *children* hebben. Elke *client* heeft wel een *multiClient* boven zich, dit wordt een *parent* genoemd. Om de structuur van de verschillende *tenants* te visualiseren is *Figuur 10.18 Auvik API datastructuur* opgesteld.



Figuur 10.18 Auvik API datastructuur

Binnen elk netwerk is er één hoofd *multiClient* aanwezig, dit is het netwerk waar alle andere netwerken onder vallen, in het geval van Helmink is deze hoofd *multiClient* Helmink zelf. In *Figuur 10.18 Auvik API datastructuur* is ook te zien dat *multiClients* ook weer *multiclients* kunnen bezitten, deze nesteling kan in theorie oneindig doorgaan. *Figuur 10.19 Ophalen tenants* laat zien hoe alle *tenants* worden opgehaald binnen de code.

```

138     /**
139      * Get a response from '/tenants'.
140      *
141      * @return \Psr\Http\Message\ResponseInterface
142      * @throws \GuzzleHttp\Exception\GuzzleException
143      */
144     public function getTenants(string $id = null){
145         return $this->client->get(
146             'tenants',
147             [
148                 'query' => [
149                     ]
150             ]
151         );
152     }

```

Figuur 10.19 Ophalen tenants

Het is mogelijk om bepaalde parameters mee te geven is dit niet noodzakelijk. Op lijn 145 wordt de '\$this->client' meegegeven, dit is de instantie van de *client* die in *Figuur 10.19 Ophalen tenants* is gedefinieerd. Alle verzoeken naar de API gaan via deze instantie om de authenticatie te garanderen.

Tenants detail

Voor extra informatie kan er gebruik worden gemaakt van een ander *endpoint* die de details van een gegeven *tenants* teruggeeft. Als het de hoofd *tenant* wordt meegegeven worden alle details van alle onderliggende *tenants* teruggegeven. in *Figuur 10.20 Ophalen tenant details* wordt de functie gebruikt om de details van een specifieke *tenant* te krijgen, deze *tenant* is gerepresenteerd door '\$this' en bevat meerdere waardes die gebruikt worde. De '\$this->domain_prefix' is de naam van de *tenant*.

```

47         // Get the details from auvik.
48         $details = $client->get("tenants/detail/{$this->id}", [
49             'query' => [
50                 'tenantDomainPrefix' => $this->domain_prefix,
51             ]
52         ]);
53
54         // Return the details.
55         return json_decode($details->getBody()->getContents())->data;
56     }

```

Figuur 10.20 Ophalen tenant details

Devices' Info

Om toegang te krijgen tot de apparaten binnen een *tenant* wordt er gebruikt gemaakt van dit *endpoint*. Net als bij de *tenants* details is het mogelijk om de apparaten van een specifiek *tenant* op te halen zowel als van alle *tenants*. Dit kan door deze te specificeren binnen de parameters van het verzoek. *Figuur 10.21 Ophalen tenant devices* laat zien hoe dit binnen de code gebeurt, veel van de parameters zijn niet geactiveerd omdat ze binnen deze instantie niet nodig zijn.

```

38
39     $response = $this->client->get(
40         'inventory/device/info',
41         [
42             [
43                 'query' => [
44                     // 'fields' => [
45                         //     'deviceDetail' => implode(',', [
46                             //         'discoveryStatus',
47                             //         'components',
48                             //         'connectedDevices',
49                             //         'configurations',
50                             //         'manageStatus',
51                             //         'interfaces'
52                             //     ]),
53                         // ],
54                         'filter' => [
55                             'deviceType' => $deviceType,
56                         ],
57                         // 'include' => 'deviceDetail',
58                         'page' => [
59                             // 'first' => 10,
60                         ],
61                         'tenants' => $this->tenant->id,
62                     ],
63                 ],
64             );
65
66     $response = json_decode($response->getBody()->getContents());
67
68     return $response;
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88

```

Figuur 10.21 Ophalen tenant devices

Op lijn 60 is te zien dat we de apparaten van een specifieke *tenant* willen, door de parameter '\$this->tenant->id'.

AUVIK NETWERK DATA OPSLAG

De eerste stap van het systeem is om data op te halen vanuit de API beschikbaar gesteld door Auvik, vervolgens wordt de data opgeslagen binnen de database van de Loader server. Om de relaties tussen de data in stand te houden moet de structuur van de data moet gerespecteerd worden als deze wordt opgeslagen.

Migrations

Om de data vanuit de API op te slaan binnen de database moeten de tabellen binnen de database worden aangemaakt. Om dit te doen wordt er gebruik gemaakt van *migrations*. *Migrations* kunnen worden vergeleken met *git* maar dan specifiek voor de database, het maakt gebruik van een schema om de database te manipuleren. De naam van het *migration* bestand kan worden door Laravel gebruikt voor de tabelnaam. *Figuur 10.22 Database migration* laat de *migration* zien die de tabel *auvik_tenants* in de database creëert.

```

7   class CreateAuvikTenantsTable extends Migration
8  {
9    /**
10   * Run the migrations.
11   *
12   * @return void
13   */
14  public function up()
15  {
16    Schema::create('auvik_tenants', function (Blueprint $table) {
17      $table->unsignedBigInteger("id")->primary();
18
19      $table->unsignedBigInteger("parent_id")->nullable(true);
20      $table->string("type");
21      $table->string("domain_prefix");
22      $table->timestamps();
23
24    });
25
26    Schema::table('auvik_tenants', function (Blueprint $table) {
27      // Foreign Keys.
28      $table->foreign('parent_id')
29          ->references('id')
30          ->on('auvik_tenants')
31          ->onUpdate('cascade')
32          ->onDelete('cascade');
33
34    });
35  }
36
37  /**
38   * Reverse the migrations.
39   *
40   * @return void
41   */
42  public function down()
43  {
44    Schema::dropIfExists('auvik_tenants');
45  }
46 }
47

```

Figuur 10.22 Database migration

Tussen lijnen 16 en 23 worden de velden opgesteld, hierbij betekend 'primary' dat het veld *id* het veld is dat gebruikt wordt om de rij te identificeren, de waardes in dit veld moeten altijd uniek zijn om conflicten te voorkomen. Dit wordt een *primary key* genoemd. (SQL PRIMARY KEY Constraint, z.d.)

Bij de *comment* 'Foreign Keys' op lijn 27 tot 32 is te zien hoe een referentie wordt gemaakt naar de *parents* binnen de datastructuur van Auvik. Deze code zegt dat de waarde *parent_id* binnen de tabel een referentie heeft naar het veld *id* binnen dezelfde tabel. Als een waarde in dit veld overeenkomt met een waarde in het veld *id* wordt de verbinding gemaakt. Dit wordt een *foreign key* genoemd en vindt vaak plaats om meerdere tabellen te verbinden met elkaar. (SQL - Foreign Key, z.d.)

De term *cascade* refereert naar het gedrag binnen de tabel zodra het veld *id* wordt aangepast, als dit gebeurt moet de corresponderende waardes binnen het veld *parent_id* ook worden aangepast. Als de waarde wordt verwijderd moet dit ook binnen de *parent_id* gebeuren. Deze optie wordt gebruikt om te zorgen dat er geen waardes binnen aanwezig zijn binnen *parent_id* die nergens naar refereren. (Babu, 2021)

StoreAuvikTenantJob

Na het opstellen van de tabel moet de data vanuit de Auvik API worden opgehaald en opgeslagen, dit brengt een paar aantal valkuilen met zich mee. Ten eerste de API niet alle informatie in één keer terug, bij grote hoeveelheden data wordt het antwoordt opgedeeld in delen die gekoppeld zijn via een *link*, dit is een veld dat verwijst naar het volgende deel van de data. Binnen de code moet er gecontroleerd worden of dit het geval is.

Een andere valkuil is de structuur van de data, *paragraaf Auvik API* beschrijft hoe de structuur in theorie oneindig diep kan zijn, zie *Figuur 10.18 Auvik API datastructuur* voor een visuele representatie. Dit probleem wordt verholpen door vanaf de *client* te werken, een *client* is altijd het diepste punt in een structuur en wordt gebruikt als startpunt om de rest van de *parents* op te halen en op te slaan.

Om data op te halen en op te slaan wordt er een *job* gebruikt die kan worden geactiveerd door een developer. Binnen de *job* komen de volgende stappen voor.

- **Start de *Client***

Om de verzoeken te maken wordt de *client* aangemaakt, de code hiervoor is toegelicht binnen *Figuur 10.16 Auvik API Client*. Het doel van deze code is om een instantie van de *client* op te zetten die alle nodige informatie bevat om toegang te krijgen tot de Auvik data van Helmink, bij het maken van alle verzoeken wordt deze *client* gebruikt.

- **Haal data op**

Het ophalen van data initieel gebeurt via de *getTenants()* functie die alle *tenants* terug krijgt, de code hiervoor is toegelicht binnen *Figuur 10.19 Ophalen tenants*. Deze code geeft niet altijd alle *tenants*, dit is afhankelijk van de hoeveelheid data. Om zeker te zijn dat alle data is opgehaald is de functie *retrieveMoreData()* opgesteld, de functie is weer gegeven in *Figuur 10.23 retrieveMoreDate()*.

```

154  /**
155   * Get additional data from a paginated request.
156   *
157   * @param string $url  The URL to access.
158   * @param array  $options  The options to include in the request.
159   * @return \stdClass
160   * @throws \Exception
161   */
162 public function retrieveMoreData(string $url, array $options = [])
163 {
164     try {
165         // Get a response.
166         $response = $this->client->get($url, $options);
167     } catch (\GuzzleHttp\Exception\RequestException $exception) {
168         // Set the error response.
169         $response = $exception->getResponse();
170     } catch (\Exception $exception) {
171         // Throw a regular exception.
172         throw $exception;
173     }
174
175     // Transform the response into a JSON object.
176     $response = json_decode($response->getBody()->getContents());
177
178     // Dump the errors if present.
179     if (isset($response->errors)) {
180         dd($response);
181     }
182
183     // Return the response.
184     return $response;
185 }
186

```

Figuur 10.23 retrieveMoreData()

De functie krijgt de *link* waarde beschreven in het begin van de *paragraaf* en probeert deze te gebruiken om nog meer data op te halen mocht deze data aanwezig zijn, de data wordt daarna toegevoegd aan de rest. Dit proces vindt net zo lang plaats totdat er geen data meer aanwezig is.

- **Stuur data naar database**

Als alle data is opgehaald moet deze worden opgeslagen in de tabel, dit moet op een bepaalde manier om de structuur te waarborgen. Omdat er gebruik wordt gemaakt van een *foreign key* zoals beschreven in de *subparagraaf Migrations* moeten eerst alle *parents* worden opgeslagen voor de *clients*. Als dit niet gebeurt ontstaat er het volgende probleem:

- Als een *tenant* (*client* of *multiClient*) wordt opgeslagen voordat de *parent* is opgeslagen blijft het veld *parent_id* leeg, deze moet nadat alles is opgeslagen nog gevuld worden wat erg omslachtig is.

Om dit probleem te voorkomen wordt de *functie getClientParents()* gebruikt. Binnen deze functie wordt per *tenant* alle *parents* opgeslagen van de hoogste tot de diepste, deze volgorde wordt gebruikt om zeker te zijn dat per *client* eerst alle *parents* worden opgeslagen. *Figuur 10.24 getClientParents* laat de functie zien.

```

187  /**
188   * Gets all the parents (MultiClients) of the given $client with these steps:
189   * 1. Get all parents of the client (From lowest to highest)
190   * 2. Save all parents of the client (From highest to lowest)
191   * 3. Save the client and return the parents
192   *
193   * @param \stdClass $client The client used.
194   * @param mixed $tenants All the Auvik tenants.
195   * @return array|null $parents Array with all the parents
196   * @throws \Exception
197   */
198  public function getClientParents(\stdClass $client, $tenants)
199  {
200      $item = $client;
201
202      // Initialized failsafe to stop limit nested parents.
203      $failsafe = 0;
204
205      // Find parent and store them.
206      while (
207          ! is_null($this->getTenantParentId($item))
208      ) {
209          // Get the parent.
210          $items = array_filter(
211              $tenants,
212              function ($tenant) use ($item) {
213                  return $tenant->id == $this->getTenantParentId($item);
214              }
215          );
216
217          // Set the parent to the array.
218          if (! empty($items)) {
219              $parents[] = $items[array_key_first($items)];
220              $item = $items[array_key_first($items)];
221
222          } else {
223              $item->id = null;
224          }
225
226          // Increment the failsafe by one.
227          $failsafe++;
228
229          // Stop the execution if condition is true.
230          if ($failsafe >= 5) break;
231      }
232
233      return $parents ?? [];
234  }

```

Figuur 10.24 getClientParents

Vanaf lijn 206 is te zien dat er een *while* loop wordt gebruikt, deze loop stopt pas als alle *parents* gevonden zijn. Binnen de loop is een *array_filter()*, een PHP functie die een array filtert op een bepaalde functie. Op lijn 213 is te zien dat alleen de *tenants* met een *id* dat overeenkomt met de *parent_id* worden opgehaald, deze lijst met *parents* wordt teruggestuurd.

Op lijn 203 is te zien dat er een *failsafe* waarde is geïnitialiseerd, elke keer dat de *while* loop een reeks doorloopt wordt de *failsafe* met 1 verhoogd. Als deze waarde hoger of gelijk aan 5 is stopt het systeem, dit is ingebouwd om ervoor te zorgen dat er geen mogelijke oneindige loop kan ontstaan. Het gebruik van een *failsafe* is niet vereist in deze situatie, toch is het een handig vangnet voor als iets misgaat.

Figuur 10.25 getClientParents 2de deel laat de rest van de logica zien. Op lijn 102 wordt de functie van *Figuur 10.24 getClientParents* aangeroepen. Als deze data is gesorteerd worden de *parents* in een *while* loop opgeslagen in de tabel, zie lijn 110 tot 119. Als dit is gebeurt worden de *clients* opgeslagen, zie lijn 128 tot 134.

```

93     // Get all clients
94     $clients = array_filter($tenants, function($client){
95         return $client->attributes->tenantType === 'client';
96     });
97
98     // Return all the parents from low to high, reverse order and push parents + child to database
99     foreach ($clients as $client) {
100
101         // Reverse parents array
102         $parents = array_reverse($this->getClientParents($client, $tenants));
103
104         // First save the parents.
105         while (!empty($parents)) {
106
107             // Get first parent in $parents
108             $parent = $parents[array_key_first($parents)];
109
110             // Save or update data in Auvik tenants table
111             $model = AuvikTenant::updateOrCreate([
112                 'id' => $parent->id,
113             ],
114             [
115                 'type' => $parent->attributes->tenantType,
116                 'domain_prefix' => $parent->attributes->domainPrefix,
117                 'parent_id' => $this->getTenantParentId($parent)
118             ]
119         );
120
121         $models[] = $model;
122
123
124         // Unset parent so a infinite loop is avoided
125         unset($parents[array_key_first($parents)]);
126     }
127
128     // Save or update the child.
129     $models[array_key_last($models)]->clients()->updateOrCreate([
130         'id' => (int)$client->id
131     ],
132     [
133         'type' => $client->attributes->tenantType,
134         'domain_prefix' => $client->attributes->domainPrefix
135     ]);
136 }

```

Figuur 10.25 getClientParents 2de deel

De database tabel van *auvik_tenants* wordt weer gegeven in *Figuur 10.26 auvik_tenants tabel*. Er wordt gebruik gemaakt van *PHPMyAdmin* om de database weer te geven.

Figuur 10.26 auvik_tenants tabel

Requirements

De *requirements* voor het ophalen en opslaan van de Auvik data staan beschreven in *Bijlage 6. SRS Auvik API Loader*. Deze software eisen komen overeen met de functionaliteiten binnen het systeem, in de **Tabel** hieronder worden deze eisen samengevat met de paragraaf waar ze worden toegelicht.

| Id | Naam | Beschrijving | Paragraaf |
|-------------------|-----------------------------|---|----------------------------|
| AUVIK_API_L_REQ-1 | Ophalen <i>tenants</i> | Alle netwerkdata van de Auvik API ophalen. De <i>request /tenants</i> wordt gebruikt om alle data van alle netwerken op te halen. | StoreAuvikTenantJob |
| AUVIK_API_L_REQ-2 | Opslaan Hoofd <i>tenant</i> | Het hoofdnetwerk (<i>main tenant</i> genoemd) opslaan in de database met de volgende velden: id, type, domain_prefix, created_at, updated_at. | StoreAuvikTenantJob |
| AUVIK_API_L_REQ-3 | Opslaan <i>multiClients</i> | Per <i>clients</i> worden alle <i>multiClients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. | StoreAuvikTenantJob |
| AUVIK_API_L_REQ-4 | Opslaan <i>clients</i> | Per <i>multiClients</i> worden alle <i>clients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. | StoreAuvikTenantJob |

Tabel 10.1 Systeemeisen Auvik Loader

KOPPELING MET KLANTGEGEVENS

Het doel van het *PoC* is om data vanuit Auvik te koppelen aan bestaande data binnen de Loader server. De *tenants* binnen Auvik representeren netwerken die bij bepaalde klanten horen, specifiek de locaties van klanten. Elke *tenant* heeft een veld genaamd *domainPrefix*. Dit veld bevat de naam van het netwerk dat overeen hoort te komen met de naam van een klant locatie.

Omdat er geen koppeling aanwezig was tussen deze velden is er veel onzuiverheid ontstaan tussen de systemen, hierdoor is het moeilijk te bepalen welk netwerk bij welke klant locatie hoort. Het *PoC* heeft deze koppeling gemaakt en opgeslagen binnen de database.

HandleAuvikClientJob

De *HandleAuvikClientJob* wordt gebruikt om de koppeling tussen de Auvik *tenant* te vinden en deze op te slaan in de database. *Figuur 10.27 HandleAuvikClientJob* laat zien hoe dit in de code wordt gedaan.

```

36  /*
37   * Handles client location with these steps:
38   * 1. Find matching customer location to Auvik tenant
39   * 2. Attach matching customer location to Auvik tenant
40   * 3. Save the relation
41   *
42   * @return void
43   */
44  public function handle()
45  {
46      // Get locations without a auvik_id
47      $locations = CustomerLocation::whereNull('auvik_id')->get();
48
49      // Get the details of the tenant.
50      $details = $this->tenant->detail();
51
52      // Filter location on tenant domain_prefix == location short_name
53      $locations = $locations->filter(function($location) use ($details) {
54          // Set the address weight fields.
55          // $location->setLocationMatches($details);
56
57          return $this->tenant->domain_prefix == $location->getJsonFileContents()->short_name;
58      });
59
60      // Check if location is empty or more than 1
61      if (
62          $locations->isEmpty()
63          || $locations->count() > 1
64      ) {
65          throw new \Exception("None or multiple locations identified");
66      }
67
68      // Get the first location
69      $location = $locations->first();
70
71      // Create association with auvik tenant
72      $location->auvikTenant()->associate($this->tenant);
73
74      // Save location
75      $location->save();
76
77      // Return job
78      return;
79  }
80
81

```

Figuur 10.27 HandleAuvikClientJob

Op lijn 47 worden vanuit de database alle klant locaties (*CustomerLocation*) zonder een *auvik_id* opgehaald. Vervolgens moet er extra informatie worden opgehaald van de *tenant* die wordt gekoppeld, dit gebeurt op lijn 50. De functie *detail()* geeft per *tenant* extra informatie, deze functie bevindt zich in het model van het PoC, hierin worden de definities van het systeem beschreven waaronder de *detail()* functie.

Tussen lijnen 52 en 58 worden de alle klant locaties gefilterd op het *domainPrefix* van de *tenant*. Het veld *domainPrefix* wordt vergeleken met het veld *short_name* op de klant locatie, deze velden worden binnen Helmink gezien als hetzelfde veld en horen gelijk te zijn om een match te vormen.

Als er een match is gevonden kan deze worden opgeslagen, dit gebeurt op lijn 72, de *associate()* functie voegt een *foreign key* toe aan de klant locatie tabel. (Laravel, z.d.)

Als dit proces is afgerond is er voor elke klant locatie die gekoppeld kan worden met een *tenant* een *auvik_id* toegevoegd in de klant locatie tabel, zie *Figuur 10.28 Klant locatie tabel*.

| <code>id</code> | <code>name</code> | <code>external_id</code> | <code>customer_id</code> | <code>json_filename</code> | <code>itglue_id</code> | <code>auvik_id</code> | <code>created_at</code> | <code>updated_at</code> |
|-----------------|-------------------|--------------------------|--------------------------|----------------------------|------------------------|-----------------------|-------------------------|-------------------------|
| 9 | schoonenberg | | | | | | 2022-04-21 16:11:58 | 2022-04-21 17:14:40 |
| 10 | schoonenberg | | | | | | 2022-04-21 16:11:58 | 2022-05-19 13:14:14 |
| 11 | schoonenberg | | | | | | 2022-04-21 16:11:58 | 2022-05-19 13:27:19 |

Figuur 10.28 Klant locatie tabel

CustomerLocation

Na het koppelen van de klant locaties op basis van de *domainPrefix* en *short_name* is er nog een vergelijking die plaatsvindt, omdat een *tenant* gekoppeld is met een klant locatie is het noodzakelijk dat de locatie binnen beide databronnen ook hetzelfde is. Dit is niet nodig om de initiële koppeling te vormen maar moet wel gelijk zijn voor operationele doeleinden.

Deze controle gebeurt binnen het model van de *CustomerLocation* en heeft vooralsnog geen directe impact op het systeem. De implementatie is binnen het PoC alleen intern en wordt niet opgeslagen binnen de database. Het doel van deze controle is om binnen de *user interface* weer te kunnen geven welke locaties nog niet helemaal kloppen. *Figuur 10.29 setLocationMatches()* laat de functie *setLocationMatches()* zien.

```

146 /**
147 * Get the location matches with Auvik tenant.
148 *
149 * @return \App\Models\Customer\CustomerLocation
150 */
151 public function setLocationMatches(\stdClass $detail)
152 {
153     // Get address from location
154     $address = $this->getJsonFileContents()->addresses[
155         array_key_first($this->getJsonFileContents()->addresses)
156     ];
157
158     // Set location match weight
159     $this->setAttribute('address_weight', 0);
160
161     // Determine if the address matches with the one on the model.
162     if (
163         isset($address)
164         && isset($detail->attributes->address)
165     ) {
166         // if true setAttribute plus increase weight
167         $this->setAttribute(
168             'matches_country',
169             $address->country_iso_code === $detail->attributes->address->country
170         );
171         $this->setAttribute(
172             'matches_zipcode',
173             $address->zipcode === $detail->attributes->address->postalCode
174         );
175         $this->setAttribute(
176             'matches_street',
177             strtolower($address->street) . " "
178             . strtolower($address->street_number) === strtolower($detail->attributes->address->address1)
179         );
180
181         // Increase the weight.
182         $this->setAttribute(
183             'address_weight',
184             (int) $this->matches_country
185             + (int) $this->matches_zipcode
186             + (int) $this->matches_street
187         );
188     }
189
190     // Return the modified location.
191     return $this;
192 }
193

```

Figuur 10.29 setLocationMatches()

De functie controleert op drie onderdelen van de locatie, het land, de postcode en de straat. De lijnen 161 tot 190 bevatten een *if statement* waarbij wordt gekeken of de locaties van beide bronnen op één van deze onderdelen matches. Mocht er een match gevonden zijn wordt er een *setAttribute()* toegepast, dit voegt een nieuw veld toe aan het JSON object van de klant locatie. Tussen lijnen 182 en 186 is er een optelsom om te kijken hoeveel onderdelen er worden gekoppeld. Deze data kan gebruikt worden binnen de *user interface*.

Requirements

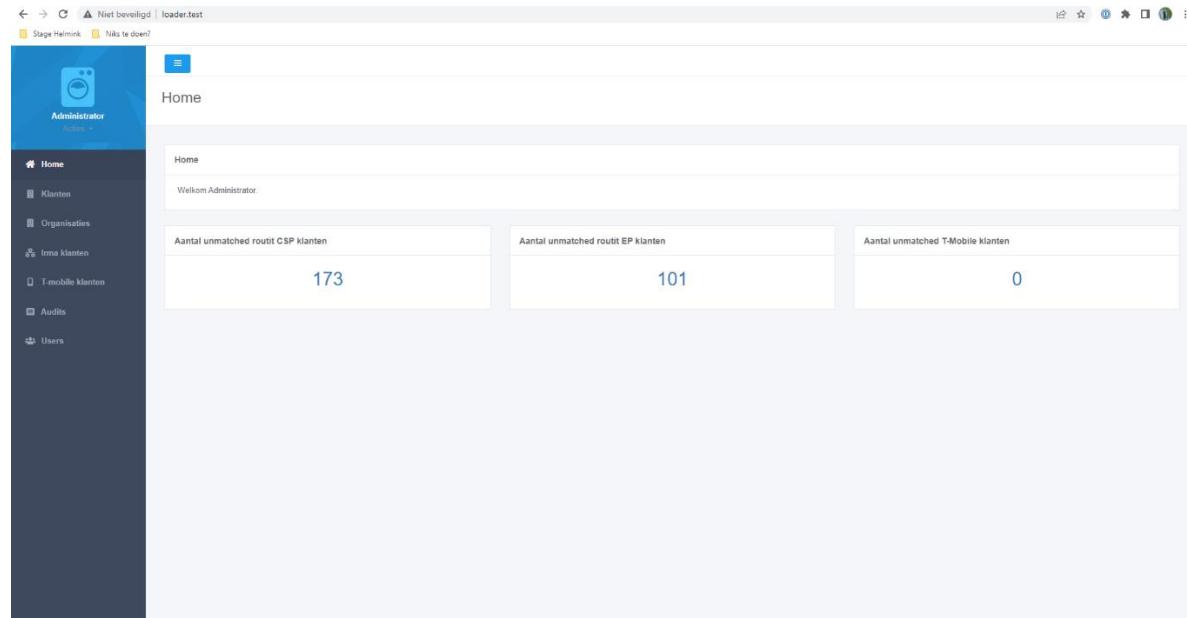
De *requirements* voor het koppelen van de Auvik data met de klantlocaties staan beschreven in *Bijlage 6. SRS Auvik API Loader*. Deze software eisen komen overeen met de functionaliteiten binnen het systeem, in de *Tabel 10.2 Software eisen koppeling* hieronder worden deze eisen samengevat met de paragraaf waar ze worden toegelicht.

| Id | Naam | Beschrijving | Paragraaf |
|-------------------|-------------------------------------|---|-----------------------------|
| AUVIK_API_L_REQ-5 | Controle Auvik data / klantgegevens | Data vanuit de database wordt vergeleken met op basis van de velden domain_prefix bij de Auvik data en short_name bij de klantgegevens. | HandleAuvikClientJob |
| AUVIK_API_L_REQ-6 | Auvik_id invullen | Bij een correcte match moet het veld auvik_id bij de klantgegevens worden ingevuld. | HandleAuvikClientJob |
| AUVIK_API_L_REQ-7 | Locatie controle | Bij een incorrecte match moet de locatie binnen de twee databronnen worden gecontroleerd, dit gebeurt door extra details op te halen vanuit de Auvik API. | CustomerLocation |

Tabel 10.2 Software eisen koppeling

VISUALISATIE

De Loade server heeft een eigen *user interface* waarin de verschillende Loaders worden weergegeven. Per Loader is een aparte omgeving opgezet waar informatie over de relevante wordt getoond, deze data komt deel uit de database en deels vanuit de API's die gebruikt worden. *Figuur 10.30 Hoofdpagina Loader server* laat de hoofdpagina van de Loader server zien.



Figuur 10.30 Hoofdpagina Loader server

Het PoC heeft ook een eigen omgeving binnen de Loader server, hierin wordt de data die is opgehaald en gekoppeld weer gegeven. *Figuur 10.31 Navigatie Loader server* laat zien welke van de onderdelen van de Loader Server horen bij de werkzaamheden van het PoC.



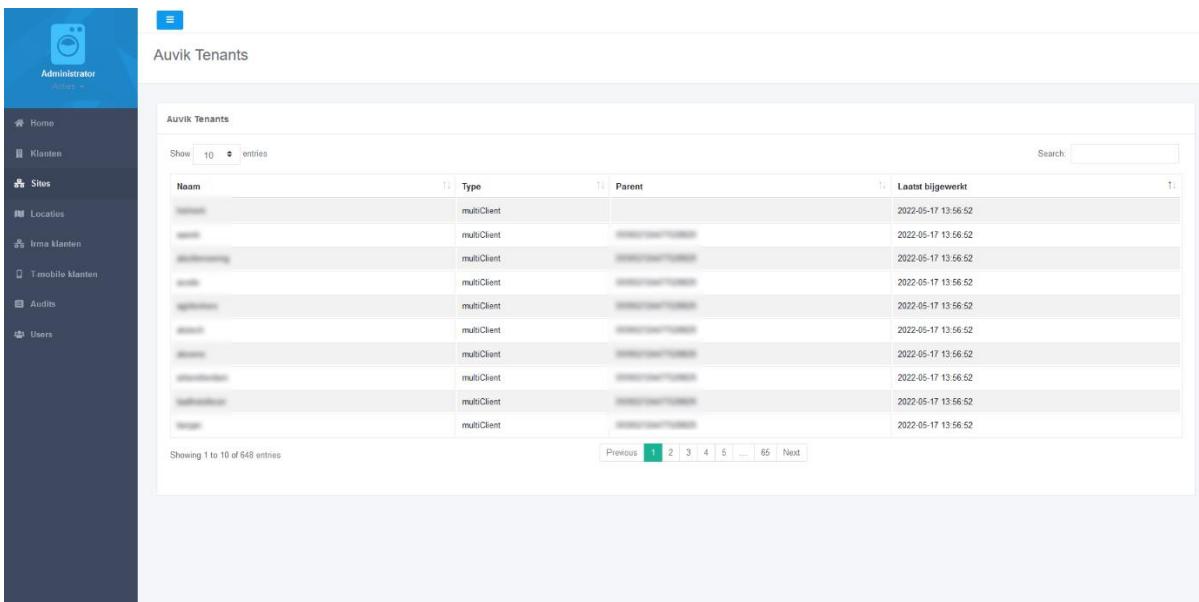
Figuur 10.31 Navigatie Loader server

Er zijn verschillende pagina's opgesteld met visuele representaties van de database tabellen. Deze zijn gemaakt met HTML en *Blade*, een manier om data simpel te kunnen verwerken binnen HTML-code, het wordt standaard meegeleverd met Laravel.

index.blade.php

*Figuur 10.31 Navigatie Loader server laat twee onderdelen zien binnen de Loader server die horen bij het PoC. Deze onderdelen zijn de *sites* en de *locaties*. *Sites* is de interne benaming voor *tenants* en visualiseert de netwerken die zijn opgeslagen binnen de database. *locaties* zijn de klant locaties en visualiseren de locaties en de koppeling met het *auvik_id*.*

*Figuur 10.32 Tenants Loader server toont de index pagina van de *tenants* (*sites*). De pagina bevat een tabel met alle *tenants* in de database, het is mogelijk om zoeken in de lijst en er zijn verschillende weergave opties.*



The screenshot shows a web-based administrative interface for 'Auvik Tenants'. On the left is a dark sidebar with navigation links: Home, Klanten, Sites (selected), Locaties, Irre klanten, Timobile klanten, Audits, and Users. The main content area has a header 'Auvik Tenants' and a sub-header 'Auvik Tenants'. It includes a search bar and a table with 10 entries per page. The table columns are Name, Type, Parent, and Laatst bijgewerkt. All entries show 'multiClient' as the type and various dates from May 17, 2022, as the last update time. At the bottom, it says 'Showing 1 to 10 of 648 entries' and has a navigation bar with links for Previous, Next, and page numbers 1 through 66.

| Name | Type | Parent | Laatst bijgewerkt |
|------------|-------------|------------|---------------------|
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |
| ██████████ | multiClient | ██████████ | 2022-05-17 13:56:52 |

Figuur 10.32 Tenants Loader server

De code bevat een tabel waarin een *foreach* loop wordt gebruikt om door alle *tenants* heen te lopen, zie *Figuur 10.33 Index pagina Tenants*.

```

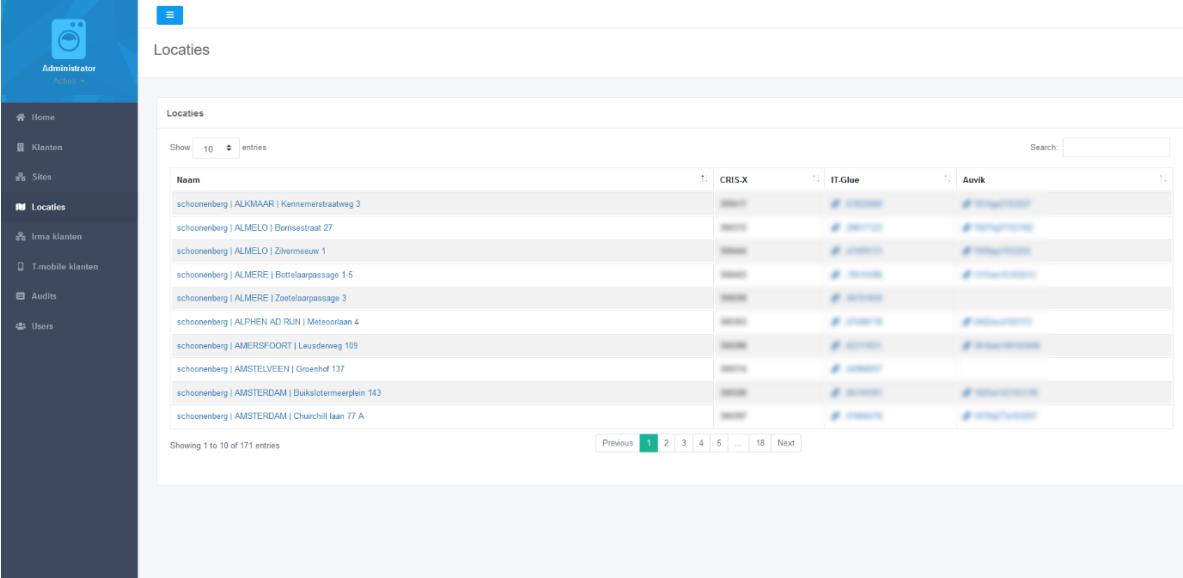
11 <div class="wrapper wrapper-content">
12     <div class="row">
13         <div class="col-12">
14             <div class="ibox">
15                 <div class="ibox-title">
16                     <h5>Auvik Tenants</h5>
17                 </div>
18                 <div class="ibox-content">
19                     <table class="table table-bordered table-striped" id="auvikTenantsTable">
20                         <thead>
21                             <tr>
22                                 <th>Naam</th>
23                                 <th>Type</th>
24                                 <th>Parent</th>
25                                 <th>Laatst bijgewerkt</th>
26                             </tr>
27                         </thead>
28                         <tbody>
29                             @foreach ($auvikTenants as $auvikTenant)
30                             <tr>
31                                 <td>{{ $auvikTenant->domain_prefix }}</td>
32                                 <td>{{ $auvikTenant->type }}</td>
33                                 <td>{{ $auvikTenant->parent_id }}</td>
34                                 <td>{{ $auvikTenant->updated_at }}</td>
35                             </tr>
36                         @endforeach
37                         </tbody>
38                     </table>
39                 </div>
40             </div>
41         </div>
42     </div>
43     </div>
44     </div>
45 @endsection
46
47 @section('scripts')
48     <script>
49         $('#auvikTenantsTable').DataTable({
50             "lengthMenu": [10, 25, 50, 100],
51             "order": [
52                 [0, "asc"],
53             ]
54         });
55     </script>
56 @endsection

```

Figuur 10.33 Index pagina Tenants

Onderaan vanaf lijn 47 wordt er een *script* uitgevoerd die de zoekfunctie en de weergave opties inschakelen.

De klant locaties gebruiken ook een *index* pagina, hierin staat niet alleen statische informatie maar worden er ook links gebruikt naar de overeenkomende *service* waar het veld naar verwijst.



Figuur 10.34 Index pagina klant locaties

In *Figuur 10.34 Index pagina klant locaties* is de *index* pagina van de klant locaties te zien. De verschillende klant locaties vallen allemaal onder één klant (in dit geval Schoonenberg). De koppelingen bevatten links naar de bijbehorende website waar de oorspronkelijke informatie zichtbaar is.

```

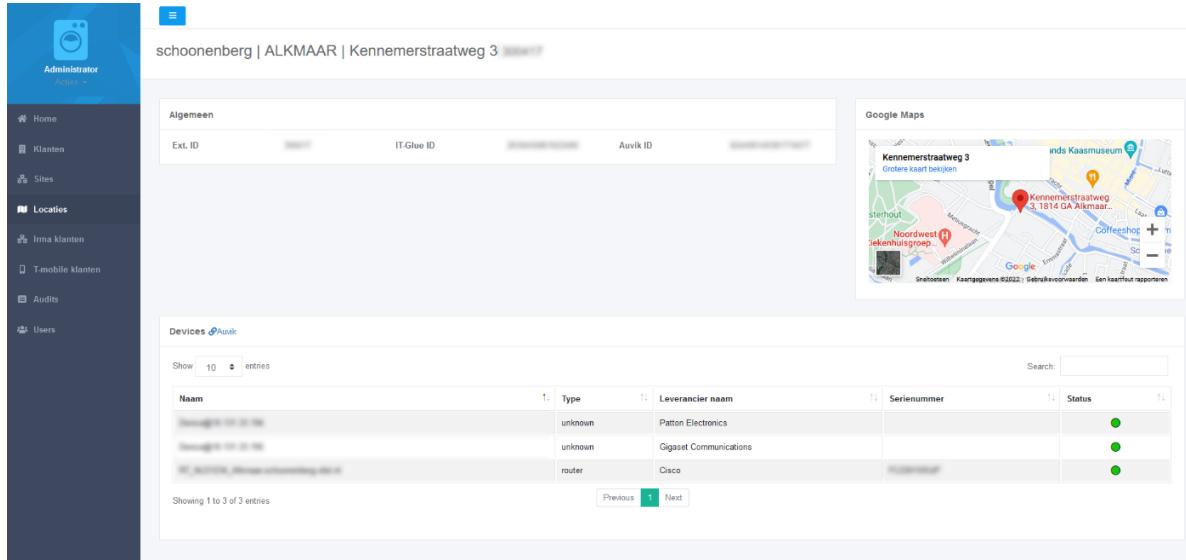
29 <
30 <
31 <
32 <
33 <
34 <
35 <
36 <
37 <
38 <
39 <
40 <
41 <
42 <
43 <
44 <
45 <
46 <
47 <
48 <
49 <
50 <
51 <
52 <
53 <
54 <
55 <
56 <
      <tbody>
        @foreach ($organizations as $organization)
          @foreach ($organization->locations as $location)
            <tr>
              <td>
                <a href="{{ route('locations.show', $location) }}>{{ $location->name }}</a>
              </td>
              <td data-order="{{ $location->external_id }}">
                {{ $location->external_id }}
              </td>
              <td data-order="{{ $location->itglue_id }}">
                @if (isset($location->itglue_id))
                  <a href="https://helmink.eu.itglue.com/{{ $organization->itglue_id }}/locations/{{ substr($location->itglue_id, -8, 8) }}"
                    | <i class="fa fa-link"></i>...{{ substr($location->itglue_id, -8, 8) }}>
                @endif
              </td>
              <td data-order="{{ $location->auvik_id }}">
                @if (isset($location->auvik_id))
                  <a href="https://{{ $location->auvikTenant->domain_prefix }}.eu2.my.auvik.com/">
                    | <i class="fa fa-link"></i> {{ $location->auvikTenant->domain_prefix }}>
                @endif
              </td>
            </tr>
          @endforeach
        @endforeach
      </tbody>
    
```

Figuur 10.35 Code index klant locaties

De code zoals weer gegeven in *Figuur 10.35 Code index klant locaties* laat zien hoe de links worden gemaakt. Op lijn 48 is te zien hoe een link wordt opgesteld met de waarde van het desbetreffende veld, dit gebeurt alleen als er een *auvik_id* aanwezig is. Verder wordt er net als *Figuur 10.33 Index pagina Tenants* een *foreach* loop gebruikt langs om alle klant locaties af te gaan.

show.blade.php

Op de *index* pagina van de klant locaties kan er worden geklikt op een individuele locatie. De creëert een nieuwe pagina met details over die locatie. Dit wordt de *show* pagina genoemd.



| Naam | Type | Leverancier naam | Serienummer | Status |
|-------------------------------------|---------|------------------------|-------------|--------|
| Unknown@192.168.1.100 | unknown | Patton Electronics | | ● |
| Unknown@192.168.1.100 | unknown | Gigaset Communications | | ● |
| RT-AC68U, Wireless Dual Band Router | router | Cisco | | ● |

Figuur 10.36 Show pagina klant locaties

Figuur 10.36 Show pagina klant locaties laat zien welke details worden opgehaald, ten eerste is te zien dat de locatie wordt gebruikt om een Google Maps object te maken, de code hiervoor wordt weer gegeven in *Figuur 10.37 Show code klant locaties*.

```

69 |     <div class="ibox-content">
70 |         <div class="mapouter">
71 |             <div class="gmap_canvas">
72 |                 <iframe id="gmap_canvas" width="100%" height="227"
73 |                     src="https://maps.google.com/maps?q={{ $location->getJsonFileContents()->addresses[0]->city_name }},
74 |                     {{ $location->getJsonFileContents()->addresses[0]->street }},
75 |                     {{ $location->getJsonFileContents()->addresses[0]->street_number }}&z=15&ie=UTF8&iwloc=&output=embed"
76 |                     frameborder="0" scrolling="no" marginwidth="0" marginheight="0"/></iframe>
77 |
78 |             <style>
79 |                 .gmap_canvas {
80 |                     overflow: hidden;
81 |                     background: none !important;
82 |                 }
83 |             </style>
84 |         </div>
85 |     </div>

```

Figuur 10.37 Show code klant locaties

In de code wordt er een link gevormd met de gegevens van de locatie, zie lijn 73 tot 76.

Naast de locatie zelf is op *Figuur 10.36 Show pagina klant locaties* te zien dat de apparaten (*devices*) van de locatie worden weergegeven. Dit zijn de apparaten binnen de Auvik *tenant* die direct worden opgehaald van de API en zijn niet opgeslagen in de database. Om deze apparaten te kunnen ophalen is er een aparte *job* gemaakt om dit verzoek te maken. *Figuur 10.21 Ophalen tenant devices paragraaf Devices' Info* laat de code zien die dit uitvoert.

De data uit de database en de API wordt meegegeven aan de desbetreffende pagina's, dit worden *views* genoemd en zijn één van de drie delen met het *model-view-controller* patroon dat wordt gehanteerd binnen het gehele systeem. Dit patroon is toegelicht in het *Hoofdstuk 9. Architectuur*. *Figuur 10.38 Index en show* laat zien hoe de *index* en *show* pagina's wordt aangemaakt in de *CustomerLocationController* met de juiste data, zie lijn 22 en 36. De functie *compact()* neemt de data en creëert een uniforme *array*. (*PHP Compact() Function*, z.d.)

```

10  class CustomerLocationController extends Controller
11  {
12  /**
13  * Display a listing of the resource.
14  *
15  * @return \Illuminate\Http\Response
16  */
17  public function index()
18  {
19      $organizations = Customer::with('locations.auvikTenant')->get();
20
21
22      return view('locations.index', compact('organizations'));
23  }
24
25
26 /**
27 * Display the specified resource.
28 *
29 * @param \App\Models\Customer\CustomerLocation $location
30 * @return \Illuminate\Http\Response
31 */
32 public function show(CustomerLocation $location)
33 {
34     $job = new GetInventoryDeviceInfo($location->auvikTenant);
35
36     return view('locations.show', compact('location', 'job'));
37 }
38
39

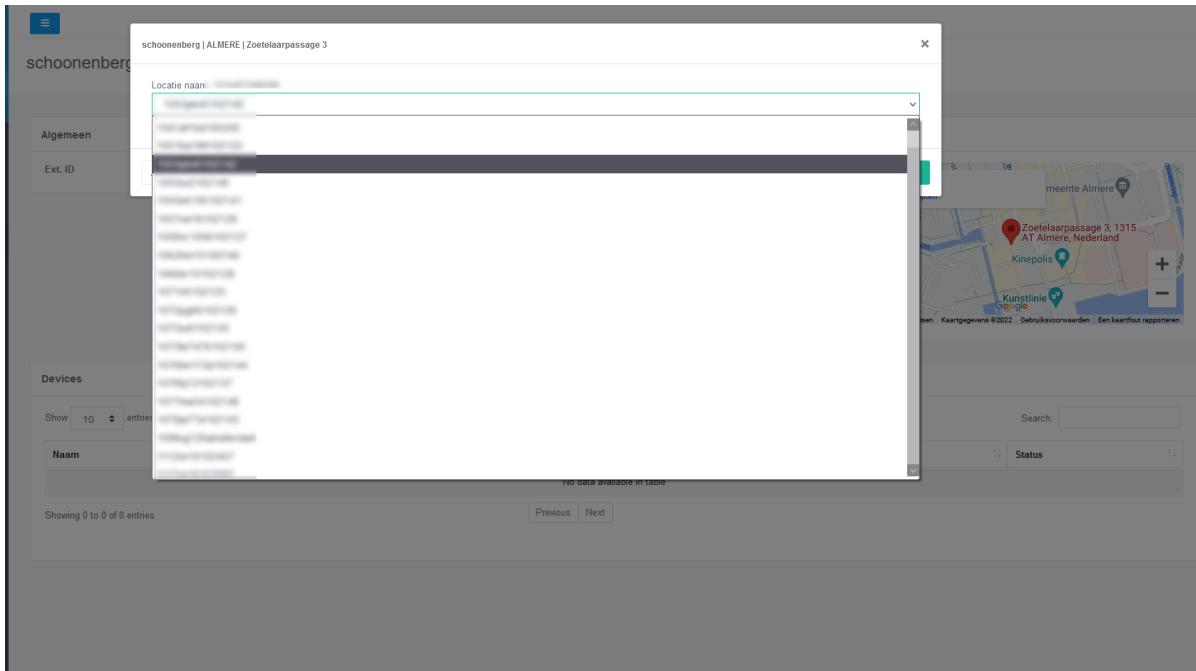
```

Figuur 10.38 Index en show

create.blade.php

Als er geen *auvik_id* is toegewezen kan dit door middel van een *modal*, dit is een *pop-up* venster dat zich op pagina extra informatie of acties kan tonen. (How To Make a Modal Box With CSS and JavaScript, z.d.)

De *show* pagina maakt gebruik van een *modal* om de mogelijkheid te bieden aan de gebruiker om zelf een *auvik_id* te koppelen. In normale omstandigheden is deze functie niet van toepassen maar na overleg met de *product owner* is functie toegevoegd. *Figuur 10.39 Modal* toont hoe het *modal* er in binnen de Loader server uitziet.



Figuur 10.39 Modal

Om de keuze op te slaan wordt er een script gebruikt dat de data naar de server stuurt, hierna wordt de keuze ook in de database opgeslagen. In *Figuur 10.40 Code data opslaan Modal* wordt het proces weer gegeven. Eerst wordt de *toevoegen* knop uitgezet zodat er niet meerdere verzoeken gestuurd kunnen worden (zie lijn 58). Vervolgens wordt er de keuze met de hulp van een *post* verzoek gestuurd naar de server, hiervoor wordt *axios* gebruikt. Dit is een API die gebruikt wordt om data binnen een *brower* te kunnen versturen of ontvangen, (*Getting Started | Axios Docs*, z.d.) in plaats van via de database zoals de rest van het systeem.

```

37
38  <script>
39    function clearErrors() {
40      // Clean up all errors in case different errors occur.
41      $('.form-group').each(function (index, element) {
42        $(element).removeClass('has-error').find('.help-block:first').html('');
43      });
44    }
45
46    function submitForm(button) {
47      // Clear errors
48      clearErrors();
49
50      // Transform the button into a jQuery object.
51      button = $(button);
52
53      // Select the form.
54      let form = button.parents('form:first'),
55          text = button.text();
56
57      // Disable the pressed button.
58      button.prepend("<span class=\"fa fa-spinner fa-pulse\"></span>").prop("disabled", true);
59
60      // Post the data to the server.
61      axios.post(form.attr('action'), form.serialize())
62        .then(function (response) {
63
64          // Reload the page.
65          location.reload();
66        }).catch(function (response) {
67          // Convert the response content to a JSON object.
68          response = JSON.parse(response.request.response);
69
70          // Enable the button and reset its text.
71          button.text(text).prop("disabled", false);
72
73          // Loop through the errors and set a 'has-error' class and add the text to the 'help-block'.
74          $.each(response.errors, function (index, messages) {
75            // Display an error block.
76            $('[name^="' + index + '"]').parents('.form-group:first')
77              .addClass('has-error')
78              .find('.help-block')
79                .html(makeUnorderedList(messages));
80          });
81        });
82    }
83

```

Figuur 10.40 Code data opslaan Modal

Het `post` verzoek werkt met een `then catch` blok, dit is een methode die wordt gebruikt om eventuele problemen op te vangen. Als een bepaalde actie niet lukt wordt de code van het `catch` deel uitgevoerd (zie lijn 62 tot 80), in het geval van dit systeem worden de opkomende problemen weer gegeven. Als het proces is afgerond wordt de keuze gevalideerd en opgeslagen binnen de `controller`. Binnen Figuur 10.41 `Store functie modal` is de functie `store` te zien, deze functie valideert het verzoek aan de hand van een paar regels, de eerste regels kijkt om de waarde van het `auvik_id` correct is geformuleerd. Vervolgens wordt er gekeken of de waarde bestaat binnen de Auvik tabel. Als dit zo is kan de keuze in de tabel van de klant locaties worden geplaatst. Dit proces wordt uitgevoerd tussen lijnen 39 en 49.

```

15 class CustomerLocationAuvikTenantController extends Controller
16 {
17     /**
18      * Show the form for creating a new resource.
19      *
20      * @param \App\Models\Customer\CustomerLocation $location
21      * @return \Illuminate\Http\Response
22      */
23     public function create(CustomerLocation $location)
24     {
25         $tenants = AuvikTenant::doesntHave('location')->where('type', '=', 'client')->orderBy('domain_prefix')->get();
26
27         return view('locations.modals.auvik-tenants.create', compact('location', 'tenants'));
28     }
29
30     /**
31      * Store a newly created resource in storage.
32      *
33      * @param \Illuminate\Http\Request $request
34      * @param \App\Models\Customer\CustomerLocation $location
35      * @return \Illuminate\Http\Response
36      */
37     public function store(Request $request, CustomerLocation $location)
38     {
39         // Validate request with auvik id and check if it exists
40         $attributes = $request->validate([
41             'auvik_id' => [
42                 'required',
43                 'numeric',
44                 Rule::exists(AuvikTenant::class, 'id')->where('type', 'client'),
45             ],
46         ]);
47
48         // Update database with value
49         $location->update($attributes);
50
51         // Return correct response
52         return response(null, 200);
53     }
54 }

```

Figuur 10.41 Store functie modal

Figuur 10.41 Store functie modal laat ook zien hoe de *modal* wordt aangemaakt (lijnen 23 tot 27), alleen de de *tenants* van het type *clients* kunnen worden gekoppeld aan een klant locaties. De *multiClients* liggen een stap hoger en worden gekoppeld aan de klanten zelf, dit proces valt buiten de scope van het PoC.

Requirements

De *requirements* voor het visualiseren van de Auvik data binnen de Loader server staan beschreven in *Bijlage 6. SRS Auvik API Loader*. Deze software eisen komen overeen met de functionaliteiten binnen het systeem, in de **Tabel** hieronder worden deze eisen samengevat met de paragraaf waar ze worden toegelicht.

| Id | Naam | Beschrijving | Paragraaf |
|--------------------|-----------------------------|---|-------------------------|
| AUVIK_API_L_REQ-8 | Weergeven netwerken | De netwerken weergeven binnen de aparte omgeving binnen de Loader server. | index.blade.php |
| AUVIK_API_L_REQ-9 | Weergeven koppeling | De koppeling tussen de klant en het Auvik netwerk weergeven binnen de omgeving van de klantlocatie | index.blade.php |
| AUVIK_API_L_REQ-10 | Details weergeven | Extra informatie geven over de klant locatie als hier op wordt geklikt, met daarbij de apparaten die binnen het Auvik netwerk worden beheert. | show.blade.php |
| AUVIK_API_L_REQ-11 | Handmatig netwerk toevoegen | De gebruiker kan handmatig een Auvik netwerk toevoegen bij een klant locatie als dit toepasselijk is. | create.blade.php |

Tabel 10.3 Visualisatie eisen 1

KARAKTERISTIEKEN

In het **Hoofdstuk architectuur** zijn de karakteristieken van de Loader server toegelicht, deze karakteristieken beschrijven de kwaliteiten waar de Loader server naar streeft. Omdat het PoC is ontwikkeld binnen het systeem moet ook dit systeem de karakteristieken zoveel mogelijk naleven. Karakteristieken zijn geen eisen en zijn bedoeld als richtlijnen voor de keuzes die gemaakt zijn. De karakteristieken beschreven zijn 'Functionele volledigheid', 'Schaalbaarheid', 'Herbruikbaarheid', 'Vervangbaarheid', 'Fouttolerantie', 'Herstelbaarheid'. In deze *paragraaf* worden onderdelen van het PoC getoetst aan de hand van de opgestelde karakteristieken.

FUNCTIONELE VOLLEDIGHEID (FUNCTIONAL COMPLETENESS)

Het product moet alle eisen van de *stakeholders* behartigen. Het systeem is afhankelijk van de juiste data en logica om een impact te hebben op de gebruikers ervan. Dit is de beschrijving binnen het *Hoofdstuk 9. Architectuur*.

Proof of Concept

Om de functionele volledigheid van het *PoC* aan te tonen is er gebruik gemaakt van het *Software Requirements Specifications* document, dit is opgesteld met het development team en de *product owner* en bevat alle informatie en eisen voor het systeem. Door het document te volgen en deze functionaliteiten te beoordelen is het mogelijk om aan te tonen dat het *PoC* aan de eisen voldoet.

SCHAALBAARHEID (SCALABILITY)

Het HABdesk product (inclusief de *backend*) moet uiteindelijk kunnen worden verkocht aan externe partijen. Om dit te realiseren moet het product de verschillende hoeveelheden gebruikers aankunnen. Bij de schaalbaarheid van de *backend* heeft gaan het om de hoeveelheid koppelingen die mogelijk ontwikkeld kunnen worden en de bijbehorende data die daarbij komt kijken.

Proof of Concept

De structuur van het *PoC* is opgebouwd met schaalbaarheid in gedachte. Alle Loaders zijn apart van elkaar ontwikkeld en hebben geen directe invloed op elkaar. Zo heeft elk van de Loaders eigen *models*, *views* en *controllers*.

HERBRUIKBAARHEID (REUSABILITY)

Om het ontwikkelproces en de uiteindelijke schaalbaarheid van de *backend* te bevorderen is er een niveau van herbruikbaarheid vereist. Het mag niet voorkomen dat elke nieuwe koppeling volledig vanaf niks ontwikkeld moet worden.

Proof of Concept

De verschillende Loaders delen veel van dezelfde eigenschappen, het gebruik van het *MVC-ontwerppatroon* betekent dat elke Loader gebruik maakt van dezelfde elementen. De *User interface* is ook structureel hetzelfde.

Binnen het *PoC* maakt worden sommige functies meerdere keren gebruikt. De *AuvikTenant* model bevat meerdere definities die op andere plekken aangeroepen worden. *Figuur 10.42 Herbruikbaarheid code* laat verschillende functies zien die herbruikbaar zijn, de functie *children()* kan op elk object met het type *AuvikTenant* worden gebruikt.

```

60  /*
61  |-----
62  | Relations
63  |-----
64  */
65
66 /**
67 * Get a Collection of all children assigned to this model.
68 *
69 * @return \Illuminate\Database\Eloquent\Relations\HasMany
70 */
71 final public function children()
72 {
73     return $this->hasMany(static::class, 'parent_id', 'id');
74 }
75
76 /**
77 * Get a Collection of all clients assigned to this model.
78 *
79 * @return \Illuminate\Database\Eloquent\Relations\HasMany
80 */
81 public function clients()
82 {
83     return $this->children()->where('type', '=', 'client');
84 }
85
86 /**
87 * Get a Collection of all children assigned to this model.
88 *
89 * @return \Illuminate\Database\Eloquent\Relations\HasMany
90 */
91 public function multiClients()
92 {
93     return $this->children()->where('type', '=', 'multiClient');
94 }
95
96 /**
97 * Get the parent of this model.
98 *
99 * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
100 */
101 public function parent()
102 {
103     return $this->belongsTo(static::class, 'parent_id', 'id');
104 }
105
106 /**
107 * Get the location associated with this model.
108 *
109 * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
110 */
111 public function location()
112 {
113     return $this->belongsTo(CustomerLocation::class, 'id', 'auvik_id');
114 }

```

Figuur 10.42 Herbruikbaarheid code

VERVANGBAARHEID (REPLACEABILITY)

Door de separatie van het systeem is het eenvoudiger om delen te vervangen met minimale consequenties. Het mag niet zijn dat een vervanging van één koppeling het gehele systeem buiten werking stelt.

Proof of Concept

Doordat de Auvik API Loader net als de andere Loaders is opgedeeld is het vervangen van onderdelen binnen de Loader eenvoudiger, sinds het geen impact heeft op de code van de andere Loaders.

Het *PoC* maakt gebruik van een *command*, Dit is de code die *jobs* uitvoeren door ze in de *queue* te plaatsen, dit wordt ook wel *dispatch* genoemd. (Laravel - The PHP Framework For Web Artisans, z.d.)

De *AuvikTenantCommand* handelt de *HandleAuvikClientJob()* af. *Figuur 10.43 Vervangbaarheid code* laat een *foreach* loop zien die door alle gespecificeerde *tenants* heengaat en de *job* uitvoert.

```
73 // Start a progress bar.
74 $progress = $this->output->createProgressBar($this->tenants->count());
75
76 // Loop through all specified tenants.
77 foreach ($this->tenants as $tenant) {
78     // Execute the job.
79     try {
80         HandleAuvikClientJob::dispatch($tenant);
81     } catch (\Throwable $th) {
82         $this->output->newLine();
83         $this->output->error("Job for tenant '{$tenant->domain_prefix}' has not been completed");
84     }
85
86     // Advance the progress bar.
87     $progress->advance();
88 }
89
90 // Finish the progress bar.
91 $progress->finish();
92
```

Figuur 10.43 Vervangbaarheid code

Deze logica heeft geen last van eventuele aanpassingen binnen de gebruikte *job*, zelfs al zou de *job* niet meer werken is de code zo ontwikkeld dat er een *error* wordt gegeven waarnaar het systeem zonder verdere problemen doorloopt. Dit concept is zichtbaar in lijnen 79 tot 83, waar de mogelijke *errors* worden opgevangen door een *try-catch* blok dat de *error* incasseert en het proces door laat lopen.

Door gebruik te maken van het *MVC-patroon* en het opdelen van functies volgens het *Single-responsibility principle (SRP)* is de vervangbaarheid van het *PoC* bevorderd.

FOUTTOLERANTIE (FAULT TOLERANCE)

Het systeem moet deze fouten kunnen incasseren en opslaan voor nader onderzoek. Het systeem mag bij onzuivere data niet buiten werking worden gesteld. Omdat data afkomstig is vanuit een API is er het risico dat de API geen zuivere data terugstuurt, met deze mogelijkheden moet rekening worden gehouden.

Proof of Concept

Bij paragraaf vervangbaarheid is aangetoond dat door gebruik te maken van technieken zoals een *try-catch* blok om mogelijke *errors* af te handelen. Binnen het PoC wordt er meer gebruik gemaakt van deze methodes, zoals binnen de *storeAuvikTenantsJob()* toegelicht in *paragraaf storeAuvikTenantsJob* of de *handleAuvikClientJob()* toegelicht in *paragraaf handleAuvikClientJob*. In *Figuur 10.44 Fouttolerantie code* wordt er gebruik gemaakt van een *failsafe* (zie lijn 203 en 227 en 230), een variabele dat ervoor zorgt dat de gebruikte *while* loop niet oneindig door loopt.

```

187  /**
188   * Gets all the parents (MultiClients) of the given $client with these steps:
189   * 1. Get all parents of the client (From lowest to highest)
190   * 2. Save all parents of the client (From highest to lowest)
191   * 3. Save the client and return the parents
192   *
193   * @param \stdClass $client The client used.
194   * @param mixed $tenants All the Auvik tenants.
195   * @return array|null $parents Array with all the parents
196   * @throws \Exception
197   */
198  public function getClientParents(\stdClass $client, $tenants)
199  {
200      $item = $client;
201
202      // Initialized failsafe to stop limit nested parents.
203      $failsafe = 0;
204
205      // Find parent and store them.
206      while (
207          ! is_null($this->getTenantParentId($item))
208      ) {
209          // Get the parent.
210          $items = array_filter(
211              $tenants,
212              function ($tenant) use ($item) {
213                  return $tenant->id == $this->getTenantParentId($item);
214              }
215          );
216
217          // Set the parent to the array.
218          if (! empty($items)) {
219              $parents[] = $items[array_key_first($items)];
220              $item = $items[array_key_first($items)];
221
222          } else {
223              $item->id = null;
224          }
225
226          // Increment the failsafe by one.
227          $failsafe++;
228
229          // Stop the execution if condition is true.
230          if ($failsafe >= 5) break;
231      }
232
233      return $parents ?? [];
234  }

```

Figuur 10.44 Fouttolerantie code

HERSTELBAARHEID (RECOVERABILITY)

Bij het geval van een storing moet het systeem zo spoedig mogelijk weer in productie zijn. Het mag niet gebeuren dat bij een crash alle data binnen het systeem niet is opgeslagen.

Proof of Concept

De data dat wordt verzameld wordt direct opgeslagen binnen de database van de Loader server, verder wordt er niet gebruikt met een lokale opslag. Als de *jobs* falen kan het proces worden voorgezet zoals beschreven in *paragraaf Vervangbaarheid*. Omdat het systeem uit meerdere onderdelen bestaat heeft een *error* in één van deze onderdelen geen directe impact op de rest van het proces, door deze verdeling is het aanpassen en herstellen van onderdelen ook eenvoudiger.

TESTEN

In deze *paragraaf* worden de verschillende manieren toegelicht waarop het *PoC* is gecontroleerd en verbeterd. Er zijn drie methodes toegepast tijdens de ontwikkeling van het *PoC*: *Code reviews*, *API Unit tests* en *Database tests*. Deze *paragraaf* beschrijft deze methodes en licht verbeteringen toe aan de hand van voorbeelden.

CODE REVIEW

Binnen Helmink is er geen officiële vorm van testen die wordt toegepast. Om functies te testen zijn er *code reviews* die wekelijks maar ook op willekeurige momenten plaatsvinden. Tijdens een dergelijke *code review* is de code of functie toegelicht door de verantwoordelijke developer waarnaar andere developers en of de *product owner* commentaar leveren aan de hand van tekeningen of voorbeelden. De documentatie van *code reviews* gebeurt binnen *Github*. *Figuur 10.45 Github pull request* laat een *code review* zien dat heeft plaatsgevonden bij het *mergen* van twee *Github Branches* waarbij de *branche* van het *PoC* wordt samengevoegd met de hoofd *branche* genaamd de *main branche*.

Merged Feature/auvik loader #35
wbpols merged 78 commits into main from feature/auvik-loader 9 days ago

RubenvangemerenHelmink reviewed 10 days ago [View changes](#)

Rubenvangemere... left a comment [Author](#)  ...
First revision Auvik Loader

wbpols requested changes 10 days ago [View changes](#)

- app\Console\Commands\Auvik\AuvikTenantCommand.php Outdated [Show resolved](#)
- app\Http\Controllers\Auvik\AuvikTenantController.php Outdated [Show resolved](#)
- app\Http\Controllers\Customer\CustomerLocationController.php Outdated [Show resolved](#)
- app\Http\Controllers\Customer\CustomerLocationController.php Outdated [Show resolved](#)
- app\Models\Auvik\AuvikTenant.php Outdated [Show resolved](#)
- app\Models\Auvik\AuvikTenant.php Outdated [Show resolved](#)
- app\Jobs\Auvik\HandleAuvikClientJob.php [Show resolved](#)
- app\Models\Auvik\AuvikTenant.php Outdated [Show resolved](#)

wbpols added 5 commits 10 days ago

- Update HandleAuvikClientJob ... d4aff86
- Add feedback when Auvik Tenant job fails in command 3013603
- Removed unused variable from AuvikTenant @index fbda706
- remove short name variable & @getJsonFileContents 653bc1d
- Merge branch 'main' into feature/auvik-loader [Verified](#) fc1e32c

wbpols approved these changes 9 days ago [View changes](#)

wbpols left a comment  ...
Fixed together with @RubenvangemerenHelmink.

wbpols merged commit 9c54157 into main 9 days ago [Revert](#)

Figuur 10.45 Github pull request

In *Figuur 10.45 Github pull request* is te zien dat 'wbpols' als developer feedback heeft achtergelaten op de code die is geschreven. Deze feedback is samen met de uitvoerende partij verbeterd, waarnaar dit is gedocumenteerd. *Code reviews* vinden ook plaats tijdens de dagelijkse datstarts, waarin problemen en oplossingen worden besproken. U wordt verwezen naar *Bijlage 10.- dagstart* voor een uitwerking van een dagstart.

TESTPLAN

Met *code reviews* worden de functionaliteiten en kwaliteiten van het systeem besproken, om dieper in te gaan op de systemen die binnen een product worden gebruikt zijn verschillende methodes beschikbaar om handmatig en automatisch tests uit te voeren.

Voor het *PoC* zijn er twee soorten tests die twee onderdelen van het systeem testen, *API Unit testing* en *Database testing*. Zoals de naam suggereert behandelen deze methodes de Auvik API en de Database van de Loader server. Om het proces te documenteren is er gebruik gemaakt van een testplan, een document dat alle strategieën, methodes en technieken beschrijft die van toepassing zijn. Het doel van een testplan is om te bepalen hoeveel werk er nodig is voor specifieke tests en welke informatie kan worden verzameld. (Hamilton, 2022)

In het testplan van de Auvik API Loader staan de *API Unit tests* en *Database tests* beschreven, samen met deadlines en de onderdelen die wel en niet getest worden. Naast een testplan wordt er ook gebruik gemaakt van *test cases*, dit zijn daadwerkelijke tests die zijn uitgevoerd en bevatten de acties en resultaten van de uitgevoerde test. Voor beide methodes die zijn toegepast is een *test case* opgezet. Binnen de testcases worden de verschillende acties beschreven met het verwachte en daadwerkelijke resultaat.

In *Bijlage 7 - Testplan* staat het Testplan voor het *PoC* uitgewerkt, in *Bijlage 8 - Testcase* en *Bijlage 9 - Testcase* staan de testcases uitgewerkt.

API UNIT TEST

API-unit tests zijn tests gespecialiseerd in het meten van de functionaliteit, prestatie en betrouwbaarheid van een gekozen API. *Unit testing* is een manier om functies binnen een systeem automatisch te testen via verschillende tools, het komt in verschillende soorten en maten. (Hamilton, 2022b)

De *API Unit test* voor de Auvik API Loader bestaat uit een aantal API verzoeken die veel gebruikt worden binnen het *PoC*. Deze verzoeken worden afgehandeld als verschillende functies binnen de test code. Laravel heeft een eigen test module waarmee geautomatiseerde test geschreven kunnen worden. (Laravel - The PHP Framework For Web Artisans, z.d.-b)

Waarom API testing?

API testing is gekozen omdat het gebruik van de Auvik API een essentieel deel uitmaakt van het systeem, daarbij is het begrijpen en testen van deze API belangrijk. Als de limitaties en mogelijkheden van de API bekend kan hieromheen worden gewerkt, wat de betrouwbaarheid van het systeem vergroot.

Resultaten

In *Figuur 10.46 API Test Code* wordt een deel de code van de test weer gegeven. De code bestaat uit het maken van een instantie van de *client* waarnaar het verzoek wordt gemaakt. Het verzoek wordt afgehandeld en op basis van het antwoord wordt de test goed- of afgekeurd. dit is te zien tussen lijnen 50 en 54.

```

12  /**
13   * Collects all Auvik Tenants on endpoint.
14   *
15   * @return void
16   */
17  public function test_can_access_tenants_endpoint()
18  {
19
20
21  $this->client = new \GuzzleHttp\Client([
22      'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
23      'base_uri' => config('services.auvik.base_url'),
24      'headers' => [
25          'Accept' => 'application/json',
26          'Content-Type' => 'application/json',
27          'Encode' => 'UTF-8',
28      ],
29  ]);
30
31
32  try {
33      // Get a response.
34  $response = $this->client->get(
35      'tenants');
36  } catch (\GuzzleHttp\Exception\RequestException $exception) {
37      $response = $exception->getResponse();
38  }
39
40  // Transform the response into a JSON object.
41  $response = json_decode($response->getBody()->getContents());
42
43  // Dump the errors if present.
44  if (isset($response->errors)) {
45      self::assertTrue(false);
46  }
47
48  self::assertFalse(empty($response->data));
49
50  if ($response != null) {
51      self::assertTrue(true);
52  } else {
53      self::assertTrue(false);
54  }
55
56 }
--
```

Figuur 10.46 API Test Code

De resultaten van de test zijn positief, alle *endpoint* die getest zijn, zijn goedgekeurd. Dit is te zien binnen de *terminal* en wordt samen getoond met de tijd die het heeft gekost. *Figuur 10.47 API Test Code resultaat* laat de resultaten zien.

Session contents restored from 13-6-2022 at 17:31:35

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\r.vangemeren\Documents\Loader> php artisan test
Warning: TTY mode is not supported on Windows platform.

    PASS  Tests\Unit\AuvikAPITest
✓ start api testing

    PASS  Tests\Feature\AuvikAPITest
✓ can access tenants endpoint
✓ can access tenants details endpoint
✓ can access tenants devices endpoint
✓ can access tenants device details endpoint

Tests: 5 passed
Time: 8.18s

PS C:\Users\r.vangemeren\Documents\Loader> []
```

Figuur 10.47 API Test Code resultaat

Hoewel deze test de *endpoint* test is binnen het resultaat niet te zien welke data is gestuurd en wat de verdere limitaties zijn, dit moet vanuit een andere testmethode komen. Het resultaat van de *testcase* is te zien in *Tabel 10.4 API testresultaten*. Een uitgebreide toelichting is te vinden in *Bijlage 8. testcase API test*

| Stap # | Beschrijving: | Verwacht resultaat: | Resultaat: | PASS / FAIL / NOT EXECUTED / SUSPENDED |
|---------------|--|----------------------------|-------------------|---|
| 1. | 'can access tenants' test functie | Test PASS | Test voltooit | PASS |
| 2. | 'can access tenants details' test functie | Test PASS | Test voltooit | PASS |
| 3. | 'can access tenants devices' test functie | Test PASS | Test voltooit | PASS |
| 4. | 'can access tenants device details' test functie | Test PASS | Test voltooit | PASS |

Tabel 10.4 API test resultaten

DATABASE TESTING

Database-Testing is een testmethode waarbij de schemas, tabellen en data binnen een database worden getest, door deze methode toe te passen is het mogelijk om de toegang en betrouwbaarheid van data te kunnen garanderen. Het testen gebeurt tijdens het development proces en wordt meestal uitgevoerd door één van de developers. (Hamilton, 2022b)

Waarom Database testing?

database-testing wordt toegepast omdat de database die gebruikt wordt alle belangrijke informatie bevat. Deze informatie wordt in meerdere systemen gebruikt, als deze data niet goed wordt opgeslagen is de integriteit van het gehele HABdesk project aangetast. Bij veel tests krijgen de functies en de *user interface* prioriteit van het testteam, toch moet de data die achter die functies en *user interface* staat correct zijn om deze tests validiteit te geven. (Hamilton, 2022b)

Resultaten

Figuur 10.47 Database test code laat een deel van de code zien die is gebruikt om de verschillende functies te testen, binnen deze code worden door middel van een *foreach* loop alle *tenants* die een koppeling hebben met een klant locatie opgehaald en geteld.

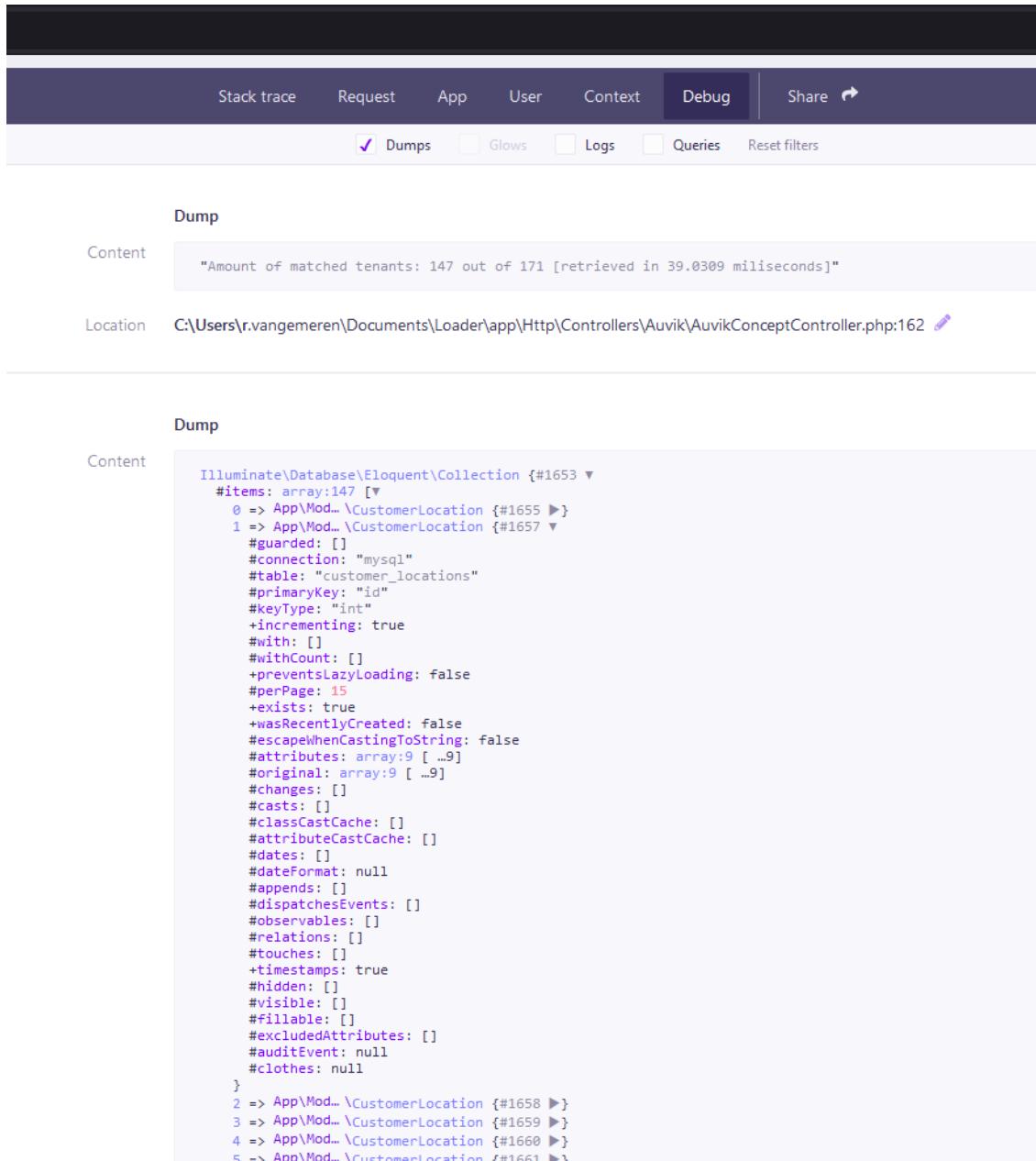
```

133 // Get all unmatched locations and attempted to match them
134 public function getMatches()
135 {
136
137     $start = hrttime(true);
138
139     $locations = \App\Models\Customer\CustomerLocation::whereNull('auvik_id')->get();
140
141     foreach ($locations as $location) {
142
143         $file = $location->getJsonFileContents();
144
145         $match = AuvikTenant::firstWhere('domain_prefix', '=', $file->short_name);
146
147         $matches[] = $location->auvikTenant()->associate($match);
148
149         $location->save();
150     }
151
152     $matched = \App\Models\Customer\CustomerLocation::whereNotNull('auvik_id')->get();
153
154     $total = \App\Models\Customer\CustomerLocation::all()->count();
155
156     $count = $matched->count();
157
158     $end = hrtime(true);
159
160     $miliSeconds = ($end - $start) / 1000000;
161
162     ddd("Amount of matched tenants: $count out of $total [retrieved in $miliSeconds milliseconds]", $matched);
163
164 }
```

Figuur 10.47 Database test code

De code kan worden uitgevoerd door het *PoC* te starten en naar de volgende URL te gaan 'auvik/test'. Deze URL is door de gehele ontwikkeling van het *PoC* gebruikt om functies te testen. *Figuur 10.48 Database test resultaat* laat zien wat voor resultaat de test van *Figuur 10.47 Database test code* is. Bovenaan staat wat het resultaat hoort te zijn en hoe snel het verzoek is voltooit, daaronder staat het daadwerkelijke

antwoord in de vorm van alle data die is opgehaald. Om de test goed te keuren om de data handmatig worden gecontroleerd.



The screenshot shows a user interface for viewing database dump results. At the top, there are tabs for Stack trace, Request, App, User, Context, Debug (which is selected), and Share. Below the tabs are filter options: Dumps (checked), Glows, Logs, Queries, and Reset filters. The main area is titled 'Dump' and contains two sections: 'Content' and 'Location'. The 'Content' section displays the following PHP code:

```

Content
Dumps
Logs
Queries
Reset filters

Dump
Content
"Amount of matched tenants: 147 out of 171 [retrieved in 39.0309 milliseconds]"

Location C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:162

```

The 'Content' section also displays a large block of PHP code representing a collection of customer locations:

```

Illuminate\Database\Eloquent\Collection {#1653 ▶
  #items: array:147 [▼
    0 => App\Mod.. \CustomerLocation {#1655 ▶}
    1 => App\Mod.. \CustomerLocation {#1657 ▶}
    #guarded: []
    #connection: "mysql"
    #table: "customer_locations"
    #primaryKey: "id"
    #keyType: "int"
    +incrementing: true
    #with: []
    #withCount: []
    +preventsLazyLoading: false
    #perPage: 15
    +exists: true
    +wasRecentlyCreated: false
    #escapeWhenCastingToString: false
    #attributes: array:9 [ ...9]
    #original: array:9 [ ...9]
    #changes: []
    #casts: []
    #classCastCache: []
    #attributeCastCache: []
    #dates: []
    #dateFormat: null
    #appends: []
    #dispatchesEvents: []
    #observables: []
    #relations: []
    #touches: []
    +timestamps: true
    #hidden: []
    #visible: []
    #fillable: []
    #excludedAttributes: []
    #auditEvent: null
    #clothes: null
  ]
  2 => App\Mod.. \CustomerLocation {#1658 ▶}
  3 => App\Mod.. \CustomerLocation {#1659 ▶}
  4 => App\Mod.. \CustomerLocation {#1660 ▶}
  5 => App\Mod.. \CustomerLocation {#1661 ▶}
}

```

Figuur 10.48 Database test resultaat

De resultaten van de database test zijn positief, op één test na zijn alle tests goedgekeurd, daarbij vallen de tijden die zijn opgenomen binnen de prestatie-eisen beschreven het *Hoofdstuk 6. requirements*. *Tabel 10.5 Database test resultaat* laat de resultaten van de database test zien, een uitgebreide toelichting is te vinden in *Bijlage 9. testcase Database test*.

| Stap # | Beschrijving: | Verwacht resultaat: | Resultaat: | PASS / FAIL / NOT EXECUTED / SUSPENDED |
|--------|----------------------------|--|---|--|
| 1 | 'getAllTenants' functie | Alle <i>tenants</i> vanuit de database. | SUCCES [in 4.8 milliseconden] | PASS |
| 2 | 'getTenantDetail' functie | Alle details van alle <i>tenants</i> vanuit database. | SUCCES [in 1354 milliseconden/ 1.35 seconden] | PASS |
| 3 | 'getAllClients' functie | Alle <i>tenants</i> van het type <i>client</i> . | SUCCES [in 5.3 milliseconden] | PASS |
| 4 | 'getAllMultiClients' | Alle <i>tenants</i> van het type <i>multiClient</i> . | SUCCES [in 5.2 milliseconden] | PASS |
| 5 | 'getMatches' functie | Alle <i>tenants</i> met een koppeling naar een klant locatie. | SUCCES [in 5.3 milliseconden] | PASS |
| 6 | 'getUnMatched' functie | Alle <i>tenants</i> zonder een koppeling naar een klant locatie. | SUCCES [in 30 milliseconden] | PASS |
| 7 | 'getDevice' functie | Alle apparaten van een willekeurige <i>tenant</i> . | SUCCES [in 1646 milliseconden/ 1.64 seconden] | PASS |
| 8 | 'getLocationMatch' functie | Klant locatie met de locatie matches van de <i>tenant</i> . | NIET UITGEVOERD | NOT EXECUTED |

Tabel 10.5 Database test resultaat

In *Tabel 10.5 Database test resultaat* is te zien dat één van de test niet is uitgevoerd, dit komt omdat de functie niet compleet was en functioneel niet kon worden getest. Voor een volgende test is het duidelijk wat er niet goed is gegaan en kan de functie wel worden getest, het testen van de *getLocationMatch* functie valt buiten in de onderzoeksperiode.

11. CONCLUSIE

Tijdens de loop van het onderzoek is gekeken naar de architectuur van het HABdesk systeem en hoe dit systeem ontwikkeld kan worden om de werkdruk van de medewerkers van Helmink te verlichten, hierbij is de volgende onderzoeksraag onderzocht: **"Hoe kan de backend van het HABdesk project binnen Helmink worden ontwikkeld om de werkdruk van Helmink te verlichten?"**

Uit onderzoek naar de huidige situatie is gebleken dat Helmink een klein development team heeft. Het HABdesk project is een samenvoeging van data binnen een web portaal en wordt volledig intern ontwikkeld, het project is één van de meerdere verantwoordelijkheden van het development team. Het team is gespecialiseerd in webapplicaties.

Het documenteren van werkzaamheden is een organisatie breed probleem dat voor veel onduidelijkheid zorgt.

De *product owner* van onder andere het HABdesk project bestaat uit alleen de directeur van het bedrijf, dit zorgt dagelijks voor oponthoud, omdat hij een drukke agenda heeft.

Aan de hand van een *stakeholdersanalyse* zijn de software eisen opgesteld van twee onderdelen van de *backend* van het HABdesk project, deze onderdelen koppelen de drie systemen verantwoordelijk voor het beheren van de klantgegevens, CRIS-X, IT-Glue en Auvik.

In het theoretisch onderzoek zijn software architectuur en software integraties als velden binnen de IT onderzocht. Uit deze twee onderzoeken zijn een aantal observaties gedaan:

- Binnen het veld software architectuur gaat alles om compromis, geen enkele oplossing is ooit perfect.
- De taken van een software architect zijn het beheren van het development proces en een goede communicatie met de *product owner(s)* onderhouden.
- Er zijn veel verschillende structuren die opgedeeld kunnen worden in twee categorieën: centraal (*monolithic*) en decentraal (*distributed*).
 - Een centraal systeem is makkelijker te ontwikkelen maar mist een niveau van schaalbaarheid.
 - En decentraal systeem heeft een complexere implementatie maar heeft potentieel een hogere schaalbaarheid en prestatie.
- Er zijn verschillende soort integratie methodes met ieder zijn voor- en nadelen.

Vanuit de huidige situatie en het theoretisch onderzoek is advies gedaan naar Helmink. Naast operationele adviezen is geadviseerd om gebruik te maken van een decentrale (*distributed*) architectuur in plaats van de huidige centrale architectuur (*monolithic*). Een decentrale architectuur biedt de voordelen die Helmink nodig heeft om het HABdesk project in de toekomst te kunnen verkopen.

Het is echter niet wenselijk om van architectuur direct te veranderen, de huidige visie voor het project moet eerst afgerond zijn. Om naar een nieuwe architectuur te

werken moet een transitie plaatsvinden waar de huidige structuur in kleine stappen aangepast wordt.

De werkdruk van Helmink wordt verlicht door het koppelen van de verschillende systemen en deze data overzichtelijk op één plek weer te geven. Het *Proof of Concept* bereikt dit doel door data van verschillende bronnen samen te voegen op één pagina, daarbij wordt de data verrijkt door links en kleuren.

Na dit onderzoek kan Helmink door met het huidige systeem en in kleine stappen werken naar een architectuur die de lange termijnvisie kan behartigen.

12. DISCUSSIE

In het hoofdstuk Discussie wordt besproken in hoeverre het onderzoek anders zou zijn gelopen als dit door een andere uitvoerende partij zou zijn gedaan. Deze besprekking gebeurt in drie delen. De huidige situatie analyseren, het geven van advies op basis van theoretisch onderzoek en het implementeren van een *Proof of Concept* of andere realisatie.

Het onderzoek naar de huidige situatie komt voort uit observaties en gesprekken die in een periode van meerdere maanden zijn gedaan, hierdoor is een goed beeld ontstaan van de huidige situatie van Helmink en het HABdesk project. Soortgelijke conclusies zouden worden getrokken, mocht het onderzoek zijn uitgevoerd door een ander. Dit hangt deels af van het niveau van initiatief en grondigheid die de uitvoerende partij toont.

Het theoretisch onderzoek naar software architectuur en software integraties is gevormd door de huidige kennis vanuit de uitvoerende partij. Mocht het onderzoek uit zijn gevoerd door een ander kan deze uitgebreider en met meer toelichting worden verricht. Hoewel de adviezen naar Helmink meer op één lijn zouden liggen, is het bij meer kennis van het vakgebied mogelijk om extra concreet te zijn over de volgende stap van het project.

Als er meer technische kennis is over webapplicaties en *PHP*, is het mogelijk om een volledig apart project te ontwikkelen. Hierin kunnen gekozen architecturen getest worden met data van Helmink. Voor een implementatie van de Auvik API Loader is er weinig verschil tussen de huidige implementatie en die van een ander, omdat de huidige implementatie functioneel voldoet aan de opgestelde eisen. Het kan zijn dat de scope van het *Proof of Concept* is uitgebreid om de technische kennis van een andere uitvoerende partij te reflecteren.

13. NAWORD

Deze scriptie markeert het einde van mijn vierjarige hbo-opleiding Informatica van de Hogeschool Rotterdam. In deze vier jaar ben ik van complete beginner op het gebied van programmeren gegroeid tot een software developer, die kan beginnen aan zijn carrière.

Tijdens de afstudeerperiode heb ik meer dan vier maanden dagelijks mogen meewerken aan de projecten binnen Helmink. In deze periode heb ik veel geleerd op technisch niveau maar ook op een operationeel en sociaal niveau.

Zo heb ik geleerd dat hoewel ik goede ideeën kan hebben, deze ideeën niks waard zijn, als ik niet mijn podium pak om ze voor te stellen. Het vragen om hulp heb ik nooit makkelijk gevonden, maar binnen Helmink was ik geforceerd om initiatief te nemen, waar ik veel van heb geleerd.

Op technisch gebied heb ik kennis opgedaan over software architectuur en integraties, zoveel zelfs dat ik na mijn opleiding dieper op deze concepten in wil gaan. Ik heb veel concepten geleerd zoals *Enterprise Service Bus* en *iPaaS*. Ik heb met het development team veel tijd besteed aan *PHP* en *Laravel*.

Met de kennis van nu zou ik een aantal dingen anders hebben aangepakt. Ten eerste zou ik de scope van het project duidelijker definiëren, als de scope goed is, worden veel van de andere besluiten een stuk makkelijker. Nu moest ik mijn onderzoek doen en tegelijkertijd bedenken wat mijn scope zou zijn, wat voor veel hoofdpijn heeft gezorgd.

Een volgende keer wil ik meer gebruik maken van de kennis om mij heen, van bijvoorbeeld andere developers of leraren. Ik heb de neiging om alleen te kijken naar boeken en onderzoeken, terwijl een simpel gesprek met iemand vaak net zoveel interessante onderwerpen kan bevatten.

Voor de planning wordt u verwezen naar Bijlage 13.

Barendrecht, 2022

Ruben van Gemeren

14. VERWIJZINGEN

5. HUIDIGE SITUATIE

Codecademy. (z.d.). *MVC: Model, View, Controller*. Geraadpleegd op 20 februari 2022, van <https://www.codecademy.com/article/mvc>

Ellis, J. (2019, 21 oktober). *Command Line Interface*. Comms InfoZone. Geraadpleegd op 20 februari 2022, van <https://www.comms-express.com/infozone/article/command-lineinterface/#:%7E:text=Abbreviated%20as%20CLI%2C%20a%20Command,visual%20prompt%20from%20the%20computer>.

online marketing agency. (2021, 2 september). *Wat is een User interface?* Geraadpleegd op 21 februari 2022, van <https://onlinemarketingagency.nl/marketingtermen/user-interface/>

Provost, N. (1999, 28 april). *Bcrypt Algorithm*. usenix. Geraadpleegd op 21 februari 2022, van https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/provos/provos_html/node5.html#:~:text=The%20problems%20present%20in%20traditional,expensive%20key%20setup%20in%20eksblowfish.

Wikipedia-bijdragers. (2021, 1 maart). *Model-view-controller-model*. Wikipedia. Geraadpleegd op 20 februari 2022, van <https://nl.wikipedia.org/wiki/Model-view-controller-model>

6. REQUIREMENTS

Bijvank, S. (z.d.). Stakeholderanalyse. House-of-control. Geraadpleegd op 12 mei 2022, van <https://www.house-of-control.nl/stakeholder-analyse-management.html>

Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., & Theofanos, M. (1993). Identifying and measuring quality in a software requirements specification. [1993] Proceedings First International Software Metrics Symposium.

<https://doi.org/10.1109/metric.1993.263792>

Pataki, G. E. (2001). NYS Project Management Guidebook (2de editie). New York State Office for Technology.

7. THEORETISCH KADER

Brown, S. (z.d.). *The C4 model for visualising software architecture*. C4Model. Geraadpleegd op 20 februari 2022, van <https://c4model.com/>

CaixaBank obtains the ISO/IEC 25000 Functional Suitability certificate for their app CaixaBankNow. (z.d.). <Https://Iso25000.Com/>. Geraadpleegd op 13 mei 2022, van <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=6>

Editor. (2020, 16 juni). *Software Architect Role, Skills, and Impact on Product Success*. AltexSoft. Geraadpleegd op 22 april 2022, van <https://www.altexsoft.com/blog/software-architect-role/>

Gartner. (z.d.). *Definition of Integration - Gartner Information Technology Glossary*.

Geraadpleegd op 4 mei 2022, van <https://www.gartner.com/en/information-technology/glossary/integration>

Henderson, B. C. (2020, 24 juli). *System Integration Process- 7 Steps to Follow*.

Hubworks. Geraadpleegd op 15 mei 2022, van <https://anyconnector.com/system-integration/system-integration-process.html>

ISO 27001. (z.d.). *ISO/IEC 27001*. ISO. Geraadpleegd op 2 mei 2022, van <https://www.iso.org/isoiec-27001-information-security.html>

Petrova, S. (2019, 18 januari). *Adopting Agile: The Latest Reports About The Popular Mindset*. Adeva. Geraadpleegd op 20 mei 2022, van <https://adevait.com/blog/remote-work/adopting-agile-the-latest-reports-about-the-popular-mindset>

RahimSoomro, T., & Hasnain Awan, A. (2012). Challenges and Future of Enterprise Application Integration. *International Journal of Computer Applications*, 42(7), 42–45. <https://doi.org/10.5120/5707-7762>

Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture*. Van Duuren Media.

Shrivastava, S., & Srivastav, N. (2022). *Solutions Architect's Handbook* (2de editie).

Van Haren Publishing.

Software Architect Job Description [Updated for 2022]. (z.d.). Software Architect Job Description: Top Duties and Qualifications. Geraadpleegd op 10 mei 2022, van <https://www.indeed.com/hire/job-description/software-architect>

Van der Linde, M. (z.d.). *SWOT analyse voorbeeld / Handig voorbeeld voor je SWOT analyse.* Strategischmarketingplan.com. Geraadpleegd op 10 mei 2022, van [https://www.strategischmarketingplan.com/swot-analyse/swot-analyse-vorbeeld/#:%7E:text=Situatieanalyse%20maken&text=Bij%20de%20situatieanalyse%20onderzoek%20je,zwaktes%2C%20bedreigingen%20en%20kansen%20op](https://www.strategischmarketingplan.com/swot-analyse/swot-analyse-voorbeeld/#:%7E:text=Situatieanalyse%20maken&text=Bij%20de%20situatieanalyse%20onderzoek%20je,zwaktes%2C%20bedreigingen%20en%20kansen%20op)

Wikipedia contributors. (2022, 6 mei). *Coupling (computer programming)*. Wikipedia. Geraadpleegd op 10 mei 2022, van

[https://en.wikipedia.org/wiki/Coupling_\(computer_programming\)](https://en.wikipedia.org/wiki/Coupling_(computer_programming))

8. ADVIES

Abels, L. (2021, 29 september). *The ultimate ESB development team: brilliant and stubborn.* Yenlo. Geraadpleegd op 13 mei 2022, van

<https://www.yenlo.com/blogs/the-ultimate-esb-development-team/>

Cherubini, M., Venolia, G., DeLine, R., & Ko, A. J. (2007). Let's go to the whiteboard. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* <https://doi.org/10.1145/1240624.1240714>

Johnston, T. (2017, 29 juni). *Enterprise Service Bus (ESB) Tools: Technical Comparison and Review*. Shadow-Soft. Geraadpleegd op 12 mei 2022, van <https://shadow-soft.com/enterprise-service-bus-esb-tools/#:%7E:text=Pricing%20starts%20at%20%2423%2C760%20for,support%20during%20normal%20business%20hours>

Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture*. Van Duuren Media.

Software as a Service (SaaS) Market Size, Share & COVID-19 Impact Analysis, By Deployment Type (Public Cloud, Private Cloud, and Hybrid Cloud), By Application (Customer Relationship Management, Enterprise Resource Planning, Content, Collaboration & Communication, Business Intelligence & Analytics, Human Capital Management, and Others), By Industry (BFSI, Retail & Consumer Goods, Healthcare, Education, Manufacturing, Travel & Hospitality, and Others) and Regional Forecast, 2021–2028. (z.d.). Fortunebusinessinsights. Geraadpleegd op 15 mei 2022, van <https://www.fortunebusinessinsights.com/software-as-a-service-saas-market-102222>

Supernor, B. (2018, 25 januari). *Why the cost of cloud computing is dropping dramatically*. App Developer Magazine. Geraadpleegd op 15 mei 2022, van <https://appdevelopermagazine.com/why-the-cost-of-cloud-computing-is-dropping-dramatically/>

8. ARCHITECTUUR

Asprino, L., Ciancarini, P., Nuzzolese, A. G., Presutti, V., & Russo, A. (2022).

A reference architecture for social robots. *Journal of Web Semantics*, 72, 100683.

<https://doi.org/10.1016/j.websem.2021.100683>

Brown, S. (z.d.). *The C4 model for visualising software architecture*. C4model.

Geraadpleegd op 10 maart 2022, van <https://c4model.com/>

Done, S. (2021). *TOGAF® Simplified: A Complete Guide To TOGAF® 9.2*

Certification. Advantage.

Hernandez, R. D. (2021, 20 april). *The Model View Controller Pattern – MVC Architecture and Frameworks Explained*. freeCodeCamp.Org. Geraadpleegd op 10 juni 2022, van <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/>

Laravel. (z.d.). *Laravel - The PHP Framework For Web Artisans*.

Geraadpleegd op 12 juni 2022, van <https://laravel.com/docs/9.x/structure>

MVC - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. (2022, 18 februari). Mozilla. Geraadpleegd op 10 juni 2022, van <https://developer.mozilla.org/en-US/docs/Glossary/MVC>

What is a Template Engine? (z.d.). Treehouse. Geraadpleegd op 10 juni 2022, van <https://teamtreehouse.com/library/what-is-a-template-engine#notes>

What is Sequence Diagram? (z.d.). Visual-Paradigm. Geraadpleegd op 12 juni 2022, van <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/>

10. IMPLEMENTATIE

Arora, G. K. (2017). *Solid Principles Succinctly*. Van Haren Publishing.

Auvik. (z.d.). *Auvik API*. Geraadpleegd op 1 mei 2022, van <https://auvikapi.us1.my.auvik.com/docs#section/Authentication>

Babu, R. (2021, 21 mei). *DELETE CASCADE and UPDATE CASCADE in SQL Server foreign key*. SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More. Geraadpleegd op 12 juni 2022, van <https://www.sqlshack.com/delete-cascade-and-update-cascade-in-sql-server-foreign-key/>

Baghel, A. S. (2019, 11 september). *Software Design Principles DRY and KISS*. Dzone.Com. Geraadpleegd op 12 juni 2022, van <https://dzone.com/articles/software-design-principles-dry-and-kiss>

Daniel, D. (2021, 13 mei). *Kaizen (continuous improvement)*. SearchERP. Geraadpleegd op 10 juni 2022, van <https://www.techtarget.com/searcherp/definition/kaizen-or-continuous-improvement>

Divine, P. (2019, 19 juli). *Case Styles: Camel, Pascal, Snake, and Kebab Case - Better Programming*. Medium. Geraadpleegd op 13 juni 2022, van <https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>

Doherty, E. (z.d.). *What is object-oriented programming? OOP explained in depth*. Educative: Interactive Courses for Software Developers. Geraadpleegd op 11 juni 2022, van <https://www.educative.io/blog/object-oriented-programming>
Getting Started / Axios Docs. (z.d.). Axios. Geraadpleegd op 15 juni 2022, van <https://axios-http.com/docs/intro>

Hamilton, T. (2022a, april 16). *TEST PLAN: What is, How to Create (with Example)*. Guru99. Geraadpleegd op 12 juni 2022, van <https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html>

Hamilton, T. (2022b, april 19). *Database (Data) Testing Tutorial with Sample Test Cases*. Guru99. Geraadpleegd op 11 juni 2022, van <https://www.guru99.com/data-testing.html>

Hamilton, T. (2022c, april 30). *API Testing Tutorial: What is API Test Automation? How to Test*. Guru99. Geraadpleegd op 10 juni 2022, van <https://www.guru99.com/api-testing.html>

How To Make a Modal Box With CSS and JavaScript. (z.d.). W3Schools. Geraadpleegd op 15 juni 2022, van https://www.w3schools.com/howto/howto_css_modals.asp

An Introduction to GitHub. (2020, 18 juni). Digital.Gov. Geraadpleegd op 10 juni 2022, van <https://digital.gov/resources/an-introduction-github/>

Janssen, T. (2021, 5 april). *SOLID Design Principles Explained: The Single Responsibility Principle*. Stackify. Geraadpleegd op 13 juni 2022, van <https://stackify.com/solid-design-principles/#:%7E:text=The%20single%20responsibility%20principle%20provides,provide%20a%20solution%20for%20everything>

Kamalizade, A. (2021, 13 december). *Why You Should Write Small Git Commits - Better Programming*. Medium. Geraadpleegd op 10 juni 2022, van <https://betterprogramming.pub/why-you-should-write-small-git-commits-c9a042737aa6>

Laravel. (z.d.). *Laravel - The PHP Framework For Web Artisans*. Geraadpleegd op 13 juni 2022, van <https://laravel.com/docs/5.6/eloquent-relationships#inserting-and-updating-related-models>

Laravel - The PHP Framework For Web Artisans. (z.d.-a). Laravel. Geraadpleegd op 15 juni 2022, van <https://laravel.com/docs/9.x/queues#generating-job-classes>

Laravel - The PHP Framework For Web Artisans. (z.d.-b). Laravel. Geraadpleegd op 10 juni 2022, van <https://laravel.com/docs/9.x/testing>

Martin, B. (z.d.). *Clean Code*. cleancoder. Geraadpleegd op 11 juni 2022, van <http://cleancoder.com/products>

Oloruntoba, S. (2021, 30 november). *SOLID: The First 5 Principles of Object Oriented Design*. DigitalOcean Community. Geraadpleegd op 11 juni 2022, van https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#single-responsibility-principle

PHP compact() Function. (z.d.). W3Schools. Geraadpleegd op 15 juni 2022, van https://www.w3schools.com/php/func_array_compact.asp

PHPDocumenter. (z.d.). *phpDocumentor*. Geraadpleegd op 13 juni 2022, van <https://docs.phpdoc.org/guide/guides/docblocks.html>

RhodeCode > Version Control Systems Popularity in 2016. (2016, 1 december). RhodeCode. Geraadpleegd op 10 juni 2022, van <https://rhodecode.com/insights/version-control-systems-2016#:~:text=To%20sum%20this%20up%3A,Mozilla%2C%20Nginx%2C%20and%20NetBeans>

SQL - Foreign Key. (z.d.). Tutorialspoint. Geraadpleegd op 11 juni 2022, van <https://www.tutorialspoint.com/sql/sql-foreign-key.htm>

SQL PRIMARY KEY Constraint. (z.d.). w3Schools. Geraadpleegd op 11 juni 2022,
van https://www.w3schools.com/sql/sql_primarykey.ASP

BIJLAGE 1: SWOT-ANALYSE

Een SWOT-Analyse (ook een sterke-zwakteanalyse) is een overzichtelijke manier om de sterke en zwakke punten van een organisatie weer te geven en te categoriseren. Aan de hand van vier onderdelen (Sterktes, zwaktes, kansen en bedreigingen) wordt er een tabel gevormd. De tabel kan gebruikt worden als overzicht en samenvatting van een uitgebreidere analyse van de organisatie.

De SWOT-Analyse is ontstaan rond de jaren 60 tijdens een congres van het Stanford Research Institute onder leiding van Albert Humphrey. In deze tijd was er bij grote bedrijven een behoefte aan een lange termijnplanning dat eenvoudig uit te voeren en makkelijk te onderhouden was. Meneer Humphrey en zijn team hebben de SWOT-Analyse waarnaar het in de afgelopen 60 jaar steeds populairder is geworden. Nu is de SWOT-Analyse een populair middel om een overzicht te creëren van organisaties, processen, producten, financiën of administratie.

DE MISSIE

Helmink heeft het HABdesk project opgestart om een probleem op te lossen waar veel bedrijven moeite mee hebben, het beheren en toepasbaar stellen van data die uit verschillende bronnen wordt geleverd. Bij het leveren van support of het interne beheer van data is het erg onhandig om alles verspreid te hebben in verschillende portalen. Elke portaal is anders ingericht en is ontworpen met een andere filosofie, hierdoor wordt er veel tijd verspild aan het opzoeken en uitzoeken van de juiste data. De support die Helmink levert neemt veel van de werktijd in beslag, het is daarbij een belangrijk onderdeel waar Helmink zich graag in wilt optimaliseren.

In de huidige situatie kan het erg lastig zijn voor Helmink om problemen te kunnen identificeren. Dit komt vooral door de manier waarop gebruikersdata wordt opgeslagen en up to date wordt gehouden. Door de afwezigheid van automatisatie wordt data vaak niet (genoeg) bijgehouden.

HABdesk moet uiteindelijk een oplossing bieden. Met HABdesk wilt Helmink een product ontwikkelen dat intern het supportproces versoepelt en dat door klanten gebruikt kan worden als een self service portaal.

SWOT-ANALYSE

Binnen de analyse zelf worden er positieve en negatieve aspecten van het project beschreven, daarnaast worden de onderdelen verdeeld in intern en extern.

STERKTEN

- Ervaring binnen IT-Communicatie branche
- Kennis van webdevelopment
- Veel contact tussen team
- Veel controle over eigen development omgeving

ZWAKTES

- Relatief klein development team
 - Te kort aan medewerkers
 - Werkdruk vanuit andere projecten die progressie belemmeren
 - Weinig ervaring met projectmanagement
-

KANSEN

- Betere ontwikkeling van API's van portalen
 - Verkleinen van de scope
 - Gebruik maken van externe pakketten
 - De groeiende iPaaS markt
-

BEDREIGINGEN

- Aanpassingen binnen API's
- Nieuwe klanten met andere systemen die niet kunnen worden geïntegreerd
- De scope te groot maken waardoor voortgang wordt vertraagd

[RUBEN] SWOT analyse HABdesk

Ruben van Gemeren | May 10, 2022

| | | Positief | Negatief |
|--------|----------|--|---|
| | | Sterkten | Zwaktes |
| Intern | Positief | <ul style="list-style-type: none">Ervaring binnen IT-Communicatie brancheKennis van webdevelopmentVeel contact tussen teamVeel controle over eigen development omgeving | <ul style="list-style-type: none">Relatief klein development teamTe kort aan medewerkersWerkdruk vanuit andere projecten die progressie belemmerenWeinig ervaring met project management |
| | Negatief | <ul style="list-style-type: none">Betere ontwikkeling van API's van portalenVerkleinen van de scopeGebruik maken van externe pakkettenDe groeiende iPaaS markt | <ul style="list-style-type: none">Aanpassingen binnen API'sNieuwe klanten met andere systemen die niet kunnen worden geïntegreerdDe scope te groot maken waardoor voortgang wordt vertraagd |

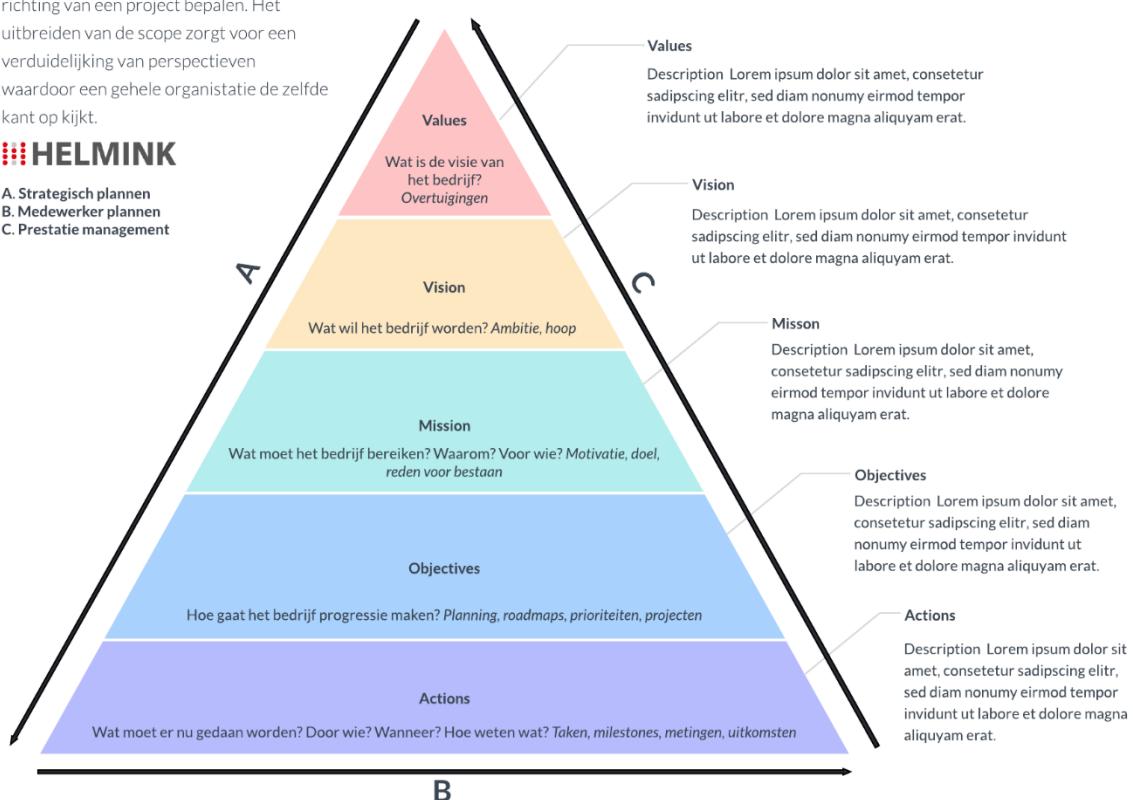
BIJLAGE 2: HOSIN KANRI MODEL

Hoshin Kanri Model

Een model van de 5 niveaus die de visie en richting van een project bepalen. Het uitbreiden van de scope zorgt voor een verduidelijking van perspectieven waardoor een gehele organisatie de zelfde kant op kijkt.

HELMINK

- A. Strategisch plannen
- B. Medewerker plannen
- C. Prestatie management



BIJLAGE 3: ONDERZOEK SOFTWARE ARCHITECTUUR

SOFTWARE EN ARCHITECTUUR

INLEIDING

Als het gaat om software development wordt er vooral aandacht besteed naar de functionaliteiten, vaak *Features* genoemd. Als er een nieuwe applicatie wordt uitgebracht gaat het om wat deze applicatie allemaal kan en welke trendy technologie er is gebruikt om dit te bereiken. Termen zoals *Cloud*, *Artificial Intelligence* en *Internet of Things (IoT)* worden gebruikt om mogelijke klanten enthousiast te maken over een nieuw product. Achter deze termen is niks meer dan code, heel veel code. Het idee dat nieuwe technologieën een soort magie zijn die alles binnen de sector veranderen is een beeld dat grote bedrijven graag willen schetsen, maar achter de schermen worden vaak veel van dezelfde principes toegepast.

Ook al lijkt de wereld steeds moderner te worden hebben de systemen vaak last van dezelfde valkuilen en limitaties. Het ontwikkelen van een goede architectuur kan deze valkuilen vermijden.

De architectuur van een software systeem is een grondlegging dat alle verdere beslissingen over dat systeem beïnvloed. Net als bij de grondlegging van een huis is het een basis dat moet worden uitgewerkt voordat er daadwerkelijk gebouwd kan worden aan het systeem. Het zijn van een software architect wordt gezien als een hoogwaardige positie waar veel kennis voor nodig is, desondanks is het niet duidelijk wat een software architect nu daadwerkelijk doet en wat voor problemen er moeten worden opgelost.

SOFTWARE ARCHITECTURE

Het definiëren van de term *Software Architecture* is net zo moeilijk als het ontwerpen van de architectuur zelf, verschillende partijen hebben een andere kijk op de term. Het boek "*Fundamentals of Software Architecture*" geschreven door meneer Richards en meneer Ford probeert de term te beschrijven aan de hand van vier kenmerken die verschillende onderdelen van een architectuur categoriseren.

ARCHITECTURE STRUCTURE

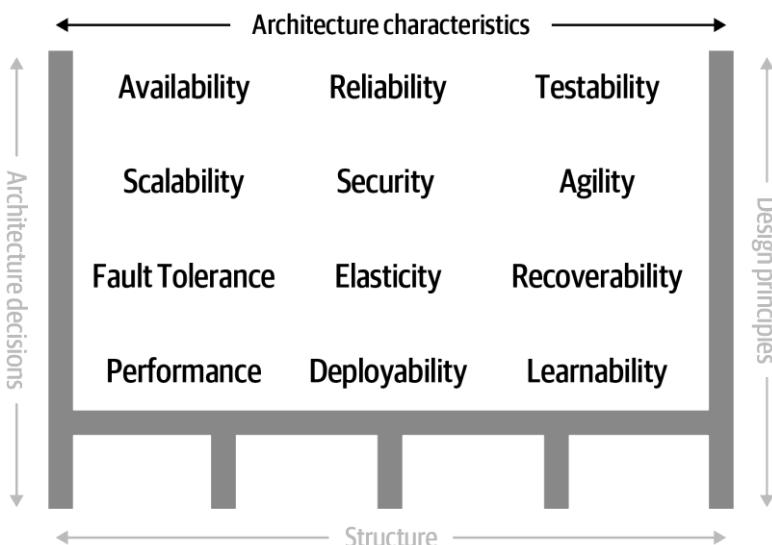
Bij de structuur wordt er verwezen naar de soort architectuur die wordt toegepast. Voorbeelden van structuren zijn *Layered Architecture*, *Event-Driven Architecture*, *Microkernel Architecture* en *Microservices Architecture*.

Hoewel de structuur van een architectuur een duidelijk beeld geeft van de globale ontwerpkeuzes die door het team gemaakt zijn, is het niet volledig. Het is niet te achterhalen of een gekozen structuur wel een optimale keuze is voor het probleem dat opgelost moet worden.

ARCHITECTURE CHARACTERISTICS

De karakteristieken (Characteristics) van een architectuur beschrijven de succesfactoren die niet direct een functionaliteit beschrijven. De termen die ook vaak worden gebruikt zijn *non-functional requirements* of *Quality attributes*.

Bij **Figuur** worden een aantal voorbeelden gegeven. Op het eerste gezicht lijken deze beschrijvingen triviaal, een soort samenvatting van wensen. In de realiteit zijn dit de blokken waarop een product wordt ontwikkeld. Door het hoge niveau van abstractie kunnen de termen gebruikt worden bij alle soorten systemen en technologieën.



ARCHITECTURE DECISIONS

Bij de *decisions* worden de regels en beperkingen van een systeem geformuleerd. Deze worden gebruikt door het development team om de stroom van data uit te werken aan de hand van de opgezette systeemeisen. Waar componenten wel of geen toegang tot hebben en hoe de communicatie wordt opgezet zijn onderdelen die in de *Architecture decisions*.

DESIGN PRINCIPLES

De termen *Design principles* en *Architecture decisions* lijken veel op elkaar, toch hebben ze een ander doel. Waar een *decision* een vaste regel is waaraan voldaan moet worden, is een *design principle* meer een richtlijn. *Design principles* hebben vaak een grotere scope en een hogere abstractie.

Als voorbeeld: 'Waar mogelijk, maak gebruik van nested API calls om relaties tussen klantdata zuiver te houden en het aantal API calls te verminderen.'

Bij deze *design principle* is er niet gespecificeerd waar er gebruik gemaakt moet worden van *nested API calls* en dat het überhaupt moet worden geïmplementeerd. Het idee is om een richtlijn te bieden waar het development team keuzes op kan baseren, de implementatie van deze richtlijnen wordt besloten door het team dat het product ontwikkeld.

Tijdens het ontwikkelen van een systeem spelen deze vier kenmerken een grote rol voor het bepalen van de juiste architectuur. Om de architectuur van een systeem te kunnen realiseren en controleren moeten deze kenmerken worden uitgewerkt en gebruikt worden tijdens het testen van het desbetreffende systeem.

DE WETTEN VAN SOFTWARE ARCHITECTURE

Het concept *software architecture* kan abstract zijn. Het is een erg breed onderwerp en beschrijft veel delen van een systeem. Hoewel de scope groot is zijn er overkoepelende ideologieën die een hulplijn kunnen zijn bij het uitwerken van een architectuur. Er zijn twee wetten die worden gebruikt als regels bij het ontwerpproces.

Everything in software architecture is a trade off

De eerste wet gaat over compromis. Binnen de wereld van de *software architecture* moeten er altijd compromis worden gesloten. Er is zelden een situatie waar er geen tegenwerkende kracht aanwezig is.

Dit idee sluit aan bij de twee wet van *software architecture*.

'Why' is more important than 'How'

De tweede wet laat zien welke denkwijze een software architect moet hebben. Binnen software development wordt er veel nadruk gelegd op 'hoe' iets kan worden gerealiseerd in plaats van 'waarom'. De reden waarom beslissingen zijn gemaakt is van essentieel belang voor niet alleen de software architect maar ook het development team en de *product owner*.

Software architecture gaat niet alleen over de structuur van een systeem maar ook de achterliggende doelen en wensen.

SOFTWARE ARCHITECT

Het zijn van een software architect is een populaire keuze binnen de wereld van de IT. Volgens het online vacature bedrijf Indeed is de 'software architect' de beste baan van 2020, waarbij de rol hoog scoorde op het gebied van salaris en marktgroei. Er zijn een stuk meer variabelen die een rol spelen bij het bepalen van de kwaliteit van een functie, desondanks wordt de functie 'software architect' geprijsd en is er vaak een hoog salaris aan verbonden. De reden waarom deze functie zo waardevol is voor bedrijven ligt bij de verschillende kwaliteiten die een architect moet hebben. Er zijn veel variaties op de functie en per project zijn er andere behoeftes.

TAKEN

Zoals eerder genoemd zijn er veel verschillende activiteiten waar een software architect te maken mee krijgt. De belangrijkste taken van een architect zijn als volgt:

- Evaluieren van software systemen
- Het ontwerpen van globale software architecturen

- Onderzoeken en noteren van risico's en valkuilen
- Ontwerpen van prototypes om levensvatbaarheid te bepalen

Deze taken omvatten het gehele development proces en worden geëvalueerd met het development team en *product owners*.

VERANTWOORDELIJKHEDEN

Naast de functies zijn er ook een aantal verantwoordelijkheden. Deze gaan over de taken die een architect heeft binnen het development team.

- Documenteren van software applicaties en systemen
- Opstellen van planningen en schema's voor het opleveren van software producten
- Het zoeken naar structurele problemen binnen software systemen
- Het onderhouden van relaties met andere afdelingen zoals marketing, managers en operationele afdelingen.

Deze verantwoordelijkheden zijn breed en hebben te maken met het onderhouden van projecten op organisatie niveau. Aan de taken is te zien waarom een software architect een gewilde functie is voor bedrijven, het omvat het gehele development proces en vereist veel kennis op technisch en sociaal gebied. Vaak is een software architect een leidinggevende rol binnen een development team.

CARRIÈRE

Om te beginnen is er vaak minimaal een aantal jaar ervaring nodig binnen software development en of engineering. In **Figuur** hieronder zijn een aantal eisen te zien voor de functie binnen verschillende bedrijven. De algemene overeenstemming is dat er een basis kennis van software ontwikkeling moet zijn voordat er naar de functie kan worden gesolliciteerd, hierdoor is het gebruikelijk om vanuit een opleiding eerst als developer te werken en later de stap te maken naar een functie zoals architect of ingenieur.

Software architect career paths

| Current company and position | Education | Key career milestones | What they say about the software architect job |
|---|--|--|--|
| 
Pinterest
David Chaiken,
a chief architect | Doctorate degree in electrical engineering and computer science | <ul style="list-style-type: none"> ✓ Software engineer at Motorola (2 years) ✓ Multiprocessor architect at MIT (6 years) ✓ Principal software and hardware engineer (4 years) ✓ Chief architect at Yahoo (4 years) | «Some days I'll focus on product strategy, and other days I'll be coding down in the guts of the system» |
| 
Sabre
Michael Nygard,
an enterprise architect | Bachelor of science degree in engineering and applied science | <ul style="list-style-type: none"> ✓ Chief scientist (3 years) ✓ Director of engineering (2 years) ✓ Technical director (4 years) ✓ VP Customer Solutions (7 years) | «Every decision you make imposes your will on your users. You should be willing to bear large burdens to ease theirs» |
| 
APACHE
Craig Russell,
a systems architect and Chairman of the Board | Bachelor of the arts degree in applied mathematics | <ul style="list-style-type: none"> ✓ VP Product Planning (6 years) ✓ Director of Product Support (7 years) ✓ Architect at Sun Microsystems (11 years) ✓ Architect at Oracle Corp. (7+ years) | «It's all about performance. When considering the implementation of a successful system, architects should always pay careful attention to it» |
| 
Daugherty
BUSINESS SOLUTIONS
Burk Hufnagel,
a technical and solution architect | DeVry Institute of Technology, certified Java developer and enterprise architect | <ul style="list-style-type: none"> ✓ Senior software development (5 years) ✓ Lead software architect (12 years) ✓ Solution architect (2+ years) | «Sometimes the best thing you can do to solve a problem is to put the mouse down and step away from the keyboard» |

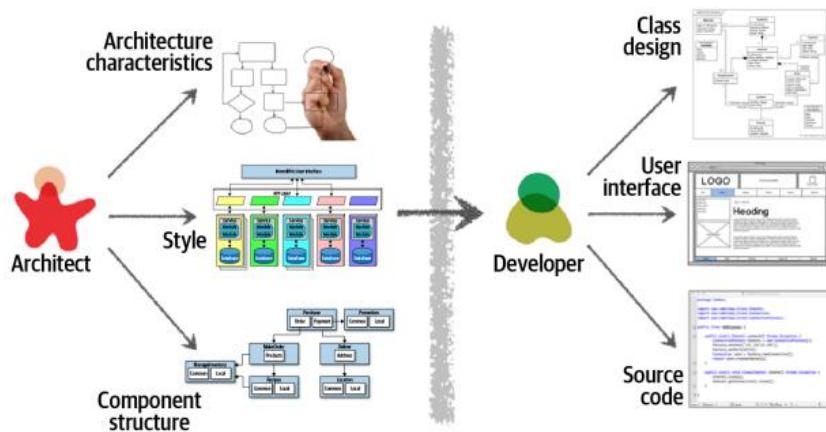


DENKWIJZE VOOR SOFTWARE ARCHITECTURE

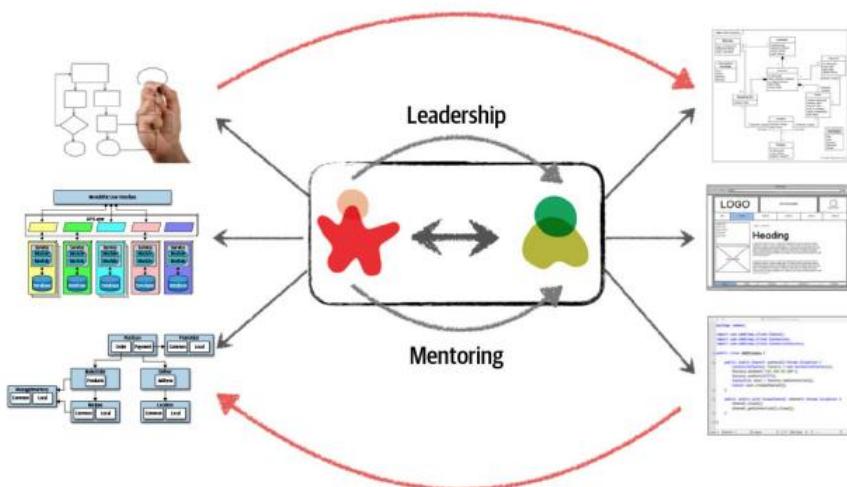
Zoals eerder genoemd gaat het er bij *software architecture* niet alleen over de 'hoe' maar over de 'waarom'. Dit is een voorbeeld van de manier waarop een architect moet denken. Er zijn vier aspecten die de denkwijze van *software architecture*. Het verschil tussen architectuur en ontwerp, technische breedte, analyseren van compromis en de bedrijfsfactoren die vertaald moeten worden naar architecturale zorgen.

Architectuur versus ontwerp

Binnen grotere organisaties met veel personeel worden de taken voor software architecten en software developers opgesplitst in taken die in **Figuur** te zien zijn. De architect werkt de structuur en kenmerken van een systeem uit, ondertussen werkt de developer aan de daadwerkelijke code en structuur van de database. Door deze traditionele verdeling ontstaat er een kloof tussen deze twee teams dat zorgt voor problemen op de lange termijn.



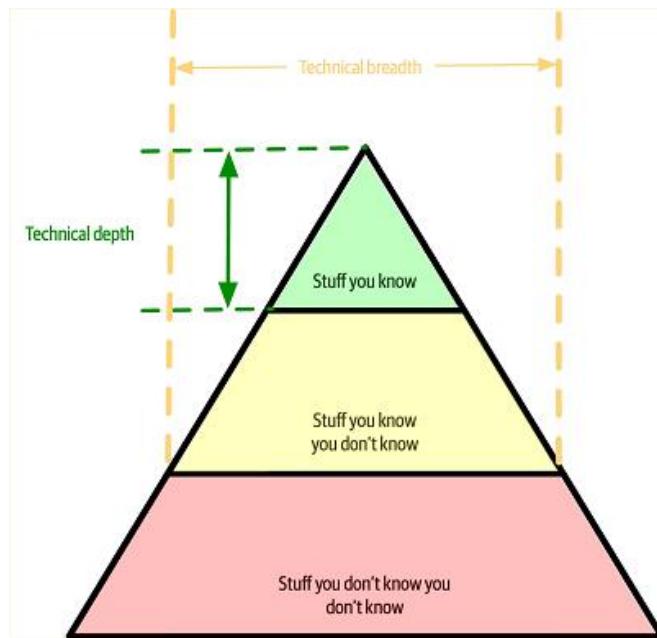
Om deze kloof te kunnen dichten moet er een nauwe samenwerking zijn tussen de twee onderdelen van een ontwikkelteam. Binnen vele 'kleinere' bedrijven worden de taken vaak verdeeld onder de beschikbare ontwikkelaars, ook binnen deze bedrijven moeten er verbindingen worden gemaakt tussen de architecturen en ontwerpen. Een architectuur moet realistisch zijn voor het development team, daarbij moet een ontwikkelaar de vooraf bepaalde architectuur respecteren binnen de code.



Technische breedte

Een andere vergelijking die moet worden gemaakt met developers is het verschil binnen de technische kennis van het individu, dit verschil wordt weergegeven binnen **Figuur**. Een ontwikkelaar haalt zijn waarde uit de grote hoeveelheid kennis die hij of zij heeft over bepaalde technologieën, dit wordt gerepresenteerd door de groene lijn.

Er moet veel tijd worden besteed aan het onderhouden van deze kennis, zeker in de wereld van de IT.



Een software architect heeft ook technische kennis nodig, maar daarnaast is een technische breedte erg waardevol. Technische breedte (*Technical breadth* in **Figuur**) is bekend zijn met veel verschillende onderdelen, in plaats van veel kennis hebben van één. Waar een software developer behoefte heeft aan technische diepte (*Technical depth*), is het voor een software architect van belang om veel technische breedte te hebben. Bij het maken van een carrière switch is het belangrijk om de andere denkwijze te begrijpen.

Analyseren van compromis

De quote hieronder komt van één van de schrijvers van '*Fundamentals of Software Architecture*' en vat in één zin samen wat de denkwijze van een software architect. Elke situatie is anders, daarbij is er nooit een perfecte oplossing.

"Architecture is the stuff you can't Google"

- Richards, Mark - Fundamentals of Software architecture

Als voorbeeld is het niet mogelijk om te zeggen welk communicatie protocol gebruikt moet worden voor een gegeven project, dit hangt er maar net vanaf. Zo is een voordeel van File Transfer Protocol (FTP) dat het gebruikt kan worden voor het versturen van grote bestanden binnen een bedrijf. Een nadeel van FTP is dat de data in principe niet is beveiligd en dus kwetsbaar is voor cyberaanvallen. Een ander protocol is Hypertext Transfer Protocol (HTTP). Een voordeel van HTTP is dat het erg flexibel is in de vorm van extensies en extra functionaliteiten die kunnen worden toegevoegd. Een nadeel is dat het net als FTP van nature in veilig is, daarbij is er bij HTTP administratieve overhead. Dit betekent dat er naast de nodige data nog meer informatie moet worden gestuurd om het systeem te laten werken.

Dit voorbeeld laat zien dat er bij elke beslissing voortkomt uit het afwegen van de voor- en nadelen en dat er altijd compromis gesloten moeten worden.

Bedrijfsfactoren vertaald naar architecturale zorgen

Een software project is geen losstaand object, het is een uitwerking van wensen en bedrijfsfactoren die weer voortkomen uit problemen binnen een bedrijf of sector. Het vertalen van deze problemen naar daadwerkelijke oplossingen is één van de probleem gevoelige aspecten voor een software architect.

Er is een bepaald niveau van bedrijfskunde en communicatie nodig om de relaties met de stakeholders te onderhouden.

Het balans tussen het ontwerpen van de architectuur en het schrijven van code kan een valkuil zijn voor de productiviteit van het team. Een software architect is geen full-time developer, hierdoor is de balans tussen het coderen en ontwerpen een prioriteit.

De waarde van een software architect binnen de code is om de onderzochte structuur te kunnen valideren en (deels) te implementeren. Het gebruik van Proof of Concept's (PoC's) is aangeraden. Bij een PoC kan er worden gefocust op de belangrijkste aspecten van een systeem en hoeft er niet worden gedacht aan het beschikbaar stellen van een volledig werkend product. Het PoC moet wel worden opgebouwd uit een goede kwaliteit code dat gelijk is aan de standaard van de rest van het project.

MODULARITEIT

De term modulariteit binnen software architectuur wordt gebruikt om een de verdeling van een systeem te beschrijven. Hoe kan het systeem worden opgedeeld in kleinere 'Modules' en hoe worden deze opgebouwd en met elkaar verbonden. Modulariteit is binnen vele talen en *frameworks* te vinden, het concept van een *framework* in het algemeen kan worden gezien als een module.

Binnen software architectuur wordt er modulariteit gebruikt om de groepering van code zoals classes of functies te beschrijven, hierbij wordt er niet alleen gekeken naar de fysiek verdeling, maar vooral de verdeling op basis van logica.

METEN VAN MODULARITEIT

Het concept modulariteit is abstract. Er is geen duidelijke definitie en het omvat veel elementen van een systeem, daarbij is het geen verplichting om je aan de regels te houden wat zorgt voor fouten die veel invloed kunnen hebben op de kwaliteit van het systeem. Er zijn drie onderdelen die worden gemeten om de modulariteit van een systeem te kunnen bepalen.

COHESION

De *cohesion* (samenhang) van een systeem meet hoe de modules binnen een systeem op functioneel gebied zijn gekoppeld aan elkaar. Er wordt bepaald hoeveel taken een module heeft en of deze allemaal thuis horen daar, wellicht dat sommige taken naar een andere module moeten worden verplaatst.

Cohesion wordt gemeten van laag naar hoog. Een lage *cohesion* betekent dat een module veel variatie heeft in de functionaliteiten die er worden uitgevoerd en geen focus heeft op één onderdeel. Een hoge *cohesion* is het tegenovergestelde, hierbij is er weinig variatie en wordt de module gebruikt voor één specifiek doeleinde. Er zijn zeven verschillende soorten *cohesion* methodes.

Functional cohesion

Elk deel dat nodig is voor het uitvoeren van een proces bevindt zich in één module. Elk deel van de module is gekoppeld aan elkaar. Het is de hoogste vorm van *cohesion*.

Sequential cohesion

Een groepering van onderdelen op basis van de data flow. De output van één module wordt gebruikt als input voor de volgende module.

Communicational cohesion

Twee modules hebben zijn gekoppeld op basis van communicatie, waarbij beide dezelfde input gebruiken of samenwerken aan een output. Als voorbeeld, Er wordt data naar een server gestuurd waar deze zelfde data wordt gebruikt om twee bronnen te verbinden met elkaar.

Procedural cohesion

Twee modules zijn verbonden aan de hand van de volgorde waarin ze worden uitgevoerd. Deze volgorde zorgt voor een zwakke verbinding tussen de modules. Als voorbeeld, eerst moet er data worden berekend voordat het kan worden opgeslagen.

Temporal cohesion

De koppeling komt voort vanuit de behoefte om bepaalde taken uit te voeren voor een bepaalde tijd. Deze taken hoeven niks met elkaar te maken hebben en puur een tijdelijke *cohesion* te vormen.

Logical cohesion

Modules zijn gerelateerd op basis van logica niet functionaliteit. Zo worden taken met soortgelijke logica binnen één module geplaatst ook al is er geen functionele verbinding. Als voorbeeld, een module waarin alle functionaliteiten worden gezet die te maken hebben met het communiceren met een database, hoewel er een verbinding is op basis van logica is geen functionele koppeling.

Coincidental cohesion

De onderdelen in een module hebben geen relatie anders dan het feit dat ze op de dezelfde locatie binnen de bron code staan. Dit is de laagste en slechtste vorm van *cohesion*.

Lack of Cohesion in Methodes (LCOM)

Het bepalen van de *cohesion* is erg subjectief, daarom zijn er verschillende statistieken die kunnen worden gebruikt om de *cohesion* van een systeem te bepalen. Een bekende manier om de *cohesion* te meten is de LCOM wat staat voor Lack of Cohesion in Methodes, waarmee wordt bepaald.

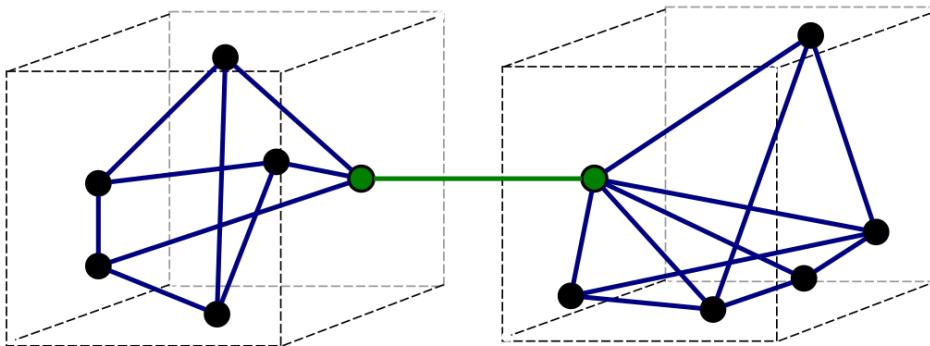
$$LCOM96b = \frac{1}{a} \sum_{j=1}^a \frac{m - \mu(A_j)}{m}$$

De formule in **Figuur** hierboven lijkt complex maar is beter te begrijpen met een voorbeeld. De LCOM meting bepaald hoeveel van de onderdelen van twee modules er niet zijn gekoppeld aan elkaar, bij een hoge LCOM score betekend dat er veel onderdelen niet zijn gekoppeld wat een lage *cohesion* betekend. Bij een lage score is er sprake van een hoge *cohesion*.

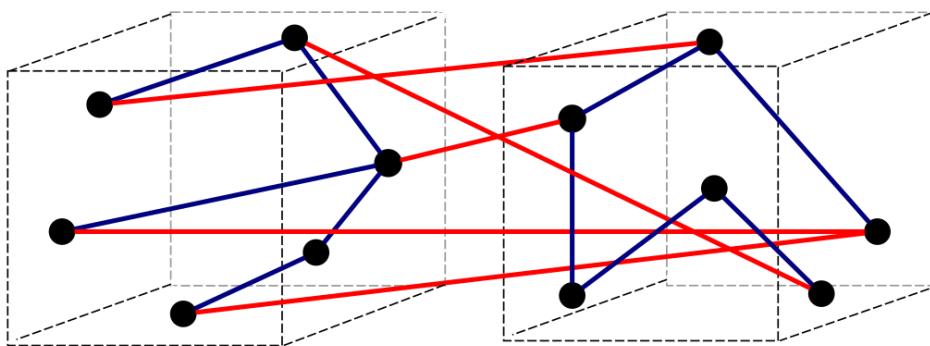
De LCOM meting kan worden gebruikt om bestaande code om te bouwen aan de hand van nieuwe architecturale principes, het kan worden gebruikt om onderdelen te vinden die niet gekoppeld zijn elkaar. Er is geen vaste meting die alle problemen binnen een systeem aantoon, daarnaast is de LCOM methode niet volledig en meet alleen wat de structurele *cohesion* is. Er moet per project handmatig worden bepaald welke andere problemen zich voordoen.

COUPLING

Met *coupling* (koppeling) wordt de afhankelijkheid tussen twee modules gemeten, het wordt vaak samen met *cohesion* gebruikt. Een hoge *coupling* betekent dat er veel afhankelijkheid is tussen twee modules wat betekend dat er een lage *cohesion* is, het tegenovergestelde is waar bij een lage *coupling*. In **Figuur** hieronder is een visuele representatie te zien waarbij twee modules worden weer gegeven als doorzichtige kubussen. De blauwe lijnen geven de *cohesion* weer van elke module de de groen/rode lijn laat de *coupling* zien.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Efferent en Afferent coupling

Er zijn verschillende soorten coupling mogelijkheden zoals data coupling, stamp coupling, external coupling of common coupling. Veel van deze soorten worden vooral gebruikt binnen software engineering.

Software architecture maakt gebruik van twee concepten binnen coupling, efferent en afferent coupling.

Efferent coupling meet de uitgaande koppelingen die een bepaald onderdeel moet gebruiken, de vraag die wordt beantwoord bij efferent coupling is: Van wie ben ik afhankelijk.

Afferent coupling meet de inkomende koppelingen die een onderdeel krijgt. De vraag die wordt gesteld en beantwoord is: Wie maakt er gebruik van mij?

Door deze twee types te meten is het duidelijk hoeveel koppelingen er worden gebruikt en hoe dit de architectuur beïnvloed.

ARCHITECTURE CHARACTERISTICS

Eerder werd het concept van *architecture characteristics* beschreven als "succesfactoren die niet direct een functionaliteit beschrijven". Eén van de taken van een software architect is om de aspecten van het systeem te definiëren die niet direct te maken hebben met de bedrijfseisen. Het opstellen van deze karakteristieken onderscheidt de architectuur van het ontwerp en de code zelf.

DEFINIËREN VAN ARCHITECTURE CHARACTERISTICS

Architecture characteristics, non-functional requirements en quality attributes zijn allemaal termen die hetzelfde concept beschrijven, dit kan het onderzoeken van de betekenis complex maken. In dit onderzoek wordt er gebruik gemaakt van *architecture characteristics* door het boek "Fundamentals of software architecture" die de term gebruiken omdat de andere opties waarde van het concept overbrengen.

Om onduidelijkheid te voorkomen wordt de definitie vanuit "Fundamentals of software architecture" gebruikt, deze bestaat uit drie delen.

- **Specificeren van niet domein gerelateerde ontwerpkeuzes.**

Tijdens het ontwerpen van een applicatie hebben de *characteristics* een specifieke rol. De systeem eisen bepalen welke functionaliteit de applicatie moet hebben, de karakteristieken bepalen welke operationele en ontwerp eisen voor het succes van de applicatie

Deze eisen komen niet op tijdens het bepalen van de systeem eisen maar zijn toch essentieel voor het werken van het systeem. Zo wordt er binnen de systeemeisen niet vermeld wat eventuele prestatie- of veiligheidseisen zijn, hoewel deze wel bepaalt moeten worden.

- **Heeft invloed op een structureel onderdeel van het ontwerp.**

Het bepalen van karakteristieken is direct gekoppeld aan het aanpassen van de ontwerp structuur. Binnen het bouwen van systeem wordt er bij elk project zonder gedefinieerde *characetristics* al rekening gehouden met standaarden op het gebied van veiligheid en prestatie. Denk aan de big O notatie bij het ontwerpen van algoritmes, een algemene notatie dat wordt gebruikt om aan te geven hoe goed een algoritme presteert bij **n** aantal elementen.

De *characetristics* zijn pas van toepassing als er structurele veranderingen moeten plaats vinden om aan de eisen te kunnen voldoen.

- **Is essentieel voor het succes van de applicatie.**

Het is niet wenselijk om zoveel mogelijk karakteristieken toe te kennen, elke nieuwe eis voegt meer complexiteit toe aan een systeem. Het is de taak van de software architect om de juiste eisen te bepalen in plaats van de grootste hoeveelheid eisen te bedenken.

Er wordt ook een onderscheid gemaakt tussen implicit en explicit. Implicite karakteristieken worden zelden buiten de *architecture characteristics* benoemd, toch zijn ze essentieel voor het ontwerpen van een architectuur. Expliciete karakteristieken komen voor op meerdere plekken dan alleen de specificatie van de karakteristieken, zoals de systeemeisen of ontwerpen.

LIJST VAN ARCHITECTURE CHARACTERISTICS

Er bestaan *characteristics* voor elke soort software dat gebouwd kan worden, daarbij is er geen definitieve lijst van alle eisen. Voor de meeste projecten is er de officiële standaard opgezet door de *International Organization for Standards* (ISO - 2500), daarnaast is het mogelijk voor project om eigen eisen op te stellen op basis van de behoeftes vanuit de stakeholders.

Voor dit onderzoek worden alleen de eisen vanuit ISO gebruikt, deze standaarden worden door meer dan 160 landen (binnen ISO members) geaccepteerd.

Functional Suitability

Beschrijft hoe correct het product de systeem eisen naleeft en uitvoert, deze systeemeisen worden gedocumenteerd binnen documenten zoals *System Requirements Specifications* (SRS). De volgende sub-eisen kunnen worden toegepast

- **Functional completeness**
 - Beschrijft hoeveel van de vooraf opgestelde eisen en wensen zijn gerealiseerd binnen het systeem.
- **Functional correctness**
 - Bepaald hoe nauwkeurig het systeem bepaalde resultaten kan formuleren.
- **Functional appropriateness**
 - Beschrijft of de taken en functie die worden uitgevoerd daadwerkelijk de eisen reflecteren.

Performance efficiency

Representeert de efficiëntie van het systeem bij vooraf bepaalde onderdelen en condities. Er zijn drie sub-eisen beschikbaar.

- **Time behaviour**
 - Meet of de prestatietijd en terugkoppeling van het systeem aan de vooraf opgestelde eisen voldoet.
- **Resource utilization**
 - Meet of de hoeveelheid data en soorten data dat gebruikt worden voldoet aan de eisen.
- **Capacity**
 - Meet of de limieten binnen het systeem voldoen aan de eisen.

Compatibility

Beschrijft in hoeverre een product, systeem of onderdeel kan communiceren met andere systemen of in hoeverre een systeem de nodige functies kan uitvoeren onder gelijke omstandigheden. Er zijn twee sub-eisen.

- **Co-existence**
 - Beschrijft in hoeverre het product de opgestelde functies kan uitvoeren binnen dezelfde omgeving met andere producten zonder impact op deze producten te hebben.
- **Interoperability**

- In hoeverre systemen of producten kunnen communiceren en data ontvangen en verwerken.

Usability

Beschrijft of het product of systeem kan worden gebruikt door gespecificeerde gebruikers om gespecificeerde taken effectief en naar verwachtvijg uit te voeren. deze karakteristiek is opgedeeld in zes sub-eisen.

- **Appropriateness recognizability**
 - Of een gebruiker kan zien dat het product of systeem voor hen van toepassing is.
- **Learnability**
 - Of het mogelijk is voor gespecificeerde gebruikers om het product of systeem te leren binnen de omgeving van het product of systeem zelf, risicotvrij en effectief.
- **Operability**
 - Of het product of systeem makkelijk te gebruiken is binnen de gekozen context.
- **User error protection**
 - In hoeverre het product of systeem voorkomt dat de gebruiker fouten kan maken en de gebruiker beschermt bij het maken van fouten.
- **User interface aesthetics**
 - In hoeverre de *user interface* (UI) soepel en goed te gebruiken is voor de gebruiker.
- **Accessibility**
 - In hoeverre het product of systeem gebruikt kan worden door gebruikers met verschillende achtergronden en kennisniveaus in de gespecificeerde omgeving.

Reliability

Specificeert of het product of systeem gebruikt kan worden binnen een bepaalde context, met een bepaalde prestatie binnen een bepaald tijdslot. Er zijn vier sub-eisen.

- **Maturity**
 - Of het product of systeem aan de eisen voor betrouwbaarheid onder normale omstandigheden doet.
- **Availability**
 - Of het product of systeem beschikbaar en operationeel is op de aangegeven momenten.
- **Fault tolerance**
 - In hoeverre het product of systeem operationeel blijft bij mogelijke hardware- en of softwareproblemen.
- **Recoverability**
 - In hoeverre het product of systeem in het geval van een interruptie of storing informatie kan terughalen en een gewenste staat kan herstellen.

Security

Specificeert hoe goed het product of systeem is in het beveiligen en schermen van informatie voor gebruikers en waar deze gebruikers alleen informatie kunnen zien die voor hun van toepassing zijn. Er zijn vijf sub-eisen.

- **Confidentiality**
 - In hoeverre een product of systeem data alleen beschikbaar maakt voor gebruikers met de correcte autoriteit.
- **Integrity**
 - In hoeverre een product of systeem onbevoegd toegang geeft tot het aanpassen van applicaties of data.
- **Non-repudiation**
 - Of het mogelijk is om te bewijzen dat processen hebben plaatsgevonden zodat deze later niet geweigerd kunnen worden.
- **Accountability**
 - Of het mogelijk is om een actie of entiteit te kunnen herleiden door middel van een unieke waarde.
- **Authenticity**
 - Of het mogelijk is voor een entiteit om zichzelf te kunnen identificeren.

Maintainability

Deze karakteristiek beschrijft in hoeverre het mogelijk is om het product of systeem aan te passen of te verbeteren en in welke mate dit effectief kan zijn voor nieuwe omgevingen en eisen. Er zijn vijf sub-eisen.

- **Modularity**
 - In welke mate een product of systeem is opgebouwd in verschillende modules of componenten, waarbij een verandering binnen één component minimale impact heeft op de andere modules.
- **Reusability**
 - Of het mogelijk is voor een onderdeel om gebruikt te worden binnen andere componenten of ik het opbouwen van nieuwe onderdelen.
- **Analysability**
 - In hoeverre het mogelijk is om de prestatie of de impact van een product of een systeem te kunnen meten om deze te kunnen onderzoeken. Of de mogelijkheid om fouten en knelpunten te kunnen identificeren.
- **Modifiability**
 - Of het mogelijk is om een product of systeem aan te passen zonder nieuwe fouten te introduceren of de bestaande kwaliteit te beschadigen.
- **Testability**
 - Of het mogelijk is om tests op te zetten en uit te voeren om de test criteria te kunnen valideren.

Portability

Beschrijft de mate waarin een product of systeem kan worden overgezet van de ene hardware naar de andere met een bepaald niveau van effectiviteit en efficiëntie. Er zijn drie sub-eisen.

- **Adaptability**
 - Of het mogelijk is voor een product of systeem om op nieuwere hardware gebruikt te kunnen worden met een niveau van efficiëntie.
- **Installability**
 - In hoeverre een product of systeem met succes geïnstalleerd en verwijderd kan worden van een bepaalde omgeving.
- **Replaceability**
 - In hoeverre een product of systeem vervangen kan worden door een ander systeem met hetzelfde doel in dezelfde omgeving.

IDENTIFICEREN VAN ARCHITECTURE CHARACTERISTICS

Eén van de eerste stappen die een software architect moet uitvoeren bij het ontwerpen van een architectuur is om de karakteristieken van het systeem te identificeren. Dit proces bestaat uit minimaal twee taken. Het onderzoeken van het domein en de huidige situatie en het bespreken van de wensen en prioriteiten van de aanwezige stakeholders.

Er zijn verschillende bronnen waar mogelijke karakteristieken vandaan kunnen komen, in dit onderzoek worden er twee veel voorkomende bronnen uitgewerkt. Het domein waarin gewerkt wordt en de opgestelde eisen.

IDENTIFICEREN VAN CHARACTERISTICS VANUIT HET DOMEIN

Bij het ontwerpen van een architectuur is er een bestaande situatie waar problemen worden geconstateerd. Aan de hand van deze problemen komen er behoeftes en wensen tot stand. Een software architect moet deze zorgen kunnen begrijpen en vertalen naar de termen die gebruikt kunnen worden bij het uitwerken van de architectuur.

Het verzamelen van de zorgen van de stakeholders lijkt op een relatief eenvoudige taak, de complexiteit ligt niet bij de wensen zelf maar bij het definiëren van een definitieve lijst waar alle stakeholder mee instemmen. Het is zelden dat alle partijen dezelfde prioriteiten stellen en het is makkelijk om hierdoor te kiezen voor een algemener ontwerp dat alle mogelijke onderdelen bevat. Dit is niet gewenst omdat elke karakteristiek die toegevoegd wordt de complexiteit van het product vergroot, daarbij kan een algemene architectuur zelden alle problemen van de stakeholders aankaarten.

Hieronder in **Figuur** worden een aantal veel voorkomende zorgen genoemd met *characteristics* die daarbij horen.

| Zorgen | Architecture characteristics |
|--------------------------|---|
| Mergers and acquisitions | Interoperability, adaptability |
| Time to market | Testability |
| User satisfaction | Performance, availability, fault tolerance, testability, security |
| Competitive advantage | Testability, availability, fault tolerance |

Tijd en budget

Reusability

De lijst die hierboven is vermeld is niet volledig en is meer een algemene vertaling van twee verschillende branches die samen moeten werken. Tijdens het opstellen van de *characteristics* is het van belang om niet alleen te luisteren naar de stakeholders, een architect heeft zelf vaak meer technische kennis dan de stakeholders. Een stakeholder kan nadruk leggen op de prestatie waar het product aan moet voldoen, als architect moet er ook worden nagedacht over veiligheid en schaalbaarheid.

IDENTIFICEREN VAN CHARACTERISTICS VANUIT SYSTEEMEISEN

Binnen een project zijn er vaak al documenten beschikbaar waar systeemeisen en wensen van stakeholder geformuleerd zijn, deze bevatten karakteristieken die gebruikt kunnen worden. De architect zelf heeft ook zowel technische als domeinkennis dat gebruikt wordt bij het opstellen van de *characteristics*, dit worden expliciete *characteristics* genoemd.

Er zijn veel methodes en termen voor het beschrijven van eigenschappen en systemen maar deze beschrijven niet alle mogelijke situaties die zich voor kunnen doen. Het is de taak van de architect om verder te kijken naar het domein zelf en eigen kennis en ervaring toe te passen om het ontwerp schaalbaar en robuust te maken.

METEN VAN ARCHITECTURE CHARACTERISTICS

Hoe meet je termen zoals *adaptability*? Het zijn vage termen die op veel verschillende manieren geïnterpreteerd kunnen worden. Een architect moet er per karakteristiek bepalen op welke manier deze wordt gemeten. Bij het uitwerken van deze methodes zijn er een paar problemen die zich voor kunnen doen. Zo kunnen de termen binnen één team al anders worden opgevat. Vele definities kunnen door architecten worden opgesplitst in meerdere aparte karakteristieken, dit kan de communicatie in de weg zitten.

Voor het meten van de gekozen *characteristics* moet er rekening worden gehouden met een paar onderdelen van het development proces waar mogelijk knelpunten zitten bij het toepassen van bepaalde eigenschappen.

OPERATIONELE METINGEN

Binnen elk team zijn er andere ideeën over het meten van bepaalde karakteristieken, dit resulteert in regels en criteria die per project anders kunnen zijn. De rol van een software architect is om deze criteria te beoordelen en bespreken, hierbij is het belangrijk om te kijken naar extreme gevallen die het systeem wellicht kunnen breken.

STRUCTURELE METINGEN

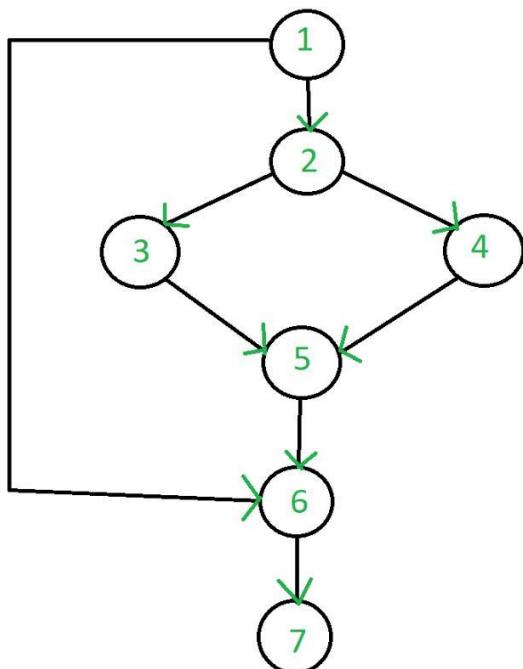
Voor het meten van structurele *characteristics* (*characteristics* die de structuur van het systeem beschrijven) is het moeilijk om één meetsysteem te gebruiken, hiervoor kan er gebruik worden gemaakt van een meting genaamd *Cyclomatic Complexity*.

Dit is een manier om de complexiteit van code te meten door te kijken naar alle mogelijke paden binnen een programma. Dit gebeurt aan de hand van een visualisatie techniek genaamd *Control Flow Graph* (CFG).

In de code hieronder wordt een simpel programma weer gegeven waarin een waarde wordt toegekend, daarna wordt deze waarde vergeleken en wordt er een nieuwe waarde toegekend.

```
if A = 10    // 1
if B > C    // 2
  A = B    // 3
else A = C  // 4
endif      // 5
endif      // 6
echo A, B, C // 7
```

Deze code kan worden uitgewerkt in een CFG, dit werkt door alle stappen die worden uitgevoerd weer te geven als cirkels en lijnen te trekken van de mogelijke paden. In **Figuur** hieronder is te zien hoe het voorbeeld van hierboven wordt vertaald naar een *graph*, hier zijn de cijfers gelijk aan de nummering binnen het voorbeeld.



Control Flow Graph

Om de complexiteit van dit voorbeeld te berekenen maken we gebruik van de volgende formule.

$$CC = E - N + 2P$$

CC = CYCLOMATIC COMPLEXITY

E = Aantal edges (alle keuzes)

N = Aantal nodes (alle nodes binnen de CFG)

P = Aantal verbonden componenten

Als deze formule wordt toegepast bij het voorbeeld is de uitkomt $7-7+(2*1) = 2$.

Het doel is om de CC waarde zo laag mogelijk te houden, een algemeen geaccepteerde waarde is een CC van maximaal 10. Deze waarde is wel maar een richtlijn, het is namelijk niet altijd mogelijk om de complexiteit van een product laag te houden. Als gewerkt wordt met een complexe dataset moeten er meerdere condities worden gebruikt om data te verwerken.

Een architect moet bepalen of de complexiteit van een functie afkomstig is van een complex domein of slechte code, dit kan de CC berekening niet onderscheiden.

PROCES METINGEN

Sommige karakteristieken komen samen met het development proces, in deze gevallen kan er gebruik worden gemaakt van bestaande methodes om systemen te testen. Het is de taak van de architect om de resultaten van deze methodes te verwerken en verdere conclusies te trekken.

Door het brede beeld dat een architect heeft op het product kunnen er conflicten zijn tussen de conclusies van het development team.

BEWAKING VAN CHARACTERISTICS

Het bewaken van de code en keuzes die worden gemaakt binnen de ontwikkeling van het product valt onder de taken van de architect. Het is een deel van software development waar vaak op wordt bezuinigd ter gunste van efficiëntie en het behalen van de strakke deadlines. Door de jaren heen zijn er veel verschillende pragmatische oplossingen bedacht die de taken voor onder andere software architecten een stuk makkelijker maken. Methodieken zoals Agile en DevOps hebben het werkfeld in de afgelopen 20 jaar erg veranderd. Er zijn binnen de code verschillende manieren waarop de *architecture characteristics* te bewaken, deze worden architecturale fitness functies genoemd.

FITNESS FUNCTIES

De term 'Fitness functie' komt oorspronkelijk uit de kant van de *evolutionary computing* en genetische algoritmes. Binnen genetische algoritmes wordt er gebruik

gemaakt van een fitness functie om de kwaliteit van een mogelijke oplossing te meten. Binnen een genetisch algoritme worden er een groot aantal oplossingen gegenereerd die worden gesorteerd worden op basis van de 'fitness'. Door dit vele malen te herhalen wordt de kwaliteit van de oplossingen steeds hoger, dit is te vergelijken met natuurlijke selectie binnen ecosystemen.

Het gebruik van fitness functies binnen software architectuur gebeurt door middel van verschillende aparte methodes om mogelijke valkuilen binnen de code op te lossen. Het concept van een fitness functie gaat om een set aan ideeën die een software architect kan gebruiken om de *characteristics* van de architectuur te bewaken.

ARCHITECTURE STYLES

Architecture styles (Architecturale structuur) beschrijven de relaties tussen verschillende componenten, hoewel elk project een bepaalde structuur anders implementeert zijn er overlappende eigenschappen en voor- en nadelen.

Door bepaalde vaste structuren een naam te geven is het voor een individu met de juiste technische kennis duidelijk welke eigenschappen een systeem heeft op basis van de gekozen structuur.

FUNDAMENTALE PATRONEN

Er zijn een aantal fundamentele patronen die al lang aanwezig zijn binnen de wereld van software development maar toch tot op de dag van vandaag gebruikt worden. De structuren die verderop worden beschreven bestaan deels uit deze patronen.

Big Ball of Mud

Bij een complete afwezigheid van enige structuur wordt er gesproken van een 'Big Ball of Mud', de term komt van een valkuil beschreven in 1997 door meneer Foote en meneer Yoder. Het vermijden van deze patroon is essentieel, toch komt het nog vaak voor binnen projecten door een gebrek aan testen en controles.

Client-server

Ook wel *two-tier* genoemd. Deze structuur is een verdeling tussen twee delen van een applicatie, de *frontend* (voorkant) en de *backend* (achterkant). Er zijn veel verschillende soorten structuren die gebruik maken van deze verdeling. De structuur wordt sinds de jaren 80 gebruikt.

Desktop + database server

Tijdens het opkomen van de *personal computers* in de jaren 70 en 80 werd het mogelijk om applicaties te maken die op de pc werd geïnstalleerd en via protocols kon communiceren met een externe server. Op de PC werd de data gerepresenteerd terwijl het zware werk op de server gebeurde.

Browser + web server

In de jaren 90 werd het gebruik van het web populair. Dankzij deze technologie was het mogelijk om een structuur te bouwen die veel weg had van de desktop + database server die hierboven is benoemd. Een split tussen de browser en de web server (deze web server verbindt met een database server). Het voordeel van een webapplicatie over een desktop app is dat de hardware specificaties minder van belang zijn, daarbij is de prestatie van de *webapp* minder afhankelijk van deze hardware.

Three-tier

In de latere jaren van het vorige decennium werd het populair om extra lagen toe te voegen.

De zo geheten *three-tier* architectuur bestaat uit verschillende lagen die de taken van een systeem verdelen. Een database laag voor de handelingen naar de database toe, een applicatie laag, een *frontend* laag waar de HTML en CSS code wordt beheert. Een gemiddelde developer heeft verschillende tools tot zijn beschikking die het navigeren van de verschillende lagen gedeeltelijk automatiseert.

MONOLITHIC VS DISTRIBUTED ARCHITECTURES

In dit onderzoek wordt er een verschil gemaakt tussen twee verschillende soort architectuur structuren, de *Monolithic* en de *distributed architecture*. De *Oxford dictionary* beschrijft de term *monolithic* als 'beschrijft een grote verticale blokken vaak van steen'. Dit komt overeen met het concept binnen software architectuur. Een *monolithic architecture* is een op zichzelf staande structuur waarbij alle onderdelen en functionaliteiten van een systeem in één codebasis.

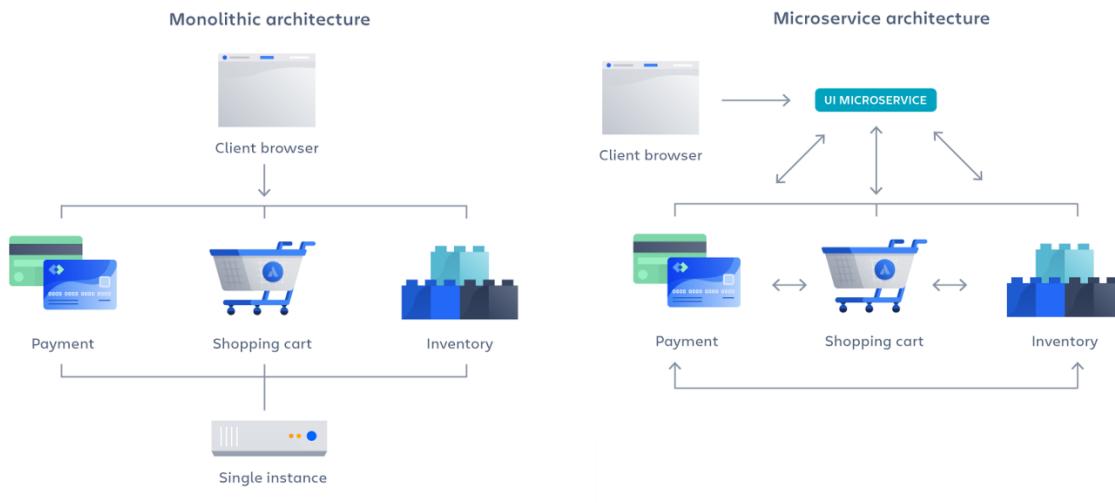
Een *monolithic* architectuur heeft een paar voordelen, omdat alles op één plek staat is het een relatief eenvoudige structuur om te begrijpen en te implementeren, daarnaast is de prestatie van het systeem groter als dit wordt vergeleken met een *distributed* systeem. Het testen van de simpele structuur is ook eenvoudiger.

De grootste nadelen van een op zichzelfstaande architectuur is de schaalbaarheid en de betrouwbaarheid van het systeem, fouten in de code kan het hele systeem onstabiel maken, daarbij is het uitbreiden en toevoegen van complexiteit een langdurig proces.

Een *distributed architecture* wordt ook wel *microservices* genoemd en is in vergelijking tot een *monolithic* architectuur een structuur gebaseerd op meerdere kleinere componenten die met elkaar verbonden zijn via een communicatie protocols zoals HTTP (HyperText Transfer Protocol) of REST (Representational State Transfer).

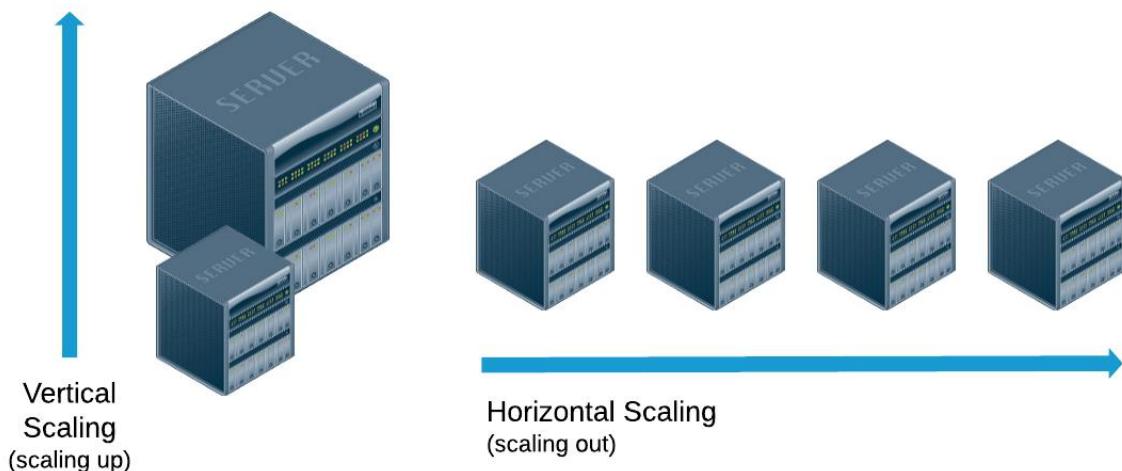
De voordelen van een *distributed architecture* zijn de schaalbaarheid, beschikbaarheid en prestatie. Door het verdelen van het systeem in kleinere onderdelen kunnen de verschillende onderdelen los van elkaar worden ontwikkeld en is het toevoegen van nieuwe componenten eenvoudig. In **Figuur** hieronder is een voorbeeld te zien waarbij de twee structuren worden gevisualiseerd. Bij de *microservice architecture* wat een veer voorkomende verdeelde structuur is te zien

hoe de verschillende onderdelen (*payment, shopping cart, inventory*) met elkaar verbonden zijn, bij de *monolithic* structuur is dit niet het geval.



Een *distributed architecture* kan gebruik maken van oneindige *horizontal scaling* (horizontaal schalen), bij horizontale schaalbaarheid wordt een systeem uitgebreid door data op te delen en te verdelen over meerdere systemen, dit wordt *partitioning* genoemd.

De andere optie is om verticaal op te schalen, bij *vertical scaling* wordt de data niet verdeeld maar wordt de locatie waar de data staat vergroot door extra opslag of RAM (Random Acces Memory - Werkgeheugen) toe te voegen. Voor de meeste flexibiliteit is horizontaal schalen de betere optie, wel is het onderhouden en ontwikkelen ervan complexer als dit wordt vergeleken met verticale schaling. *Vertical scaling* is eenvoudiger om te implementeren en is vaak goedkoper. Net als bij *monolithic architecture* is *vertical scaling* makkelijker om te bouwen maar zijn er limitaties op het gebied van betrouwbaarheid en de extra werk die nodig is om veranderingen toe te voegen.



Een *distributed architecture* heeft een aantal valkuilen waar een *monolithic* systeem geen last van heeft. Een architect moet deze valkuilen meenemen bij het maken van beslissingen.

Valkuil 1: Het netwerk is stabiel

Bij het ontwerpen van structuren en het uitwerken van ideeën wordt er vanuit gegaan dat het netwerk en de communicatie altijd stabiel is. Tijdens de eerste fase van het project wordt er gewerkt vanuit een utopie waarin alle kleine problemen worden genegeerd om de creativiteit van het team hoog te houden, dit kan later een probleem worden. Bij een *distributed architecture* kan netwerkproblemen een groot deel van het systeem onbereikbaar maken, hiermee moet actief rekening worden gehouden tijdens het ontwerpen. Er kan gebruik worden gemaakt van *timeouts* en *breakers* om data en verbindingen af te breken als deze niet werken, waardoor data niet verloren gaat.

Valkuil 2: Bandbreedte is oneindig

Naast de betrouwbaarheid van het netwerk is de prestatie ook een onderdeel dat vaak wordt genegeerd. Bij het opsplitsen van een systeem is de communicatie tussen de systemen cruciaal. Het doel is om zo weinig mogelijk data te sturen, er moeten compromis worden gesloten binnen het systeem over welke data moet worden gecommuniceerd. Ook moet het aantal berichten dat wordt gestuurd geminimaliseerd worden.

Valkuil 3: Het netwerk is veilig

Veel architecten zijn erg gewend geraakt aan het gebruiken van beveiligde netwerken, zelfs buiten de werkplek is een beveiligde verbinding de standaard en wordt er niet nagedacht over de risico's.

Het is daarom belangrijk om duidelijk te maken dat een netwerk van nature niet veilig is, maar veilig gemaakt moet worden. Elk communicatiepunt moet beveiligd worden en er kan niet zomaar worden gewerkt met persoonlijke data. Veel bedrijven sluiten zich aan bij de ISO/IEC 27001 certificering over informatieveiligheid. Een certificering voor het omgaan met persoonlijke data en het beveiligen door middel van standaarden.

Valkuil 4: De topologie verandert niet

De topologie van een netwerk is constant aan het veranderen, hardware en software moet vaak worden aangepast of opnieuw worden geconfigureerd, dit kan impact hebben op het netwerk. Als er geen rekening wordt gehouden met deze veranderingen kan het veel tijd kosten voordat een probleem bij een groot netwerk wordt herkend. Een architect heeft de taak om constant in contact te zijn met de netwerk engineers en server administratoren. Dit lijkt logisch maar vaak is netwerk beheer en development gescheiden.

Valkuil 5: Er is maar één administrator

Binnen een bedrijf zijn er meerdere teams die allemaal eigen belangen en niveaus van kennis bezitten. Voor een project team is het soms moeilijk om goed contact te houden met de verschillende afdelingen van een organisatie. Door de technische maar vooral het abstracte taal die wordt gebruikt is het te begrijpen waarom er wrijving kan ontstaan. Bij het gebruiken van een *distributed system* is het beheren ervan een taak dat serieus genomen moet worden. Het onderhouden van relaties en het bespreken van aanpassingen is een essentieel onderdeel dat veel impact heeft op de prestatie van het gehele systeem, toch wordt hier vaak niet veel aandacht aan besteed.

Valkuil 6: Data consistentie

Als een team overstapt van een *monolithic system* naar een *distributed system* moet er worden nagedacht over de consistentie van data. Binnen een verdeeld systeem is het concept *eventual consistency* van kracht. Bij dit concept worden veranderingen binnen de verschillende systeem lokaal opgeslagen waarnaar deze worden samengevoegd op een later moment.

"The idea is simple: replicate the data across participants, on each participant, perform updates tentatively locally, and propagate local updates to other participants asynchronously, when connections are available."

-Burckhardt, S - *Principles of Eventual Consistency*

De kracht van schaalbaarheid en beschikbaar komt met een nadeel, de extra complexiteit met betrekking tot de datastromen en de verdere problemen die eerder zijn benoemd.

LAYERED ARCHITECTURE STYLE (MONOLITHIC STRUCTURE)

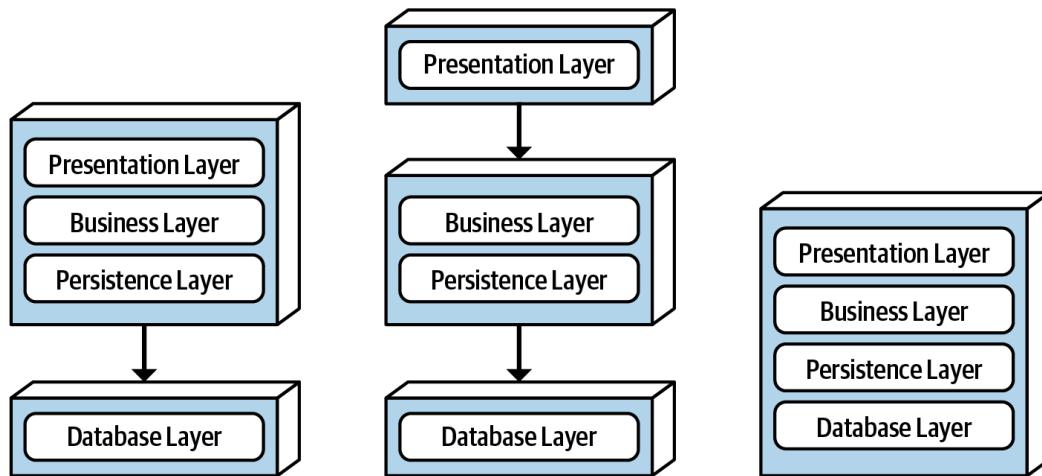
De *layered architecture style* (ook wel *n-tiered architecture* genoemd) is één van de bekendste en meeste gebruikte manieren om een architectuur op te bouwen. Het idee achter de structuur is om alle soortgelijke functies op te delen in verschillende componenten, dit zorgt voor een natuurlijke verdeling van taken. Er is geen limiet aan hoeveel lagen gebruikt kunnen worden.

Elke laag kan open of dicht zijn. Bij een dichte architectuur moet een request langs alle lagen gaan om uitgevoerd te worden, dit kan nodig zijn bij sommige systemen maar meestal kunnen sommige lagen worden overgeslagen. Het concept *layers of isolation* zorgt ervoor dat aanpassingen in één laag geen effect hebben op een andere laag, dit kan alleen als de structuur goed is ontwikkeld. De lagen die essentieel zijn voor de *main-flow* van het systeem moeten dicht zijn om te voorkomen dat het systeem foutieve resultaten levert.

De structuur kan goed worden gebruikt als er nog geen duidelijkheid is over de uiteindelijke architectuur. Bij het uitwerken van microservices is het gebruik van een lagen structuur als basis erg populair.

Topologie

Het **Figuur** hieronder laat verschillende manieren zien waarop de architectuur ingericht kan worden. Er is geen officiële regel over de hoeveelheid lagen, vier lagen komt het meest voor. *Presentation, Business, persistence en database*. Elke laag heeft zijn eigen verantwoordelijkheid en rol binnen het systeem. De lagen architectuur is *technically partitioned*, wat inhoudt dat de componenten zijn opgedeeld op basis van de technische rol die zij hebben. Er kan ook een *domain-partitioned* architectuur gebruikt worden, waarbij de verdeling plaats vindt op basis van rol binnen het gekozen domein.



Eén van de concepten binnen de lagen architectuur is *separation of concerns* (SoC), waarbij de verschillende lagen geen kennis hoeven te hebben over wat er in de andere lagen gebeurt. De *business* laag hoeft niet te weten hoe de *persistence* laag werkt en welke functies er worden uitgevoerd.

Voor- en nadelen

In de tabel hieronder is een kort overzicht van de voor- en nadelen van de lagen architectuur. De rol van de architect is om deze eigenschappen per project af te wegen.

| Voordelen | Nadelen |
|--|--|
| Het is relatief eenvoudig om te leren en te implementeren. | Schaalbaarheid is minimaal door de structuur van de stijl, toevoeging van lagen kan voor problemen zorgen. |
| Er is minder afhankelijkheid omdat de lagen apart worden ontwikkeld. | Onderhouden kan ingewikkeld zijn, sinds een verandering in een laag het hele systeem aan kan passen. |
| Het testen van is eenvoudiger door de gescheiden lagen. | Er is een onderlinge afhankelijkheid tussen lagen sinds de ene laag afhankelijk is van een andere. |
| De kosten liggen laag door de simpele implementatie. | |

PIPELINE ARCHITECTURE STYLE (MONOLITHIC STRUCTURE)

Ook wel bekend als de *pipe and filters* structuur. De *pipeline* architectuur is een fundamentele stijl die veel wordt gebruikt. Bij deze structuur wordt er een éénrichtingsweg opgezet waarin data per fase wordt aangepast.

Topologie

De structuur bestaat uit *pipes* en *filters*. De *pipes* zijn *point-to-point* verbindingen waar data doorheen stroomt, ze worden gebruikt als communicatie middelen voor de *filters* en gebruiken vaak kleine hoeveelheden data om de prestatie hoog te houden.

Filters zijn losse systemen die *self-contained* zijn, dit houdt in dat ze geen logica van buitenaf nodig hebben, alleen data. Er bestaat vier soorten *filters*

- **Producer**

De start van een proces.

- **Transformer**

Krijgt input, verwerkt de input en stuurt deze door. Binnen sommige *frameworks* wordt de term *map* gebruikt.

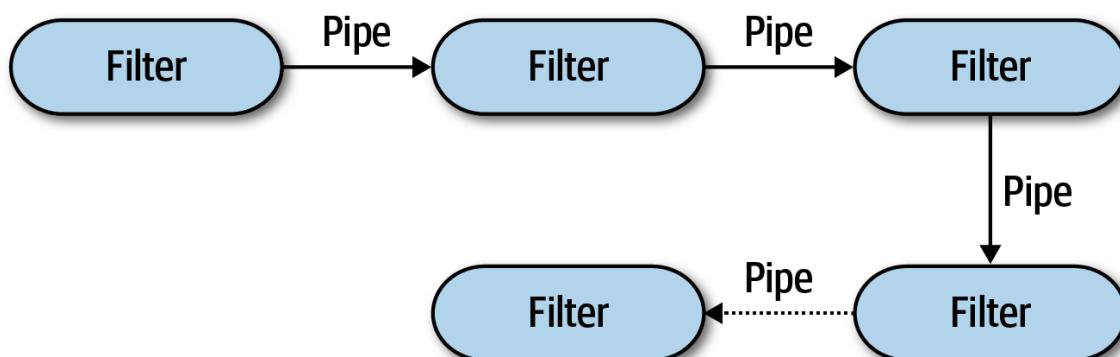
- **Tester**

Krijgt input, voert één of meerder tests uit en stuurt de aangepaste data door.

- **Consumer**

Dit is het einde van het proces, data kan worden opgeslagen of worden weer gegeven.

In **Figuur** hieronder wordt een voorbeeld gegeven van de structuur, waarbij de *pipes* de verschillende *filters* verbindt.



Deze architectuur stijl komt veel voor bij simpel éénrichtingsverkeer waarbij data van de ene bron via een aantal filters naar een andere bron wordt gestuurd. Binnen verschillende *Orchestrators* producten wordt er ook gebruik gemaakt van een *pipe and filter* architectuur, het principe *orchestrator* wordt vaak vergeleken met automatisering. De twee termen liggen dicht bij elkaar maar hebben toch wat

verschillen. *Orchestration* is het automatiseren en beheren van processen, het gaat hier niet alleen over het automatiseren maar ook over het maken van beslissingen en het aanpassen op basis van verschillende conditie.

| Voordelen | Nadelen |
|--|---|
| Structuur is makkelijk te begrijpen en te implementeren, geen complexe relaties. | De elasticiteit is laag, deze term beschrijft hoe flexibel een systeem is bij een verandering van <i>workload</i> . |
| Kosten liggen relatief laag. | Schaalbaarheid is laag door de complexiteit die nodig is om dit te bereiken. Dit komt door de op zichzelfstaande structuur. |
| Door de simpele structuur is het eenvoudig om elementen toe te voegen. | |

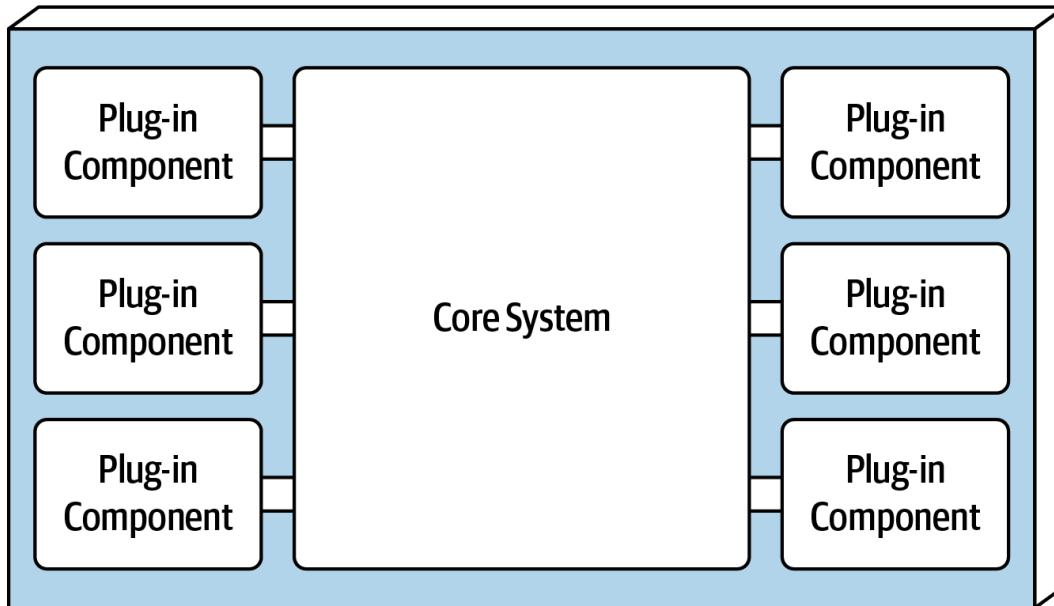
MICROKERNEL ARCHITECTURE STYLE (*MONOLITHIC STRUCTURE*)

De *microkernel* structuur is de laatste *monolithic system* die in dit onderzoek aan bod komt. De structuur maakt gebruik van verschillende integraties die samenkommen in één systeem genaamd het *core system*. Als het onduidelijk is welke systemen er worden toegevoegd in de toekomst is het gebruik van een soortgelijke structuur gewenst.

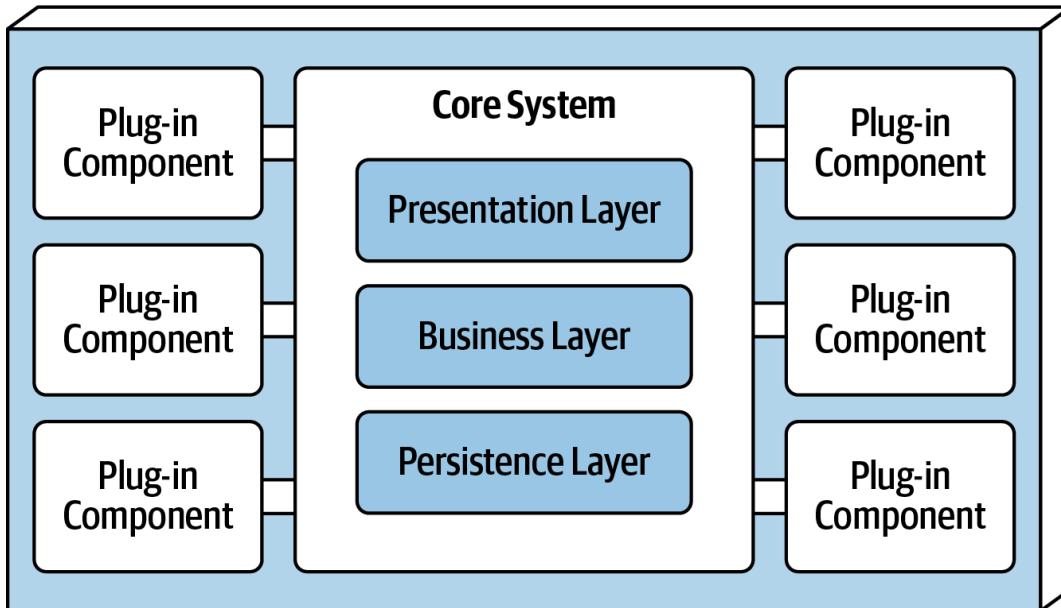
Topologie

In **Figuur** is een simpele versie te zien van de architectuur. De verschillende integraties worden *plug-in components* genoemd. Het *core system* bevat de minimale functionaliteit die nodig is om het product te gebruiken, de meeste complexiteit en functionaliteit wordt gedecentraliseerd naar de componenten.

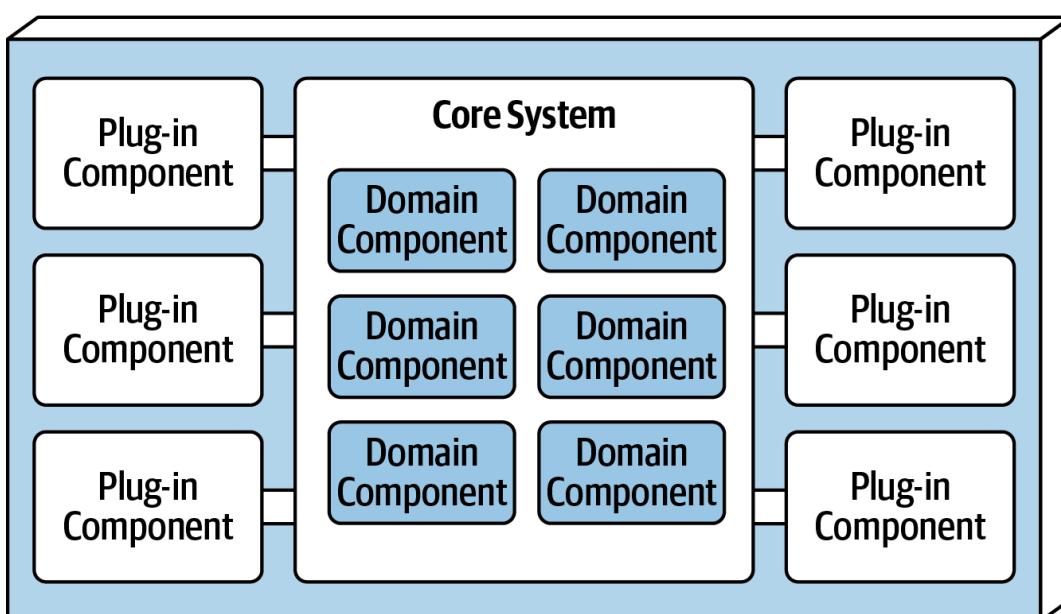
Als voorbeeld, Microsoft Word is een complexe applicatie vol met allerlei functionaliteiten. Als deze functies worden verwijderd blijf je over met een applicatie dat erg lijkt om de *notepad* app die Windows gebruikers goed kennen. Deze super simpele *text editor* kan worden vergeleken met een *core system* binnen de *microkernel* structuur, daarbij zijn alle functies die Word oorspronkelijk had verplaatst naar de verschillende *plug-in components*.



Afhangend van de grote van het project kan het *core system* worden opgebouwd met een *layered* structuur zoals eerder benoemd. Het is mogelijk om dit centrale systeem op te delen in kleinere componenten die ieder eigen *plug-in components* beheren, deze componenten worden *domain components* genoemd. In **figuur** is een schematisch voorbeeld zichtbaar van de twee mogelijke implementaties. Zoals eerder benoemd is een *layered* structuur verdeeld op technisch niveau, dit betekend dat de verdeling wordt gemaakt op basis van wat het systeem doet. Het *modular core system* wat hieronder is benoemd is verdeeld op domein niveau, waarbij er wordt gekeken vanuit het perspectief van de gebruiker.



Layered Core System (Technically Partitioned)



Modular Core System (Domain Partitioned)

De *plug-in components* die worden gebruikt bevatten specifieke functionaliteiten die als toevoeging dienen voor het *core system*. De communicatie tussen de componenten en het centrale systeem gebeurt via *point-to-point* integraties, dit zijn 1 op 1 relaties. Binnen bepaalde *frameworks* is de verbinding tussen het centrale systeem en de componenten mogelijk met het gebruik van *namespaces*, dit zijn stukken van een systeem die zijn afgezonderd. Door dit aparte stuk code een naam (vaak met de volgende structuur: app.plug-in.<domein>.<context>) te geven is het mogelijk om relaties te vormen tussen verschillende *namespaces*.

Voor- en nadelen

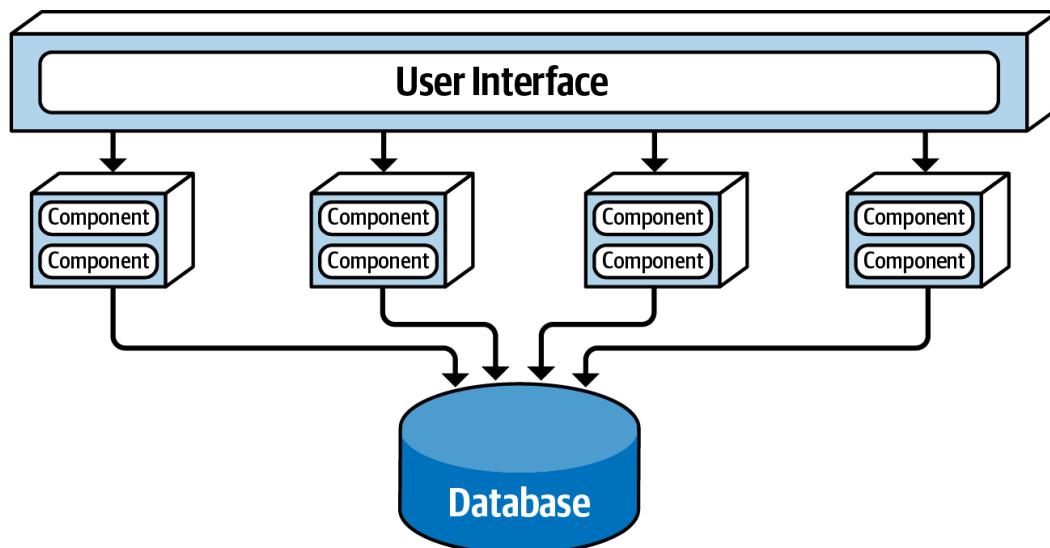
| Voordelen | Nadelen |
|--|--|
| Door de <i>monolithic</i> structuur is de implementatie relatief eenvoudig | De schaalbaarheid is laag door de structuur van het <i>core system</i> . |
| De kosten liggen zijn ook laag | De gevoeligheid voor fouten is hoog door dezelfde structuur. |
| Modulariteit is hoger in vergelijking met andere <i>monolithic</i> structuren door de verdeling van het <i>core system</i> en de <i>plug-in components</i> . | |
| Kan worden verdeeld op basis van het domein of de techniek. | |

SERVICE-BASED ARCHITECTURE STYLE (*DISTRIBUTED ARCHITECTURE*)

De *service-based architecture* is populair bij bedrijfsapplicaties omdat het veel van de voordelen van een *distributed architecture* zonder veel van de complexiteit en de kost van vele andere structuren.

Topologie

De algemene structuur wordt weer gegeven in **Figuur**. Er is een aparte *UI* en aparte systemen die *domain services* worden genoemd. Als opslag wordt er een *monolithic* database gebruikt waar de data van de verschillende services samenkomt. Gemiddeld zijn er zeven *domain services* aanwezig binnen een systeem.



De flexibiliteit van het systeem is hoog door de vele combinaties die mogelijk zijn, zo kunnen er meer databases gebruikt worden en is kan de *UI* worden opgedeeld om beter aan te sluiten bij de *domain components*, daarbij kunnen de *domain components* zelf ook worden aangepast.

Voor- en nadelen

| Voordelen | Nadelen |
|--|---|
| Door het opsplitsen van het systeem kunnen veranderingen met minder moeite worden doorgevoerd. | Schaalbaarheid is lager dan andere <i>distributed</i> structuren. |
| Testen is duidelijker dankzij een duidelijke verdeling. | De elasticiteit lager dan andere <i>distributed</i> structuren. |
| Veel flexibiliteit door de vele verschillende configuraties. | |
| De kosten en de complexiteit liggen lager in vergelijking met andere architecturen met een <i>distributed</i> structuur. | |
| Betrouwbaarder dan andere <i>distributed</i> structuren dankzij de manier waarop de <i>domain services</i> zijn opgedeeld. | |

EVENT-DRIVEN ARCHITECTURE STYLE (*DISTRIBUTED ARCHITECTURE*)

De structuur die wordt gebruikt bij een *event-driven architecture* is erg populair voor het maken van applicaties met een hoge schaalbaarheid en prestatie. De meeste applicaties die worden ontwikkeld gebruiken een concept genaamd *request-based modeling*, bij dit model wordt er een *request* gemaakt vanuit een *request orchestrator*, dit is een component dat data krijgt via een *request*. De rol van de *orchestrator* is om alle *requests* die binnen komen te beheren en de juiste *flows* in gang te zetten, zoals het ophalen of opslaan van data.

Een *event-based model* worden er *flows* in gang gezet op basis van een bepaalde actie die wordt uitgevoerd, wat er gebeurt is afhankelijk van de situatie. Een vergelijking die kan worden gemaakt is het verschil tussen iets kopen en op iets bieden.

Het kopen van een auto in een winkel is net een soort *request-based* systeem, als consument laat u weten welke auto u wilt kopen, dit is het *request*. De autowinkel zorgt ervoor dat de auto klaar wordt gemaakt en regelt al het papierwerk, dit is de *request orchestrator*. Als de consument de sleutels in handen heeft is de *flow* klaar.

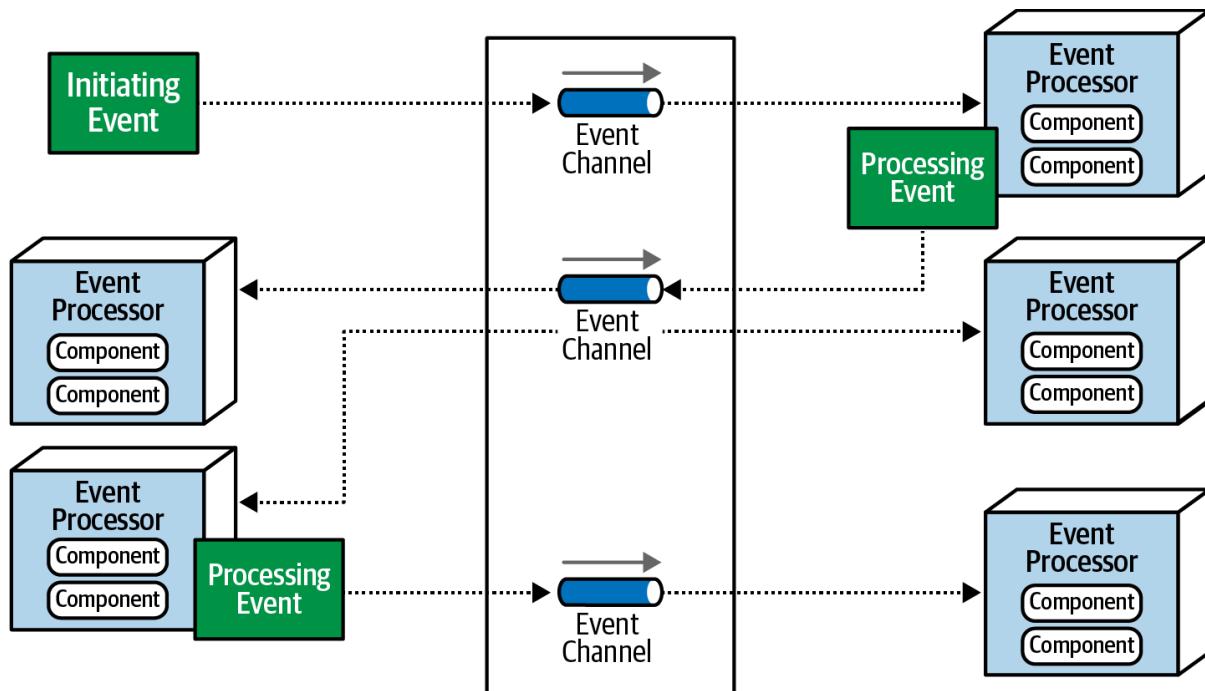
Bij het bieden op een auto wordt er een *event* in gang gezet zodra een auto tentoon wordt gesteld en de bieder een bod doet, dit *event* betekent niet dat de bieder de auto krijgt. In plaats daarvan wordt er op basis van de situatie (zijn er andere bidders) bepaald wat de volgende stap moet zijn. Ditzelfde principe komt terug bij een *event-based model*.

Topologie

De topologie van een *event-driven architecture* komt in twee smaken, de *broker* en de *mediator*. De *mediator* wordt gebruikt als er controle nodig is over de verschillende processen, de *broker* is nuttig als er behoefte is aan een hoog niveau van ontvankelijkheid en dynamisch gedrag.

De broker

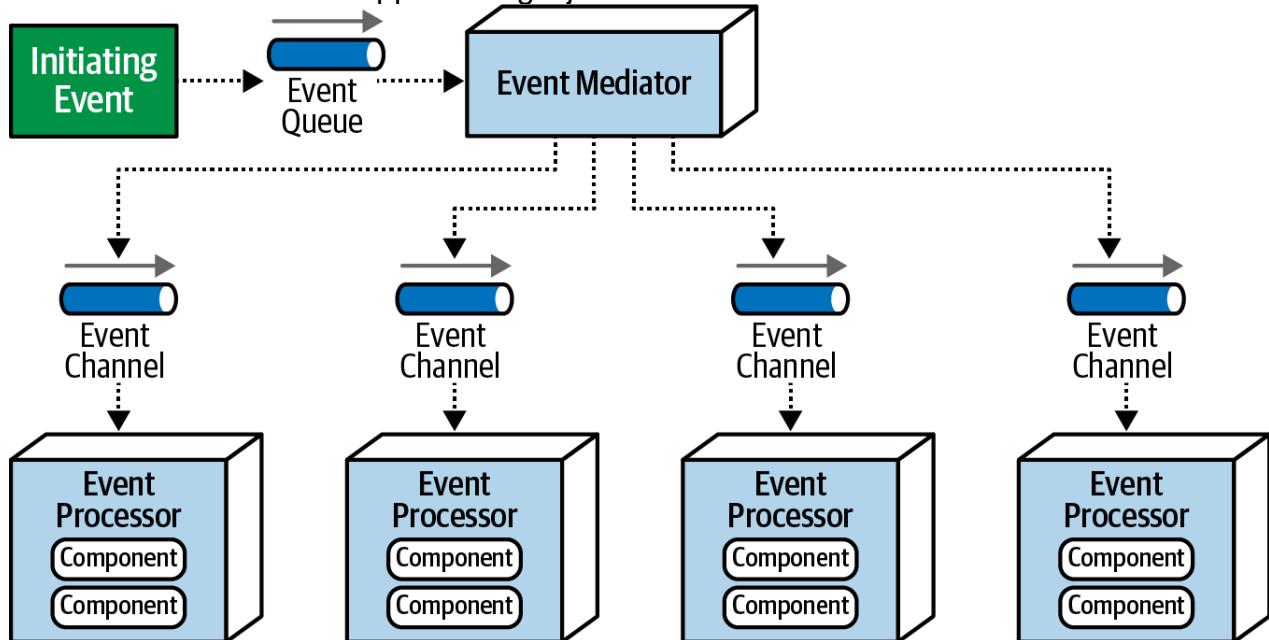
Het grootste verschil tussen de *broker* en de *mediator* is dat de broker geen centrale manager heeft die de verschillende events beheert. De *broker* werkt het beste als er simpele datastromen zijn waar geen extra organisatie voor nodig is.



| Voordelen | Nadelen |
|----------------------|---|
| Hoge schaalbaarheid. | Geen controle over de <i>workflow</i> . |
| Goede prestatie. | Omdat er geen <i>mediator</i> aanwezig is, is er minder inzicht in problemen. |

De mediator

Bij deze soort wordt er gebruik gemaakt van de *mediator* en is goed geschikt voor situaties waar er meer stappen nodig zijn om een *event* te verwerken.

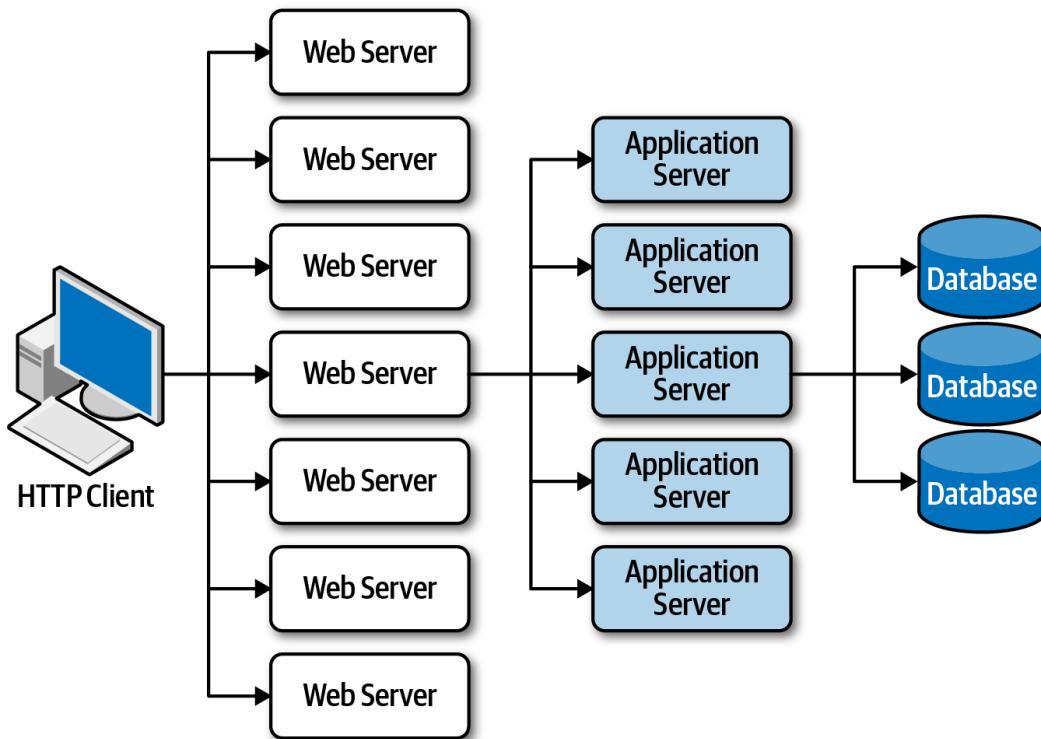


| Voordelen | Nadelen |
|--|---|
| In tegenstelling tot de <i>broker</i> topologie is er wel controle over de <i>workflow</i> | In tegenstelling tot de <i>broker</i> topologie een lagere schaalbaarheid |
| Door de aanwezigheid van een <i>mediator</i> is er meer inzicht in problemen. | In tegenstelling tot de <i>broker</i> topologie een lagere prestatie |

SPACE-BASED ARCHITECTURE STYLE (*DISTRIBUTED ARCHITECTURE*)

De *space-based architecture* is ontworpen om een probleem op te lossen binnen webapplicaties. Een veel voorkomend probleem binnen dit soort applicaties is de schaalbaarheid die plaats moet vinden door een gevarieerde hoeveelheid gebruikers. Een website kan van 100 bezoekers naar 100 duizend bezoekers gaan in seconden, deze variatie kan een architecturaal probleem zijn voor de software architect.

De architectuur van een standaard webapplicatie bestaat uit drie onderdelen, De webserver, de applicatieserver en de databaseserver. De structuur wordt weer gegeven in **Figuur**. Als het systeem moet worden opgeschaald is het gebruikelijk om extra *web servers* toe te voegen, dit is makkelijk te implementeren niet te duur. Het probleem wordt niet verholpen maar verplaatst naar de volgende laag, opeens moet de *application server* veel meer input verwerken, wat opnieuw een knelpunt creëert. Als de *application server* wordt versterkt met meer capaciteit is de database aan de beurt, op deze manier wordt het probleem niet alleen verplaatst maar kost het drie keer zoveel inspanning het gehele systeem aangepast moet worden.

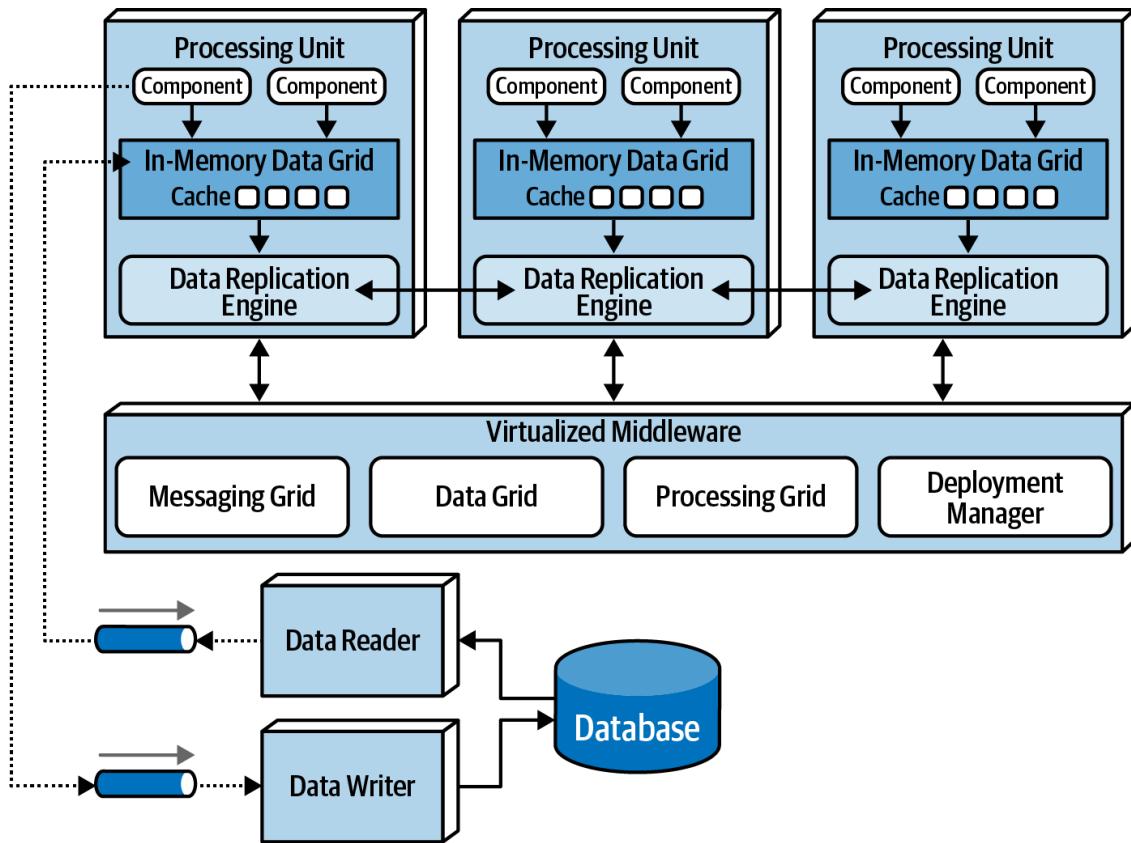


De *space-based* structuur is ontworpen om problemen van schaalbaarheid, gelijktijdigheid en kan worden gebruikt voor applicaties met een onvoorspelbare hoeveelheid gebruikers.

Het voorkomen van problemen is beter dan het aanpassen op basis van problemen.

Topologie

Hieronder is **Figuur** te zien hoe de standaard structuur van een *space-based* architectuur is opgebouwd. Het systeem bestaat uit een aantal componenten, de *processing unit* bevat de code waar de applicatie op is gebaseerd, het managen van verschillende *processing units* gebeurt door een *virtualized middleware*, de data wordt gestuurd door *data pumps* naar de database. De communicatie met de database gebeurt met de *data readers* en *data writers*.



Een probleem dat kan ontstaan bij de bovenstaande structuur is het concept van *data collisions*, waar data niet goed wordt opgeslagen waardoor sommige data wordt overschreven. Dit gebeurt door vertraging binnen de communicatie. Om te meten hoe frequent *collisions* voorkomen kan de volgende formule worden gebruikt. De **tabel** laat een voorbeeld zien van een berekening en welke waarde dit oplevert. Door de formule te inspecteren is te zien dat vertraging van individuele updates (gerepresenteerd door RL) een exponentieel effect heeft op de uiteindelijke *rating*. Het minimaliseren van de vertraging is essentieel voor de effectiviteit van het systeem.

$$\text{CollisionRate} = N * \frac{UR^2}{S} * RL$$

| Update Rate (UR) | 20 updates per seconde |
|---------------------------------------|-------------------------------|
| Number of instanties (N) | 5 |
| Grote van Cache (S) | 50 000 rijen beschikbaar |
| Vertraging (Replication latency) (RL) | 100 milliseconde |
| Hoeveelheid updates | 72 duizend per uur |
| Collision rate | 14,4 |
| Percentage | 0.02% |

Voor- en nadelen

| Voordelen | Nadelen |
|------------------|----------------|
|------------------|----------------|

| | |
|---|---|
| Flexibiliteit is erg hoog door grote hoeveelheid parallelle processen | Erg complexe implementatie |
| Goede prestatie door bovenstaand punt | Hoge kosten door licenties van complexe producten |
| Schaalbaarheid ligt hoog door de vele parallelle processen die plaats kunnen vinden | Door de parallelle processen is het testen een complex proces |

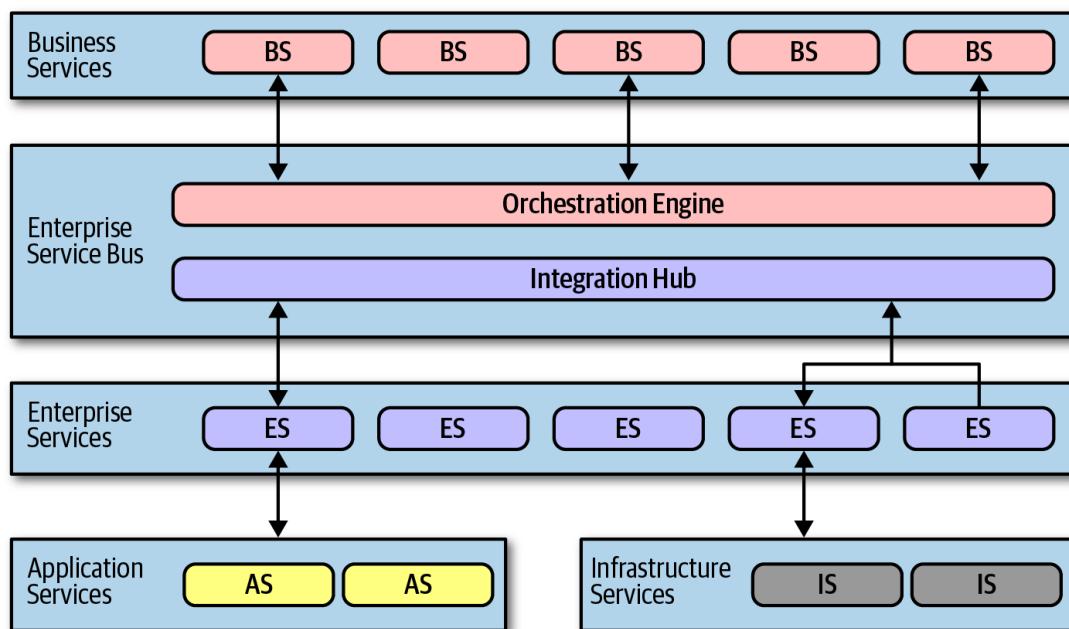
ORCHESTRATION-DRIVEN SERVICE-ORIENTED ARCHITECTURE (DISTRIBUTED ARCHITECTURE)

De oorsprong van de volgende architectuur komt vanuit een conflict tussen de bedrijfsbehoeftes en technische limitaties. In de jaren 90 zijn bedrijven enorm gegroeid door aspecten zoals de ontwikkeling van het internet, één van de meest impactvolle uitvindingen van de 20ste eeuw.

De architectuur maakt gebruik van verschillende *services* die de functionaliteiten met elkaar verbindt. De structuur is ontstaan door de stress en gelimiteerde bronnen die in de jaren 90 beschikbaar waren voor de development teams. Door bestaande componenten her te gebruiken in plaats van compleet nieuwe systemen te hoeven ontwikkelen bespaarde teams kostbare tijd en vooral geld. Software was vroeger extreem duur, zeker omdat *open source* software nog niet betrouwbaar genoeg was voor grote organisaties. Pas rond de begin van het nieuwe millennium werd de *open source* markt een concurrent van de gevestigde partijen.

Topologie

Zoals eerder genoemd was het idee achter de *orchestration-driven service-oriented architecture* is het hergebruiken van grote systemen. Elk component is uitgewerkt met deze filosofie. In **Figuur** is één van de variaties te zien die gebruikt wordt, er zijn meer mogelijkheden maar alle bevatten ze soortgelijke *services*.



Business services

Deze bevatten geen code, alleen maar input. Ze zijn bedoeld om de verschillende input binnen het domein van het bedrijf, deze onderdelen worden ook bepaald door het bedrijf en niet het development team.

Enterprise services

De *enterprise services* bevatten speciaal ontwikkelde code die als doel heeft om zo granulair mogelijk te zijn. Bij de term granulair wordt er gesproken over de manier waarop een systeem of een functie is opgebouwd en hoe het systeem de nodige taak uitvoert.

Een functie met een fijne granulair betekent dat de functie die alleen code bevat die daadwerkelijk nodig is, in plaats van veel code die extra onnodige complexiteit toevoegen aan de functie. Het doel is om een balans te vinden zodat functies of hele systemen in andere project opnieuw gebruikt kunnen worden.

Door de vele veranderingen binnen software development en de technieken die worden gebruikt is het idee van *enterprise services* achterhaald.

Application services

Het concept van granulariteit is niet zo even sterk van toepassing binnen *application services*. Binnen deze *services* staan de functionaliteiten die van toepassing zijn voor het specifieke probleem.

Infrastructure services

De *infrastructure services* bevatten functionaliteiten die worden gebruikt om het systeem te controleren, denk aan monitoring en het loggen van activiteiten.

Orchestration Engine

Als het systeem een mens is, is de *orchestration engine* het hart, dit beheert de inkomende data en functionaliteit van de andere *services*. De *engine* bepaalt de relatie tussen de *services*, hoe ze zijn verbonden met elkaar en welke transacties plaats kunnen vinden.

Waarom het niet werkt

Hoewel het idee om verschillende *sevices* te gebruiken klinkt als een goed idee maar in de realiteit loopt het niet zo. Er zijn een paar aspecten die voor problemen zorgen bij deze specifieke architectuur.

Niveau van granulariteit

Het kost veel tijd om een systeem zo te bouwen dat zoveel mogelijk onderdelen opnieuw gebruikt kunnen worden in hele andere systemen. Dit zorgt voor veel druk

op de ontwikkelaars en maakt het systeem zelf onnodig complex, omdat het niet mogelijk is om directe functionaliteiten in te implementeren.

Conway's law

De wet van Conway (Conway's law) beschrijft het fenomeen waarbij de architectuur van een systeem de structuur van de organisatie gaat nabootsen. Een onderzoek uit 2013 genaamd '*A Decade of Conway's Law: A literature review from 2003-2012*' laat zien dat de wet al meer dan 45 jaar wordt gebruikt en dat de daadwerkelijke betekenis van de wet verschilt per bron. Het boek '*Fundamentals of software architecture*' interpreteert de wet als een overkoepelend gedrag dat door een hele organisatie plaatsvindt en beslissing op elke gebied van ontwikkeling beïnvloed.

De wet is van toepassing bij de *service base* architectuur door de invloed die de *orchestration engine* heeft op het gehele project, waarbij de wet van Conway zegt dat dit deel een grote invloed heeft op de ontwikkeling van het gehele project. Dit is een probleem wanneer er wordt gewerkt met verschillende services die afkomstig zijn van andere projecten en teams. Er ontstaat een bureaucratisch knelpunt binnen het systeem dankzij de structuur van het systeem.

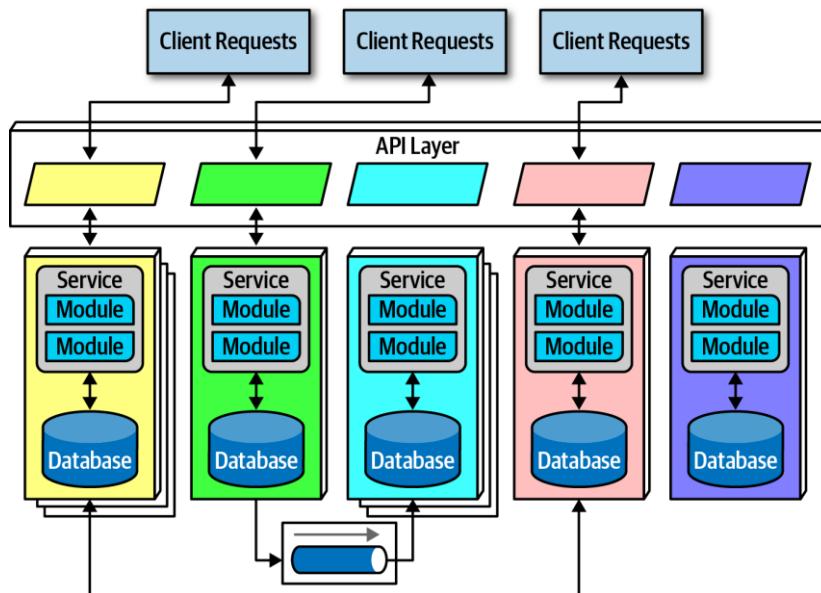
MICROSERVICES ARCHITECTURE (*DISTRIBUTED ARCHITECTURE*)

De *microservices* architectuur is één van de bekendste structuren die in de laatste jaren steeds populairder is geworden. Het idee achter *microservices* is geïnspireerd door *domain-driven design (DDD)*. DDD is een set van principes en structuren die proberen het gat tussen de bedrijfseisen en de code te verkleinen.

Eén van de concepten van DDD dat een basis is voor de *microservices architecture* is *bounded context*, hierbij worden alle onderdelen die een systeem gebruikt in één context geplaatst, zodat alleen de nodige code en logica wordt gebruikt. Hoewel die positief klinkt is er het probleem van *coupling*, wat zorgt voor een onnodig complex systeem.

Topologie

Een algemene topologie van de *microservices* structuur is te zien bij **Figuur**. Het is duidelijk te zien dat elke service zijn logica en opslag zelf beheert.



Distributed

Zoals zichtbaar is in **Figuur** heeft elke service zijn eigen proces, vroeger betekende dit dat er een fysieke computer dat proces moest draaien. Met technologieën zoals *Virtual Machines (VM)*, een techniek waarbij er meerdere software instanties aanwezig zijn op één fysieke computer (ook wel de *host* genoemd) en *containers* (techniek waarbij alle nodige code op één plek wordt samengevoegd) is het mogelijk om gebruik te maken van het concept *multi-tenancy*.

Gartner definieert *multi-tenancy* als: "Een architectuur waarbij meerdere instanties van één of meerdere applicaties opereren in dezelfde omgeving".

Multi-tenancy is een populaire structuur binnen *cloud computing* en wordt gezien als de standaard architectuur voor het ontwerpen van een goede *Software as a Service (SaaS)* applicatie.

Bounded context

Zoals eerder benoemd is het *bounded context* één van belangrijkste principes binnen *microservices*. Voor veel software architecten is het een concept dat veel invloed heeft op de besluiten die worden gemaakt. Bij *bouned context* wordt er gebruik gemaakt van duplicatie van code in plaats van koppelingen binnen code, dit is wennen voor developers die nog geen ervaring hebben met *microservices* en soortgelijke *distributed systems*.

Data isolatie

Een ander concept binnen *microservices* is het gebruik van aparte databases per service. Het is gebruikelijk om één centrale database te hebben dat dient als de hoofdopslag die leidend is voor alle data. Binnen de *Microservices* architectuur heeft elke service zijn eigen keuzes als het gaat om hoe data wordt opgeslagen, dit heeft

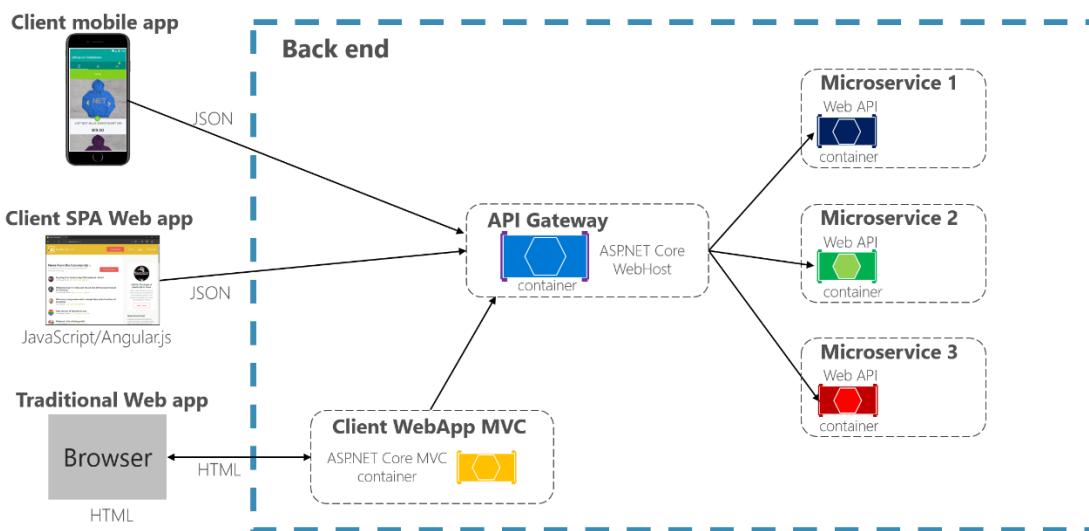
zowel voor als nadelen. Een nadeel van dit systeem is dat het de opslag van data complex maakt en een extra valkuil toevoegt waarop het systeem kan falen.

Het grote voordeel van dit concept is dat de datamogelijkheden compleet naar eigen wens kunnen worden ingericht per *services*, de soort opslag, de hoeveelheid en de prijs kunnen per *service* worden aangepast zonder de rest van het systeem lastig te vallen, voor grote projecten kan ik veel flexibiliteit leveren.

API Layer

In de meeste variaties van de architectuur wordt er gebruik gemaakt van een *API layer*. Dit is een 'laag' tussen de *UI* en de daadwerkelijke *sevices*. Het is niet verplicht, als het niet wordt gebruikt worden er verbindingen direct naar de *services* gemaakt. In **Figuur** is te zien hoe een *API gateway* wordt toegepast.

Using a single custom API Gateway service



Hoewel een *API layer* niet verplicht is voor de werking van het systeem wordt het sterk aangeraden om de schaalbaarheid van het systeem hoog te houden. Als er geen gebruik wordt gemaakt van een soortgelijke laag moeten de koppelingen tussen de *clients* en het systeem direct worden verbonden aan de *services* zelf, dit zorgt voor een paar problemen.

Vaak wordt er bij een vraag naar data niet gebruik gemaakt van één *service*, maar meerdere. Dit betekent dat er meerdere *calls* gemaakt moeten worden die niet alleen meer bandbreedte opneemt maar ook het systeem onnodig complex maken, daarnaast is er een directe afhankelijkheid tussen de *client* en het systeem. Als er aanpassingen zijn binnen de verschillende *services* heeft dit direct impact op de *client*, dit maakt het uitbreiden of aanpassen van het systeem een stuk meer weer en voegt nog extra complexiteit toe.

Communicatie

Het kiezen van de juiste communicatie binnen het systeem is per project afhankelijk en kan een moeilijke beslissing zijn voor architecten. Er zijn twee opties, *synchronous* (synchroon) of *asynchronous* (asynchrone) communicatie. Bij *synchronous* moet er door het systeem worden gewacht totdat een antwoord is ontvangen totdat een nieuwe vraag kan worden gestuurd, bij *asynchronous* kunnen er meerdere *requests* worden gedaan die tergelijktijd door het systeem worden verwerkt. Asynchrone communicatie kan extra snelheid en schaalbaarheid bieden aan een systeem als dit goed wordt uitgevoerd, dit komt met een extra complex systeem dat meer valkuilen toevoegt. De meest gebruikte manier van communicatie is *protocol-aware heterogeneous interoperability*. Deze communicatie methode bestaat uit drie onderdelen.

Protocol-aware betekent dat er binnen het systeem dezelfde protocol van communicatie gebruikt wordt, omdat elke *service* ander is ingericht is het belangrijk om wel een gelijke communicatiemethode toe te passen. Een voorbeeld van een synchrone communicatie protocol is HTTP/HTTPS.

Heterogeneous (heterogenen) betekent letterlijk: opmakend uit verschillende mensen op dingen. Binnen een heterogene omgeving wordt er vanuit gegaan dat er verschillende soorten eenheden aanwezig zijn. Binnen *microservices* is dit het geval sinds elke *service* anders kan worden ingericht.

Met *interoperability* wordt aangegeven dat de verschillende *services* met elkaar communiceren om data te versturen.

Voor- en nadelen

| Voordelen | Nadelen |
|---|---|
| De schaalbaarheid van het systeem ligt hoog door de losse <i>services</i> . | De prestatie van het systeem is erg afhankelijk van verschillende factoren, zoals het netwerk en beveiliging. |
| De flexibiliteit van het systeem is hoog omdat de <i>services</i> los van elkaar kunnen worden ingericht. | Het ontwerp en implementeren van het systeem niet eenvoudig. |
| De modulariteit van het systeem is hoog dankzij de losse <i>services</i> . | |

KIEZEN EN DE JUISTE ARCHITECTUUR

Het kiezen van de juiste structuur is een taak waar veel verschillende onderdelen bij komen kijken. Als architect zijn er ontelbaar verschillende manieren waarop de naar een situatie gekeken kan worden. Hier worden een paar onderdelen opgenoemd die invloed hebben op de keuze die gemaakt moet worden.

Het domein

In welk domein vindt het project zich? Het identificeren en begrijpen van het domein is één van de eerste stappen waar een architect zich bezig mee houdt. Door deze analyse kunnen de karakteristieken van de architectuur worden bepaald.

Architectuur van de data

Samen met het team moet de data worden onderzocht. Welke soort data er wordt gebruikt en in hoeverre dit impact heeft op de structuur.

Organisatie factoren

De vele externe factoren buiten het project hebben veel invloed op de uiteindelijke keuzes die gemaakt moeten worden. Het budget wat beschikbaar is, de impact die het systeem heeft op de medewerker en of klanten van het bedrijf en wanneer de verwachte deadlines zijn.

Kennis van het proces en het team

Dieper ingaand op het vorige aspect. Vaak wordt er vanuit gegaan dat een medewerker met de titel 'developer' een bepaald niveau van kennis heeft over zijn veld. Maar de realiteit is dat er een groot niveau verschil kan zijn dat ook impact heeft op de hoeveelheid kennis die beschikbaar is. De motivatie en drang om te innoveren zijn ook factoren die een rol kunnen spelen bij het bepalen van een architectuur.

UITWERKEN ARCHITECTUUR

Het gebruik van een *Proof of Concept* is een goede methode om op een simpele manier de waarde en valkuilen van een architectuur te laten zien. Een PoC moet de hoofdstructuur van het systeem bevatten en worden ontwikkeld met de juiste regels en principes die binnen het development team worden gebruikt.

CONCLUSIE

In dit onderzoek worden de vele aspecten van software architectuur benoemd en uitgewerkt aan de hand van voorbeelden en vergelijkingen.

VERWIJZINGEN

BIJLAGE 4: SYSTEEM INTEGRATIE METHODES

ONDERZOEK NAAR DE MOGELIJKHEDEN EN UITDAGINGEN VOOR HET KOPPELEN VAN APPLICATIES VIA VERSCHILLENDE INTEGRATIEMOGELIJKHEDEN.

INLEIDING

Een gemiddeld bedrijf heeft 1295 *cloud-services* in gebruik, dit is één van de observaties die Netskope heeft uitgebracht in de 2019 editie van hun *Cloud Report*. Hoewel Netskope als *Cloud provider* graag de groei van hun eigen markt wil laten zien, zijn zij niet de enige die dit soort observaties maken. Gartner, het grootste analistenbureau van de wereld en Fortune Business Insight, een groot markt analyst gevestigd in India rapporteren allebei een grote groei binnen de *Software as a Service* (SaaS) markt, waarbij een groei wordt verwacht van 25% voor 2028.

De groei van deze *services* zie je niet alleen terug in de cijfers, ook in het dagelijks leven komen mensen steeds meer in aanraking met *cloud* applicaties. Denk maar aan Office 365, Slack, MS Teams of Google services zoals Google Drive of Photos.

De groei in deze markt zorgt ervoor dat er meer en meer vraag is naar SaaS-oplossingen, dit zorgt ervoor dat bedrijven zoals eerder vernoemd gebruik maken van honderden of zelfs meer dan duizend services die allemaal hun eigen *layout* en architectuur hebben. Deze vorm van digitale transformatie, waarbij bedrijven overstappen naar nieuwe technologieën loopt in de praktijk vaak niet goed.

In eind 2020 concludeerde de Boston Consulting Group (BCG) dat 70% van alle pogingen tot digitale transformatie falen, waarbij dit in de Verenigde Staten in voor zo'n 260 miljard dollar aan schade kosten.

Als bedrijf welvarend wil zijn is het belangrijk om inzicht te hebben in de inkomende en uitgaande datastromen van al deze *services*. Het liefst heb je de data van verschillende applicaties gecentraliseerd, zo voorkom je verwarring en bespaar je tijd. Om dit te doen heb je integraties nodig.

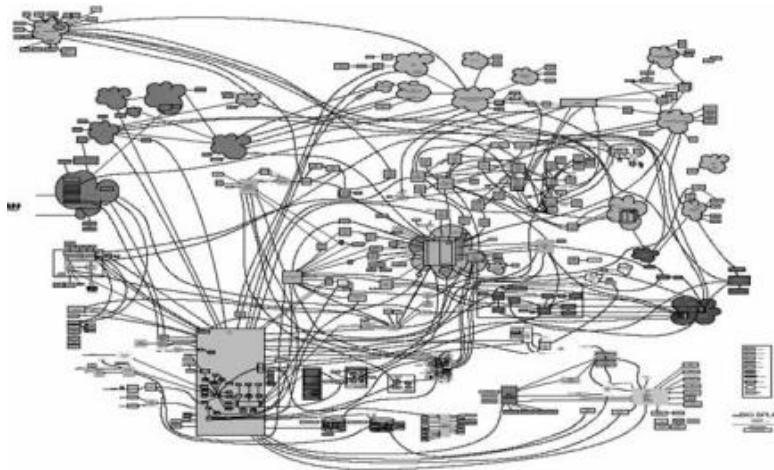
SOFTWARE INTEGRATIE

Software integration (integratie) is een tak van software development waar verschillende applicaties aan elkaar worden gekoppeld. Gartner definieert het als: Specifieke ontwerpen en implementaties die functionaliteiten of data binnen applicaties aan elkaar verbinden.

Het concept van integraties is erg breed en kan op veel verschillende manieren worden geïnterpreteerd, daarbij zijn er ook veel verschillende modellen die worden gebruikt bij het koppelen van systemen. De behoefte voor integraties is niet nieuw. In de jaren 80 hadden bedrijven al applicaties die een directe verbinden moesten

hebben met andere applicaties. Dit waren 1 op 1 verbindingen die verder geen niet schaalbaar waren, zogeheten *Point-to-Point* integraties.

Al deze individuele integraties maakten de IT-structuur van het bedrijf compleet onleesbaar. Zie **Figuur**



Sinds de jaren 80 zijn er verschillende modellen en architecturen bedacht om integraties te versimpelen en tegelijkertijd te verbeteren. Deze modellen proberen de valkuilen van het integreren van software te vermijden. Volgens een onderzoek gedaan door Al Ain en de SZABIST universiteit zijn er zes valkuilen bij het ontwikkelen van software integraties:

1. Te weinig technische kennis
2. Niet inzien dat integratie software geen product is, maar een architectuur
3. Verwaarlozen van beveiliging, prestatie en monitoring
4. Het combineren van integraties met andere projecten
5. Integraties ontwikkelen zonder strategie
6. Slechte communicatie binnen het team

Omdat het bouwen van eigen software duur is moet er rekening worden gehouden met deze valkuilen. Het negeren ervan kan leiden tot een architectuur die fundamenteel foutief is en resulteert in een verslechterde datastream.

WAAROM INTEGRATIES

Bij de term integraties wordt er vaak alleen gedacht aan het koppelen van twee systemen puur voor het versturen en ontvangen van data, een directe verbindingen of een centrale plek waar data samen komt en verder wordt verwerkt. Dit is inderdaad één van de redenen waarom integraties nodig zijn binnen een organisatie, daarnaast zijn er nog een aantal principes waar integraties bij helpen.

Consistentie

Eén van de voordelen als er integraties worden gebruikt is de consistentie van de datastromen binnen een organisatie. Als er meerdere systemen met allebei een eigen *dataset* zijn is het synchroniseren van deze data een tijdsintensief proces.

Door een integratie op te zetten tussen deze systemen kan data automatisch gesynchroniseerd worden en kan men zeker zijn dat de data in beide systemen accuraat en *up-to-date* is.

Verrijking van data

Met de term 'verrijking' wordt niet alleen de toevoeging van data bedoeld maar ook het gebruik ervan.

Stel er zijn twee systemen in een organisatie die allebei dezelfde hardware monitoren, het gaat hier over een communicatiernetwerk van organisaties. Het ene systeem (systeem A) heeft informatie over het netwerk zelf zoals de locatie en alle hardware die zich in dit netwerk bevindt. Het tweede systeem (systeem B) heeft externe informatie over de klanten binnen dit netwerk, informatie zoals wie de klant is en welke diensten ze leveren.

Er is geen directe koppeling tussen de data, dit betekent dat het netwerkmonitoringsysteem (systeem A) niet weet bij welke klant het netwerk hoort. Op dezelfde manier weet het CRM systeem (systeem B) niks over het netwerk waar de klant gebruik van maakt. Alleen de medewerkers van de organisatie weten de link tussen deze twee aparte *datasets*.

Als er binnen de organisatie een integratie wordt ontwikkeld tussen deze twee systemen is het mogelijk om deze data met elkaar te verbinden. Binnen het CRM systeem (systeem B) is het opeens duidelijk welk netwerk er wordt gebruikt en kan informatie zoals de status van de hardware direct zichtbaar zijn voor de engineering medewerker. Support medewerkers in het algemeen hebben veel voordeel bij het samenbrengen van data, sinds de klant zelf vaak geen duidelijk beeld heeft van hardware die zij gebruiken. Op deze manier wordt data binnen een organisatie 'verrijkt' zonder nieuwe data te hoeven genereren, hoewel dit wel een optie kan zijn.

Groei in de markt

Zoals benoemd in de inleiding is er een grote groei binnen de markt van *cloud* diensten en SaaS applicaties. Deze groei gaat gepaard met de groei van software integratie software. Eén van de *cloud* oplossingen als het gaat om software integraties is *Integration Platform as a Service* (iPaaS).

De iPaaS markt is sinds het ontstaan in eind 2000 alleen maar aan het groeien. Volgens ReportLinker zal de marktwaarde van iPaaS groeien van 3.7 miljard dollar in 2021 naar 13.9 miljard in 2026, dit is een samengesteld jaarlijks groeipercentage van 30.3%. De reden voor deze groei ligt bij steeds groter wordende competitie binnen velen sectoren, door deze competitie gaan meer grote bedrijven en Midden en Klein Bedrijf (MKB's) investeren in iPaaS software. Later in dit onderzoek wordt de structuur van iPaaS en andere integratie mogelijkheden besproken.

ONDERDELEN VAN SOFTWARE INTEGRATIES

Om een integratie te implementeren of te ontwikkelen zijn er een paar stappen die nodig zijn om de nodige context van een systeem te krijgen. De stappen die

hieronder worden beschreven zijn de taak van de software architect die de leiding heeft in het ontwerpen en organiseren van software projecten.

Opstellen van systeemeisen

Voor een project kan beginnen moeten de systeemeisen worden verzameld. Het doel van *system requirements* is om een duidelijk beeld te geven van de behoeftes en wensen die zijn opgesteld door de *stakeholders*.

Het opstellen van deze eisen gebeurt aan de hand van observaties en interviews met medewerkers binnen de organisatie. Het *NYS Project Management Guidebook* beschrijft het verzamelen van systeemeisen als een spons. Het team moet alle mogelijke data verzamelen, hierbij is de *product owner* of klant van het systeem de belangrijkste bron. Sommige data lijkt misschien niet even nuttig voor het project, toch moet dit verzameld worden.

Situatie analyse

Na het opstellen van de verschillende systeemeisen moet de huidige situatie van het bedrijf in kaart worden gebracht. De reden hiervoor is omdat de zwakke en sterke punten van een organisatie veel invloed kunnen hebben op de scope van het project. Denk aan de grote en niveau van het development team, de hiërarchie binnen het bedrijf, de mogelijke kosten en de tijd die beschikbaar.

Deze punten kunnen op verschillende manieren worden samengevat. Een populaire methode is de SWOT-Anlyse (Strengths, Weaknesses, Opportunities en Threats) of Sterkte-zwakteanalyse, waarbij de hierboven benoemde aspecten worden verdeeld in de verschillende categorieën.

Naast de huidige situatie is het ook mogelijk om de doelstellingen en motivatie van het bedrijf te vermelden, hiervoor wordt het Hoshin Kanri model gebruikt. Dit is een methode die bestaat uit verschillende visualisatie methodes om de doelen van een organisatie vast te stellen.

De conclusies uit dit onderzoek wordt gebruikt om de scope van het project verder te optimaliseren.

Ontwerpen systeem /layout architectuur

Als de scope van het project goed is bepaald kan er worden ontworpen. Deze ontwerpen hebben als doel om het systeem te visualiseren en context te geven voor de developer en software architecten die het systeem gaan realiseren. Een uitgebreide ontwerp principe die gebruikt wordt is het C4 model, dit is een model dat bestaat uit vier lagen die steeds meer abstractie aan het ontwerp toevoegen.

Naast dat model is het belangrijk om verschillende UML (Unified Modeling Language) diagrammen toe te passen om de context verder uit te werken. De *product owner* moet sommige ontwerpen ook kunnen begrijpen, er moet een balans zijn tussen complexiteit en hoeveelheid informatie dat zichtbaar is binnen het ontwerp.

Opstellen van systeem management documentatie

Voordat er daadwerkelijk kan worden gebouwd aan de oplossing is het belangrijk om de externe management op orde te hebben. Dit houdt in dat de ontwikkeling van het systeem goed verloopt en het voor iedereen duidelijk is wie wat doet en welke problemen zich voordoen. Het opzetten van de management documentatie is per organisatie anders en is erg afhankelijk van de ervaring binnen het team.

Het gebruiken van de Scrum methode is een populaire manier om op korte termijn producten op te leveren, bij Scrum worden er sprints van tussen 1-4 weken gebruikt waarin een deel van een product volledig afgerond moet zijn. Door *daily standups* en *sprint reviews* krijgt de *product owner* inzicht in de progressie van het systeem en mogelijke problemen die zich voordoen.

Het duidelijk documenteren van de voortgang is essentieel. Binnen het team moeten er keuzes worden gemaakt over welke *tools* er gebruikt kunnen worden. Github, Slite, Confluence, Trello, Slack, Discord, MS Teams ect. zijn allemaal mogelijke opties en het combineren van een paar services is gebruikelijk.

Implementeren van systeem integraties

Het meest tijdsintensieve onderdeel van het project is het implementeren van de integraties en alle code die daarbij hoort. Door een goede voorbereiding is het duidelijk welke onderdelen en er en niet moeten worden gebouwd. Door een goede documentatie en regulier contact tussen het team en de *product owner* is het project beter te onderhouden en kunnen veranderingen snel worden doorgevoerd.

Testen van systeem integraties

Voordat het systeem gebruikt kan worden door de medewerkers van het bedrijf moet het eerst worden getest. Afhankelijk van het soort systeem dat wordt ontwikkeld moet er gebruik worden gemaakt van bepaalde testmethodes. Bij integraties is het gebruik van API's populair. Met een API test kan de functies en limitaties van een API duidelijk worden. Als een onderdeel van het systeem is afgerond is het belangrijk om dit te testen met nep data om de verschillende aspecten te controleren. Het gebruik van Unit testen is aangeraden omdat het gebruikt kan worden bij kleine onderdelen van een systeem.

Zodra het testen is afgerond kan het systeem worden opgezet binnen de organisatie of bij klanten.

SOORTEN INTEGRATIES

Er zijn veel manieren waarop de integraties kunnen worden uitgevoerd, nieuwe technieken worden bedacht aan de hand van problemen die voordoen. Hoewel het lijkt alsof er een groot verschil zit tussen verschillende methodes is het concept vrijwel altijd hetzelfde. Een integratie verbindt twee systemen of onderdelen van systemen, de 'verbetering' zit in de manier waarop valkuilen worden behandeld. Binnen dit onderzoek worden vier verschillende soorten integraties besproken, dit zijn

de *Point-to-Point* (P2P), *Hub-and-Spoke*, *Enterprise Service Bus* (ESB) en *Cloud* oplossingen (zoals iPaaS).

POINT-TO-POINT

Point-to-point integration (one-to-one) wordt vaak afgekort door P2P en is de meeste simpele vorm van integratie, bij deze methode wordt er een directe verbinding gemaakt tussen twee systemen. Met deze verbinding is het mogelijk om data tussen de twee systemen te sturen. Het gebruik van P2P is overal te zien. Door de relatief eenvoudige implementatie is het een goede optie voor problemen waarbij een simpele link tussen twee databronnen moet worden ontwikkeld, er is geen extra database nodig.

Een voorbeeld van een P2P integratie is het updaten van een waarde op basis van een ander systeem. In het voorbeeld bij het hoofdstuk 'Verrijking van data' wordt een systeem genoemd dat netwerken beheert en een systeem dat klantgegevens beheert. Data vanuit systeem A kan worden opgehaald en opgeslagen binnen de database van systeem B door middel van een P2P integratie, de verbinding is direct en er is geen verdere opslag vereist.

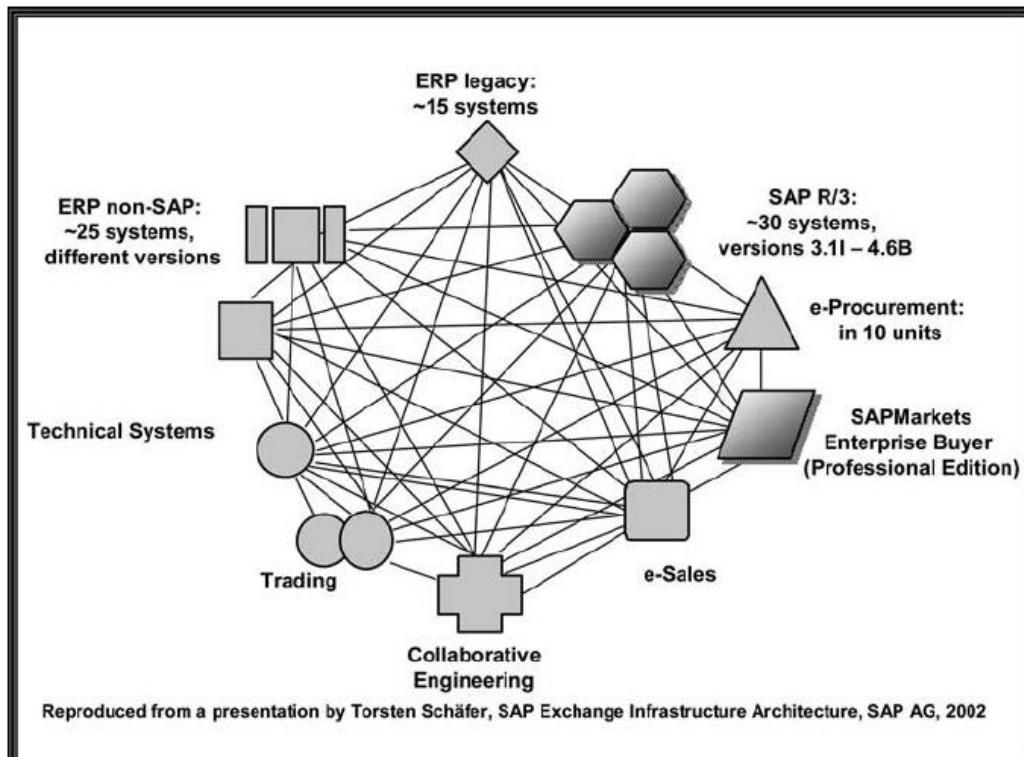
De stappen die worden uitgevoerd bij een *point-to-point* integratie zijn als volgt:

1. De afzender (*Sender*) stuurt een verzoek naar de ontvanger (*receiver*), dit verzoek wordt in een wachtrij (*queue*) geplaatst.
2. De integratie tussenpersoon (*broker*) stuurt het bericht naar de juiste ontvanger (*receiver*). Voordat dit gebeurt is het mogelijk dat er nog aanpassingen plaatsvinden, denk aan het aanpassen van het datatype of het verrijking met informatie voor *logging*.
3. De ontvanger (*receiver*) ontvangt het bericht en kan de data verder verwerken.

Dit communicatie patroon wordt ook wel een *federated request pattern* genoemd.

TOPOLOGIE

In **Figuur** hieronder is een voorbeeld te zien van een organisatie die alleen maar gebruik maakt van *point-to-point* integraties, de verschillende vormen representeren de verschillende systemen die worden gebruikt. De lijnen zijn de 'verbindingen' tussen de systemen.



De eerste indruk bij het zien van deze afbeelding is voor de meeste verwarring, zelfs in een schematisch overzicht waar alle complexe code achterwegen is gelaten is het nog steeds geen touw aan vast te knopen. Van een afstand heeft het zelfs iets weg van spaghetti, dit kan wel kloppen omdat een structuur zoals hierboven vaak wordt beschreven als 'spaghetti code'.

Waar deze methode eenvoudig is voor het maken van een een simpele koppeling tussen twee systemen is de schaalbaarheid van P2P erg laag. Hoe meer integraties er nodig zijn hoe complexer het systeem wordt, het **Figuur** hierboven is een goed voorbeeld van de gevolgen bij als er alleen P2P integraties worden toegepast. Complexe code is nodig maar niet gewenst, hoe complexer een code is hoe moeilijker het is voor nieuwe medewerkers om snel functioneel te zijn binnen het team te werken. Naast de complexiteit is de structuur ook niet flexibel of snel.

DE N(N-1) NOTATIE

Om de schaalbaarheid van *point-to-point* integratie te noteren is er de $n(n-1)$ notatie (ook wel het n -kwadraat probleem). Bij deze notatie is 'n' het aantal systemen waar een verbinding moet komen. Als er vijf systemen moeten worden verbonden met elkaar zijn er dus 20 losse integraties nodig. In moderne systemen komt dit weinig voor maar deze notatie laat wel zien hoe snel de methode uit de hand loopt.

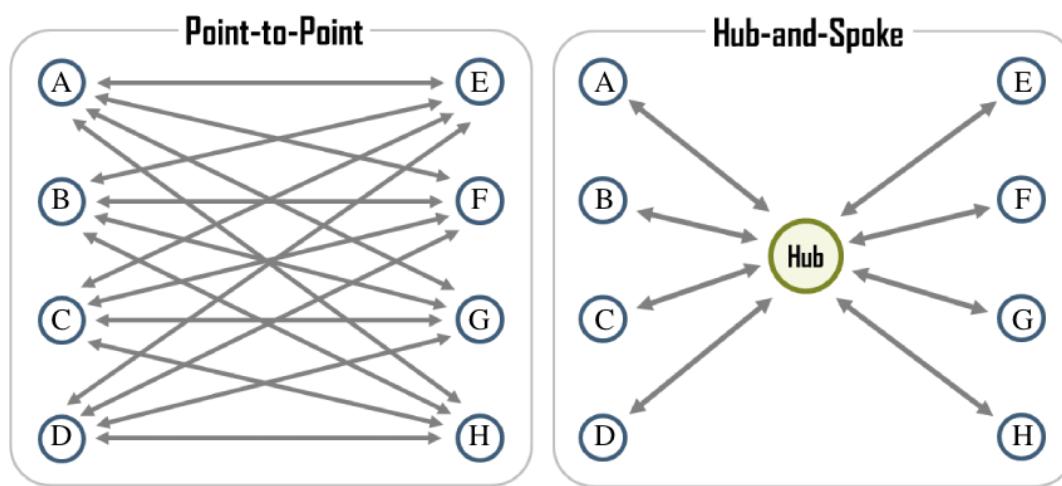
FRAGIEL

Een ander probleem waar op gelet moet worden als er gebruik wordt gemaakt van P2P integraties is de afhankelijkheid die aanwezig is. Omdat er een directe communicatie is tussen twee systemen moeten deze systemen beide werkend zijn om enige data te kunnen sturen/ontvangen, als één systeem niet meer werkt is stopt

de hele koppeling met werken. Bij het gebruiken van P2P integraties moet hiermee rekening worden gehouden.

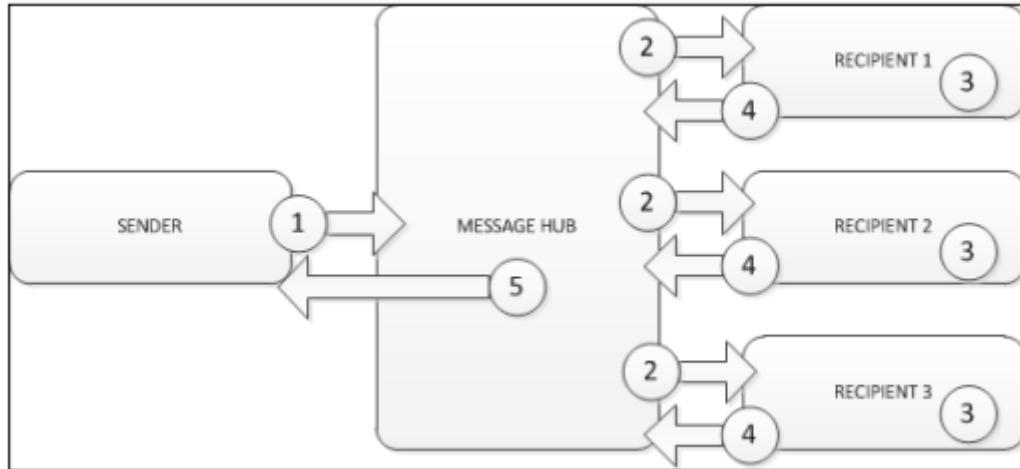
Waar kan *point-to-point* integratie gebruikt worden?

Binnen kleine organisaties met een paar systemen is het niet vreemd om P2P integraties toe te passen. De relatief eenvoudige implementatie is aantrekkelijk, daarbij dit soort integraties makkelijk te onderhouden. Voor organisaties met meer systemen wordt het concept van P2P uitgebreid naar het *hub-and-spoke* model. In **Figuur** is het verschil tussen de twee integratie methodes duidelijk zichtbaar, waarbij het *hub-and-spoke* model een stuk overzichtelijker is.



HUB-AND-SPOKE

Binnen grotere organisaties met complexe systemen is het doel van P2P net wat anders. Binnen deze systemen wordt P2P gebruikt om verschillende bronnen te koppelen aan een centraal integratieplatform die de data vanuit meerdere bronnen ontvangt en verwerkt. Door gebruik te maken van een degelijke structuur is het systeem niet afhankelijk van de werking van de losse P2P koppelingen, sinds data niet direct afkomstig is van de koppeling. Deze structuur is gevisualiseerd binnen **Figuur**. Het centrale platform wordt vaak een *message hub* genoemd. Deze structuur heet het *hub-and-spoke* model.



De term *hub-and-spoke* wordt ook vervangen door *Enterprise Application Integration* (EAI) of stertopologie en is een verzamelnaam voor veel verschillende structuren. In **Figuur** hierboven is algemene topologie te zien zoals beschreven in het boek 'Applied Architecture Patterns on the Microsoft Platform', de volgende stappen worden genoemd:

- (1) De afzender stuurt een verzoek naar meerdere ontvangers via de *message hub*.
- (2) De *message hub* beslist waar het verzoek heen moet op basis van informatie dat mee wordt gegeven vanuit de afzender, dit kan een simpel veld zijn of een verwijzing naar de locatie waar de ontvanger wordt bepaald.
- (3) De ontvangers ontvangen het verzoek en verwerken de data, in de meeste gevallen moet er een antwoordt terug naar de afzender worden gestuurd.
- (4) De ontvangers sturen het antwoord van het verzoek terug naar de *message hub* met daarbij de juiste success/failure codes (200 voor succes, 400-500 voor mislukking).
- (5) De *message hub* verzameld alle antwoorden en stuurt deze terug naar de afzender.

TOPOLOGIE

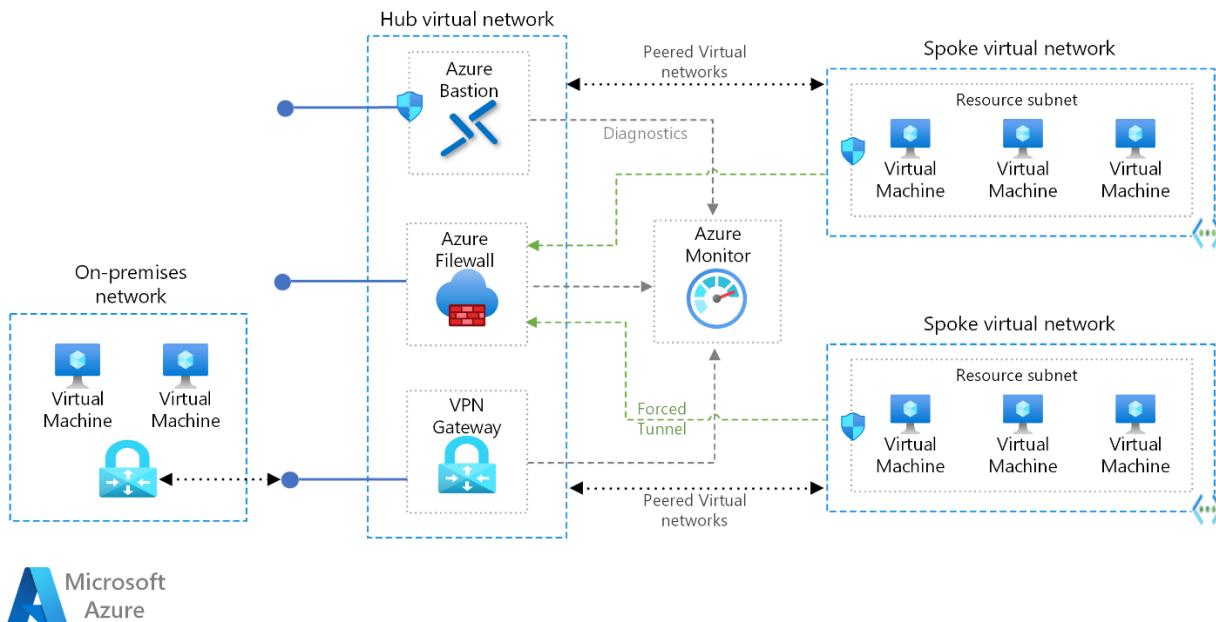
Er zijn veel verschillende manieren waarop een *hub-and-spoke* structuur kan worden opgebouwd, op basis van de behoeftes van de klant zijn veel variaties mogelijk. Om een beeld te geven over hoe de structuur wordt toegepast wordt hieronder de *hub-and-spoke* structuur van drie van de grootste *cloud providers* ter wereld beschreven met als doel om inzicht te bieden in de verschillen en gelijkenissen van deze modellen.

MICROSOFT AZURE

Microsoft Azure is één van de *cloud providers* ter wereld, met een jaarlijkse omzet van rond de 55 miljard euro. Azure biedt een structuur aan voor het opzetten van

virtuele *hub* en twee *spokes*. Het is mogelijk om meer onderdelen te implementeren via de Azure omgeving.

In **Figuur** hieronder is de structuur te zien, met daarin de volgende onderdelen:



Virtueel *hub* netwerk:

In dit netwerk waar de *on-premise* netwerken (op locatie) mee verbinden. Binnen het virtuele *hub* netwerk kunnen services worden gehost die gebruikt worden door de virtuele *spoke*-netwerken.

Virtuele spoke-netwerken:

De virtuele *spoke*-netwerken isoleren hun eigen werk van de andere *spokes*, deze netwerken kunnen bestaan uit meerdere lagen en zijn verbonden aan andere netwerken voor het gebruik van verschillende Azure services.

Peering van virtuele netwerken:

Binnen de structuur worden er zogenoemde 'peeringverbindingen' gebruikt, dit zijn verbindingen tussen twee of meer netwerken met een lage *latency* (vertraging). *Peering* is de methode waarbij communicatie kan plaatsvinden tussen netwerken zonder het internet te gebruiken.

Bastion-host:

De *bastion-host* is een service die Azure aanbiedt waarmee u toegang kunt krijgen tot virtuele machines via de webbrowser. Er wordt gebruik gemaakt van een beveiligde verbinding.

Azure Firewall:

Een service die Azure aanbiedt waarmee het netwerk wordt beveiligd.

Virtuele VPN-netwerkgateway of ExpressRoute-gateway.

De virtuele VPN-netwerkgateway is een service waar versleutelde data door wordt gestuurd. Er kan verbinding worden gemaakt met het *on-premise* netwerk.

Verbinding tussen spokes

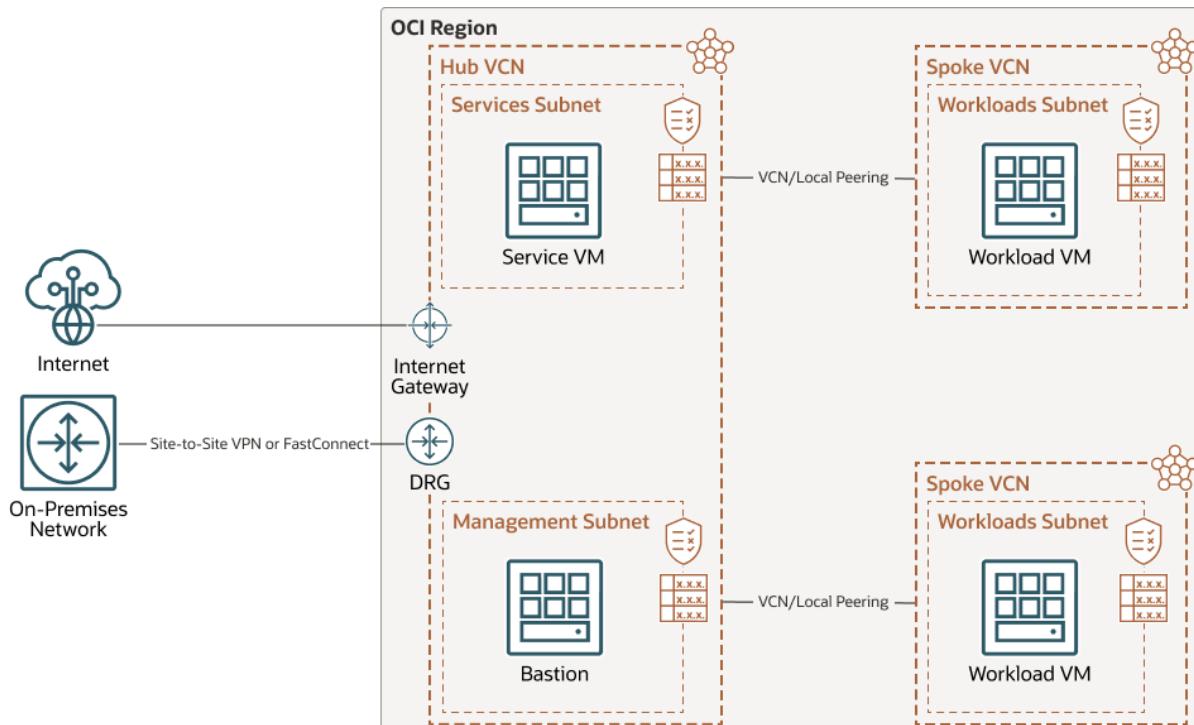
Als het nodig is om een communicatiemiddel op te zetten tussen twee *spokes* kan dit worden geïmplementeerd met de *Azure Firewall* of de *Azure VPN-gateway*. Bij het toevoegen van extra communicatiemogelijkheden moet er rekening worden gehouden met de bandbreedtelimieten die de *firewall* heeft.

Kosten bij dit netwerk

Binnen dit netwerk zijn er twee elementen waar extra kosten aan verbonden zitten. Dit zijn *Peering* van virtuele netwerken (minimaal 0,033 euro per gigabyte) en de *firewall* (1,17 per uur plus 0,015 per gigabyte).

ORACLE

Oracle is een groot Amerikaans bedrijf dat onder andere *cloud* diensten levert. De jaarlijkse omzet van de *cloud services* van Oracle is ronde 26 miljard euro. Oracle biedt een soortgelijke structuur aanwezig bij de Microsoft Azure omgeving, met eigen services om de gebruiker een soepele en beveiligde service te kunnen bieden. In **Figuur** is de topologie zichtbaar, met daarin de volgende onderdelen:



On-premises netwerk

Dit is één van de *spokes* in het netwerk, net als bij de Azure structuur represeneert dit het lokale netwerk dat wordt gebruikt door de organisatie.

Region / Regio

Een *region* (*Oracle Cloud Infrastructure region*) is een regio met één of meerdere fysieke *datacenters* (Een industrieel pand waar veel servers aanwezig zijn) . *Datacenters* worden binnen Oracle *availability domains* genoemd.

Virtual cloud network (VCN)

Het *virtual cloud network* is een netwerk dat door de gebruiker opgezet kan worden binnen de Oracle omgeving. Er is veel controle over de omgeving en onderdelen kunnen achteraf worden aangepast. de VCN kan worden opgedeeld in *subnets* die kunnen worden toegekend aan verschillende *availability domains*. Binnen deze architectuur wordt er gebruik gemaakt van een VCN *hub* en een meerdere VCN *spokes*.

Security list

Voor elke *subnet* kunnen er veiligheidsregels worden opgezet die het verkeer van en naar het *subnet* bepalen.

Route table

Een *route table* bevat de regels voor het verkeer buiten de VCN's en gaan normaal gesproken door *gateways*.

Dynamic routing gateway (DRG)

De *dynamic routing gateway (DRG)* is een virtuele router (verbint netwerken met elkaar) voor binnen en buiten een *virtual cloud network* zoals de *on-premise* netwerken van de gebruiker.

Bastion host

Net als binnen het netwerk van Azure wordt er gebruik gemaakt van een *Bastion host*. Binnen Oracle is dit een beveiligde omgeving waarvan uit verbinding kan worden gemaakt met het netwerk, dit kan door het gebruik van een *Demilitarized zone (DMZ)*. In de wereld van netwerken is dit een streng beveiligde buffer tussen twee netwerken met vaak één of twee *firewalls* voor nog meer bescherming.

Bastion service

Dit is een *service* die Oracle biedt om op een veilige manier tijdelijk toegang te krijgen tot beveiligde data. Deze service geeft de gebruiker de mogelijkheid om snel en veilig deze data te kunnen inzien.

Local peering gateway (LPG)

Met een *Local peering gateway (LPG)* is het mogelijk om te communiceren tussen twee VCN's via privé IP addresen.

Site-to-Site VPN

Met deze service is het mogelijk om de lokale netwerken van de klant te verbinden met de VCN's van Oracle, met het gebruik van IPSec (Internet Protocol Security) wordt encryptie toegepast op de data die wordt verstuurd.

FastConnect

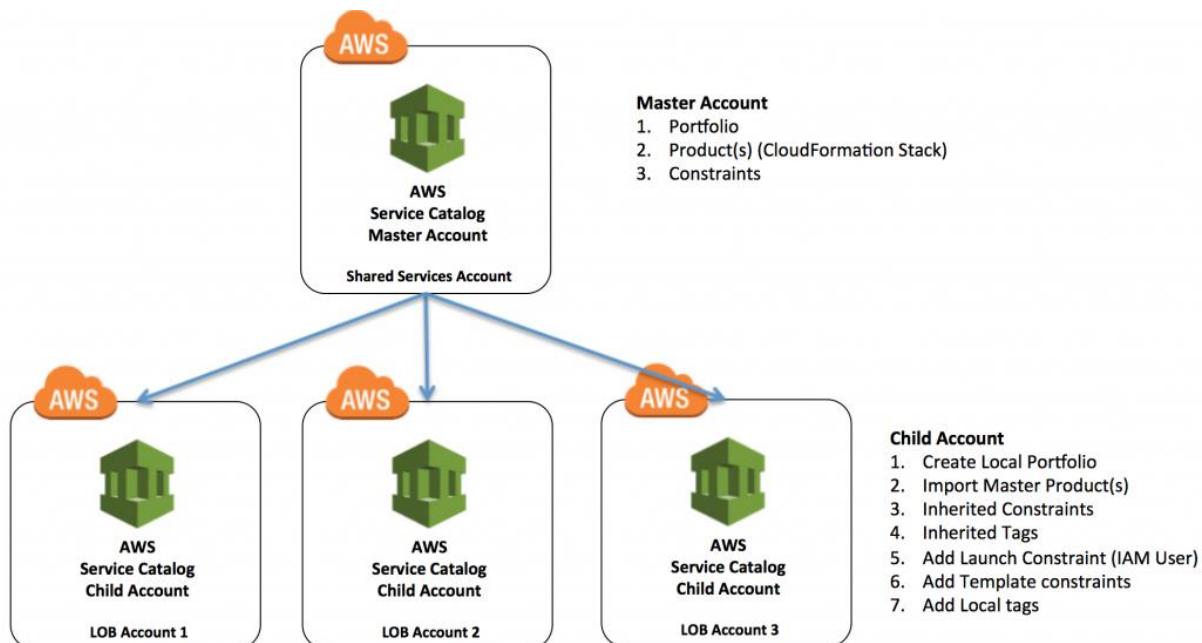
De *FastConnect* van Oracle is een simpele manier om verbindingen op te zetten tussen de *datacenter* en het Oracle netwerk.

Overwegingen

Naast de prijs van het draaien van het systeem en de *FastConnect* service is er geen extra prijs verbonden aan het gebruiken van dit netwerk. De snelheid binnen een regio blijft hetzelfde onafhankelijk van de hoeveelheid VCN's die gebruikt worden, verkeer buiten de regio's kan met vertraging komen.

AMAZON WEB SERVICES

Amazon Web Services (AWS) biedt net als de andere *providers* een *hub-and-spoke* model met verschillende opties voor het personaliseren van het systeem. In **Figuur** is te zien de structuur is opgebouwd, binnen AWS is het ook mogelijk om het model toe te passen met meerdere accounts. In de huidige structuur bevinden zich de volgende elementen:



Product

Met een *product* wordt één van de *services* bedoeld die AWS aanbiedt, zoals VPC, web servers of databases die binnen AWS kunnen worden ingezet.

Portfolio

De volledige naam is *AWS Service Catalog portfolio*. Dit is een verzameling van producten en bijbehorende configuraties.

Constraint

De *constraints* zijn een verzameling van restricties die horen bij de verschillende services die AWS biedt.

Provisioned Product

Ook wel een *landscape* genoemd, dit is een collectie van draaiende *services* die zijn opgestart samen met het *product*.

WAAR KAN HUB-AND-SPOKE INTEGRATIE GEBRUIKT WORDEN?

Het gebruik van het *hub-and-spoke* model binnen de *cloud* heeft een aantal voordelen, ten eerste is er in vergelijking met een traditionele implementatie en kosten vrijwel geen extra werk dat verricht moet worden, alle belangrijke aspecten zoals veiligheid en prestatie wordt al voor de gebruiker geregeld. Om deze reden kan de structuur goed gebruikt worden voor het opzetten van een ontwikkelomgeving. Het kan ook worden gebruikt om verschillende onderdelen binnen een systeem te isoleren.

Het grote probleem bij het gebruik van een *hub-and-spoke* model is het knelpunt dat zich vormt. Hoewel lossen P2P koppelingen wellicht minder effect hebben op de werking van het systeem is er bij deze structuur één punt waar alle communicatie doorheen moet. De *hub* vormt een groot knelpunt waar rekening mee moet worden gehouden. Zelfs binnen de hypermoderne *cloud* oplossingen waar de beste hard- en software wordt gebruikt is er een afhankelijkheid van de *hub*. Het knelpunt wat ontstaat is ook slecht voor de prestatie van het model.

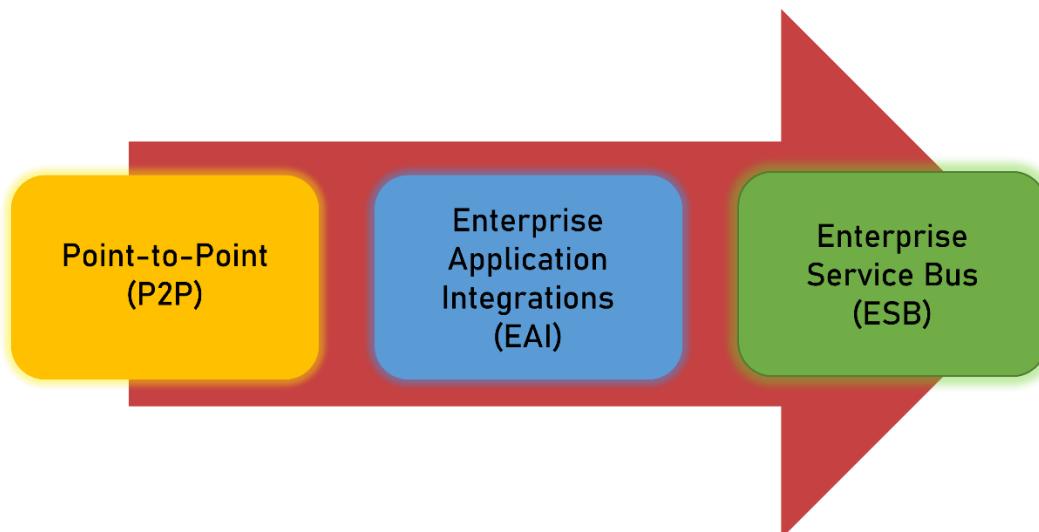
ENTERPRISE SERVICE BUS (ESB)

Een *Enterprise Service Bus (ESB)* is een structuur die tussen de verschillende applicaties ligt en de communicatie tussen deze applicaties regelt. De term is relatief nieuwe en wordt vaak aangevuld met architectuur *Service Oriented Architecture (SOAs)*. Dit is een type architectuur waar de verschillende stukken van het systeem zijn opgedeeld in zogeheten *domain services*. Deze services zijn meestal verbonden met één centrale database.

De beste manier om een ESB te beschrijven is als een collectie aan verschillende structuren die samen een complex en krachtig systeem vormen.

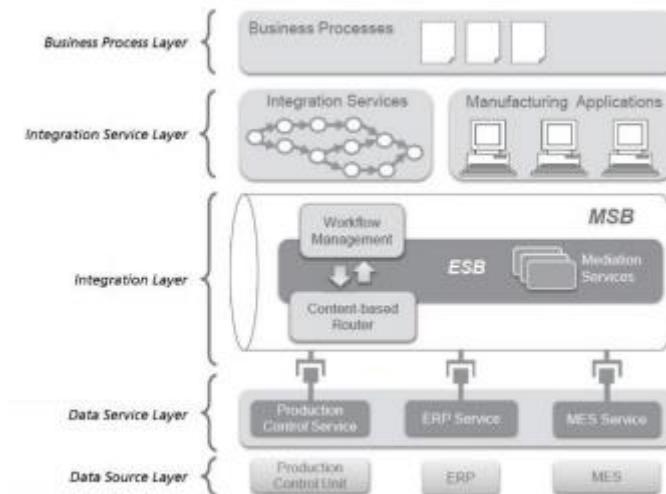
Sinds de jaren 90 heeft er zich een evolutie plaats gevonden binnen de wereld van software integraties. In het begin werden er alleen gebruik gemaakt van simpele *point-to-point* integraties tussen twee systemen, zoals benoemd in het hoofdstuk 'point-to-point' zitten hier veel limitaties aan. Om de integratiestructuur van organisaties minder complex te maken werden er andere manieren bedacht waarop P2P gebruikt kan worden, hieruit is de *Enterprise Application Integration* (EAI) of terwijl het *hub-and-spoke* model ontstaan, dat wordt beschreven binnen het vorige hoofdstuk. Deze structuur wordt nog steeds gebruikt maar vaak binnen kleine organisaties of met de hulp van services zoals de *cloud*.

Het grootste verschil tussen het toepassen van een EAI met de hulp van het *hub-and-spoke* model en het gebruik van een ESB is de manier waarop de ESB is ingericht. Een *Enterprise Service Bus* maakt gebruik van de SOA principes, dit is een structuur die gebruik maakt van een *distributed architecture*, een structuur waarbij verschillende componenten los staan van elkaar en samenwerken door de communiceren via verschillende protocollen. Door systemen op te delen wordt er meer complexiteit toegevoegd maar is het systeem minder kwetsbaar en wordt het opschalen een stuk eenvoudiger. In het **Figuur** hieronder is een natuurlijke progressie te zien van software integratie. Elke structuur is ontwikkeld vanuit de knelpunten en zwakke plekken van andere methodes.

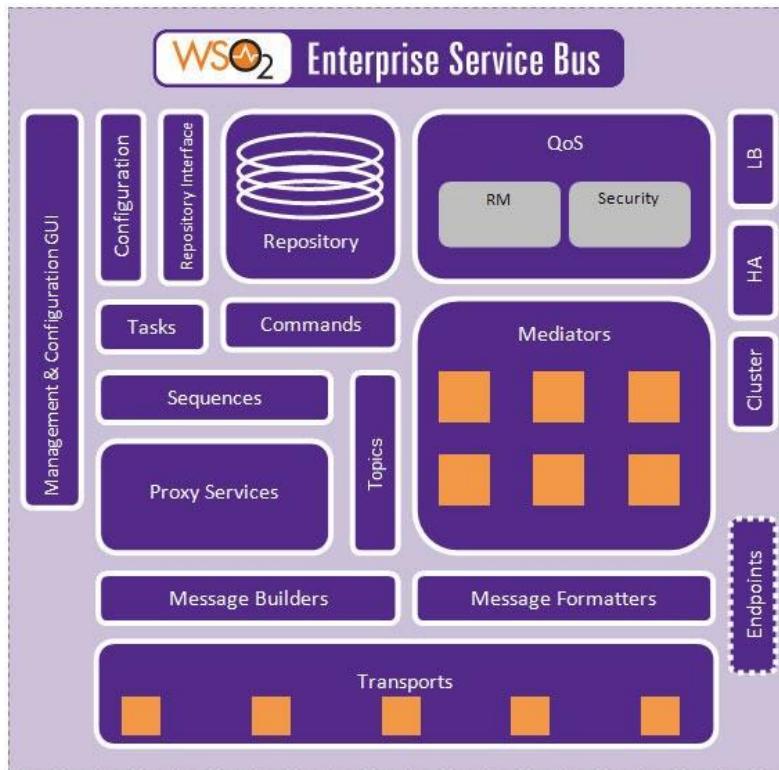


TOPOLOGIE

In het vorige hoofdstuk is vermeld dat het concept van de ESB bestaat uit het samenvoegen van veel verschillende losse systemen die gekoppeld worden met communicatie protocollen. De essentie van een ESB is dat het dus erg variabel kan zijn en verschillende oplossingen bieden andere diensten. **Figuur** laat zien waar een ESB zich bevindt in een systeem van integraties. Deze topologie komt uit een onderzoek genaamd 'Manufacturing service bus: an implementation' en bevat vijf lagen van een integratie platform, dit is gebaseerd op de SOA architectuur. Bij de *integration layer* is te zien dat een ESB wordt toegepast om de *data service layer* en de *integration service layer* te koppelen. In dit voorbeeld word de ESB dus gebruikt naast andere lagen



In **Figuur** hieronder is de topologie te zien van de ESB die wordt aangeboden door WS02, een *open source cloud provider* die onder andere een ESB integratie pakket aanbiedt. Binnen deze ESB zijn de volgende onderdelen aanwezig:



Transports

De ESB van WS02 kan gebruik maken van de meest gebruikte transport mogelijkheden. Er worden gebruik gemaakt van **Message builders** om de berichten op te bouwen en **Message formatters** om de berichten om te vormen zodat deze gebruikt kunnen worden binnen het systeem.

Proxy Services

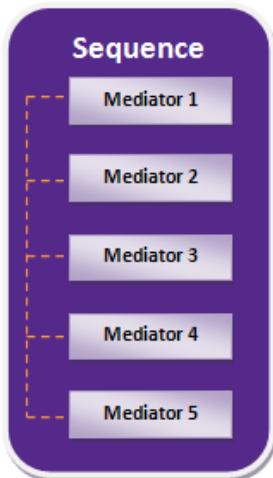
Proxy services zijn services tussen het internet en een applicatie. Binnen de ESB worden ze gebruikt om bericht te ontvangen en verwerken voor extra functionaliteit binnen de ESB.

Mediators

Mediators zijn één van de drijfveren van de ESB, ze werken door berichten te ontvangen, aan te passen en door te sturen. Zo kan een *mediator* bericht opdelen om naar verschillende onderdelen binnen de ESB te sturen. Binnen de ESB van WSO2 heeft de *mediator* volledig toegang tot elk onderdeel en het inkomend bericht op veel verschillende manieren worden getransformeerd.

Sequences

Ook wel een *mediation sequence* genoemd, dit is een verzameling van *mediators* die gebruikt kunnen worden om een *workflow* op te stellen waar een bericht langs alle *mediators* gaat. In **Figuur** wordt een schematisch overzicht getoond van een *sequence*.



QoS Components

Werkt samen met de *proxy services* om betrouwbaar en veilig berichten te kunnen sturen. Door de *flow* van inkomende berichten te beheren, wat zorgt voor een gestroomlijnde en consistente stroom van data.

Configuration, Repository/Registry

Deze drie onderdelen worden gebruik om de structuur en *metadata* van de ESB op te slaan. *Metadata* is de extra informatie over data, in het geval van de ESB wordt configuratiedata over de ESB zelf opgeslagen.

Management and Configuration GUI

Dit zijn de tools en de visuele representatie waarmee de ESB kan worden geconfigureerd en gemonitord. Via de *GUI (graphical user interface)* kan de gebruiker instellingen aanpassen en meerdere *busses* tegelijkertijd beheren.

WAAR KAN ESB GEBRUIKT WORDEN?

ESB structuren zijn vooral terug te vinden binnen grote bedrijven, dit komt mede door het hoge prijskaartje als een ESB of soortgelijk integratie gekocht moet worden. ESB en *middleware provider* Shadow-soft vermeldt in een artikel uit 2017 dat de prijs tussen de 20 en de 100 duizend euro per jaar ligt. Er zijn ook andere minder bekende opties die onder deze prijs liggen maar deze partijen kunnen niet dezelfde functionaliteiten bieden.

Een ESB pakket is groot, zwaar en bevat talloze componenten die veel tijd en kennis nodig hebben om onder de knie te krijgen. Als een bedrijf een ESB wil implementeren moeten er genoeg technische mensen in dienst zijn om ESB te kunnen configureren en onderhouden. Zoals eerder aangegeven is er geen vaste structuur aanwezig en heeft elke *provider* een eigen implementatie en extra services (meestal van het bedrijf zelf) die bij het pakket komen. Tenzij een bedrijf een eigen oplossing ontwikkeld is het afhankelijk van deze bedrijven.

Hoewel het een uitgebreid systeem is heeft het veel voordeelen. De SOA structuur is van nature goed schaalbaar en is het mogelijk om onderdelen te vervangen door het gebruik van een *distributed architecture*. Er wordt vaak gebruik gemaakt van *load balancing*. Dit is een techniek waarbij over en onder belaste processoren worden herkent en de hoeveelheid werk wordt aangepast voor optimale prestatie.

CLOUD OPLOSSINGEN

De structuren die besproken zijn binnen dit onderzoek kunnen binnen de *cloud* worden opgebouwd en gebruikt. Het zijn bestaande structuren die door de *providers* worden geïmplementeerd. De kracht van de *cloud* is dat het mogelijk is om uitgebreide en beveiligde systemen te gebruiken zonder daar zelf iets van hoeven te bouwen, daarbij is er ook geen eigen *datacenter* nodig om de volledige functionaliteiten te kunnen gebruiken. Vanuit dit idee is iPaaS ontstaan, dit staat voor *integration Platform as a Service* en bevat alle onderdelen die je maar kan bedenken om het integreren van systemen zo soepel mogelijk te maken.

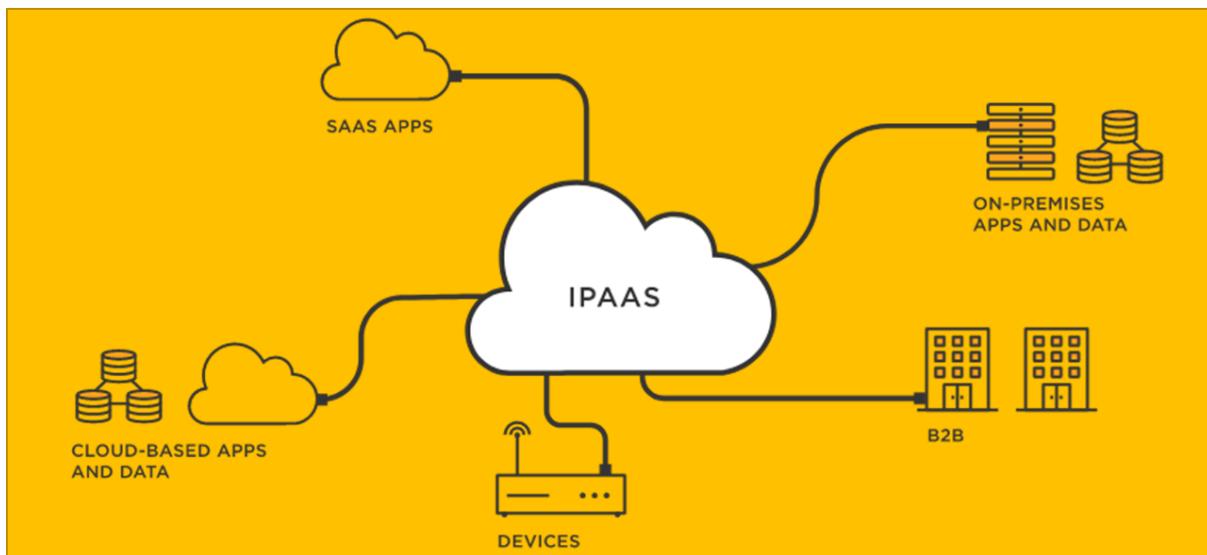
IPAAS VS ESB

Het verschil iPaaS en ESB ligt in de principes die worden gebruikt bij beide systemen. iPaaS is makkelijk te gebruiken en kan horizontaal worden opgeschaald, dit betekent dat er binnen de *cloud* meer computers kunnen worden gebruikt. Dit is bij een ESB een stuk lastiger.

ESB systemen zijn complex, grote systemen die veel geld kosten maar ook veel aankunnen. iPaaS is per *provider* anders en is over het algemeen een stuk goedkoper maar ook meer limitaties hebben door de *providers*.

IPAAS

Gartner, het grootste analistenbureau van de wereld definieert iPaaS als een set aan Cloud services die het development, de uitvoering en het beheren van integratiestromen mogelijk maken voor lokale (on-premise) en Cloud processen, services en applicaties voor één of meerdere bedrijven.



Het ontstaan van iPaaS is niet exact vastgelegd, sommige bronnen zeggen dat de voorgenoemde Gartner de term iPaaS in het wereld heeft gebracht, hoewel andere het Amerikaanse bedrijf Boomi zien als de eerste partij die een iPaaS heeft uitgebracht.

Sinds 2009 onderzoekt Gartner wat de sectorleiders zijn van meerdere IT sectoren en dus ook de Enterprise Integration Platform as a Service, Worldwide (oftewel EiPaaS). In **Figuur** is te zien hoe de verschillende partijen zijn verdeeld, dit is geen definitieve lijst van de 'beste' oplossingen maar eerder een verzameling van de grootste partijen en de rol die zij hebben in de markt.



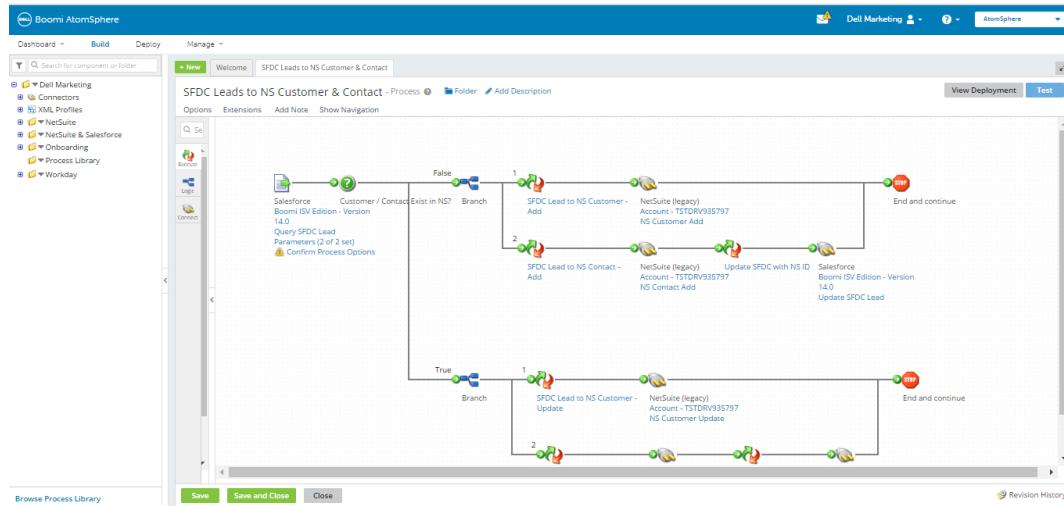
TOPOLOGIE

Elke iPaaS oplossing is anders, sinds elk zijn eigen functionaliteiten en expertise toevoegt. Het is gebruikelijk om de SOA structuur toe te passen. Sommige providers leveren een *low-code* omgeving waarin de gebruiker geen code hoeft te schrijven om aanpassingen te maken. Voor het algemeen zitten alle componenten ingebouwd en is er een subscriptiemodel waarin de gebruiker extra betaald voor meer services.

BOOMI

iPaaS service providers leveren een set aan tools die gebruikt kunnen worden om integraties op te zetten, beheren en data tussen belangrijke bronnen *up to date* houden. Zo levert één van de grootste leveranciers van iPaaS software [Boomi](#) (voorheen Dell Boomi) een pakket genaamd [AtomSphere](#). Dit pakket biedt functionaliteiten zoals:

- **Cloud native platform**
 - Door gebruik te maken van de kracht van de Cloud kan Boomi een platform aanbieden dat gebruik maakt van de flexibiliteit, schaalbaarheid en automatische updates.
- **low code, visual UI**
 - De AtomSphere omgeving is gebouwd met een *low-code* filosofie, hierdoor wordt er gebruik gemaakt van een *drag and drop* systeem waarbij de gebruiker onderdelen hoeft op te pakken en te slepen waarnaar de technische koppeling automatisch wordt gegenereerd.



- **Intelligentie**
 - Boomi gebruikt grote hoeveelheden geanonimiseerde data om gebruikers te helpen door verschillende suggesties te geven. Zo helpt [Boomi Suggest](#) met het *mappen* van data uit twee verschillende bronnen.

MOGELIJKHEDEN VOOR IPAAS

In principe kan elk bedrijf een iPaaS omgeving opzetten. Maar er is een grote afhankelijkheid bij de bedrijven die iPaaS oplossingen leveren. Omdat integraties per organisatie anders zijn is het onmogelijk om één oplossing te bedenken die alle mogelijke situaties aan kan. Binnen de meeste omgevingen zijn er een set aan systemen waar integraties mee kunnen worden gemaakt. Voor specifieke functies moet er maatwerk worden verricht wat de kosten verhoogd. Zodra een omgeving is ingericht is het moeilijk als bedrijf om uit deze omgeving te stappen, dit is een probleem bij vrijwel alle *cloud* oplossingen.

BIJLAGE 5: SOFTWARE REQUIREMENTS ANALYSE CRIS-X LOADER

OPGESTELD DOOR: RUBEN VAN GEMEREN

ORGANISATIE: HELMINK

DATUM: 25-02-2022

REVISIEGESCHIEDENIS

| Name | Date | Reason For Changes | Version |
|-------------------------|----------|--------------------|------------|
| Initialisatie | 24-02-22 | n.v.t | Versie 1.0 |
| Eerste volledige versie | 10-03-22 | extra overleg | Verzie 1.0 |

OVERZICHT EISEN

| Id | Prioriteit | Naam | Beschrijving |
|---------------------|------------|--|---|
| CRIS-X_L_REQ-1 | 10 | Ophalen Organisaties | De data van het label "Organisaties" ophalen naar de Loader. |
| CRIS-X_L_REQ-2 | 10 | Ophalen Locaties | De data van het label "Locaties" ophalen naar de Loader. |
| CRIS-X_L_REQ-3 | 8 | Conversie 1 op meer klant organisaties | De klanten met meerder organisaties moeten worden geconverteerd voor binnen IT-Glue. |
| CRIS-X_L_REQ-4 | 8 | Rauwe data in JSON | De data vanuit de bron moet worden opgeslagen als een. JSON bestand binnen de Loader. Dit bestand bevat alle data vanuit de bron. |
| CRIS-X_L_REQ-5 | 8 | Opslaan verwerkte data | Nadat de conversies zijn uitgevoerd moet de verwerkte data worden opgeslagen binnen de Loader. |
| CRIS-X_L_REQ-6 | 10 | POST request naar IT-Glue | Het gebruiken van een POST request om de verwerkte data naar IT-Glue te sturen. |
| CRIS-X_L_NONF_REQ-7 | 7 | Data binnen Loader verwijderen | Na het gebruiken van de data binnen de Loader, moet deze worden verwijderd. Dit gebeurt nadat de logging heeft plaatsgevonden. |
| CRIS-X_L_NONF_REQ-8 | 7 | Flexibiliteit data input | Bij het aanpassen van velden binnen CRIS-X moet het systeem zonder problemen doorwerken. Om deze aanpassingen te |

| | | |
|--|--|---|
| | | integreren met de datastream is er wel actie nodig. |
|--|--|---|

1. INTRODUCTIE

1.1 HET DOEL

De Loader bestaat is één van de onderdelen van de backend die uiteindelijk HABdesk automatische verwerkte data moet voeden. Binnen de Loader server zal er vanuit verschillende bronnen data worden opgehaald en opgeslagen binnen een de server. Deze data moet worden verwerkt, hiervoor wordt de loader ontwikkeld.

Binnen de loader server worden er verschillende koppelingen ontwikkeld. Elke koppeling heeft dezelfde gedachte erachter, maar bestaat uit andere onderdelen en alle koppelingen moet los van elkaar ontwikkeld worden.

De eerste koppeling die wordt gerealiseerd is de koppeling tussen CRIS-X en IT-Glue. CRIS-X (ook wel CRIS) is het CRM / ERP systeem dat gebruikt wordt binnen Helmink. IT-Glue is een framework dat wordt gebruikt om alle IT documentatie, wachtwoorden, en gebruikers van onze klanten te organiseren, structureren en op te slaan.

De loader wordt ontwikkeld om gegevens binnen IT-Glue automatisch te updaten door gegevens vanuit CRIS-X door te stromen naar IT-Glue.

1.2 DOCUMENTCONVENTIES

Bron: Hoofdstuk Requirements

Lettertype: Roboto

1.3 BEOOGD PUBLIEK EN LEESSUGGESTIES

Dit document is bedoeld voor Team Development binnen Helmink. Dit het team bestaat uit:

- T. Treffers
- W. Pols
- T. v Herwijnen
- J. Bellekom
- Ruben van Gemeren (stagiair)

Dit SRS-bestand is opgezet om de systeem eisen voor een specifieke loader te documenteren en uit te werken. Dit bestand is bedoeld als richtlijn voor het ontwikkelen en testen van deze opgestelde eisen. Het document bevat:

- De scope van het product
- De beschrijving van het product
- Systeem eisen (functioneel)

- Externe Interfaces
- Systeem eisen (non-functieel)

Het bestand kan worden gelezen van boven naar beneden. De focus ligt bij de functionele en non-functionele systeem eisen.

1.4 PROJECT SCOPE

Helmink wil automatisch data van bestaande bronnen kunnen ophalen en verwerken voor omgevingen zoals IT-Glue en HABdesk. Met de hulp van een loader server wordt data opgehaald en per systeem verwerkt.

Het doel van de loader server is om een HUB te zijn voor het verwerken databronnen binnen Helmink en zijn klanten.

CRIS-X → IT-GLUE LOADER

De koppeling tussen CRIS-X en IT-Glue heeft als doel om de gegevens binnen IT-Glue up to date te houden. Gegevens die binnen CRIS-X worden aangemaakt of bewerkt moeten automatisch worden verzonden naar IT-Glue. De datakoppling is 1 op 1, dit betekend dat data vanuit CRIS direct kan worden gebruikt in IT-Glue.

2. GLOBALE BESCHRIJVING

2.1 PRODUCTPERSPECTIEF

De CRIS-X → IT-Glue loader is een klein onderdeel van het HABdesk project. Dit is een project waarbij een centraal portaal wordt ontwikkeld waar verwerkte data vanuit verschillende omgevingen wordt verzameld en op een "slimme" manier wordt weer gegeven.

Het HABdesk project is groot en complex, hierdoor moet er worden ingezoomd en per onderdeel worden gekeken hoe het wordt ontwikkeld.

2.2 PRODUCTEIGENSCHAPPEN

De CRIS-X → IT-Glue loader bestaat uit het drie stappen

Stap 1:

CRIS-X probeert rauwe data naar de Loader database te pushen.

Stap 2:

De Loader verwerkt de rauwe data en slaat dit op.

Stap 3:

De data wordt vanuit van de Loader opgehaald en wordt zichtbaar binnen IT-Glue.

2.3 USER CLASSES EN KENMERKEN

De Loader vanuit Cris-X naar IT-Glue is een interne Loader dat gebruikt wordt door de medewerkers van Helmink. De data dat wordt gebruikt is aanwezig binnen de systemen met Helmink.

De technische kennis binnen Helmink is gevarieerd, hoewel niet alle medewerkers toegang hoeven krijgen tot de UI (User Interface) van de Loader. De Loader wordt beheerd door het Development team. Zij hebben veel kennis over de Loader en hoe deze werkt.

2.4 BEDRIJFSOMGEVING:

Helmink bestaat uit +- 15 medewerkers, hiervan hebben vier medewerkers de technische kennis om de Loader te kunnen beheren.

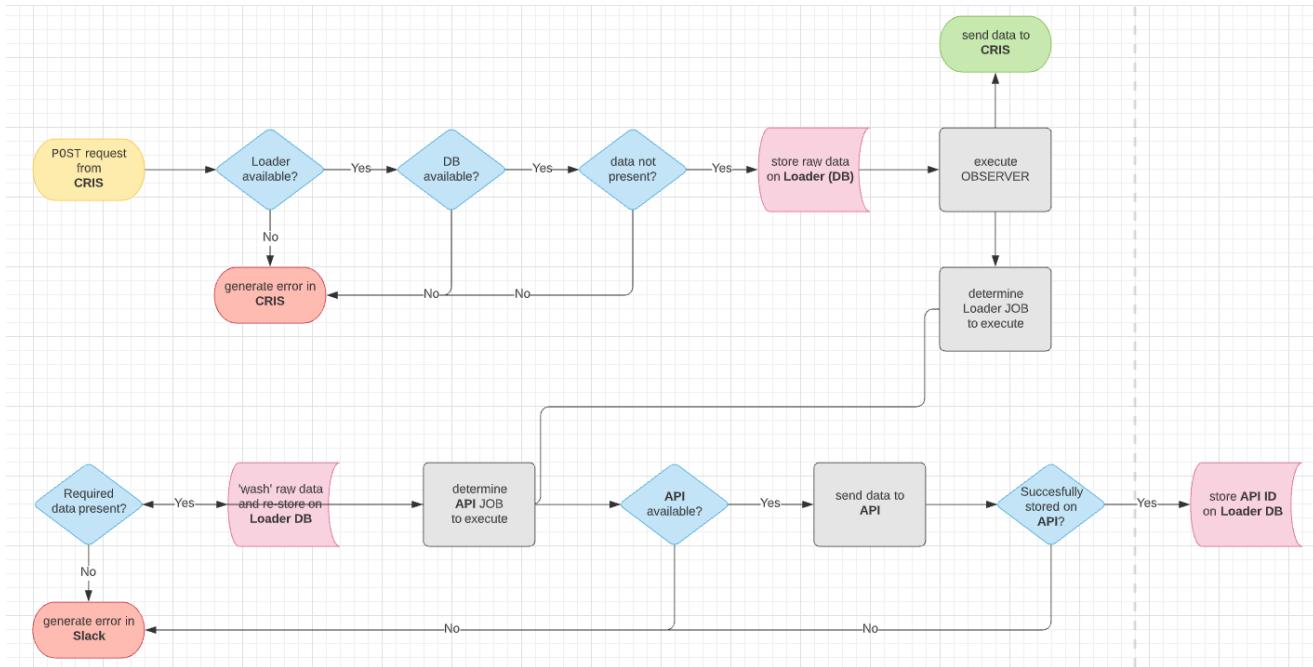
Afdelingen:

- Sales
- Marketing
- Operations
- Service & Support
- Administratie
- Development
- Facilitair

Binnen Helmink wordt er gebruikt gemaakt van verschillende systemen voor het beheren van klanten en assets.

2.5 ONTWERP- EN IMPLEMENTATIEBEPERKINGEN

De Loader moet ontwikkeld worden voor CRIS-X en IT-Glue. Deze systemen brengen limitaties met zich mee en kunnen voor problemen zorgen tijdens de ontwikkeling van de Loader.



2.7 AANNAMES EN AFHANKELIJKHEDEN

De Loader moet ontwikkeld worden voor CRIS-X en IT-Glue. Deze systemen brengen limitaties met zich mee en kunnen voor problemen zorgen tijdens de ontwikkeling van de Loader.

3. SYSTEEM FEATURES

De Loader wordt ontwikkeld om data van CRIS-X naar IT-Glue te synchroniseren. Het synchroniseren moet plaatsvinden zodra er nieuwe data wordt aangemaakt of bestaande data wordt bijgewerkt. De data worden opgehaald vanuit CRIS-X, hierna wordt de juiste data gekoppeld naar IT-Glue en wordt de gepaste actie uitgevoerd. Na het verwerken van de data wordt deze met een POST request naar IT-Glue gestuurd.

3.1 OPHALEN VAN DATA

3.1.1 Beschrijving en prioriteit

Vanuit CRIS-X moet de data van de labels "Organisaties" en "Locaties" worden opgehaald. Deze data is essentieel voor het gebruik van de CRIS-X Loader maar ook de andere Loaders die worden ontwikkeld.

De prioriteit van dit systeemkenmerk is een 10.

3.1.2 Stimulus/reactiesequenties

Gebruikersacties:

- De data moet worden ingevuld binnen CRIS-X.
- De data of aanpassingen moeten worden opgeslagen binnen CRIS-X.

Systeemacties:

- De data moet worden opgehaald binnen de Loader, dit gebeurt met een POST request vanuit CRIS-X.

3.1.3 Functionele Requirements

| Id | Prioriteit | Naam | Beschrijving |
|----------------|------------|----------------------|--|
| CRIS-X_L_REQ-1 | 10 | Ophalen Organisaties | De data van het label "Organisaties" ophalen naar de Loader. |
| CRIS-X_L_REQ-2 | 10 | Ophalen Locaties | De data van het label "Locaties" ophalen naar de Loader. |

3.2 DATA CONVERSIES**3.2.1 Beschrijving en prioriteit**

Niet alle data kan worden 1 op 1 worden geplaatst binnen IT-Glue. Binnen CRIS-X zijn er verschillende organisaties die onder één klant vallen. Binnen IT-Glue is dit niet praktisch, sinds je voor elke organisatie binnen CRIS een klantkaart krijgt binnen IT-Glue. Binnen IT-Glue zorgt dit voor een onhandige structuur.

Het systeem moet deze gevallen kunnen herkennen en ervoor zorgen dat er binnen IT-Glue maar één klantkaart wordt aangemaakt.

Dit kenmerk heeft een prioriteit van 8.

Systeemacties:

- De data moet vanuit CRIS worden verwerkt zodat deze de structuur van IT-Glue respecteert.

3.2.2 Functionele Requirements

| Id | Prioriteit | Naam | Beschrijving |
|----------------|------------|--|---|
| CRIS-X_L_REQ-3 | 8 | Conversie 1 op meer klant organisaties | De klanten met meerdere organisaties moeten worden geconverteerd voor binnen IT-Glue. |

3.3 DATA OPSLAG**3.3.1 Beschrijving en prioriteit**

Binnen de Loader moet de rauwe data worden opgeslagen. Deze data komt zonder aanpassingen vanuit de bron en is de basis voor de Loader. Naast het opslaan van de rauwe data moet ook de verwerkte data worden opgeslagen.

3.3.2 Functionele Requirements

| Id | Prioriteit | Naam | Beschrijving |
|----------------|------------|------------------------|---|
| CRIS-X_L_REQ-4 | 8 | Rauwe data in JSON | De data vanuit de bron moet worden opgeslagen als een .JSON bestand binnen de Loader. Dit bestand bevat alle data vanuit de bron. |
| CRIS-X_L_REQ-5 | 8 | Opslaan verwerkte data | Nadat de conversies zijn uitgevoerd moet de verwerkte data worden opgeslagen binnen de Loader. |

3.4 DATA POSTING

3.4.1 Beschrijving en prioriteit

Na het uitvoeren van de conversies kan de data worden gestuurd naar IT-Glue, dit gebeurt met een POST request.

3.4. Functionele Requirements

| Id | Prioriteit | Naam | Beschrijving |
|----------------|------------|---------------------------|---|
| CRIS-X_L_REQ-6 | 10 | POST request naar IT-Glue | Het gebruiken van een POST request om de verwerkte data naar IT-Glue te sturen. |

4. EXTERNE INTERFACE REQUIREMENTS

4.1 SOFTWARE INTERFACES

- [MySQL](#) database beheert met [PhpMyAdmin](#)
- Wordt gewerkt met [Laravel](#)

5. ANDERE NON-FUNCTIONELE REQUIREMENTS

5.1 SECURITY REQUIREMENTS

| Id | Prioriteit | Naam | Beschrijving |
|---------------------|------------|--------------------------------|--|
| CRIS-X_L_NONF_REQ-7 | 7 | Data binnen Loader verwijderen | Na het gebruiken van de data binnen de Loader, moet deze worden verwijderd. Dit gebeurt nadat de logging heeft plaatsgevonden. |

5.2 SOFTWARE QUALITY ATTRIBUTES

| Id | Prioriteit | Naam | Beschrijving |
|----|------------|------|--------------|
| | | | |

| | | | |
|---------------------|---|--------------------------|--|
| CRIS-X_L_NONF_REQ-8 | 7 | Flexibiliteit data input | Bij het aanpassen van velden binnen CRIS-X moet het systeem zonder problemen doorwerken. Om deze aanpassingen te integreren met de datastream is er wel actie nodig. |
|---------------------|---|--------------------------|--|

BIJLAGE 6: SOFTWARE REQUIREMENTS ANALYSE AUVIK API LOADER

VOOR: AUVIK API LOADER

OPGESTELD DOOR: RUBEN VAN GEMEREN

ORGANISATIE: HELMINK

DATUM: 21-04-2022

REVISIE GESCHIEDENIS

| Name | Date | Reason For Changes | Version |
|-----------------|------------|--------------------|---------|
| Eerste versie | 21-04-2022 | - | 1.0 |
| Complete versie | 30-04-2022 | Aanpassingen scope | 2.0 |

OVERZICHT EISEN

| Id | Prioriteit | Naam | Beschrijving |
|-------------------|------------|-------------------------------------|---|
| AUVIK_API_L_REQ-1 | 10 | Ophalen <i>tenants</i> | Alle netwerkdata van de Auvik API ophalen. De <i>request /tenants</i> wordt gebruikt om alle data van alle netwerken op te halen. |
| AUVIK_API_L_REQ-2 | 10 | Opslaan Hoofd <i>tenant</i> | Het hoofdnetwerk (<i>main tenant</i> genoemd) opslaan in de database met de volgende velden: id, type, domain_prefix, created_at, updated_at. |
| AUVIK_API_L_REQ-3 | 10 | Opslaan <i>multiClients</i> | Per <i>clients</i> worden alle <i>multiClients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. |
| AUVIK_API_L_REQ-4 | 10 | Opslaan <i>clients</i> | Per <i>multiClients</i> worden alle <i>clients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. |
| AUVIK_API_L_REQ-5 | 10 | Controle Auvik data / klantgegevens | Data vanuit de database wordt vergeleken met op basis van de velden domain_prefix bij de Auvik data en short_name bij de klantgegevens. |
| AUVIK_API_L_REQ-6 | 10 | Auvik_id invullen | Bij een correcte match moet het veld auvik_id bij de |

| | | | |
|--------------------|----|-----------------------------|--|
| | | | klantgegevens worden ingevuld. |
| AUVIK_API_L_REQ-7 | 10 | Locatie controle | Bij een incorrecte match moet de locatie binnen de twee databronnen worden gecontroleerd, dit gebeurt door extra details op te halen vanuit de Auvik API. |
| AUVIK_API_L_REQ-8 | 10 | Weergeven netwerken | De netwerken weergeven binnen de aparte omgeving binnen de Loader server. |
| AUVIK_API_L_REQ-9 | 10 | Weergeven koppeling | De koppeling tussen de klant en het Auvik netwerk weergeven binnen de omgeving van de klantlocatie |
| AUVIK_API_L_REQ-10 | 10 | Details weergeven | Extra informatie geven over de klant locatie als hier op wordt geklikt, met daarbij de apparaten die binnen het Auvik netwerk worden beheert. |
| AUVIK_API_L_REQ-11 | 10 | Handmatig netwerk toevoegen | De gebruiker kan handmatig een Auvik netwerk toevoegen bij een klant locatie als dit toepasselijk is. |
| AUVIK_API_L_REQ-12 | 6 | Data inladen | Binnen alle instanties van het inladen van data mag de tijdsduur niet langer zijn dan 30 seconden voor extreme gevallen. reguliere verzoeken moeten worden afgehandeld binnen 10 seconden. |

1. INTRODUCTIE

1.1 HET DOEL

Het product beschrijft één van de koppelingen binnen de Loader server van Helmink. Het product gaat om een koppeling tussen het Auvik asset monitoring service en de Loader server van Helmink. Het product maakt gebruik van de officiële API van Auvik om netwerk data van Helmink op te halen en te verwerken. Het product zal bestaande data binnen de Loader en data van de Auvik API koppelen voor toekomstige werkzaamheden.

1.2 DOCUMENTCONVENTIES

Lettertype: Roboto

1.3 BEOOGD PUBLIEK EN LEESSUGGESTIES

Dit document is bedoelt voor Team Development binnen Helmink. Dit het team bestaat uit:

- T. Treffers (*Backend developer*)
- W. Pols (*Backend developer*)
- T. v Herwijnen (*Frontend developer*)
- J. Bellekom (Directeur)
- Ruben van Gemeren (stagiair)

Dit SRS bestand is opgezet om de software eisen voor een specifieke Loader en flow te documenteren en uit te werken. Dit bestand is bedoeld als richtlijn voor het ontwikkelen en testen van deze opgestelde eisen. Het document bevat:

- De scope van het product
- De beschrijving van het product
- Systeem eisen (functioneel)
- Externe Interfaces
- Systeem eisen (non-functioneel)

Het bestand kan worden gelezen van boven naar beneden. De focus ligt bij de functionele software eisen en de software karakteristieken.

1.4 PROJECT SCOPE

De software die wordt ontwikkeld is een Loader omgeving voor de Auvik API. Deze omgeving bevat:

- Het ophalen en opslaan van data.
- Het matchen van Auvik data met bestaande data binnen de Loader Server.
- Het opslaan van de verrijkte data.

Er wordt alleen gewerkt met de API's die beschikbaar zijn gesteld, daarbij hoeft er geen server worden opgezet. Er wordt gebruik gemaakt van de Loader server van Helmink. Er hoeft geen aparte matching server te worden ontwikkeld. Er kan gebruik worden gemaakt van de matching server die als is opgezet door T. Treffers.

2. GLOBALE BESCHRIJVING

2.1 PRODUCTPERSPECTIEF

De Auvik API Loader is één van de Loaders die ontwikkeld wordt voor de verrijking van informatie binnen de Loader server. De Loader maken deel uit van een netwerk dat datastromen beheert tussen de drie systemen die samen de CRM omgeving van Helmink vormt. Deze systemen zijn het CRM systeem CRIS-X, het CMDB systeem IT-Glue en het service portaal HABdesk.

De Auvik API Loader maakt gebruik van de Auvik API en haalt data op over de netwerken van die Helmink beheren. Deze data heeft een directe relatie met de klantgegevens binnen CRIS-X. Deze relatie is op domeinniveau en bestaat niet

binnen het systeem. In **Figuur 1** is het model te zien van de Auvik API Loader en hoe deze past binnen de rest van de Loader server.

2.2 PRODUCT EIGENSCHAPPEN

De software die wordt ontwikkeld is een Loader omgeving voor de Auvik API. Deze omgeving bevat:

- Het ophalen en opslaan van data.
- Het matchen van Auvik data met bestaande data binnen de Loader Server.
- Het opslaan van de verrijkte data.

2.3 USER CLASSES EN KENMERKEN

De Loader vanuit de Auvik API naar IT-Glue is een interne Loader dat gebruikt wordt door de medewerkers van Helmink. De data dat wordt gebruikt is aanwezig binnen de systemen met Helmink.

De technische kennis binnen Helmink is gevarieerd, hoewel niet alle medewerkers toegang hoeven krijgen tot de UI (User Interface) van de Loader. De Loader wordt beheert door het Development team. Zij hebben veel kennis over de Loader en hoe deze werkt.

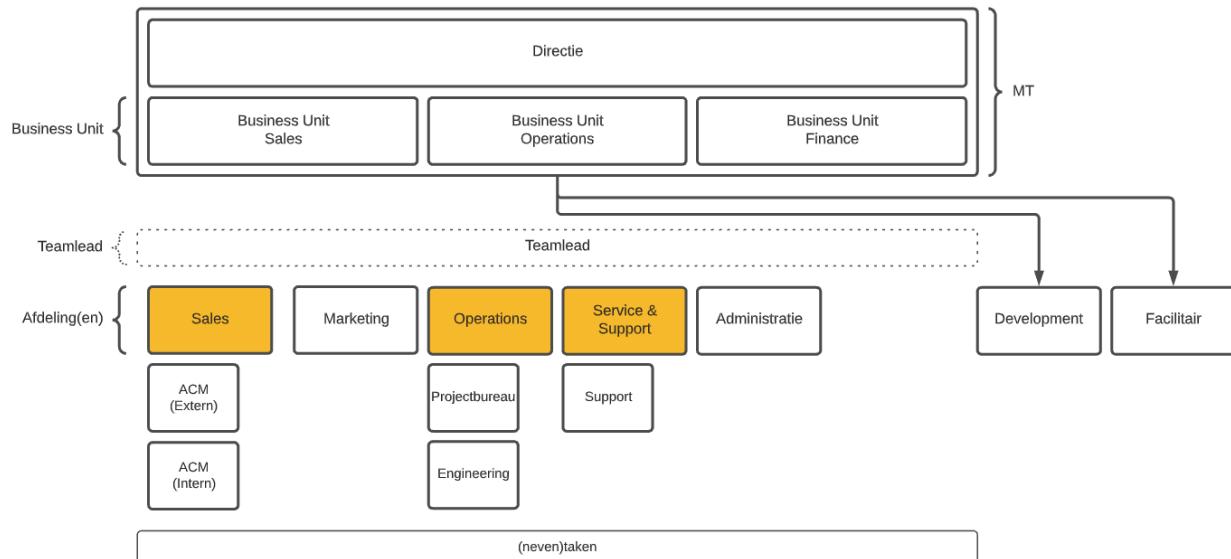
2.4 BEDRIJFSOMGEVING:

Helmink bestaat uit +- 15 medewerkers, hiervan hebben vier medewerkers de technische kennis om de Loader te kunnen beheren.

Afdelingen:

- Sales
- Marketing
- Operations
- Service & Support
- Administratie
- Development
- Facilitair

Figuur 2 laat de organogram zien van Helmink.



2.5 Ontwerp- en implementatiebeperkingen

Omdat er gebruik wordt gemaakt van verschillende API's is de data gelimiteerd tot het aanbod van deze API's. Het kan ook zijn dat de ondersteuning van deze interfaces wordt beperkt, dit kan als gevolg hebben dat het systeem niet meer werkbaar is.

Het product wordt ontwikkeld binnen de Loader server, de limitaties binnen de server zullen de scope van het product beperken.

2.7 AANNAMES EN AFHANKELIJKHEDEN

Omdat er gebruik wordt gemaakt van verschillende API's is de data gelimiteerd tot het aanbod van deze API's. Het kan ook zijn dat de ondersteuning van deze interfaces wordt beperkt, dit kan als gevolg hebben dat het systeem niet meer werkbaar is.

Het product wordt ontwikkeld binnen de Loader server, de limitaties binnen de server zullen de scope van het product beperken.

3. SYSTEEM FEATURES

De Auvik API Loader bestaat uit drie functies met ieder minimaal één software eis bevat. De drie functies zijn:

1. Ophalen en opslaan van de Auvik API data
2. Het koppelen van de Auvik API data met de klantgegevens binnen de Loader server
3. Het visualiseren van de data binnen de *user interface* van de Loader server.

Het ophalen van de data gebeurt handmatig en wordt verricht door het development team. Het koppelen van data gebeurt semiautomatisch en is gebaseerd op het

niveau van compleetheid. De mogelijk voor handmatig koppelen moet beschikbaar zijn. Er wordt gebruik gemaakt van een prioriteiten schaal van 1-10, waarbij 1 aangeeft dat de eis vrijwel geen vereiste is om het product functioneel werkend te maken. Een 10 is absoluut noodzakelijk om het systeem functioneel werkend te maken.

3.1 OPSLAAN VAN DATA

3.1.1 Beschrijving en prioriteit

Vanuit de Auvik API worden netwerk gegevens opgehaald naar de Loader Server. Deze data bestaat uit het type netwerk en de naam (wordt gerepresenteerd door "domain prefix"). De structuur van de data bestaat uit *clients* en *multiClients*. De *clients* hebben altijd een *multiClient* boven zich, daarentegen kunnen *multiClients* nog een *multiClients* boven zich hebben, dit is niet altijd het geval.

De prioriteit van dit systeemkenmerk is een 10. Dit betekent dat de functie noodzakelijk is voor de werking van het product.

3.1.2 Stimulus/reactie sequenties

Gebruikersacties:

- Het systeem moet gestart worden door één van de developers van Helmink.

Systeemacties:

- De netwerkdata wordt opgehaald vanuit de Auvik API.
- Het hoofdnetwerk van Helmink wordt opgeslagen in de database.
- De juiste *multiClients* worden gevonden en opgeslagen voordat de *clients* opgeslagen worden.

3.1.3 Functioneel Requirements

| Id | Prioriteit | Naam | Beschrijving |
|-------------------|------------|-----------------------------|---|
| AUVIK_API_L_REQ-1 | 10 | Ophalen <i>tenants</i> | Alle netwerkdata van de Auvik API ophalen. De <i>request /tenants</i> wordt gebruikt om alle data van alle netwerken op te halen. |
| AUVIK_API_L_REQ-2 | 10 | Opslaan Hoofd <i>tenant</i> | Het hoofdnetwerk (<i>main tenant</i> genoemd) opslaan in de database met de volgende velden: id, type, domain_prefix, created_at, updated_at. |
| AUVIK_API_L_REQ-3 | 10 | Opslaan <i>multiClients</i> | Per <i>clients</i> worden alle <i>multiClients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. |

| | | | |
|-------------------|----|-----------------|---|
| AUVIK_API_L_REQ-4 | 10 | Opslaan clients | Per <i>multiClients</i> worden alle <i>clients</i> opgeslagen met de volgende velden: id, parent_id, type, domain_prefix, created_at, updated_at. |
|-------------------|----|-----------------|---|

3.2 KOPPELING VAN DATA

3.2.1 Beschrijving en prioriteit

De Auvik data binnen de Loader server moet worden gekoppeld aan de klantgegeven binnen de Loader server zelf, deze gegevens zijn oorspronkelijk afkomstig vanuit CRIS-X. De koppeling gebeurt op twee niveaus. Het eerste niveau is van de naam van de klant binnen de Auvik data, dit is lijnend en als er geen gelijkenis is kan er geen automatische koppeling worden gemaakt.

Als er geen koppeling gemaakt kan worden kan de locatie gebruikt worden. De locatie wordt gecontroleerd op drie elementen, land, postcode en straat. Als één van deze elementen niet overeenkomt kan er geen automatische koppeling worden gevormd.

3.2.2 Stimulus/reactie sequenties

Gebruikersacties:

- Indien verantwoord handmatig een klant koppelen aan een Auvik netwerk (*tenant*).

Systeemacties:

- Data vanuit de database wordt vergeleken met op basis van de velden domain_prefix bij de Auvik data en short_name bij de klantgegevens.
- Bij een correcte match moet het veld auvik_id bij de klantgegevens worden ingevuld.
- Bij een incorrecte match moet de locatie binnen de twee databronnen worden gecontroleerd, dit gebeurt door extra details op te halen vanuit de Auvik API.

3.2.3 Functioneel Requirements

| Id | Prioriteit | Naam | Beschrijving |
|-------------------|------------|-------------------------------------|---|
| AUVIK_API_L_REQ-5 | 10 | Controle Auvik data / klantgegevens | Data vanuit de database wordt vergeleken met op basis van de velden domain_prefix bij de Auvik data en short_name bij de klantgegevens. |
| AUVIK_API_L_REQ-6 | 10 | Auvik_id invullen | Bij een correcte match moet het veld auvik_id bij de klantgegevens worden ingevuld. |

| | | | |
|-------------------|----|------------------|---|
| AUVIK_API_L_REQ-7 | 10 | Locatie controle | Bij een incorrecte match moet de locatie binnen de twee databronnen worden gecontroleerd, dit gebeurt door extra details op te halen vanuit de Auvik API. |
|-------------------|----|------------------|---|

3.3 VISUALISATIE

3.3.1 Beschrijving en prioriteit

De data binnen de database moet worden gevisualiseerd in de *user interface* van de Loader server. De visualisatie bestaat uit een aantal onderdelen:

1. De netwerken vanuit de Auvik API moet worden weer gegeven binnen de Loader sever door middel van een tabel.
2. De koppeling die zich tussen Auvik en de klantgegeven bevindt moet worden weer gegeven binnen de locatie omgeving van de klanten.
3. Als er op een klant met een Auvik koppeling wordt geklikt moeten er details van deze klant worden opgehaald, daarbij horen de verschillende apparaten die zich bevinden in het netwerk van specifieke klant locatie.
4. Bij klant locaties zonder een Auvik koppeling moet het mogelijk zijn om handmatig een netwerk te selecteren uit een selectie van de database. Deze keuze moet automatisch worden opgeslagen binnen de database.

3.3.2 Stimulus/reactie sequenties

Gebruikersacties:

- Indien toepasselijk het handmatig kiezen van een Auvik netwerk uit een selectie van de database

Systeemacties:

- De netwerken weergeven binnen de aparte omgeving binnen de Loader server.
- De koppeling tussen de klant en het Auvik netwerk weergeven binnen de omgeving van de klantlocatie
- Extra informatie geven over de klant locatie als hier op wordt geklikt, met daarbij de apparaten die binnen het Auvik netwerk worden beheert.
- De gebruiker kan handmatig een Auvik netwerk toevoegen bij een klant locatie als dit toepasselijk is.

3.3.3 Functioneel Requirements

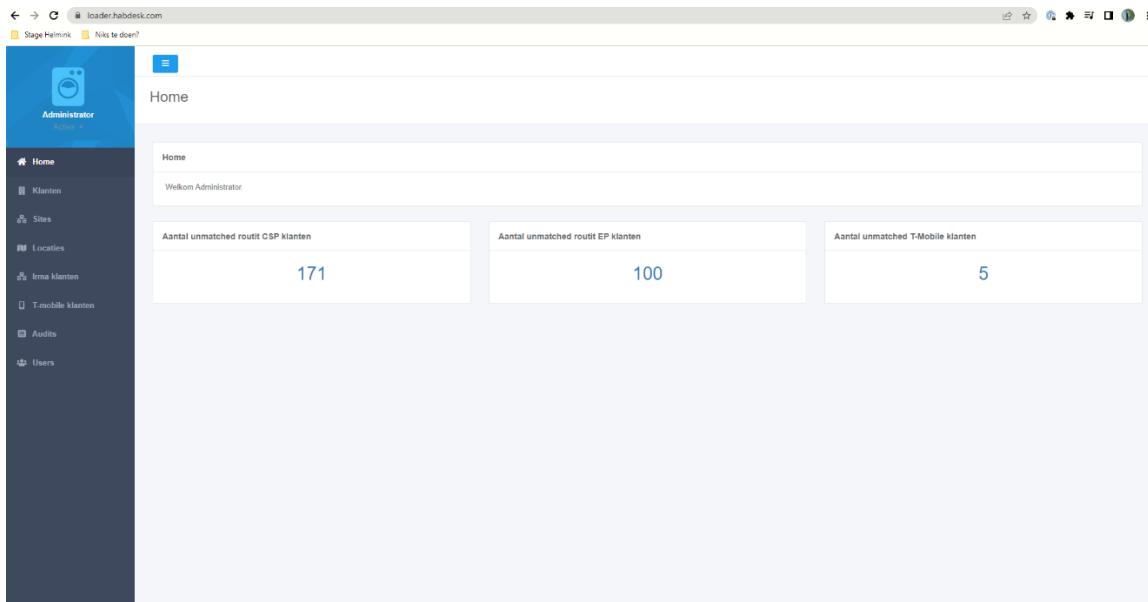
| Id | Prioriteit | Naam | Beschrijving |
|-------------------|------------|---------------------|---|
| AUVIK_API_L_REQ-8 | 10 | Weergeven netwerken | De netwerken weergeven binnen de aparte omgeving binnen de Loader server. |

| | | | |
|--------------------|----|-----------------------------|---|
| AUVIK_API_L_REQ-9 | 10 | Weergeven koppeling | De koppeling tussen de klant en het Auvik netwerk weergeven binnen de omgeving van de klantlocatie |
| AUVIK_API_L_REQ-10 | 10 | Details weergeven | Extra informatie geven over de klant locatie als hier op wordt geklikt, met daarbij de apparaten die binnen het Auvik netwerk worden beheert. |
| AUVIK_API_L_REQ-11 | 10 | Handmatig netwerk toevoegen | De gebruiker kan handmatig een Auvik netwerk toevoegen bij een klant locatie als dit toepasselijk is. |

4. EXTERNE INTERFACE REQUIREMENTS

4.1 USER INTERFACES

Het visuele element bestaat uit een deel van de web omgeving van de Loader sever. Deze omgeving wordt weergegeven binnen **figuur 3**.



De visuele elementen moeten binnen de omgeving van **figuur 3** passen en een soortgelijke stijl hanteren.

4.2 SOFTWARE INTERFACES

Auvik API: De officiële van Auvik die toegang biedt tot alle netwerkdata beheert door Auvik

Database: De database die wordt gebruikt, een MySQL database met een *user interface* in de vorm van een PHPMyAdmin omgeving.

4.3 COMMUNICATIONS INTERFACES

De communicatie binnen dit product gebeurt aan de hand van API verzoeken, deze verzoeken maken gebruik van het *File Transfer Protocol (FTP)*.

5. SOFTWARE CHARACTERISTICS

5.1 PERFORMANCE REQUIREMENTS

| Id | Prioriteit | Naam | Beschrijving |
|--------------------|------------|--------------|--|
| AUVIK_API_L_REQ-12 | 6 | Data inladen | Binnen alle instanties van het inladen van data mag de tijdsduur niet langer zijn dan 30 seconden voor extreme gevallen. reguliere verzoeken moeten worden afgehandeld binnen 10 seconden. |

BIJLAGE 7: AUVIK API LOADER TESTPLAN

VOOR: HABDESK

OPGESTELD DOOR: RUBEN VAN GEMEREN

ORGANISATIE: HELMINK

DATUM: 30-5-2022

REVISIEGESCHIEDENIS

| Naam | Datum | Reden voor verandering | Versie |
|--------------|-----------|------------------------|--------|
| Eerste opzet | 30-5-2022 | n.v.t | 1.0 |

1.1 INTRODUCTIE

De Auvik Loader is een onderdeel van het HABdesk project, specifieker is het één van Loaders binnen de Helmink Wasmachine (Loader server). Deze Loader moet data vanuit de Auvik API ophalen en koppelen met het CRIS-X id dat binnen de Loader server aanwezig is.

2.0 DOELSTELLINGEN EN TAKEN

2.1 DOELSTELLINGEN

Het doel van deze test is om te garandeert dat data vanuit Auvik automatisch kan worden gekoppeld aan CRIS-X, mocht er geen koppeling kunnen worden gemaakt moet deze data beschikbaar zijn voor de gebruiker.

2.2 TAKEN

1. Testen van de *flow* van Auvik naar de Loader server.
2. Controleren of koppeling tussen Auvik en CRIS-X werkt
3. Code Clean up
4. Test review
5. Verbetering test

3.0 SCOPE

Algemeen

Het onderdeel dat wordt getest bestaat uit twee delen

1. Of alle nodige data wordt opgehaald vanuit Auvik
2. Of de data wordt gekoppeld aan de data vanuit CRIS-X (al aanwezig in de Loader server).

De onderdelen die niet worden getest zijn

1. Of alle data vanuit Auvik overeenkomt met de data vanuit CRIS-X.
2. Of alle velden vanuit Auvik correct zijn.

Tactieken

Om te controleren of alle data wordt opgehaald wordt er gebruik gemaakt van een API test, waarbij een gecontroleerde input wordt gebruikt waar de output van bekend is. Met deze methode wordt de API op een kleine schaal gecontroleerd.

Om te controleren of de data correct koppelt aan de data vanuit CRIS-X wordt er manueel gekeken of de data overeenkomt.

4.0 TESTSTRATEGIE

4.1 DATABASE TESTEN

Definitie

Database-Testing is een testmethode waarbij de schemas, tabellen en data binnen een database worden getest, door deze methode toe te passen is het mogelijk om de toegang en betrouwbaarheid van data te kunnen garanderen. Het testen gebeurt tijdens het development proces en wordt meestal uitgevoerd door één van de developers. database-testing wordt toegepast omdat de database die gebruikt wordt alle belangrijke informatie bevat. Deze informatie wordt in meerdere systemen gebruikt, als deze data niet goed wordt opgeslagen is de integriteit van het gehele HABdesk project aangetast.

Deelnemers

Uitvoering: Ruben van Gemeren

Controle: W. Pols

Methodologie

Dit zijn de stappen die worden uitgevoerd tijdens het testen:

1. Het opzetten en uitwerken van het testplan en testcases.
2. Het creëren van een testomgeving binnen de code.
3. Het uitvoeren van de tests.
4. Het noteren van de resultaten binnen de testcase
5. Bespreken van de testresultaten.
6. Wanneer nodig test herhalen.

4.2 API UNIT TESTEN

Definitie

API unit tests zijn tests gespecialiseerd in het meten van de functionaliteit, prestatie en betrouwbaarheid van een gekozen API. Een API is een Applicatie Programming Interface en wordt gebruikt als communicatiemiddel tussen twee of meerdere applicaties, API's moeten worden onderhouden om data zuiver te houden en conflicten te voorkomen. Veel bedrijven die een API vrijgeven hebben moeite met het bijhouden en updaten van hun API's, om deze reden is het testen van API's essentieel. *Unit testing* is een manier om tests automatisch uit te laten voeren, er zijn meerdere frameworks en externe tools om tests op te zetten en uit te voeren.

Deelnemers

Uitvoering: Ruben van Gemeren

Controle: W. Pols

Methodologie

De stappen die worden gevuld zijn:

1. Het opzetten en uitwerken van het testplan en testcases.
2. Het creëren van een testomgeving binnen de code.
3. Het uitvoeren van de tests.
4. Het noteren van de resultaten binnen de testcase
5. Bespreken van de testresultaten.
6. Wanneer nodig test herhalen.

5.0 HARDWARE EISEN

Om de test te kunnen uitvoeren is er een pc nodig met een besturingssysteem dat beschikbaar is voor het Laravel Framework dat wordt gebruikt.

6.0 ENVIRONMENT EISEN

6.1 MAIN FRAME

Noodzakelijke eisen:

1. Een stabiele internet verbinding
2. Een pc met de juiste software eisen

6.2 WORKSTATION

De tests worden uitgevoerd op locatie binnen Helmink. De pc is verbonden op het netwerk binnen Helmink en bevat de volgende specificaties:

Besturingssysteem: Windows 10

Processor: Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz 2.90 GHz

RAM-Geheugen: 8,00 GB

7.0 BEDIENINGSPROCEDURES

PROBLEEMRAPPORTAGE

Als er een probleem wordt geconstateerd binnen de testomgeving wordt deze genoteerd binnen de testcase, deze testcase wordt besproken met de medewerker die is aangewezen voor controle. De problemen die worden genoteerd moeten terug te vinden zijn om verdere ontwikkelen wellicht aan te passen.

VERANDERINGEN IN DE CODE

Als er veranderingen in de code wordt aangebracht worden deze met de hulp van het *version control* platform Github opgeslagen. Binnen het project wordt er een aparte *branche* aangemaakt waar de testcode kan worden opgeslagen. Als verbeteringen zijn doorgevoerd worden deze eerst gecontroleerd voordat de code kan worden *pushed* naar de Github *main branche*.

8.0 FUNCTIES DIE GETEST MOETEN WORDEN

Functies die worden getest:

1. Werken alle API *calls* vanuit Auvik zoals vermeld.
2. Is het mogelijk om data tussen Auvik en CRIS-X te koppelen.

9.0 FUNCTIES DIE NIET GETEST MOETEN WORDEN

1. Of alle data vanuit Auvik overeenkomt met de data vanuit CRIS-X.
2. Of de waardes vanuit Auvik correct zijn.

10.0 MIDDELEN/ROLLEN & VERANTWOORDELIJKHEDEN

Beheren: Ruben van Gemeren

Ontwerpen: Ruben van Gemeren

Voorbereiden: Ruben van Gemeren

Uitvoeren: Ruben van Gemeren

Controle: W. Pols

Opplossen: Ruben van Gemeren

11.0 SCHEMA'S

BELANGRIJKE RESULTATEN

De onderdelen die moeten worden opgeleverd zijn:

- Het testplan
- De testcases die gebruikt zijn
- De verwerkte resultaten

12.0 AFHANKELIJKHEDEN

Het testen moet voor 17 juni 2022 zijn afgerekend i.v.m bepaalde deadlines. Het testen kan alleen worden uitgevoerd binnen Helmink en uitgevoerd op de lokale server van Helmink.

13.0 RISICO'S / AANNAMES

Bij vertragen van het testen moet de schaal van de uitgevoerde test worden aangepast op basis van de tijd die over is.

14.0 TOOLS

Er wordt gebruik gemaakt van de ingebouwde tools die het *framework* Laravel aanbiedt.

BIJLAGE 8: AUVIK API LOADER API UNIT TESTCASE

VOORBEREIDING:

| | | | |
|----------------------|-------------------------------------|---------------------------------------|---|
| Test Case ID: | AUVIK_API_LOADER_API_TEST | Beschrijving: | Geautomatiseerde Auvik API Test. |
| Gemaakt door: | Ruben van Gemeren | Reviewer: | W. Pols |
| | | Versie: | 1.0 |
| Tester: | Ruben van Gemeren | Datum: | 30-5-2022 |
| | | Resultaat: | PASS |
| | | Test Scenario: | Automatisch testen van verschillende API verzoeken. |
| # | Vereisten: | Testdata: | |
| 1 | Auvik API moet bereikbaar zijn. | Auvik API Authenticatie | |
| 2 | De Testcase code moet werkend zijn. | Testcase code (<i>AuvikAPITest</i>) | |

TESTDATA:

| Stap # | Beschrijving: | Verwacht resultaat: | Resultaat: | PASS / FAIL / NOT EXECUTED / SUSPENDED |
|--------|--|---------------------|---------------|--|
| 1. | 'can access tenants' test functie | Test PASS | Test voltooit | PASS |
| 2. | 'can access tenants details' test functie | Test PASS | Test voltooit | PASS |
| 3. | 'can access tenants devices' test functie | Test PASS | Test voltooit | PASS |
| 4. | 'can access tenants device details' test functie | Test PASS | Test voltooit | PASS |

'CAN ACCESS TENANTS' TEST FUNCTIE

CODE

```
12  /**
13  * Collects all Auvik Tenants on endpoint.
14  *
15  * @return void
16  */
17  public function test_can_access_tenants_endpoint()
18  {
19
20
21  $this->client = new \GuzzleHttp\Client([
22      'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
23      'base_uri' => config('services.auvik.base_url'),
24      'headers' => [
25          'Accept' => 'application/json',
26          'Content-Type' => 'application/json',
27          'Encode' => 'UTF-8',
28      ],
29  ]);
30
31
32  try {
33      // Get a response.
34  $response = $this->client->get(
35      'tenants');
36  } catch (\GuzzleHttp\Exception\RequestException $exception) {
37      $response = $exception->getResponse();
38  }
39
40  // Transform the response into a JSON object.
41  $response = json_decode($response->getBody()->getContents());
42
43  // Dump the errors if present.
44  if (isset($response->errors)) {
45      self::assertTrue(false);
46  }
47
48  self::assertFalse(empty($response->data));
49
50  if ($response != null) {
51      self::assertTrue(true);
52  } else {
53      self::assertTrue(false);
54  }
55
56 }
```

'CAN ACCESS TENANTS DETAILS' TEST FUNCTIE

CODE

```
58  /**
59  * Collects all the details for the tenants contained in main tenant on endpoint.
60  *
61  * @return void
62  */
63  public function test_can_access_tenants_details_endpoint()
64  {
65
66
67  $this->client = new \GuzzleHttp\Client([
68      'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
69      'base_uri' => config('services.auvik.base_url'),
70      'headers' => [
71          'Accept' => 'application/json',
72          'Content-Type' => 'application/json',
73          'Encode' => 'UTF-8',
74      ],
75  ]);
76
77  try {
78      // Get a response.
79      $response = $this->client->get(
80          'tenants');
81  } catch (\GuzzleHttp\Exception\RequestException $exception) {
82      $response = $exception->getResponse();
83  }
84
85  // Transform the response into a JSON object.
86  $response = json_decode($response->getBody()->getContents());
87
88  // Dump the errors if present.
89  if (isset($response->errors)) {
90      self::assertTrue(false);
91  }
92
93  self::assertFalse(empty($response->data));
94
95
96  $tenant = AuvikTenant::first();
97  $details = $this->client->get("tenants/detail/", [
98      'query' => [
99          'tenantDomainPrefix' =>$tenant->domain_prefix,
100         ]
101     ]);
102
103
104
105  if ($details != null) {
106      self::assertTrue(true);
107  } else {
108      self::assertTrue(false);
109  }
110
111
112 }
```

'CAN ACCESS TENANTS DEVICES' TEST FUNCTIE

CODE

```

114 /**
115 * Collects all device on tenant on endpoint.
116 *
117 * @return void
118 */
119 public function test_can_access_tenants_devices_endpoint()
120 {
121
122     $this->client = new \GuzzleHttp\Client([
123         'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
124         'base_uri' => config('services.auvik.base_url'),
125         'headers' => [
126             'Accept' => 'application/json',
127             'Content-Type' => 'application/json',
128             'Encode' => 'UTF-8',
129         ],
130     ]);
131 }
132
133 try {
134     // Get a response.
135     $response = $this->client->get(
136         'tenants');
137 } catch (\GuzzleHttp\Exception\RequestException $exception) {
138     $response = $exception->getResponse();
139 }
140
141 // Transform the response into a JSON object.
142 $response = json_decode($response->getBody()->getContents());
143
144 // Dump the errors if present.
145 if (isset($response->errors)) {
146     self::assertTrue(false);
147 }
148
149 self::assertFalse(empty($response->data));
150
151 $id = AuvikTenant::first()->id;
152 $response = $this->client->get(
153     'inventory/device/info',
154     [
155         'query' => [
156             // 'fields' => [
157                 // 'deviceDetail' => implode(',', [
158                     // 'discoveryStatus',
159                     // 'components',
160                     // 'connectedDevices',
161                     // 'configurations',
162                     // 'manageStatus',
163                     // 'interfaces'
164                 ]),
165                 // ],
166                 'filter' => [
167                     ],
168                     // 'include' => 'deviceDetail',
169                     'page' => [
170                         'first' => 10,
171                     ],
172                     'tenants' => $id,
173                 ],
174             ],
175         );
176
177 $response = json_decode($response->getBody()->getContents());
178
179 if ($response != null) {
180     self::assertTrue(true);
181 } else {
182     self::assertTrue(false);
183 }
184

```

'CAN ACCESS TENANTS DEVICE DETAILS' TEST FUNCTIE

CODE

```

188 /**
189 * Collects device details on tenant on endpoint.
190 *
191 * @return void
192 */
193 public function test_can_access_tenants_device_details_endpoint()
194 {
195
196     $this->client = new \GuzzleHttp\Client([
197         'auth' => [config('services.auvik.username'), config('services.auvik.api_key')],
198         'base_uri' => config('services.auvik.base_url'),
199         'headers' => [
200             'Accept' => 'application/json',
201             'Content-Type' => 'application/json',
202             'Encode' => 'UTF-8',
203         ],
204     ]);
205
206     try {
207         // Get a response.
208         $response = $this->client->get(
209             'tenants');
210     } catch (\GuzzleHttp\Exception\RequestException $exception) {
211         $response = $exception->getResponse();
212     }
213
214     // Transform the response into a JSON object.
215     $response = json_decode($response->getBody()->getContents());
216
217     // Dump the errors if present.
218     if (isset($response->errors)) {
219         self::assertTrue(false);
220     }
221
222     self::assertFalse(empty($response->data));
223
224     $id = AuvikTenant::first()->id;
225     $response = $this->client->get(
226         'inventory/device/detail',
227         [
228             'query' => [
229                 // 'fields' => [
230                     //     'deviceDetail' => implode(',', [
231                         //         'discoveryStatus',
232                         //         'components',
233                         //         'connectedDevices',
234                         //         'configurations',
235                         //         'manageStatus',
236                         //         'interfaces'
237                         //     ]),
238                         // ],
239                     'filter' => [
240                         ],
241                     // 'include' => 'deviceDetail',
242                     'page' => [
243                         'first' => 10,
244                     ],
245                     'tenants' => $id,
246                 ],
247             ],
248         );
249
250     $response = json_decode($response->getBody()->getContents());
251
252     if ($response != null) {
253         self::assertTrue(true);
254     } else {
255         self::assertTrue(false);
256     }
257

```

RESULTAAT:

Session contents restored from 13-6-2022 at 17:31:35

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Try the new cross-platform PowerShell https://aka.ms/pscore6
```

```
PS C:\Users\r.vangemeren\Documents\Loader> php artisan test
Warning: TTY mode is not supported on Windows platform.
```

```
PASS Tests\Unit\AuvikAPITest
✓ start api testing
```

```
PASS Tests\Feature\AuvikAPITest
✓ can access tenants endpoint
✓ can access tenants details endpoint
✓ can access tenants devices endpoint
✓ can access tenants device details endpoint
```

```
Tests: 5 passed
Time: 8.18s
```

```
PS C:\Users\r.vangemeren\Documents\Loader> □
```

BIJLAGE 9: AUVIK API LOADER DATABASE TESTCASE

VOORBEREIDING:

| | | | |
|----------------------|--|--|---|
| Test Case ID: | AUVIK_API_LOADER_DATABASE_TES T | Beschrijving : | Testen van verschillende functies naar de database van de Loader server met betrekking tot de Auvik API Loader. |
| Gemaakt door: | Ruben van Gemeren | Reviewer: | W. Pols |
| Tester: | Ruben van Gemeren | Versie: | 1.0 |
| | | Datum: | 20-4-2022 |
| | | Resultaat: | PASS |
| | | Test Scenario: | Met verschillende functies de data van de database testen. |
| # | Vereisten: | Testdata: | |
| 1 | De Auvik API moet bereikbaar zijn. | Auvik API Authenticatie | |
| 2 | Database moet Auvik <i>tenant</i> data hebben. | Testcode (<i>AuvikConceptController</i>) | |

TESTDATA:

| Stap # | Beschrijving: | Verwacht resultaat: | Resultaat: | PASS / FAIL / NOT EXECUTED / SUSPENDED |
|--------|------------------------------|---|--|--|
| 1 | 'getAllTenants' functie | Alle <i>tenants</i> vanuit de database. | SUCCES [in 4.8 milliseconden] | PASS |
| 2 | 'getTenantDetail' functie | Alle details van alle <i>tenants</i> vanuit database. | SUCCES [in 1354 milliseconden / 1.35 seconden] | PASS |
| 3 | 'getAllClients' functie | Alle <i>tenants</i> van het type <i>client</i> . | SUCCES [in 5.3 milliseconden] | PASS |
| 4 | 'getAllMultiClients' functie | Alle <i>tenants</i> van het type <i>multiClient</i> . | SUCCES [in 5.2 milliseconden] | PASS |

| | | | | |
|----------|----------------------------|--|--|--------------|
| 5 | 'getMatches' functie | Alle <i>tenants</i> met een koppeling naar een klant locatie. | SUCCES [in 5.3 milliseconden] | PASS |
| 6 | 'getUnMatched' functie | Alle <i>tenants</i> zonder een koppeling naar een klant locatie. | SUCCES [in 30 milliseconden] | PASS |
| 7 | 'getDevice' functie | Alle apparaten van een willekeurige <i>tenant</i> . | SUCCES [in 1646 milliseconden / 1.64 seconden] | PASS |
| 8 | 'getLocationMatch' functie | Klant locatie met de locatie matches van de <i>tenant</i> . | NIET UITGEVOERD | NOT EXECUTED |

'GETALLTENANTS' FUNCTIE

CODE

```

59 // Get all Tenants
60 public function getAllTenants()
61 {
62     $start = hrttime(true);
63
64     $tenants = AuvikTenant::all();
65
66     $end = hrttime(true);
67
68     $miliSeconds = ($end - $start) / 1000000;
69     $seconds = $miliSeconds / 1000;
70
71     ddd("All Tenants [retrieved in $miliSeconds miliseconds / $seconds seconds]", $tenants);
72 }
73

```

RESULTAAT

Dump

Content

```
"All Tenants [retrieved in 4.8615 milliseconds / 0.0048615 seconds]"
```

Location

C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:71 

Dump

Content

```
Illuminate\Database\Eloquent\Collection {#1998 ▼
  #items: array:649 [▼
    0 => App\Mod... \AuvikTenant {#2000 ▶}
    1 => App\Mod.. \AuvikTenant {#2001 ▼
      #guarded: []
      +incrementing: false
      #casts: array:2 [ ...2]
      #connection: "mysql"
      #table: "auvik_tenants"
      #primaryKey: "id"
      #keyType: "int"
      #with: []
      #withCount: []
      +preventsLazyLoading: false
      #perPage: 15
      +exists: true
      +wasRecentlyCreated: false
      #escapeWhenCastingToString: false
      #attributes: array:6 [ ...6]
      #original: array:6 [ ...6]
```

'GETTENANTDETAIL' FUNCTIE

CODE

```
74 // Get all the tenant details
75 public function getTenantDetail()
76 {
77
78     $start = hrttime(true);
79
80     $tenant = AuvikTenant::first();
81
82     $details = $this->client->get("tenants/detail/", [
83         'query' => [
84             'tenantDomainPrefix' =>$tenant->domain_prefix,
85             ]
86         ]);
87
88     $end = hrttime(true);
89
90     $miliSeconds = ($end - $start) / 1000000;
91     $seconds = $miliSeconds / 1000;
92
93     ddd("All Tenant Details [retrieved in $miliSeconds milliseconds / $seconds seconds]", json_decode($details->getBody()->getContents()));
94
95 }
```

RESULTAAT

Dump
Content

```
"All Tenant Details [retrieved in 1354.4067 milliseconds / 1.3544067 seconds]"
```

Location

C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:94 

Dump
Content

```
{#13480 ▼
  +"data": array:649 [▼
    0 => {#1362 ▶}
    1 => {#1370 ▶}
    2 => {#1389 ▶}
    3 => {#1408 ▶}
    4 => {#1427 ▼
      +"type": "tenants"
      +"id": "614344563100490493"
      +"attributes": {#1428 ...8}
      +"relationships": {#1432 ...2}
    }
    5 => {#1446 ▶}
    6 => {#1465 ▶}
    7 => {#1482 ▶}
  ]
```

'GETALLCLIENTS' FUNCTIE

CODE

```
97   // Get all Clients
98   public function getAllClients()
99   {
100     $start = hrtime(true);
101
102     $clients = AuvikTenant::where('type', '=', 'client')->get();
103
104     $clientCount = $clients->count();
105
106     $count = AuvikTenant::count();
107
108     $end = hrtime(true);
109
110     $miliSeconds = ($end - $start) / 1000000;
111
112     ddd("$clientCount out of $count tenants are of type client [retrieved in $miliSeconds milliseconds]", $clients);
113 }
```

RESULTAAT

Dump
Content

```
"492 out of 649 tenants are of type client [retrieved in 5.3748 milliseconds]"
```

Location

C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:112 

Dump
Content

```
Illuminate\Database\Eloquent\Collection {#1841 ▼
  #items: array:492 [▼
    0 => App\Models\AuvikTenant {#1843 ▼
      #guarded: []
      +incrementing: false
      #casts: array:2 [ ...2]
      #connection: "mysql"
      #table: "auvik_tenants"
      #primaryKey: "id"
      #keyType: "int"
      #with: []
      #withCount: []
      +preventsLazyLoading: false
      #perPage: 15
      +exists: true
    }
  ]
```

'GETALLMULTICLIENTS' FUNCTIE

CODE

```

115 // Get all multiClients
116 public function getAllMultiClients()
117 {
118     $start = hptime(true);
119
120     $multiClients = AuvikTenant::where('type', '=', 'multiClient')->get();
121
122     $multiClientCount = $multiClients->count();
123
124     $count = AuvikTenant::count();
125
126     $end = hptime(true);
127
128     $miliSeconds = ($end - $start) / 1000000;
129
130     ddd("$multiClientCount out of $count tenants are of type multiClient [retrieved in $miliSeconds miliseconds]", $multiClients);
131 }

```

RESULTAAT

Dump

Content

```
"157 out of 649 tenants are of type multiClient [retrieved in 5.2146 miliseconds]"
```

Location C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:130 

Dump

Content

```
Illuminate\Database\Eloquent\Collection {#1506 ▶
    #items: array:157 [▼
        0 => App\Mod... \AuvikTenant {#1508 ▷}
        1 => App\Mod... \AuvikTenant {#1509 ▷}
        2 => App\Mod... \AuvikTenant {#1510 ▷}
        3 => App\Mod... \AuvikTenant {#1511 ▼
            #guarded: []
            +incrementing: false
            #casts: array:2 [ ...2]
            #connection: "mysql"
            #table: "auvik_tenants"
            #primaryKey: "id"
            #keyType: "int"
            #with: []
            #withCount: []
            +preventsLazyLoading: false
            #perPage: 15
            +exists: true
            +wasRecentlyCreated: false
    ]
}
```

'GETMATCHES' FUNCTIE

CODE

```

133     // Get all unmatched locations and attempted to match them
134     public function getMatches()
135     {
136
137         $start = hrtimer(true);
138
139         $locations = \App\Models\Customer\CustomerLocation::whereNull('auvik_id')->get();
140
141         foreach ($locations as $location) {
142
143             $file = $location->getJsonFileContents();
144
145             $match = AuvikTenant::firstWhere('domain_prefix', '=', $file->short_name);
146
147             $matches[] = $location->auvikTenant()->associate($match);
148
149             $location->save();
150         }
151
152         $matched = \App\Models\Customer\CustomerLocation::whereNotNull('auvik_id')->get();
153
154         $total = \App\Models\Customer\CustomerLocation::all()->count();
155
156         $count = $matched->count();
157
158         $end = hrtimer(true);
159
160         $miliSeconds = ($end - $start) / 1000000;
161
162         ddd("Amount of matched tenants: $count out of $total [retrieved in $miliSeconds miliseconds]", $matched);
163
164     }

```

RESULTAAT

Dump

| | |
|----------|--|
| Content | "Amount of matched tenants: 147 out of 171 [retrieved in 39.0309 miliseconds]" |
| Location | C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:162 |

Dump

| | |
|---------|--|
| Content | <pre>Illuminate\Database\Eloquent\Collection {#1653 ▼ #items: array:147 [▼ 0 => App\Mod... \CustomerLocation {#1655 ▶} 1 => App\Mod... \CustomerLocation {#1657 ▼ #guarded: [] #connection: "mysql" #table: "customer_locations" #primaryKey: "id" #keyType: "int" +incrementing: true #with: [] #withCount: [] +timestamps: false]</pre> |
|---------|--|

'GETUNMATCHED' FUNCTIE

CODE

```
// Count all unmatched locations
public function getUnMatched()
{
    $start = hrtime(true);

    $locations = \App\Models\Customer\CustomerLocation::whereNull('auvik_id')->get();

    $total =\App\Models\Customer\CustomerLocation::count();

    $unMatched = \App\Models\Customer\CustomerLocation::whereNull('auvik_id')->count();

    $end = hrtime(true);

    $miliSeconds = ($end - $start) / 1000000;

    ddd("Amount of unmatched tenants: $unMatched out of $total [retrieved in $miliSeconds miliseconds]" , $locations);
}
```

RESULTAAT

Dump

Content

"Amount of unmatched tenants: 24 out of 171 [retrieved in 6.1484 miliseconds]"

Location

C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:181 

Dump

Content

```
Illuminate\Database\Eloquent\Collection {#1379 ▾
  #items: array:24 [▼
    0 => App\Mod... \CustomerLocation {#1381 ▶}
    1 => App\Mod... \CustomerLocation {#1383 ▶}
    2 => App\Mod... \CustomerLocation {#1389 ▶}
    3 => App\Mod... \CustomerLocation {#1387 ▶}
    4 => App\Mod... \CustomerLocation {#1390 ▶}
    5 => App\Mod... \CustomerLocation {#1391 ▶}
    6 => App\Mod... \CustomerLocation {#1392 ▶}
    7 => App\Mod... \CustomerLocation {#1393 ▶}
    8 => App\Mod... \CustomerLocation {#1394 ▶}
    9 => App\Mod... \CustomerLocation {#1395 ▶}
    10 => App\Mod... \CustomerLocation {#1396 ▶}
    11 => App\Mod... \CustomerLocation {#1397 ▶}
    ...]
```

'GETDEVICE' FUNCTIE

CODE

```

185 // Get all devices from random Tenant
186 public function getDevice()
187 {
188     $start = hrtime(true);
189
190     $tenant = AuvikTenant::inRandomOrder()->get()->first();
191
192     $response = $this->client->get(
193         'inventory/device/info',
194         [
195             'query' => [
196                 // 'fields' => [
197                     // 'deviceDetail' => implode(',', [
198                         // 'discoveryStatus',
199                         // 'components',
200                         // 'connectedDevices',
201                         // 'configurations',
202                         // 'manageStatus',
203                         // 'interfaces'
204                     ]),
205                     ],
206                     // 'filter' => [
207                         // 'deviceType' => $deviceType,
208                         ],
209                     // 'include' => 'deviceDetail',
210                     'page' => [
211                         // 'first' => 10,
212                     ],
213                     'tenants' => $tenant->id,
214                 ],
215             ],
216         );
217
218     $response = json_decode($response->getBody()->getContents());
219
220     $end = hrtime(true);
221
222     $miliSeconds = ($end - $start) / 1000000;
223     $seconds = $miliSeconds / 1000;
224
225     ddd("Get Devices of random tenant: '$tenant->domain_prefix' [retrieved in $miliSeconds milliseconds / $seconds seconds]", $response);
226 }
227

```

RESULTAAT

Dump

Content

"Get Devices of random tenant: 'manpower' [retrieved in 1646.2125 miliseconds / 1.6462125 seconds]"

Location

C:\Users\r.vangemeren\Documents\Loader\app\Http\Controllers\Auvik\AuvikConceptController.php:225 

Dump

Content

```

{#2840 ▾
+ "data": array:100 [▼
  0 => {#2632 ▶}
  1 => {#2627 ▶}
  2 => {#2613 ▶}
  3 => {#2599 ▶}
  4 => {#2585 ▶}
  5 => {#2571 ▶}
  6 => {#2557 ▶}
  7 => {#2543 ▶}
  8 => {#2529 ▶}
  9 => {#2515 ▶}
  10 => {#2501 ▶}
  11 => {#2487 ▶}
  12 => {#2473 ▶}
  ...
]

```

'GETLOCATIONMATCH' FUNCTIE

CODE

```
229 |     public function getLocationMatch()
230 |     {
231 |
232 |         //Check if fields exist
233 |         //Get all already matched locations
234 |
235 |         //setAttribute the fields
236 |         //Return both full and not full matches
237 |
238 |         $details = $this->getTenantDetail();
239 |
240 |
241 |         $locations = CustomerLocation::whereNotNull('auvik_id')
242 |             // ->with('auvikTenant')
243 |             ->get();
244 |
245 |         $customerLocations = \App\Models\Customer\CustomerLocation::whereNotNull('auvik_id')->get();
246 |
247 |
248 |         $auvikLocation = $this->getTenantDetail(AuvikTenant::first())->data;
249 |
250 |
251 |
252 |         $addresses = $customerLocations->map(
253 |             function ($location) {
254 |                 return $location->getJsonFileContents()->addresses[0];
255 |             }
256 |         );
257 |         $auvikLocation->matches = $addresses->filter(function ($address, $key) use ($auvikLocation) {
258 |             $weight = 0;
259 |
260 |             return $address;
261 |
262 |             if ($address->country_iso_code == $auvikLocation->attributes->address->country) {
263 |                 $weight++;
264 |             }
265 |
266 |             if ($address->zipcode == $auvikLocation->attributes->address->postalCode) {
267 |                 $weight++;
268 |             }
269 |
270 |             return $weight >= 2;
271 |         });
272 |
273 |         ddd($auvikLocation);
274 |
275 |     }
276 }
```

BIJLAGE 10: DEVELOPMENT DAGSTART 20 JUNI 2022

deelnemers: **Ruben van Gemeren, T. Treffers, T v. Herwijnen en W. Pols**

datum: 10 juni 2022

notulist: Ruben van Gemeren

WAT HEBBEN WE GISTEREN GEDAAN?

Van 16:00 tot 16:30 is er een maandmeeting gehouden waarbij onder andere de voortgang van de callcenters zijn besproken.

RUBEN VAN GEMEREN:

- Gewerkt aan de Auvik Loader

T. TREFFERS:

- Informatie van de callcenters gedocumenteerd.
- Toelichting gegeven bij de maandmeeting

T V. HERWIJNEN:

- Begonnen met de CRIS-X FusionAuth connector. Voor de LDAP en andere connectors is een Reactor license (FusionAuth premium) nodig.
- De HABdesk API verbeteren.

W. POLS:

- Verder met de T-Mobile mobiele orders. Ze staan klaar maar ze worden niet gebruikt omdat er nog steeds gegevens vanuit Doceri.
- Samen met Ruben door de code van Auvik heenlopen.
- Heeft voorbereidingen gemaakt voor het live zetten van de Auvik Loader.

WAT GAAN WE VANDAAG DOEN?

RUBEN VAN GEMEREN:

- Met W. Pols werken aan het beschikbaarstellen van de Auvik Loader
- Verder werken aan het vak functional programming.

T. TREFFERS:

- De callcenters uitwerken

- Samen met Job de CRIS-X klanten nalopen om ze vervolgens naar ITGlue te pushen.
- Pbx item wat nu nog in een TestController gebeurt ombouwen naar een Job en in de cronjob zetten.
- Een start maken met het inladen van de users / devices vanuit de Broadsoft. Kan pas gedaan worden als het punt hierboven correct is ingeregeld. Dit is namelijk afhankelijk ervan.

T V. HERWIJNEN:

- HABdesk endpoint ombouwen naar een daadwerkelijke API
- Achter de vakantie aan bij T v.d. Waal

W. POLS:

- De Auvik Loader live zetten met de hulp van Ruben.

PROBLEMEN DIE EXTRA AANDACHT NODIG HEBBEN

T V. HERWIJNEN:

- Het usermodel van FusionAuth kan niet worden gebruikt binnen Laravel, W. Pols heeft een mogelijke oplossing bedacht met de *guard*, dit gaat Tim proberen.

W. POLS:

- Doceri mits bepaalde data.

BIJLAGE 11: DEFINITIELIJST HABDESK

Bij het HABdesk project komen veel verschillende bronnen bij elkaar samen, daarbij moet er data van deze bronnen worden gematched met andere data. Omdat dit project een grote schaal heeft en over een lange periode wordt ontwikkeld is het van belang om een referentie document op te stellen waar de belangrijkste begrippen worden toegelicht.

Interne begrippen worden hier ook toegelicht.

HABDESK:

- **HABdesk** = Handy Assistant Buddy
- **HABapp / HAP / Happlicatie** = HABdesk Applicatie
- **Client** = De (Tennacy-)Tenant onder een-/van een- MSP

FUSIONAUTH

- **FAT** = FusionAuth Tenant
- **FAP / FApp** = FusionAuth Applicatie

LOADER SERVER

- **Wasmachine** = De gehele Loader server
- **<naam> Loader** = Een individuele Loader binnen de Loader server, denk aan de Auvik API Loader.

BIJLAGE 12: INTERVIEW DIRECTEUR HELMINK

OPZET

- In de afgelopen 10 jaar is Cloud steeds populairder geworden. zo ook applicaties die lijken op die van HABdesk. check
- In mijn onderzoek heb ik gemerkt dat het product HABdesk van jouw visie veel overeenkomsten heeft met een structuur van een *cloud* app. Waarbij een licentie wordt gekocht die toegang geeft tot bepaalde functionaliteit. daarbij wordt er ook gebruik gemaakt van mulitenantie wat ook een standaard is binnen *Cloud* en *SaaS* apps.
- Ik vroeg mij af waar de inspiratie voor HABdesk vandaan is gekomen?
- Cloud wordt vaak gezien als de "beste oplossing" voor vele problemen, ben je het daar mee eens? check
- HABdesk wordt/is een web applicatie. was dit altijd al het plan?
- Is er een bepaalde reden waarom HABdesk wordt ontwikkeld zoals dat nu gebeurt?
- Dit hebben we al een keer eerder snel besproken maar ik zou het toch nog even duidelijk willen horen. Wat is de hoofdreden waarom de Loaders intern worden ontwikkeld? en waarom er geen gebruik wordt gemaakt van een *cloud* oplossing of een andere derde partij oplossing.
- Het feit dat alle developers elke dag bezig moeten zijn met HABdesk is uiteindelijk ook niet goedkoop. als je de progressie nu bekijkt. is dat wat je voor ogen had?
- Waar is HABdesk over 5 jaar? wordt dit een nieuwe standaard binnen Helmink?
- Het idee om Loaders te gebruiken, hoe is dit ontstaan? check
- ik weet dat bepaalde data intern wordt gehost. dit kan gezien worden aan ouderwets. wat is de reden waarom dit gebeurt?
- Als je zo naar het HABdesk project kijkt, Wat zijn de grootste valkuilen/problemen.

UITWERKING

Op 4 mei 2022 is er een interview plaatsgevonden met de CEO van Helmink meneer Bellekom. Hij is al meer dan 15 jaar werkzaam bij Helmink en is sinds 2014 Algemeen directeur. Meneer Bellekom is bij alle afdelingen binnen Helmink actief aanwezig en heeft veel operationele en managing kennis, daarnaast heeft meneer

Bellekom kennis van de technische kant van de telecommunicatie branche. Hij is onofficieel lead developer en is de leidinggevende bij alle projecten binnen Helmink.

In dit semigestructureerde interview wordt er gevraagd naar de visie van Helmink met betrekking tot het HABdesk project. Er wordt gevraagd naar inspiratie en toekomstplannen van HABdesk en Helmink zelf, daarnaast wordt er dieper ingegaan op de architectuur van HABdesk waarom bepaalde keuzes zijn gemaakt. Meneer Bellekom heeft veel ideeën en is het creatieve brein achter veel van de development gerelateerde beslissingen binnen Helmink. Tijdens het interview werd het duidelijk hij graag voorbeelden en verwijzingen gebruikt om abstracte termen toe te lichten.

VRAAG EN ANTWOORD MENEER BELLEKOM

In mijn onderzoek heb ik gemerkt dat het product HABdesk overeenkomsten heeft met een structuur van een cloud app. Als het gaat om de licentie, multitenantie, en architectuur. Is dit toeval?

- ik vroeg mij af waar de inspiratie voor HABdesk vandaan is gekomen?

Het is een combinatie van heel veel dingen. Je begint op een gegeven moment ergens. Helmink is begonnen bij het leveren van connectiviteit, waarbij Helmink hardware leverde. Hierna wilde Helmink de status van deze apparaten kunnen monitoren. Voordat er gewerkt wordt met het huidige pakket Auvik zijn er twee andere pakketten gebruikt. bij het monitoren van apparaten wordt het apparaat zelf wel gemonitord, maar de verbinding niet, hierdoor is er een gat tussen de hardware en de dienst. Een dienst wordt weergegeven binnen software van de provider, zoals KPN of Starlink. Het apparaat maakt gebruik van deze dienst maar het is binnen het monitoringssysteem niet te zien wat voor verbinding er wordt gebruikt, alleen dat het apparaat werkt en welke gebruikers er mee hebben gecommuniceerd. Je kan een monitoringssysteem wel 'misbruiken' om toch bepaalde data te gebruiken, maar dan zit je nog steeds te worstelen, daarbij is de data nog steeds niet in één keer beschikbaar.

Helmink bleef groeien en nieuwe problemen deden zich voort. Als er een storing is in een bepaalde regio of bij een bepaalde drager, is het binnen de huidige situatie moeilijk te achterhalen wat het probleem is. Het monitoringssysteem laat zien dat er x aantal apparaten problemen hebben, maar kan niet zien wat het probleem is.

Dit is één van de redenen waarom wij onderdelen aan elkaar willen koppelen.

Als er wordt gekeken naar de architectuur van HABdesk zijn er overeenkomsten, zo is multitenancy standaard binnen de Cloud, of je nu Slack pakt, of Harvest, het bestaat allemaal uit dezelfde onderdelen. Het gene wat HABdesk zich doet individualiseren is dat het informatie van verschillende pakketten consolidiert, dit gebeurt bij de meeste cloud oplossingen niet, die leggen de focus op hun eigen data. HABdesk levert licht in de duisternis als het gaat om welke data er bij welke klant is. HABdesk bestaat uit verschillende applicaties, zo wordt er niet alleen gebruik gemaakt van data uit KPN, maar ook van T-Mobile, dit concept is unieker.

Om een vergelijking te maken, Slack heeft één ding, dat leveren ze, daarnaast hebben ze heel veel koppelingen (via API Exchange) met andere applicaties. Deze koppelingen geven andere applicaties de mogelijkheid om gebruik te maken van Slack als een manier om informatie te geven aan hun gebruikers. Naast deze koppelingen doet Slack maar één ding, je kan chatten met elkaar. Voor elke klant is dat naast een paar cosmetische en rechtelijke hetzelfde.

HABdesk is bedoeld als een samenvoeging van allemaal van dit soort applicaties, er zijn daarbij twee kanten.

Ten eerste is HABdesk ontstaan om de werkdruk van Helmink te verlichten, Hierdoor kunnen nieuwe medewerkers een stuk sneller worden geïntegreerd binnen het bedrijf. Met het voordeel dat Helmink krijgt met HABdesk kunnen de klanten ook beter worden geholpen. Dit levert bestaansrecht op voor Helmink.

Ik ben altijd geïnteresseerd geweest in SaaS en Cloud applicaties, maar daarnaast vond ik het ook leuk om te kijken naar hoe deze oplossingen geïntegreerd konden worden met elkaar. Daar ligt voor mij de waarde van de applicatie.

Als ik het zo hoor wordt HABdesk een verzamelplek van allemaal applicaties gebasseerd op de diensten die jullie op dit moment leveren.

Ja, daarbij sluit ik het niet uit om na onze eigen diensten ook daarbuiten te kijken, als ons het ene lukt, waarom het andere niet?

Voordat je services kan koppelen moet er eerst achterhaald worden waar de data vandaan komt, zodra we kunnen zien van wie de data is, kunnen we dit ook makkelijk aan de klant zelf laten zien, dat geeft rust voor Helmink maar verhoogt ook de snelheid van informatiestromen. Door deze verbetering wordt de gebruikerservaring ook beter.

Het concept van de Loader is bedacht voordat ik hier stage ben gaan lopen, hoe is dit concept tot stand gekomen?

Ik heb mij in de afgelopen tijd verdiept in iPaaS en low-code oplossingen. [Betty Blocks](#), [Dell Boomi](#), [Workato](#) en [Zapier](#).

heel veel van die pakketten kunnen wel data ophalen van meerdere bronnen, daarnaast kan je data nog aanpassen of sorteren. Het probleem is dat de oplossing de data snel weer kwijt wil, deze data kan niet worden opgeslagen. Naast de kosten (die nog wel te veroorloven zijn) is dat een pakket zoals Workato geen oplossing heeft bij problemen met datazuiverheid. Als er een activiteit wordt gestuurd waarbij een nieuwe medewerker moet worden aangemaakt, Weet Workato niet of deze medewerker bestaat, er wordt deze dus gewoon toegevoegd. Er zijn manieren om dit probleem op te lossen, maar dit wordt handwerk dat ook extra kosten met zich mee neemt.

Voor het idee van HABdesk en de Loader omgeving deden we veel met point-to-point API calls, ging er iets mis met een API ging alles mis, omdat data niet wordt gecentraliseerd op één omgeving. Hier is het idee van de Loader uit ontstaan.

Voor de Loader is er nog een probleem met het reageren om nieuwe data, waarbij het belangrijk is dat er een bepaalde flexibiliteit aanwezig is die ervoor zorgt dat een verandering in data niet zomaar alles uit elkaar laat vallen. Oplossingen die je kan kopen hebben vaak al een systeem opgezet die goed om kunnen gaan met dit soort problemen, dat zouden wij nog moeten bouwen. Dit is ook een soort evolutie van software, waarbij je niet alles in één keer goed kan hebben staan.

De Loader is opgebouwd als een ESB (Enterprise Service Bus), is dit met opzet gedaan?

Ja ik herken de term. Niet elk iPaaS systeem maakt gebruik van de ESB architectuur. Het probleem met ESB's is dat ze overladen zijn met functionaliteiten, ook zijn de kosten erg hoog. Deze twee componenten leveren veel druk op de ontwikkeling. Als je als bedrijf je helemaal inricht op één systeem ([MuleSoft](#), [Mendix](#)), is het bij een grote ESB moeilijk om dit om te zetten, daarbij ben je ook beperkt in wat je allemaal kan doen met de gekozen oplossing.

Vroeger had je hele zware koppelingen met heel veel data. Nu krijg je een simpele XML of JSON bestand. Er zijn geen lichte ESB pakketten.

Als het gaat om de architectuur van de Loader?

Ja deze is inderdaad geïnspireerd door de ESB architectuur. Ik heb nooit beweert dat de architectuur door ons is verzonden. Veel applicaties zijn toch nieuwere versies van bestaande ideeën. Zo is Slack niet veel meer dan een nieuwere versie van de MSN chatroom.

De klant zelf maakt het niet uit wat voor architectuur er gebruikt wordt, en is alleen geïnteresseerd in het eindresultaat. Daarom is development een erg oneerbiedige baan, IT algemeen. Als alles goed werkt hoor je niemand, maar zodra er iets mis gaat is er veel kritiek. Mensen hebben geen interesse in hoe dingen werken, maar dat ze werken.

Ik heb meer perspectief gekregen, ik heb voor dit gesprek nooit echt meegekregen hoe HABdesk en de Loader tot stand zijn gekomen.

Mijn develop proces is als volgt. Er ontstaan dingen, dan probeer ik te kijken waar de knelpunten kunnen komen, dan ga ik onderzoek doen, en dan komen oplossingen naar voren. Deze pakketten komen in allemaal verschillende soorten en maten. in de Cloud, on premise, met database connectie of zonder, met import of zonder.

Ik laat mij ook informeren, zo ben ik bij een bedrijf geweest in Rotterdam genaamd [We Are Frank](#). Zij hadden een soortgelijk pakket ontwikkeld bij Nationale Nederlanden. Banken zijn erg complex en maken gebruik van oude systemen om data te verwerken. We are Frank heeft dit pakketten open source gemaakt. Alles wat ik wilde kon, het is gebouwd met Java, wat het super flexibel maakt. Maar na een demonstratie was het duidelijk dat het systeem erg groot en zou erg veel tijd kosten voor hiermee gewerkt kon worden. We hebben onderdelen van dit programma gebruikt om het concept van de Loader te bedenken. Het is mogelijk om zonder

programmeer ervaring integraties op te bouwen, maar voor maatwerk wordt het al snel duur.

Als ik het zo hoor komt dit bij meer partijen voor, alles kan, maar als je specifieke functionaliteit wil moet je extra betalen

Ja dit werkt bij alle partijen zo, Exact, Avas. Alles kan, maar er moet wel voor betaald worden.

Ik heb dus wel wat tijd gestoken in het uitzoeken van deze oplossingen. Ik ben erachter gekomen dat het integreren van een soortgelijk systeem binnen Helmink teveel tijd zou kosten, dat is tijd die we niet hebben. Dit is de reden om uiteindelijk voor ons eigen systeem te gaan. Als het product uiteindelijk werkt kunnen we met de winst het systeem nog groter maken.

En daar is de architectuur ook voor ontworpen dus?

Ja het is erg flexibel, dit betekent dat sommige zware processen uiteindelijk verplaatst kunnen worden mocht dit mogelijk zijn.

En die schaalbaarheid kunnen jullie ook toepassen omdat alles intern wordt ontwikkeld.

Jazeker.

Ik had nog een vraag over jullie server die in het kantoor staat, wat staat daar allemaal op?

De hele HABdesk omgeving draait in de Cloud, in Amsterdam. Deze kunnen dus ook worden opgeschaald. Wat we hier intern hebben is de meest gevoelige informatie van klanten, paspoorten, klantgegevens, getekende contracten env.

Dit heeft te maken met privacy redenen als ik het zo hoor?

Ja, ook is het niet heel veel data dus is het niet nodig om dit ergens anders te plaatsen. Hier hebben we een mooie virtuele omgeving waar dit allemaal wordt opgeslagen. We moeten de data ook 7 jaar bewaren, voor de belastingdienst. We hebben deze data niet vaak nodig maar we moeten het wel hebben.

CRIS-X draait wel nog intern, maar dit is ook een intern systeem dat buiten de deur geen extra waarde heeft. Medewerkers kunnen via een VPN een verbinding opzetten met CRIS-X vanuit huis.

Voor de rest gebeurt alles buiten de deur.

We gaan lokaal backups opslaan van het CMDB systeem IT-Glue en deze backups komen ook weer op de Cloud.

De Loader server staat dus ook in de Cloud?

Ja

In mijn onderzoek werd het duidelijk dat de algemene mening is dat Cloud en iPaaS de "beste" keuze zijn als het gaat om een integratie oplossing, als ik dit zeg, ben je het daar mee eens?

Ligt eraan waar je data staat. Als je het hebt van schaalbaarheid en processor kracht die nodig is om data te verwerken is een Cloud oplossing de beste keuze (flexibeler). Maar uiteindelijk is de Cloud gewoon een andere PC. Er zijn twee factoren die invloed hebben op de keuze die gemaakt moet worden. 1. Hoeveel data moet er worden verwerkt, en daarbij hoeveel processing kracht heb je nodig om met deze data te werken. 2. Waar staat je data?

Als je 14 SaaS applicaties heb die al niet on-premise staan met elkaar wil laten communiceren, is het beter om een Cloud oplossing (iPaaS) toe te passen. Hierdoor voorkom je ook een Single Point Of Failure (SPOF). Als je als bedrijf een eigen datacenter heb staan met veel data, is het niet ideaal om al deze data naar de Cloud te brengen (met alle kosten en problemen die daarbij komen kijken) om deze data vervolgens weer intern op te slaan. Je kan een ferrari kopen, maar als je alleen maar naar de winkels wil is dit niet even handig.

Binnen de IT is het pushen van Trends heel normaal. Garnter en Forester zijn geen onafhankelijke partijen, de partij met het meeste geld komt altijd bovenaan in een ranglijst. Als een grote partij wint betekend dat niet dat het het beste pakket was. En als mensen zeggen "Cloud, cloud, cloud" betekend dat niet dat Cloud de enige of beste oplossing is voor elk probleem.

Zie je Helmink als een bedrijf dat laat zien dat Cloud niet altijd de beste oplossing hoeft te zijn?

Ja en nee, als HABdesk populair wordt en het wordt uitgebreid op een internationaal niveau, dan is het handiger om een AWS en een Azure te gebruiken die al datacenters heeft staan overal, dan het is om dat zelf allemaal te gaan doen.

Maar om te zeggen dat in dit stadium AWS of Azure de betere oplossing zou zijn? Dat denk ik niet, het is vooral veel duurder.

Een leger gaat ook geen Cloud omgeving gebruiken, om hun eigen redenen.

Als je het systeem zelf opbouwt is het ook aan jezelf om het te onderhouden, die last heb je niet als je gebruik maakt van een Cloud oplossing. Maar hoe meer eigen functionaliteit je toepast binnen de Cloud, hoe afhankelijker je wordt.

Wat ik met HABdesk wil is hetzelfde als wat je hebt met Slack. Slack biedt veel functionaliteit om zelf workflows op te zetten en koppelingen te maken. Maar als je Slack voor een tijdje gebruikt. Probeer er maar eens vanaf te komen, dat gaat niet. Want dan moet je het gaan nabouwen in een ander pakket, dan komen dezelfde problemen naar voren.

Het idee van Cloud is aantrekkelijk omdat het laagdrempelig is, maar zodra je wil veranderen van pakket kom je in dezelfde problemen als grote bedrijven die hun on-premise applicaties willen vervangen.

De markt doet alsof alles veranderd, maar eigenlijk wordt alles verplaatst. Er zijn voordelen zoals schaalbaarheid en machine learning die onrealistisch zijn voor on-premise oplossingen. Maar dat is maar voor een specifiek gedeelte van de markt.

In mijn onderzoek werd Cloud erg geprijsd, en alles moet Cloud en iedereen moet Cloud, en ik heb niet veel nieuwe bronnen gevonden waar Cloud werd bekritiseerd.

Het pushen van trends is een groot deel van de markt. En het gebeurt al jaren. Trends worden gedreven door de leveranciers. De servers draaien in Amsterdam. Azure is niet nodig om we nog geen behoefte hebben aan internationale services, daarbij is Azure niet goedkoop.

Want jullie leveren Azure als service toch?

Ja, sommige klanten zitten in het Azure ecosysteem waardoor ze geforceerd zijn om daadwerkelijk Azure te gebruiken, dat is ook een manier voor Microsoft om trends te pushen.

Het is goed dat er geen "beste oplossing" is. Zo is er ook niet 1 programmeertaal. Sommige zijn goed in berekeningen, terwijl andere meer gericht zijn om web development. Door deze verschillen is er ook meer innovatie

Hoe ziet HABdesk er uit over vijf jaar?

In deze markt is vijf jaar erg lang. Op korte termijn hoop ik dat HABdesk een grote rol gaat spelen binnen Helmink en onze klanten. Waarbij klanten niet meer weten wat er voor HABdesk allemaal gebeurden.

Ik hoop dat Helmink commercieel kan groeien. De gesprekken die ik heb met klanten zijn positief en als bedrijf loopt Helmink zeker niet achteraan als het gaat om techniek.

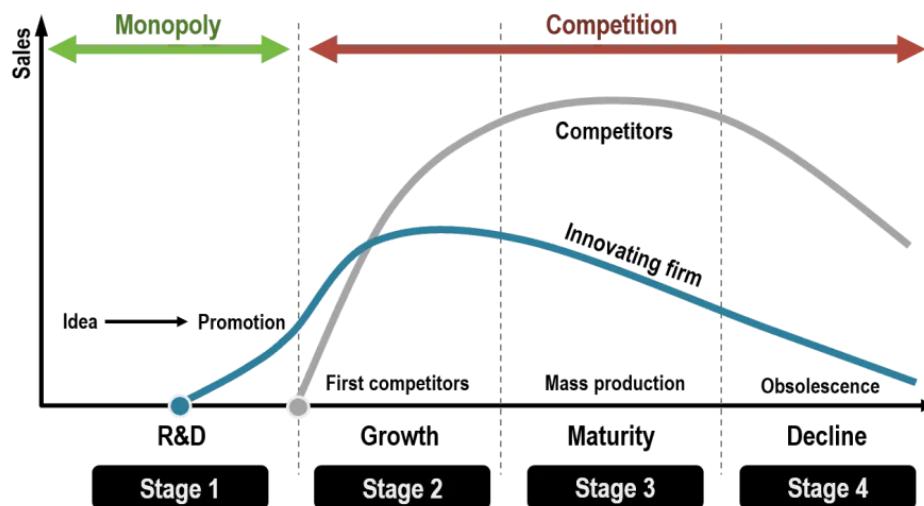
Ik ben van mening dat hoewel de meeste software bedrijven qua techniek op ons voorlopen, dat er bij die bedrijven een zekere hoeveelheid creativiteit mist.

Als voorbeeld. Apple legt niet veel nadruk op de technische aspecten van de telefoons die zij verkopen. De gemiddelde klant (van apple) wil gewoon resultaat. Het moet er goed uitzien, het moet werken en het moet naar verwachting zijn. Aan de andere kant heb je Android telefoons die meer gericht zijn op de technische aspecten, de snelheid van nieuwheid van de hardware.

Ik hoop dat wij als bedrijf extra aandacht kunnen geven aan onze klanten, waardoor we een persoonlijk advies kunnen geven. Als we bij de waterbus niet creatief waren geweest hadden wij nooit zo'n mooi project kunnen leveren, een gemiddeld IT bedrijf zou dit niet kunnen doen, die leveren een product, that's it. Wij gaan daarin een stap

verder en leveren persoonlijk advies en helpen de klant met het visualiseren van ideeën. Ik zie dat HABdesk op korte termijn veel mogelijkheden biedt om Helmink te doen groeien.

Weet je wat het is met deze markt (en eigenlijk alle markten). Als je geen grote partij hebben moet je innoveren om bestaansrecht te hebben. Alle markten moeten zich onderscheiden. Als klein bedrijf vecht je eerst tegen de gevestigde orde, hier komt de meeste inspiratie naar voren. Uiteindelijk groei je als bedrijf waardoor je zelf de gevestigde orde.



Ik heb voor de rest geen vragen. Bedankt voor alle antwoorden.

