OP WEG NAAR EEN MICROSERVICE FRONTEND-ARCHITECTUUR

Afstudeerverslag door Werner J.H. Struis

Datum:	18-01-2020
Versie:	1.2
School:	Saxion
Academie:	Academie Creatieve Technologie
Opleiding:	HBO-ICT
Student:	Werner J. H, Struis
Studentnummer:	418013
Telefoon:	0681748172
E-mail:	werner.struis@gmail.com
Organisatie:	Topicus
Begeleider:	Ruud Wiegers

Algemeen

Versiebeheer

Versie	Datum	Status	Beschrijving
0.1	09-09-2019	Concept	Eerste opzet
0.2	24-09-2019	Concept	Onderdelen PVA toegevoegd
0.3	30-09-2019	Concept	Toelichting projectdefinitie
0.4	10-10-2019	Concept	Uitvoering + Definitie projectfases
0.5	13-10-2019	Concept	Onderzoek huidige architectuur
0.6	17-10-2019	Concept	Requirements + Angular architectuur
0.7	22-10-2019	Concept	Mogelijke strategieen + beoordeling
0.8	29-10-2019	Concept	Conclusie onderzoek
0.9	10-11-2019	Concept	Feedback verwerkt
0.10	15-11-2019	Concept	Ontwerp: Opdelen applicaties + Wrapper
0.11	19-11-2019	Concept	Ontwerp: Authenticatie
0.12	07-12-2019	Concept	Implementatie: Angular Libraries
0.13	15-12-2019	Concept	Implementatie: Single-SPA
0.14	22-12-2019	Concept	Prestatiemetingen
0.15	29-12-2019	Concept	Conclusie
0.16	30-12-2019	Concept	Samenvatting + Voorwoord
1.0	30-12-2019	Concept	Opmaak + Verwijzingen
1.1	17-01-2020	Concept	Feedback verwerkt
1.2	18-01-2020	Final	Voorwoord

Verspreiding

Versie	Datum	Naam

Goedkeuring

Versie	Datum	Naam	Goedgekeurd JA/NEE

Inhoudsopgave

Voorwoord

Microservices zijn vandaag de dag een bekend begrip binnen de softwareontwikkeling. Het opdelen van een system naar verschillende onafhankelijke, maar samenwerkende services zorgt ervoor dat systemen en de ontwikkelteams daarachter efficiënter kunnen werken.

Tot op heden richten de microservices zich voornamelijk op de backend. De frontend van het systeem bestaat vaak uit één verzameling van code, ook wel monoliet genoemd. Na verloop van tijd groeit de frontend-laag in omvang en wordt deze moeilijker te onderhouden.

Tijdens mijn afstudeerproject heb ik gekeken naar de mogelijkheden die er zijn bij het implementeren van een microservice architectuur binnen de frontend. Dit document beschrijft het verloop van mijn project en de keuzes die gemaakt zijn.

Ik wil mijn bedrijfsbegeleider Ruud Wiegers, en mijn scriptiebegeleider Dick Heijink, hartelijk danken voor de begeleiding en het geven van feedback tijdens mijn afstudeeropdracht. Daarnaast wil ik alle collega's van Topicus bedanken voor de tijd die ze hebben genomen om mijn vragen te beantwoorden en mij advies te geven, waardoor ik in staat was mijn afstudeerperiode te voltooien.

Samenvatting

Het doel van de afstudeeropdracht was om erachter te komen wat de beste strategie is om de huidige frontend-architectuur van de 'IkBenVIP'-applicatie, gebouwd in Angular, op te delen naar een micro-service architectuur.

Om antwoord te krijgen op die vraag heb ik een onderzoek gedaan naar verschillende strategieën om een microservice architectuur te implementeren binnen de frontend van 'IkBenVIP', hierbij werd al snel bekend dat er geen eenduidig antwoord gegeven kan worden op de hoofdvraag; Het antwoord hangt sterk af van de eisen, wensen en beperkingen binnen het bedrijf.

Om toch tot een advies te komen over de beste strategie voor Topicus heb ik een keuzehulp gemaakt die door middel van een vijftal vragen advies geeft over de beste strategie voor die situatie. Door deze keuzehulp te gebruiken in combinatie met de requirements, die ik samen heb opgesteld met Topicus, ben ik tot het advies gekomen dat een implementatie met Angular libraries of Single-SPA uitkomst kan bieden.

Tijdens de implementatie van de geadviseerde strategieën werd al snel bekend dat het implementeren van een microservice architectuur binnen Angular problematisch is. Dit komt doordat Angular een 'totaaloplossing' levert, waarbij de applicatiefunctionaliteiten onlosmakelijk verbonden zijn. Wanneer functionaliteiten toch losgehaald worden, ontstaan er conflicten en problemen tijdens het bouwen en integreren van micro-applicaties.

Op basis van de bevindingen en ervaringen die ik heb opgedaan tijdens het uitvoeren van dit project ben ik tot de conclusie gekomen dat deze manier van werken niet wenselijk is. De hoofdreden hierbij is dat er te veel problemen ontstaan wanneer van de reguliere werking van Angular wordt afgestapt.

Wanneer toch een microservice architectuur geïmplementeerd gaat worden in Angular, dan zou mijn advies zijn om de mogelijkheden van een implementatie met Hyperlinks nader te onderzoeken. Het voordeel van deze oplossing is dat deze zich niet mengt in de werking van Angular, terwijl toch een groot deel van de voordelen, zoals gescheiden bouwen en deployen, behaald kunnen worden.

1 Achtergrond

1.1 Topicus & Topicus Zorg

Topicus is opgericht in 1998 en begon als adviesbureau. Vanaf de start heeft Topicus zich beziggehouden met de vraag: “Hoe kunnen de verschillende IT-systemen met elkaar verbonden worden, zodat iedereen van elkaar weet wat er aan de hand is?”

Figuur 1: Logo Topicus

Op dit moment is Topicus uitgegroeid tot een softwareleverancier met meer dan 1000 werknemers. Topicus is aanwezig in de sectoren Finance, Onderwijs, Zorg, Overheid en Security. Mijn project vindt plaats binnen de sector Zorg, waar ongeveer 300 mensen werkzaam zijn.

Topicus Zorg
Topicus Zorg specialiseert zich in het ontwikkelen van software waarmee zorgverleners, patiënten, cliënten en mantelzorgers informatie met elkaar kunnen delen. Voor het leveren van optimale zorg is het cruciaal dat zorgverleners, patiënten, cliënten en mantelzorgers informatie kunnen wisselen. Topicus Zorg specialiseert zich in het ontwikkelen van software om op een veilige en snelle manier informatie uit te wisselen. Op deze manier verbindt Topicus de zorg.

1.2 Angular

Angular is een open-source framework, dat wordt onderhouden door Google en een collectief van ontwikkelaars en biedt een oplossing bij het ontwikkelen van client-side webapplicaties. In tegenstelling tot een library biedt Angular een totaaloplossing waarin alle taken en onderdelen van een applicatie gebundeld en geïntegreerd worden aangeboden. Dit heeft als nadeel dat er een stevige leercurve is, maar levert tegelijkertijd het voordeel dat het een consistentere en eenvoudigere (en dus ook goedkoper te onderhouden) applicatie als resultaat heeft.

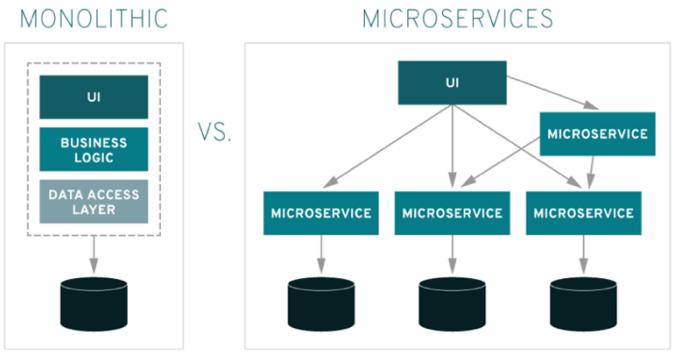

Figuur 2: Logo Angular

Wanneer gesproken wordt over Angular, gaat dit vaak over Angular versie 2 of later. Angular is de opvolger van AngularJS en is compleet opnieuw geschreven. Het grootste verschil tussen AngularJS en Angular is dat AngularJS gebaseerd is op Javascript en Angular op TypeScript. Binnen mijn project is gewerkt met Angular versie 8.

1.3 Microservices

Microservices zijn een softwareontwikkelingstechniek, waarbij een applicatie zodanig wordt gestructureerd dat het bestaat uit verschillende onafhankelijke, maar samenwerkende, services. In feite worden de verschillende functionele componenten van een applicatie losgetrokken en bestaat de applicatie uit een collectie van services. Dit is in tegenstelling tot een traditionele “monolithische” architectuur, welke gestructureerd is als één project.

Het voordeel van een microservice-architectuur is dat componenten onafhankelijk van elkaar kunnen worden ontwikkeld en gedeployed. Dit voordeel geldt met name voor de complexere applicaties met een grote codebase, waar met meerdere teams aan gewerkt wordt. In deze gevallen kan het interessant zijn om gebruik te maken van Microservices.



Figuur 3: Monolithische- vs. Microservice-architectuur

2 Projectopdracht

2.1 Aanleiding

‘IkbenVIP’ is een platform waarin patiënten o.a. hun medisch dossier kunnen inzien en thuismetingen kunnen delen. ‘IkbenVIP’ bestaat uit een app (iOS en Android) en een webportaal (Angular).

IkbenVIP

Figuur 4: Logo IkbenVIP

Het webportaal is opgezet met een microservice-architectuur. De verschillende functionaliteiten zijn gebouwd als services die onafhankelijk van elkaar kunnen worden ontwikkeld, onderhouden, aangepast, gereleased etc. Dit geldt echter alleen voor de back-end. De Angular front-end bestaat uit één grote monolithische webapplicatie waarin alle functionaliteiten worden geïntegreerd tot één geheel; Dit past echter niet bij het idee van microservices en Topicus zou dit graag anders zien.

2.2 Doelstelling

Het doel van mijn project is om erachter te komen wat de beste strategie is om de huidige frontend-architectuur van het ‘IkbenVIP’-systeem, gebouwd in Angular, op te delen naar de onderliggende services.

Om antwoord te kunnen krijgen op die vraag ga ik een onderzoek doen naar mogelijke strategieën om de Angular frontend op te splitsen naar de onderliggende microservices. Hierbij is het gebruik van Angular de enige “harde” eis. Om een optimaal advies te kunnen geven, dienen de voor- en nadelen van verschillende mogelijkheden geïnventariseerd te worden, waarna vervolgens de afweging gemaakt kan worden: wat is de optimale strategie voor Topicus?

De uitkomst van het onderzoek zal worden onderbouwd met behulp van een prototype waarin het volgende gedemonstreerd wordt:

- end-to-end functionaliteit, met front-end in Angular, en REST-API’s als backend (in een taal naar keuze)
- een manier om integraties tussen functionaliteiten te ondersteunen

2.3 Resultaten

De resultaten van het project zullen bestaan uit:

1. Een onderzoeksverslag waarin:
   - een analyse van de voor- en nadelen van de verschillende architecturen wordt gedaan
   - de verschillende architecturen met elkaar vergeleken worden
   - advies wordt gegeven over de optimale architectuur voor Topicus

2. Een prototype van de 'IkBenVIP'-applicatie waarin de geadviseerde architectuur geïmplementeerd is en het volgende gedemonstreerd wordt:
   - end-to-end functionaliteit, met front-end in Angular en REST-API als back-end
   - een manier om integraties tussen functionaliteiten te ondersteunen

2.4 Projectgrenzen

Het project loopt van 02-09-2019 tot 31-01-2020 en is gericht op de vraag: hoe kan de huidige frontend-architectuur van het 'IkbenVIP'-systeem, gebouwd in Angular, opgedeeld worden naar de onderliggende microservices?

In het onderzoek ga ik kijken naar de verschillende mogelijkheden om een frontend, geschreven in Angular, op te delen in micro-frontends. Hierbij wordt gekeken naar de verschillende voor- en nadelen van elke strategie, denk hierbij aan:

- complexiteit van ontwikkelen
- testen en deployen
- performance-impact
- mogelijkheid tot integraties tussen onderdelen

Met een prototype zal de gekozen strategie onderbouwd worden, door een prototype-versie van de 'IkbenVIP'-applicatie om te bouwen naar de nieuwe architectuur, waarbij minimaal het volgende wordt gedemonstreerd:

- end-to-end functionaliteit, met front-end in Angular en REST-API als back-end
- een manier om integraties tussen functionaliteiten te ondersteunen

2.5 Het onderzoek

De eerste fase van mijn project bestaat uit een (voor)onderzoek. Het doel van dit onderzoek is om extra kennis op te doen op het gebied van micro-frontends en de huidige applicatie-architectuur van de 'IkbenVIP'-applicatie van Topicus. De uitvoering van dit onderzoek wordt gedaan aan de hand van een hoofdvraag en een aantal deelvragen, deze staan hieronder genoemd.

2.5.1 Hoofdvraag

Met het onderzoek wil ik antwoord kunnen geven op de hoofdvraag. Deze heb ik samen met Topicus geformuleerd. De bedoeling is dat met het antwoord op de hoofdvraag, ik een idee kan maken voor de implementatie ervan binnen 'IkbenVIP'. De hoofdvraag van mijn onderzoek luidt:

> *"Wat is de beste strategie om de huidige monolithische frontend van de 'IkbenVIP'-applicatie van Topicus, geschreven in Angular, op te delen naar de onderliggende services?"*

2.5.2 Deelvragen

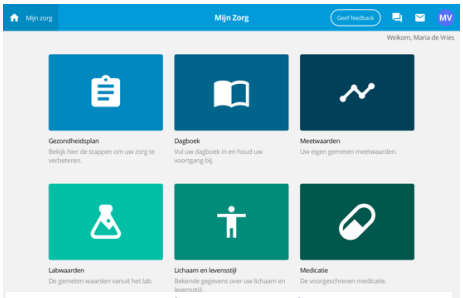
Naast de hoofdvraag, heb ik ook een aantal deelvragen geformuleerd. De bedoeling is dat aan de hand van de deelvragen stapsgewijs gewerkt kan worden naar een antwoord op de hoofdvraag. Ik heb bij het opstellen van de deelvragen geprobeerd rekening te houden met de verschillende aspecten die spelen bij het wisselen van applicatie-architectuur. Ik heb de volgende deelvragen geformuleerd:

- Hoe ziet de huidige architectuur van de 'Ik ben VIP'-applicatie eruit?
- Hoe ziet de onderliggende architectuur/infrastructuur van Angular eruit?
- Welke eisen, beperkingen en wensen zijn er binnen Topicus voor een nieuwe architectuur?
- Welke technieken zijn er om de huidige architectuur op te delen naar de onderliggende services?
- Welke mogelijke strategieën zijn er om de huidige architectuur op te delen naar de onderliggende services?

2.6 Het proof-of-concept (POC)

Tijdens de tweede fase van mijn afstudeerproject ben ik bezig geweest met het onderbouwen van de uitkomst van het onderzoek dat in de eerste fase is uitgevoerd. Dit heb ik gedaan met behulp van een proof-of-concept, afgekort naar POC. In de definitiefase ben ik, samen met mijn afstudeerbegeleider, gaan kijken naar mogelijke casussen voor het POC.

In eerste instantie was de bedoeling om (minimaal) één van de onderdelen van de 'IkbenVIP'-applicatie om te bouwen naar de nieuwe architectuur, maar uiteindelijk hebben wij besloten om niet de huidige applicatie te gebruiken. Wij hebben hiervoor gekozen aangezien de complexiteit en waarde die het werken met de "echte" applicatie meebrengt weinig toevoegt aan het onderbouwen van mijn onderzoek.



Figuur 5: IkBenVIP applicatie UI

Als alternatief hebben we gekozen om gebruik te maken van een prototype-versie van de 'IkbenVIP'-applicatie. Deze versie heeft minder features dan de huidige applicatie maar bevat wel de essentiële logica zoals autorisatie en sessie management. Het enige probleem met het prototype is dat deze geen backend heeft maar gebruik maakt van mock-data[1] voor het verkrijgen van data. Om ook de interactie met een backend te kunnen tonen heb ik een REST backend geschreven welke de informatie opslaat en aanlevert.

[1] Mock-data is een gebruik waarbij het gedrag van een object op gecontroleerde wordt gesimuleerd.

3 Uitvoering

Het verloop van mijn afstudeerproject is verdeeld in een aantal fases. Deze fases hebben elk hun eigen resultaat en dragen bij aan het uiteindelijke resultaat. In de volgende hoofdstukken wil ik laten zien hoe ik mijn project gefaseerd heb en tot welke resultaten deze fasen hebben geleid.

3.1 Definitie van de projectfases

Hieronder staat een overzicht van de fases die ik heb doorlopen en de resultaten die deze hebben opgeleverd.

Fase 1: Definiëren van de opdracht
Gedurende de eerste fase van mijn afstudeeropdracht ben ik bezig geweest met definiëren van de projectopdracht. In deze fase heb ik samen met mijn afstudeerbegeleider bepaald:

- welke onderdelen wel en welke niet bij de opdracht horen
- wat het onderzoek naar voren moet brengen
- hoe ik de uitkomst van mijn onderzoek ga bewijzen

De resultaten van deze fase zijn te vinden in het plan van aanpak en een beknopte versie hiervan is tevens ook te vinden in hoofdstuk 2: de projectopdracht.

Fase 2: Uitvoeren van het (voor)onderzoek
In de tweede fase van mijn project ben ik bezig geweest met het uitvoeren van mijn (voor)onderzoek. Dit onderzoek gebeurt aan de hand van de hoofd- en deelvragen die ik in de vorige fase heb opgesteld en wordt geconcludeerd met een advies over welke architectuur het beste zal passen bij het opsplitsen van de 'IkbenVIP'-applicatie naar de onderliggende services.

De resultaten en bevindingen van het onderzoek zijn te vinden in hoofdstuk 5.

Fase 3: Bouwen van een proof-of-concept (POC)
In de laatste fase van mijn project ben ik bezig geweest met het onderbouwen van de uitkomst van mijn onderzoek. Dit wordt gedaan door een implementatie te maken van de geadviseerde strategieën, voortkomend uit het onderzoek. De implementatie bestaat uit het ombouwen van een prototype-versie van de 'IkbenVIP'-applicatie naar een microservice architectuur, waarbij de verschillende modules losgetrokken moeten worden en, waar mogelijk, onafhankelijk gemaakt worden.

Het proces dat doorlopen wordt en de tussenresultaten van deze fase worden beschreven in hoofdstuk 6 (ontwerp) en hoofdstuk 7 (implementatie)

4 Definiëren van de opdracht

Ik ben de eerste fase van mijn afstudeerperiode begonnen met een oriëntatie op het bedrijf en de producten die zij bouwen. Na deze kennismaking ben ik, in overleg met mijn afstudeerbegeleider, bezig gegaan met het vormgeven en concreet maken van mijn projectopdracht. Dit heeft uiteindelijk geleid tot de eerste versie van mijn plan van aanpak. Hieronder geef ik een kleine toelichting op de acties die ik uitgevoerd heb en de keuzes die gemaakt zijn.

Actie: Development-versie van 'IkbenVIP' installeren
Om een idee te krijgen welke applicatie er opgedeeld moet worden, heb ik al vroeg een development-versie van de 'IkbenVIP'-applicatie op mijn laptop geïnstalleerd. Binnen deze applicatie heb ik kunnen kijken naar de verschillende functionaliteiten welke aangeboden worden en ook hoe dit onder water geregeld wordt.

Keuze: Gebruik van 'IkbenVIP'-prototype voor proof-of-concept
In eerste instantie was het de bedoeling dat de uitkomst van het onderzoek bewezen zal worden door een feature module van de huidige 'IkbenVIP'-applicatie los te trekken en als onafhankelijke micro-applicatie beschikbaar te stellen. Uiteindelijk heb ik de keuze gemaakt om voor het POC gebruik te maken van een prototype-versie van de 'IkbenVIP'-applicatie, destijds ontwikkeld voor demonstratieve doeleinden. Ik heb hiervoor gekozen omdat een aantal features ontbreken in het prototype welke ook niet relevant zijn bij het onderbouwen van mijn onderzoek. Een voorbeeld hiervan is de gezondheidsplan-module, welke vanuit een technisch oogpunt niet verschilt t.o.v. de dagboekmodule.

Extra actie: Backend schrijven voor het POC
Aan het prototype van de 'IkbenVIP'-applicatie ontbraken niet alleen een aantal feature-modules. Het prototype werkte ook alleen met mock-data. Deze "nep"-data stond statisch gedefinieerd en werd binnen de applicatie opgehaald in plaats van een REST call naar een server welke vervolgens de data terugstuurt.
Er is besloten om voor het prototype een back-end te schrijven om hiermee ook end-to-end communicatie aan te kunnen tonen. Ik heb een back-end geschreven welke gebruik maakt van de MEN-stack (Mongo, Express, NodeJS). Binnen deze back-end kunnen gebruikers zich autoriseren met behulp van JWT's [2]en kunnen zij hiermee ook data ophalen welke weergegeven wordt binnen de front-end.

Keuze: Beperken van scope
Tijdens de eerste kennismaking met mijn bedrijfs- en afstudeerbegeleider heb ik feedback kunnen krijgen op mijn plan van aanpak. Een van de feedbackpunten was dat mijn onderzoek heel breed ingestoken werd door eerst te kijken naar de mogelijkheden om een microservice architectuur te implementeren en vervolgens pas naar de requirements binnen Topicus. Dit brengt het risico met zich mee dat het project uit gaat lopen. Om dit te voorkomen heb ik, samen met mijn afstudeerbegeleider, ervoor gekozen om eerst naar de requirements te kijken en vervolgens naar de mogelijkheden.

[2] JSON Web Token: Een internet standaard voor het maken van JSON-gebaseerde accesstokens

5 Onderzoek

In dit hoofdstuk ga ik verder in op het onderzoek dat ik heb uitgevoerd om een advies te kunnen geven over de beste strategie om een microfrontend-architectuur te implementeren in 'IkBenVIP'.

5.1 Hoe ziet de huidige architectuur van de 'IkbenVIP'-applicatie eruit?

Ik ben mijn onderzoek gestart met het analyseren van de huidige architectuur van de 'IkBenVIP'-applicatie. Het doel van deze deelvraag is om inzicht te krijgen in hoe de functionaliteiten binnen 'IkBenVIP' verdeeld zijn. Hiermee kan vervolgens een plan gemaakt worden over hoe de applicatie opgedeeld moet worden.

5.1.1 Groepering van functionaliteiten in Modules

Binnen Angular is het gebruikelijk om componenten en andere logica te groeperen in modules, deze zijn te herkennen aan de @NgModule decorator[3]. De groepering van functionaliteiten gebeurt op basis van feature en het resultaat wordt ook wel een 'feature module' genoemd. Uiteindelijk komen al deze feature modules komen samen in één (hoofd)module, welke beschrijft hoe de applicatie samenkomt, deze wordt ook wel de 'root module' genoemd.

Feature modules kunnen ook verder geclassificeerd worden op basis van hun functie. Er kan onderscheid gemaakt worden tussen de volgende soorten feature modules:

Soort	**Omschrijving**
Domain	Levert functionaliteiten voor een bepaald applicatiedomein. Domain modules bevatten vaak alleen declaraties, welke samenkomen in één component, welke vervolgens geëxporteerd wordt.
Routed	Lijkt veel op een domain feature module, maar verschilt doordat deze een router-configuratie exporteert i.p.v. een component. Per definitie zijn alle lazy-loaded modules een routed feature module
Routing	Levert een router-configuratie welke gebruikt wordt als aanvulling op een andere module.
Service	Een service module levert alleen functionaliteiten en geen componenten. Idealiter gezien bevat een service module alleen generieke functionaliteiten/ services. Een voorbeeld hiervan is de HttpClientModule van Angular.
Widget	Levert componenten en functionaliteiten welke gebruikt kunnen worden in andere modules. Bevat veelal component declaraties, welke ook allemaal geëxporteerd worden. Een voorbeeld hiervan is een Component Library.

Ik heb een overzicht gemaakt van de modules die aanwezig zijn binnen de IkbenVIP-applicatie en hun functie binnen de applicatie. Ook heb ik de modules geclassificeerd op soort volgens de tabel hierboven. Dit overzicht is te zien in bijlage A: Overzicht modules 'IkbenVIP'.

[3] Decorators zijn een ontwerppatroon waarmee gedrag dynamisch aan een object toegevoegd kan worden

5.1.2 Verbinden van modules

In het vorige hoofdstuk is aangegeven dat alle feature modulen samen in één module. Er zijn echter verschillende manieren waarop een module geïmporteerd kan worden, deze staan hieronder:

1. Directe import
Bij deze strategie wordt de module geïmporteerd door deze toe te voegen aan het 'imports'-veld van een module. Deze module wordt dan mee gecompileerd in de tijdens het bouwproces.

2. Dynamische import
Bij deze strategie wordt de module dynamisch geïmporteerd (zie afbeelding 8). De module wordt dan niet mee gecompileerd met de rest van de applicatie. In plaats hiervan wordt de module in een aparte bundel gezet, welke dynamisch geladen wordt wanneer deze nodig is. Dit is alleen mogelijk bij routed feature modules, welke vervolgens geladen worden bij het bezoeken van een bepaalde pagina.

```
1) loadChildren: () => import('./dagboek/dagboek.module')
2)                          .then(m => m.DagboekModule)
```

Figuur 6: Dynamic Import Syntax

5.2 Hoe ziet de onderliggende architectuur/ infrastructuur van Angular eruit?

In het vorige hoofdstuk heb ik gekeken naar, hoe de functionaliteiten binnen IkbenVIP zijn verdeeld. Onder water heeft Angular ook nog een aantal functionaliteiten welke ervoor zorgen dat de applicatie in zijn geheel werkt en binnen de browser gebruikt kan worden.
In dit hoofdstuk ben ik verder gaan kijken naar de manier waarop Angular dit mogelijk maakt. Hierbij wil ik een aantal onderdelen benoemen welke ook van belang kunnen zijn bij het maken van een implementatie.

5.2.1 Angular packages

De functionaliteiten die door Angular aangeboden worden zijn verdeeld in een aantal packages. Hieronder staat een overzicht van alle packages die minimaal nodig zijn binnen een Angular applicatie, een volledige lijst is te vinden op de website van Angular.

@angular/core - Bevat de basisfunctionaliteiten voor elke Angular applicatie. Hier worden bijvoorbeeld de [@NgModule] en [@Component] decorators gedefinieerd.

@angular/common - Bevat de veel voorkomende functionaliteiten zoals de HttpModule, waarmee HTTP requests gedaan kunnen worden.

@angular/platform-browser - Bevat functionaliteiten voor interactie met en het gebruik binnen de browser en stelt ook algemene functionaliteiten beschikbaar in andere modules.

@angular/platform-browser-dynamic - Bevat functionaliteiten om Angular binnen de browser te kunnen draaien.

@angular/router - Bevat functionaliteiten waarmee binnen de applicatie genavigeerd kan worden. Deze is niet verplicht maar wordt zeer vaak gebruikt.

5.2.2 Dependency Injection (DI)

Dependency injection (DI) is een ontwerppatroon dat ervoor zorgt dat klassen onderling data kunnen uitwisselen zonder dat er een directe relatie is. Binnen Angular is het soms zo dat een service of component gebruik moet maken van een andere service. Om aan deze afhankelijkheid (oftewel dependency) te voldoen maakt Angular gebruik van DI. Hiervoor dient de service alleen nog wel beschikbaar gesteld te worden als 'provider'.

Een service beschikbaar stellen middels DI kan op verschillende manieren:

- Bij een module definitie, aangeduid met de @NgModule decorator, kan een service geregistreerd worden als provider. Deze is vervolgens binnen de scope van de module beschikbaar.
- Bij een component definitie, aangeduid met een @Component decorator kan een service ook geregistreerd worden als provider. Deze is vervolgens binnen de scope van het component beschikbaar.
- Met een @Injectable({ providedIn: 'root' }) decorator op de service kan een service geregistreerd worden als provider binnen de root module. Deze is binnen de scope van de applicatie beschikbaar.

Services kunnen ook op meerdere plekken geregistreerd worden, deze hebben dan elk hun eigen instantie binnen de scope waar deze geregistreerd is.

5.2.3 Zone.js & Change Detection

Zone.js maakt het mogelijk om te werken met Zones in Javascript.
Een Zone is een executiecontext. Binnen deze context kunnen verschillende asynchrone acties uitgevoerd worden, zonder dat hierbij de context verloren gaat. Om dit te realiseren worden binnen de Zone voor alle asynchrone acties een Task (taak) aangemaakt. Vervolgens kan er op verschillende stadia van de uitvoering ingehaakt worden, om zo grip te houden op de executie van een asynchrone taak.

Figuur 7: Logo Zone.js

Change detection met Zone.js
Angular maakt gebruik van Zones om te bepalen wanneer het beeld geüpdatet moet worden. Om dit te bereiken heeft Angular zijn eigen zone, de NgZone, waarin alle asynchrone acties van de applicatie worden uitgevoerd. Wanneer een actie klaar is, start Angular de change detection cycle om te bepalen of het beeld geüpdatet moet worden. Wijzigingen die gemaakt worden buiten de Zone worden niet opgemerkt door Angular en zullen dus ook niet getoond worden.

```
1) ObservableWrapper.subscribe(
2)     this._zone.onTurnDone,
3)     (_) => { this._zone.run(() => { this.tick(); }); }
4) );
```

Figuur 8: Change detection in Angular

5.2.4 Webpack

Figuur 9: logo Webpack

Webpack is een Javascript module bundler welke onder water door Angular gebruikt wordt tijdens het compilatieproces. Deze stap is nodig omdat de code van een Angular applicatie veelal geschreven wordt in TypeScript met gebruik van ECMAScript[4] features. Deze code kan niet direct door de browser begrepen worden en dient dus eerst vertaald te worden naar standaard Javascript.

Code splitting met Webpack
Een andere belangrijke feature van Webpack is het faciliteren van Code Splitting. Door het configureren van Code Splitting wordt Webpack geïnstrueerd om bepaalde applicatiedelen in een aparte bundel te plaatsen, welke vervolgens dynamisch ingeladen kan worden.

Binnen Angular is het ook mogelijk om gebruik te maken van Code Splitting. Dit kan alleen voor routed feature modules door gebruik te maken van de dynamic import syntax. Onderwater ziet Webpack deze imports en wordt Webpack geïnstrueerd om, met de module als beginpunt, een aparte bundel te creëren. Angular haalt deze bundel vervolgens op wanneer een bepaalde URL bezocht wordt.

Dependency injection met Webpack
In hoofdstuk 5.2.2 heb ik verteld over Dependency Injection en hoe Angular dit faciliteert. Onder water speelt Webpack hier echter ook een grote rol in; Het is namelijk zo dat tijdens het bundelproces meerdere bestanden samengevoegd worden naar één bestand, waarbij relaties tussen bestanden verloren gaan. Om dit probleem op te lossen bouwt Webpack tijdens het compilatieproces een 'dependency graph', waarin alle afhankelijkheden (dependencies) binnen de applicatie opgenomen worden. Tegelijkertijd vervangt Webpack ook alle import statements door '__webpack_require__'. Hiermee wordt aangegeven dat de import binnen de dependency graph van Webpack opgevraagd en aangeleverd kan worden.

5.3 Welke eisen, beperkingen en wensen zijn er binnen Topicus voor een nieuwe architectuur?

Binnen Topicus zijn er ook een aantal eisen, beperkingen en wensen voor een nieuwe architectuur. Om deze requirements duidelijk te krijgen heb Ik samen met Topicus gekeken naar wat zij belangrijk vinden. Het resultaat hiervan is een lijst met requirements welke door Topicus geprioriteerd is op basis van hoe belangrijk zij het punt vinden. De bedoeling is dat deze lijst later in het onderzoek gebruikt gaat worden om de verschillende strategieën te beoordelen en onderling te vergelijken.
In dit hoofdstuk wil ik een aantal belangrijke requirements verder toelichten.
De volledige lijst met geprioriteerde requirements is te zien in bijlage B: Requirements.

- **Applicaties kunnen onafhankelijk van elkaar gebouwd worden zonder bijkomen van andere ontwikkelteams.**

Dit requirement is voor Topicus belangrijk omdat door de grootte van hun applicatie de bouwtijd behoorlijk lang is geworden. Dit kan verminderd worden wanneer niet alle onderdelen van de applicatie altijd mee gebouwd moeten worden.

- **Applicaties kunnen (evt. met behulp van een wrapper) afzonderlijk gedraaid worden**

Dit requirement is voor Topicus ook belangrijk omdat het tijdens het ontwikkelen handig is om de applicatie te draaien, zonder dat hierbij ook alle andere applicaties geladen worden.

[4] ECMAScript is een scripttaal specificatie, bedoeld voor het standaardiseren van Javascript.

5.4 Welke technieken zijn er om de huidige architectuur van de 'IkbenVIP'-applicatie op te delen naar de onderliggende services?

Een belangrijke vraag bij het implementeren van microservice architectuur is de vraag hoe de opgedeelde applicaties weer samengevoegd kunnen worden. In dit hoofdstuk ga ik kijken naar de verschillende richtingen waarin een oplossing gezocht kan worden.

5.4.1 Client-side vs. Server-side UI composition

Als eerste dient gekeken te worden naar de plek waar de verschillende UI-componenten samenkomen. Hierbij kan gekozen worden tussen:

- **Server-side UI composition**

Bij server-side UI composition worden de verschillende applicatieonderdelen binnen de server samengevoegd en vervolgens als één naar de gebruiker gestuurd. Om dit te realiseren worden binnen de server de individuele rendering pipelines gebruikt om delen van de applicatie te renderen. Vervolgens kunnen deze delen samengevoegd worden om zo het beeld te vormen. Deze wordt dan in zijn geheel naar de gebruiker gestuurd.

- **Client-side UI composition**

Bij client-side UI composition wordt het beeld samengesteld binnen de client. Hierbij haalt de browser zelf de verschillende onderdelen van een applicatie binnen en voegt deze samen. Om dit te realiseren dient vaak wel een extra 'schil' toegevoegd te worden, welke het laden en navigeren tussen applicaties regelt.

Server-side of Client-side UI composition?
Hieronder heb ik de voor- en nadelen van beide technieken om opgedeelde applicaties weer samen te voegen, samengevat:

Server-side UI composition	**Client-side UI composition**
Voordelen - Beter vindbaar in zoekmachines - Snellere eerste laadtijd - Betere performance op langzame devices **Nadelen** - De huidige applicatie ondersteunt geen Server-Side Rendering. - Sommige onderdelen hebben extra configuratie nodig voor gebruik binnen een server.	**Voordelen** - Huidige architectuur maakt ook gebruik van CSR. - Kan worden geïmplementeerd zonder ingrijpende wijzigingen in de broncode. **Nadelen** - Extra configuraties vereist - Client-side SPA's worden minder goed geïndexeerd door Web crawlers.

Samen met Topicus is besloten om voor dit project als requirement mee te nemen dat de uiteindelijke oplossing gebruik gaat maken van client-side UI composition. De reden voor deze keuze is dat de huidige applicatie ook binnen de client geconstrueerd wordt. Bovendien zijn er binnen Topicus nog geen beweegredenen om dit proces naar de server te verplaatsen en om die reden worden voor dit project de oplossingen die gebruik maken van server-side UI composition buiten scope gelaten.

5.4.2 Run-time vs. Build-time includes

Naast de vraag op welke plek het beeld samengesteld moet worden, is er ook de vraag op welk moment de onderdelen beschikbaar gesteld moeten worden. Hierin zijn de volgende keuzes mogelijk:

- **Build-time includes**

De eerste mogelijkheid is om tijdens de bouwfase delen van de applicatie op te halen en om deze als 'normale' applicatiedelen te composeren. Dit ligt het dichts bij het standaardgedrag van een Angular applicatie, waarbij alle onderdelen van een applicatie tegelijkertijd gebouwd worden en zo de applicatie vormen. Het nadeel van build-time including is dat de uiteindelijke applicatie altijd opnieuw gebouwd moet worden wanneer een deel van de applicatie geüpdatet wordt.

- **Run-time includes**

Een andere optie is om delen van een applicatie tijdens de runtime toe te voegen. Hierbij worden de delen van een applicatie van tevoren zelfstandig gecompileerd en gebundeld. Vervolgens kunnen deze bundels opgevraagd worden om delen van de applicatie in te laden.

Run-time of Build-time includes?
Hieronder heb ik de voor- en nadelen van beide technieken om opgedeelde applicaties weer samen te voegen, samengevat:

Build-time includes	**Run-time includes**
Voordelen - Weinig configuratie vereist - Services en componenten kunnen onderling gedeeld worden. - Navigatie(state) wordt gedeeld **Nadelen** - Wrapper dient altijd opnieuw gebouwd te worden. - Gevaar op afhankelijkheden door minder scheiding tussen delen. - Geen vrije keuze in technieken en versies.	**Voordelen** - Betere isolatie tussen applicatiedelen - Afzonderlijk ontwikkelen, bouwen en deployen. - Vrije keuze in technieken en versies - Kleinere eerste bundel **Nadelen** - Veel extra configuratie nodig - Componenten en services kunnen niet gedeeld worden. - Geen gedeelde navigatie, dient centraal geregeld te worden

In eerste instantie hebben mogelijkheden welke gebruik maken van run-time includes de voorkeur. De reden hiervoor is dat bij het bouwen van een applicatie welke vervolgens tijdens de build-time toegevoegd wordt aan een andere applicatie altijd het bijeffect heeft dat beide applicaties opnieuw gebouwd moeten worden. Dit is niet het geval bij run-time includes en daarom heeft deze de voorkeur.

Het is echter nog niet duidelijk of de geadviseerde oplossing gebruik zal maken van run-time of build-time includes, hiervoor moet een afweging gemaakt worden a.d.h.v. de requirements die zijn opgesteld in hoofdstuk 5.3.

5.5 Welke mogelijke strategieën zijn er om de huidige architectuur van de 'IkbenVIP'-applicatie op te delen naar de onderliggende services?

In de volgende hoofdstukken wil ik aandacht besteden aan een aantal mogelijke strategieën om de huidige architectuur van de IkbenVIP applicatie op te delen naar de onderliggende services. Hierbij ga ik kijken naar de:

- mogelijkheden van een strategie,
- de extra packages en tools die nodig zijn en
- de voor- en nadelen bij het gebruik van de strategie.

5.5.1 Hyperlinks

Hyperlinks worden gebruikt om verbindingen te maken tussen verschillende webpagina's. Deze functionaliteit kan ook gebruikt worden om te navigeren tussen verschillende applicaties, welke elk achter een eigen URL gehost zijn. Deze strategie past echter niet geheel binnen het idee van een Single-Page Application of micro-frontend, omdat de pagina bij het navigeren opnieuw geladen moet worden en het niet mogelijk is om meerdere applicaties op één pagina te tonen.

Voor- en nadelen van het gebruik van hyperlinks
Het voordeel van hyperlinks is dat deze alleen betrekking hebben op het navigeren tussen webpagina's. Elke webpagina heeft verder zijn eigen context waardoor er veel isolatie is tussen applicaties. Het nadeel is dat navigeren met hyperlinks ervoor zorgt dat de pagina herladen wordt (eigenlijk wordt er een andere pagina geladen) en dat er geen mogelijkheden zijn om meerdere applicaties op één pagina te plaatsen, wat niet binnen het idee van een micro-frontend of Single-Page Application past. Ook zullen generieke onderdelen zoals een header bij beide applicaties geplaatst moeten worden.

Voordelen	Nadelen
- Veel isolatie - Gemakkelijk te implementeren - Biedt ondersteuning voor meerdere frameworks	- Geen Single-Page experience - Pagina herlaadt bij navigeren - Geen ondersteuning voor meerder applicaties op één pagina

Conclusie
Samenvattend kan gesteld worden, dat het gebruik van hyperlinks een simpele manier is om webpagina's met elkaar te verbinden. Dit kan handig zijn wanneer een applicatie op pagina-niveau gescheiden moet worden. Ook voorkomt de sterke isolatie conflicten tussen verschillende applicaties. Met het gebruik van hyperlinks vervalt wel de Single-Page Experience, waardoor deze strategie niet geheel binnen het idee van een micro-frontend past.

5.5.2 iFrames

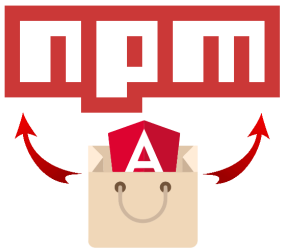
HTML Inline Framework Elements, ook wel bekend als iFrames, zijn elementen waarmee een 'frame' uit een andere website ingesloten kan worden, deze kunnen dan op de eigen website getoond worden. Hetzelfde kan gedaan worden bij webapplicaties om zo een micro-frontend te creëren. Doordat elk iFrame een eigen context heeft waarin de applicatie kan draaien, kunnen gemakkelijk meerdere applicaties, onafhankelijk, naast elkaar gedraaid worden op één pagina.

Voor- en nadelen van het gebruik van iFrames
Het voordeel van iFrames is dat deze een eigen context hebben. Binnen deze context kan gemakkelijk een applicatie gedraaid worden, zonder dat deze effecten heeft van/op andere applicaties.
Het nadeel van iframes is dat ze niet flexibel zijn. Ook worden iFrames niet geïndexeerd, wat leidt tot slechtere vindbaarheid in zoekmachines en zorgt de grote mate van isolatie tussen iFrames ervoor dat er extra maatregelen genomen moeten worden voor de navigatie en communicatie tussen applicaties.

Voordelen	Nadelen
- Veel isolatie - Meerder applicaties op één pagina mogelijk - Applicatieonderdelen kunnen hergebruikt worden.	- Slechter vindbaar in zoekmachines - iFrames zijn niet goed schaalbaar - Vereist extra maatregelen voor navigatie en communicatie - Er zijn veiligheids-risico's bij het gebruik van iFrames

Conclusie
IFrames zijn een goede optie wanneer er een hele grote mate van isolatie tussen applicaties vereist is doordat deze een grote mate van isolatie bieden. Dit kan handig zijn wanneer er gewerkt wordt met verouderde software, welke gebruik maakt van achterhaalde technieken.
Wanneer een sterke isolatie tussen applicaties niet vereist is, zijn er andere mogelijkheden, die nagenoeg dezelfde voordelen bieden, maar een stuk minder nadelen.

5.5.3 Angular Libraries (met build-time integratie)



Figuur 10: logo NPM

Er kan gebruik gemaakt worden van Angular Libraries om delen van de applicatie op te delen. Een Angular library is een Angular project, welke niet zelfstandig kan draaien. Om een library te kunnen gebruiken dient deze gepubliceerd te worden naar een NPM-register, waarna deze vervolgens geïmporteerd kan worden in een ander Angular project. Deze strategie valt niet volledig binnen het idee van een micro-frontend, maar heeft hier wel veel kenmerken van.

Benodigde packages en tools
Het is sinds Angular versie 6 zeer makkelijk geworden om een Angular library aan te maken, deze strategie heeft dan ook nauwelijks extra configuratie of extra tooling nodig.

Browser support
Omdat het gebruik van Angular Libraries een alternatieve vorm is van 'normale' Angular projecten is de browser support gelijk aan die van Angular zelf. Hieronder staat een tabel waarin staat welke browsers ondersteund worden:

Chrome	Firefox	Edge	IE	Safari	iOS	Android
Latest	Latest	2 most recent	11	2 most recent	2 most recent	Nougat 7.0

Voor- en nadelen van het gebruik van libraries
Het grootste voordeel van het gebruik van Angular libraries is dat deze dicht op de Angular infrastructuur aangesloten zijn, waardoor weinig extra configuratie nodig is en componenten herbruikbaar opgesteld kunnen worden. Het nadeel aan het gebruik van libraries is dat de applicatie welke deze gebruikt telkens opnieuw gebouwd moet worden bij een update. Ook is het niet mogelijk om meerder frameworks te integreren omdat de libraries standaard Angular functionaliteiten zijn.

Voordelen	Nadelen
- Standaard Angular functionaliteit - Weinig/geen configuratie - Libraries kunnen onafhankelijk gebouwd worden. - Meerdere 'applicaties' op één pagina mogelijk	- Applicatie dient opnieuw gebouwd te worden bij elke update - Geen ondersteuning voor meerdere frameworks

Conclusie
Het werken met libraries binnen Angular is zeer eenvoudig omdat deze direct aansluiten op de applicatie welke deze gebruikt. Libraries zijn dan ook een goede optie wanneer geen rekening gehouden hoeft te worden met het gebruiken van meerdere frameworks of het onafhankelijk kunnen deployen. Is dit geen probleem? Dan zijn Libraries een goede optie.

5.5.4 Angular Libraries (met run-time integratie)

In de vorige sectie is verteld over het gebruik van (standaard) Angular Libraries om een applicatie samen te stellen uit onderdelen. De integratie bij het normaal gebruik van Angular Libraries gebeurt tijdens de bouwfase van een applicatie. Het is echter ook mogelijk om Angular Libraries te integreren tijdens de run-time van een applicatie met SystemJS. Dit wordt niet standaard door Angular ondersteund en vereist een eigen implementatie. Op deze manier past het gebruik van Libraries ook meer binnen het idee van een micro-frontend.

Benodigde packages/tools
Naast de packages en tools die nodig zijn voor het werken met Angular Libraries, is er ook nog een package nodig voor het inladen van Libraries, deze staat hieronder.

SystemJS	Maakt het mogelijk om Angular Libraries vanuit een URL te importeren tijdens de runtime

Voor- en nadelen van het gebruik van libraries
De voor- en nadelen van Angular Libraries met run-time integratie zijn grotendeels gelijk aan die met build-time integratie. Hierbij is het kunnen integreren tijdens de run-time een grote plus is van deze strategie. Het afwijken van het standaardgedrag van Angular brengt wel een risico op ongewenst gedrag en bijeffecten met zich mee.

Voordelen	Nadelen
- Applicatie hoeft niet opnieuw gebouwd te worden bij elke update - Libraries kunnen onafhankelijk gedeployed worden.	- Kans op problemen doordat afgeweken wordt van het standaardgedrag van Angular. - Extra complex t.o.v. build-time integratie

Conclusie
Het voordeel van Libraries met run-time integratie t.o.v. de build-time integratie is dat het mogelijk is om onafhankelijk te bouwen en deployen. Hiertegenover staat een hoop extra complexiteit bij het integreren van deze strategie. Wanneer onafhankelijk bouwen en deployen een eis is, dan kan het gebruik van Libraries met run-time integratie uitkomst bieden.

5.5.5 Web-Components met Angular Elements

Web-components zijn een collectie van web platform APIs waarmee 'Custom Elements' gemaakt kunnen worden. Met de komst van Angular Elements is het ook eenvoudig geworden om Angular componenten te verpakken in een Web Component. Deze kunnen vervolgens binnen HTML-pagina's en webapplicaties gebruikt worden.

Benodigde packages/tools
Angular heeft een eigen package beschikbaar gesteld welke bedoeld is voor het werken met web-components binnen Angular. Daarnaast hebben sommige browsers een Polyfill nodig omdat deze standaard nog geen web-components ondersteunen. Hieronder staan de benodigde packages:

@angular/elements	Maakt het mogelijk om Angular componenten te verpakken in web-components.
Document-register-element	Polyfill voor gebruik van web components in sommige browsers.
Webpack of andere build tool	Nodig voor het bundelen van Angular, hiervoor kan niet de standaard Angular build tool gebruikt worden.

Browser support
Web-components worden in de meeste browsers ondersteund. Oudere browsers worden ook ondersteund m.b.v. Polyfills. Hieronder staat een tabel waarin staat welke browsers ondersteund worden:

Chrome	Firefox	Edge	IE	Safari	iOS	Android
Latest	Latest	Latest	Polyfill Beschikbaar	Latest	Latest	Latest

Voor- en nadelen van het gebruik van web-components
Omdat web-components gebruik maken van native platform APIs bieden zij een brede ondersteuning aan frameworks en mogelijkheden. Ook bieden web-components een eigen stijl-context waardoor stijl-inmenging voorkomen kan worden en is het aanmaken van web-components binnen Angular zeer eenvoudig met de Angular elements library.

Het nadeel van Angular Elements is dat deze zich alleen richt op het verpakken van Angular componenten in Web Components. Wanneer grotere delen van een applicatie verpakt worden in web-components dienen ook zaken zoals navigatie en communicatie van/in/tussen web-components door de ontwikkelaar geregeld te worden, wat leidt tot extra complexiteit. Ook is een custom bouw-configuratie vereist om Angular Elements tijdens de run-time te kunnen integreren.

Voordelen	Nadelen
- Meerder applicaties op één pagina mogelijk - Bied ondersteuning voor meerdere frameworks - Applicatieonderdelen kunnen hergebruikt worden. - Style encapsulation	- Vereist extra maatregelen voor navigatie en communicatie - Custom bouw-configuratie vereist - Het verpakken van grote(re) applicatie-onderdelen in Angular is problematisch.

Conclusie
Het gebruik van Web Components biedt een hoop mogelijkheden zoals het afzonderlijk kunnen deployen van applicaties en het kunnen integreren van meerdere frameworks. Omdat Web Components redelijk nieuw zijn en zich alleen richten op het aan kunnen maken van custom HTML-elementen dienen zaken als navigatie en communicatie nog wel geregeld te worden, wat aan de andere kant ook weer mogelijkheden biedt. Wanneer de voorgaande oplossingen niet voldoen aan de requirements, dan kan een oplossing met Web Component uitkomst bieden.

5.5.6 Single-SPA

Single-SPA is een meta-framework. In tegenstelling tot de strategieën die hierboven staan levert Single-SPA geen mogelijkheid om een applicatie te verpakken maar werkt Single-SPA als een 'top level router' welke ervoor zorgt dat de juiste applicatie(s) geladen worden bij het bezoeken van een bepaalde URL. Doordat binnen Single-SPA ook aangegeven kan worden op welke plek een applicatie-onderdeel getoond moet worden, kunnen meerdere applicaties gebruikt worden om één gezamenlijke interface te vormen.

Figuur 11: Logo Single-SPA

Benodigde packages/tools
Om een micro-frontend te bouwen met Single-SPA zijn een aantal packages nodig, deze staan hieronder:

Single-spa	Binnen deze package worden sub-applicaties geregistreerd en wordt de applicatie in zijn geheel gestart en aangeleverd.
Single-spa-angular	Deze package wordt als wrapper om een Angular applicatie geplaatst, om zo als Single-SPA applicatie gebruikt te kunnen worden.
Webpack of andere build tool	Nodig voor het bundelen van Angular, hiervoor kan niet de standaard Angular build tool gebruikt worden.

Voor- en nadelen van het gebruik van Single-SPA
Single-SPA biedt een uitkomst bij het navigeren tussen applicaties. Er zijn verschillende packages beschikbaar waarmee meerdere frameworks als Single-SPA-applicatie verpakt kunnen worden. Deze kunnen vervolgens met weinig configuratie geïntegreerd worden.
Het nadeel van Single-SPA is dat deze alleen werkt met Single-SPA-applicaties en andersom, wat het hergebruik van componenten bemoeilijkt. Ook kan het gebrek aan configuratie-mogelijkheden belemmerend zijn wanneer er bijvoorbeeld onderlinge afhankelijkheden tussen applicatie-onderdelen aanwezig zijn. Tot slot zijn er nog een aantal problemen, waardoor Single-SPA niet binnen een productie-omgeving gebruikt kan worden.

Voordelen	Nadelen
- Weinig configuratie vereist - Biedt oplossing voor navigeren binnen/tussen applicaties - Biedt ondersteuning voor meerdere frameworks - Meerdere applicaties op één pagina mogelijk	- Sterke mening over implementatie en configuratie. - Custom bouw-configuratie vereist - Alle onderdelen dienen verpakt te zijn als Single-SPA-applicatie en kunnen alleen binnen Single-SPA gebruikt worden. - Nog niet klaar voor productie

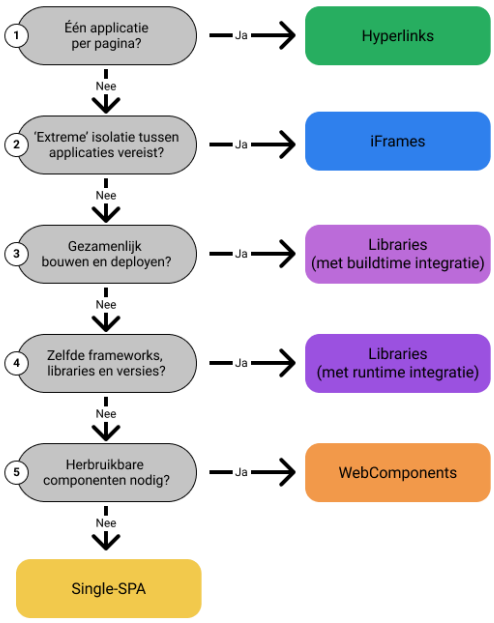
Conclusie
Het gebruik van Single-SPA is zeer eenvoudig omdat deze zich alleen richt op het laden van de juiste applicatie(s) bij het bezoeken van een bepaalde URL, zonder dat hierbij de SPA-beleving verloren gaat. Ook kan aangegeven worden op welke plek een applicatie geplaatst moet worden, waardoor de beleving van één gezamenlijke interface gemaakt kan worden.

Er zijn nog een aantal problemen die ervoor zorgen dat Single-SPA niet klaar is om gebruikt te worden in een productie-omgeving. Wanneer deze problemen opgelost zijn dan kan Single-SPA handig zijn. Een potentieel gevaar is het gebrek aan customizatie-mogelijkheden.

5.6 Conclusie

In de voorgaande hoofdstukken zijn meerdere mogelijke strategieën benoemd die gebruikt kunnen worden bij het implementeren van een microservice-architectuur binnen een frontend.
Hierbij kan geen eenduidig antwoord gegeven worden over welke strategie de beste strategie is. Elke strategie heeft eigen voor- en nadelen en zal daarmee in de ene situatie beter passen dan een andere strategie.

Aan de hand van een vijftal vragen kan een advies gegeven worden over de strategie die het beste past bij de situatie. De vragen met de bijbehorende adviezen staan in figuur 12.



Figuur 12: Keuzehulp micro-frontend strategie

Wat is de beste strategie voor Topicus
De hoofdvraag van mijn onderzoek luidt:

> *"Wat is de beste strategie om de huidige monolithische frontend van de 'Ik ben VIP'-applicatie van Topicus, geschreven in Angular, op te delen naar de onderliggende services?"*

Wanneer hiervoor gekeken wordt naar de keuzehulp dan kom ik tot de volgende antwoorden:

1. Één applicatie per pagina? → Nee
2. 'Extreme' isolatie tussen applicaties vereist? → Nee
3. Gezamenlijk bouwen en deployen → Nee
4. Zelfde frameworks, libraries en versies? → Ja (= Libraries met runtime integratie)
5. Herbruikbare componenten nodig? → Nee (= Single-SPA)

Met deze antwoorden kan de conclusie getrokken worden dat voor de situatie van Topicus het beste gebruik gemaakt kan worden van libraries met run-time integratie. Daarnaast is de optie Single-SPA ook interessant omdat deze grotendeels dezelfde mogelijkheden biedt als het gebruik van Angular libraries met run-time integratie, waarbij Single-SPA ook de mogelijkheid biedt om andere frameworks en/of versies te gebruiken.

In de volgende hoofdstukken ga ik verder in op het ontwerp dat gemaakt is voor de nieuwe microservice architectuur. Ook laat ik zien hoe een implementatie van deze nieuwe architectuur eruitziet voor zowel de library- als de Single-SPA -strategie.

6 Ontwerp

Dit hoofdstuk beschrijft het ontwerp dat gemaakt is voor het nieuwe microservice frontend-architectuur van de IkBenVIP-applicatie. Het ontwerp in zijn geheel is te vinden in Bijlage C: Architectuurplaat nieuw en in de volgende hoofdstukken worden verschillende aspecten van de nieuwe architectuurs-ontwerp benoemd en toegelicht.

Het ontwerp kan, samen met een van de strategieën die voortgekomen zijn uit het onderzoek, gebruikt worden om een implementatie te bouwen van een microservice front-end architectuur. De gemaakte implementaties van dit ontwerp binnen de IkBenVIP-applicatie zijn te vinden in Hoofdstuk 7.

6.1 Opdelen van applicatielogica naar micro-applicaties

Als eerste is gekeken op welke wijze de huidige applicatielogica van de IkbenVIP-applicatie opgedeeld kan worden in micro-applicaties.

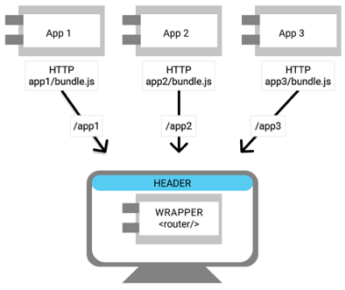
Opdeling op basis van sub-domein
Bij de opsplitsing is als uitgangspunt genomen dat elke micro-applicatie verantwoordelijk is voor een bepaald onderdeel (sub-domein) van de applicatie. Binnen een bedrijf is het ook vaak zo dat ontwikkelteams zich bezighouden met één bepaald onderdeel van de applicatie.
Deze splitsing is al gedaan binnen de back-end en door deze opdeling ook toe te passen binnen de front-end zijn ontwikkelteams alleen bezig binnen hun eigen applicatiedomein, wat een menging van functionaliteiten tussen verschillende applicatiedomeinen voorkomt.

De applicatielogica zal worden opgesplitst in de volgende micro-applicaties:

Micro-app	Functie
Account	Bevat de UI en onderliggende logica welke nodig zijn voor het weergeven en wijzigen van een gebruikers-profiel.
Dagboek	Bevat de UI en onderliggende logica welke nodig zijn voor het weergeven van het dagboek.
Gesprekken	Bevat de UI en onderliggende logica welke nodig zijn voor het weergeven van gesprekken met de zorgverlener.
Labwaarden	Bevat de UI en onderliggende logica welke nodig zijn voor het weergeven van labwaarden.
Meetwaarden	Bevat de UI en onderliggende logica welke nodig zijn voor het weergeven van meetwaarden.

6.2 Samenstellen van micro-applicaties

Om de gebruiker de beleving te bieden van één geïntegreerde applicatie wordt gebruik gemaakt van een Wrapper.
De Wrapper bestaat uit een applicatie welke gezien kan worden als een portaal, waarin verschillende micro-applicaties geladen kunnen worden. Daarbij zorgt de Wrapper ook voor het weergeven van generieke onderdelen zoals de header. Deze wordt te allen tijde getoond en is niet specifiek voor één applicatiedomein.



Figuur 13: Applicatie wrapper

Functies en verantwoordelijkheden van de Wrapper
Om een geïntegreerde gebruikerservaring te bieden dient de Wrapper te zorgen voor het:

- Bepalen waar en wanneer micro-applicaties getoond moet worden
- Ophalen van micro-applicaties van een externe bron
- Initialiseren en weergeven van micro-applicaties
- Weergeven van generieke onderdelen (zoals een header)
- Initialiseren van functionaliteiten die maar één keer geïnitialiseerd mogen worden (Zie hoofdstuk 6.4 voor een uitleg over de functionaliteiten welke maar één keer geïnitialiseerd mogen worden)

6.3 Inladen van externe applicaties

Omdat de micro-applicaties op verschillende plaatsen gehost staan, dienen deze opgehaald te worden door de Wrapper, voordat deze getoond kunnen worden. Echter is het met de normale import syntax niet mogelijk om bestanden uit een URL op te halen, omdat deze door Webpack niet verwerkt kunnen worden. (Zie hoofdstuk 5.4.2 – Dependency injection met webpack voor een toelichting)

```
1) // Voor Webpack build
2) import("http://app-url")
3)
4) // Na Webpack build
5) __webpack_require__.x = "http://app-url"
```

Figuur 14: Webpack import syntax

SystemJS
SystemJS is een module loader met een eigen import syntax. Het is met SystemJS wel mogelijk om modules te importeren vanuit een URL en doordat deze los staat van de normale import syntax wordt deze ook niet door Webpack omgezet.

```
1) // Voor en na Webpack build
2) (window as any).System
3)     .import(app.url)
```

Figuur 15: SystemJS import syntax

6.4 Delen van functionaliteiten via libraries

Er zijn functionaliteiten die binnen meerdere applicaties gebruikt worden. Dit kunnen globale services zijn maar ook componenten zoals Buttons, welke op veel plaatsen terugkomen. Voor het delen van deze generieke functionaliteiten wordt gebruik gemaakt van libraries.

De functionaliteiten die gedeeld worden, kunnen verder opgedeeld worden naar 2 groepen op basis van de manier waarop deze geïntegreerd worden. De functionaliteiten worden opgedeeld naar de volgende libraries:

1. Core Library
De Core library bevat functionaliteiten die éénmaal geregistreerd moeten worden om vervolgens binnen de gehele applicatie gebruikt te kunnen worden. Dit zijn veelal singleton services zoals een authenticatie-service.

De Core library werkt als volgt:

1. Alle gedeelde services worden geregistreerd in de CoreModule die door de library geëxporteerd wordt. Ook worden de service definities geëxporteerd zodat deze binnen een applicatie gerefereerd kan worden.
2. De CoreModule wordt éénmaal geïmporteerd in de AppModule van de Wrapper-applicatie. Hiermee wordt gegarandeerd dat er één instantie aanwezig is binnen het domein van de applicatie.
3. Vervolgens kunnen de services binnen micro-applicaties gebruikt worden door deze op te vragen aan de dependency injector met de service-definitie van stap 1.

2. Shared Library

De Shared functionaliteiten bestaan veelal uit generieke componenten (denk aan buttons) die, in tegenstelling tot de Core functionaliteiten, altijd geïmporteerd moeten worden voordat deze gebruikt kunnen worden.

De Shared library werkt als volgt:

1. Alle generieke componenten worden geregistreerd in de SharedModule, welke door de library geëxporteerd wordt. Het is hierbij niet nodig om ook de component-definities te exporteren.
2. Vervolgens kan de SharedModule geïmporteerd worden in elke module die een generiek component nodig heeft. Deze is vervolgens beschikbaar binnen de scope van die module

6.5 Delen van gemeenschappelijke dependencies

Er zijn dependencies die binnen elke (micro-) applicatie gebruikt worden. Een voorbeeld hiervan zijn de dependencies welke het Angular Framework laden en de Core- en Shared library uit het vorige hoofdstuk. Deze dependencies zorgen tijdens het bundelproces voor meer dan de helft van de uiteindelijke bundelgrootte. Door deze dependencies extern aan te leveren kan de bundelgrootte aanzienlijk verminderd worden.

Externe dependencies configureren met Webpack externals

In hoofdstuk 5.2.4 wordt uitgelegd wat Webpack is en hoe Webpack ervoor zorgt dat de applicatiecode vertaald wordt naar code die door de browser begrepen wordt. Voor dit proces maakt Webpack gebruik van een configuratiebestand dat aangeeft hoe de verschillende stukken code verwerkt moeten worden. Dit configuratiebestand bevat een object, waarin verschillende regels en opties gedefinieerd kunnen worden.

Een van de configuratiemogelijkheden binnen Webpack is het aangeven van external dependencies (zie afbeelding 16). Met Webpack externals kan aangegeven worden dat de dependency niet mee gebundeld moet worden. Webpack bundelt deze dependency vervolgens niet mee, maar gaat er vervolgens vanuit dat de globaal beschikbaar wordt gesteld.

```
1) webpackConfig.externals = {
2)     '@angular/core': 'ng.core',
3)     '@angular/common': 'ng.common',
4)     '@angular/common/http': 'ng.common-http',
5)     '@angular/forms': 'ng.forms',
6)     '@angular/router': 'ng.router',
7)     '@angular/platform-browser': 'ng.platform-browser',
8)     '@angular/platform-browser-dynamic': 'ng.platform-browser-dynamic',
9)     '@angular/material': 'ng.material',
10)    '@biv/core': 'biv.core',
11)    '@biv/shared': 'biv.shared',
12)}
```

Figuur 16: Webpack externals configuratie

In afbeelding 16 is een voorbeeld te zien van hoe een Webpack externals configuratie eruitziet. Deze configuratie bestaat uit een object waarvan elke rij een external dependency aangeeft. Hierbij geeft de key aan welke dependency extern beschikbaar gesteld wordt. De value geeft plek aan waar deze dependency binnen de globale scope beschikbaar gesteld wordt.

Externe dependencies aanleveren
Om de applicatie te laten werken dienen de externe dependencies bij gebruik beschikbaar te zijn. Er is voor gekozen om deze verantwoordelijkheid bij de wrapper neer te leggen, omdat deze te allen tijden geladen is. De wrapper is dan ook de enige applicatie welke alle dependencies mee bundelt. Vervolgens plaatst deze wrapper de dependencies aan het globale variabele. (Zie afbeelding 17)

```
1) // Importeer alles van @angular/core en plaats dit in een variable ngCore
2) import * as ngCore from '@angular/core';
3)
4) // Stel ngCore binnen het window beschikbaar via de identifier ng.core
5) (window as any).define('ng.core', [], () => ngCore)
```

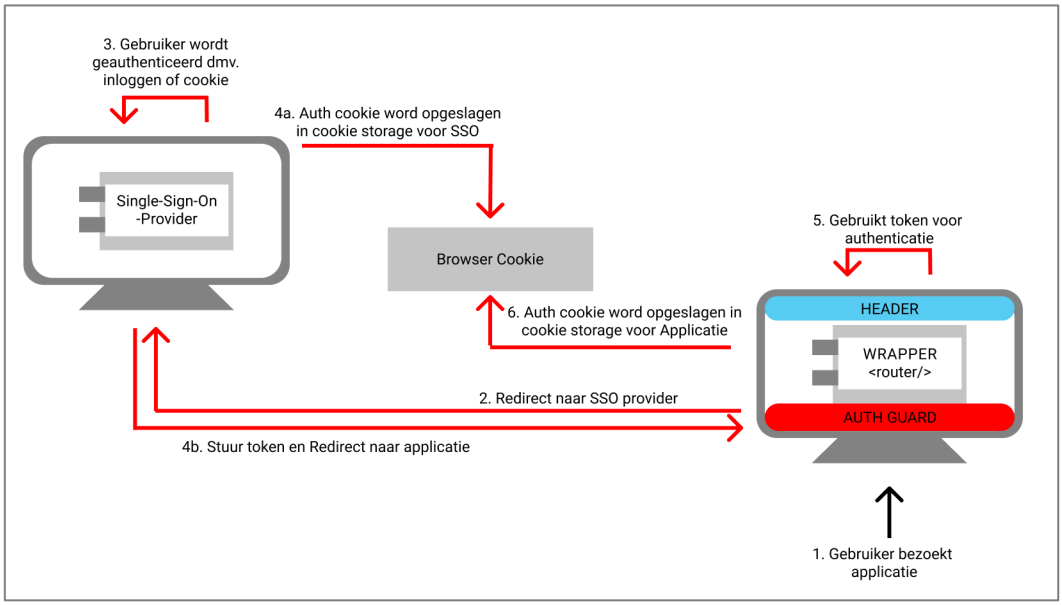
Figuur 17: Externe dependency beschikbaar stellen

Het nadeel van deze strategie is dat de wrapper en alle micro-applicaties dezelfde versies moeten draaien. Ook moet de wrapper opnieuw gebouwd worden wanneer een dependency toegevoegd of gewijzigd moet worden.

6.6 Authenticatie m.b.v. Single sign-on (SSO) provider

Voor het authentiseren van de gebruiker wordt gebruik gemaakt van een single sign-on identificatieproces. Bij single sign-on wordt het authentiseren van een gebruiker door de applicatie uitbesteed naar één centrale authenticatieprovider, waardoor deze logica gedeeld kan worden.

Authenticatieproces:
Het authenticatieproces zoals weergegeven in de afbeelding hieronder laat zien hoe een gebruiker zich authentiseert binnen één applicatiedomein. Dit komt doordat het Auth Cookie, dat als identificatie van de gebruiker geldt, uitgegeven wordt voor één applicatiedomein (stap 6). Andere applicaties dienen hetzelfde proces te doorlopen om ook een Auth Cookie voor hun domein te krijgen. De gebruiker hoeft echter niet steeds opnieuw in te loggen, omdat de SSO-provider ook in een Auth Cookie bijhoudt of de gebruiker al ingelogd is binnen zijn domein (de SSO-applicatie) (stap 4a).



Figuur 18: Single sign-on authenticatieproces

Toelichting op de stappen van het authenticatieproces:

1. De gebruiker bezoek de applicatie
2. De gebruiker wordt meteen ge-redirect naar de authenticatie pagina van de SSO- provider
3. Deze controleert of de gebruiker geauthentiseerd is door te kijken of er reeds een cookie, uitgegeven door de provider, aanwezig is. Indien nee dient de gebruiker in te loggen.
4. Na het inloggen wordt een Auth cookie aangemaakt dat beschikbaar is binnen het domein van de SSO. Ook wordt de gebruiker met token teruggestuurd naar de applicatie.
5. Vervolgens valideert de applicatie het teruggestuurde token bij de SSO provider. Op deze manier waarborgt de SSO provider de identiteit van de gebruiker.
6. Wanneer de SSO provider het token gevalideerd heeft, wordt ook een Auth cookie aangemaakt welke beschikbaar is binnen het domein van de applicatie. Hiermee wordt voorkomen dat het token telkens gevalideerd moet worden.

7 Implementatie

In dit hoofdstuk wordt het implementatieproces beschreven van een nieuwe microservice frontend-architectuur in de IkBenVIP-applicatie. De nieuwe architectuur is gebaseerd op de conclusie van het onderzoek naar strategieën om een microservice frontend-architectuur te implementeren (hoofdstuk 5) en het daarbij horende ontwerp (hoofdstuk 6).

Tijdens het onderzoek is naar voren gekomen dat er meerdere strategieën zijn om een microservice frontend-architectuur te implementeren. In de conclusie kan dan ook geen eenduidig antwoord gegeven worden maar zijn er twee kandidaten welke nagenoeg dezelfde mogelijkheden bieden; **Angular Libraries** (met run-time integratie) en **Single-SPA.**
In overleg met Topicus is besloten om voor beide strategieën een implementatie te maken, om zo de effecten en de prestaties van beide strategieën te kunnen meten en vergelijken.

7.1 Implementatie van Angular Libraries (met run-time integratie)

De eerste implementatie maakt gebruik van Angular libraries, welke tijdens de run-time geïntegreerd worden voor het opzetten van een microservice frontend-architectuur.

7.1.1 Project configuratie

In essentie bestaat elke micro-applicatie binnen deze strategie uit een 'standaard' Angular applicatie. Deze kan vervolgens, net als een gewone applicatie, gebouwd en getoond worden.

Om de applicatie ook als library te kunnen bouwen dient een extra configuratie toegevoegd te worden waarbij hetzelfde project nogmaals geconfigureerd wordt, maar dan als library. Hiermee wordt aangegeven dat dezelfde code gebruikt moet worden maar met een ander doel, het bouwen van een library.

In afbeelding 19 is een deel van zo'n configuratie te zien. In de configuratie is goed te zien dat beide projecten dezelfde bron delen voor hun code (sourceRoot).

```
1)  // Standalone configuratie
2)  "dagboek": {
3)      "projectType": "application",
4)      "root": "apps/dagboek",
5)      "sourceRoot": "apps/dagboek/src",
6)      "architect": {
7)          "build": { ... },
8)          "serve": { ... },
9)          "test": { ... }
10)     }
11) },
12)
13) // Library configuratie
14) "@dagboek/lib": {
15)     "projectType": "library",
16)     "root": "apps/dagboek",
17)     "sourceRoot": "apps/dagboek/src",
18)     "prefix": "biv",
19)     "architect": {
20)         "build": { ... },
21)         "test": { ... }
22)     }
23) }
24)
```

Figuur 19: Micro-frontend library configuratie

7.1.2 Bouwen van een micro-applicatie

Doordat in de extra configuratie aangegeven is dat het project een library betreft, zal Angular ook proberen om de code te bouwen naar een library. Om een library te kunnen bouwen is nog wel een extra bestand nodig die de structuur van de library aangeeft, namelijk **ng-package.json**.

```
6)  {
7)    "$schema": "../../node_modules/ng-packagr/ng-package.schema.json",
8)    "dest": "../../dist/libs/dagboek",
9)    "lib": {
10)     "entryFile": "src/public_api.ts"
11)   }
12) }
```

Figuur 20: ng-package.json configuratie

Ng-package.json is een configuratiebestand welke aangeeft hoe een library gebouwd moet worden. Hierbij is voornamelijk het 'entryFile' veld belangrijk, omdat deze het startpunt van de library aangeeft. Tijdens het bouwproces zal de code vanuit dit punt doorlopen worden om hier vervolgens een bundel van te maken. Met het startpunt wordt ook voorkomen dat logica die nodig is voor het zelfstandig draaien van de applicatie mee gebundeld wordt. Er worden dus alleen functionaliteiten gebundeld welke nodig zijn voor het weergeven van dat applicatiedeel.

7.1.3 Samenvoegen van micro-applicaties

Voor het inladen van micro-applicaties wordt gebruik gemaakt van de lazy-loading feature van Angular. Hierbij kan gebruik gemaakt worden van de importfunctie van SystemJS voor het dynamisch importeren van een micro-applicatie uit een externe bron (zie hoofdstuk 6.3).

```
13)// Normal lazy-loading
14){
15)  path: 'dagboek',
16)  loadChildren: () => import('../dagboek.module')
17)                              .then(m => m.DagboekModule)
18)}
19)
20)// SystemJS lazy-loading
21){
22)  path: 'dagboek',
23)  loadChildren: () => (window as any).System
24)                          .import('http://localhost:4200/dist/dagboek.bunde.js')
25)                          .then(m => m.DagboekModule)
26)}
```

Figuur 21: Normale vs SystemJS import syntax

7.1.4 Opmerkingen tijdens de implementatie van Angular Libraries

Tijdens de implementatie van Angular Libraries met run-time integratie zijn een aantal dingen naar voren gekomen die ervoor zorgen dat deze strategie niet zonder meer gebruikt kan worden binnen een productieomgeving. De volgende opmerkingen zijn naar voren gekomen:

- **Angular Libraries kunnen geen dependencies bundelen**

Het eerste probleem is dat het met de huidige versie van Angular niet meer mogelijk is om dependencies mee te bundelen met een library. Dit betekent dat alle dependencies globaal aangeleverd moeten worden, zonder dat er verdere afspraken over gemaakt zijn.
Het gevolg hiervan is dat de wrapper alle dependencies globaal beschikbaar moet stellen, waarbij rekening gehouden moet worden met de versies die de verschillende micro-applicaties nodig hebben. Hierbij wordt het onvermijdelijk dat het ontwikkelteam eerst de wrapper moet updaten, wanneer zij een nieuwe dependency of versie willen toevoegen aan een micro-applicatie.

- **Er is geen ondersteuning meer voor het maken van productie-build**

Het tweede, en meest ingrijpende, probleem is dat het bij de Angular Libraries strategie niet langer mogelijk is om een productie-build te maken. Dit komt omdat Angular bij het maken van een productie-build ook verschillende optimalisatie-stappen uitvoert op de gehele code.
Doordat de libraries binnen deze strategie hun eigen optimalisatie-proces uitvoeren sluiten deze niet meer aan op de wrapper, die op zijn eigen manier geoptimaliseerd is. Dit kan ondervangen worden met een niet-productie-build waarbij deze stap niet uitgevoerd wordt en de verschillende applicatie-onderdelen wel op elkaar zullen aansluiten.

Het niet kunnen maken van een productie-build is funest voor deze strategie omdat men binnen een productieomgeving gebruik wil maken van een productie-build en de prestatiewinsten (zie snellere laadtijd) die daarbij te behalen zijn.

7.2 Implementatie van Single-SPA

De tweede implementatie maakt gebruik van Single-SPA voor het opzetten van een microservice front-end architectuur.

7.2.1 Project configuratie

Om een applicatie gereed te maken voor gebruik binnen Single-SPA, moet deze verpakt te worden als Single-SPA-applicatie.

Voor Angular kan gebruik gemaakt worden van de **single-spa-angular** package om de applicatie te verpakken. Deze package heeft een 'wrapper'-functie voor het verpakken van een Angular applicatie als Single-SPA-applicatie. Binnen deze functie kan de logica geplaatst worden welke normaal de applicatie start. (zie afbeelding 22)

```
1) // Applicatie starten in de standaard situatie
2) platformBrowserDynamic().bootstrapModule(AppModule)
3)   .catch(err => console.error(err));
4)
5) // Single-SPA plaatst een wrapper om deze functie heen
6) const lifecycles = singleSpaAngular({
7)   bootstrapFunction: singleSpaProps => {
8)     singleSpaPropsSubject.next(singleSpaProps);
9)     return platformBrowserDynamic().bootstrapModule(AppModule);
10)  },
11)  template: '<dagboek-root />',
12)  Router,
13)  NgZone: NgZone,
14)});
15)
16)export const bootstrap = lifecycles.bootstrap;
17)export const mount = lifecycles.mount;
18)export const unmount = lifecycles.unmount;
```

Figuur 22: de Single-SPA wrapper wordt om de bootstrapfunctie van Angular geplaatst

Door het plaatsen van deze 'wrapper'-functie komen een drietal lifecyle functies beschikbaar gesteld waarmee de applicatie beheerd kan worden. De volgende functies worden geleverd:

bootstrap()	Start de applicatie, wordt alleen aangeroepen wanneer de applicatie voor het eerst getoond moet worden.
mount()	Toont de applicatie en zorgt ervoor dat de juiste onderdelen getoond worden a.d.h.v. de huidige URL.
unmount()	Verbergt de applicatie. De applicatie draait onderwater nog wel maar alle elementen en variabelen worden verwijderd.

Doordat de lifecyle functies ook geëxporteerd worden kunnen deze ook aangeroepen worden van buitenaf. De wrapper maakt hier gebruik van om de applicatie te kunnen starten, tonen en verbergen.

7.2.2 Bouwen van een micro-applicatie

Omdat de applicatie niet meer op zichzelf staat, maar geconfigureerd is als Single-SPA- applicatie, moeten er een aantal wijzigingen gemaakt worden in de bouwconfiguratie.

Bouwen naar één bundel
Single-SPA verwacht dat een applicatie wordt aangeleverd als één code-bundel. Normaliter worden voor een Angular applicatie meerdere bestanden aangemaakt. Binnen Webpack kan dit geconfigureerd worden onder het veld 'entries'. Door alle entries te vervangen door één entry (main entry) wordt Webpack geïnstrueerd om alle code te bouwen naar één bundel.
Ook wordt in de bouwconfiguratie aangegeven het bestand met de Single-SPA wrapper functie het startpunt van de applicatie is. Hierdoor wordt de applicatie als Single-SPA-applicatie gebouwd, waarvan de geëxporteerde lifecyle- functies ook buiten de bundel beschikbaar worden.

Bundel in UMD-format
Ook verwacht Single-SPA een format wat binnen de browser gebruikt kan worden. Hiervoor wordt in de configuratie aangegeven dat de code gebouwd moet worden in UMD[5]-format. Door het maken van een UMD-bundel kan deze binnen de browser gebruikt worden.

In afbeelding 23 is te zien hoe in de configuratie aangegeven wordt dat er één bundel gemaakt moet worden (main entry point) in UMD-format (libraryTarget).

```
1) // Normal webpack config
2) {
3) mode: 'production',
4) context: '/.../dagboek',
5) entry: {
6)   main: [
7)     '/.../dagboek/src/main.ts'
8)   ],
9)   polyfills: [
10)    '/.../dagboek/src/polyfills.ts'
11)  ],
12)  styles: [
13)    '/.../dagboek/src/styles.scss'
14)  ]
15)},
16)output: {
17)    libraryTarget: 'var',
18)    filename: '[name].[chunkhash].js'
19)}
```

```
1) // Single-SPA webpack config
2) {
3) mode: 'production',
4) context: '/.../dagboek',
5) entry: {
6)   main: [
7)     '/.../dagboek/src/styles.scss',
8)     '/.../dagboek/src/main.s-spa.ts'
9)   ]
10)},
11)output: {
12)    library: 'app3',
13)    libraryTarget: 'umd',
14)    filename: '[name].bundle.js',
15)}
16)}
17)
```

Figuur 23: Webpack configuratie, normaal en voor Single-SPA

[5] Universal Module Definition (UMD) is een format welke een brede ondersteuning biedt voor verschillende vormen van Javascript (AMD & CommonJS)

Standalone mode

Voor het zelfstandig kunnen bouwen en draaien van de applicatie is een extra configuratie toegevoegd, dat een ander startpunt van de applicatie aangeeft (zie Afbeelding 24). In deze standalone configuratie wordt het startpunt genomen welke niet verpakt is door Single-SPA. Hierdoor kan de applicatie als normaal gestart worden.

```
"configurations": {
    "production": { ... },
    "standalone": {
        "main": "src/main.ts"
    }
}
```

Figuur 24: De standalone configuratie vervangt de Single-SPA wrapper door de normale startlogica

Met een check (zie afbeelding 25) kan a.d.h.v. het aangegeven startpunt bepaald worden welke Webpack configuratie gebruikt moet worden. Op deze manier kan de applicatie gebouwd worden om zelfstandig te kunnen draaien, of voor gebruik binnen Single-SPA.

```
const isStandaloneMode = (options) => (
    options.main === 'src/main.ts';
)
```

Figuur 25: Check voor bepalen of de applicatie in standalone mode moet draaien

7.2.3 Samenvoegen van micro-applicaties

Voor het samenvoegen van Single-SPA-applicaties wordt gebruikt gemaakt van Single-SPA parcels. Met Single-SPA parcels kan een Single-SPA-applicatie ingeladen worden binnen een andere applicatie. Hierbij moet nog wel handmatig bepaald worden wanneer een applicatie ingeladen, getoond of verborgen moet worden.

SpaHostComponent

Voor het inladen van Single-SPA parcels is een generiek component gemaakt, het SpaHostComponent. Binnen dit component wordt de Single-SPA-applicatie opgehaald vanuit zijn externe bron, deze kan vervolgens in het component geplaatst worden met de geëxporteerde bootstrap() en mount() functies. Het ophalen en plaatsen van de Single-SPA-applicatie wordt uitgevoerd tijdens de initialisatie van het component (ngOnInit), waardoor wachttijden beperkt worden. Met de unmount() functie kan vervolgens weer gezorgd worden dat de applicatie verborgen wordt wanneer het component vernietigd wordt.

Laden op basis van URL

Doordat de logica die Single-SPA-applicaties inlaadt, verpakt is in een component, kunnen Single-SPA-applicaties gewoon opgenomen worden in de router-configuratie van een component.
(zie Afbeelding 26)

```
{
  path: 'dagboek',
  children: [{
    path: '**',
    component: SpaHostComponent,
    data: { app: 'dagboek'}
  }]
}
```

Figuur 26: Router-configuratie voor Single-SPA-applicatie

Binnen de configuratie wordt het Component niet direct gekoppeld aan de route. In plaats daarvan wordt een nieuwe (child) route gedefinieerd, welke hoort bij de prefix ‘dagboek’.

Met deze configuratie worden bij alle routes die beginnen met ‘/dagboek’ het SpaHostComponent getoond. Met de route-parameters kan vervolgens aangegeven worden welke Single-SPA-applicatie geladen moet worden.

7.2.4 Opmerkingen tijdens de implementatie van Single-SPA

Tijdens de implementatie van Single-SPA zijn een aantal opmerkingen naar voren gekomen die ervoor zorgen dat deze strategie niet zonder meer gebruikt kan worden binnen een productieomgeving. De volgende opmerkingen zijn naar voren gekomen:

- **Applicaties moeten met dezelfde configuratie gebouwd worden wanneer de Angular packages onderling gedeeld worden**

Wanneer Single-SPA-applicaties onderling de Angular packages delen moeten zij ook gebruik maken van dezelfde bouwconfiguratie. De reden hiervoor is dat er tijdens de initialisatie van een Angular applicatie een platform wordt geconfigureerd waarop de applicatie draait. Wanneer Angular packages (met ook het platform) onderling gedeeld worden tussen applicaties die niet dezelfde configuratie delen, wordt een foutmelding getoond.

Deze opmerking is niet zozeer een belemmering, maar meer een aandachtspunt. Tijdens het ontwikkelen kan het voorkomen dat bepaalde applicaties in ontwikkelmodus staan terwijl andere applicaties (extern) in productiemodus draaien. In dit geval kunnen dependencies onderling niet gedeeld worden.

- **Routing-probleem zorgt ervoor dat Single-SPA nog niet klaar is voor productie.**

Binnen de Single-SPA community is er op dit moment een bekend probleem waarbij het navigeren tussen applicaties soms niet werkt wanneer gebruik gemaakt wordt van lazy-routing. Dit probleem dient eerst verholpen te worden, alvorens deze strategie gebruikt kan worden binnen een productieomgeving.

8 Prestatiemetingen

In dit hoofdstuk worden de prestaties van de geïmplementeerde strategieën met elkaar vergeleken. Ook wordt er een 0-meting gedaan van het originele prototype van de IkBenVIP-applicatie. De resultaten hiervan kunnen vervolgens gebruikt worden bij het inzichtelijk maken van de performance-impact van de gekozen strategie.

Voor het meten van de prestaties van een strategie wordt gebruik gemaakt van de volgende meetwaarden:

Meetwaarde	**Beschrijving**
Bundelgrootte	Grootte van de uiteindelijke applicatiebundel(s) in kB. Een grote bundle heft impact op de laadtijd en prestaties.
Load time	De tijd, gemeten in milliseconden (ms), die nodig is voor het inladen van applicaties. De laadtijd is relevant voor de gebruikerservaring.
Largest Contentful Paint (LCP)	De tijd (in ms) tussen het navigeren naar een pagina en het moment waarop het 'grootste' deel van de content getoond wordt. Dit is het moment waarop de pagina voor de gebruiker 'bruikbaar' wordt.

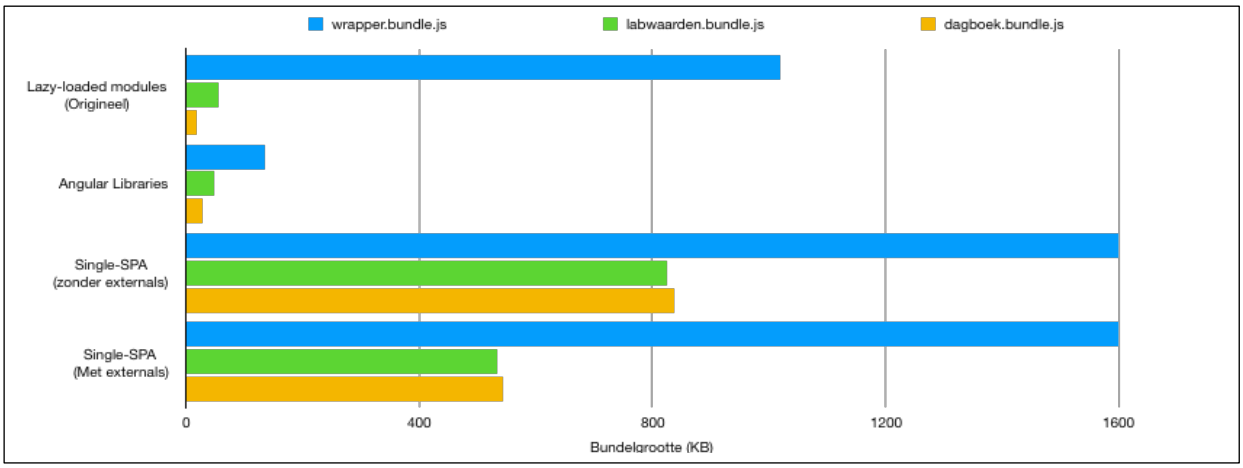
De metingen werden verricht bij onderstaande (micro-)applicatie-bundels:

(Micro-)applicatie	**Beschrijving**
Wrapper	De main bundle van de applicatie waarin alle applicatieonderdelen samenkomen.
Labwaarden	De bundle waarin alle functionaliteiten van het applicatie-onderdeel 'labwaarden' staan
Dagboek	De bundle waarin alle functionaliteiten van het applicatie-onderdeel 'dagboek' staan

De resultaten van deze metingen staan in de volgende paragrafen.

8.1 Bundelgrootte

Als eerste heb ik de groottes van de verschillende applicatiebundels met elkaar vergeleken. Een grotere bundel betekent, dat er meer volume gedownload moet worden, wat impact heeft op laadtijden en prestaties.



Figuur 27: Bundelgrootte van de applicatie

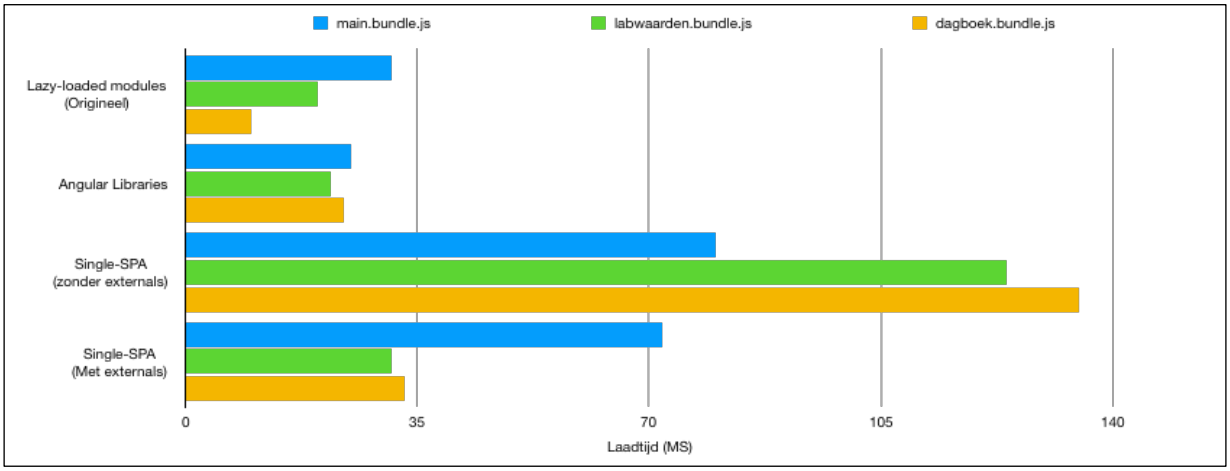
In het algemeen valt op dat de grootte van de hoofdbundel (wrapper.bundle.js) significant groter is, dan de bundels van een applicatiedomein. Dit komt doordat de hoofdbundel ook de basis-functionaliteiten bevat die nodig zijn voor het kunnen draaien van de applicatie.

Wanneer gekeken wordt naar de grootte van de bundels uit de Angular Libraries-strategie, valt op dat:
- de bundelgroottes van de “micro-applicaties” bij de Library-strategie kleiner zijn dan die van de Single-SPA-strategie en bijna gelijk aan het origineel. Dat is te verklaren omdat met libraries alleen delen code van elkaar gescheiden worden. De hoeveelheid code blijft hierbij, ook na het bundelen gelijk.
- Het is binnen deze strategie niet meer mogelijk is om gebruik te maken van de geoptimaliseerde productiebundel (2MB). Hoewel de niet-geoptimaliseerde bundel (137KB) een stuk kleiner is, duurt verwerken toch langer.

Voor de met Single-SPA gebundelde applicatie delen, is te zien dat:
- de bundelgrootte van de hoofdbundel in verhouding met de andere strategieën duidelijk groter is. Dit komt door de extra logica die voor Single-SPA nodig is om de micro-applicaties te beheren.
- de Single-SPA bundels van het dagboek- en labwaarden applicatiedeel circa tien keer groter zijn dan de standaard (lazy loaded) modules. Dit komt doordat elke bundel een volledige Angular applicatie bevat. Het scheelt in grootte wel bijna 300KB wanneer de dependencies van Angular extern aangeleverd worden.
- gebruik maken van webpack-externals bij Single-SPA voor een bundel zorgt, die 25% kleiner is dan zonder externals.

8.2 Laadtijden

Naast de grootte van de applicatiebundel(s) heb ik ook gekeken naar de tijd die nodig is voor het ophalen en verwerken van een applicatiebundel. Deze loopt een beetje gelijk op met de bundelgrootte. De laadtijd is relevant voor de impact op de gebruikerservaring: niemand wil graag wachten!



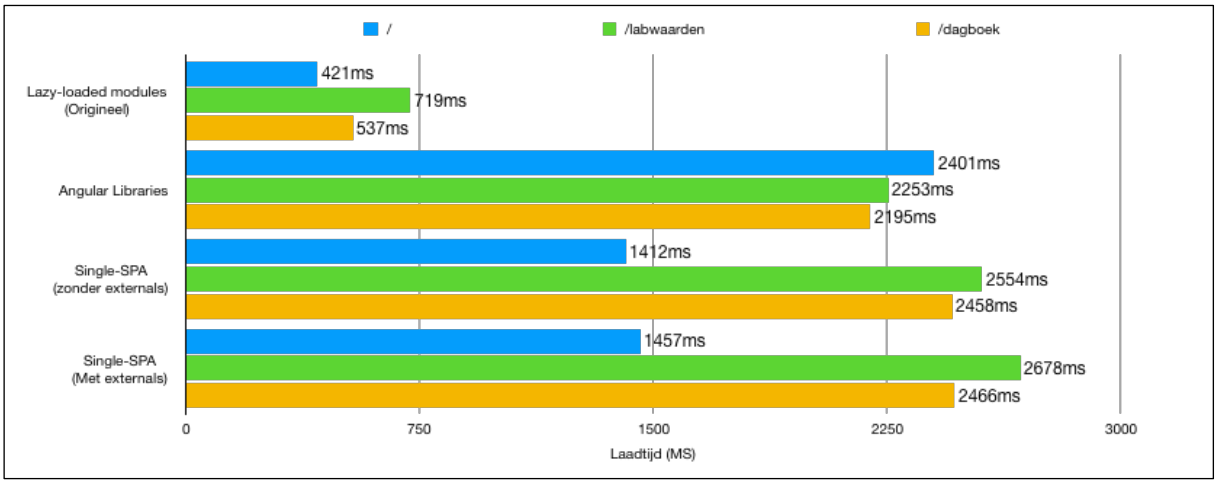
Figuur 28: Laadtijden van de applicatie

Zoals te verwachten is de laadtijd navenant aan de bundelgrootte; de grootste bundels kosten dan ook de meeste tijd om in te laden. Hierbij blijven de laadtijden wel tussen de 30 en 150ms, wat nog acceptabel is.

Verder is te zien dat voor Single-SPA:
- het gebruik van webpack-externals zorgt voor een significant kortere laadtijd. Voor de micro-applicaties loopt dit op tot een 75% kortere laadtijd.

8.3 Largest Contentful Paint (LCP)

Largest Contentful Paint (LCP) meet het moment waarop het 'grootste' deel van de content getoond wordt. Deze waarde is belangrijk omdat dit het moment is waarop de pagina voor de gebruiker bruikbaar wordt.

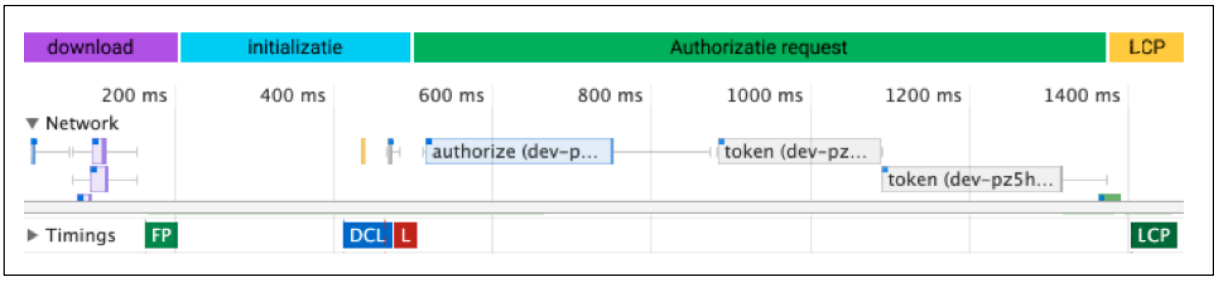


Figuur 29: LCP van de applicatie

Bij de Angular Libraries strategie valt ook op dat de LCP -waarde aanzienlijk hoger ligt dan de andere strategieën. Dit komt doordat de Library-strategie geen geoptimaliseerde productie-build gemaakt kan worden. De niet-geoptimaliseerde bundel mag dan wel kleiner zijn dan de geoptimaliseerde bundel; het verwerken van de bundel binnen de browser duurt een stuk langer.

Impact van SSO-authenticatie op de laadtijden

Ook werd getest hoeveel extra tijd het kost als een autorisatieverzoek in het proces is opgenomen. Het resultaat daarvan staat in onderstaand figuur 31.



Figuur 30: Tijdlijn applicatie laden met authorisatieverzoek

In het algemeen is te zien dat de content later getoond wordt (LCP) dan bij het origineel. Dit komt doordat elke bundel een autorisatie-request verstuurt, dat ongeveer 1 seconde duurt. (zie de groene balk in figuur 31). De content wordt pas getoond wanneer dit authorisatieverzoek afgehandeld is. Wanneer men bijvoorbeeld de labwaardenpagina wil bezoeken, waarvoor ook de wrapper geladen dient te worden, worden onderwater twee authorisatieverzoeken verstuurd, waardoor een extra 2 seconde toegevoegd wordt voordat de content getoond wordt.

9 Conclusies en aanbevelingen

Het doel van mijn project was om erachter te komen wat de beste strategie is om de huidige frontend-architectuur van de 'IkBenVIP'-applicatie, gebouwd in Angular, op te delen naar een microservice architectuur.

Tijdens het onderzoek heb ik gekeken naar de verschillende mogelijkheden, die er zijn om de huidige applicatie op te splitsen naar een microservice architectuur. Hierbij werd al snel bekend dat er geen éénduidig antwoord kan worden gegeven op die vraag. Het antwoord hangt sterk af van de eisen, wensen en beperkingen binnen het bedrijf.
Op basis van een vijftal vragen over de eisen en de requirements, die ik samen met Topicus heb opgesteld, ben ik tot de conclusie gekomen dat een implementatie met Angular Libraries of Single-SPA mogelijkheden bied om de bestaande applicatie op te splitsen.

Tijdens de implementatie van bovenstaande strategieën kwam ik er echter al snel achter dat het implementeren van een microservice architectuur binnen Angular op deze wijze problematisch is. De reden hiervoor is dat Angular een 'totaaloplossing' levert voor het maken van webapplicaties, waarbij de applicatiefunctionaliteiten onlosmakelijk verbonden zijn. Wanneer functionaliteiten toch losgehaald worden, ontstaan er conflicten en problemen tijdens het bouwen en integreren van micro-applicaties. Eigenlijk is dit een weg, die je in Angular niet wil bewandelen.

Wanneer men toch een microservice architectuur wil realiseren binnen een Angular applicatie, dan zou ik de volgende aanbevelingen willen doen:

- Binnen de onderzochte strategieën zou mijn advies zijn om de mogelijkheden van een implementatie met Hyperlinks nader te onderzoeken. Het voordeel van deze oplossing is dat deze buiten Angular om werkt en zich niet mengt in de werking van Angular.
- Single-SPA is nog niet klaar zijn voor productie vanwege een aantal problemen, deze zijn bekend binnen de Single-SPA community. Wanneer deze problemen verholpen zijn kan Single-SPA wel degelijk uitkomst bieden bij het implementeren van een micro-frontend.
- Afhankelijk van de requirements binnen het bedrijf, kan een MonoRepo-structuur een uitkomst bieden. Binnen een MonoRepo kunnen applicatiedelen van elkaar gescheiden en georganiseerd worden. Ook kunnen applicatieonderdelen afzonderlijk gebouwd worden.
  Omdat het niet mogelijk is om op deze manier applicatie-onderdelen afzonderlijk te deployen, past deze optie niet binnen het idee van een micro-frontend.
- De volgende versie van Angular gaat hoogstwaarschijnlijk gebruik maken BazelJS, een nieuwe buildtool voor het bouwen van Angular applicaties. Angular belooft hiermee enorme tijdswinsten bij het bouwen van applicaties en dit kan potentieel een oplossing bieden wanneer de bouwtijd van een applicatie een issue is.
- Veel van de problemen tijdens de implementatie van de microservice architectuur zijn het gevolg van het gebruik van Angular. Een deel van deze problemen komen niet voor wanneer een oplossing als React of Vue gebruikt wordt. Wanneer de behoefte aan een micro-frontend blijft bestaan, zou een oplossing welke gebruik maakt van een ander framework overwogen kunnen worden.
- Tot slot moet afgevraagd worden of er een echte noodzaak is voor een microservice frontend-architectuur. Het opzetten van een microservice architectuur binnen de frontend is zeer complex en eigenlijk alleen waardevol voor hele grote applicaties, waar met meerdere teams tegelijk aan gewerkt wordt. Voor minder omvangrijke applicaties en/of kleinere teams weegt de extra complexiteit niet op tegen de voordelen die een microservice architectuur biedt, zeker wanneer gekeken wordt naar de mogelijkheden van andere oplossingen, zoals een MonoRepo.

10 Bibliografie

Angular. (sd). *Angular docs*. Opgehaald van Angular: https://angular.io/docs

Angular. (sd). *Architecture overview*. Opgehaald van Angular: https://angular.io/guide/architecture

Angular University. (sd). *how does angular change detection really work?* Opgehaald van Angular University: https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/

AngularInDepth. (sd). *Building a extensible dynamic pluggable enteprise application in Angular*. Opgehaald van Angular in depth: https://blog.angularindepth.com/building-extensible-dynamic-pluggable-enterprise-application-with-angular

Denning, J. (sd). *single-spa docs*. Opgehaald van Single-SPA: https://single-spa.js.org/docs/

Fowler, M. (sd). *Microservices*. Opgehaald van MartinFowler: https://martinfowler.com/

Geers, M. (sd). *Micro Frontends*. Opgehaald van Micro Frontends: https://micro-frontends.org/

Huang, P. (sd). *Micro-frontend Architecture in Action with six ways*. Opgehaald van Dev.to: https://dev.to/phodal/micro-frontend-architecture-in-action-4n60#comparison-of-micro-front-end-solutions-complex-ways

InDepth. (sd). *Hooking into the Angular bootstrap process* . Opgehaald van InDepth: https://indepth.dev/hooking-into-the-angular-bootstrap-process/

Joeldenning. (sd). *Lazy loaded routes cause problems #102*. Opgehaald van Github: https://github.com/CanopyTax/single-spa-angular/issues/102

NX. (sd). *Extensible Dev Tools for Monorepos*. Opgehaald van NX: https://nx.dev/web

Pattern: Microservice Architecture. (sd). Opgehaald van microservices.io: https://microservices.io/patterns/microservices.html

Pattern: Monolithic Architecture. (sd). Opgehaald van Microservices.io: https://microservices.io/patterns/monolithic.html

Richardson, C. (sd). *MicroService UI patterns*. Opgehaald van MicroServices.IO: https://microservices.io/patterns/ui

SoftwareArchitekt.at. (sd). *It's about cutting your domain, not (first and foremost) about micro frontends!* Opgehaald van SoftwareArchitekt: https://www.softwarearchitekt.at/aktuelles/its-about-cutting-your-domain-not-first-and-foremost-about-micro-frontends/

Topicus. (sd). *Over Topicus*. Opgehaald van Topicus: werkenbijtopicus.nl/over-topicus

Webpack. (sd). *Docs*. Opgehaald van Webpack: https://webpack.js.org/concepts/

Zone.JS. (sd). *Zone.js*. Opgehaald van Github: https://github.com/angular/angular/tree/master/packages/zone.js/

11 Begrippenlijst

API	Application Programming Interface: een verzameling van definities op basis waarvan een computerprogramma kan communiceren met een ander programma. Veelal gebruikt om een scheiding te vormen tussen verschillende lagen van abstractie.
devOps	Gebruik binnen software engeering wat zich voornamelijk bezighoud met automatisering en monitoring van alle onderdelen bij het bouwen van software, integratie, testen, release en deployment.
Javascript	Een veelgebruikte scripttaal om webpagina's interactief te maken. Het script wordt door middel van HTML overbracht in de webbrowser en wordt hierin uitgevoerd.
micro-frontend	een architectuur waarin één grote applicatie gereduceerd is tot meerdere kleine stukjes/applicaties**.**
micro-applicatie(s)	Een van de kleine applicaties binnen een microfrontend-architectuur
microservice	Software ontwikkelingstechniek waarbij een applicatie bestaat uit een verzameling lossse diensten (services).
mock-data	dummy data, waarmee de werking van een object op gecontroleerde wijze kan worden gesimuleerd.
model-view-controller (MVC)	ontwerppatroon dat het ontwerp van complexe toepassingen opdeelt in drie eenheden met verschillende verantwoordelijkheden: datamodel, datapresentatie en applicatielogica.
monolithisch	een architectuur waarin alle applicatiecode gezamelijk geplaatst staat
NPM-registry	Een database van (open source) JavaScript packages.
REST	Representational State Transfer is een communicatiestandaard voor de communicatie tussen systemin. Deze wordt veelal gebruikt bij de communicatie tussen client en server.
SPA	Single Page Application

Bijlage A: Overzicht modules ‘IkbenVIP’

In deze bijlage staat een overzicht van de verschillende modules die aanwezig zijn binnen het prototype van de IkbenVIP applicatie van Topicus, deze staan gegroepeerd op soort module.

Root Module:

App	Hier worden alle andere modules geïmporteerd en gebruikt om de applicatie te vormen.

Domain Feature Modules:
Onderstaande modules leverende gebruikerservaring voor een deel of domein van de applicatie. De export hiervan bestaat uit één pagina/component.

Gesprekken	Bevat alle componenten en logica die nodig zijn voor het weergeven van gesprekken
Login	Bevat alle componenten en services die nodig zijn voor het inloggen
Dagboek	Bevat alle componenten en logica die nodig zijn voor het weergeven en wijzigen van het dagboek
Startpagina	Bevat alle componenten en logica die nodig zijn voor het weergeven van het menu.

Widget Feature Modules:
Onderstaande modules leveren componenten en functionaliteiten welke gebruikt worden in één of meerdere andere modules.

Contacten	Bevat alle componenten en logica die nodig zijn voor het weergeven van de contactenlijst
Shared	Bevat componenten en logica welke door meerdere modules gedeeld worden
Menu	Bevat componenten en logica welke gebruikt worden voor het weergeven van het menu

Service Feature Modules:
Onderstaande modules leveren Services welke gebruikt worden in één of meerdere andere modules.

Toegang	Bevat alle logica welke nodig is voor het beheren van toegang en rollen binnen de applicatie

Routed Feature Modules:
Net als Domain Feature Modules leveren onderstaande modules ook gebruikerservaring voor een deel of domein van de applicatie. In tegenstelling tot Domain Feature Modules worden hier Route configuraties geëxporteerd.

Labwaarden	Bevat alle componenten en logica die nodig zijn voor het weergeven van labwaarden
Meetwaarden	Bevat alle componenten en logica die nodig zijn bij het weergeven en invoeren van dag curves en lichaamsgewicht
Profiel	Bevat alle componenten en logica die nodig is bij het weergeven en wijzigen van het gebruikersprofiel
Zorgnetwerk	Bevat alle componenten en logica die nodig zijn bij het tonen van het zorgnetwerk

Bijlage B: Requirements

Zelfstandigheid	Prio (MoSCoW)
Applicaties zijn zelf verantwoordelijk voor hun data, componenten en subjectieve styling. (Draaien in hun eigen applicatie context)	Must
Is het mogelijk om meerdere microfrontends op een pagina te plaatsen. (Widgets)	Should

Developer Experience

Nieuwe applicaties kunnen zero-config toegevoegd worden	Could
De beoogde oplossing kan met minimale wijzigingen in de 'core functionallity' geïmplementeerd worden.	Should

Bouwen:

Applicaties kunnen onafhankelijk van elkaar gebouwd worden zonder bijkomen van andere ontwikkelteams.	Must
Gemeenschappelijke dependencies worden centraal beschikbaar gesteld.	Should

Deployen:

Applicaties kunnen onafhankelijk van elkaar gedeployed worden zonder bijkomen van andere ontwikkelteams.	Should
Applicaties kunnen (evt. met behulp van een wrapper) afzonderlijk gedraaid worden	Must

Testen:

Applicaties kunnen afzonderlijk voorzien worden van Unit-tests	Must
Applicaties kunnen in zijn geheel (verticaal[6]) voorzien worden van E2E-tests	Must

User Experience:

Er is een globaal stijlelement aanwezig waarin de huisstijl (kleuren, groottes en basiselementen) gedefinieerd is	Must
De applicatie in zijn geheel is een SPA of wordt door de gebruiker als een SPA ervaren.	Must
Applicaties zijn compatible met de laatste versie van de 'grootste' browsers	Must

[6] Met verticaal wordt bedoeld van front- tot back-end voor een bepaalde 'feature' of domein

Bijlage C: Architectuurplaat nieuw