Micro frontend architecture for cross framework reusability in practice

Tim De Smet

Supervisor(s): prof. dr. Veerle Ongenae, dhr. Thierry Bruyneel, dhr. Wim Vergouwe, ing. Luuk Ostendorf

Abstract— **Micro frontends are a recent evolution in frontend development. This study around the possibility to apply micro frontends for creating reusable uniform functionalities was commissioned by delaware, a global technology consultancy company. The goal is to reduce the workload of frontend development and the coinciding development cost. The thesis starts with a literature study about the micro frontend architectural style that also discusses Domain Driven Design and microservices. The state of the art chapter describes current possible implementation techniques for micro frontends. Built upon the findings of the literature study and the state of the art a proof of concept was developed. It serves as an on research based answer to the issue that delaware needed addressed. Thus, a decoupled, performant, technology independent and search engine optimization friendly micro frontend was constructed. The micro frontend was both nested in a React and an Angular application to demonstrate the independence related to technology. The proof of concept also shows how micro frontends can be configured and how communication between them can be enabled. An in-detail description of the chosen approach used to construct the micro frontend as well as an extensive technical report is provided. The thesis concludes with an analysis of the proof of concept from a performance and search engine optimization point of view.**

Keywords—**Micro frontends, architecture, reusability, frontend development, LitElement, consultancy**

I. Introduction

Partly due to the increasing trend of platform economies, there is an increasing need for scalable development of frontends. In addition, it has been established that customers frequently wish to have similar functionalities implemented. delaware is looking for a way to cater to its customers in this area and to anticipate this evolution. The company sees the novel trend of micro frontend architecture as the possible solution to this problem. It is still unclear how this approach can be applied within the consultancy landscape. Therefore research should be conducted into the possibility of using micro frontends to reduce the pressure on frontend development and the development costs associated with it.

II. Domain-Driven Design

One of the fundamental concepts related to microservices, Domain-Driven Design (DDD), was introduced in 2004 by Evans [1]. DDD is a software development approach that builds upon the following basic principles: during the process the focus must be on the domain, the model must be designed collaboratively by all parties involved and the parties involved must communicate on the basis of a shared and domain-specific language [2].

Millet & Tune [3] propose that the DDD approach offers solutions for designing software in the form of strategic and tactical patterns. The strategic patterns allow to structure the architecture succefully. The tactical patterns are code patterns that serve as building blocks to model complex bounded contexts effectively. The bounded context strategical pattern is a description of a boundary within which a model is defined. DDD provides a framework for shaping design decisions as well as a technical vocabulary to discuss the design of the domain [1].

III. Microservices

Microservices are the result of applying the microservice architectural style. The style uses the approach of building a complex application from a collection of collaborating software components called services, each organized around a business function, a so-called business capability, each of which can be independently deployed [4]. In a monolithic architecture, the application is executed as a whole. This architectural style is disadvantageous when the size of codebase becomes too large.

Microservices as architectural style provides the following benefits according to Richardson [5]: the style allows for continuous delivery and deployment of large and complex applications; the services are small and therefore easier to maintain and each service is independently deployable, scalable and testable. In addition to a generally more reliable application, this leads to more autonomy for teams, both in terms of development and technological choices. Given that the landscape of technology does not stand still, the speed at which one can adopt new technologies is also a significant advantage compared to monolithic architectures.

However, Newman [6] points out that applying microservices is not the right approach for every context, given the additional complexity that is introduced when using distributed systems. An additional difficulty with microservices is the decomposition of the whole into several services. Each service should be responsible for a business capability. This can be realized on the basis of various patterns. Dividing using the DDD approach is one of the most prevalent patterns for this. The resulting vertical division into microservices with own datastores is an application of the concept of vertical decomposition. However, the multidisciplinary teams responsible for these vertical splits across the backend need not be limited to the backend.

IV. Micro frontends

The limitations of monolithic systems in terms of scaling also apply on a monolithic frontend [7]. Micro frontends are the result of the architectural style of microservices where services are extended with user interfaces. The vertically organized teams are also responsible for the frontend. The teams are, as shown in Figure 1 organized around business capabilities and are therefore fully interdisciplinary and completely end-to-end [8].

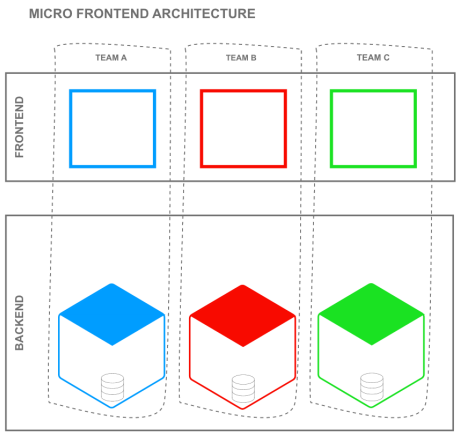


Fig. 1. Micro service architecture

From micro frontends, like microservices, it is expected that: they are organized around business capabilities; can be deployed independently and in a continuous manner; can work together regardless of technological implementation choices; promote the availability and the extent to which they are maintainable. Every implementation technique for micro frontends must therefore be tested against these desired properties. Furthermore, the performance that the possible techniques facilitate is also a measure of their suitability.

V. State of the art

The implementation of the micro frontend architecture is already carried out in various ways in practice. In the thesis, the techniques mentioned below as described by Jackson [9], Mezzalira [7] and Geers [8] are tested against the aforementioned foundations of the micro frontends architectural style. The most common implementation methods of micro frontends include: using hyperlinks between applications to simulate one application; integrating micro frontends packages at build-time; using iframes, JavaScript or Web Components to integrate micro frontends; server side integration and edge side integration.

VI. Proof of Concept

delaware wants to offer a number of uniform functionalities that can be plugged into the existing sites of customers. Since it has been established that some functionality keeps returning, the reusability of these solutions would reduce the cost as well as the delivery time. The PoC serves to illustrate how this can be achieved within the consultancy landscape. Technology independence, performance and Search Engine Optimization (SEO) are particularly important in the initial phases.

Since no individual implementation method was found to be appropriate for the problem, a hybrid method was used. The micro frontend itself was built using the Web Components integration method, a second version was built using LitElement, a light-weight library that provides a class for creating Web Components. In order to distribute the micro frontend, it was packaged and the package was published.

The desired technology independence was demonstrated by using the micro frontend in both a React and an Angular application. Meeting the SEO requirement required a stand-alone infrastructural solution; a proxy server was provided that could provide the necessary translation for Server Side Rendering to improve the ranking.

VII. Analysis

By distributing the micro frontend instead of deploying, the independent and continuous deployment foundations of micro frontends have been violated. The PoC does meet the other foundations. For the performance analysis three scenarios were compared in terms of rendering speed: a page that uses an unoptimized Web Components micro frontend; a pure React application without micro frontends; a React application with a LitElement micro frontend. The sample mean values for the three scenarios are respectively: 93878.90 ms, 5553.73 ms and 2298.37 ms. The SEO analysis showed that page content was invisible to web crawlers who do not posses JavaScript functionality. However, analysis shows that the infrastructural solution remedied this in the desired manner.

VIII. Conclusion

The study described, as requested by delaware, the possible patterns, solutions and approaches for micro frontend architectures and also discussed which technologies can be used for this approach. The PoC illustrated how micro frontends can be used to offer uniform functionalities in a performant manner using the LitElement library. Based on the findings of the research and the methodology used during the PoC, a micro frontend was developed for production purposes that will be demonstrated to various customers. From this it can be concluded that the design of clarifying how this approach can be applied within the consultancy landscape has also been successful.

The underlying question that gave rise to this research, whether or not the use of the micro frontend approach has the desired influence on the pressure of frontend development and the development costs associated with it, cannot be answered in an empirical way by this research. Developing a single micro frontend does not provide enough data for well-founded statistical analysis in this regard. For this future research, the throughput of a number of projects when the micro frontend approach is used could be compared with historical data. A cost and benefit analysis should also be performed to describe the financial impact. Alternatively, instead of using historical data, one could use split testing to eliminate the periodic influences that may be present within historical data.

References

[1] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
[2] ——, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.
[3] S. Millett and N. Tune, *Patterns, principles, and practices of domain-driven design*. John Wiley & Sons, 2015.
[4] J. Lewis and M. Fowler. Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html
[5] C. Richardson, *Microservice Patterns*.
[6] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.
[7] L. Mezzalira, *Building Micro-Frontends*.
[8] M. Geers, *Micro Frontends In Action*.
[9] C. Jackson. Micro frontends. [Online]. Available: https://www.martinfowler.com/articles/micro-frontends.html

FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Micro frontend architectuur voor cross-framework herbruikbaarheid in de praktijk

Tim De Smet
Studentennummer: 01715492

Promotoren: prof. dr. Veerle Ongenae, dhr. Thierry Bruyneel (Delaware)
Begeleiders: dhr. Wim Vergouwe (Delaware), dhr. Luuk Ostendorf (Delaware)

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2019-2020

UNIVERSITEIT
GENT

FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Micro frontend architectuur voor cross-framework herbruikbaarheid in de praktijk

Tim De Smet
Studentennummer: 01715492

Promotoren: prof. dr. Veerle Ongenae, dhr. Thierry Bruyneel (Delaware)
Begeleiders: dhr. Wim Vergouwe (Delaware), dhr. Luuk Ostendorf (Delaware)

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2019-2020

UNIVERSITEIT
GENT

Voorwoord

Deze masterproef is de culminatie van mijn academische carrière. Het proces is een uitzonderlijke leerervaring geweest die me zal bijblijven, niet het minst door de uitzonderlijke omstandigheden van een pandemie waarin die plaats vond. Het uiteindelijke resultaat had niet verwezenlijkt kunnen worden zonder de hulp van verschillende mensen.

Vooreerst wil ik mijn ouders bedanken voor morele en financiële ondersteuning van mijn academisch traject. Het streven naar een ingenieursdiploma na Humane Wetenschappen te volgen is geen klassiek traject. Toch hebben jullie mij steeds jullie vertrouwen geschonken en mij alle kansen gegeven. Dit heeft een resultaat teweeggebracht van onschatbare waarde voor mijn toekomst. Hetgeen ik bereikt heb, heb ik te danken aan jullie.

Mijn promotor prof. dr. Veerle Ongenae wil ik uitdrukkelijk bedanken voor de waardevolle feedback en het in goede banen leiden van het werkstuk. Ook aan mijn promotor Thierry Bruyneel wil ik mijn dank betuigen voor het inspireren van het onderwerp en de uitzonderlijke kans om in de praktijk de masterproef te kunnen uitwerken binnen een ontzettend sterk team. Begeleider Wim Vergouwe verdient een woord van dank voor de kennisoverdracht, het actief meedenken aan de richting van het onderzoek en de mate waarin ik betrokken werd binnen het groter verhaal. Ook aan mijn begeleider Luuk Ostendorf wil ik mijn dank betuigen voor de consequente en uitgebreide feedback die enorm bijgedragen heeft tot het eindresultaat. Tevens heeft u mij steeds voorzien van een transportmogelijkheid naar het kantoor. Dit heeft een unieke ervaring mogelijk gemaakt. Ook wil ik graag Simon Vandeputte bedanken voor meedenken op het technische gebied.

Daarnaast wil ik mijn grootvader en peter, Jean-Pierre De Smet, van een speciaal woord van dank voorzien voor het grondig nalezen van de finale versie van deze masterproef.

Tot slot richt ik mijn oprechte dank aan de mensen die onrechtstreeks een enorme bijdrage hebben geleverd aan deze masterproef door er altijd voor mij te zijn. Mijn grootouders, mijn broer Robbert en mijn vriendin Marie-Céline, ontzettend bedankt.

Tim De Smet, mei 2020

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Woordenlijst

CDN Content Delivery Network. 20

CLI Command Line Interface. 40

CQRS Command-Query Responsibility Segregation. 29

DDD Domain-Driven Design. 3

DOM Document Object Model. 16

ECMAScript European Computer Manufactureres Association Script. 15

ESI Edge Side Includes. 20

HTTP Hypertext Transfer Protocol. 20

iframe Inline Frame. 14

npm Node Package Manager. 13

PoC Proof of Concept. 23

REST Representational State Transfer. 26

SEO Search Engine Optimization. 21

SPA Single Page Application. 11

SSI Server Side Include. 19

SSR Server Side Rendering. 42

XML Extensible Markup Language. 20

Inleiding

Dit onderzoek werd uitgevoerd in opdracht van het globale technologie consultancybedrijf delaware dat werd opgericht in 1981 als onderdeel van Bekaert, Andersen en Deloitte en sinds 2003 als onafhankelijke partnership opereert. delaware is een snelgroeiende, internationale organisatie die geavanceerde oplossingen en services levert aan organisaties die ernaar streven om op een duurzame manier de concurrentie een stap voor te blijven. Ze helpen hun klanten met hun zakelijke transformatie, waarbij de technologie van hun partners SAP, Microsoft en OpenText wordt toegepast. Het bedrijf telt meer dan tweeduizendvierhonderd medewerkers verspreid over twaalf landen en werkt voor klanten zoals Microsoft, Procter & Gamble en Mazda [1].

In het verleden werden er dankzij architecturale ontwikkelingen bij backendsystemen verschillende verbeteringen, onder meer op het vlak van onderhoudbaarheid en onafhankelijkheid, gerealiseerd. Men kan zich afvragen of dit voordelen zijn die ook bij de *frontend* gerealiseerd kunnen worden dankzij een gelijkaardige aanpak. Onder andere door de toenemende trend van platformeconomieën is er steeds meer nood aan een schaalbare ontwikkeling van *frontends*. Aanvullend wordt vastgesteld dat klanten frequent gelijkaardige functionaliteiten wensen te laten implementeren. delaware is op zoek naar een manier om hun klanten op dit vlak tegemoet te komen en zo voor te lopen op deze evolutie en ziet de jonge trend van *micro frontend* architectuur, die voortbouwt op de eerder vermelde architecturale ontwikkelingen bij backendsystemen, als de mogelijke oplossing voor dit vraagstuk. *Micro frontends* zijn een recente evolutie op vlak van *frontend* ontwikkeling. Hoe deze aanpak kan toegepast worden binnen het consultancylandschap is nog onduidelijk. Daarom dient er onderzoek gedaan te worden naar de mogelijkheid om *micro frontends* aan te wenden om op herbruikbare wijze uniforme functionaliteiten aan te bieden. Met als doel de druk op *frontend* ontwikkeling en de ontwikkelingskost die ermee geassocieerd wordt te reduceren.

Van het onderzoek wordt verwacht dat het mogelijke patronen, oplossingen en aanpakken voor *micro frontend* architecturen en de mogelijke technologieën die hiervoor gebruikt kunnen worden beschrijft. Aanvullend dient het ook verduidelijking te bieden over hoe deze aanpak kan toegepast worden binnen het consultancylandschap. Er wordt verwacht dat het werkstuk als onderbouwde aanbeveling fungeert voor deze problematiek.

De studie vertrekt vanuit een literatuuronderzoek over het begrip *micro frontends* dat de aanleiding tot en de context rond *micro frontends* beschrijft. Zo wordt er dieper ingegaan op de begrippen Domain-Driven Design en *microservices*. Nadien wordt het begrip *micro frontend* zelf uiteengezet om dusdanig de nodige basis voor het verdere verloop van het onderzoek te leggen. Aanvullend biedt het een zicht op de kenmerkende eigenschappen van *micro frontends*. Vervolgens wordt er als *state of the art* een uiteenzetting gegeven van courante implementatietechnieken voor *micro frontends* en de theoretische principes waarop ze berusten. Deze uiteenzetting is bedoeld als leidraad tijdens de beslissingsvorming over de optimale aanpak naargelang de probleemstelling van de lezer en onderbouwt de keuze van implementatietechniek die gekozen werd om de problematiek bij delaware van een oplossing te voorzien. Bovendien worden de implicaties van elke implementatietechniek op de kenmerkende eigenschappen van *micro frontends* beschreven. Op basis van de bevindingen die voortkwamen uit het literatuuronderzoek werd een Proof Of Concept uitgewerkt die ter illustratie dient over de manier waarop de *micro frontend* architectuur kan toegepast worden binnen het consultancylandschap. Naast de in het literatuuronderzoek bepaalde kenmerkende eigenschappen van *micro frontends* werd de nadruk gelegd op performantie en Search Engine Optimization gezien de aanpak in de praktijk bruikbaar dient te zijn. Zowel het eindresultaat van de Proof of Concept als de afgelegde weg naar dit eindresultaat worden verduidelijkt, onder meer aan de hand van pseudocode. Nadien volgt een analyse van de Proof Of Concept. Er wordt onder meer beschreven in welke mate de gekozen aanpak voldoet aan de kenmerkende eigenschappen van *micro frontends*. Aanvullend wordt de performantie geanalyseerd door het vergelijken van verschillende aanpakken. Ook wordt het Search Engine Optimization aspect van de Proof of Concept onder de loep genomen. Om af te sluiten worden de conclusies uit zowel het literatuuronderzoek als de Proof of Concept beschreven.

Allereerst wordt een van de fundamentele concepten met betrekking tot microservices, met name Domain-Driven Design (DDD), uiteengezet. Na een vluchtige introductie tot het basisidee worden het waarom achter DDD en enkele fundamentele concepten van DDD behandeld. De eerste bouwsteen voor het micro frontend verhaal wordt in dit hoofdstuk gelegd.

1

Domain-Driven Design

De term, Domain-Driven Design, werd in 2004 geïntroduceerd door Evans [2]. DDD is een aanpak voor softwareontwikkeling die vertrekt vanuit de volgende basisprincipes: gedurende het proces dient de focus op het domein te liggen, het model moet collaboratief ontworpen worden door alle betrokken partijen en deze betrokken partijen dienen te communiceren aan de hand van een gedeeld en domeinspecifiek taalgebruik [3]. De structurering van de software gebeurt rond een objectgeoriënteerd domeinmodel [4]. De DDD-aanpak legt de klemtoon niet op de technologieën die gebruikt kunnen worden om het doel te bereiken maar op de gewenste functionaliteit. De software wordt hier slechts beschouwd als het middel om het doel te bereiken.

De volgende basisconcepten van DDD werden allen gedefinieerd door Evans [3]. Het domein is een geheel van kennis, invloed en activiteit. *Subdomains* zijn het resultaat van de onderverdeling van het domein. Het is het probleemgebied waarvoor software wordt gemaakt. Een model is een systeem van abstracties dat specifieke delen van het domein beschrijft. *Ubiquitous language* is taal gestructureerd rond het domeinmodel en kan gebruikt worden door alle betrokkenen binnen een *bounded context*. Een *bounded context* is de beschrijving van een grens waarbinnen een model gedefinieerd wordt.

Software heeft als hoofddoel problemen op te lossen voor diens eindgebruikers. Een probleem op effectieve wijze kunnen aanpakken vereist kennis over het probleemdomein [2]. Op deze kennis wordt abstractie, een fundamenteel concept binnen computerwetenschappen, toegepast. In essentie is het een techniek waarbij concepten gedistilleerd worden naar meer algemene vormen. Abstractie laat toe softwareobjecten te identificeren waarmee men een model van probleemdomein kan ontwerpen, namelijk het domeinmodel [5].

Software succesvol ontwerpen en opleveren is geen eenvoudige zaak. Millet & Tune [6] beschreven diverse soorten complexiteiten die gepaard gaan met het ontwikkelen van software. Complexiteit kan onder andere voortkomen uit het creëren van code zonder duidelijke structurering of bij gebrek aan gemeenschappelijk taalgebruik. Daarnaast kan het Big Ball of Mud patroon extra complexiteit doorheen de tijd introduceren. Dit patroon heeft namelijk een negatieve impact op de mate waarin code onderhoudbaar en uitbreidbaar is. Foote & Yoder [7] omschrijven het Big Ball of Mud patroon als onbedachtzaam gestructureerde code. Door Millet & Tune [6] wordt vooropgesteld dat de DDD-aanpak hiertoe oplossingen biedt in de vorm van de *strategic* en *tactical patterns*. De *strategic patterns* laten toe de architectuur succesvol te structureren. De *tactical patterns* zijn codepatronen die dienen als bouwblokken voor het effectief modelleren van complexe *bounded contexts*. DDD biedt bijgevolg een framework voor het vormen van ontwerpbeslissingen evenals een technisch vocabulaire om het ontwerp van het domein te bespreken [2].

De tactical patterns die men kan gebruiken om code te structureren is voor het verdere verloop van dit architecturale onderzoek irrelevant. Hoe men de architectuur structureert, door middel van het *bounded contexts strategical pattern*, is fundamenteel voor het volgende hoofdstuk waarin bovendien de relatie tussen DDD en *microservices*/*micro frontends* wordt beschreven.

In het volgende hoofdstuk wordt een uiteenzetting gegeven hoe DDD gerelateerd is aan *microservices* en *micro frontends*.

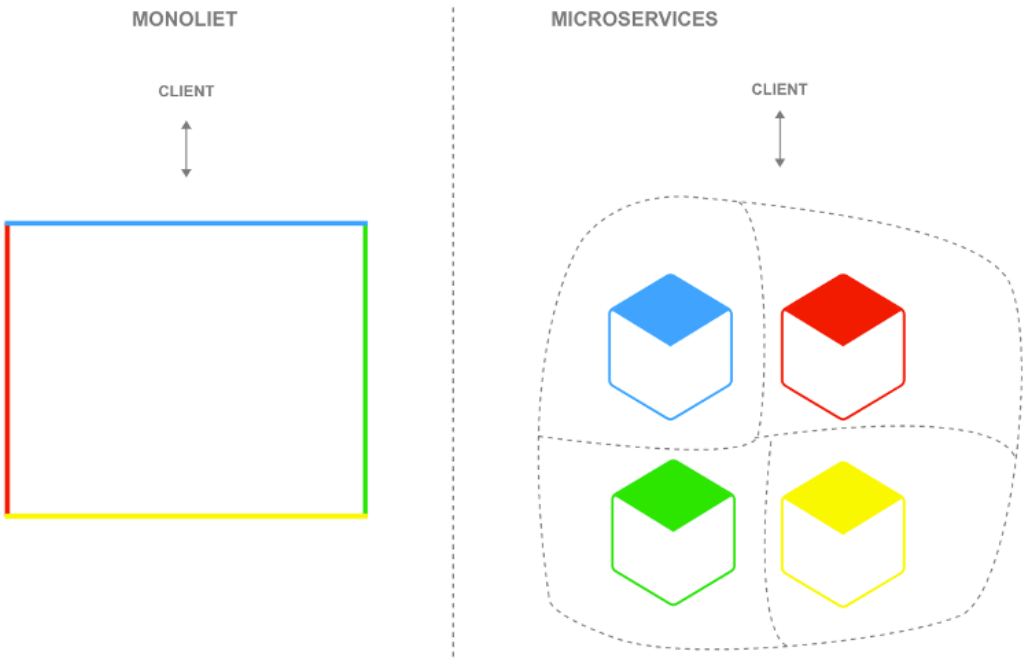
Dit hoofdstuk behandelt het begrip microservices. Opnieuw wordt zowel het begrip zelf als de context waarin het voorkomt geschetst. Hieruit zal blijken dat microservices, zoals men ze de dag van vandaag kent, de drijvende kracht achter het begrip micro frontends zijn. De kenmerkende eigenschappen van microservices worden later in de scriptie gemapt naar het micro frontend concept.

2

Microservices

Het vorige hoofdstuk gaf reeds aan dat het ontwerpen van software geen evidentie is. Daartoe werd in 2001 een leidraad gepubliceerd [8], met name het Agile Manifesto. Het beschrijft een visie op softwareontwikkeling die berust op de volgende vier basisprincipes: individuen en interacties zijn belangrijker dan processen en tools; werkende software is belangrijker dan uitgebreide documentatie; samenwerken met de klant is belangrijker dan contractonderhandelingen; wendbaar zijn is belangrijker dan een plan volgen [8]. Het softwarelandschap ondergaat continu ontwikkelingen en bijgevolg de vereisten voor softwareontwikkelingen ook. Het Reactive Manifesto [9] definieert de term *reactive systems* die als doel dienen nagestreefd te worden. Een systeem is *reactive* wanneer het intern asynchroon communiceert aan de hand van messaging om onder andere te garanderen dat de onderdelen losgekoppeld zijn van elkaar. Aanvullend wordt verwacht dat het systeem steeds tijdig reageert en responsief blijft gedurende fouten en gevarieerde werklast. Dit garandeert dat *reactive systems* onderhoudbaar en uitbreidbaar zijn. De manier waarop code gestructureerd wordt, de architecturale stijl, bepaalt de mate waarin de visies zoals beschreven door het Agile en Reactive Manifesto gevolgd worden.

De oorsprong van de term *microservices* is een conferentie over cloud computing in 2005 waar Peter Rodgers de term beschreef [10]. *Microservices* zijn het resultaat van het toepassen van de *microservice* architecturale stijl. De stijl hanteert de aanpak waarbij men een complexe applicatie opbouwt uit een verzameling van samenwerkende softwarecomponenten, *services*, die elk georganiseerd zijn rond een bedrijfsfunctie, een zogenaamd *business capability*. Bovendien kunnen elk van deze *services* onafhankelijk uitgerold worden [11]. De *microservice* aanpak resulteerde uit de schaalbaarheidsproblemen die zich voordeden bij monolithische architecturen. Bij een monolithische architectuur wordt de applicatie als één allesomvattend geheel uitgevoerd. Deze architecturale stijl is nadelig wanneer de totaliteit van code te groot wordt. De omvang van de code laat dan niet langer toe om als ontwikkelaar het project volledig te verstaan. Ook wordt het ontwikkelingsproces en de *deployment* hierdoor geremd. Echter biedt het voor eenvoudigere problemendomeinen wel de voordelen van eenvoud op gebied van ontwikkelen, testen, *deployment* en schalen [4]. Figuur 2.1 illustreert het conceptuele verschil tussen monolithische en *microservice* architecturen.



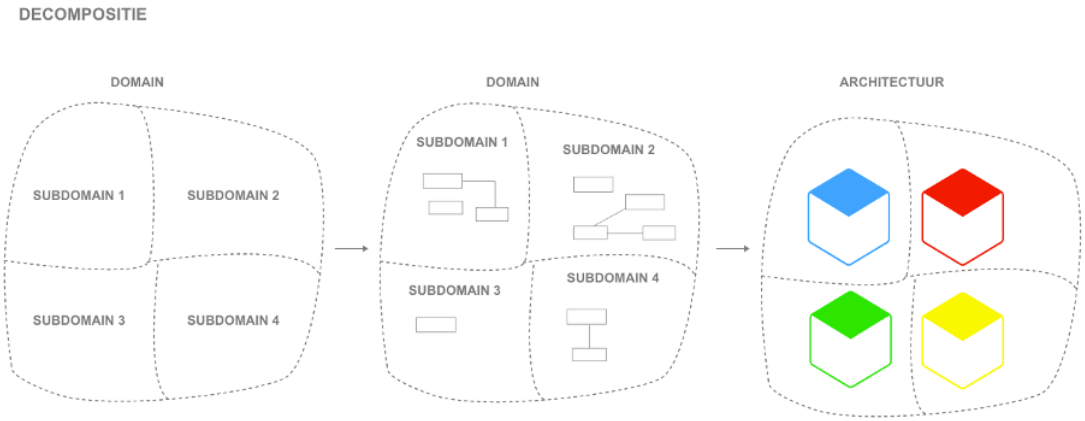
Figuur 2.1: Monolithische architectuur versus *microservice* architectuur

Richardson [4] geeft het woord micro in *microservices* de betekenis dat een service onderhoudbaar moet zijn door een klein team. Dit is zinvoller dan kleine *services* nastreven. Verder beschrijft hij *business capabilities* als een bedrijfsfunctie die waarde genereert.

De *microservice* architecturale stijl volgen levert, zoals beschreven door Richardson [4], de hierna vermelde voordelen op: de stijl laat continue oplevering en *deployment* van grote en complexe applicaties toe; de *services* zijn klein en bijgevolg makkelijker onderhoudbaar, bovendien is elke service, onafhankelijk van alle andere, uitrolbaar, schaalbaar en testbaar. Naast een over het algemeen meer betrouwbare applicatie leidt dit voor teams tot meer autonomie, zowel op vlak van ontwikkeling als technologische keuzes. Gezien het landschap van technologie niet stilstaat is de snelheid waarop men nieuwe technologieën kan adopteren in vergelijking met monolithische architecturen ook een noemenswaardig voordeel. Verder wordt elke service voorzien van een eigen datastorage. Aanvullend bemerkt Newman [12] dat services structureren rond de zogenaamde *business capabilities* het aligneren van de softwarearchitectuur met die van de organisatie ten goede komt.

Newman [12] wijst er echter wel op dat *microservices* toepassen niet voor elke context de correcte aanpak is gezien de additionele complexiteit die men introduceert wanneer gedistribueerde systemen toegepast worden.

Een bijkomende moeilijkheid bij *microservices* is de decompositie van het geheel tot meerdere services. Zoals eerder besproken dient elke service verantwoordelijk te zijn voor een *business capability*. Dit realiseren kan gebeuren aan de hand van verscheidene patronen. Het opdelen aan de hand van de DDD-aanpak is hier een van de meest prevalente patronen voor. Richardson [4] beschrijft dit decompositiepatroon, tevens geïllustreerd door Figuur 2.2, als volgt: per subdomein wordt bij DDD een domeinmodel gemaakt. Subdomeinen worden op dezelfde manier geïdentificeerd als *business capabilities*, de organisatie en haar diverse expertisegebieden worden geanalyseerd. De subdomeinen en hun domeinmodellen vormen de basis voor de *microservices*. Wanneer dit patroon wordt toegepast bepaalt DDD de *microservice* architectuur.



Figuur 2.2: Decompositie aan de hand van Domain-Driven Design

De resulterende verticale opsplitsing in *microservices* met eigen *datastorage* zijn een toepassing van het begrip verticale decompositie. De multidisciplinaire teams die verantwoordelijk zijn voor deze verticale splitsingen doorheen de *backend* hoeven daar echter niet toe beperkt te blijven. De groei aan de *frontend* kant van het softwareverhaal, waar men nu vaak nog geconfronteerd wordt met een monolithische architectuur, zorgt voor gelijkaardige complexiteiten als bij de *backend* ervaren werd.

Het idee van het volledig doortrekken van de verticale splitsing doorheen de applicatie, inclusief de *frontend*, ligt aan de basis voor *micro frontends*. In het volgende hoofdstuk worden het begrip *micro frontends* en de link met *microservices* uiteengezet.

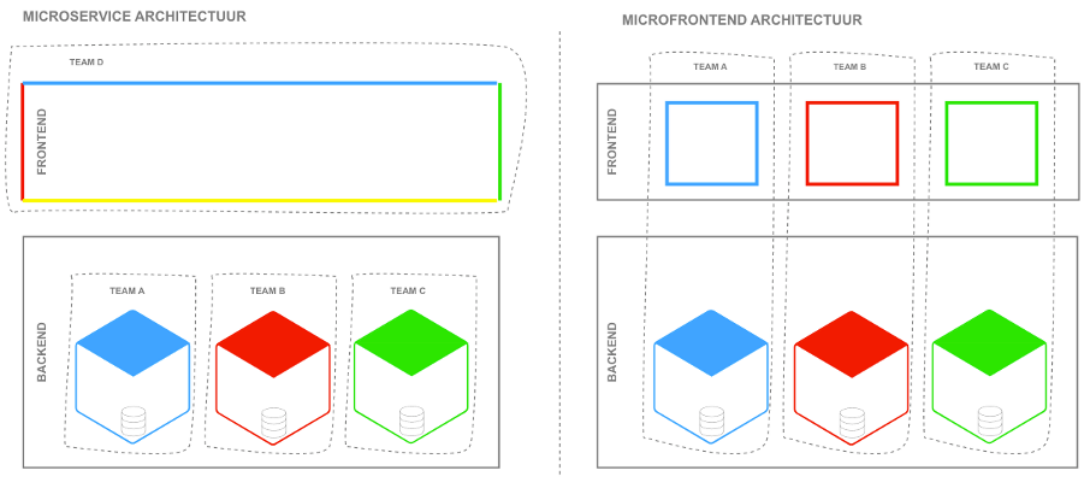
Het vorige hoofdstuk besprak microservices, een architecturale stijl waarbij men verticaal de applicatie splitst. Deze splitsing van een monolithisch systeem, die meestal enkel in de backend wordt toegepast, gebeurt vaak aan de hand van DDD. In dit hoofdstuk wordt de term micro frontends uiteengezet. Hoe deze term de denkwijze bij microservices doorheen de volledige applicatie, inclusief de frontend, doortrekt komt eveneens aan bod.

3

Micro Frontends

De beperkingen die monolithische systemen met zich meebrengen op vlak van schalen zijn ook van kracht op een monolithische *frontend*: het gehele systeem moet bij elke update in zijn geheel geüpdatet worden; bij elke wijziging moet het volledige systeem opnieuw uitgerold worden; werken met grote teams aan één *code base* brengt centralisering van de beslissingsvorming met zich mee; er is een *single point of failure*, waardoor de risico's gerelateerd aan het wijzigen van de *code base* groter zijn [13].

Micro frontends zijn het resultaat van de architecturale stijl van *microservices* waarbij *services* worden uitgebreid met *userinterfaces*. De verticaal georganiseerde teams zijn hierbij ook verantwoordelijk voor de *frontend*. Binnen de organisatie worden de teams georganiseerd rond *business capabilties* en zijn ze bijgevolg volledig interdisciplinair en volledig *end-to-end* [14]. Zie Figuur 3.1 ter illustratie.



Figuur 3.1: *Microservice* architectuur versus *micro frontend* architectuur

Jackson [15] beschrijft de populariteitstoename van *microservices* in de laatste jaren en de beperkingen omtrent een monolithische *frontend*. Zo worden onder meer de moeilijkheden die een bedrijf kan ervaren omtrent het adopteren van nieuwe technologieën besproken evenals de problemen die ondervonden worden bij het schalen van ontwikkeling. Uit het vorige hoofdstuk bleek dat dit probleem zich ook in de *backend* stelde. Ook de voordelen die zich voordoen volgens Jackson, zijn parallel aan die van *microservices*: de individuele delen kunnen incrementeel geüpdatet worden; de *code bases* zijn per definitie eenvoudiger en losgekoppeld van elkaar; ook zijn *micro frontends* onafhankelijk van elkaar uitrolbaar; de *code bases* zijn telkens volledig de eigendom van de betreffende teams. Zodoende kunnen ze zelfstandig van elkaar opereren.

Net zoals bij *microservices* introduceert men ook bij *micro frontends* additionele complexiteit door gedistribueerd te werk te gaan. Daarnaast haalt Geers [14] aanvullend nog andere nadelen aan: de browser kan heel wat redundante code ontvangen; het oplossen van een *library* gerelateerde bug door één team moet door elk team, die de *library* gebruikt, apart uitgevoerd worden; infrastructuur of geautomatiseerde *deployment* verbeteringen die door één team gemaakt worden, moeten door elk team apart uitgevoerd worden. Indien teams binnen een organisatie gestructureerd zijn rond *business capabilities* is er minder flexibiliteit voor het pivoteren van het *businessmodel*. Mezzalira [13] duidt *micro frontends* aan als een gepaste keuze wanneer men in verscheidene iteraties over een langdurige periode wil ontwikkelen. Ook wanneer men legacy code uit een bestaand *frontend* systeem wenst weg te werken of wanneer het ontwikkelaarsteam uit meer dan dertig mensen bestaat worden *micro frontends* als een correcte aanpak geacht.

De term *frontend* omvat elke soort gebruikersomgeving. Echter zijn *native* applicaties steeds monolithisch door hun omgeving. De architecturale stijl van *micro frontends* dient dus voornamelijk voor webtoepassingen. Dit valt echter wel te omzeilen door gebruik te maken van een ingebedde webbrowser [14]. Het begrip webtoepassingen omvat opnieuw verschillende types van toepassingen. De eenvoudigste soort, statische webpagina's, zijn snel en per definitie secure aangezien er geen *backend* systeem is [16]. Dynamische webpagina's laten toe de inhoud aan te passen van de website, dit onder andere door gebruikersinput. Ook voor webapplicaties bestaan verschillende aanpakken voor implementatie. Bij een traditionele webapplicatie ontvangt de webserver een HTTP *request*, de webserver rendert de pagina en stuurt deze als antwoord terug. De browser ververst en geeft de ontvangen webpagina terug. Eén van de mogelijke en meest voorkomende alternatieve aanpakken heeft de naam Single Page Application (SPA). Bij een SPA bestaat de webtoepassing slechts uit één pagina die bij de eerste *request* door de webserver aangeleverd wordt. Vanaf dat punt communiceert de *client* verder met de webserver, echter zonder dat er volledige paginaverversingen uitgevoerd worden [17].

De diversiteit in het web-landschap verandert niets aan het architecturale concept van *micro frontends*, de implementatiemogelijkheden zijn uiteraard wel gebonden aan de context waarin men zich bevindt. Zoals aangegeven door Richards & Ford [18] zijn er verscheidene mogelijkheden voor het ontwikkelen van *micro frontends*.

Zoals reeds eerder vermeld verwacht men van *micro frontends* dat ze net zoals *microservices*: rond *business capabilities* georganiseerd zijn; onafhankelijk van elkaar en op een continue wijze uitgerold kunnen worden; kunnen samenwerken ongeacht technologische implementatiekeuzes; de beschikbaarheid en de mate waarin ze onderhoudbaar zijn bevorderen. Elke implementatietechniek voor *micro frontends* dient bijgevolg ten opzichte van deze gewenste eigenschappen afgetoetst te worden. Verder is de performantie die de mogelijke technieken faciliteren ook een maatstaf voor de geschiktheid ervan. In het volgende hoofdstuk worden de meest courant toegepaste technieken beschreven alsook de theoretische principes waarop ze berusten.

De theoretische basis van micro frontends is reeds gelegd, het vervolg van de scriptie behandelt voornamelijk het praktische luik. Dit hoofdstuk vormt een uiteenzetting van de meest courante reeds geïmplementeerde architecturale en technische strategieën voor het realiseren van micro frontend architectuur zoals beschreven door Jackson [15], Mezzalira [13] en Geers [14]. Deze uiteenzetting is bedoeld als leidraad tijdens de beslissingsvorming over de optimale aanpak naargelang de probleemstelling van de lezer.

4

State of the Art

Het implementeren van de *micro frontend* architectuur wordt reeds in de praktijk op verscheidene manieren uitgevoerd. De hierna vermelde technieken zullen telkens afgetoetst worden op de fundamenten van de *micro frontends* architecturale stijl. De mate waarin *micro frontends* kunnen: georganiseerd worden rond *business capabilities*; onafhankelijk van elkaar en op een continue wijze uitgerold worden; samenwerken ongeacht technologische keuzes rond implementatie; onderhouden worden en beschikbaar, zelfs bij falen. Ook de performantie zal een aandachtspunt zijn bij de bespreking die volgt.

4.1 Hyperlinks tussen Applicaties

Het gebruik van Hyperlinks om samenwerking tussen meerdere verschillende applicaties te verschaffen kan beschouwd worden als een implementatie van het *micro frontend* principe. Echter laat dit niet toe de componenten uit applicaties bij elkaar te integreren.

Business capabilties kunnen door middel van deze aanpak strikt van elkaar gescheiden worden. Echter is er bij de pure vorm van deze aanpak geen mogelijkheid tot overvloeiing tussen *capabilities*. Elke *micro frontend* kan hier apart en op continue wijze uitgerold worden. De teams kunnen bijgevolg volledig zelfstandig van elkaar werken. Gezien de samenwerking bij deze aanpak enkel het navigeren tussen Hyperlinks omvat heeft de technologische keuze omtrent implementatie geen invloed op deze samenwerking. Bovendien kunnen de verschillende *micro frontends* verspreid worden over verschillende webservers. Deze decentralisering kan de onderhoudbaarheid en beschikbaarheid ten goede komen. De pure implementatie van deze techniek voorziet echter geen mechanismen om een gefaalde *micro frontend* op te vangen. Jackson [15] bemerkt wel dat, wanneer men besluit verschillende webservers te gebruiken, een tussenliggende proxyserver noodzakelijk is om navigatielinks uniform te maken om zodoende de gebruikerservaring te optimaliseren. Echter introduceert men hiermee een *single point of failure*, waardoor per definitie de beschikbaarheidsvoordelen ongedaan gemaakt worden indien men hiervoor geen aanvullende voorzieningen opzet. Conceptueel gezien impliceert deze methodiek dat er frequenter volledige pagina's herladen zullen worden wat meer verkeer vergt en bijgevolg nadelig is voor de performantie. Het grootste nadeel is echter dat delen van *micro frontends* niet met elkaar geïntegreerd kunnen worden. Bijgevolg is deze techniek nauwelijks in de praktijk bruikbaar, op de uitzonderingen waarbij men geen integratie wenst na.

4.2 Build-time Integratie

Zoals aangegeven door Jackson [15] kan men elk onderdeel *packagen* en als *dependency* gebruiken in een overkoepelende containerapplicatie. De integratie van de *micro frontends* gebeurt hier alvorens de software in productie gaat.

Op de website van Yarn, een van de grootste *package managers*, wordt aangegeven welke voordelen code *packagen* oplevert: projecten kunnen op een veilige en stabiele manier gereproduceerd worden [19]. Echter zijn er heel wat verschillende package managers die gebruikt kunnen worden afhankelijk van het niveau waarop ze opereren of de taal van het betreffende project. Node Package Manager (npm) is een noemenswaardig alternatief voor de eerder vermelde Yarn.

Het *packagen* van individuele stukken code laat teams toe om onafhankelijk van elkaar en georganiseerd rond *business capabilities* code te ontwikkelen. De *packages* kunnen individueel gepubliceerd worden. Echter vergt het updaten van de *package* in het overkoepelende project dat er gebruik van maakt dat het volledige project opnieuw uitgerold wordt. Continue *deployment* is bij deze methodiek dus niet mogelijk. Van code die *gepackaged* werd verwacht men wel dat deze samenwerkt met de gekozen technologie, de techniek laat bijgevolg geen volledige technologieon-

afhankelijkheid toe. Men is namelijk gebonden aan de taal die overheen de applicatie gebruikt wordt. Wel is de technologiekeuze op *framework* niveau vrij. Onderhoud of uitbreidingen van de ingepakte code resulteert in nieuwe versies van de *package*. Op vlak van beschikbaarheid wordt verondersteld dat *packages* volledig vrij van fouten gepubliceerd worden. Daarna worden ze opgenomen in een overkoepelend project. Echter bestaat de mogelijkheid dat *packages* niet vrij van fouten gepubliceerd worden en zodoende een *single point of failure* introduceren. Ook is het mogelijk dat verschillende *packages* afhankelijk zijn van verschillende versies van eenzelfde *package*. De toegevoegde complexiteit hieromtrent heeft invloed op de onderhoudbaarheid van het project als geheel door de risico's die het introduceert. Deze uitdagingen vergen extra aandacht wanneer men deze integratietechniek wenst toe te passen. Daartoe kan bijvoorbeeld al het mogelijke gedaan worden om het aantal afhankelijkheden van een *package* te minimaliseren. Om performantie te verzekeren is het van belang dat de omvang van elke *package* zo veel mogelijk gereduceerd wordt gezien ook deze code telkens verscheept moet worden van de webserver naar de *client*.

4.3 Run-time Integratie

Een alternatieve aanpak die toelaat om de *micro frontends* zelfstandig te *deployen*, voert de integratie uit tijdens het draaien van de software. Deze aanpak kent opnieuw verschillende varianten waarvan de meest courante in de volgende secties uiteen worden gezet.

4.3.1 Iframes

Het Inline Frame (iframe) HTML-element laat toe een HTML-pagina in de huidige in te vlechten, dit aan de hand van een geneste browser context. Elke iframe is een volwaardige omgeving en verbruikt dus geheugen en andere computatiemiddelen [20].

Media streaminggigant Spotify maakte voor de originele versie van de *web player* gebruik van dit architecturale patroon. Iframes werden gebruikt om isolatie voor de teams te garanderen. Daarenboven werd de code telkens gebruikt voor zowel de desktop als de webtoepassing. De desktop toepassing kon alle benodigdheden downloaden, echter bleken de iframes voor de webtoepassing op termijn geen schaalbare aanpak. In 2016 werd een nieuwe *web player* geschreven als SPA gebruikmakend van een API om toegang te krijgen tot de *microservices* in de *backend* [21]. Verder beschrijft Pérez [21] dat deze overschakeling de performantie bevorderde ten opzichte van de versie waarbij de iframes integratietechniek werd toegepast.

Luigi is een *micro frontend* integratie *framework* dat gebruikmaakt van de iframe aanpak. Elke *micro frontend* wordt weergegeven in een eigen iframe die binnen een *client-side orchestrator* worden weergegeven. Om de bedenkingen omtrent de performantie te adresseren maakt het gebruik van een ladingsmechanisme waarbij het eerder bezochte iframes in de achtergrond bewaart. Zo wordt vermeden dat constant de onderliggende technologieën voor de *micro frontends* moeten ingeladen worden [22].

Volgens de MDN Web Docs [20] is toegang tot inhoud van de iframe aan de hand van scripts onderdanig aan de *same origin policy*. Scripts kunnen de meeste *properties* van andere *window* objecten niet *cross domain* raadplegen. Bovendien wordt vermeld dat *cross origin* communicatie dient geïmplementeerd te worden aan de hand van de postMessage functionaliteit.

Indien men zelfstandig de iframe integratie wil implementeren maar enkel voor communicatie beroep wenst te doen op een *library* kan men hiervoor Postmate gebruiken. Postmate is een communicatie *library* voor iframes geïmplementeerd door het Amerikaanse bedrijf Dollar Shave Club.

Het gebruik van iframes heeft geen implicaties op de mogelijkheid om de *micro frontends* te organiseren rond *business capabilities*. De geneste browser contexten die zich binnen iframes zullen begeven kunnen onafhankelijk van elkaar uitgerold worden. Bovendien kan door de scheiding van deze contexten een vrije keuze gemaakt worden in de keuze omtrent implementatietechnologieën. Zoals eerder aangegeven is, door de communicatierestricties tussen iframes, samenwerking minder eenvoudig maar niet onoverkomelijk. Door de strikte scheiding zijn de *micro frontends* onafhankelijk van elkaar onderhoudbaar en hebben ze geen invloed op elkaars beschikbaarheid. Er is echter geen mogelijkheid om een gefaalde browser context binnen een iframe te detecteren gezien hier geen *events* voor voorzien zijn. Bovendien zijn er de eerder aangehaalde performantie implicaties waarmee rekening gehouden moeten worden.

4.3.2 JavaScript

Bij de meest algemene methode voor *run-time* integratie worden de gewenste *micro frontends* ingeladen door middel van JavaScript.

JavaScript is een scripttaal die niet gecompileerd maar wel geïnterpreteerd wordt [23]. Het is steeds een implementatie van een bepaalde versie van de European Computer Manufactureres Association Script (ECMAScript) standaard. De versie van ECMAScript bepaalt ondermeer de te benutten types, objecten, eigenschappen en functies [24].

De taal biedt object georiënteerde functionaliteiten aan en is *loosely typed*. Javascript wordt voornamelijk gebruikt in webbrowsers om interactiviteit te voorzien op webpagina's en om op dynamische wijze de inhoud van webpagina's te manipuleren [23].

De integratie van de verschillende *micro frontends* gebeurt bij deze methodiek dus op de *client* door aan de hand van JavaScript het Document Object Model (DOM) dynamisch te manipuleren. Het DOM is een API die een mapping van de webpagina naar een boomstructuur voorstelt. Bijgevolg kan deze voorstelling van de webpagina programmatorisch gemanipuleerd worden [24].

De JavaScript aanpak omvat een ruim landschap aan mogelijke oplossingen en kan aan de hand van verschillende patronen toegepast worden. Echter blijft het onderliggende concept steeds hetzelfde. De mogelijkheden hangen samen met de gebruikte JavaScript versie. Zo is bijvoorbeeld een implementatie van ECMAScript 6 vereist voor het gebruik van *template literals*. Wel kan *backward compatibility* verschaft worden door beroep te doen op een *transpiler*, bijvoorbeeld Babel.

Opnieuw heeft de integratietechniek geen impact op de mate waarin de *micro frontends* georganiseerd kunnen worden rond *business capabilities*. Gezien bij deze techniek van de teams verwacht wordt dat ze een script afleveren kunnen teams onafhankelijk van elkaar en op continue wijze deze scripts *deployen*. Zolang het eindresultaat van de *micro frontend* een JavaScript module is zijn teams volledig vrij een eigen *technology stack* te bepalen. Gezien alle JavaScript *micro frontends* in het DOM terechtkomen kan het DOM als klankbord voor *events* gebruikt worden om communicatie tussen de *micro frontends* te realiseren. Alle *micro frontends* kunnen namelijk het document raadplegen. Bijgevolg kunnen *micro frontends events* afvuren naar het document en kunnen andere *micro frontends event listeners* via het document definiëren. Indien er duidelijke afspraken zijn rond hoe deze messages heten en wat hun verwachte inhoud is kunnen teams onafhankelijk van elkaar de nodige functionaliteiten implementeren bij hun *micro frontend* wat het samenwerken van *micro frontends* mogelijk maakt. Gezien de teams afzonderlijk van elkaar kunnen werken komt het onderhoud niet in het gedrang. Men kan JavaScript gebruiken om de situaties op te vangen waarbij *micro frontend* niet beschikbaar is. Het gros van het werk wordt uitgevoerd door de *client*, dus performantie bij schaalbaarheid is geen pijnpunt bij deze methodiek voor integratie.

4.3.3 Web Components

Jackson [15] beschrijft integratie aan de hand van Web Components als gelijkaardig aan de JavaScript aanpak, met het verschil dat er een component gebaseerde en bijgevolg meer gestructureerde aanpak gehandhaafd wordt. Deze aanpak is onder meer interessant voor ontwikkelaars die vertrouwd zijn met het principe van componenten bijvoorbeeld door ervaring met *frameworks* zoals Angular of *libraries* zoals React. React is een component gebaseerde JavaScript *library* die ontwikkeld werd door Facebook. Er wordt gebruikgemaakt van het Virtual DOM om performantie te bevorderen. De Virtual DOM is een representatie van het DOM die in het geheugen bewaard wordt. Deze wordt bij wijzigingen vergeleken, zodoende kan men garanderen dat enkel de gewijzigde zaken opnieuw geladen worden [25].

MDN Web Docs [26] beschrijft Web Components als een verzameling van drie technologieën die het mogelijk maken om herbruikbare zelf gedefinieerde elementen te gebruiken. De eerste technologie is Custom Elements, wat een verzameling is van JavaScript API's die het mogelijk maken om zelf herbruikbare, ingekapselde elementen te definiëren. De naam van een Custom Element dient steeds een koppelteken te bevatten. De tweede technologie, HTML-templates, laat toe herbruikbare HTML-sjablonen te schrijven die niet getoond worden maar de basis vormen voor de zelf gedefinieerde elementen.

De laatste technologie, het Shadow DOM, vergt meer uitleg. Een Shadow DOM is een *subtree* van het DOM waarvan alle kindelementen afgeschermd zijn van het DOM. De HTML en CSS zijn in beide richtingen geïsoleerd van elkaar. De elementen in een Shadow DOM worden niet beïnvloed door deze in het DOM en vice versa. Shadow DOM ondersteunt ook compositie aan de hand van *slots*. *Slots* laten toe *placeholders* in te brengen in de op maat gemaakte componenten die achteraf zelf kunnen ingevuld worden. De elementen die meegegeven worden door de gebruiker van het component in kwestie om de *slots* te vullen vormen samen de Light DOM. Deze *slots* kunnen *default* inhoud krijgen voor het geval waarin gebruikers deze *placeholders* niet invullen [27]. Figuur 4.1 illustreert dit aan de hand van een vereenvoudigd voorbeeld.

```
// Aanmaken Web Component met slot
customElements.define‘(example-’element, class extends HTMLElement {
    render() {
        this.innerHTML = `
        <slot name’=’example>
            <h1>Dit is de default tekst die getoond zou worden
                wanneer niets wordt meegegeven in de Light DOM
            </h1>
        </slot>`
    }
})

    …

// Gebruik van Web Component met slot
<example-element>
<h1 slot"="example>Deze tekst zal verschijnen in Shadow DOM</h1>
</example-element>

```

Figuur 4.1: *Slots* gebruiken bij Web Components

Het is mogelijk dat in bepaalde probleemstellingen men de op maat gemaakte component niet wil afzonderen door het gebruik van Shadow DOM. Men kan opteren om het gebruik van Shadow DOM achterwege te laten, dit kan onder meer gewenst zijn indien men wil dat de algemene CSS ook van kracht is binnenin de op maat gemaakte component.

Parallel aan de JavaScript integratietechniek wordt opnieuw van de teams verwacht dat ze een script afleveren. Teams kunnen dus ook bij deze aanpak onafhankelijk van elkaar en op continue wijze deze scripts *deployen*. Gelijkaardig aan de JavaScript integratiemethodiek zijn er geen implicaties voor de mate waarin *micro frontends* georganiseerd kunnen worden rond *business capabilities*. Ook de onderlinge samenwerking tussen de *micro frontends* wordt op dezelfde manier gerealiseerd. Opnieuw verwacht men als eindresultaat van de *micro frontend* een JavaScript module. Web Components vereisen echter geen aanvullende *frameworks* om ze te realiseren, al is de keuze hiertoe vrij voor elk team. Gezien de teams afzonderlijk van elkaar kunnen werken komt het onderhoud niet in het gedrang. Ook bij deze aanpak kan beroep gedaan worden op JavaScript om de situaties op te vangen waarbij *micro frontend* niet beschikbaar is. Gelijkaardig aan de pure JavaScript aanpak wordt het meeste werk uitgevoerd op de *client*, de performantie bij schaalbaarheid is ook hier geen knelpunt bij schaling.

4.4 Server Side Integratie

Bij *server side* integratie worden de *micro frontends* met elkaar geïntegreerd door de webserver vooraleer het antwoord de webserver verlaat. Deze injectie kan op verschillende manieren geïmplementeerd worden, maar het achterliggende principe blijft ongewijzigd. Om dit te realiseren kan men een beroep doen op Server Side Include (SSI) opdrachten. SSI's zijn opdrachten die men in HTML-documenten kan plaatsen om inhoud toe te voegen aan het document [28]. Figuur 4.2 illustreert de werking hiervan aan de hand van pseudocode gebaseerd op code van Apache [28].

```
<!DOCTYPE html>
<html lang="en">
<head>
    ...
</head>
<body>
    <!--#include file="voorbeeld.html" -->
</body>
</html>

```

Figuur 4.2: Voorbeeld code van Server Side Include *statement*

Alternatief zijn er ook verscheidene *frameworks* en *libraries* die hetzelfde realiseren. Podium is een *library* voor *server side* integratie van *micro frontends* die ontwikkeld werd door de Noorse advertentiesite Finn.no. Conceptueel wordt er gebruikgemaakt van twee types webservers. Het eerste type is verantwoordelijk voor het beschikbaar stellen van HTML-fragmenten. Het tweede type webserver voert de *server side* integratie uit van de gewenste HTML-fragmenten die beschikbaar gesteld worden [29].

Server side integratie heeft geen invloed op het organiseren van *micro frontends* rond *business capabilities*. Bovendien kunnen alle *micro frontends* onafhankelijk van elkaar en op een continue wijze uitgerold worden, wat ook de onderhoudbaarheid en beschikbaarheid ten goede komt. Gezien het resultaat van deze integratie als één geheel zal teruggegeven worden aan de *client* zijn er geen restricties op het samenwerken tussen deze elementen. Uiteraard moet de code als geheel samengebracht worden. De gekozen technologieën moeten wel samenwerken. Bijgevolg is er geen volledige technologieonafhankelijkheid. Verder zijn er performantie implicaties verbonden met het integreren aan de kant van de server gezien dit bijkomend werk extra belasting betekent voor de webserver.

4.5 Edge Side Integratie

Edge Side Includes (ESI) is een *markup*-taal gebaseerd op Extensible Markup Language (XML). Gebruikmakend van ESI kan men *resources* samenbrengen in Hypertext Transfer Protocol (HTTP) *clients* op de rand van het netwerk. De taal wordt geïnterpreteerd alvorens de pagina aan de *client* wordt afgeleverd. Hiervoor kan men een beroep doen op Content Delivery Networks [30]. Een Content Delivery Network (CDN) is een verzameling van geografisch gedistribueerde servers die informatie verspreidt op een efficiënte manier. De originele informatie wordt gekopieerd naar verschillende locaties waar deze raadpleegbaar is. Daardoor moeten alle *requests* van eindgebruikers niet langer afgehandeld worden door de oorsprong van de informatie. Bovendien hoeft men niet, door de geografische distributie, de volledige afstand naar de oorsprong af te leggen. Men kan gebruikmaken van deze geografische optimalisatie om de wachttijd te reduceren [31]. Naast het invoegen van inhoud dat over een netwerk gevraagd wordt beschrijven Tsimelzon et al. [30] een aantal aanvullende functionaliteiten: ESI ondersteunt variabelen; er kan conditionele logica toegepast worden om de inhoud te verwerken; naast foutafhandeling kan men ook *defaults* declareren. Figuur 4.3 illustreert de werking hiervan aan de hand van pseudocode gebaseerd op de ESI specificatie [30].

```
<html lang="en">
<head>
    ...
</head>
<body>
    <esi:try>
        <esi:attempt>
            <esi:include src="/voorbeeld.html" />
        </esi:attempt>
        <esi:except>
            <esi: include src="/fout-pagina.html">
        </esi:except>
    </esi:try>
</body>
</html>

```

Figuur 4.3: Edge Side Include *statement* met conditionele logica en foutafhandeling

Ook *edge side* integratie heeft geen invloed op het organiseren van *micro frontends* rond *business capabilities*. Net zoals bij *server side* integratie kunnen alle *micro frontends* onafhankelijk van elkaar en op een continue wijze uitgerold worden, wat zoals eerder aangegeven de onderhoudbaarheid en beschikbaarheid bevordert. Opnieuw wordt het resultaat van de integratie als één geheel teruggegeven aan de *client* dus zijn er ook bij deze aanpak geen restricties op het samenwerken tussen de *micro frontends*. Hetgeen gold voor *server side* integratie is ook hier weer van toepassing: de code moet als geheel samengebracht worden, de gekozen technologieën moeten bijgevolg kunnen samenwerken. Ook bij deze integratiemethode is er geen volledige technologieonafhankelijkheid. Er moet opnieuw extra werk geleverd worden dat niet zal plaatsvinden op de *client* maar bijvoorbeeld op de CDN. Bij deze aanpak wordt de last op de webserver echter gereduceerd.

4.6 Overzicht

De hoeveelheid implementatiemogelijkheden en *frameworks* waarmee *micro frontends* kunnen gerealiseerd worden maakt dat de keuze die dient gemaakt te worden, geen eenvoudige is. Tabel 4.1 geeft een samenvattend overzicht van eerder besproken mogelijkheden. Echter zijn er naast de fundamenten van *micro frontends* nog verscheidene andere factoren die naargelang de probleemstelling van de lezer al dan niet kunnen doorwegen, bijvoorbeeld: Search Engine Optimization (SEO), de mate waarin de gekozen oplossing toekomstbestendig is, *browser compatibility*, etc.

Wanneer geopteerd wordt om een *framework* te gebruiken dient men, net zoals voor elke ander type *framework* rekening te houden met onder andere de volgende punten; hoe groot de leercurve ervan is; de omvang van de *community*; de kwaliteit van de documentatie; de licenties omtrent het gebruik in bepaalde contexten.

	4.1 Hyperlinks	4.2 Build-time Integratie	4.3.1 Iframes	4.3.2 JavaScript	4.3.3 Web Components	4.4 Server Side Integratie	4.5 Edge Side Integratie
Organisatie rond business capabilities	✓	✓	✓	✓	✓	✓	✓
Onafhankelijke deployment	✓	✓	✓	✓	✓	✓	✓
Continue deployment	✓	X	✓	✓	✓	✓	✓
Onderlinge samenwerking	±	✓	±	✓	✓	✓	✓
Vrije technologiekeuze	✓	±	✓	±	±	±	±
Onderhoudbaar	✓	±	✓	✓	✓	✓	✓
Beschikbaarheid	✓	±	±	✓	✓	✓	✓
Performantie	±	✓	±	✓	✓	±	✓

Tabel 4.1: Overzicht integratietechnieken

Dit hoofdstuk beschrijft de probleemstelling die speelt bij delaware. De aanpak die gekozen werd op basis van de beschrijvingen uit het vorige hoofdstuk, state of the art, wordt beschreven en gemotiveerd. Tot slot bevat dit hoofdstuk het volledige technische verslag van de opgeleverde Proof of Concept (PoC).

5

Proof of Concept

5.1 Probleemstelling

Bij delaware wordt vastgesteld dat klanten frequent gelijkaardige functionaliteiten wensen te laten implementeren. Daartoe wil delaware een aantal uniforme functionaliteiten aanbieden die kunnen ingeplugd worden in de bestaande sites van de klanten. De herbruikbaarheid van deze oplossingen zou de kost alsook de opleveringstijd reduceren. Gezien klanten mogelijks telkens verschillende technologieën gebruiken, moeten de opgeleverde stukken code kunnen samenwerken met eender welke webtechnologieën.

Als mogelijke oplossing voor dit probleem overweegt delaware deze herbruikbare componenten te ontwikkelen volgens de *micro frontend* architectuur. De onafhankelijkheid van *micro frontends* op gebied van ontwikkeling, *deployment* en technologieën zijn alvast interessante gegevens gezien de probleemstelling. Echter heerst er nog onduidelijkheid over hoe deze aanpak kan toegepast worden binnen het consultancylandschap gezien *micro frontend* architectuur toegepast wordt om de interne ontwikkeling te optimaliseren bij bedrijven. Vooral in de initiële fases zijn technologieonafhankelijkheid, performantie en SEO belangrijke aspecten.

Van de thesis wordt verwacht dat het werkstuk als onderbouwde aanbeveling fungeert voor deze problematiek. De PoC dient ter illustratie van de gekozen aanpak. De onderbouwing van de aanbevelingen en vaststellingen moest voldoende diepgaand zijn gezien deze onmiddellijk in de praktijk voor klanten zou geïmplementeerd worden.

5.2 Aanpak

Simultaan met het literatuuronderzoek werd de *backend* voor de PoC ontworpen en geïmplementeerd. Gezien de implementatie reeds van start ging tijdens het onderzoek was flexibiliteit van de *backend* uiterst belangrijk. Voor de PoC werden vier Java Spring Boot *microservices* opgezet, waarvan drie voorzien werden met een eigen database. De keuze werd genomen om een relationele, een document gebaseerde en een *timeseries datastore* te gebruiken.

De opmerking dient gemaakt te worden dat de *backend* als apart geheel beschouwen niet strookt met de verticale decompositie van de totale applicatie. Er wordt namelijk een horizontale grens getrokken tussen de *frontend* en de *backend*. Deze keuze is echter bewust gemaakt. De PoC is namelijk specifiek voor het consultancylandschap waarbij het team consultants als afzonderlijk team *micro frontends* zal aanleveren. Deze uitgebreide en achteraf gezien te ruime opzetting verschafte de nodige flexibiliteit. Deze opzetting bood een ruim aanbod aan functionaliteiten waardoor het nodige voor de PoC met zekerheid a priori kon verschaft worden.

Deze flexibiliteit garanderen was noodzakelijk gezien het onderzoek zich situeerde binnen een bedrijfscontext waar *requirements* doorheen de tijd konden veranderen naargelang de bevindingen die voortkwamen uit het literatuuronderzoek.

Nadien werd een monolithische React applicatie geïmplementeerd waarvan de aangeboden functionaliteit in de daaropvolgende fases geïsoleerd zou worden als *micro frontend*. Dit voornamelijk als referentiepunt voor analyse doeleinden van dit onderzoek. De *micro frontend* zélf werd gebouwd aan de hand van de Web Components integratiemethode. Een tweede versie werd gebouwd aan de hand van LitElement, een *light-weight library* die een klasse verschaft voor het creëren van Web Components waarover het technisch verslag meer uitleg biedt. Hiertoe waren verschillende redenen, maar de voornaamste was performantie bij veranderingen. De boost in performantie is voornamelijk te wijten aan een zoekmechanisme voor de veranderende zaken die gelijkaardig werkt aan zoeken in een hashmap.

Om de *micro frontend* te distribueren werd deze gepackaged en gepubliceerd als npm *package*. Nadien kon de *micro frontend* zonder problemen gebruikt worden in andere projecten. Deze technologieonafhankelijkheid heb ik gedemonstreerd door deze zowel in een React als in een Angular applicatie te gebruiken.

Aan de SEO-*requirement* voldoen vergde een losstaande infrastructurele oplossing. Er werd een proxyserver voorzien die de nodige vertaalslag kon bieden om de *ranking* ten goede te komen. Opnieuw wordt hier dieper op in gegaan in het technisch verslag.

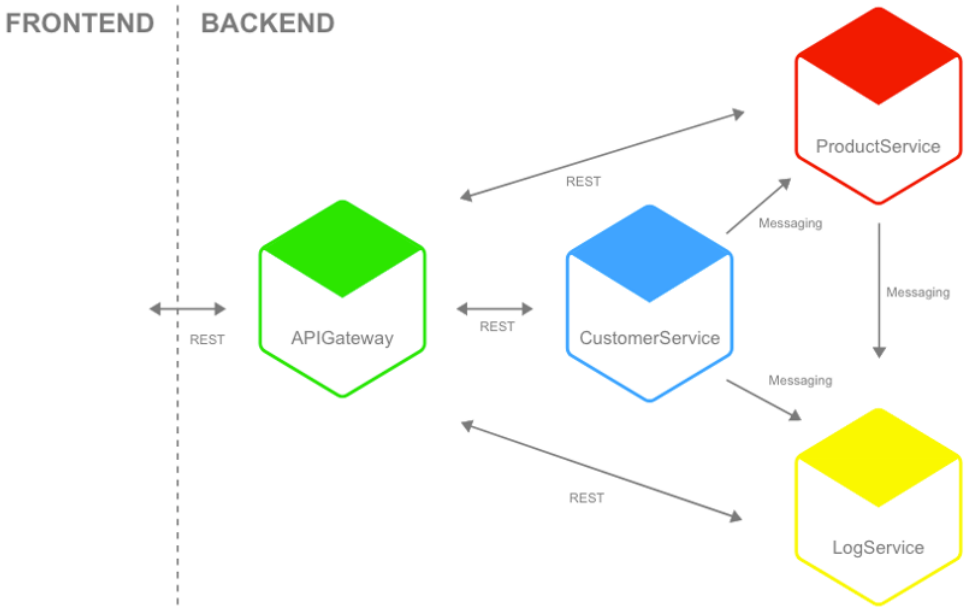
5.3 Technisch Verslag

Deze sectie beschrijft de technische details van de gevolgde aanpak zoals die in sectie 5.2 werd beschreven.

5.3.1 Structuur van de Backend

Om de gedachtegang die achter *micro frontends* schuilt toe te passen op het volledige project worden er vier *microservices* opgezet. Elke *microservice*, uitgezonderd één die een ondersteunende functie heeft, is een minimale implementatie van een *business capability*: het klantenbestand bewaren; de producten beheren; raadplegen van uitgevoerde gebruikershandelingen. Respectievelijk zijn hiervoor de CustomerService, de ProductService en de LogService verantwoordelijk. De vierde *microservice*, de APIGateway service, fungeert als enig aanspreekpunt voor elke mogelijke *client* en zal alle aanvragen naar de gewenste achterliggende service doorsturen.

De vier *microservices* zijn geïmplementeerd als alleenstaande applicaties in het Java *framework* Spring. Spring Boot, wat een extensie is van het Spring *framework*, werd gebruikt om dit proces te vereenvoudigen. Figuur 5.1 illustreert de overkoepelende structuur van de *backend*.



Figuur 5.1: Overkoepelende architectuur van *backend*

De *backend* kan aangesproken worden over HTTP volgens de Representational State Transfer (REST) architectuur. Om de APIGateway in staat te stellen de *requests* door te sturen naar de gewenste services werd de Spring Cloud Gateway *library* gebruikt. Deze gebruikt routes om te bepalen welke service de *request* moet ontvangen. Daartoe werden de Bean uit Figuur 5.2 toegevoegd aan de applicatie.

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder){
    return builder.routes()
    .route(r->r.host("*").and().path("/customers/**").uri("http://customer:2000"))
    .route(r->r.host("*").and().path("/products/**").uri("http://product:2001"))
    .route(r->r.host("*").and().path("/logs/**").uri("http://log-server:2002"))
    .build();
}

```

Figuur 5.2: Verantwoordelijke Bean voor de routering van *requests*

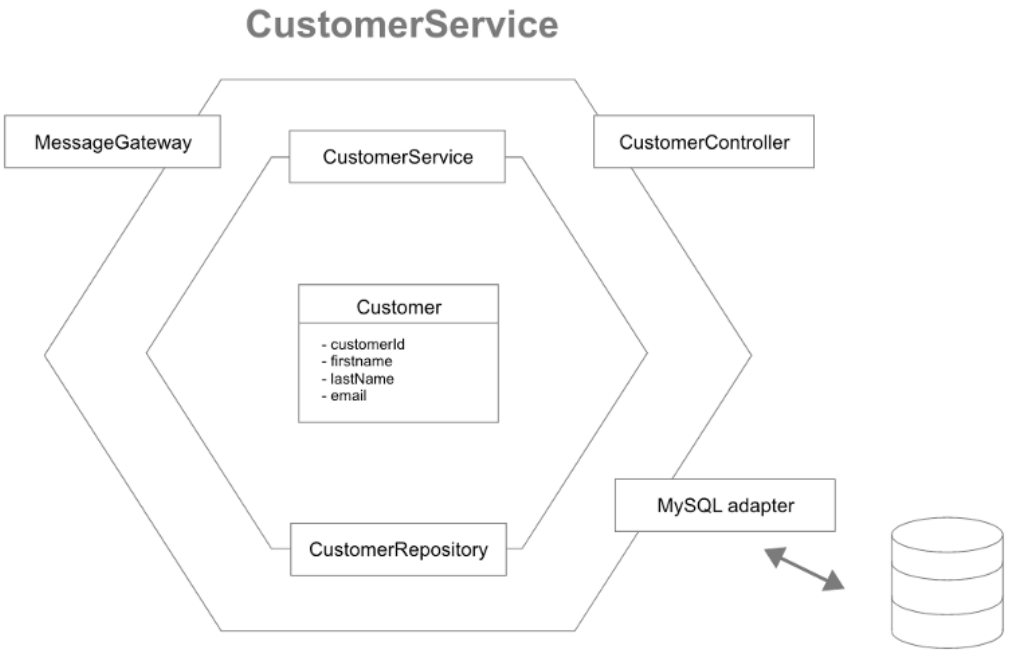
Alle communicatie die zich voordoet tussen de drie andere services gebeurt aan de hand van asynchrone *messaging*. Om dit te faciliteren werd er gebruikgemaakt van het gedistribueerde streamingsplatform Apache Kafka. Bovendien wordt er gebruik gemaakt van Apache Zookeeper die als gecentraliseerde service onder meer de Kafka *topics* bijhoudt en zodoende voor een consensus zorgt over alle andere services heen.

De CustomerService zal *messages* publiceren naar een Kafka *topic* wanneer een gebruiker wordt verwijderd. De ProductService is geabonneerd op diezelfde *topic* en zal alle producten die bij de verwijderde gebruiker hoorden eveneens verwijderen om zodoende dataconsistentie overheen de *microservices* te garanderen. De LogService is eveneens geabonneerd op deze *topic* en zal het verwijderen van de gebruiker opnemen in de logs. Verder publiceert de ProductService voor elke creatie en verwijdering van producten ook *messages* naar een andere *topic*. Ook op deze *topic* is de LogService geabonneerd. Opnieuw worden deze gebeurtenissen vastgelegd in de logs.

In sectie 5.2 werd de opmerking reeds aangehaald dat de *backend* als apart geheel beschouwen niet strookt met de verticale decompositie van de applicatie. Er wordt namelijk een horizontale grens getrokken tussen de *frontend* en de *backend*. Gezien delaware een methodiek zoekt voor *micro frontends* die in elk systeem kan ingeweven worden moet de aanpak onafhankelijk zijn van de *backend*. Klanten kunnen zo bijvoorbeeld al een eigen *backend* hebben. Bovendien was de *backend* in deze vorm uiterst flexibel voor verdere ontwikkeling. Deze flexibiliteit garanderen was noodzakelijk gezien het onderzoek zich situeerde binnen een bedrijfscontext waar *requirements* mogelijks doorheen de tijd veranderen. De uitgebreide opzetting van de *backend* bood een ruim aanbod aan functionaliteiten waardoor het nodige voor de PoC met zekerheid a priori kon verschaft worden. *Microservices* reflecteren de opsplitsing rond *business capabilities* zoals ze idealiter overheen zowel de *frontend* als de *backend* zou gebeuren. Daarom werd er gekozen voor een minimalistische *microservice* architectuur in plaats van een monolithische architectuur.

De CustomerService, ProductService en LogService zijn allen geïmplementeerd volgens de hexagonale architecturale stijl. Deze stijl houdt in dat de kern van de applicatie, de bedrijfslogica, afgeschermd dient te worden om technologieonafhankelijkheid te garanderen. Zodoende is de *business* logica onafhankelijk van technologie specifieke keuzes. Daartoe worden er conceptueel poorten voorzien die de bedrijfslogica aanspreekbaar maken voor technologie specifieke adapters. Deze adapters zijn door deze ingreep ontkoppeld van de kern van de applicatie, wat het onder meer eenvoudiger maakt deze later uit te wisselen voor nieuwere technologieën [32].

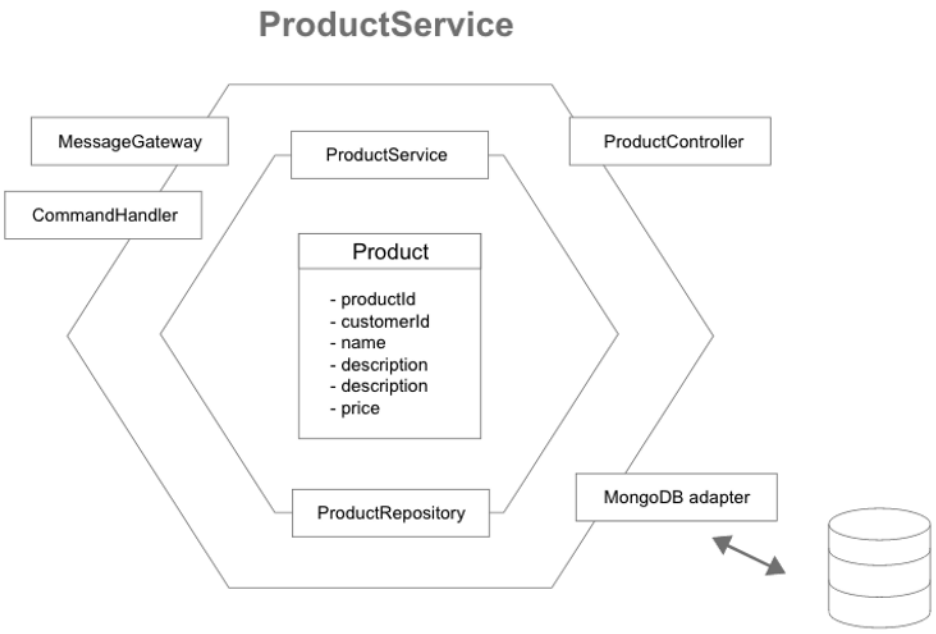
De CustomerService staat in voor het beheer van het fictieve klantenbestand. Figuur 5.3 geeft de structuur van de CustomerService terug. Het domeinmodel van deze *service* heeft slechts één klasse, namelijk de klasse Customer die de klant voorstelt. De domeinlogica is gestructureerd volgens het Transaction Script Pattern. Bij dit patroon worden de klassen die domeinlogica implementeren gescheiden van het domeinmodel. In de *microservice* is de CustomerService klasse hiervoor verantwoordelijk. De technologie specifieke adapterklassen kunnen via deze klasse de domeinlogica benutten. De CustomerController is het aanspreekpunt voor de *restful* communicatie. Via de MessageGateway kan de *service messages* publiceren naar de Kafka *message broker*. Tot slot is de *service* uitgerust met de CustomerRepository *interface* waarvan Spring gebruikmaakt om de MySQL adapter te voorzien. De *service* benut een relationele MySQL databank. Desondanks de data waarmee gewerkt wordt volledig fictief zijn, werd gekozen voor een relationele databank gezien deze fictieve data een rigide structuur heeft.



Figuur 5.3: Hexagonale architectuur van de CustomerService

De ProductService, waarvan Figuur 5.4 de structuur weergeeft, staat in voor het beheer van de fictieve producten die door gebruikers aangemaakt kunnen worden. Voor elk product worden de volgende zaken bijgehouden: een id; het id van de gebruiker die het product heeft aangemaakt; de naam, een beschrijving en de prijs van het product.

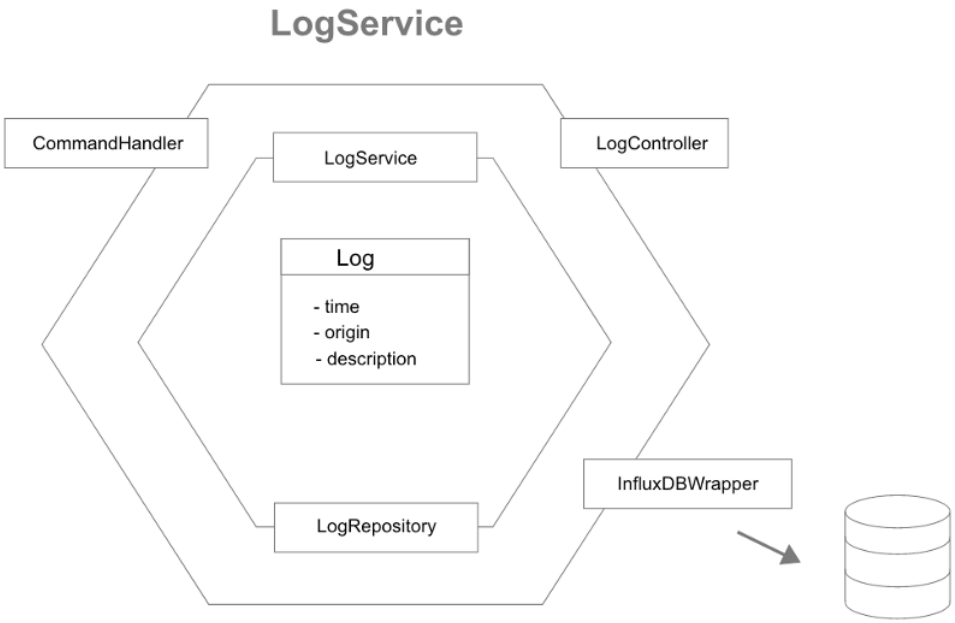
Parallel aan de CustomerService *microservice* bevindt de domeinlogica zich in een serviceklasse, de ProductService klasse. De ProductController is het aanspreekpunt voor de *restful* communicatie. De CommandHandler zal reageren op asynchrone *messages* die de CustomerService uitstuurt naar de Kafka *message broker* wanneer een klant verwijderd wordt. De *microservice* kan zelf ook opnieuw *messages* publiceren door middel van de MessageGateway. Spring maakt gebruik van de ProductRepository *interface* om de MongoDB adapter te voorzien. Bij de ProductService viel de keuze op een NoSQL database met name de document-georiënteerde database MongoDB. In de realiteit zouden verschillende soorten producten verschillende eigenschappen hebben. Een document-georiënteerde database is bijgevolg de logische keuze dankzij de *schema on read* eigenschap die het bezit.



Figuur 5.4: Hexagonale architectuur van de ProductService

De laatste *microservice*, de LogService, heeft als enige functie het luisteren naar allerlei *messages* om op basis daarvan logs bij te houden van de gebeurtenissen die zich hebben voorgedaan. Het Command-Query Responsibility Segregation (CQRS) patroon werd toegepast om van deze *microservice* een *query-only service* te maken.

Figuur 5.5 toont de hexagonale architectuur van de LogService. Deze *microservice* doet beroep op InfluxDB, een *timeseries* database, om de logs te persisteren. De connectie met influx is bij deze *service* niet voorzien door Spring zelf. De InfluxDBWrapper klasse voorziet hiervoor het nodige. De logRepository klasse benut de InfluxDBWrapper om de database te gebruiken. Voor elke log wordt de tijd van creatie, de oorsprong en een beschrijving opgeslagen. Dankzij de CommandHandler is de LogService in staat gepast te reageren en logs aan te maken wanneer asynchrone *messages* ontvangen worden. De domeinlogica die zich in de LogService bevindt wordt door de CommanHandler benut. De *service* is eveneens voorzien van een aanspreekpunt voor *restful* communicatie, de LogController, om de logs te kunnen raadplegen.



Figuur 5.5: Hexagonale architectuur van de LogService

5.3.2 React Applicatie ter Referentie

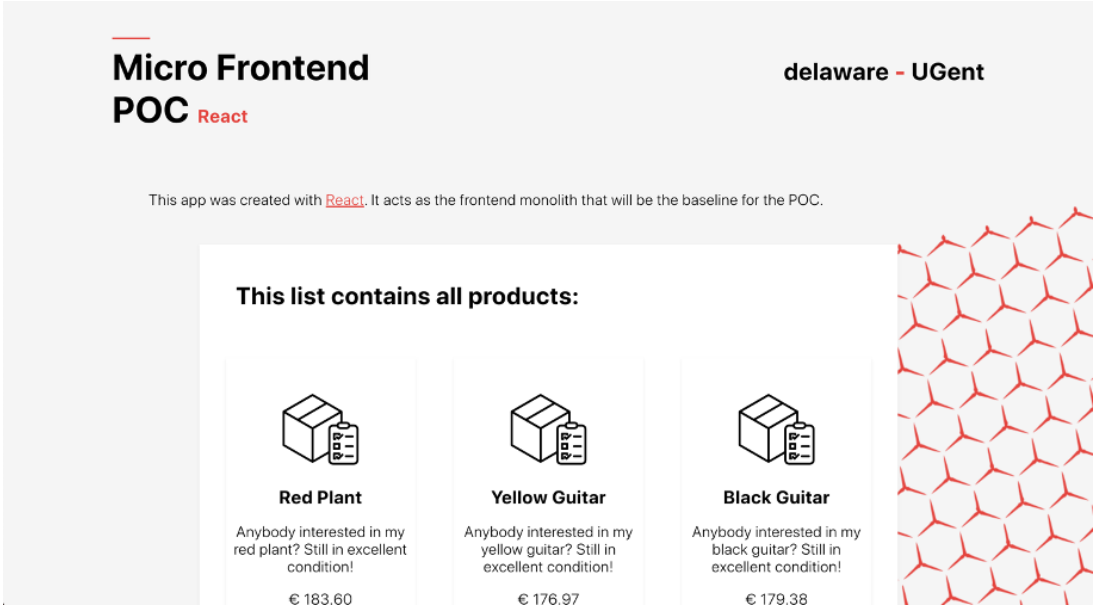
Om de gerealiseerde *micro frontend* aanpak uit de PoC te kunnen vergelijken op vlak van performantie en SEO was een referentiepunt nodig. Daartoe werd de functionaliteit die de PoC zou voorzien allereerst volledig geïmplementeerd in een monolithische React applicatie. De applicatie zélf werd initieel opgezet door gebruik te maken van de Create React App Toolchain van Facebook.

Verder werden twee functionele componenten gemaakt, namelijk ProductList en Product. De ProductList mapt een lijst van producten naar Product componenten die telkens de eigenschappen van het product in kwestie meekrijgen als *properties* dankzij de spreidingsoperator, zie Figuur 5.6. De App component, de hoogst liggende component van de applicatie, maakt gebruik van de ProductList component.

```
function ProductList() {
    return (
        <div className="product-list">
        <h1>This list contains all products: </h1>
        <div className="products">
        //map elk product in de lijst naar een Product component
            {products.map(p => <Product key={p.productId} {...p} ></Product>)}
        </div>
        </div>
    )
}

    …

// De definitie van het Product component
function Product(props) {
    return (
        <div className="product">
        <img className="product-image" src={im} alt={"image of product: " + props.
    name} ></img>
        <h2>{props.name}</h2>
        <p>{props.description}</p>
        <p€> {props.price.toFixed(2)}</p>
        </div>
    )
}

```

Figuur 5.6: ProductList en Product componenten uit React applicatie ter referentie

De resulterende React applicatie geeft een visuele representatie van de volledige productcatalogus, zie Figuur 5.7. De PoC moet de productcatalogusfunctionaliteit als *micro frontend* afzonderen zodanig dat deze technologieonafhankelijk, performant en SEO-vriendelijk is.



Figuur 5.7: Schermafbeelding van de React applicatie

5.3.3 Micro Frontend met Web Components

Zoals eerder beschreven in sectie 5.2 viel de keuze, gezien de context van de PoC, op de Web Components methodiek voor *micro frontends*.

Beide React componenten werden omgevormd naar Web Components. Bovendien werden hun functionaliteit verder uitgebreid om de mogelijkheden van deze methodiek te demonstreren. Zo werd er bidirectionele communicatie geïmplementeerd door het DOM als klankbord te gebruiken aan de hand van Custom Events. Dit om de mogelijkheid tot samenwerking tussen verschillende *micro frontends* te demonstreren. Ook werd de lijst van producten voorzien van in- en uitklapfunctionaliteit. Dit om te demonstreren hoe functionaliteiten kunnen geconfigureerd worden door de gebruikers van de *micro frontend*. Zo kunnen deze componenten binnen het consultancylandschap, dankzij de configuratiemogelijkheden, meer algemeen gemaakt worden en in meer situaties aangewend worden.

Beide componenten worden geïmplementeerd als JavaScript klassen die de HTMLElement klasse uitbreiden en daarna als Custom Element worden gedefinieerd.

Allereerst wordt telkens een element in de Shadow DOM gemaakt waar later de innerHTML eigenschap van gezet wordt. Ook de stijling wordt hierin geplaatst door gebruik te maken van een *style tag*. Dankzij de overerving van HTMLElement kon er gebruikgemaakt worden van enkele essentiële functies: de connectedCallback en disconnectedCallback levenscyclifuncties worden

respectievelijk gebruikt voor het aanmaken en opkuisen van de nodige Event Listeners, onder meer voor de Custom Events die de bidirectionele communicatie mogelijk maken; de observedAttributes functie laat toe te definiëren welke attributen geobserveerd moeten worden voor het element; de attributeChangedCallback voorziet het nodige om te kunnen reageren wanneer een van de geobserveerde attributen wijzigt. Figuur 5.8 geeft voor elk van deze levenscyclifuncties de pseudo code terug ter illustratie.

```
// Reactie op Custom Event definiëren
connectedCallback() {
    document.addEventListener('naam-van-event', () => {
        //Wat er moet gebeuren wanneer het event zich voordoet
    })
}
// Opkuisen van Event Listener
disconnectedCallback(){
    document.removeEventListener('naam-van-event')
}
// Te observeren attributen definiëren
static get observedAttributes() {
    // Lijst van de te observeren attributen
    return ['attr1'];
}
// Reactie op verandering van attribuut definiëren
attributeChangedCallback(attrName){
    if(attrName === 'attr1'){
        // Wat er moet gebeuren indien attr1 wijzigt
    }
}

```

Figuur 5.8: Pseudocode van levenscyclifuncties van HTMLElement

De configuratie van de in- en uitklapfunctionaliteit, en de functionaliteit zélf, worden voorzien door gebruik te maken van deze attributen en de wijziging ervan. De volledige code voor beide Web Components is raadpleegbaar in Bijlage B.

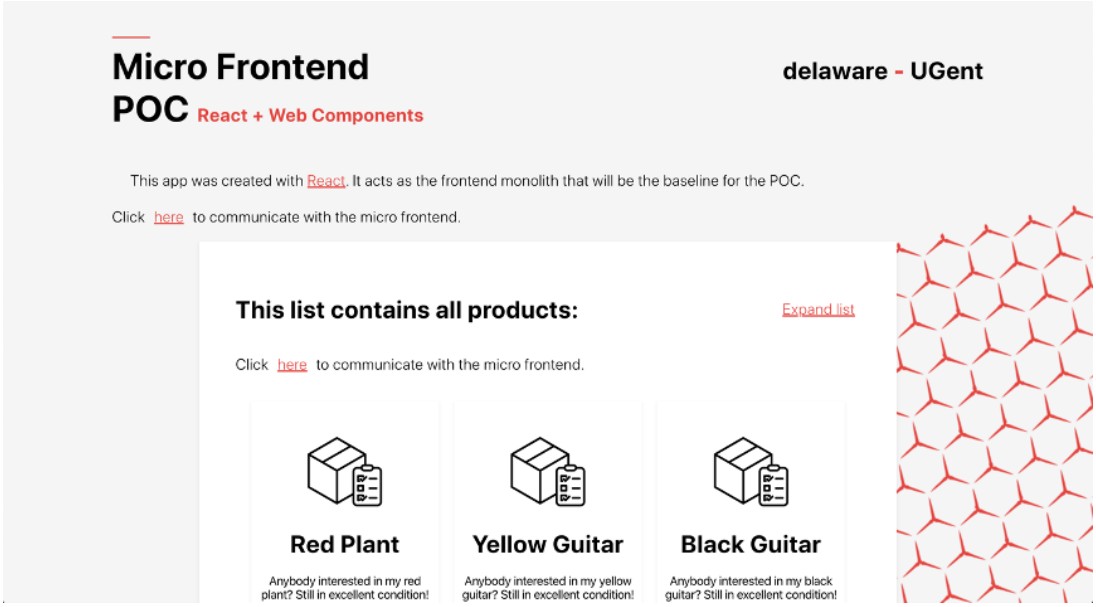
De *micro frontend* werd initieel als JavaScript module ingeladen in een tweede React project om de integratiemogelijkheid ervan aan te tonen. Eens de module geïmporteerd werd aan de hand van een *import statement* kon de tag van de Web Component gebruikt worden binnen de App component van de React applicatie. Om de toegevoegde functionaliteit in de containerapplicatie in te weven werd er in de functionele App component gebruikgemaakt van twee React Hooks, namelijk useEffect en useState. React Hooks werden in 2018 geïntroduceerd en laten onder meer

toe functionele componenten van *state* en andere React functionaliteiten te voorzien. De useEffect Hook wordt gebruikt om de Event Listeners aan te maken en op te kuisen. De useState Hook wordt gebruikt om een boodschap weer te geven wanneer de containerapplicatie een Custom Event via het DOM ontvangt. De pseudocode voor deze aanpak is gegeven in de volgende Figuur:

```
useEffect(() => {
    document.addEventListener('naam-van-event', () => {
        //Wat er moet gebeuren wanneer het event zich voordoet
    })
    return () => {
        document.removeEventListener('naam-van-event')
    }
}, [])

```

Figuur 5.9: Pseudocode voor aanmaken/opkuisen Event Listeners via React useEffect Hook

Om de functionaliteiten te demonstreren die werden toegevoegd aan de *micro frontend* werden er klikbare elementen voorzien waarmee de lezer van dit werkstuk deze kan uitproberen. Figuur 5.10 beeldt het resultaat van deze integratiemethodiek af.



Figuur 5.10: Schermafbeelding van de React applicatie met *micro frontend*

Uit de aanpak bleek dat het doorgeven van attributen aan Web Components beperkte mogelijkheden kent. Zo worden er alleen Strings zonder witruimte verwacht indien dubbele aanhalingstekens worden gebruikt. Witruimte kan wel voorzien worden, wanneer bijvoorbeeld volzinnen doorgegeven dienen te worden, indien enkelvoudige aanhalingstekens gebruikt worden. Objecten of arrays doorgeven is niet mogelijk, tenzij men deze naar een string omvormt en deze nadien opnieuw omzet naar het originele type. Op Figuur 5.11 is zichtbaar hoe een String met witruimte geforceerd wordt doorgegeven vanuit de ProductList voor elk Product component en wat er in het Product component moet gebeuren om ervan gebruik te kunnen maken.

```
// Voor elk product wordt de volgende structuur ingevoegd.
// Een extra set enkelvoudige aanhalingstekens wordt voorzien voor het attribuut
<product-element image-alt='${"Image of a " + p.name}'>
    <span slot="name">${p.name}</span>
    <span slot="description">${p.description}</span>
    <span slot="price">${p.price.toFixed(2)}</span>
</product-element>

   …

// De waarde van het attribuut wordt uitgelezen
alt = JSON.parse(this.getAttribute('image-alt'))

   …

<div class="product">
    // Het attribuut kan naar wens gebruikt worden.
    // Hier dient het opnieuw als attribuut, vandaar opnieuw de aanhalingstekens
    <img className="product-image" src=${im} alt='${alt}'></img>
    <h1><slot name="name">Default Name</slot></h1>
    <p><slot name="description">Default Description</slot></p>
    <p€> <slot name="price">Default Price</slot></p>
</div>

```

Figuur 5.11: Doorgeven van niet-eenvoudige attribuut

Bovendien werd tijdens het uitbouwen van het gerealiseerde vastgesteld dat de algemene flexibiliteit die deze aanpak kent mogelijke nadelen tot gevolg zou hebben. Bij grote projecten, zoals de doeleinden waarvoor delaware de *micro frontends* wil gaan aanwenden, zou deze flexibiliteit kunnen leiden tot additionele complexiteit en weinig onderhoudbare *codebases*.

LitElement: real-world problemen aanpakken

Wat theoretisch een gepaste aanpak leek werd door de problemen die *real-world* toepassingen zouden introduceren verworpen. Om op deze problematieken een antwoord te kunnen bieden werd op zoek gegaan naar een verbetering op deze aanpak. Een mogelijk antwoord werd gevonden in de vorm van LitElement.

LitElement is een *light-weight library* die een klasse verschaft voor het creëren van webcomponenten die ernaar streeft om het ontwikkelingsmodel ervan te vereenvoudigen. LitElement is een onderdeel van het Polymer Project. Het introduceren van LitElement in de PoC vereenvoudigt en verduidelijkt het ontwikkelingsmodel van de webcomponenten die samen de *micro frontend* vormen. Zodoende is er een duidelijk kader waarbinnen gewerkt dient te worden, gelijkaardig aan *frameworks* zoals React of Angular. Echter heeft het als voordeel, dat er geen moeilijkheden zijn omtrent het gebruik binnen verschillende *frameworks*. Bovendien levert het gebruik van LitElement een verbetering op het gebied van performantie gezien het gebruikmaakt van de lit-html *library*.

Ook de lit-html *library* is een onderdeel van het Polymer Project. De *library* laat toe om HTML-templates te schrijven die efficiënt renderen. Er wordt gebruikgemaakt van JavaScript Tagged Template Literals. Deze worden omgevormd naar strings met *placeholders* waar er zich voordien expressies bevonden. Daarvan worden HTML-templates gemaakt waarin de *placeholders* worden ingevuld met de verwachte waarden. Dit is mogelijk aangezien er een mapping van de expressies naar de *placeholders* wordt bijgehouden. Dankzij deze mapping kan men het exacte element onmiddellijk achterhalen dat dient geüpdatet te worden wanneer een waarde wijzigt. Dit is niet mogelijk bij andere technieken zoals de Virtual DOM die door het React *framework* gebruikt wordt. Bij de Virtual DOM techniek wordt een versie van het DOM bijgehouden als object. Wanneer er aanpassingen worden gemaakt wordt dit virtuele object volledig geüpdatet. Daarna wordt het object vergeleken met het effectieve DOM om vervolgens enkel de gewijzigde stukken opnieuw te laden.

Door het gebruik van LitElement dat op zijn beurt afhankelijk is van lit-html, introduceren we twee afhankelijkheden in de PoC. Deze afhankelijkheden zijn echter minimaal in grootte. Er werd besloten dat deze bedenking niet opwoog tegen de voordelen die deze aanpak potentieel zou bieden.

5.3.4 Micro Frontend met LitElement

De gecreëerde *micro frontend* bestaat tot hiertoe uit de combinatie van de Web Components. Om de voordelen van LitElement te benutten dienen deze webcomponenten LitElement elementen te worden.

Het omvormen van de Web Components naar LitElement elementen vergt dat er in eerste instantie van de eerder vermelde LitElement klasse wordt overgeërfd. Deze klasse erft op haar beurt van de HTMLElement klasse die eerder gebruikt werd. Zodoende kunnen opnieuw de connectedCallback en disconnectedCallback levenscyclifuncties gebruikt worden voor het aanmaken en opkuisen van de nodige Event Listeners. Echter is dit enkel nodig voor de inkomende *events*. Lit-html *event binding* kan namelijk toegepast worden binnenin de renderfunctie om Event Listeners aan componenten te binden. Wat de noodzaak tot het definiëren van de Event Listeners voor uitsturen van Custom Events in de connectedCallback overbodig maakt en bijgevolg tot codereductie leidt. Figuur 5.12 geeft aan hoe de gewenste code kan afgevuurd worden wanneer er zich een klik voordoet op de in- en uitklapschakelaar met behulp van lit-html *event binding*.

```
render(){
    …
    return html`
        …
        ${ this.expanding && html`
        // Vuur de handleToggle methode af wanneer op de button geklikt wordt
            <button @click="${() => this.handleToggle(!this.short)}">
            ${this.toggleText}</button>`
        }
        …
    }

```

Figuur 5.12: *Event binding* met lit-html

De door LitElement aangeboden klasse biedt zoals reeds vermeld een structuur aan voor het creëren van webcomponenten om het ontwikkelingsmodel ervan te vereenvoudigen. Lit-html *event binding* is niet de enige optimalisatie dat door het gebruik van LitElement benut kan worden. LitElement ondersteunt ook het concept van *properties* en zorgt voor het automatisch updaten van de corresponderende code wanneer deze wijzigen. Dit laatste bespaart werk dat manueel bij de aanpak beschreven in sectie 5.3.3 diende te gebeuren aan de hand van de attributeChanged-Callback levenscyclifunctie. Het uitsparen van regels code of manuele updateregels uit zich in een kortere ontwikkelingsperiode en dus een lagere ontwikkelingskost. Bovendien is de grootte van

een *codebase* gerelateerd aan de onderhoudbaarheid ervan. Op vlak van integratie in een groter project wijzigt LitElement niets. De code kan nog steeds als module ingeladen worden. Wel introduceert het gebruik van LitElement een klasse die het creëren van webcomponenten structureert en vereenvoudigd en de werking van de webcomponenten optimaliseert dankzij uitbreidingen op de HTMLElement klasse.

5.3.5 Distributie van de micro frontend

Om de functionaliteiten geïmplementeerd aan de hand van LitElement in de praktijk te kunnen gebruiken bij verschillende bedrijven dient de integratie van deze *micro frontend* zo eenvoudig mogelijk te gebeuren. Bovendien is het van belang dat er geen single point of failure is. Moest dit wel het geval zijn zou een bug die geïntroduceerd wordt impact hebben op alle klanten die de functionaliteit gebruiken. Deze *real-world concern* sluit het eenvoudigweg deployen van de JavaScript module en run-time integratie uit.

De gekozen aanpak bestaat erin om de integratietechniek van Web Components te combineren met de build-time integratietechniek door deze te packagen en in stabiele releases beschikbaar te maken. Mits voldoende *testing* garandeert dit dat een klant een integratie heeft van een blijvend werkende versie van de functionaliteit. Gezien een functionaliteit geïmplementeerd aan de hand van LitElement slechts twee afhankelijkheden heeft, blijft de complexiteit die deze afhankelijkheden introduceren beperkt.

Om de *micro frontend* te packagen wordt deze in eerste instantie gebundeld tot een productiewaardig geheel. Daartoe wordt gebruikgemaakt van Webpack. Webpack zal de JavaScript module die de *micro frontend* definieert afschuimen op zoek naar afhankelijkheden. Deze worden dan gebundeld tot *static assets*. Bovendien zal het een *minification* uitvoeren van onze code waardoor de totaliteit in omvang gereduceerd wordt. Om performantie te verzekeren is het van belang dat de omvang van de *micro frontend package* zo veel mogelijk gereduceerd wordt gezien deze code verscheept moet worden van de webserver naar de *client*. Om Webpack hiertoe in staat te stellen worden de nodige afhankelijkheden geïntroduceerd in het LitElement project. Echter gaat het hier om afhankelijkheden die enkel van toepassing zijn voor het ontwikkelingsproces, deze zullen dus niet gebundeld worden en hebben bijgevolg geen invloed op het aantal afhankelijkheden van de resulterende *package*. Daarna wordt Webpack geconfigureerd in het webpack.config.js bestand. Dit dient te gebeuren naargelang de aard van ieder project. In Bijlage C kan de nodige configuratie voor de PoC geraadpleegd worden.

Om de *package* te publiceren wordt bij de PoC gebruikgemaakt van de eerder vermelde npm. Het publiceren van de *package* faciliteert de distributie van de *micro frontend*. Hierdoor kan de functionaliteit geïntegreerd worden als Node module.

5.3.6 Technologieonafhankelijkheid demonstreren

Eens de distributie van de LitElement *micro frontend* voorzien was, werd aan de hand van zowel een React als een Angular applicatie de eenvoud van gebruik en bovendien de technologieonafhankelijkheid aangetoond. Daartoe werden de React en Angular containerapplicaties parallel aan elkaar gemaakt.

De React containerapplicatie werd gelijkaardig aan de referentieapplicatie gemaakt. In de containerapplicatie wordt echter de LitElement *micro frontend* als npm *package* ingeladen. Om de Web Component te kunnen gebruiken werd opnieuw een ProductList functioneel component aangemaakt. Binnen dit component wordt echter enkel de Web Component opgeroepen met de gewenste configuratie. Figuur 5.13 toont het volledige gebruik van de *micro frontend* in het functioneel component. Zo wordt de npm *package* bovenaan ingeladen. Ook wordt de te gebruiken *backend* geconfigureerd. Oorspronkelijk werden de producten vanuit de containerapplicatie doorgegeven aan de *micro frontend*. Nadien werd beslist dat het bevragen van de *backend* beter kon gebeuren door de *micro frontend* zelf gezien dit beter de gewenste verticalisering biedt. Tot slot wordt hiermee ook aangetoond hoe een HTML-attribuut voor configuratiemogelijkheden kan zorgen.

```
1  import React from 'react'
2  //Inladen van npm package
3  import 'micro-frontend-poc-delaware-ugent'
4
5  function ProductList(props){
6      return (
7          <product-list expanding url={props.url}></product-list>
8      )
9  }
10
11 export default ProductList
12
```

Figuur 5.13: ProductList functioneel component uit React containerapplicatie

Om de *micro frontend* volledig te benutten worden in het App component van de containerapplicatie de nodige EventListeners en CustomEvents voorzien. De relevante code voor de React containerapplicatie is raadpleegbaar in Bijlage D.

Gezien de *micro frontend* in essentie slechts een node *package* is kan deze op een gelijkaardige manier ook zonder problemen in een Angular containerapplicatie gebruikt worden. Aan de hand van de Angular Command Line Interface (CLI) werd de containerapplicatie initieel aangemaakt. Figuur 5.14 geeft deels de HTML-template voor de AppComponent terug waaruit zichtbaar is hoe de LitElement *micro frontend* op volledig gelijkaardige wijze kan gebruikt worden als de React containerapplicatie.

```
<div class="app-container">
    <div>
        …
        <p class="landing-text">
        …
        Click <button (click)="sendEvent()">here</button> to communicate with the
    micro frontend. </p>
        <p [hidden]="!message" class="landing-text">{{message}}</p>
        <product-list url='http://gateway:2003/products/normal' expanding></product-
    list>
        <img class="bg-image" src="./assets/bg.png" alt="background hexagonalshapes
    ">
    </div>
</div>

```

Figuur 5.14: HTML-template voor de AppComponent uit React containerapplicatie

Eveneens is op Figuur 5.14 zichtbaar hoe *event binding* wordt toegepast om een CustomEvent af te kunnen vuren bij het klikken van de button. Naast *event binding* toont de daaropvolgende p-tag door middel van *property binding* aan hoe een bericht al dan niet getoond wordt wanneer deze ontvangen wordt door het afvuren van een CustomEvent binnen de *micro frontend*. Het bericht zélf wordt aan de hand van de Angular interpollatie syntax in de HTML-template ingeladen. Het inladen van de npm *package* gebeurt, in tegenstelling tot de React containerapplicatie, niet in hetzelfde bestand. Dit gebeurt in het bestand dat het component beschrijft dat gebruikmaakt van de HTML-template. Figuur 5.15 toont naast het inladen van de npm *package* ook aan hoe in Angular gebruik dient gemaakt te worden van HostListener om op CustomEvents van de *micro frontend* te kunnen reageren. De relevante code voor de Angular containerapplicatie is raadpleegbaar in Bijlage E.

```
import { Component, HostListener } from '@angular/core';
//Inladen npm package
import 'micro-frontend-poc-delaware-ugent'

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
})

export class AppComponent {

    message = null

    @HostListener('document:products:trigger', ['$event'])
    setMessage(event){
    this.message = `Received a message from the micro frontend:
${event.detail.amount} products are displayed`
    setTimeout(()=>{
        this.message = null
    }, 3000)
    }

    sendEvent(){
    document.dispatchEvent(new CustomEvent('container:trigger'));
    }
}

```

Figuur 5.15: Gebruik Hostlistener bij Angular

Net zoals de *backend* werden ook deze applicaties in containers geplaatst om het functioneren ervan te garanderen. Opnieuw wordt naar de handleiding uit Bijlage A verwezen voor de nodige stappen om de besproken code op te starten.

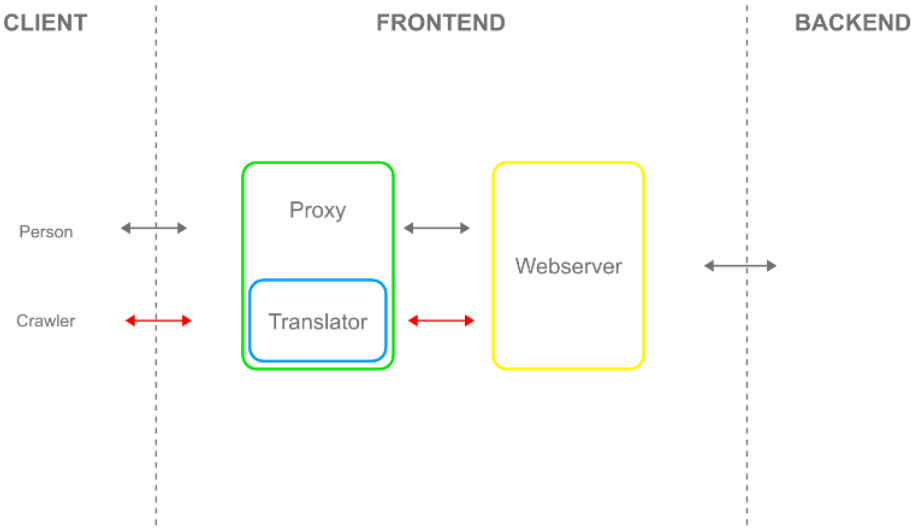
5.3.7 Ondersteunen van SEO

Om de functionaliteiten geïmplementeerd aan de hand van LitElement in de praktijk te kunnen gebruiken is SEO uiterst belangrijk. Wanneer klanten een integratie wensen mag deze geen negatieve impact hebben op de *ranking* van de webpagina voor zoekmachines. Het goed scoren op zoekmachines zoals Google, Bing, Baidu etc. vergroot het bereik van organisaties.

De score die een webpagina krijgt wordt toegekend door *bots* die het web afschuimen en telkens de aangetroffen inhoud gaan analyseren. Op basis van deze analyses worden scores toegekend en wordt de webpagina geïndexeerd. Wanneer men een zoekmachine gebruikt wordt bijgevolg niet het internet doorzocht naar het passende antwoord, wél wordt de index van de zoekmachine in kwestie geraadpleegd. Verschillende zoekmachines gebruiken hiervoor, mogelijks meerdere, verschillende *bots*. Echter maken socialemediaplatformen ook van *crawlers* gebruik. Bijgevolg wordt ook de deelbaarheid van de website bepaald door de mate waarin *crawlers* de website kunnen verstaan. De mate waarin *bots* JavaScript ondersteunen varieert sterk. Dit heeft tot gevolg dat het dynamisch toevoegen van elementen aan de hand van JavaScript door verschillende *crawlers* niet zal uitgevoerd worden wanneer deze de pagina opvragen. Dat resulteert in het feit dat de dynamische inhoud van de webpagina voor de *crawlers* in kwestie onzichtbaar is. Daardoor gaat relevante informatie verloren, wat de *ranking* niet ten goede komt. Bij SPA's wordt dit probleem traditioneel opgelost door Server Side Rendering (SSR).

Totnogtoe kan de functionaliteit die geïmplementeerd werd aan de hand van LitElement voor verscheidene zoekmachines als onzichtbaar beschouwd worden. In de praktijk is dit voor klanten een sterk nadeel. Gezien met de PoC gerealiseerd wenst te worden dat het minimale wijzigingen vergt voor de klanten is het eisen van een SSR-voorziening weinig realistisch.

Voor dit probleem wordt in de PoC een infrastructurele oplossing uitgewerkt. Er wordt een proxyserver geconfigureerd die staat tussen de *client* en de webserver waar de webpagina die de *micro frontend* integreert op is gehost. Indien de *request* afkomstig is van een *crawler* zal de nodige vertaalslag reeds server-side gebeuren, zodanig dat elke *crawler*, ongeacht het al dan niet ondersteunen van JavaScript, de volledige pagina te zien krijgt. Figuur 5.16 geeft de infrastructurele architectuur weer die gehandhaafd werd.



Figuur 5.16: Infrastructurele aanpak SEO

De proxyserver werd geïmplementeerd als een Express server en werd ter demonstratie voorzien van een caching laag. Deze proxy zal aan de hand van de User Agent string bepalen of de *request* al dan niet afkomstig is van een *crawler*. De User Agent string wordt bij elke *request* verzonden als HTTP-header en identificeert telkens het type browser, de versie en het onderliggende besturingssysteem van de oorsprong. Indien de oorsprong een *crawler* blijkt te zijn gaat de proxy een versie proberen teruggeven die gecached werd. Indien er geen geldige cache aanwezig is wordt beroep gedaan op de *headless* Chrome Node.js API Puppeteer. Puppeteer wordt aangewend om de webpagina op te halen gezien deze over de nodige functionaliteiten beschikt om dit te realiseren. Echter voorziet de Puppeteer API nog geen functionaliteit voor het vertalen van de volledige webpagina inclusief de elementen die zich in het Shadow DOM bevinden. Om dit te voorzien werd vooraf aan het opvragen van de pagina de inhoud ervan getransformeerd om de elementen uit de Shadow DOM te verplaatsen naar het DOM. De bekomen oplossingsmethode is een eigen implementatie van de methodiek beschreven door Goes [33]. Figuur 5.17 bevat de pseudocode voor deze aanpak.

```
function flatten(root) {
  // Haal alle onderliggende knopen op
  Array.from(root.querySelectorAll('*'))
    //Overloop de knopen die shadow inhoud bevatten
    .filter((element) => element.shadowRoot)
    .forEach((node) => {
      // Haal de elementen op uit de ShadowRoot
            ...
      // Recursief verder gaan
      flatten(node)
    })

  // Voor alle elementen die content zouden doorgeven aan een slot
  root.querySelectorAll('[slot]').forEach((n) => {
    let dest = n.getAttribute('slot')
    // Voeg de content aan het doelelement voor de slot
    ...
    // Verwijder slot
    ...
  })
}

```

Figuur 5.17: Pseudocode voor Shadow DOM

Wanneer na het uitvoeren van deze functie de inhoud van de pagina opgevraagd wordt via Puppeteer is deze getransformeerd en is de Shadow DOM niet langer aanwezig als obstakel. Op deze manier wordt SSR toegepast voor de *requests* van de *crawlers*. Door dit te doen wordt het patroon van oplossingen zoals Rendertron toegepast om de *crawlers* van een SSR-versie van de totale webpagina te voorzien. Rendertron is een *headless* Chrome rendering toepassing die ontwikkeld werd door het Google Chrome team. Voor de PoC werd voor zowel de React als de Angular containerapplicatie een eigen proxy voorzien.

Om te garanderen dat de PoC overal werkt werd voor elk subproject van de PoC een Docker container opgezet. Tabel **??** beschrijft welke containers aanwezig zijn.

Om het opzetten van de multi-container omgeving te vereenvoudigen is de PoC voorzien van een Docker Compose bestand. Dankzij dit bestand kunnen alle nodige containers geconfigureerd en opgestart worden met een eenvoudig commando. De handleiding in Bijlage A beschrijft in detail hoe dit gerealiseerd kan worden.

Container	**Type**	**Port**
gateway	APIGateway	2003
customer	CustomerService	2000
customer-db	MySQL datastore	3306
product	ProductService	2001
product-db	MongoDB datastore	27017
log-server	LogService	2002
log-db	InfluxDB datastore	8086
zookeeper	Consensustool	2181
kafka	Streamingsplatform	9092
baseline	React	3000
custom	React + Web Components	3001
react	React + LitElement	3002
angular	Angular + LitElement	4200
pure	HTML/JS + Web Components	2998
perf	HTML/JS + LitElement	2999
react-proxy	Express proxy	3010
angular-proxy	Express proxy	3011

Tabel 5.1: Opsomming Docker containers voor PoC

Dit hoofdstuk toetst de fundamenten van de micro frontend architecturale stijl af op de opgeleverde PoC. Nadien wordt de analyse van de PoC beschreven op vlak van performantie en SEO. Op deze twee gebieden wordt de PoC vergeleken ten opzichte van de in sectie 5.3.2 beschreven referentiepunt applicatie.

6 Analyse

6.1 Aftoetsen van de micro frontend fundamenten

In deze sectie wordt de opgeleverde PoC op vlak van de fundamenten de *micro frontend* architecturale stijl geanalyseerd. Daartoe worden de volgende aspecten onderzocht. De mate waarin de *micro frontends*: kunnen georganiseerd worden rond *business capabilities*; onafhankelijk van elkaar en op een continue wijze gedeployed worden; onderhouden worden en beschikbaar zijn, zelfs bij falen; samenwerken met elkaar ongeacht technologische keuzes rond implementatie. De performantieanalyse wordt als apart geheel beschreven in de volgende sectie.

De mate waarin de *micro frontends* kunnen georganiseerd worden rond *business capabilities* komt niet in het gedrang bij de PoC. Dit blijkt uit het feit dat de *micro frontend* georganiseerd werd rond het weergeven van producten. Dit fundament van *micro frontends* bleek uit het literatuuronderzoek overigens voor geen enkel van de besproken courante implementatiemethoden een moeilijkheid.

In de PoC werd geopteerd om de *micro frontend* niet te deployen maar in de plaats daarvan deze te distribueren als *package*. Dit heeft uiteraard als gevolg dat de *micro frontend* de fundamenten van onafhankelijke en continue deployment achter zich laat. Deze keuze werd bewust gemaakt

gezien de gevonden oplossing in de praktijk bruikbaar diende te zijn binnen de context van het consultancylandschap. Het deployen van de *package* zou in de praktijk meer risico met zich meebrengen gezien er meerdere klanten tegelijkertijd afhankelijk zouden zijn van de *micro frontend* en de infrastructuur waarop deze wordt aangeboden. Om dit tegen te gaan werd er beslist om de *micro frontend* te distribueren als een *package* waardoor de extra infrastructurele risico's onmiddellijk wegvielen. De *micro frontend* kan wel volledig onafhankelijk van andere micro frontends gedistribueerd worden. Ook kan de *micro frontend* op continue wijze gedistribueerd worden door nieuwe versies te publiceren van de package. Echter dienen de webtoepassingen van de klanten telkens geüpgraded te worden naar deze nieuwe versie indien dit gewenst is. Dit versiesysteem dient echter als veiligheidsmechanisme om klanten steeds een werkend geheel te kunnen garanderen. Het versiesysteem zou gezien de context van de thesis eveneens een vereiste geweest zijn bij een deployment-aanpak. Wanneer de *micro frontend* architectuur binnen één organisatie intern wordt toegepast is deze aanpak niet wenselijk. Wanneer de *micro frontend* echter meerdere belanghebbenden heeft zou het niet toepassen van een versiesysteem onverstandig zijn.

De mogelijkheid tot onderlinge samenwerking werd in de PoC aangetoond door de bidirectionele communicatiemogelijkheid tussen de containerapplicatie en de *micro frontend*. De methodiek die hierbij gebruikt werd, het gebruiken van het DOM als klankbord om Custom Events door te geven, kan eveneens gebruikt worden om meerdere *micro frontends* met elkaar te laten samenwerken.

Tijdens de PoC werd vastgesteld dat de algemene flexibiliteit die de pure Web Components aanpak kent mogelijke nadelen op vlak van onderhoudbaarheid tot gevolg zou hebben. Echter werd door het gebruik van de LitElement een meer rigide stramien geïntroduceerd wat voor uniformiteit binnen de *micro frontend* zorgde. Bovendien zijn teams, wanneer ze de methodiek volgen die werd toegepast in de PoC, volledig in staat om onafhankelijk van elkaar de *micro frontends* te ontwikkelen. Ook dit brengt de mate waarin de *micro frontends* onderhoudbaar zijn bijgevolg niet in het gedrang.

Gezien de *micro frontend* na het inladen van de *package* als geheel kan beschouwd worden van de containerapplicatie heeft deze noch negatieve noch positieve gevolgen ten opzichte van de beschikbaarheid. Wanneer de SEO-oplossing buiten beschouwing gelaten wordt zijn er namelijk geen infrastructurele verschillen tussen bijvoorbeeld de React referentieapplicatie en de React containerapplicatie met de LitElement *micro frontend*. Bijgevolg zijn er geen wijzigingen in de mate waarin deze beschikbaar zijn.

Uit het literatuuronderzoek bleek reeds dat zowel voor de JavaScript als Web Component methodiek als eindresultaat een JavaScript module werd verwacht. Volledig technologieonafhankelijk kan dit bijgevolg niet genoemd worden. Wanneer men naar de technologieën op hogere niveaus

kijkt, zoals bijvoorbeeld *frameworks*, zijn deze integratiemethodes wel technologieonafhankelijk. Hetzelfde bleek uit de PoC waar de *micro frontend* gebouwd uit Web Componenten als JavaScript module werd gepacked en daarna *framework* onafhankelijk kon gebruikt worden. De enige vereiste blijft dat er JavaScript kan gebruikt worden.

De analyse van de technologieonafhankelijkheid biedt inzicht op het eerste belangrijke aspect dat door delaware werd vooropgesteld. De twee andere aspecten, performantie en SEO, worden telkens als apart geheel uitgewerkt in de volgende secties.

6.2 Performantieanalyse

Naast de technologieonafhankelijkheid was performantie een belangrijk vooropgesteld aspect. Om de performantie te analyseren werden tienduizend fictieve producten in de *datastore* van de ProductService geladen om zodoende de last op de betrokken systemen op te voeren. Deze grote hoeveelheid producten ophalen gebeurt op basis van een ander *endpoint* dan de normaalsituatie. De analyse van de performantie start met het onderzoeken van een methodiek die geen optimalisaties benut. Zo is het effect van de toegepaste optimalisatiestrategie uit de PoC aan te tonen. Daartoe werd in een HTML pagina de *micro frontend* die gebouwd werd met Web Components (zie Sectie 5.3.3) ingeladen met JavaScript. Deze applicatie werd eveneens in een container geplaatst, met name de *pure* container. De performantieanalyse gaat verder met de React referentieapplicatie als maatstaf voor de normaalsituatie gezien er dankzij React automatisch een optimalisatiestrategie wordt toegepast zoals in de praktijk verwacht wordt. Zoals reeds aangehaald wordt er bij React toepassingen gebruik gemaakt van Virtual DOM om te bepalen wat er opnieuw gerenderd dient te worden. Deze aanpak wordt vergeleken met de aanpak die LitElement hanteert. De analyse van beide geoptimaliseerde methoden gebeurt met behulp van Chrome DevTools. Chrome DevTools is de set van de in Google Chrome ingebouwde ontwikkelaarsfuncties. De handleiding in Bijlage F bevat de nodige stappen om het experiment te reproduceren.

Voor beide applicaties werd een steekproef met grootte dertig uitgevoerd. Hierbij werd als reactie op een *click-event* het middelste product, op de vijfduizendste plaats in de array, aangepast. Dit veroorzaakt een noodzaak tot rendering om deze wijziging weer te geven. Elke meting uit de steekproef bevat de tijd die nodig is voor de browser om de wijziging weer te geven, gemeten vanaf de klik gebeurtenis. De gemeten waarden zijn raadpleegbaar in Bijlage G. Tabel 6.1 beschrijft de gemiddelden en standaarddeviaties van de steekproeven voor de applicaties:

	$\overline{X}$ **(tijd in ms)**	S **(tijd in ms)**
HTML/JS + Web Components	93878,90	9416,78
React	5553,73	699,18
HTML/JS + LitElement	2298,37	277,79

Tabel 6.1: Gemiddelden en standaarddeviaties voor de performantieanalyse van rendering

De resultaten tonen in eerste instantie aan dat een niet geoptimaliseerde aanpak weinig performant is. Eveneens is onmiddelijk duidelijk wat de impact van de andere optimalisatiestrategieën is. Uit de resultaten blijkt dat beide strategieën een drastische performantiewinst teweegbrengen. Echter blijkt uit de steekproef dat de performantiewinst van de lit-html aanpak groter is ten opzichte van de Virtual DOM. Deze resultaten volgen de hypothese van een mogelijke performantiewinst gezien bij Virtual DOM de verschillen bepaald worden aan de hand van een heuristisch algoritme met tijdscomplexiteit $O(n)$ [34]. De grootte van het verschil tussen de lineaire tijdscomplexiteit van de Virtual DOM aanpak bij React is uiteraard te wijten aan de plaats van het gewijzigde element.

6.3 3. SEO-analyse

Voor de SEO-analyse werd een *webcrawler* die geen javascript kan uitvoeren gesimuleerd. Het zijn namelijk dergelijke *crawlers* die niet in staat zijn om met behulp van JavaScript dynamisch elementen aan het DOM toe te voegen. In dat geval is elk dynamisch stuk inhoud onzichtbaar voor de *crawler*. Om dit soort *crawler* te simuleren werd gebruikgemaakt van twee functionaliteiten die Chrome DevTools voorziet.

Om de impact van de infrastructurele oplossing voor SEO na te gaan wordt eerst de normaalsituatie beschreven om te dienen als referentiepunt. Wanneer de React en Angular containerapplicaties opgehaald worden als normale gebruiker vanuit een webbrowser, bekomen we na het renderen de in Tabel 6.2 omschreven resultaten.

Containerapplicatie	**HTML-bestand (bytes)**
React	41235
Angular	33342

Tabel 6.2: Aantal bytes in normaalsituatie

De eerste functionaliteit van Chrome DevTools waar beroep op wordt gedaan is het uitschakelen van JavaScript voor de browser. Dit om de beperking van sommige *crawlers* na te bootsen. Om een correct resultaat te garanderen werd de cache verwijderd en de pagina geforceerd opnieuw geladen. De handleiding in Bijlage F beschrijft in detail hoe deze stappen uitgevoerd dienen te worden met Google Chrome. Na het uitschakelen van de JavaScript functionaliteit van de browser is door de afwezigheid van de inhoud bij zowel de React als Angular containerapplicaties onmiddellijk duidelijk waarom dit problematisch is voor *crawlers*. Wanneer de React en Angular containerapplicaties opgehaald worden als JavaScript uitgeschakeld is, bekomen we na rendering de resultaten uit Tabel 6.3.

Containerapplicatie	**HTML-bestand (bytes)**	**Container**	**Micro frontend**
React	15184	Onzichtbaar	Onzichtbaar
Angular	12585	Onzichtbaar	Onzichtbaar

Tabel 6.3: Aantal bytes en zichtbaarheid zonder JavaScript

De tweede functionaliteit die gebruikt wordt is het overschrijven van de User Agent string. Opnieuw wordt de cache verwijderd en de pagina geforceerd opnieuw geladen om een correct resultaat te garanderen. Deze stap wordt eveneens beschreven door de handleiding in Bijlage F .

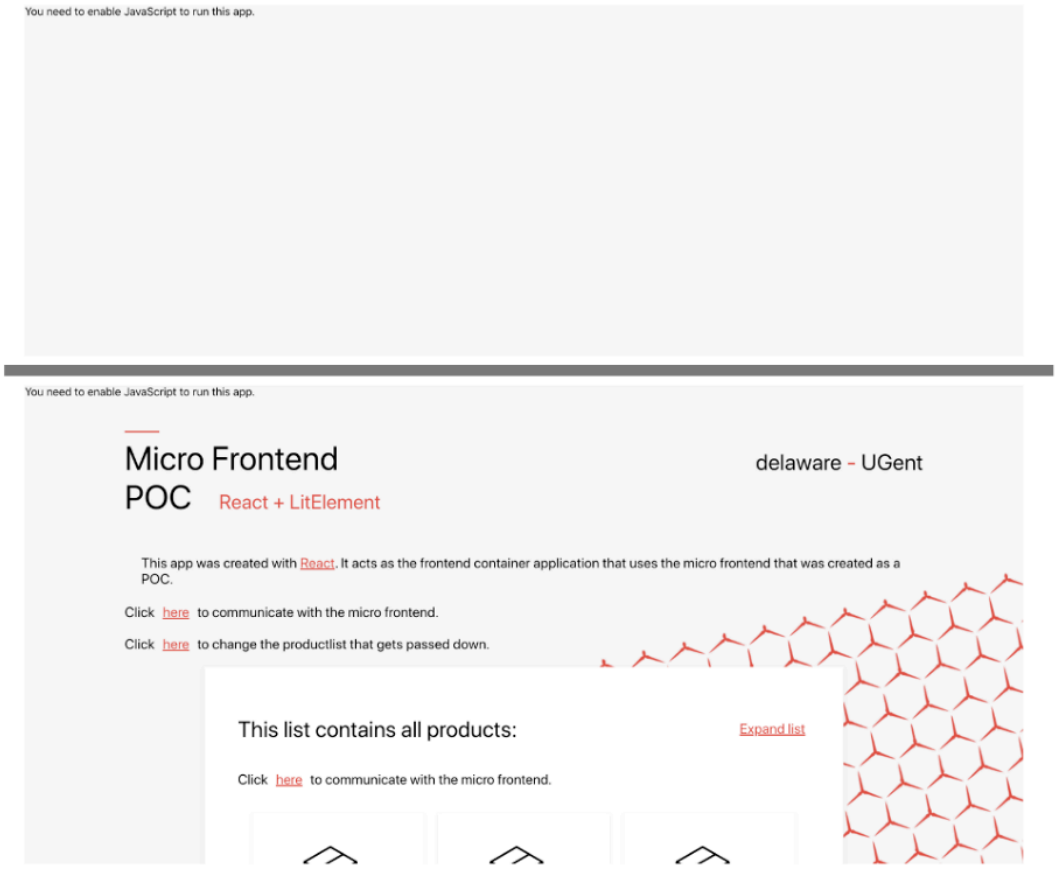
Nu de JavaScript uitgeschakeld is en de browser een *crawler* nabootst, simuleert het ophalen van beide containerapplicaties de SEO-problematiek. Tabel 6.4 beschrijft de resultaten van deze simulatie.

Containerapplicatie	**HTML-bestand (bytes)**	**Container**	**Micro frontend**
React	40927	Zichtbaar	Zichtbaar
Angular	33043	Zichtbaar	Zichtbaar

Tabel 6.4: Aantal bytes en zichtbaarheid dankzij SEO-aanpak

Het uitschakelen van JavaScript heeft tot gevolg dat het dynamisch toevoegen van inhoud aan de webpagina niet langer wordt uitgevoerd. Dit verklaart meteen het merkbare verschil in omvang van de HTML-bestanden uit Tabel 6.2 en Tabel 6.3. De uitgewerkte SSR-aanpak neemt het renderingswerk op zich waardoor de inhoud van de volledige webpagina opnieuw ontvangen wordt. Ditmaal als onmiddelijk antwoord van de *webserver*. Wanneer de resultaten uit Tabel 6.3 met die van tabel 6.4 vergeleken worden is duidelijk zichtbaar dat er meer inhoud zichtbaar is

voor de oorspronkelijke afzender van de *request*. Figuur 6.1 illustreert het verschil tussen beide resultaten voor het opvragen van de React Applicatie waarbij bovenaan het resultaat zoals beschreven in Tabel 6.3 zichtbaar is en onderaan het resultaat uit Tabel 6.4. De verschillen uit Tabel 6.2 en Tabel 6.4 zijn te wijten aan het verdwijnen van enkele HTML *tags* die werden gebruikt voor de functionaliteiten van de Shadow DOM maar door de code uit Figuur 5.17 werden weggewerkt.



Figuur 6.1: React containerapplicatie zonder/met SEO-oplossing

De resultaten van de vergelijking tussen de resultaten uit Tabel 6.2 en Tabel 6.4 geeft aan dat de infrastructurele oplossing voor de SEO-problematiek een gelijkaardige hoeveelheid inhoud verschaft als de normaalsituatie waarbij een normale gebruiker de containerapplicaties opvraagt.

7

Conclusie

Uit het literatuuronderzoek volgt dat een monolithische applicatie dusdanig groot kan worden dat deze niet langer onderhoudbaar is. Net dit gegeven was bij backendsystemen de katalysator voor *microservices*. Met *micro frontends* probeert men hetzelfde te realiseren voor de *frontend*. De aanpak is bijgevolg gebaseerd op de aanpak bij *microservices* en dus gelijkaardig. Verder bleek uit het literatuuronderzoek dat de *micro frontend* aanpak door bedrijven zelf wordt toegepast om interne teams *end-to-end* te structuren dankzij verticale splitsingen. Deze verticale splitsingen kunnen gebeuren op basis van één van de strategische patronen dat DDD bepaalt, namelijk *bounded contexts*. Dit resulteert in verticale structuren die elk rond een bedrijfsfunctie georganiseerd zijn. Uit de literatuur volgt dat zowel het toepassen van de *microservices* architectuur als de *micro frontend* architectuur additionele complexiteit geïntroduceerd wordt door de gedistribueerde aanpak waarop ze berusten. Dankzij het literatuuronderzoek werden nog enkele nadelen van *micro frontends* gevonden. Echter zijn deze voornamelijk toepasbaar wanneer een bedrijf zich volledig organiseert rond deze architectuur. Gezien delaware opereert binnen het consultancylandschap en slechts functionaliteiten op basis van deze architectuur wil aanbieden zijn zowel de voor- als nadelen van de klassieke aanpak niet langer van kracht. De onafhankelijkheid op vlak van ontwikkeling en technologie zijn echter wél van kracht. Dat leidt tot de conclusie dat de *micro frontend* architectuur in theorie een oplossing kan bieden voor de problematiek van delaware.

Uit het *state of the art* gedeelte van het literatuuronderzoek valt samenvattend te concluderen dat gezien het specifieke probleemdomein van het consultancylandschap geen enkele individuele aanpak een totaaloplossing biedt. Vandaar de beslissing om een hybride aanpak te handhaven voor de PoC. Deze oplossingsmethode geniet van voordelen die de *run-time* methode aan de hand van Web Components met zich meebrengt. De nadelen van de eerste aanpak worden aangepakt door een combinatie te maken met de *build-time* integratiemethode om de *micro frontend* te distribueren. Eveneens werd benadrukt dat verschillende teams slechts onafhankelijk van elkaar kunnen ontwikkelen wanneer er duidelijke afspraken gemaakt zijn omtrent de *micro frontends*. Het valt aan te raden om deze *best practices* ook binnenin het consultancylandschap te handhaven. Dit vertaalt zich onder meer in het documenteren van de naamgevingen van de componenten, de *events* die ze afvuren, de *events* waar ze reacties op voorzien en of stijlen al dan niet kunnen overvloeien naar andere DOM-elementen.

Tijdens de PoC werd geconcludeerd dat de algemene flexibiliteit die de Web Components aanpak kent mogelijke nadelen tot gevolg zou hebben in de praktijk. Bij grote projecten, zoals de doeleinden waarvoor delaware de *micro frontends* wil gaan aanwenden, zou deze flexibiliteit kunnen leiden tot additionele complexiteit en weinig onderhoudbare codebases. Het introduceren van *light-weight library* LitElement in de PoC vereenvoudigt en verduidelijkt het ontwikkelingsmodel van de webcomponenten die samen de *micro frontend* vormen. Eveneens zorgde dit, door ingebakken functionaliteiten, voor code reductie en een optimalisatiestrategie. De PoC bevestigde dat technologieonafhankelijk realiseren mogelijk is met de gebruikte methodiek. Er werd nadien vastgesteld dat het gerealiseerde als onzichtbaar kan beschouwd worden voor *webcrawlers* die geen JavaScript ondersteunen. De infrastructurele ontwikkeling die nadien werd uitgewerkt verhelpt dit probleem. Maar gezien de API waarvan gebruik gemaakt wordt nog niet voorzien is van functionaliteit om met alle aspecten van Web Componenten om te gaan waardoor aanvullende code nodig bleek. Daaruit volgt de conclusie dat SEO ondersteunen bij *micro frontends* geen evidentie is die nood heeft aan een infrastructurele oplossing wanneer men ook rekening wenst te houden met *crawlers* die niet over JavaScript functionaliteit beschikken.

Uit het aftoetsen van de PoC aan de *micro frontend* fundamenten tijdens de analyse bleek dat de PoC aan deze fundamenten voldeet met uitzondering van onafhankelijke en continue deployment. Dit is gezien de context van het probleemdomein net gewenst gezien de eerder vermelde *real-world concern* van het introduceren van een single point of failure voor meerdere klanten vermeden dient te worden. De conclusie die volgt uit de performantieanalyse is dat de optimalisatiestrategie die LitElement hanteert uiterst performant is en bijgevolg gebruikt kan worden in de praktijk. Tot slot bevestigt de SEO-analyse de noodzaak voor de infrastructurele oplossing wanneer men wenst rekening te houden met *crawlers* die niet over JavaScript functionaliteit beschikken.

Het onderzoek beschreef, zoals door delaware verwacht, de mogelijke patronen, oplossingen en aanpakken voor *micro frontend* architecturen en behandelde tevens welke technologieën bij deze aanpak kunnen gebruikt worden. De PoC illustreerde hoe *micro frontends* aangewend kunnen worden om uniforme functionaliteiten aan te bieden. Er werd op basis van de bevindingen die voortkwamen uit het onderzoek en de gehanteerde methodiek een *micro frontend* ontwikkeld voor productiedoeleinden die gedemonstreerd zal worden aan verschillende klanten. Daaruit kan geconcludeerd worden dat ook de opzet van het verduidelijken hoe deze aanpak kan toegepast worden binnen het consultancylandschap geslaagd is.

De achterliggende vraag die aanleiding gaf tot dit onderzoek, of het hanteren van de *micro frontend* aanpak al dan niet de gewenste invloed heeft op de druk van *frontend* ontwikkeling en de ontwikkelingskost die ermee geassocieerd wordt, kan door dit onderzoek niet op een empirische manier beantwoord worden. Het ontwikkelen van één *micro frontend* biedt niet genoeg data voor gegronde statistische analyses hieromtrent. Echter kan de aangetoonde technologieonafhankelijkheid en de eenvoud waarmee de *micro frontend* in meerdere applicaties kan geïntegreerd worden een basis vormen voor de hypothese dat deze aanpak weldegelijk de gewenste invloed heeft. Aanvullend intern onderzoek bij delaware zou dit, wanneer *micro frontends* reeds in verscheidene projecten bij verschillende klanten gebruikt zijn, wél op een empirische manier beantwoorden. Voor dit toekomstig onderzoek zou men de *throughput* van een aantal projecten, wanneer de *micro frontend* aanpak gehanteerd wordt, kunnen vergelijken met historische gegevens. Eveneens dient een kosten- en batenanalyse uitgevoerd te worden om de monetaire impact te beschrijven. Als alternatief zou men in plaats van historische gegevens van *split testing* kunnen gebruik maken om de periodegebonden invloeden die mogelijks aanwezig zijn bij historische data weg te werken.

Bibliografie

[1] Over ons. delaware. [Online]. Available: https://www.delaware.pro/nl-NL/About-Us

[2] E. Evans, *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[3] ——, *Domain-Driven Design Reference: Definitions and Pattern Summaries.* Dog Ear Publishing, 2014.

[4] C. Richardson, *Microservice Patterns.*

[5] V. S. Alagar and K. Periyasamy, *Specification of software systems.* Springer Science & Business Media, 2011.

[6] S. Millett and N. Tune, *Patterns, principles, and practices of domain-driven design.* John Wiley & Sons, 2015.

[7] B. Foote and J. Yoder, "Big ball of mud," *Pattern languages of program design*, vol. 4, pp. 654–692, 1997.

[8] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development. [Online]. Available: https://agilemanifesto.org/

[9] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. The reactive manifesto. [Online]. Available: https://www.reactivemanifesto.org/

[10] J. Biggs, V. H. García, and J. L. C. Salanova, *Building Intelligent Cloud Applications: Develop Scalable Models Using Serverless Architectures with Azure.* O'Reilly Media, 2019.

[11] J. Lewis and M. Fowler. Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html

[12] S. Newman, *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[13] L. Mezzalira, *Building Micro-Frontends.*

[14] M. Geers, *Micro Frontends In Action.*

[15] C. Jackson. Micro frontends. [Online]. Available: https://www.martinfowler.com/articles/micro-frontends.html

[16] R. Camden and B. Rinaldi, *Working with Static Sites: Bringing the Power of Simplicity to Modern Sites.* " O'Reilly Media, Inc.", 2017.

[17] G. Fink, I. Flatow, S. Group *et al.*, *Pro Single Page Application Development: Using Backbone. Js and ASP. Net.* Apress, 2014.

[18] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.*

[19] Home. Yarn. [Online]. Available: https://yarnpkg.com/

[20] The inline frame element. MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

[21] J. Pérez. Building spotify's new web player. [Online]. Available: https://labs.spotify.com/2019/03/25/building-spotifys-new-web-player/

[22] Getting started. Luigi. [Online]. Available: https://docs.luigi-project.io/docs

[23] D. Flanagan, *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[24] M. Frisbie, *Professional JavaScript for Web Developers.* John Wiley & Sons, 2019.

[25] A. Fedosejev, *React. js essentials.* Packt Publishing Ltd, 2015.

[26] Web components. MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components

[27] E. Bidelman. Shadow dom v1: Self-contained web components. [Online]. Available: https://developers.google.com/web/fundamentals/web-components/shadowdom

[28] Apache httpd tutorial: Introduction to server side includes. The Apache Software Foundation. [Online]. Available: https://httpd.apache.org/docs/2.4/howto/ssi.html#what

[29] Conceptual overview. Finn.no. [Online]. Available: https://podium-lib.io/docs/podium/conceptual_overview

[30] M. Tsimelzon, B. Weihl, J. Chung, D. Frantz, J. Basso, C. Newton, M. Hale, L. Jacobs, and C. O'Connell. Esi language specification 1.0. [Online]. Available: https://www.w3.org/TR/esi-lang

[31] G. Held, *A practical guide to content delivery networks.* CRC Press, 2010.

[32] T. Hombergs, *Get Your Hands Dirty on Clean Architecture: A hands-on guide to creating clean web applications with code examples in Java.* Packt Publishing Ltd, 2019.

[33] P. Goes. stab at rendering shadow dom server side. [Online]. Available: https://www.petergoes.nl/blog/my-stab-at-rendering-shadow-dom-server-side/

[34] Reconciliation. Facebook Inc. [Online]. Available: https://reactjs.org/docs/reconciliation.html

A

Algemene handleiding

A.1 Code

De volledige *codebase* voor deze scriptie is raadpleegbaar in de volgende Github *repository*: `https://github.com/TimDeSmet/masterproef-public`

De *repository* heeft de volgende structuur:

```
.
├── README.md               = De handleiding
├── assets                  = De afbeeldingen voor de README.md
├── backend                 = De 4 micro services die samen de backend vormen
├── docker-compose.yml      = Docker Compose instructies om containers op te zetten
├── frontend                = Alle frontend projecten
├── html                    = Resulterende html antwoorden uit SEO-analyse
└── proxies                 = De twee proxy servers
```

A.2 Containers

De volgende tabellen beschrijven voor elke container de bijhorende project, het type project en de aanspreekbare poort.

Backend

Container	Project	Type	Port
gateway	backend/APIGateway	Java Spring Boot Microservice	2003
customer	backend/CustomerService	Java Spring Boot Microservice	2000
customer-db	CustomerService	MySQL datastore	3306
product	backend/ProductService	Java Spring Boot Microservice	2001
product-db	CustomerService	MongoDB datastore	27017
log-server	backend/LogService	Java Spring Boot Microservice	2002
log-db	CustomerService	InfluxDB datastore	8086
zookeeper	Apache Zookeeper	Consensustool	2181
kafka	Apache Kafka	Streamingsplatform	9092

Frontend

Container	Project	Type	Port
baseline	frontend/react-baseline	React	3000
custom	frontend/react-custom-elements	React + Web Components	3001
react	frontend/react-lit-elements	React + LitElement	3002
angular	frontend/angular-lit-elements	Angular + LitElement	4200
pure	frontend/pure-with-web-components	HTML/JS + Web Components	2998
perf	frontend/pure-with-lit-elements	HTML/JS + LitElement	2999

Proxies

Container	Project	Type	Port
react-proxy	proxies/react-proxy	Express	3010
angular-proxy	proxies/angular-proxy	Express	3011

A.3 Opstarten

De volgende stappen beschrijven nauwgezet hoe elk project dat vermeld werd in de scriptie gestart en geraadpleegd kan worden.

- Clone de code van `https://github.com/TimDeSmet/masterproef-public`
- Navigeer naar deze repository.
- Voor in de root map van deze *repository* het volgende commando uit:

```
$ docker-compose up
```

Opmerking: Er wordt gebruik gemaakt van Docker Compose. Indien dit niet geïnstalleerd is op uw machine, gelieve de stappen op deze webpagina uit te voeren: `https://docs.docker.com/compose/install/`

Opmerking: Deze stap vereist heel wat geheugenruimte op uw machine om te kunnen slagen.

- Wacht tot alle containers gestart zijn.

Opmerking: De containers zijn pas echt opgestart eens het loggen uitdooft.

- Navigeer naar `http://localhost:3000` om het baseline React project te zien.
- Navigeer naar `http://localhost:3001` om het React + Web Components project te zien.
- Navigeer naar `http://localhost:3002` om het React + LitElement project te zien.
- Navigeer naar `http://localhost:4200` om het Angular + LitElement project te zien.
- Navigeer naar `http://localhost:3010` om het React + LitElement project via zijn proxy te raadplegen.
- Navigeer naar `http://localhost:3011` om het Angular + LitElement project via zijn proxy te raadplegen.
- Navigeer naar `http://localhost:2998` om het HTML/JS + Web Component project te zien.
- Navigeer naar `http://localhost:2999` om het HTML/JS + LitElement project te zien.

B

Web Components

B.1 product-list.js

```
class ProductList extends HTMLElement {

  constructor(){
    super()
    this.root = this.attachShadow({mode: 'closed'})
  }

  static get observedAttributes() {
        return ['expanding', 'short', 'products'];
    }

    attributeChangedCallback(attrName){
    if(attrName === 'short'){
      this.root.getElementById('products') && this.populateList()
    }
    if(attrName === 'products'){
      this.getAttribute('products') !== '[]' && this.populateList()
    }
  }

```

```
21  populateList(){
22    console.log(this.getAttribute('products'))
23    let product_list = JSON.parse(this.getAttribute('products'))
24    if(this.hasAttribute('expanding')){
25      product_list = this.hasAttribute('short') ? JSON.parse(this.getAttribute('
    products')).slice(0,3) : JSON.parse(this.getAttribute('products'))
26      this.root.getElementById('toggle').innerHTML = (this.hasAttribute('short') ? '
    Expand' : 'Collapse') + ' list'
27    }
28    this.root.getElementById('products').innerHTML = product_list.map(
29      p => `
30      <product-element image-alt='${"Image of a " + p.name}'>
31        <span slot="name">${p.name}</span>
32        <span slot="description">${p.description}</span>
33        <span slot="price">${p.price.toFixed(2)}</span>
34      </product-element>
35    `).join(' ')
36  }
37
38  makeExpandable(){
39    this.setAttribute('short', '')
40    this.root.getElementById('toggle').addEventListener('click', () => {
41      if(this.hasAttribute('short')){
42        this.removeAttribute('short')
43      }else{
44        this.setAttribute('short', '')
45      }
46    })
47  }
48
49    connectedCallback() {
50    console.log('ok')
51    this.render()
52    this.hasAttribute('expanding') && this.makeExpandable()
53    this.populateList()
54
55    document.addEventListener('container:trigger', () => {
56      this.root.getElementById('responder').innerText= "You pressed the button in
    the container application."
57      setTimeout(() => {
58        this.root.getElementById('responder').innerText = ""
59      }, 3000)
60    })
61
62    this.root.getElementById('trigger').addEventListener('click', () => {
63      document.dispatchEvent(new CustomEvent('products:trigger', {
```

```
64        detail: {
65          amount: JSON.parse(this.getAttribute('products')).length
66        }
67      }))
68    })
69  }
70
71  render(){
72    this.root.innerHTML = `
73      <style>
74        p, button {
75          font-size: 1.25rem;
76          font-weight: 300;
77        }
78
79        button {
80          text-decoration: underline;
81          color: var(--primary);
82          border: none;
83        }
84
85        button:focus {
86          outline:none;
87        }
88
89        button:hover {
90          cursor: pointer;
91        }
92
93        .product-list {
94          margin: 0 10%;
95          background-color: white;
96          box-shadow: 0 .125rem .25rem rgba(0,0,0,.075);
97          overflow:auto;
98          margin-top:1rem;
99          padding: 3rem;
100       }
101
102       .product-list>h1 {
103         margin: 3rem 0rem;
104       }
105
106       #products {
107         display: flex;
108         flex-wrap: wrap;
109         justify-content: space-evenly;
```

```
110         }
111
112         product-element{
113           width: 30%;
114           margin: 1rem 0;
115                     box-shadow: 0 .125rem .25rem rgba(0,0,0,.075);
116         }
117
118         .header{
119           display: flex;
120           justify-content: space-between;
121         }
122
123       </style>
124       <div class="product-list">
125         <div class="header">
126           <h1>This list contains all products: </h1>
127           <button id="toggle"></button>
128         </div>
129         <p id="responder"><p>
130         <div id="products"></div>
131       </div>`
132   }
133
134   disconnectedCallback(){
135     this.root.getElementById('toggle').removeEventListener('click')
136     this.root.getElementById('trigger').removeEventListener('click')
137     document.removeEventListener('container:trigger')
138   }
139 }
140
141 customElements.define('product-list', ProductList)
```

B.2 product.js

```
1  const im = require('../assets/box.svg')
2
3  class Product extends HTMLElement{
4
5      connectedCallback(){
6          const root = this.attachShadow({mode: 'closed'})
7          let alt = this.getAttribute('image-alt') ?
8              this.getAttribute('image-alt') :
9              'No alt text given for product image'
10         root.innerHTML = `
11             <style>
```

```
12                  .product {
13                      text-align: center;
14                      padding: 10px;
15                  }
16                  .product-image{
17                      width: 45%;
18                      margin-top:2rem;
19                  }
20              </style>
21
22              <div class="product">
23                  <img class="product-image" src=${im} alt='${alt}'></img>
24                  <h1><slot name="name">Default Name</slot></h1>
25                  <p><slot name="description">Default Description</slot></p>
26                  <p€> <slot name="price">Default Price</slot></p>
27              </div>
28          `
29      }
30  }
31
32  customElements.define('product-element', Product);
```

C

Webpack

C.1 webpack.config.js

```
const path = require("path");

module.exports = {
    entry: "./index.js",
    target: 'node',
    output: {
        filename: "main.js",
        path: path.resolve(__dirname, "dist"),
        publicPath: "dist/"
    },
    module: {
        rules: [
            {
                test: /\.css$/,
                // css-loader: interprets our import statement for the css
                // style-loader: injects our css to the DOM
                use: ["style-loader", "css-loader"]
            },
            {
                test: /\.(jpg|png|svg)$/,
```

```
21                 // url-loader: transforms files into base64
22                 loader: "url-loader",
23                 options: {
24                     name: "[name].[ext]",
25                     outputPath: "images",
26                     limit: Infinity
27                 }
28             }
29         ]
30     }
31 };
```

D

React containerapplicatie

D.1 App.js

```
import React, { useEffect, useState } from 'react'
import './App.css'

import ProductList from './components/ProductList'

const background = require('./assets/bg.png')

function App() {
  const [message, setMessage] = useState(null)

  useEffect(() => {
    document.addEventListener('products:trigger', (event) => {
      setMessage(
        `Received a message from the micro frontend: ${event.detail.amount} products are displayed`
      )
      setTimeout(() => {
        setMessage(null)
      }, 3000)
    })
```

```
20      return () => {
21        document.removeEventListener('products:trigger')
22      }
23    }, [])
24
25    return (
26      <div className="app-container">
27        <div>
28          <hr />
29          <div className="page-titles">
30            <h1 className="big-title">
31              Micro Frontend
32              <br />
33              POC <span className="red version">React + LitElement</span>
34            </h1>
35            <h1 className="medium-title">
36              delaware <span className="red">-</span> UGent
37            </h1>
38          </div>
39        </div>
40        <p className="landing-text">
41          This app was created with <a href="https://reactjs.org/">React</a>. It acts
      as the
42          frontend container application that uses the micro frontend that was created
       as a
43          POC.
44        </p>
45        <p>
46          Click{' '}
47          <button
48            onClick={() => {
49              document.dispatchEvent(new CustomEvent('container:trigger'))
50            }}
51          >
52            here
53          </button>{' '}
54          to communicate with the micro frontend.
55        </p>
56        {message && <p className="landing-text">{message}</p>}
57        <ProductList url="http://gateway:2003/products/normal"></ProductList>
58        <img className="bg-image" src={background} alt="background hexagonal shapes
      "></img>
59      </div>
60    )
61  }
62
```

```
63 export default App
```

D.2 ProductList.js

```
1  import React from 'react'
2  //Inladen van npm package
3  import 'micro-frontend-poc-delaware-ugent'
4
5  function ProductList(props){
6      return (
7          <product-list expanding url={props.url}></product-list>
8      )
9  }
10
11 export default ProductList
```

E

Angular containerapplicatie

E.1 app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  schemas: [
    CUSTOM_ELEMENTS_SCHEMA
  ]
})
export class AppModule { }
```

E.2 app.component.ts

```
import { Component, HostListener } from '@angular/core';
import 'micro-frontend-poc-delaware-ugent'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})

export class AppComponent {

  message = null

  @HostListener('document:products:trigger', ['$event'])
  setMessage(event){
    this.message = `Received a message from the micro frontend: ${event.detail.
    amount} products are displayed`
    setTimeout(()=>{
      this.message = null
    }, 3000)
  }

  sendEvent(){
    document.dispatchEvent(new CustomEvent('container:trigger'));
  }
}
```

E.3 app.component.html

```
<div class="app-container">
  <div>
    <hr />
    <div class="page-titles">
      <h1 class="big-title">Micro Frontend<br />POC  <span class="red version">
    Angular + LitElement</span></h1>
      <h1 class="medium-title">delaware <span class="red">-</span> UGent</h1>
    </div>
    <p class="landing-text">
      This app was created with <a href="https://angular.io/">Angular</a>. It acts
    as the frontend container application that uses the micro frontend that was
    created as a POC.</p>
      <p>Click <button (click)="sendEvent()">here</button> to communicate with the
    micro frontend. </p>
      <p className="landing-text">{{message}}</p>
```

```
12      <product-list url='http://gateway:2003/products/normal' expanding></product-
    list>
13      <img class="bg-image" src="./assets/bg.png" alt="background hexagonal shapes">
14    </div>
15  </div>
```

F

Handleiding experiment

F.1 Openen van Google Chrome DevTools

Voer de volgende stappen uit om de DevTools van Google Chrome te kunnen gebruiken:

- Open Google Chrome

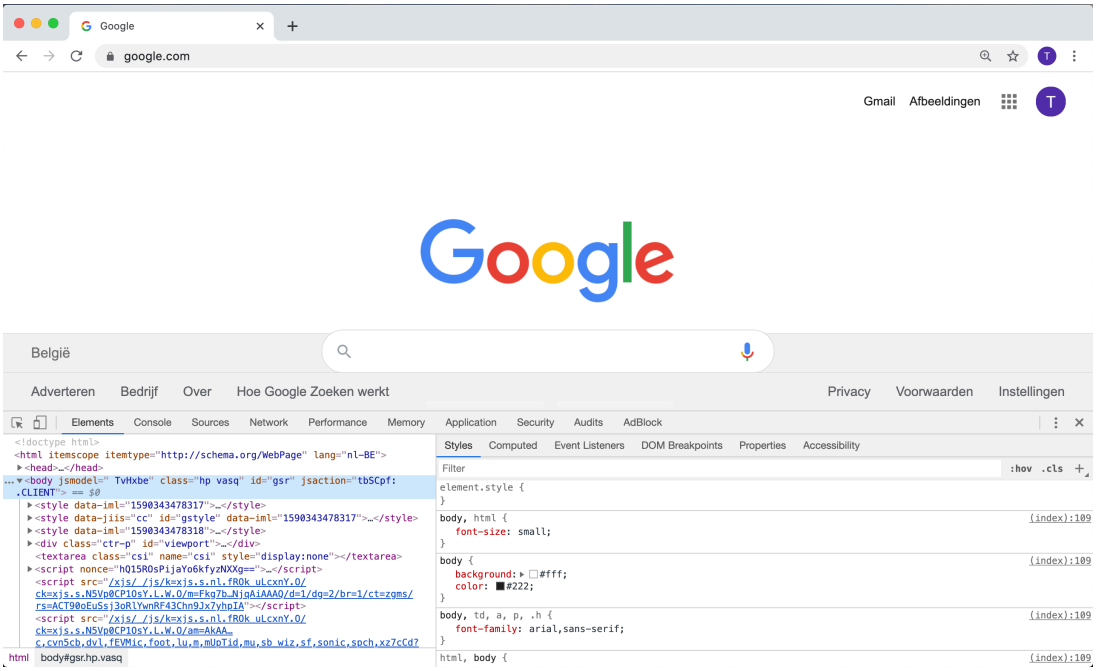
Opmerking: Indien u niet over de Google Chrome browser beschikt dient u deze eerst te downloaden. Dit kan via de volgende link: `https://www.google.com/intl/nl/chrome/`

- Voer de volgende toetsencombinatie uit:

  Mac: Command+Shift+C

  Windows: Control+Shift+C

- De DevTools zijn nu zichtbaar:

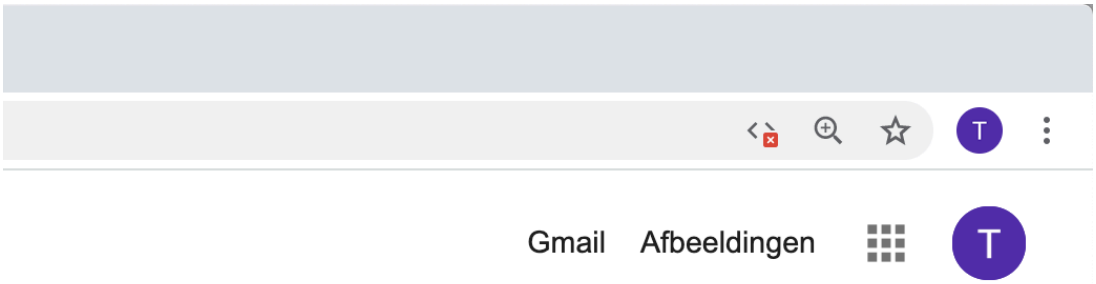


F.2 Uitschakelen van JavaScript

- Voer de stappen in Sectie F.1 uit.
- Klik eender waar in het DevTools paneel.
- Voer de volgende toetsencombinatie uit:

  Mac: Command+Shift+P

  Windows: Control+Shift+P
- Geef het volgende in: ”Disable Javascript”.
- Druk op enter.
- JavaScript is uitgeschakelt:

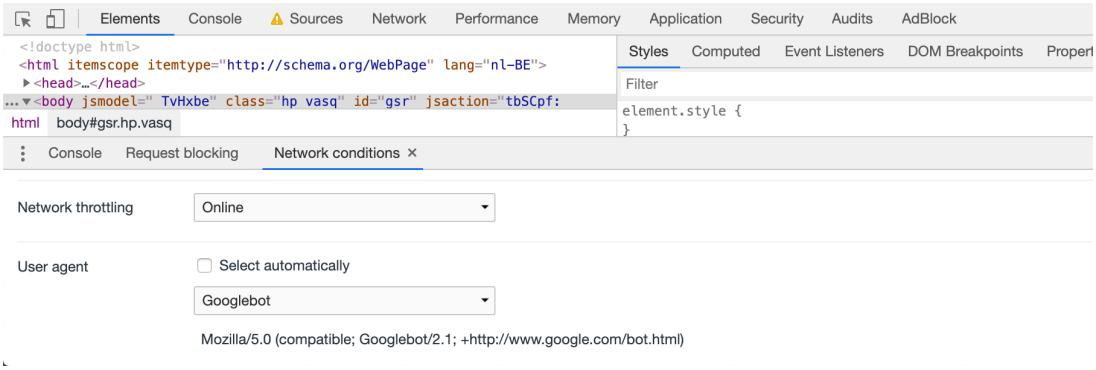


F.3 User Agent string vervangen

- Voer de stappen in Sectie F.1 uit.
- Klik eender waar in het DevTools paneel.
- Voer de volgende toetsencombinatie uit:

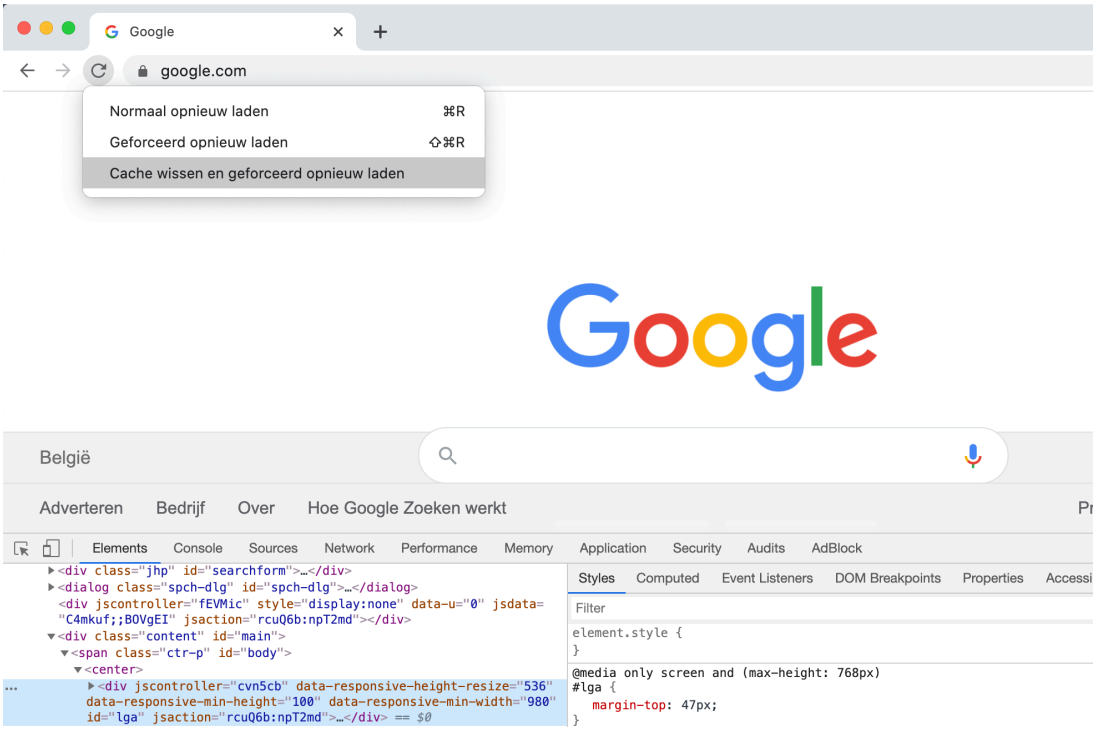
  Mac: Command+Shift+P

  Windows: Control+Shift+P
- Geef het volgende in: ”Show Network conditions”.
- Druk op enter.
- Vink de ”Select automatically”-optie bij de User Agent-sectie af.
- Selecteer Googlebot uit de lijst.
- De User Agent string is vervangen:



F.4 Harde pagina reload

- Voer de stappen in Sectie F.1 uit.
- Rechterklik links bovenaan de pagina op de reload knop (ronde pijl).
- Selecteer de optie: ”Cache wissen en geforceerd opnieuw laden”.



G

Resultaten

G.1 HTML/JS + Web Components

RESULT (in ms)	TOTAL (in ms)	IDLE (in ms)
78617	79295	678
83077	85309	2232
88543	90515	1972
98271	99856	1585
87822	88620	798
84879	86522	1643
86119	87570	1451
83104	86513	3409
85335	90218	4883
92852	99955	7103
89198	91983	2785
86311	87685	1374
114124	114641	517
110007	110891	884

110157	111308	1151
94004	96111	2107
100051	101948	1897
93539	96351	2812
98001	99874	1873
97093	98565	1472
111502	114139	2637
97472	99836	2364
86454	87061	607
91622	92321	699
91898	94095	2197
96819	98544	1725
83092	84178	1086
93090	94921	1831
108448	110902	2454
94866	96992	2126

G.2 React

RESULT (in ms)	TOTAL (in ms)	IDLE (in ms)
5487	6551	1064
5737	6809	1072
5498	5952	454
3919	4796	877
6233	7011	778
6034	6797	763
4955	5931	976
6108	6492	384
5429	6499	1070
5596	6664	1068
7041	8119	1078
5685	6774	1089
6007	7934	1927
5649	6965	1316
6657	7041	384
5332	7259	1927
4564	5520	956
6488	7074	586
4839	5229	390
5450	6861	1411
5986	7310	1324
6219	8050	1831
5231	5748	517
6390	7906	1516
4814	5213	399
5685	6428	743
4717	5563	846
4806	5660	854
4716	5337	621
5340	6318	978

G.3 HTML/JS + LitElement

RESULT (in ms)	TOTAL (in ms)	IDLE (in ms)
2516	2909	393
2507	3281	774
2261	3043	782
2458	3259	801
2217	2660	443
1972	2370	398
2914	3376	462
2105	3127	1022
2024	2583	559
2329	3039	710
2069	2455	386
2121	2372	251
3199	3515	316
2036	2404	368
1979	2435	456
2344	3290	946
2329	2960	631
2164	2694	530
2494	2905	411
2071	2621	550
2261	2820	559
2546	3167	621
2093	2556	463
2046	2431	385
2603	2944	341
2393	2964	571
2299	2791	492
2321	2766	445
2050	2498	448
2230	2912	682