Afstudeerverslag

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
14-04-2020
Versie 1.0

Referaat

Performance optimalisatie bij PostNL

Onderwijsinstelling

Onderwijsinstelling:	De Haagse Hogeschool Afdeling IT en Design
Eerste beoordelaar:	De heer D.L. Kam
Tweede beoordelaar:	De heer E.M van Doorn

Opdrachtgever

Afstudeerbedrijf:	Nalta Software B.V.
Opdrachtgever: Functie:	De heer G. Merkelbach CEO
Contactpersoon Functie:	De heer M. Beenes Lead-developer

Student

Naam:	Koen Cornelis Jacobus van Zeijl
Studentennummer:	16067614

Data

Afstudeerperiode	10-02-2020 tot 05-06-2020
Inleverdatum	05-06-2020

Voorwoord

Voor u ligt het afstudeerverslag “Performance optimalisatie bij PostNL” geschreven door Koen van Zeijl, 4e jaars student Software Engineering De Haagse Hogeschool. In dit verslag heb ik de werkzaamheden voor het afronden van mijn afstudeeropdracht beschreven. Ik heb deze afstudeeropdracht uitgevoerd bij het bedrijf Nalta. Nalta is een internetbedrijf welke oplossingen biedt voor elke digitale transformatie.

Ik heb voor deze afstudeeropdracht gekozen omdat het een interessante en uitdagende opdracht leek te zijn en raakvlakken had met verschillende IT-gebieden. Tijdens het lezen van dit afstudeerverslag krijgt u een goed beeld van mijn afstudeeropdracht. Ik beschrijf mijn onderzoeken, de keuzes die ik heb gemaakt en mijn aanbevelingen.

Mijn dank wil ik graag uitspreken naar mijn bedrijfsmentor, de heer Martijn Beenes heeft mij begeleid tijdens het uitvoeren van de afstudeeropdracht en stond gedurende deze periode altijd voor mij klaar. Daarnaast wil ik alle andere collega’s van Nalta bedanken voor het beantwoorden van mijn gespecialiseerde vragen tijdens mijn onderzoeksfase.

Ook wil ik mijn dank uitspreken naar mijn afstudeerdocent de heer van Doorn van De Haagse Hogeschool. De heer van Doorn heeft mij in lastige tijden als deze corona periode zo goed mogelijk begeleid en schakelde snel om daar digitale vormen om de begeleiding door te zetten. Ik heb van de heer van Doorn bruikbare feedback gekregen om mijn documenten te verbeteren.

Ik wens u veel leesplezier toe.

Wateringen, 1 juni 2020

Koen van Zeijl

Inhoudsopgave

1. Inleiding

In dit verslag leest u hoe ik de afgelopen 17 weken gewerkt heb aan mijn afstudeeropdracht bij Nalta Software BV. Verder in dit verslag zal Nalta Software BV. Nalta genoemd worden. Nalta is een internetbedrijf, welke oplossingen biedt voor elke digitale transformatie. Dit varieert van het bedenken van strategieën tot en met de implementatie van de daadwerkelijke oplossingen. Hiervoor ontwerpt en ontwikkelt Nalta responsive websites, online platformen, en applicaties. Eén van de belangrijkste klanten van Nalta is PostNL, omdat zij voor een groot deel van de omzet zorgt. Hiervoor ontwikkelt Nalta onder andere het internationale track and trace klantenportaal. Dit klantenportaal is het project waar ik mij op heb gericht tijdens de afstudeeropdracht.

Het probleem waar PostNL mee kampt is dat de performance van het zakelijke klantenportaal niet optimaal werkt. Het merendeel van de zakelijke klanten van PostNL bevindt zich buiten Europa. Deze zakelijke klanten vinden dat het te lang duurt voor zij resultaat krijgen vanaf het klantenportaal wanneer zij een webpagina opvragen. Dit zorgt voor klachten vanuit de zakelijke klanten van PostNL.

Tijd voor onderzoek dus.
Tijdens de onderzoeksfase van mijn afstudeeropdracht ben ik opzoek geweest naar verschillende methoden en technieken om de performance van het klantenportaal te verbeteren. Hierop volgend heb ik tijdens de implementatiefase een prototype geïmplementeerd welke de effectiviteit van de gekozen oplossing aantoont.

Zoals hierboven beschreven heb ik een prototype geïmplementeerd en geen productie waardige oplossing. Hierom schrijf ik na de onderzoeks- en implementatiefase een adviesrapport. In dit adviesrapport is de gekozen oplossing beargumenteerd en is de effectiviteit beschreven.

Allereerst ben ik begonnen met een beschrijving van de organisatie Nalta en de manier waarop zij te werk gaat. Vervolgens heb ik de opdracht beschreven en de context waarin deze opdracht wordt uitgevoerd.

Verder komen in dit verslag onderwerpen aan bod zoals: de projectaanpak van de methoden en technieken die tijdens mijn afstudeeropdracht gebruikt zijn, de verschillende werkzaamheden die ik heb uitgevoerd, de resultaten en de evaluatie.

2. De organisatie

In dit hoofdstuk maak je kennis met het bedrijf waar ik mijn afstudeeropdracht uitgevoerd heb, Nalta. Vervolgens wordt er dieper ingegaan op de werkwijze die bij Nalta wordt gehanteerd.

Nalta Software BV
Het bedrijf waar ik mijn afstudeeropdracht heb gerealiseerd heet Nalta Software BV, voorheen heette zij Webbeat. Nalta is een internetbedrijf. Zij ontwerpt en ontwikkelt responsive websites, online platformen en applicaties. Een aantal voorbeeld applicaties die Nalta heeft ontwikkeld zijn Digital Angel, Datumprikker en het klantenportaal van PostNL (Nalta, 2020). Een groot deel van de werkzaamheden binnen Nalta bevat het ontwikkelen, echter doen zijn daarnaast ook nog andere werkzaamheden. Op de website van Nalta staat beschreven dat Nalta oplossingen biedt voor elke digitale transformatie. Dit varieert van het bedenken van strategieën tot en met de implementatie van de daadwerkelijke oplossingen. Nalta heeft 40 werknemers verdeeld over 2 filialen. Deze zijn gevestigd in Almere en Wateringen.

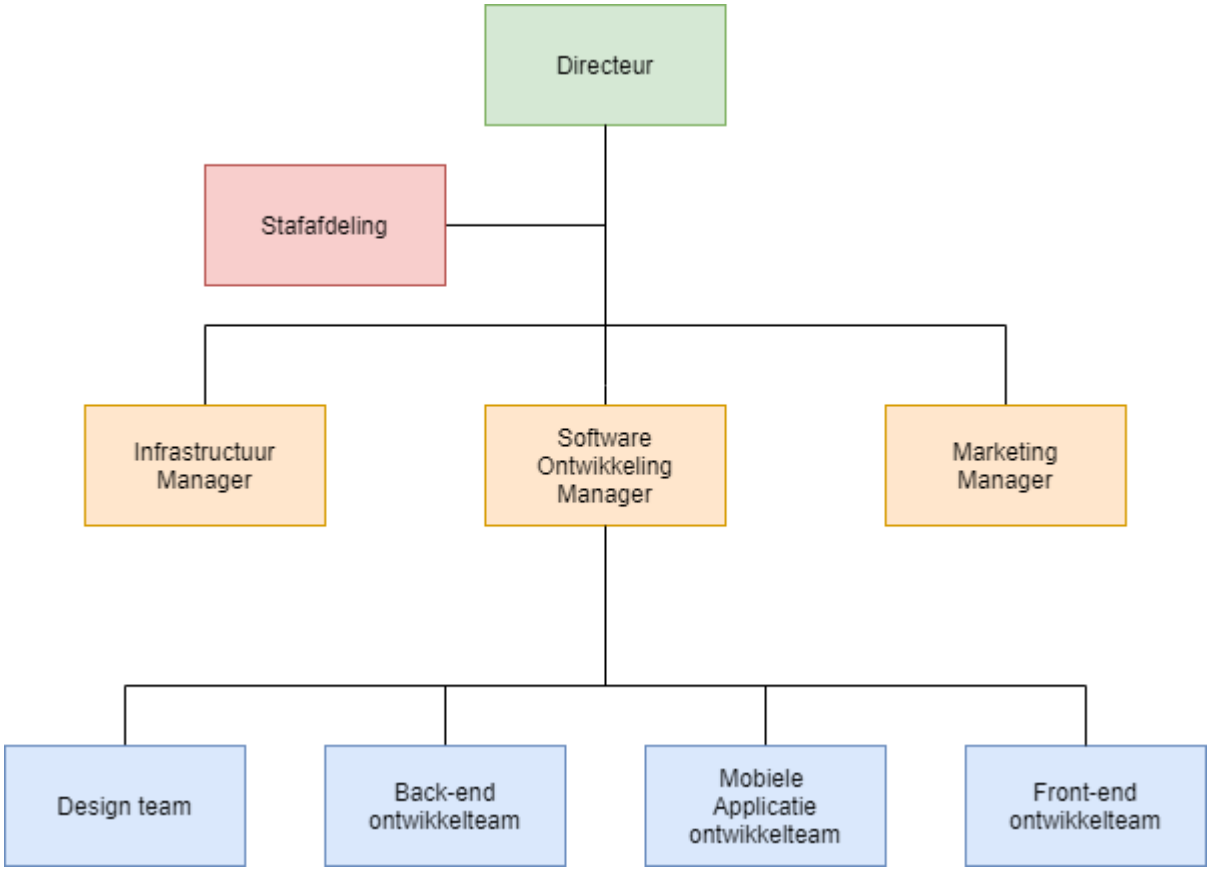
Locatie en indeling
Het hoofdkantoor van Nalta bevindt zich in Almere. Daar wordt hoofdzakelijk gewerkt aan de verschillende strategieën voor digitale transformatie. Het ontwikkelen van de daadwerkelijke applicaties wordt voornamelijk gedaan in Wateringen op de ontwikkelafdeling. Binnen deze afdeling ben ik werkzaam geweest in het front-end ontwikkelteam. In Wateringen is er een aantal flexplekken aanwezig, de meeste werknemers hebben echter een vaste plek.

Werkwijze
Binnen Nalta wordt er gewerkt met teams. Deze teams bestaan over het algemeen uit een aantal werknemers die samen aan een project werken. Aan deze projecten wordt er vervolgens weer gewerkt met een Agile methode, meestal wordt er gekozen voor Scrum. Scrum is een methode waarbij wordt gewerkt met korte cycli om zo snel mogelijk feedback te ontvangen. Door deze gekozen methode wordt er gewerkt in sprints, variërend in lengte per project, welke als doel hebben aan het eind van de sprint een werkend product op te leveren.

De projecten zelf worden meestal opgedeeld in twee delen, een back-end en een front-end. Zo is de functionaliteit net als de verantwoordelijkheid duidelijk verdeeld. De back-end werd doorgaans geschreven in de taal C# .NET Framework en gehost op een Windows Server. Tegenwoordig is Nalta steeds meer projecten aan het omzetten naar C# .NET Core projecten. Het front-end projecten worden voornamelijk in de taal JavaScript geschreven. Met name implementaties van Angular, VueJS en jQuery komen herhaaldelijk voor in het front-end projecten.

Organisatiestructuur

Om de organisatiestructuur meer overzichtelijk te maken is ervoor gekozen om een organogram te maken, welke te zien is in figuur 2.1. Het organogram is opgesteld volgens de standaard richtlijnen (Mittelmeijer & van Stratum, 2014, pp. 76–79). Er is voor gekozen om alleen de benodigde hiërarchie inzichtelijk te maken omdat het organogram anders onoverzichtelijk zou worden. Zoals beschreven in de paragraaf ‘Locatie en indeling’ heb ik mijn afstudeeropdracht uitgevoerd op de ontwikkelafdeling en in het front-end ontwikkelteam. Het front-end ontwikkelteam is in figuur 2.1 met blauw aangegeven. Tevens is mijn opdrachtgever de heer G. Merkelbach in het groen als directeur aangegeven en mijn begeleider de heer M. Beenes in het oranje als software ontwikkeling manager.



Figuur 2.1 - Organisatiestructuur

3. De afstudeeropdracht

Dit hoofdstuk is opgedeeld in drie delen. In de eerste paragraaf wordt de probleemstelling nog eens beschreven. Vervolgens beschrijf ik in de tweede paragraaf de huidige situatie om een beeld te krijgen over wat er momenteel aan de hand is. In de laatste paragraaf worden de verschillende tussenproducten beschreven die opgeleverd zullen worden tijdens de opdracht.

3.1 Probleemstelling

De afstudeeropdracht is gericht op het klantenportaal van PostNL. Het klantenportaal van PostNL wordt gebruikt door de zakelijk klanten van PostNL. Dit klantenportaal wordt steeds intensiever gebruikt. Naast Europa wordt de applicatie nu ook veelvuldig gebruikt in onder andere Hong Kong en Canada. Het merendeel van de klanten die zijn aangesloten bij PostNL bevinden zich dan ook buiten Europa. Echter, de performance is buiten Europa niet optimaal.

Wanneer een webpagina in Azië wordt geladen, duurt het aanzienlijk langer om het resultaat hiervan aan de gebruiker te tonen dan wanneer dit binnen Europa wordt gedaan. Doordat de heer G. Merkelbach heeft aangegeven dat PostNL hier al een tijd mee kampt is er al een aantal mogelijke oplossingen bedacht om de performance van het klantportaal van PostNL te verbeteren. Helaas hebben die oplossingen niet het gewenste resultaat opgeleverd.

PostNL wil graag dat Nalta uitzoekt welke andere mogelijkheden er nog meer gebruikt kunnen worden om de performance van het klantenportaal te verbeteren.

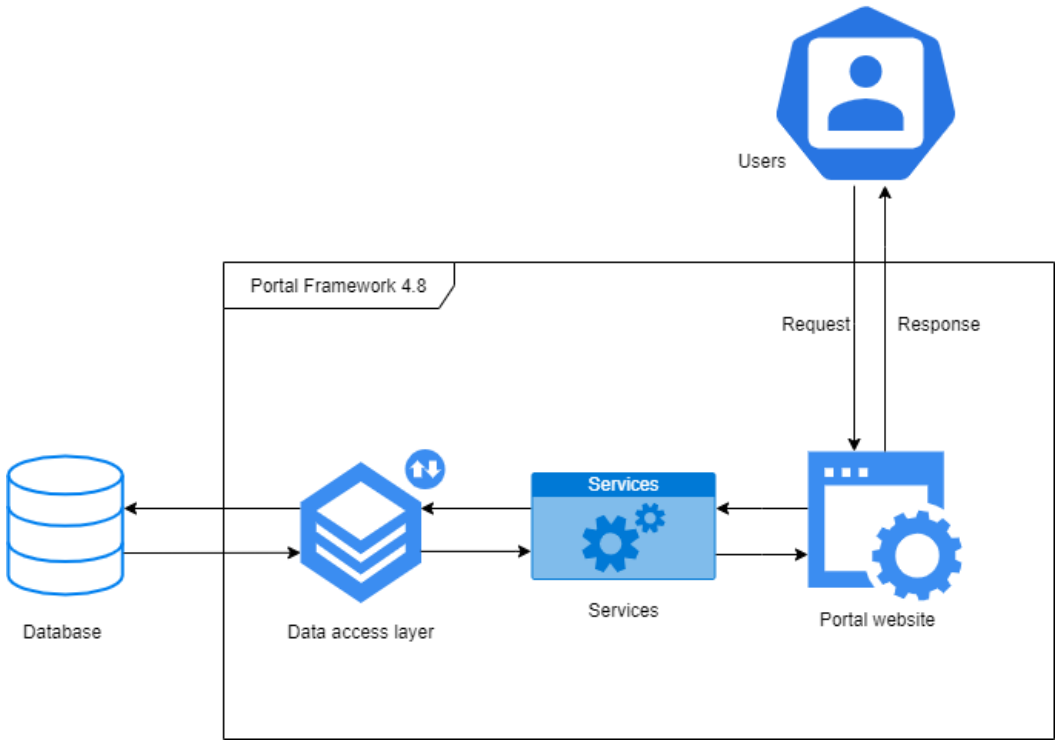
3.2 Huidige situatie van het klantenportaal

Om meer informatie in te winnen over de huidige situatie heb ik met verschillende stakeholders binnen het bedrijf gesproken. Onder deze stakeholders vallen de personen die zich bezighouden met het klantenportaal: de product owner, het hoofd van het ontwikkelteam en mijn begeleider. Ik heb expliciet voor deze mensen gekozen omdat zij het meeste verstand hebben van het product en de achterliggende gedachte. Uit de gesprekken met deze stakeholders is naar voren gekomen dat er twee verschillende versies van het klantenportaal bestaan welke ook nog onderhouden worden. De versie die momenteel in productie staat is ontwikkeld in .NET Framework en een vernieuwde versie welke is ontwikkeld in .NET Core.

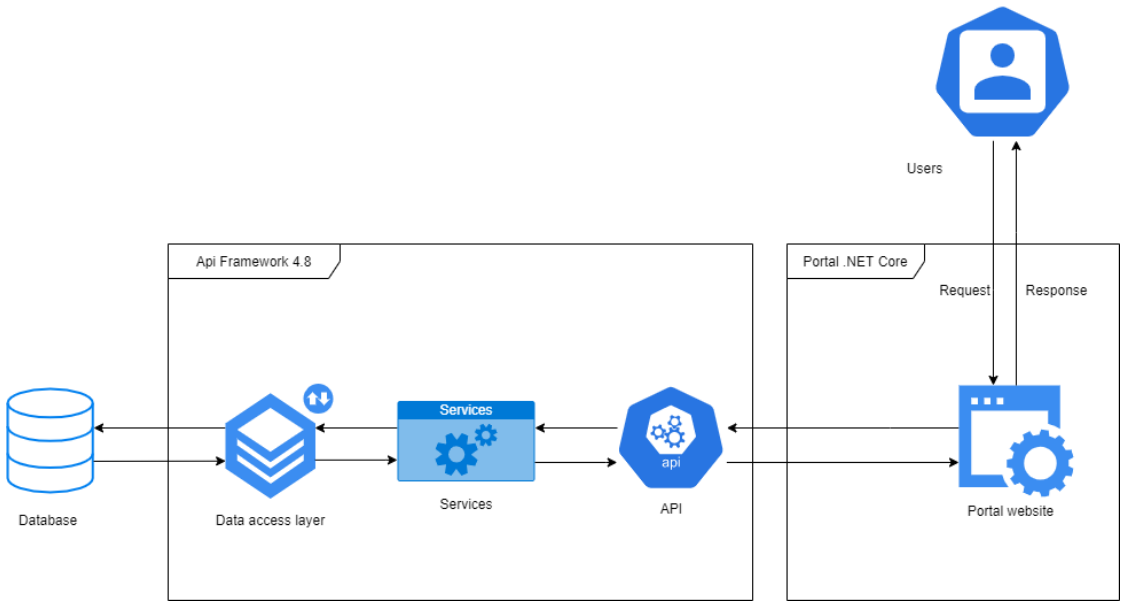
De .NET Framework versie die wordt gebruikt voor de productieomgeving is 4.8. Echter, in het voorjaar van 2019 kwam Microsoft met het nieuws naar buiten dat .NET Framework 4.8 de laatste versie zal zijn van het .NET Framework (Asthana, 2019). In november van 2020 komt .NET 5 uit welke de opvolger is van .NET Core 3.1. Voor deze naamgeving is gekozen om mogelijke verwarring van .NET Framework en .NET Core in de versie 4 range te voorkomen.

Het stoppen van .NET Framework was één van de redenen voor Nalta om over te stappen naar .NET Core. Hiermee blijven ze op technisch niveau vooruitstrevend. Een andere reden

voor de overstap is dat het front-end te veel verweven is met de back-end in de huidige productieomgeving. Hierdoor kunnen het front-end en back-end niet los gepubliceerd worden. Deze scheiding is vanuit Nalta wel gewenst, omdat dit haar standaard manier van werken is. Om de transitie naar .NET Core visueel inzichtelijk te maken heb ik figuur 3.1 en 3.2 gemaakt waarvan 3.2 te zien is op de volgende pagina. In figuur 3.1 is de .NET Framework 4.8 infrastructuur te zien terwijl in figuur 3.2 de infrastructuur van .NET Core is weergeven.



Figuur 3.1 - .NET Framework 4.8 infrastructuur



Figuur 3.2 - .NET Core infrastructuur

Nalta hoopte dat de infrastructuur transitie ook een verbetering zou bieden op het gebied van performance. Echter, dit bleek niet het gewenste resultaat op te leveren. Om aan te tonen dat het probleem zich nog steeds voordoet in de .NET Core versie heb ik een test voor beide versies uitgevoerd met behulp van de website Pingdom.com (https://www.pingdom.com/) (Burge, 2014). Er is gekozen om deze test met Pingdom uit te voeren omdat zij een groot aantal servers verspreid over de wereld heeft. Via de website van Pingdom kan een snelheidstest worden aangevraagd voor een specifieke website vanaf een van de servers van Pingdom. Op dat moment wordt de website vanaf die server opgehaald en wordt bijgehouden hoelang het duurt voor de website volledig geladen is.

Uit een onderzoek van Bland en Altman (Bland & Altman, 1996) blijkt dat een test minstens tweemaal voor verschillende onderwerpen moet worden uitgevoerd om een betrouwbare standaarddeviatie te verkrijgen. Mijn uitgevoerde tests zijn per continent tienmaal uitgevoerd om een accuraat resultaat te behalen. In tabel 3.1 staat per testlocatie het gemiddelde van de testen weergeven met tussen haakjes de standaarddeviatie. Tevens is met rood de langzaamste laadtijd weergeven en in het groen de snelste. Een hogere laadtijd is in dit geval slechter. Dit betekent dat gebruikers langer moeten wachten op de website en daarom de website als langzaam ervaren.

Regio	.NET Framework	.NET Core
Europa, Duitsland, Frankfurt	595 ms (σ 93)	**540 ms** (σ 97)
Europa, Engeland, Londen	607 ms (σ 97)	565 ms (σ 101)
Noord-Amerika, USA, San Francisco	3021 ms (σ 314)	2766 ms (σ 267)
Azië, Japan, Tokio	5196 ms (σ 480)	4286 ms (σ 445)
Oceanië, Australië, Sydney	**5974 ms** (σ 557)	5214 ms (σ 521)

Tabel 3.1 - Laadsnelheid

3.3 Opdrachtomschrijving

Zoals eerder beschreven, kampt PostNL met een performance probleem op het klantenportaal. Voor dit probleem dient er een advies te worden gegeven over hoe de performance van het klantportaal verbeterd kan worden. Voordat hier een duidelijk en betrouwbaar advies over gegeven kan worden, zal eerst de oorzaak en de mogelijke oplossingen moeten worden onderzocht. Daarnaast is gewenst dat een geschikte oplossing uitgewerkt wordt door middel van een werkend prototype om de effectiviteit aan te tonen. De manier waarop ik dit van plan ben te gaan doen staat beschreven in het volgende hoofdstuk. Tevens staat in het volgende hoofdstuk ook beschreven welke tussenproducten ik denk nodig te hebben om tot een gegrond advies te komen.

4. Projectaanpak

Dit hoofdstuk biedt inzicht in de aanpak die ik tijdens mijn afstudeeropdracht heb gehanteerd. Hierin komen de volgende onderwerpen aanbod: op te leveren producten, ontwikkelmethodiek, fasering, planning en een risicoanalyse.

4.1 Op te leveren producten

Om tot een oplossing voor het klantenportaal te komen dient er een advies te worden gegeven over hoe de performance van het klantportaal geoptimaliseerd kan worden. Voordat dit advies geleverd kan worden ga ik een aantal tussenproducten opleveren. Deze tussenproducten zijn naar mijn mening het meest geschikt om tot een gegrond advies te komen. Onder deze producten vallen: requirementsrapport, onderzoeksrapport, ontwerp, prototype, testverslag en een adviesrapport. Al deze producten staan beschreven in dit verslag.

Requirementsrapport
In het requirementsrapport ga ik op zoek naar de requirements en technische beperkingen die betrekking hebben op mijn opdracht. Door middel van het requirementsrapport kan ik het onderzoek en het prototype afbakenen.

Onderzoeksrapport
Vervolgens lever ik een onderzoek op, waarin ik op zoek ga naar methoden om de performance van het klantenportaal te verbeteren. De resultaten van dit onderzoek zal ik beschrijven in een onderzoeksrapport. Uiteindelijk levert dit onderzoek een onderzoekshypothese op. Deze onderzoekshypothese wordt aan de hand van de andere producten verder onderzocht.

Ontwerp
Om gericht te kunnen starten met ontwikkelen, is het belangrijk dat er eerst een ontwerp wordt opgesteld voor de software. Na het uitvoeren van het onderzoek ga ik een ontwerp opleveren waarin de architectuur gebaseerd is op de gevonden onderzoekshypothese. Tevens ben ik van plan in om voor dit ontwerp ook een klassendiagram te maken waar het prototype op gebaseerd zal worden.

Prototype
Aan de hand van dit ontwerp ga ik van start met het gericht ontwikkelen van een prototype. Tijdens het ontwikkelen van dit prototype ga ik gebruik maken van de oplossing die wordt aanbevolen uit het onderzoek. Daarbij ga ik de verschillende modellen van het prototype baseren op het klassendiagram van het ontwerp.

Testverslag
Vervolgens zal ik het prototype gaan testen om te bepalen of de gevonden hypothese voldoet aan de verwachtingen van de onderzoekshypothese. Deze testen ga ik beschrijven in het testverslag.

Adviesrapport
Aan de hand van alle eerder beschreven tussenproducten kan ik een gegrond adviesrapport schrijven voor Nalta. In het advies dat ik schrijf aan Nalta zal ik uitleggen waarom de gevonden onderzoekshypothese mogelijk effect zal hebben op het klantenportaal.

4.1 Ontwikkelmethodiek

Zoals beschreven in paragraaf 4.1 lever ik tijdens deze afstudeeropdracht verschillende producten op. Een aantal van deze tussenproducten valt onder het software ontwikkeltraject. Na het verwerven van de requirements en het uitvoeren van het onderzoek ga ik beginnen met dit ontwikkeltraject. Gedurende de afstudeeropdracht ben ik van plan om met behulp van een softwareontwikkelmethode te werken aan het ontwikkeltraject. Voordat dit proces gestart kan worden heb ik de volgende drie mogelijke softwareontwikkelmethoden bekeken, welke gebruikt zouden kunnen worden voor de opdracht: Waterval, Scrum en Kanban. Gezien Waterval en Scrum tijdens de opleiding zijn aangeleerd en dit onder de algemene kennis van IT'ers valt geef ik hier geen uitgebreide uitleg over. Wel geef ik meer informatie over de toepasbaarheid voor de afstudeeropdracht.

Waterval
Tijdens mijn opleiding ben ik in aanraking gekomen met de watervalmethode en de ideologie erachter. Echter, om te bepalen welke softwareontwikkelmethode het beste lijkt aan te sluiten op het software ontwikkeltraject heb ik het boek van Balaji & Sundararajan Murugaiyan (2012) nog eens geraadpleegd. Met de verwachting dat er een aanzienlijke kans is dat er met een nieuwe techniek gewerkt gaat worden, is de kans groot dat er tijdens het ontwikkelen nieuwe inzichten worden opgedaan. Deze inzichten kunnen leiden tot aanpassingen in onder andere het ontwerp en het prototype. Om deze reden lijkt de lineaire aanpak van de watervalmethode niet geschikt voor de afstudeeropdracht.

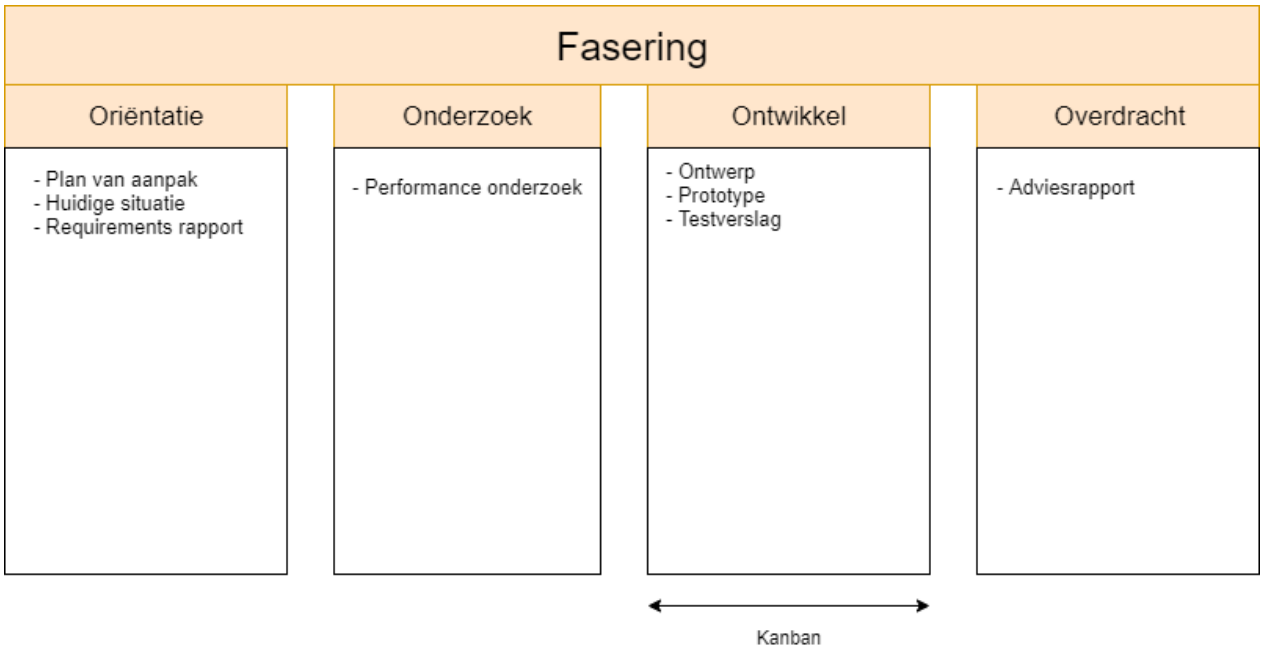
Scrum
Een andere optie is om Scrum te gebruiken. Net als de watervalmethode is er tijdens de opleiding tot Software Engineer ook ervaring opgedaan met Scrum. Echter, om er zeker van te zijn dat mijn kennis nog actueel is, is ook voor Scrum de literatuur geraadpleegd (Rising & Janoff, 2000, pp. 26–32). Scrum werkt met sprints welke zich erop richten om in een korte periode, meestal twee tot vier weken, werkende software op te leveren. Scrum is bedoeld voor projectteams. Een dergelijk team bestaat uit een product owner, een scrum master en een aantal ontwikkelaars en testers. Scrum lijkt beter te passen bij de afstudeeropdracht gezien Scrum iteratief aan de verschillende producten werkt. Zoals eerder beschreven brengt het mogelijk werken met een nieuwe techniek met zich mee dat er aanpassingen doorgevoerd dienen te worden na het verkrijgen van nieuwe inzichten, deze kunnen vervolgens per iteratie worden meegenomen. Echter, de afstudeeropdracht wordt niet in een team uitgevoerd, ook is er geen scrum master of product owner aanwezig. Om deze redenen valt de techniek Scrum af.

Kanban
Een derde mogelijkheid is om gebruik te maken van de Kanban methode, Kanban wordt gebruikt om het werk wat gedaan dient te worden visueel inzichtelijk te maken.Om een Kanban bord wordt er gewerkt met swimlanes, de term die wordt gebruikt om het Kanban proces aan te duiden. In deze swimlanes staat het ontwikkelproces weergegeven en aan welke taken er op het moment wordt gewerkt. De Kanban methode probeert het ontwikkelproces voor het product van de klant te optimaliseren (Ahmad, Markkula, & Oivo, 2013, pp. 9–16). Tijdens het Kanban proces wordt er continu gekeken naar welke taken op dat moment de meeste waarde zouden toevoegen aan het product van de klant. Deze taken staan in een geordende lijst op basis van de toegevoegde waarde die ze opleveren. Wanneer er taken zijn die geen toegevoegde waarde hebben voor de klant worden deze van het Kanban bord verwijderd. Hierdoor ontstaat het meest optimale proces om een waardevol product te ontwikkelen voor de klant. Net als Scrum is Kanban een iteratieve ontwikkelmethode waardoor deze goed aansluit op de afstudeeropdracht. Rising en Janoff geven aan dat in een situatie waarin maar één persoon aan het ontwerp, prototype en testverslag werkt Kanban in tegenstelling tot Scrum wel gebruikt kan worden. Als voorwaarde dient deze persoon wel altijd te werken aan het belangrijkste issue. Om deze reden ga ik de Kanban methode gebruiken tijdens het ontwikkelen van de software.

4.2 Fasering

Nadat de ontwikkelmethodiek is bepaald, ben ik gaan onderzoeken welke manier van fasering het best aansluit op de verschillende tussenproducten. Hierbij leken de verschillende fasen die worden aangeraden door de Universiteit Leiden het meest gepast (Filippo, 2012). Er is een scheiding gemaakt tussen de oriëntatiefase, onderzoeksfase, ontwikkelfase en overdrachtsfase. In mijn geval ga ik tijdens de ontwikkelfase de Kanban methode gebruiken om een prototype te ontwikkelen en te testen. De verschillende fasen zijn visueel gemaakt in figuur 4.1. De uitwerkingen van de tussenproducten komen later in dit verslag aanbod.

Figuur 4.1 - Fasering



4.2.1 Methodieken oriëntatiefase

In deze paragraaf worden de verschillende methoden en termen beschreven die ik van plan ben te gaan gebruiken tijdens de oriëntatiefase. Figuur 4.1 laat zien dat er in de oriëntatiefase drie producten worden opgeleverd. Het plan van aanpak en de beschrijving van de huidige situatie zijn alvorens dit document gemaakt en opgeleverd. Om deze reden worden die producten niet beschreven en zijn ze ook niet meegenomen in de planning van paragraaf 4.3.

Interviews
Om de requirements voor het requirementsrapport te vergaren wordt de interviewtechniek gebruikt voor het eliciteren van informatie bij verschillende stakeholders. Met deze stakeholders worden dezelfde stakeholders bedoeld zoals deze in paragraaf 3.2 zijn beschreven. Voor deze interviews stel ik van tevoren open vragen op zodat er niet van de vraag richting kan worden afgeweken tijdens het doorvragen (de Swart, 2010, pp. 192–194).

4.2.2 Methodieken onderzoeksfase

In deze paragraaf worden de verschillende methoden en termen beschreven die ik van plan ben te gaan gebruiken tijdens de onderzoeksfase. Figuur 4.1 laat zien dat er in de onderzoeksfase alleen een onderzoeksrapport wordt opgeleverd. De onderstaande methoden en termen hebben allemaal betrekking op dit onderzoeksrapport.

Deskresearch
Om correct antwoord te kunnen geven op de verschillende vragen die naar voren zijn gekomen tijdens de afstudeeropdracht is er veelal gebruik gemaakt van deskresearch. Deskresearch is een techniek waarbij vanaf achter het bureau geprobeerd wordt om resultaten uit bestaande literatuur te combineren om zo tot nieuwe conclusies te komen (Krul, 2017). Tijdens het onderzoek ben ik van plan om deskresearch te gebruiken om te onderzoeken of er al antwoorden zijn op geformuleerde vragen die naar voren zijn gekomen en ook om te valideren of de gevonden antwoorden correct zijn en eveneens door andere partijen worden gevalideerd.

4.2.3 Methodieken ontwikkelfase

In deze paragraaf worden de verschillende methoden en termen beschreven die ik van plan ben te gaan gebruiken tijdens de ontwikkelfase. Figuur 4.1 laat zien dat er in de ontwikkelfase drie producten worden opgeleverd: het ontwerp, een prototype en een testverslag. Per methode wordt beschreven voor welk product ik verwacht de desbetreffende methode te gebruiken.

UML
UML staat voor Unified Modeling Language. UML geeft een aantal standaarden aan voor het beschrijven, specificeren, visualiseren van software. Voor het ontwerp is er vanuit de opdrachtgever geen eis gesteld aan de techniek die wordt gebruikt voor het opstellen van het ontwerp. Om deze reden ben ik van plan om de UML-modelleertaal te gebruiken voor het opstellen van deze diagrammen. Deze diagrammen geven een duidelijk overzicht over hoe de applicatie er uiteindelijk uit komt te zien. Daarbij is de UML-modelleertaal de standaard binnen softwareontwikkeling en aangeleerd vanuit school.

Deskresearch
Net als voor het onderzoek wordt er voor het prototype en het testverslag gebruik gemaakt van deskresearch. Uit het onderzoek volgt een onderzoekshypothese en aanbeveling. Om meer informatie over de uitgewezen techniek van het onderzoek in te winnen lijkt het gebruik van deskresearch een geschikte manier. Als de software is ontwikkeld wordt er door middel van deskresearch een passende manier van testen onderzocht om te controleren of het prototype voldoet aan de verwachtingen van de onderzoekshypothese. Tevens zal deskresearch gebruikt worden om oplossingen te zoeken voor mogelijke problemen die ontdekt kunnen worden tijdens het testen.

Microsoft Azure Portal
De Microsoft Azure Portal staat bedrijven toe om verschillende soorten applicaties te hosten in de cloud. Onder de cloud providers heeft Azure ongeveer 17 tot 18% van de markt in handen (Chapel, 2020). Ik heb ervoor gekozen om de Microsoft Azure Portal toe te voegen aan de methoden voor het prototype omdat ik van plan ben om mijn prototype te hosten in de Azure Cloud. Microsoft Azure Portal is een betrouwbare cloud provider en wordt tevens door Nalta als standaard gebruikt.

BitBucket & DevOps
Tegenwoordig maakt bijna ieder modern IT-bedrijf gebruik van source control en code automatisering. Ook bij Nalta is dit het geval. Tijdens de oriëntatie binnen het bedrijf is de harde eis naar voren gekomen dat de private git omgeving van Nalta op BitBucket gebruikt dient te worden voor het opslaan van de sourcecode. Naast het opslaan van de sourcecode, is het bouwen, testen en publiceren van het prototype geautomatiseerd. Hiervoor wordt de Azure DevOps omgeving van Nalta gebruikt. Dit lijkt een geschikte methode te zijn daar het gemakkelijk aansluit op de Microsoft Azure Portal.

4.2.4 Methodieken overdrachtsfase

Als afsluiting van de afstudeeropdracht wordt er in de overdrachtsfase een adviesrapport uitgebracht. Voor het adviesrapport wordt er geen nieuwe kennis op gedaan en wordt alle informatie gehaald uit de resultaten van de eerder opgestelde tussenproducten.

4.3 Planning

Om een duidelijk overzicht te krijgen van het geplande verloop van de afstudeerperiode is er aan de hand van methoden per product en fasering een planning gemaakt voor de op te leveren tussenproducten. Deze planning is weergegeven in tabel 4.1. In deze planning is er rekening gehouden met de verwachte duur van de tussenproducten zoals deze is beschreven in het afstudeerplan. Als aanpassing ten opzichte van het afstudeerplan zijn in deze planning ook het ontwerp en het testverslag meegenomen welke ik van tevoren niet had voorzien. Ik denk dat deze producten echter wel nodig zijn om tot een gegrond advies te kunnen komen.

Product	Startdatum	Datum van oplevering
Requirementsrapport	28 februari	13 maart
Onderzoeksrapport	14 maart	3 april
Ontwerp	4 april	20 april
Prototype	8 april	25 april
Testverslag	10 april	8 mei
Adviesrapport	9 mei	25 mei

Tabel 4.1 - Planning

4.4 Risicoanalyse

In tabel 4.2 zijn de risico's uiteengezet die tijdens het project zouden kunnen ontstaan. Hiervoor is er gekeken naar de eerdere algemene risico's van soortgelijke projecten die tijdens de opleiding zijn uitgevoerd. Deze risico's zijn verder gedefinieerd aan de hand van een projectrisico tabel waarin de kans maal het effect is aangeven als de waarde van het risico. Er is voor deze aanpak voor een dergelijke tabel gekozen doordat hier in een eerder stadium van de opleiding ervaring mee is opgedaan en deze erg effectief bleek te zijn.

Nr.	Risico	Gevolg	Tegenmaatregel	Kans (1-5)	Effect (1-5)	Risico (K*E)
1	De gevonden mogelijke oplossing blijkt geen optimalisatie te leveren.	De herziende versie van het klantenportaal is langzamer dan dat het voorheen was.	Meerdere opties onderzoeken en oplossingen bieden zodat mogelijk een andere oplossing geadviseerd kan worden.	3	4	12
2	Het gekozen ontwerp van het prototype blijkt onjuist te zijn.	Het prototype kan niet ontwikkeld worden zoals dat van tevoren was bedacht.	Genoeg tijd inplannen om het gekozen ontwerp te controleren bij de infrastructuurarchitect en deze waar nodig aan te passen.	2	4	8
3	Onvolledige dataverzameling van huidige bedrijfsvoering/ processen/systemen.	Onvolledig beeld van de huidige bedrijfsvoering/ processen/systemen waardoor een onjuist advies naar boven kan komen.	Goede voorbereiding voor dataverzameling en juist in kaart brengen van de bedrijfsvoering/processen/ systemen	2	5	10

Tabel 4.2 - Risico's

Het eerste risico blijkt door de kans en het effect het grootste risico te zijn. Om het risico hiervan in te perken zal ik tijdens het uitvoeren van het onderzoek direct rekening houden met de tegenmaatregel door meerdere opties te onderzoeken. Vervolgens is voor het ontwikkelen van het prototype genoeg tijd ingepland zodat wanneer het prototype geen optimalisatie oplevert mogelijk een andere techniek gebruikt kan worden om een prototype te ontwikkelen. Ook voor minder risicovolle risico's geldt vooral dat er genoeg tijd moet worden uitgetrokken voor de tegenmaatregel zodat de schade wordt ingeperkt. Op deze manier wordt er preventief aan de risico's gewerkt om de kans te verminderen dat het optreedt.

5. Requirementsrapport

In dit hoofdstuk staan de verschillende taken beschreven die zijn uitgevoerd in het requirementsrapport. Hieronder vallen het afbakenen van de scope, het werven van de requirements en het opstellen van de technische beperkingen.

Tijdens het werven van de requirements is er rekening gehouden met een prototype dat ontwikkeld gaat worden. Het opgeleverde prototype moet voldoen aan de basis functionaliteit van het huidige klantenportaal. Pas wanneer de gekozen basis functionaliteit representatief is voor de functionaliteit van het huidige portaal kunnen beide platformen met elkaar worden vergeleken. Vervolgens kan er een uitspraak worden gedaan of er daadwerkelijk een optimalisatie heeft plaatsgevonden op het klantenportaal.

Om de requirements inzichtelijk te maken heb ik verschillende open vragen gesteld aan de stakeholders, met de stakeholders worden dezelfde personen bedoeld als die zijn beschreven in de paragraaf 3.2 ‘huidige situatie’. Voorbeelden van deze vragen zijn; “Welke functionaliteit moet het prototype minimaal hebben?” en “In welke mate mag het performance verschil tussen binnen en buiten Europa merkbaar zijn?”

5.1 Scope en gebruikers

Als eerste onderdeel van het requirementsrapport heb ik het project afgebakend. Ook wel de project scope genoemd. Deze afbakening is voornamelijk gehaald uit de opdrachtomschrijving maar duidelijk gemaakt door middel van een gesprek met mijn begeleider.

Het huidige klantenportaal van PostNL moet worden herzien omdat het met performance problemen kampt voor bezoekers vanuit buiten Europa. In sommige regio’s kan het 5 a 6 seconden duren voordat ze resultaat krijgen van de inlogpagina, welke één van de minst grootste pagina’s is. Deze gebruikers ervaren hierdoor dat het te lang duurt voor ze de schermen van het portaal te zien krijgen. Hierbij is vanuit de opdrachtgever gegeven dat er geen aanpassingen aan de back-end zullen komen en dat deze dus buiten de scope valt.

Na de afbakening zijn de verschillende gebruikersgroepen onderzocht waaruit is gebleken dat er twee verschillende gebruikersgroepen zijn die momenteel gebruik maken van het klantenportaal. De eerste groep bestaat uit de zakelijke klanten van PostNL. Wanneer het prototype performance optimalisatie oplevert buiten Europa zal deze groep hier het meeste van merken. Zoals beschreven in paragraaf 3.1 bevinden veel zakelijke klanten van PostNL zich buiten Europa. Naast de zakelijke klanten wordt het klantenportaal ook gebruikt door PostNL haar eigen werknemers. Deze werknemers testen het klantenportaal steekproefsgewijs om te controleren of het klantenportaal nog werkt naar behoren.

5.2 Business requirements

Nadat de scope van de opdracht is afgebakend heb ik de antwoorden van de interviews geanalyseerd. Uit deze analyse zijn de business requirements naar voren gekomen die betrekking hebben op mijn afstudeeropdracht. Deze business requirements zijn te zien in tabel 5.1.

Id	Beschrijving	Stakeholder	Bron
BR1	De werknemers van PostNL willen dezelfde functionaliteit hebben als in het originele portaal.	Werknemers	Opdracht beschrijving
BR2	PostNL wil dat de geografische locatie weinig effect heeft op de laadsnelheid van het portaal.	PostNL	Opdracht beschrijving
BR3	PostNL wil dat het portaal sneller wordt.	PostNL	Opdracht beschrijving

Tabel 5.1 - Requirements

5.3 Functionele requirements

In deze paragraaf worden de verschillende functionele requirements beschreven. In de eerder beschreven business requirements heeft alleen BR1 betrekking op functionaliteit. De functionaliteit wordt in de tabel 5.2 beschreven door een aantal use cases. De use cases gaan alleen in op de basisfunctionaliteit die in het prototype dient te zitten zoals in de inleiding van het hoofdstuk staat. Deze use cases zijn gekozen omdat deze representatief staan voor de volledige functionaliteit van het huidige platform. De kern van de opdracht zit voornamelijk in BR2 en BR3, welke meer betrekkingen hebben op de non-functionele requirements.

Gezien het prototype een groot deel van reeds bestaande functionaliteit dient te bevatten, heeft het weinig zin om aan de hand van MoSCoW (de Swart, 2010, pp. 225–228) te prioriteren. In dat geval had het overgrote deel een ‘must have’ geweest. De prioritering is gedaan aan de hand van de meest gebruikte methoden binnen het klantenportaal. Deze prioritering wordt ook wel de relatieve prioritering methode genoemd (de Swart, 2010). De prioritering is gevalideerd met de leidinggevende binnen het bedrijf. Een groot voordeel van relatieve prioritering is dat de verschillende taken in dezelfde volgorde op het Kanban bord komen te staan.

#U	Use case	Beschrijving	Prioritering
U1	Beginscherm	De klant moet het beginscherm kunnen laden	2
U2	Inloggen	De klant moet kunnen inloggen op het platform	1
U3	Manifest upload	De klant moet manifesten kunnen uploaden	3

U4	Manifest overview	De klant moet het manifest overzicht kunnen bekijken	6
U5	Download manifest	De klant moet een manifest kunnen downloaden	4
U6	Genereer assist label	De klant moet de mogelijkheid hebben om een assist label te genereren	5

Tabel 5.2 - Use cases

5.4 Niet-functionele requirements

In de volgende tabel, tabel 5.3, staan de niet-functionele requirements voor de opdracht beschreven. Naast een identifier staat het requirement ook beschreven. Het requirement wordt met een kwaliteitskenmerk gecategoriseerd volgens de ISO 25010. Hiervoor is gekozen omdat de categorisering van ISO 25010 duidelijk laat zien waar de focus van de software op ligt. Daarbij wordt ISO 25010 vaak als standaard gebruikt binnen de IT-wereld doordat het duidelijkheid schept.

Id	Requirement	ISO 25010 Kwaliteitskenmerk	Bron
NF1	Een webpagina moet buiten Europa sneller resultaat geven dan in de huidige situatie	Time behaviour	Opdracht omschrijving
NF2	De systeem taal is Engels	Usability	Opdracht omschrijving
NF3	Binnenkomende requesten moeten via een secure connection worden verwerkt	Functionality	Opdracht omschrijving
NF4	De UI van applicatie moet automatisch getest kunnen worden	Testability	Opdracht omschrijving

Tabel 5.3 - Niet-functionele requirements

Om aan te tonen dat aan de standaarden van deze niet-functionele requirements is voldaan dient het ontwikkelde prototype getest te worden op deze requirements. Wanneer aan deze niet-functionele is voldaan kunnen ook BR2 en BR3 worden aangetoond.

5.5 Technische beperkingen

Het laatste onderdeel van requirementsrapport bevat de technische beperkingen, deze beperkingen zijn opgesteld vanuit Nalta en PostNL. Het onderzoek en prototype moeten voldoen aan de opgestelde technische beperkingen anders worden de producten als niet valide gezien. Deze technische beperkingen zijn te zien in tabel 5.4

Id	Beperking
TB1	Het front-end en de back-end moeten apart van elkaar gepubliceerd kunnen worden.
TB2	Autorisatie aan de hand van JWT-technieken.
TB3	Het moet cloud platform agnostisch zijn.
TB4	Er moet gebruikgemaakt worden van binnen het bedrijf bekende programmeertalen (C#, JavaScript).
TB5	Er worden geen aanpassingen gedaan aan de back-end.
TB6	Geografische replicatie van de back-end is geen optie.

Tabel 5.4 - Technische beperkingen

5.6 Vervolg

Nadat de verschillende requirements en technische beperkingen zijn opgesteld en gecontroleerd heb ik het tussenproduct opgeleverd. Hierdoor is het verwerven van requirements voltooid en kan er een start gemaakt worden aan het volgende product; het performance onderzoek. Tevens is hiermee de oriëntatiefase afgesloten en wordt de onderzoeksfase gestart. Hoe ik het performance onderzoek heb aangepakt en welke resultaten dit onderzoek heeft opgeleverd is te lezen in hoofdstuk 6.

6 Onderzoeksrapport

Na het afronden van het requirementsrapport is de oriëntatiefase voltooid, zoals hij te zien is in de fasering. Hierdoor heb ik volgens planning van mijn afstudeeropdracht kunnen starten met de onderzoeksdaden. Figuur 4.1 in paragraaf 4.2 laat zien dat ik in de onderzoeksfase alleen een onderzoeksrapport oplever. Dit onderzoek is bijgevoegd als een bijlage op pagina 81. Mochten bepaalde methoden en resultaten niet duidelijk genoeg zijn beschreven kunnen deze hieruit opgehaald worden.

6.1 Onderzoeksvraagstelling

Het onderzoeksrapport is een belangrijk deel van mijn afstudeeropdracht. Tijdens dit onderzoek ben ik achter een onderzoekshypothese gekomen die mogelijk de performance van het klantenportaal kan verbeteren. De onderstaande hoofdvraag stond in het onderzoek centraal:

“Hoe kan Nalta de performance van het klantenportaal van PostNL in regio’s buiten Europa significant verbeteren?”

Deze hoofdvraag is opgesteld en vervolgens gevalideerd met mijn bedrijfsbegeleider om er zeker van te zijn dat er in de juiste richting onderzoek wordt gedaan. De hoofdvraag laat zien welke punten er onderzocht dienen te worden voordat deze beantwoord kan worden. Om deze hoofdvraag te beantwoorden zijn er een tweetal deelvragen opgesteld die hiermee helpen:

1. *“Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa?”*
2. *“Welke bestaande methoden kunnen worden toegepast om het serveren van het klantenportaal te versnellen?”*

Aan de hand van deze deelvragen kan de hoofdvraag worden beantwoord en kan er een aanbeveling worden gedaan voor een geschikte oplossing om de onderzoekshypothese te valideren.

6.2 Deelvraag 1: Oorzaak van het probleem

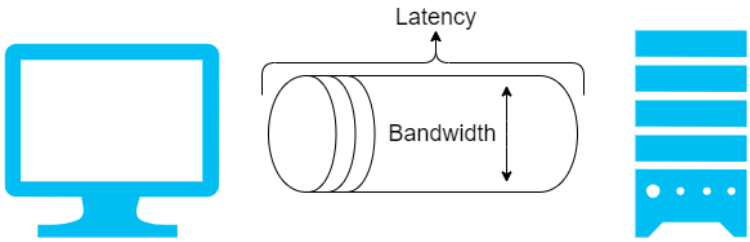
Om een juist en accuraat antwoord te kunnen geven op de eerste deelvraag, *“Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa?”,* ben ik begonnen met het doen van deskresearch (Krul, 2017). Naar mijn mening past deskresearch het beste in deze situatie, omdat veel voorkomende problemen hierbij extra zijn toegelicht.

Tijdens dit onderzoek is literatuur verzameld waarmee rekening wordt gehouden met de testresultaten uit paragraaf 3.2. Deze testresultaten geven aan dat de website inderdaad langzamer laadt in de regio’s buiten Europa. Ik ben begonnen met het vergaren van informatie over de reden dat sommige webapplicaties langzamer zijn dan andere. Souders geeft in zijn boek aan dat er verschillende oorzaken zijn die ervoor kunnen zorgen dat een website langzaam laadt (Souders, 2007, pp. 165-170).

Een groot deel van deze oorzaken zijn softwarematig op te lossen. Om te controleren of het klantenportaal gebruik maakt van deze software optimalisaties, heb ik het hoofd van het ontwikkelteam gevraagd deze aan te tonen. Hieruit blijkt dat aan de meeste software optimalisaties al is voldaan en dat ze van weinig verbeterend effect zijn voor de performance buiten Europa.

Tijdens dit onderzoek is mij opgevallen dat er twee oorzaken worden aangegeven die niet softwarematig opgelost kunnen worden maar wel een grote impact kunnen hebben op de performance (Souders, 2007, pp. 169-170). Dit zijn een hoge latency en een lage bandbreedte. Deze bevinding heeft de vraag bij mij opgeroepen wat het effect hiervan is op het klantenportaal.

Om hier antwoord op te kunnen geven heb ik in mijn onderzoek eerste de begrippen bandbreedte en latency uitgelegd (Plug things in, z.d.). Een veel gebruikt voorbeeld om deze begrippen uit te leggen is het voorbeeld van een pijp. Bandbreedte wordt vaak aangegeven als de breedte van een pijp, hoe breder de pijp hoe meer er doorheen kan. Latency heeft met de lengte van de pijp te maken; hoe snel kan het van de ene naar de andere kant bewegen. Beide begrippen staan visueel weergeven in figuur 6.1.



Figuur 6.1 - Latency & bandbreedte

Bandbreedte

Uit de deskresearch resultaten van het onderzoek is naar voren gekomen dat een lage bandbreedte meestal wordt veroorzaakt door het medium van de eindgebruiker. Zo zijn

Coaxiale kabels veel minder effectief in het overbrengen van data dan glasvezelkabels. Echter, voor veel bedrijven is het niet mogelijk om het medium van de klant te wijzigen.

Latency
Deze deskresearch resultaten laten ook zien dat een hoge latency veelal komt doordat de afstand tussen de eindgebruiker en de hosting server te groot is. Dit laatste kan wel door een bedrijf worden aangepast door de website op een server dichterbij te plaatsen.

Nadat de mogelijke oorzaken van het performance probleem zijn beschreven, heb ik bekeken wat de invloeden van latency en bandbreedte zijn op het klantenportaal van PostNL. Hierbij heb ik ook onderzocht wat Nalta kan doen om deze invloeden te verminderen.

Om dit te onderzoeken zijn de testresultaten uit paragraaf 3.2 gebruikt. Het is heel goed mogelijk dat de gebruikers gebruik maken van een traag medium. Echter is dit niet af te leiden vanuit de testresultaten omdat Pingdom, de tool die gebruikt is voor het uitvoeren van de testen, voor al haar testen glasvezel gebruikt. Daarbij is het voor Nalta niet mogelijk om het medium van haar klanten aan te passen. Uit de testresultaten kan wel worden geconcludeerd dat het klantenportaal last heeft van een hoge Latency wanneer de afstand tussen de server en de eindgebruiker te groot wordt. Om deze reden heb ik mij in de tweede deelvraag gefocust op oplossingen van het latency probleem. In de volgende paragraaf wordt onderzocht welke mogelijk oplossingen er zijn om dit probleem aan te pakken.

6.3 Deelvraag 2: Mogelijke oplossingen

Voorafgaand aan het onderzoek is de tweede deelvraag als volgt opgesteld: *“Welke bestaande methoden kunnen worden toegepast om het serveren van het klantenportaal te versnellen?”*. Vervolgens is uit de eerste deelvraag naar voren gekomen dat Nalta de latency van het klantenportaal kan verlagen door de website dichter bij haar klanten te plaatsen. De resultaten van de eerste deelvraag geven hiermee direct een oplossingsrichting aan voor de tweede deelvraag. Om deze reden worden er in de tweede deelvraag verschillende methoden onderzocht die zich richten op het sneller serveren van een webapplicatie door de afstand tussen de server en eindgebruiker te verminderen.

Voor het beantwoorden van de tweede deelvraag is er opnieuw gebruik gemaakt van deskresearch om mogelijke oplossingen te vinden voor het latency probleem. In de gevonden literatuur zijn drie veel gebruikte oplossingen naar voren gekomen; back-end replicatie, edge computing of gebruik maken van een CDN (Content Distribution network). De gevonden oplossingen zijn met behulp van de ingewonnen informatie uit de literatuur in het onderzoek uitgelegd.

6.3.1 Back-end replicatie

Back-end replicatie wordt door Microsoft als een oplossing aangegeven om de afstand tussen de eindgebruiker en de server te verminderen (Microsoft, 2020b). Back-end replicatie houdt in dat dezelfde back-end code geplaatst in verschillende continenten met ieder een eigen URL. Op deze manier hoeft een aanvraag niet eerst naar Europa te gaan, maar blijft

deze binnen hetzelfde continent waardoor de latency wordt verlaagd. Echter, om dit principe mogelijk te maken moeten er een groot aantal code wijzigingen worden doorgevoerd. Om deze reden is het geen optie voor PostNL gezien het ingaat tegen TB5 en TB6 zoals deze in paragraaf 5.5 zijn beschreven.

6.3.2 Edge computing

Een andere mogelijke oplossing is door gebruik te maken van edge computing. Bij edge computing wordt er een server on-premise, het terrein van de klant, geplaatst om de berekeningen uit te voeren (Arabi, 2014). Wanneer de data het terrein van de klant niet af hoeft zijn de resultaten ook veel sneller terug bij de klant. Daarbij scheelt het bandbreedte voor de centrale server omdat niet alle data teruggestuurd hoeft te worden naar deze server. Een nadeel hiervan is dat er voor elke klant een edge computing server geïnstalleerd dient te worden. Ook hiervoor moeten er meerdere back-end wijzigingen worden doorgevoerd en daarbij zijn edge computing servers erg kostbaar. Dit is geen optie voor PostNL, gezien het ingaat tegen TB5: “Er worden geen aanpassingen gedaan aan de back-end.”.

6.3.3 CDN

Zoals in de inleiding van de paragraaf staat beschreven kan ook een CDN gebruikt worden om de latency van een website te verlagen. Een CDN repliceert de originele inhoudt en plaatst een kopie hiervan op verschillende servers rond de wereld (Peng, 2018). Wanneer er een aanvraag naar de bestanden wordt gedaan, wordt er gekeken welke server het dichtst bij de gebruiker staat. Vervolgens worden de bestanden vanaf die server gehaald. Echter is uit het onderzoek naar voren gekomen dat een website op een static file hosting server gehost moet kunnen worden voor het gerepliceerd kan worden op een CDN-netwerk. Om dit voor het klantenportaal mogelijk te maken dient het front-end omgezet te worden naar statische bestanden. Dit is een mogelijke oplossing voor Nalta en gaat niet in tegen de vooropgestelde technische beperkingen. Om deze manier toe te passen dient er eerst meer kennis op gedaan te worden door onderzoek naar static file hosting te doen.

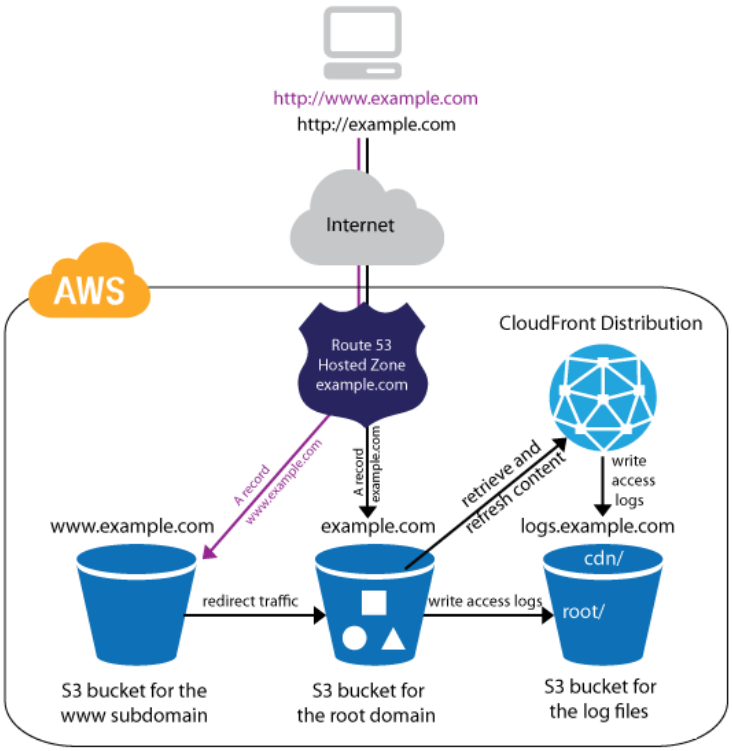
Om een beter beeld te krijgen bij static file hosting en welke applicaties kunnen draaien op een static file hosting server, heb ik hier in mijn onderzoek een hoofdstuk aan gewijd. Dit beslaat ook het onderwerp van de volgende paragraaf.

6.3.4 Static file hosting

Uit de tweede deelvraag is naar voren gekomen dat het gebruik van een CDN een mogelijke oplossing lijkt te zijn voor het performance probleem van het klantenportaal. Echter, bij gebruik van een CDN kunnen alleen statische bestanden gerepliceerd worden en hierdoor dicht bij de eindgebruiker komen te staan. Om een volledige website te repliceren moet hij enkel bestaan uit statische bestanden en dus gehost kunnen worden op een static file hosting server. In het laatste hoofdstuk van mijn onderzoek ben ik ingegaan op static file hosting en hoe deze oplossing toegepast kan worden.

Wederom heb ik gebruik gemaakt van deskresearch om informatie te winnen over de verschillende mogelijkheden van static file hosting. Static file hosting houdt in dat bestanden aangeboden worden aan een gebruiker zonder dat hier extra server-side computaties voor nodig zijn. De werking hiervan staat afgebeeld in figuur 6.2.

Uit de gevonden literatuur zijn drie veelvoorkomende toepassingen van het hosten van een applicatie op een static file hosting server gehaald: static site generators, de website gedeeltelijk laten hosten vanaf een CDN en het gebruik van een modern front-end framework



Figuur 6.2 - Static file hosting (Amazon, z.d.)

Static site generators

Static site generators bieden een ideale oplossing voor websites met dynamische content welke nagenoeg niet wijzigt en dezelfde inhoud levert voor iedere gebruiker. Met dynamische content wordt bedoeld dat de inhoud van de pagina eerst van een centrale server gehaald dient te worden voor hij getoond kan worden. Een blogwebsite gemaakt met behulp van een CMS (Content Management System) is een goed voorbeeld van een website met dynamische content die nagenoeg niet wijzigt. Een gebruiker verwacht altijd de nieuwste lijst van blogs te zien wanneer deze op de website komt. Om deze lijst te tonen wordt er een request naar het CMS gedaan. In deze situatie kan een static site generator gebruikt worden om de volledige site om te zetten in statische bestanden. Wanneer er een nieuwe blog wordt geplaatst worden deze statische bestanden vervolgens op een cloud storage geplaatst om hem zo te laten repliceren via een CDN en hierdoor goedkoop te hosten. Deze oplossing kan echter niet gebruikt worden voor het klantenportaal van PostNL omdat de inhoud van het klantenportaal vaak wijzigt en verschillend is per gebruiker.

Gedeeltelijk laten hosten op een CDN

Een andere oplossing is om een deel van de website te laten repliceren via een CDN-netwerk. Hierbij worden alle statische bestanden, denk hierbij aan afbeeldingen en JavaScript bestanden, geplaatst op een static file hosting server. Op deze manier kunnen die bestanden snel geleverd worden aan de eindgebruiker. Enkel de statische bestanden

laten repliceren vanaf een CDN is wel mogelijk echter moet het volledige scherm vervolgens alsnog op de centrale server worden opgebouwd waardoor het weinig optimalisatie zal opleveren.

Modern front-end framework
Een modern front-end framework behoort tot de laatste mogelijkheid om dit probleem op te lossen. De meeste front-end frameworks kunnen namelijk gehost worden vanaf een static file hosting server. De frameworks genereren vaak single page applications en hier heb ik mij dan ook op gefocust tijdens het onderzoek.

Om erachter te komen welke moderne front-end frameworks toepasbaar zijn voor het klantenportaal is er opnieuw gebruik gemaakt van deskresearch. Tijdens het vergaren van informatie zijn er twee veel voorkomende framework technieken ontdekt voor het ontwikkelen van single page applications; JavaScript en WebAssembly.

JavaScript frameworks, zoals Angular en Vue.js, maken gebruik van JavaScript om per request te bepalen wat er aan de gebruiker getoond moet worden zonder dat deze het volledige scherm van de server op moet halen. Het ontwikkelen van single page applications met JavaScript is een trend die ondertussen al een aantal jaren aan het groeien is. Mede dankzij het hiervoor benoemde voordeel. Ook voor het klantenportaal lijkt het een mogelijke oplossing te zijn om het front-end om te zetten met behulp van een JavaScript framework.

Een andere manier om single page applications te ontwikkelen is door gebruik te maken van WebAssembly. WebAssembly is nog redelijk nieuw en krijgt pas sinds anderhalf jaar veel aandacht doordat het ontzettend snel is. WebAssembly maakt veel beter gebruik van de lokale resources van de browser doordat het assembly code genereert. Assembly code draait op een laag level net boven de hardware en kan hierdoor efficiënt bepalen waar welk geheugen gealloceerd dient te worden. Mijn onderzoek wijst twee verschillende oplossingen aan die gebruik te maken van de snelheid van WebAssembly: Rust en Blazor.

Rust is een efficiënte programmeertaal die gecompileerd kan worden naar WebAssembly code waardoor het snel intensieve berekeningen op de browser van de eindgebruiker uit kan voeren (Robert, 2019). Het is voor Nalta zeker mogelijk om het klantenportaal om te zetten naar Rust echter heeft geen enkele ontwikkelaar binnen het Natla ontwikkelteam hier ervaring mee en staat dit ook niet op de planning (Interne communicatie, 2020).

Nadat deze informatie bekend is geworden blijkt Blazor een uitstekende oplossing te zijn voor het klantenportaal. Blazor biedt net als Rust de optie op ontzettend snel intensieve berekeningen op de browser van de eindgebruiker uit te voeren. De implementatie van Blazor is echter wel anders dan bij Rust. Bij Blazor is alleen de .NET runtime geschreven in WebAssembly en worden de standaard .NET dll’s hierdoor uitgevoerd. Deze dll’s zijn geschreven in C# en alle kunnen files omgezet worden naar statische files die gehost kunnen worden op een static file hosting server. Zoals beschreven in het verslag heeft Blazor naast de WebAssembly variant ook een Blazor Server variant waarbij alle schermen worden opgebouwd vanaf de server. Echter kan dit niet op een CDN geplaatst worden.

Zowel JavaScript als WebAssembly frameworks zijn toepasbaar voor het front-end van het klantenportaal. Nalta heeft in het verleden al eens applicaties ontwikkeld met JavaScript

frameworks en wil de WebAssembly techniek nauw blijven volgen. In het volgende hoofdstuk staat de conclusie van het onderzoek beschreven.

6.5 Conclusie

In het onderzoeksverslag heb ik mijn conclusie beschreven beschreven volgens de richtlijnen van Scribbr (Swaen, 2020). In deze conclusie heb ik de opgestelde hoofdvraag beantwoord door middel van de eerder beschreven tussentijdse conclusies. Per deelvraag heb ik nog eens kort de resultaten uitgelegd en heb ik het effect wat ze hebben op de hoofdvraag hiervan aangeven.

Het antwoord van de eerste deelvraag geeft aan dat latency een oorzaak is van de performance problemen van het klantenportaal buiten Europa. Om dit te verbeteren kan Nalta de latency verlagen door de afstand tussen de server en de eindgebruiker te verkleinen. Vervolgens zijn in paragraaf 6.3 de mogelijke oplossingen beschreven die het mogelijk maken om deze afstand te verkleinen. Van de gevonden oplossingen vallen back-end replicatie en edge computing af doordat ze niet voldoen aan de vooropgestelde technische beperkingen. De enige gevonden toepasbare methode voor het klantenportaal is om de website te laten repliceren op een CDN-netwerk. In paragraaf 6.4 zijn de manieren onderzocht om een applicatie te laten hosten op een static file hosting server zodat de website gerepliceerd kan worden op een CDN-netwerk.

Uit dit onderzoek blijkt dat static file generators niet toepasbaar zijn door de complexiteit van het klantenportaal. Alleen de statische bestanden laten repliceren op een CDN is een optie maar lost het probleem maar gedeeltelijk op. Voor een completere oplossing dient het klantenportaal met behulp van een modern front-end framework omgezet te worden naar een single page application en deze in zijn geheel op een CDN-netwerk te plaatsen.

Aan de hand van de bovenstaande resultaten is er een hypothese getrokken uit het onderzoek, welke getest dient te worden met behulp van een prototype. De hypothese die volgt uit het onderzoek luidt: “*Door middel van een modern front-end framework, in combinatie met een CDN, wordt de performance van het klantenportaal van PostNL verbeterd.*”

6.6 Aanbeveling

In het laatste hoofdstuk heb ik een aanbeveling gedaan waarbij het onderzoek en de requirements in het achterhoofd zijn gehouden. In mijn aanbeveling heb ik de verschillende oplossingen tegen elkaar afgewogen en heb ik beschreven waarom deze wel of niet gebruikt kunnen worden als oplossing voor het PostNL klantenportaal. Hierbij raad ik aan om gebruik te maken van een WebAssembly framework omdat het de lokale resources van de browser optimaal kan benutten. Zoals in paragraaf 6.4 staat beschreven heeft het onderzoek Rust en Blazor uitgewezen als vergevorderde WebAssembly ontwikkelmethoden.

Applicaties die Rust gebruiken zijn hierbij ontwikkeld in de taal Rust. Dit is een taal waar Nalta nog geen ervaring mee heeft en in een gesprek met de heer M. Beenes bleek dat dit vooralsnog ook niet op de planning stond. Blazor is een nieuw WebAssembly framework ontwikkeld door Microsoft waarbij er geprogrammeerd kan worden in de taal C#. Doordat de back-end ontwikkeld is in C#, is een wens van het ontwikkelteam dat ze het front-end ook het liefst in C# zouden ontwikkelen. Het argument dat hierbij gegeven is, is dat bepaalde modellen en functionaliteiten hergebruikt kunnen worden en maar eenmalig geschreven hoeven te worden.

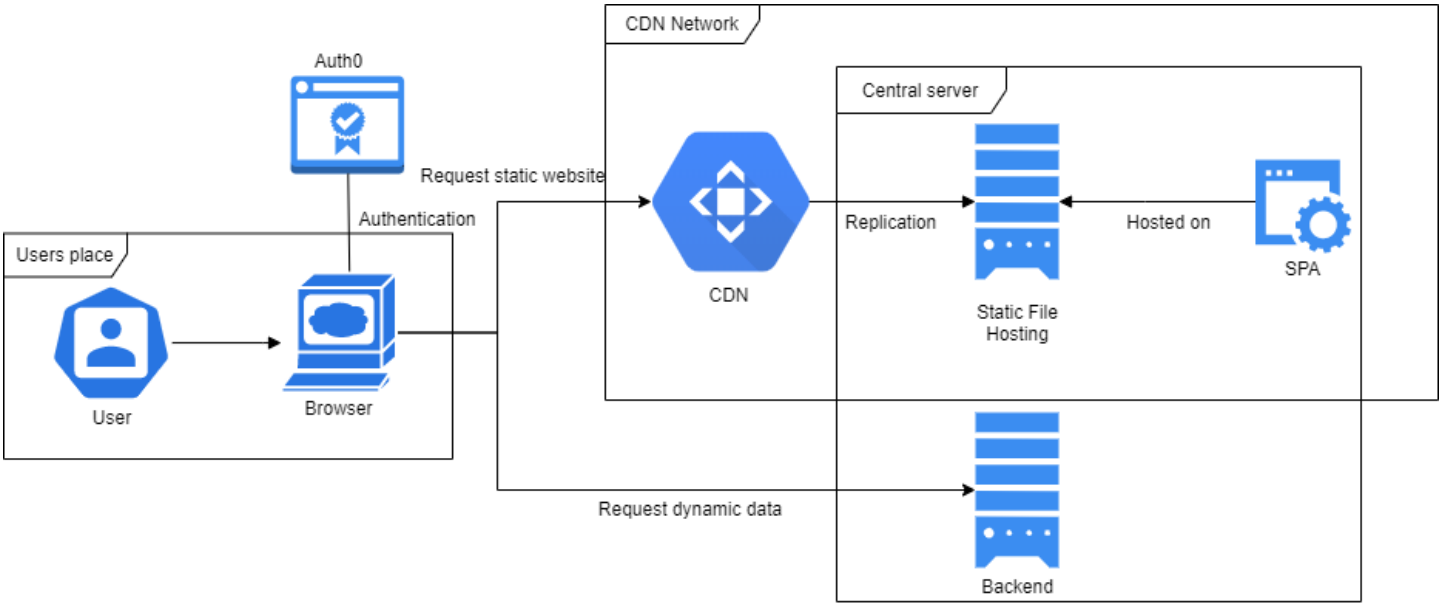
Nadat deze informatie bekend is geworden, blijkt Blazor een uitstekende oplossing te zijn voor het klantenportaal. Blazor biedt net als Rust de optie om ontzettend snelle intensieve berekeningen op de browser van de eindgebruiker uit te voeren. Blazor staat het developers toe om in C# te programmeren en alle files om te zetten naar statische files die gehost kunnen worden op een static file hosting server. Om deze reden wordt op basis van de hypothese aanbevolen om Blazor als techniek te gebruiken voor het prototype.

6.7 Toepasbaarheid op PostNL

In de aanbeveling van mijn onderzoek raad ik aan om een prototype in Blazor te ontwikkelen om optimaal gebruik te maken van de lokale resources. Wanneer naar de situatie van het klantenportaal wordt gekeken is dit zeker een goede oplossing omdat er verschillende zware berekeningen in de code worden uitgevoerd om de resultaten van de API om te zetten voordat ze bruikbaar zijn voor de schermen. Daarbij kan zoals aangegeven door de ontwikkelaars een groot deel van de huidige logica worden hergebruikt. In de volgende hoofdstukken worden de verschillende tussenproducten beschreven die controleren of de hypothese van toepassing is op het klantenportaal. Tijdens deze ontwikkelfase wordt de Kanban methode gebruikt zoals deze in paragraaf 4.2 ‘fasering’ is beschreven.

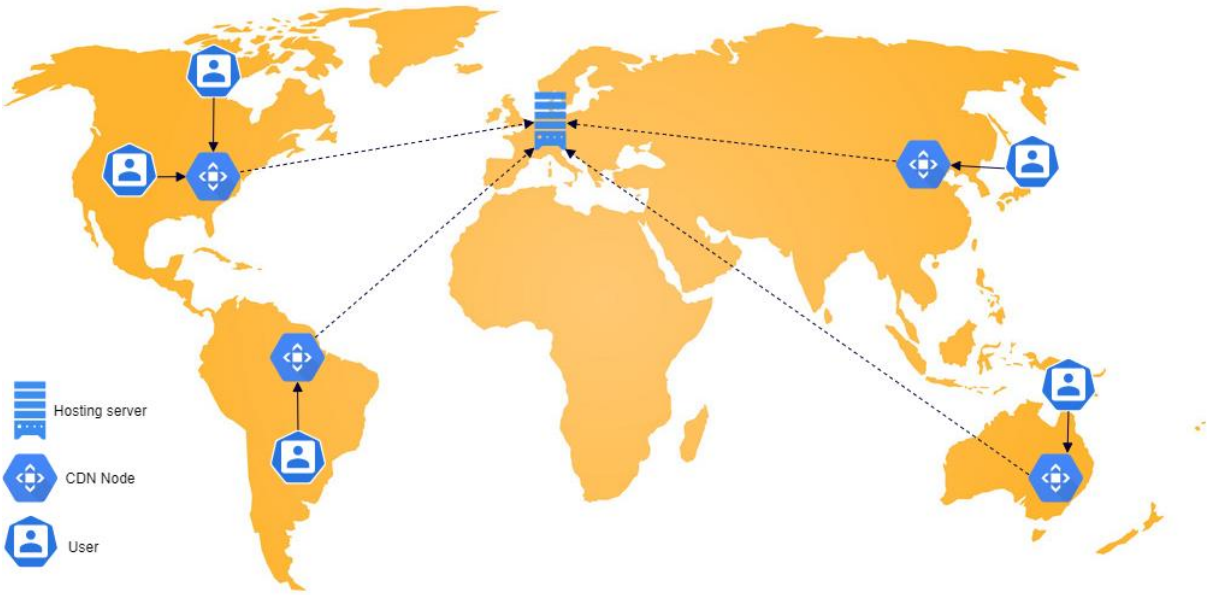
7. Ontwerp

Om aan te tonen dat de gevonden hypothese, zoals gesteld in paragraaf 6.4 ‘Conclusie’, juist is, dient deze gecontroleerd te worden aan de hand van een prototype. Voordat dit prototype gemaakt kan worden ben ik eerst begonnen met maken van een ontwerp. Aan de hand van dit ontwerp ben ik begonnen met gericht ontwikkelen van een prototype. Allereerst heb ik de infrastructuur in kaart gebracht. Deze is te zien in figuur 7.1. De grootste verandering ten opzichte van de huidige infrastructuur is dat de website nu gerepliceerd wordt op een CDN-netwerk.



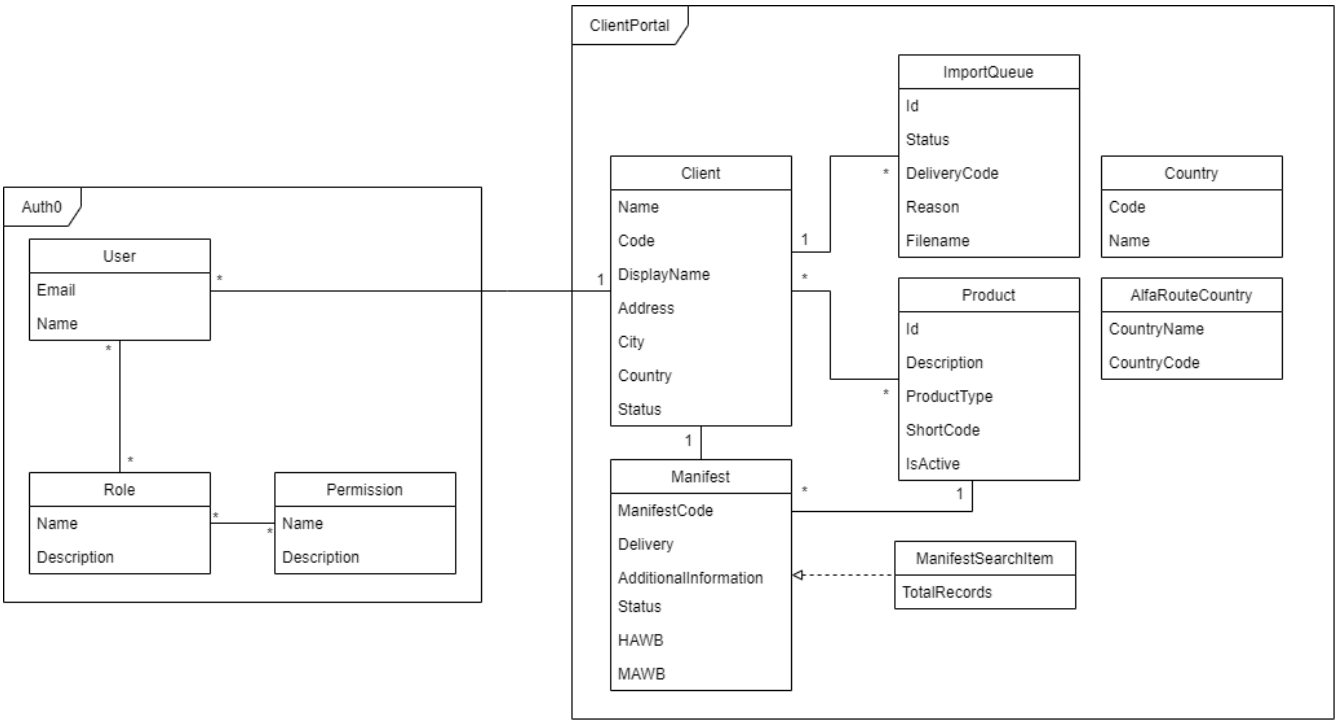
Figuur 7.1 - Infrastructuur ontwerp

Om de effectiviteit van een CDN visueel inzichtelijk te maken, heb ik figuur 7.2 gemaakt welke de werking van een CDN-netwerk laat zien. Bij het diagram is een legenda toegevoegd welke aantoont waar de verschillende iconen voor staan. De connectie tussen de CDN-servers en de static file hosting server staat met stippellijntjes aangegeven omdat de website niet altijd opgehaald hoeft te worden vanaf de static file hosting server wanneer deze nog gecached staat op de CDN-server.



Figuur 7.2 - CDN infrastructuur

Na het opstellen van de herziende infrastructuur heb ik aan de hand van de UML-modelleertaal een versimpelde versie van het klantenportaal klassendiagram opgesteld. Deze is te zien in figuur 7.3 (Fowler, Scott, Kobryn, & Safari Tech Books Online, 2004). Ik heb ervoor gekozen om dit met UML te doen aangezien dit een voor mij bekende techniek is welke is aangeleerd tijdens de opleiding Software Engineering. Hierdoor ken ik de voordelen van deze techniek. Het geeft een duidelijk overzicht over hoe de applicatie er uiteindelijk uit komt te zien. Dit klassendiagram wordt als basis gebruikt voor de SPA, Single Page Application, van figuur 7.1.



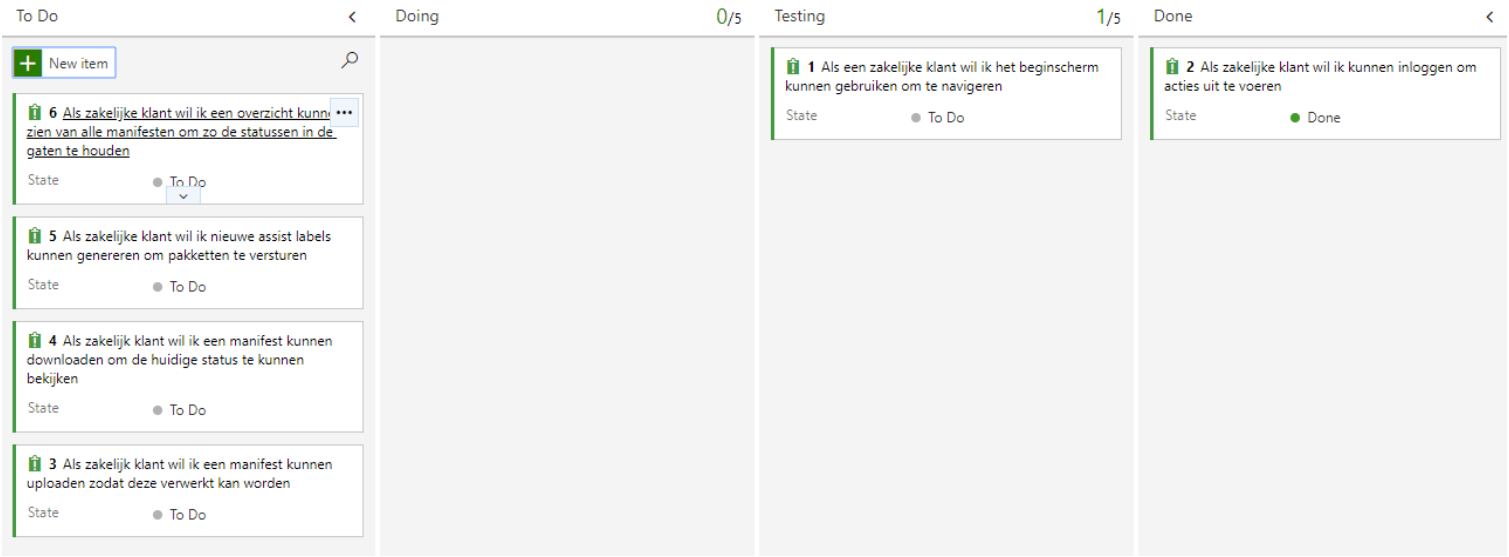
Figuur 7.3 - Klassendiagram

8. Blazor prototype

Na het opzetten van het ontwerp voor het prototype, ben ik aan de slag gegaan met het prototype zelf. Zoals in paragraaf 6.5 is aanbevolen, is het prototype gemaakt met behulp van het Blazor framework. In de volgende paragrafen staat meer informatie over hoe ik hier mee om ben gegaan.

8.1 Planning

Tijdens het ontwikkelen van het prototype heb ik gebruik gemaakt van de Kanban techniek zoals deze in paragraaf 4.1 staat beschreven. Om te beginnen is er op Azure DevOps een bord aangemaakt. Vervolgens zijn alle use cases van het requirementsrapport omgezet naar issues en op het bord geplaatst. Hierna zijn alle issues geprioriteerd op basis van de beschreven prioritering in het requirementsrapport. Een voorbeeld van hoe het bord er uit zag tijdens het development proces is in figuur 8.1 weergegeven.



Figuur 8.1 - Kanban bord

8.2 Aanpak van het prototype

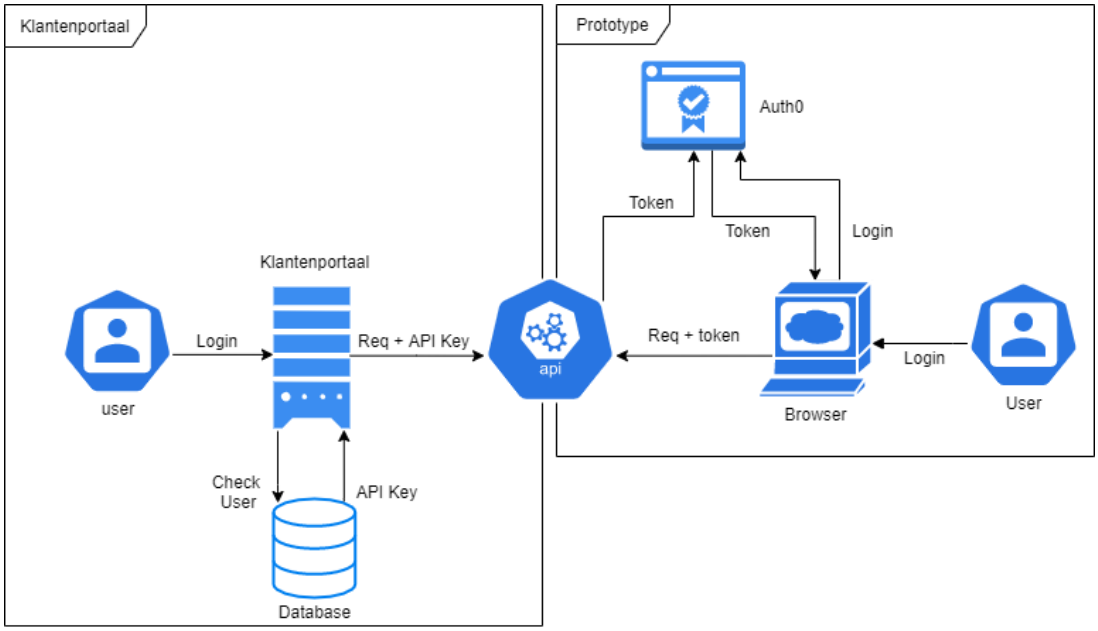
Na het invoeren van de verschillende issues op het Kanban bord, ben ik aan de slag gegaan met het ontwikkelen van het prototype. Voorafgaand aan de afstudeeropdracht heb ik al de benodigde informatie opgedaan om in C# te kunnen programmeren. Daarentegen was er met het WebAssembly framework Blazor door mij nog geen informatie opgedaan. Op het moment van schrijven was Blazor dan ook in de laatste preview fase en wordt halverwege mei volledig uitgebracht. Om informatie over Blazor op te doen, heb ik gebruik gemaakt van de originele Blazor documentatie welke wordt aangeboden door Microsoft (Microsoft, 2020a). Naar mijn mening helpt het lezen van officiële documentatie vaak om de achterliggende gedachte van de techniek te ontdekken Dit is dan ook de reden dat ik ervoor heb gekozen om de officiële documentatie te lezen.

Na het lezen van deze documentatie ben ik begonnen met het exploreren van het voorbeeldproject dat wordt aangeboden door Microsoft. Dit voorbeeldproject wordt voor je aangemaakt wanneer je tijdens het maken van het project in Visual Studio het Blazor template selecteert. Het voorbeeldproject bevat een groot deel van de functionaliteit en laat op een begrijpbare manier de werking hiervan zien. Dit voorbeeld heeft mij geleerd op welke manier de verschillende componenten van elkaar gescheiden dienen te worden en hoe de SOLID-principes (Oloruntoba, 2015) in Blazor kunnen worden toegepast. Via deze principes is ook het prototype in elkaar gezet.

8.3 Aanpassingen ten opzichte van oude klantenportaal

Om alle functionaliteit van het originele klantenportaal te behouden, is de achterliggende code vaak wel gewijzigd. Dit komt voornamelijk doordat het prototype een single page application is. Wanneer een kwaadwillend persoon echt wil kan die de volledige code achterhalen uit de statische bestanden van een single page application. Om deze reden mogen er geen belangrijke waardes in de code staan om een connectie met een database of dergelijke te maken.

Een goed voorbeeld van een code wijziging is de login functionaliteit. Het originele klantenportaal had een eigen database waarin alle gebruikers werden opgeslagen en geauthentiseerd. Dit is in een single page application niet mogelijk. Hierom maakt Nalta voor single page applications gebruik van Auth0 om haar gebruikers te authentiseren. Dit is ook de manier die ik gebruikt heb voor het implementeren van authenticatie. Via Auth0 kan een gebruiker inloggen, waar Auth0 vervolgens een beveiligde token terugstuurt die gevalideerd kan worden. Het token wordt ook gebruikt om naar de API te sturen waarbij de API vervolgens weer bij Auth0 kan valideren of de gebruiker is ingelogd. Beide manieren kunnen tegelijkertijd gebruikt worden. Het verschil tussen de twee methoden is te zien in figuur 8.2.



Figuur 8.2 - Authenticatie

Ook is de achterliggende functionaliteit van het up- en downloaden van bestanden gewijzigd. Het klantenportaal maakt gebruik van een Azure Storage account om bestanden tussen de API en het portaal uit te wisselen. Wanneer een manifest wordt aangevraagd via de API, maakt de API een Excel aan en plaatst deze op de Azure Storage en stuurt de locatie op de storage naar het portaal zodat deze hem aan de gebruiker kan serveren. Ook dit is, vanwege de wachtwoorden die nodig zijn om een connectie te maken, in een single page application niet mogelijk. Om deze functionaliteit aan te kunnen bieden wordt er via de API nog steeds een manifest op het Azure Storage account geplaatst, echter wordt hier direct een downloadlink bij gegenereerd. Deze downloadlink kan één keer gebruikt worden en is tien minuten geldig. De link wordt teruggestuurd naar het prototype en wordt vanaf daar gebruikt om het bestand te downloaden in de browser van de gebruiker.

8.4 Problemen tijdens het ontwikkelen

Tijdens het ontwikkelen van het prototype ben ik ook tegen een aantal problemen aangelopen welke grote negatieve invloed op de performance bleken te hebben.

Het eerste probleem waar ik tegenaan ben gelopen is het implementeren van een filtreerbaar dropdown. In het originele klantenportaal wordt hiervoor een jQuery library gebruikt genaamd Select2. Door de werking van het component model van Blazor was het niet mogelijk om deze library te gebruiken. Via google ben ik opzoek gegaan naar andere libraries die deze functionaliteit aanbieden. Uit dit onderzoek zijn SyncFusion en Telerik UI naar voren gekomen. Telerik UI is echter ontzettend duur en hierdoor niet gewenst vanuit Nalta. SyncFusion kan wel gratis gebruikt worden. Om deze reden heb ik besloten om SyncFusion te implementeren. Nadat ik de dropdown heb geïmplementeerd heb ik de performance hiervan onderzocht. Hieruit bleek dat het tot wel twee à drie seconden te duren voordat het component te zien was. Om achter de oorzaak te komen, heb ik de forums van SyncFusion geraadpleegd. Hieruit bleek dat het een bug in de implementatie van Syncfusion bleek te zijn. Om dit probleem op te lossen heb ik zelf een filtreerbaar dropdown component geschreven welke binnen enkele milliseconden werd geladen en er precies zo uitziet als in het originele klantenportaal.

Een ander groot performance probleem waar ik tegenaan ben gelopen is het deserialiseren van JSON-objecten (JavaScript Object Notation). Na een aantal tests bleek het deserialiseren in de Blazor applicatie veel langer te duren dan in de .NET Core applicatie. Dit is opmerkelijk gezien het deserialiseren volgens dezelfde manier gebeurt, de manier die wordt aangeraden vanuit Microsoft. Ook hier is de oorzaak in de forums gevonden en bleek dat de onderliggende code nog niet geoptimaliseerd was. Hierom heb ik een onderzoek uitgevoerd om te controleren of alternatieve libraries hetzelfde probleem ondervonden. Op deze manier ben ik erachter gekomen dat de Utf8Json library meer dan elf keer sneller deserialiseerde dan de standaard van Microsoft en heb ik verder in het project ook de Utf8Json library gebruikt. In paragraaf 9.3 staat hier meer informatie over.

8.5 Deployment

Gezien het prototype via de Kanban methode is ontwikkeld, heb ik ook via de Kanban methode de geschreven software gepubliceerd. Nadat een issue is afgerond, wordt deze op de private git omgeving van Nalta geplaatst en door middel van Azure DevOps gepubliceerd.

Op Azure DevOps heb ik een build pipeline aangemaakt welke de code van BitBucket haalt en controleert op syntax fouten. Wanneer de code geen syntax fouten bevat wordt de code getest door middel van de geschreven unit tests. In de laatste stap wordt er een versie gebouwd die gepubliceerd kan worden en deze wordt klaargezet voor de release pipeline. In de release pipeline haal ik de versie van de build pipeline op en zet ik hem op de static file hosting server. Daarbij zorg ik ervoor dat het CDN-netwerk de oude versie verwijderd.

Zoals in figuur 7.1 te zien is, was het initieel de bedoeling om het prototype via IIS te laten hosten op de server van de centrale API. Uit een gesprek met de heer M. Beenes is echter gebleken dat het niet gewenst is om dit prototype op dezelfde server als die van de centrale API te hosten. Dit zou betekenen dat het prototype op een eigen server gehost moet worden. Om deze reden ben ik opzoek gegaan naar alternatieven. In dit onderzoek ben ik erachter gekomen dat het ook mogelijk is om een static site te laten hosten op een Azure Storage account, de kosten worden vergeleken in tabel 8.1. Beide opties heb ik met de heer M. Beenes besproken. Uiteindelijk is er besloten om het prototype te hosten op een Azure Storage.

Naam	Prijs per maand		Naam	Prijs per maand
Storage account	$0.50		App service	$54.75
Azure CDN	$1.61		Azure CDN	$1.61
Totaal:	$2.11		Totaal:	$56.36

Tabel 8.1 - Hosting kosten

In tabel 8.1 is ook te zien dat in beide gevallen het Azure CDN wordt gebruikt. Het Azure CDN is gekozen omdat het goedkoop is en gemakkelijk geconfigureerd kan worden met andere applicaties. Daarbij wordt het Azure CDN gehost op Azure wat Nalta gebruikt als standaard voor het hosten van websites.

9. Testverslag

Tijdens het ontwikkelen van software dient deze software ook getest te worden. De wijze waarop ik heb getest staat uitgebreid beschreven in de bijlage op pagina 113 het testverslag. Nadat de software voor een issue van het Kanban bord is ontwikkeld ben ik direct testen gaan schrijven om te valideren of de component voldoet aan mijn verwachtingen. Om te valideren of de geschreven voldoet aan de verwachten zijn er verschillende testmethoden uitgevoerd. Deze test methoden staan in de volgende paragrafen beschreven. De testen zijn uitgevoerd op een uitgewerkt prototype.

9.1 Integratie testen

Volgens Technosoft vallen integratie testen onder de functionele testen. Onder functioneel testen verstaan zij: “Het testen van software op basis van de functionele omschrijving van de software om de kwaliteit hiervan te beoordelen en mogelijke fouten op te sporen.” (Gucinsky, z.d.). En andere term voor integratie testen is usability testen. Deze manier van testen wordt gebruikt om te achterhalen of de software reageert zoals ervan verwacht wordt. Op deze manier kan BR1 zoals beschreven in paragraaf 5.2 worden aangetoond.

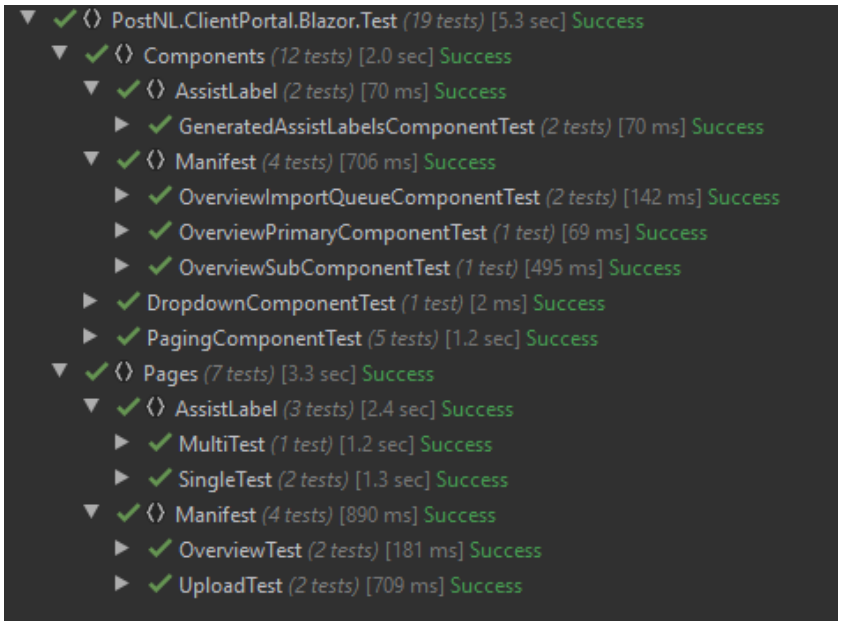
De manier waarop ik de integratie testen heb uitgevoerd is als volgt. Aan de hand van de vooropgestelde functionele requirements zijn de resultaten van deze functionaliteiten van het originele klantenportaal opgeslagen. Vervolgens is heb ik in het prototype de software ontwikkeld welke dezelfde functionaliteit zou moeten bieden als van het originele klantenportaal waarmee er is voldaan aan de geformuleerde functionele eisen. Wanneer ik dacht aan de requirements te hebben voldaan werden ook deze resultaten van het prototype opgeslagen. Beide resultaten zijn met elkaar vergeleken om te controleren of de software in het prototype voldoet aan mijn verwachtingen. Als deze hieraan voldeed ben ik gaan onderzoeken welke unit testen er geschreven dienen te worden zodat de software automatisch getest kan worden. Deze worden in de volgende paragraaf beschreven.

9.2 Unit tests

Naast integratie testen is ervoor gekozen ook unit tests te schrijven voor het prototype. Unit tests vallen net als integratie testen onder de functionele testen. Er is voor gekozen om unit tests te schrijven voor het prototype gezien ze snel feedback geven op de geschreven code. Het is hierom ook een best practice om zoveel mogelijk unit tests te schrijven en later pas te kijken naar end-to-end tests. End-to-end tests doen er langer over om uit te voeren waardoor je ook pas later feedback ontvangt. Door de unit tests wordt voldaan aan het vooropgestelde niet-functionele requirement NF4: “De UI van applicatie moet automatisch getest kunnen worden”.

Voor dit project is gebruik gemaakt van het test framework bUnit. bUnit is een uitbreiding op xUnit. Hiervoor is gekozen omdat Nalta momenteel al meerdere applicaties test met xUnit en het een duidelijk manier van testen geeft. xUnit is een standaard test framework voor C#, door de uitbreiding van bUnit kan niet alleen ‘gewone’ C# code getest worden, maar ook Blazor componenten. In het geheugen kunnen de Blazor componenten, het UI, worden opgebouwd om hier vervolgens de verschillende testen op uit te voeren. Dit is een verschil

ten opzichte van de testen die in .NET Core worden uitgevoerd. In .NET Core worden de automatische testen uitgevoerd door middel van een JavaScript UI framework genaamd NightmareJS gezien in-memory testen hiervoor niet mogelijk is. De resultaten van deze test zijn te zien in figuur 9.1.



Figuur 9.1 - Unit tests

9.3 Performance tests

Zoals beschreven in paragraaf 3.2 ben ik in de oriëntatiefase begonnen met het uitvoeren van een performance test via de website Pingdom.com. Pingdom is gekozen omdat ze een groot aantal servers verspreid over de wereld hebben staan die gebruikt kunnen worden om de snelheid van een website te testen vanuit verschillende locaties. Daarbij wordt Pingdom al voor verschillende applicaties van Nalta gebruikt om te controleren of hij nog benaderbaar is. Na het ontwikkelen van het prototype is deze test nogmaals uitgevoerd om te controleren of het ook daadwerkelijk voldoet aan de verwachtingen van de onderzoekshypothese.

Zoals beschreven in paragraaf 3.2 laten Bland en Altman weten dat een test minstens tweemaal voor verschillende onderwerpen moet worden uitgevoerd om een betrouwbare standaarddeviatie te verkrijgen. Mijn uitgevoerde tests zijn per continent tienmaal uitgevoerd om een nog accurater resultaat te behalen. In tabel 9.1 staat per testlocatie het gemiddelde van de testen weergeven met tussen haakjes de standaarddeviatie. Tevens is met rood de langzaamste laadtijd weergeven en in het groen de snelste. Een hogere laadtijd is in dit geval slechter. Dit betekent dat gebruikers langer moeten wachten op de website en daarom de website als langzaam ervaren.

Regio	.NET 4.8	.NET Core	Blazor	Blazor + CDN
Europa, Duitsland, Frankfurt	595 ms (σ 93)	540 ms (σ 97)	510 ms (σ 92)	**158 ms** (σ 29)
Europa, Engeland, Londen	607 ms (σ 97)	565 ms (σ 101)	540 ms (σ 98)	174 ms (σ 35)
Noord-Amerika, USA, San Francisco	3021 ms (σ 314)	2766 ms (σ 267)	2654 ms (σ 97)	218 ms (σ 30)
Azië, Japan, Tokio	5196 ms (σ 480)	4286 ms (σ 445)	3676 ms (σ 314)	264 ms (σ 50)
Oceanië, Australië, Sydney	**5974 ms** (σ 557)	5214 ms (σ 521)	3918 ms (σ 354)	294 ms (σ 57)

Tabel 9.1 - Nieuwe laadsnelheid

De test laat zien dat bij gebruik van alleen Blazor de website binnen Europa iets sneller laadt dan .NET Framework 4.8 en .NET Core voor zowel binnen als buiten Europa. Wanneer het Azure CDN-netwerk wordt gebruikt om de website te repliceren laadt de website in elk continent ontzettend snel. Nog steeds zit er een klein verschil in laadsnelheid tussen binnen en buiten Europa. Dit is dan weer te verklaren doordat er een script van Auth0 moet worden ingeladen die niet op het Azure CDN staat. Hier heeft Auth0 een eigen CDN-netwerk voor. Ze hebben in dit netwerk minder CDN-servers verspreid staan over de wereld waardoor het iets langer duurt om het script op te halen.

JSON library vergelijking

Pingdom kon echter niet gebruikt worden voor pagina's waarbij een gebruiker geauthentiseerd dient te zijn. Hierom heb ik voor die pagina's handmatig testen uitgevoerd. Een veelvuldig gebruikte pagina van het klantenportaal waarbij een gebruiker geauthentiseerd dient te zijn is de 'manifest overview' pagina. Op deze pagina ben ik iets opmerkelijks tegengekomen. In de Blazor applicatie bleek het tonen van de manifesten langer te duren dan in .NET Core. Omdat dit ingaat tegen de verwachtingen heb ik onderzoek naar de herkomst van dit probleem gedaan, zoals eerder aangegeven in paragraaf 8.4. Het probleem bleek in het deserialiseren van JSON Objecten te zitten. Tijdens het onderzoek heb ik verschillende JSON deserialisers met elkaar vergeleken met behulp van de C# stopwatch class om te timen hoe lang de libraries er over deden om het object te deserialiseren. De resultaten van dit onderzoek staan in tabel 9.2. In de resultaten van deze test is de eerste laadtijd meegenomen en het gemiddelde van de deserialisatie tijd, welke tussen haakjes staat.

	System.Text.Json	Newtonsoft.Json	Utf8Json	Jil
Blazor	113 ms (196)	63 ms (334)	10 ms (260)	12 ms (2725)
.NET Core	0.68 ms (478)	0.92 ms (483)	0.14 ms (317)	0.25 ms (882)

Tabel 9.2 - JSON vergelijking

Tijdens het testen leek er een groot verschil te zitten tussen de eerste keer dat het model werd omgezet en de daaropvolgende keren. Dit is te verklaren doordat Blazor en .NET Core gebruik maken van JIT (Just-In-Time) compilatie. Hierbij wordt de code pas gecompileerd wanneer het gebruikt gaat worden. Echter, wanneer de models tijdens het laden eenmalig

worden gedeserialiseerd kan de applicatie deze manier onthouden en verloopt het deserialiseren verder in de applicatie veel sneller. De eerste keer dat een JSON-object wordt gedeserialiseerd is “System.Text.Json” het snelste, maar in alle daaropvolgende deserialisaties is Utf8Json het snelste. Tijdens het opstarten van Blazor kan een dergelijk model al een keer in de achtergrond gedeserialiseerd worden waardoor altijd de snelle variant van het deserialiseren wordt gebruikt, waarin Utf8Json ruim elf keer sneller is dan “System.Text.Json”. Wel is Blazor, zelfs met Utf8Json, langzamer in het deserialiseren dan .NET Core. Met behulp van Utf8Json is het tonen van de schermen in alle gevallen geoptimaliseerd en laden de schermen sneller dan in de .NET Core variant. Deze resultaten tonen aan dat de onderzoekshypothese niet is verworpen. Daarbij tonen de performance testen aan dat aan BR2 en BR3 is voldaan. Echter was ik nog niet tevreden met resultaten en ben ik verder onderzoek gaan doen.

Benodigde data
Het bleek dat het JSON-response veel meer informatie bevat dan dat het klantenportaal daadwerkelijk nodig heeft. Van de 18 kB aan data, wat verstuurd wordt door de API, wordt er slechts 3 kB gebruikt. Het deserialiseren van enkel de benodigde data duurt 1,36 ms in Blazor. Dit is nog eens meer dan zeven keer sneller dan de situatie in Blazor met de Utf8Json library.

Binair formaat
Na het aanpassen van de deserialisatie library en het vooraf omzetten van een model werd de website in alle gevallen sneller getoond in de Blazor applicatie. Tijdens dit onderzoek bedacht ik me dat het opsturen van het object in een binair formaat mogelijk nog meer verbeteringen in snelheid oplevert gezien JSON is geoptimaliseerd voor JavaScript terwijl C# een binair formaat makkelijker kan deserialiseren. Om te controleren of dit ook daadwerkelijk een performance optimalisatie oplevert voor het klantenportaal heb ik hier onderzoek naar gedaan. Uit dit onderzoek zijn twee veel voorkomende methoden naar voren gekomen die data versturen in een binair formaat, Protocol buffers en MessagePack.

Protocol buffers, ook wel ProtoBuf genoemd, is een techniek welke is ontwikkeld door Google (Google, 2019). Protocol buffers zetten de data om in een kleine en gestructureerde binaire data. Bij het gebruik van protocol buffers worden de verschillende protocol buffer messages types opgeslagen in .proto bestanden. Deze bestanden zijn zo opgesteld dat vanuit deze bestanden gemakkelijk de bijbehorende requesten gegenereerd kunnen worden. Dit blijkt ook voor C# mogelijk te zijn met een ProtoBuf library.

Een andere manier om modellen in een binair formaat op te sturen is door gebruik te maken van MessagePack. Zoals MessagePack zelf zegt: “It's like JSON. but fast and small.”. MessagePack verschilt van protocol buffers doordat in het model zelf wordt toegevoegd wat voor type elke property heeft. MessagePack geeft de gebruiker twee opties om de data op te zetten. De eerste manier is om net als bij JSON alle velden een naam te geven en hier een value bij plaatsen, deze wordt in tabel 4.2 op de derde rij aangegeven. Hierdoor is het request compacter, maar nog niet optimaal. Een tweede manier is om alle waardes van het object in een array achter elkaar te plaatsen, deze wordt in tabel 4.2 op de vierde rij aangegeven. Dit is nog compacter dan de eerste manier.

Test
Om te bepalen welke manier de meeste optimalisatie oplevert, is er een test uitgevoerd voor elk van de gevonden methoden. Zoals eerder is beschreven maakt Blazor gebruik van JIT, om deze reden heb ik ook de eerste laadtijd in de tabel meegenomen. De test is uitgevoerd op de dataset met alleen de benodigde data. De test is na de eerste keer nog tien keer uitgevoerd. Het gemiddelde van de testen is opgenomen in tabel 4.2 met de standaarddeviatie tussen de haakjes.

Naam	Eerste laadtijd	Opvolgende laadtijden
Utf8Json	260 ms	1.36 ms (σ 0.416)
Protocol buffers	190 ms	2.76 ms (σ 2.436)
MessagePack (string)	240 ms	1.1 ms (σ 0.797)
MessagePack (array)	220 ms	0.4 ms (σ 0.071)

Tabel 9.3 - deserialisatie tijden

De resultaten van tabel 9.3 wijzen uit dat het opsturen van het model in een binair formaat niet altijd sneller hoeft te zijn. Wanneer protocol buffers gebruikt worden is de eerste laadtijd het snelst, maar is deze in het vervolg langzamer dan Utf8Json. Dit is mogelijk te verklaren doordat Google de libraries voor C++ heeft geschreven en ik een library voor C# dien te gebruiken. MessagePack is langzamer dan protocol buffers in de eerste laadtijd, maar sneller dan Utf8Json. Wanneer de manier gebruikt wordt om alle waardes in een array achter elkaar te zetten blijkt dat MessagePack na de eerste keer tot wel 3.4 keer sneller deserialiseert dan Utf8Json. Daarbij is MessagePack ook consistent heel snel wat te zien is aan de standaarddeviatie.

9.4 Conclusie

Na het uitvoeren van zowel integratie tests, unit tests als performance tests is het prototype op drie verschillende manieren getest. Door middel van de integratie en unit tests heb ik de vooropgestelde functionele requirements getest. Hiermee is aan BR1 voldaan. In paragraaf 9.3 staan de verschillende performance testen beschreven. In deze performance testen zijn een aantal bevindingen gedaan waardoor delen van code herschreven diende te worden. Uiteindelijk voldeed het prototype aan de verwachtingen van de onderzoekshypothese en is hiermee ook aan BR2 en BR3 voldaan. Met deze resultaten heb ik een dragend advies kunnen geven welke in het volgende hoofdstuk beschreven staat.

10. Adviesrapport

Aan het einde van mijn afstudeeropdracht heb ik een adviesrapport aan Nalta geschreven. Ik ben aan dit adviesrapport begonnen nadat ik het prototype heb ontwikkeld en getest. De uitkomst van het onderzoek en de testresultaten hebben gediend als basis voor dit advies. Uit het onderzoek is een hypothese gekomen welke verwacht dat het gebruik van een modern front-end framework de performance van het klantenportaal zal verbeteren. Deze hypothese is getest door middel van een prototype in Blazor, omdat Blazor hiervoor is geadviseerd vanuit het onderzoek.

10.1 Alternatieven

Naast Blazor zijn er uit het onderzoek nog een aantal alternatieven naar voren gekomen. Deze alternatieven worden in de volgende paragrafen beschreven.

Rust
Blazor is aangeraden doordat het gebruik maakt van WebAssembly en geschreven kan worden in C#. Echter, het gebruik van Blazor is niet de enige optie om gebruikt te maken van WebAssembly. Ook Rust is hiervoor een goede kandidaat, zoals beschreven in 6.4 (R.,2019). In het adviesrapport beschrijf ik nog eens kort wat Rust is en welke voor- en nadelen het gebruik van rust met zich meebrengt.

Het grootste nadeel dat Nalta bij Rust zal ondervinden, is dat C# niet gebruikt kan worden om de applicatie te ontwikkelen en ervaring in Rust moet worden opgedaan. Het leren van een nieuwe programmeertaal kost tijd en geld.

JavaScript framework
Naast de WebAssembly techniek is het ook mogelijk om het klantenportaal om te zetten naar een single page application met behulp van een modern JavaScript framework. In het onderzoek zijn Angular en Vue.js als voorbeelden aangegeven omdat Nalta al ervaring heeft met deze technieken. Beide opties kunnen gebruikt worden om de website vanaf een static file hosting server te laten hosten.

Het grootste voordeel van een single page application die is ontwikkeld met een JavaScript framework, is dat het omzetten van JSON naar een JavaScript object geoptimaliseerd is. Hierdoor wordt de modellen ontzettend snel gedeserialiseerd. Een nadeel is echter dat de logica moet worden herschreven van C# naar JavaScript en het front-end hierdoor in een andere taal is ontwikkeld dan de back-end.

10.2 Back-end Aanpassingen

Naast het omzetten van het klantenportaal naar een single page application heb ik ook andere manieren gevonden die gebruikt kunnen worden om de performance van het klantenportaal te verbeteren. Deze methoden zullen op zichzelf echter minder effectief zijn dan de eerder beschreven oplossingen, maar ze kunnen wel gecombineerd worden met een van deze oplossingen.

In het testverslag van het prototype is er een onderzoek gedaan naar de achterliggende reden van het performance probleem voor het deserialiseren van modellen. Uit dit onderzoek bleek dat de aangeraden deserialisatie methode van Microsoft nog niet geoptimaliseerd was voor grote JSON-objecten in Blazor. Door een andere deserialisatie library te gebruiken is het probleem al verminderd echter ben ik blijven zoeken naar een manieren om het probleem nog meer in te perken.

Benodigde data
Het bleek dat het JSON-response veel meer informatie bevat dan dat het klantenportaal daadwerkelijk nodig heeft. Van de 18 kB aan data, wat verstuurd wordt door de API, wordt er slechts 3 kB gebruikt. Het deserialiseren van enkel de benodigde data duurt 1,36 ms in Blazor. Dit is nog eens meer dan zeven keer sneller dan de situatie in Blazor met de Utf8Json library. Het nadeel is echter dat er aanpassingen aan de back-end moeten worden doorgevoerd om dit te bewerkstelligen, wat ingaat tegen de vooropgestelde technische beperkingen. Om deze reden is het niet als officiële oplossingen aangegeven.

Binair
Een andere oplossing die mogelijk gebruikt kan worden, is door het request vanuit de API in een binair formaat terug te sturen. JSON is geoptimaliseerd voor JavaScript terwijl C# een binair formaat makkelijker kan omzetten. In paragraaf 9.3 staat meer informatie over het onderzoek waarin deze manier wordt onderzocht. Hieruit blijkt dat er inderdaad manieren zijn om data in een binair formaat op te sturen waardoor het sneller wordt gedeserialiseerd dan JSON-objecten. Echter, net als bovenstaande methode moeten er aanpassingen aan de backend worden doorgevoerd om dit te bewerkstelligen. Hierom is dit niet verder onderzocht en niet als officiële oplossingen aangegeven.

10.3 Aanbeveling

Uit het onderzoek is een hypothese gekomen welke vervolgens is getest aan de hand van een Prototype. De gekozen techniek om het prototype te bouwen is Blazor, want Blazor maakt gebruik van het ontzettend snelle WebAssembly en er kan ontwikkeld worden in C#. Het nadeel van Blazor is echter dat het op moment van schrijven nog in preview is. Ondanks dat ik met preview software heb gewerkt tijdens het ontwikkelen van het prototype is de verwachten dat er weinig aanpassen aan de code worden doorgevoerd voordat de officiële release plaatsvindt. Dit is te verklaren doordat het prototype is ontwikkeld met behulp van de release candidate versie van Blazor. Release candidate software is veelal klaar voor de officiële release maar wordt nog gecontroleerd op bugs (Beal, z.d.).

Op basis van de onderzoeksresultaten en een zorgvuldige afweging van de voor- en nadelen van de beschreven alternatieven is het advies om het klantenportaal in Blazor te ontwikkelen. Echter, wanneer Nalta vindt dat de voordelen van Blazor niet opwegen tegen het werken met preview software is, wordt geadviseerd om het klantenportaal te herschrijven in Vue.js of Angular. Dit advies wordt gegeven gezien Nalta niet van plan is om aan de slag te gaan met Rust en er met behulp van Vue.js en Angular, waar Nalta al eerder ervaring mee heeft opgedaan, ook een single page application ontwikkeld kan worden die gehost kan worden op een static file hosting server.

Naast de techniek voor het klantenportaal raad ik Nalta ook aan om te kijken naar de mogelijkheden van de gegeven back-end aanpassingen. Deze aanpassingen kunnen de performance van het klantenportaal nog verder kunnen verbeteren.

11. Resultaten

Een uitgevoerde opdracht levert over het algemeen ook resultaten op. De resultaten van mijn onderzoek staan in dit hoofdstuk beschreven. De resultaten bevatten geen beschrijvingen van de opgeleverde producten, deze is te vinden in de hoofdstukken 5, 6 en 7.

11.1 Onderzoekshypothese

Tijdens de afstudeeropdracht is er een onderzoek uitgevoerd naar mogelijke oplossingen om de performance van het klantenportaal van PostNL te verbeteren. Uit dit onderzoek is gebleken dat het klantenportaal last heeft van een hoge latency. Een geschikte oplossing hiervoor is om een CDN voor de website te plaatsen. Als PostNL een CDN wil gaan gebruiken moet de website echter gehost kunnen worden op een static file hosting server. Nadat verschillende de manieren zijn onderzocht om het hosten op een static file hosting server mogelijk te maken blijkt het gebruik van een modern front-end framework een passende oplossing te zijn.

Het onderzoek leverde een hypothese op welke vervolgens gecontroleerd dient te worden aan de hand van een prototype. In het geval dat het testen van het prototype voldoet aan de verwachtingen wordt de hypothese niet verworpen. De hypothese die uit het onderzoek naar voren is gekomen luidt als volgt: “Door middel van een modern front-end framework, in combinatie met een CDN, wordt de performance van het klantenportaal van PostNL verbeterd.”

11.2 Test resultaten

Na het onderzoek is het prototype ontwikkeld en getest. Deze testresultaten zijn beschreven in paragraaf 9.3 en staan verder uitgewerkt in het testverslag. Tijdens het testen van het prototype heb ik nog een aantal kinderziektes gevonden in Blazor. Dit is te verklaren doordat Blazor momenteel nog in preview is. Om de performance optimaal te houden zijn er manieren onderzocht om hier omheen te werken. Uiteindelijk voldoet het prototype met Blazor, in combinatie met een CDN, aan de verwachten en werkt het sneller dan het originele klantenportaal. Hierdoor kan gesteld worden dat de hypothese niet is verworpen. De laadsnelheden zoals aangegeven in paragraaf 7.3 zijn nogmaals weergeven in tabel 11.1.

Regio	.NET 4.8	.NET Core	Blazor	Blazor + CDN
Europa, Duitsland, Frankfurt	595 ms (σ 93)	540 ms (σ 97)	510 ms (σ 92)	**158 ms** (σ 29)
Europa, Engeland, Londen	607 ms (σ 97)	565 ms (σ 101)	540 ms (σ 98)	174 ms (σ 35)
Noord-Amerika, USA, San Francisco	3021 ms (σ 314)	2766 ms (σ 267)	2654 ms (σ 97)	218 ms (σ 30)
Azië, Japan, Tokio	5196 ms (σ 480)	4286 ms (σ 445)	3676 ms (σ 314)	264 ms (σ 50)

Oceanië, Australië, Sydney	**5974 ms** (σ 557)	5214 ms (σ 521)	3918 ms (σ 354)	294 ms (σ 57)

Tabel 11.1 - Nieuwe laadsnelheid

11.3 Advies

Nadat het onderzoek is uitgevoerd en het prototype is ontwikkeld en getest heb ik een gegrond advies kunnen geven aan Nalta. Uit het onderzoek is een hypothese gekomen welke vervolgens is getest aan de hand van een Prototype. De gekozen techniek om het prototype te bouwen is Blazor, want Blazor maakt gebruik van het ontzettend snelle WebAssembly en er kan ontwikkeld worden in C#. Het nadeel van Blazor is echter dat het op moment van schrijven nog in preview is. Ondanks dat ik met preview software heb gewerkt tijdens het ontwikkelen van het prototype, is de verwachting dat er weinig aanpassingen aan de code doorgevoerd hoeft te worden wanneer de officiële release plaatsvindt. Dit staat beschreven in paragraaf 10.3.

Op basis van de onderzoeksresultaten en een zorgvuldige afweging van de voor- en nadelen van de beschreven alternatieven is het advies om het klantenportaal naar Blazor om te schrijven. Echter, wanneer Nalta vindt dat de voordelen van Blazor minder waard zijn dan het werken met preview software is, dan wordt geadviseerd om het klantenportaal te schrijven in Vue.js of Angular. Dit advies wordt gegeven gezien Nalta niet van plan is om aan de slag te gaan met Rust en er met behulp van Vue.js en Angular, waar Nalta al eerder ervaring mee heeft opgedaan, ook een single page application ontwikkeld kan worden die gehost kan worden op een static file hosting server.

Naast de techniek voor het klantenportaal raad ik Nalta ook aan om te kijken naar de mogelijkheden van de gegeven back-end aanpassingen. Deze aanpassingen kunnen de performance van het klantenportaal nog verder kunnen verbeteren.

12. Evaluatie en Reflectie

In dit hoofdstuk geef ik een evaluatie op de afgelopen afstudeerperiode. Als eerste beschrijf ik de productevaluatie waarin ik de verschillende tussenproducten beschrijf en evalueer. Vervolgens kijk ik terug op het process en hoe ik dit heb ervaren. Als laatste ga ik in op de vooropgestelde beroepstaken en reflecteer ik op de aanpak tijdens deze beroepstaken en wat ik hiervan heb geleerd.

12.1 Productevaluatie

In deze paragraaf beschrijf en evalueer ik de verschillende opgeleverde tussenproducten. Als eerste staat het adviesrapport beschreven, omdat met dit product voldaan wordt aan de opdrachtomschrijving.

Adviesrapport
Voorafgaand aan de afstudeeropdracht is de opdracht opgesteld om een advies te leveren aan Nalta over hoe zij de performance van het klantenportaal van PostNL kunnen verbeteren. Aan dit advies zijn van tevoren geen harde eisen gesteld waardoor ik zelf heb mogen bepalen hoe ik tot dit advies ben gekomen. Ik heb ervoor gekozen om een tal verschillende tussenproducten op te leveren, die samen hebben geleid tot een gegrond advies.

Ik ben erg tevreden over de manier waarop ik het advies aan Nalta heb geschreven. Het advies is een gegrond advies doordat de onderzoekshypothese is getest door middel van een prototype. Tevens lever ik in het adviesrapport een extra advies omdat ik er na het doen van onderzoek achter ben gekomen dat kleine aanpassingen aan de back-end mogelijk nog meer performance optimalisatie kunnen opleveren. Ik vind dat ik hiermee een uitgebreider advies heb kunnen geven dan dat ik van tevoren had verwacht.

Tussenproduct: Requirementsrapport
Het requirementsrapport is gebruikt om de verschillende requirements en technische beperkingen te beschrijven. Eerder in de opleiding tot Software Engineer heb ik al eens een requirementsrapport opgeleverd echter was de techniek hiervan deels verwaterd. Na het lezen van de literatuur kwam deze kennis al snel weer boven drijven en heb ik op de juiste manier de requirements kunnen werven. Ik denk dat ik de correcte requirements heb verzameld omdat ze ertoe hebben geleid dat ik een prototype heb kunnen opleveren waar mijn begeleider tevreden mee was. Daarbij heb ik complimenten gekregen dat de manier waarop ik vragen heb gesteld tijdens het werven van requirements erg fijn is overgekomen.

Tussenproduct: Onderzoek
Het onderzoek heeft als basis voor het prototype en advies gediend. Tijdens het onderzoek ben ik op kritische wijze, zoals aangeleerd tijdens blok 8 van de opleiding Software Engineering, op zoek gegaan naar bronnen die mij konden helpen bij het vinden van een oplossing. Gedurende het onderzoek vond ik steeds meer verschillende oplossingen waarbij er ook weer een aantal afvielen vanwege technische beperkingen. Uiteindelijk heeft het voor mij geleid tot een gerichte oplossing die op een aantal verschillende manieren kan worden uitgewerkt. Ik heb de onderzoekshypothese breed genoeg opgesteld dat alle gevonden

oplossingen uitgewerkt zouden kunnen worden. Echter, om met specifiek één techniek aan de slag te kunnen gaan tijdens het ontwikkelen van het prototype doe ik in het onderzoek ook nog een aanbeveling welke mij het meest gepast lijkt.

Ik ben erg tevreden over de manier waarop ik het onderzoek heb aangepakt. Ik ben van meening dat ik veel heb geleerd qua zowel kennis als kritisch onderzoeken. Wat ik in het vervolg anders zou doen is dat ik eerst een aantal proof-of-concepts zou opzetten om snel aan te tonen of de gevonden techniek wel echt toepasbaar is.

Tussenproduct: Ontwerp
Het ontwerp is gemaakt omdat er aan de hand van een ontwerp gerichter gewerkt kan worden aan een prototype. Tijdens het maken van het ontwerp heb ik verschillende diagrammen gemaakt om het volledige ontwerp zou duidelijk mogelijk in beeld te krijgen.

In het vervolg zou ik meer tijd willen uittrekken voor het ontwerpen van software. Het is een belangrijke taak en ik ben er achter gekomen dat het naderhand aanpassen veel tijd kost. Door van tevoren beter na te denken over hoe de uiteindelijke applicatie er uit dient te zien en hoe deze zal werken kunnen mogelijke aanpassingen worden voorkomen.

Tussenproduct: Prototype
Het prototype is ontwikkeld om te controleren of de gevonden onderzoekshypothese ook daadwerkelijk juist is voor het klantenportaal. Dit betekende dat ik daadwerkelijk aan de slag kon gaan met programmeren. Het programmeren heb ik als zeer prettig ervaren doordat ik het prototype heb ontwikkeld met een nieuwe techniek. Zelf vind ik het altijd prettig om nieuwe dingen te leren en dit heb ik zeker gedaan tijdens deze afstudeeropdracht.

De volgende keer wil ik het ontwikkelen van het prototype wel anders aanpakken. Het volledige product is nu ontwikkeld door mij zonder dat dit door andere personen is gecontroleerd. In de toekomst zou ik de geschreven code laten peer-reviewen door collega’s om een hogere kwaliteit van de sourcecode te kunnen garanderen voor het bedrijf.

Tussenproduct: Testverslag
Een ander tussenproduct beslaat het testverslag. In het testverslag staan de verschillende testen beschreven die zijn gebruikt om het prototype te testen. De testen verifiëren of de geschreven software ook daadwerkelijk voldoet aan de verwachtingen welke staan beschreven in hoofdstuk 9. Het testen van de software heeft ervoor gezorgd dat ik een hoop nieuwe kennis heb opgedaan. Zo had ik zelf nog maar weinig ervaring met het uitvoeren van performance testen maar heeft de literatuur mij uitgewezen hoe ik deze testen het beste kan uitvoeren.

Ik vind dat ik op de juiste manier heb getest tijdens mijn afstudeeropdracht. Ik heb verschillende vormen van testen gebruikt om te controleren of het prototype voldoet aan de verwachtingen van het onderzoek. Daarbij heeft het mij voorbereid op de toekomst hoe ik software in de praktijk op verschillende manieren kan testen.

12.2 Procesevaluatie

De fasering is achteraf gezien grotendeels aangehouden. Het vooronderzoek was iets eerder voltooid dan verwacht wat misschien kwam doordat het door mij wat te pessimistisch was ingeschat. Hierdoor ben ik eerder aan het onderzoek begonnen. Eenmaal bij de ontwikkelfase aangekomen liep ik weer gelijk met de planning doordat het onderzoek iets meer uitloop had. Tijdens de ontwikkelfase heb ik gewerkt met de Kanban methode. Het gebruik van Kanban vond ik erg prettig tijdens de afstudeeropdracht. Het gaf mij een duidelijk overzicht in het werk wat gedaan was en het werk wat gedaan moest worden. Zelf heb ik vaak de neiging om aan meerdere taken tegelijkertijd te werken. Hierdoor lever ik soms half werkende software op doordat ik ben vergeten om het af te ronden. Wanneer de Kanban methode gebruikt wordt mag er echter maar één taak in voortgang staan. Zoals beschreven in paragraaf 4.2 is dit ook de voorwaarde dat ik de Kanban methode heb kunnen gebruiken. Dit principe heeft mij goed geholpen om mij op deze ene taak te focussen. Deze aanpak heeft mij zeker geholpen om kwalitatief goede code op te leveren waarmee de onderzoekshypothese gecontroleerd kon worden.

Achteraf gezien ben ik redelijk tevreden met de aanpak van mijn afstudeeropdracht. Ik heb een redelijk goede planning kunnen maken en heb mij hier zo veel mogelijk aan gehouden. Daarbij heeft de Kanban methode mij geholpen tijdens de ontwikkelfase. In het vervolg zou ik de planning hetzelfde houden maar zou ik liever de Scrum methodiek toepassen omdat ik het persoonlijk fijner vind om in een team te werken.

12.3 Verantwoording beroepstaken

In deze paragraaf behandel ik alle beroepstaken die ik van tevoren heb opgesteld. Dit zijn de beroepstaken in mijn stageplan en worden nogmaals extra benoemd in Bijlage A (Beroepstaken). Per beroepstaak voer ik een evaluatie en reflectie uit.

A 01 - Analyseren probleemdomein & opstellen probleemstelling
Tijdens elke afstudeeropdracht wordt deze beroepstaak behandeld.
Het is een verplichte beroepstaak, daar het analyseren van het probleemdomein en het opstellen van de probleemstelling te allen tijde ter sprake komt. Mijn afstudeeropdracht was van tevoren geformuleerd, waardoor een groot gedeelte van het probleem hieruit opgehaald kon worden. Hierbij heb ik zelf onderzoek gedaan naar de situatie waarin dit probleem zich voordoet. Beide bronnen zijn vervolgens gecombineerd om gericht te kunnen starten met de opdracht.

Ik vind dat ik deze beroepstaak goed heb uitgevoerd. Zoals hierboven beschreven heb ik meerdere bronnen onderzocht om te verifiëren wat het probleem domein is en welke richting ik het moest zoeken. Een verbeterpunt voor de toekomst is om een vast stappenplan te maken die hergebruikt kan worden om achter het probleemdomein en de probleemstelling te komen. Op deze manier kan ik nog doelgerichter werken.

B 04 - Gemotiveerd selecteren van ICT gerelateerde oplossingen
Tijdens het onderzoek ben ik opzoek gegaan naar verschillende manieren om de performance van het klantenportaal te verbeteren. Het was zaak dat ik alle verschillende manieren met elkaar vergeleek gezien ik nog geen kennis had over het optimaliseren van webapplicaties. Deze vergelijkingen staan beschreven in hoofdstuk 6. In het onderzoek benoem ik per oplossing de voor- en nadelen en beschrijf ik vervolgens of de oplossing toegepast kan worden op het klantenportaal. Door dit proces te blijven volgen ben ik uiteindelijk tot een oplossing gekomen waarvan ik verwacht welke de meeste optimalisatie gaat opleveren.

Ik heb deze beroepstaak op de juiste manier uitgevoerd tijdens mijn afstudeeropdracht. Alle mogelijke oplossingen zijn aandachtig onderzocht. Wanneer ze niet toegepast konden worden zijn ze niet verder uitgewerkt. Een aandachtspunt voor mijn vervolg carrière is om eerder te herkennen wanneer een oplossing niet toepasbaar is voor het probleem, zodat er nog gerichter onderzoek gedaan kan worden naar de andere oplossingen.

B 05 - Adviseren over inrichting van ICT gerelateerde oplossingen en processen
De kern van mijn afstudeeropdracht is het leveren van een advies aan Nalta over hoe zij de performance van het klantenportaal van PostNL kunnen verbeteren. Dit advies staat beschreven in hoofdstuk 10. Om aan deze beroepstaak te voldoen dient het advies ook onderbouwd en juist te zijn. Om deze redenen heb ik aan het begin van de afstudeeropdracht een aantal tussenproducten gekozen die samen hebben geleid tot een onderbouwd en juist advies. Zoals in B 04 beschreven is het onderzoek gebruikt om te onderzoeken welke oplossingen er allemaal toegepast kunnen worden. De ontwikkelfase is vervolgens gebruikt om te controleren of de gevonden oplossing ook effect zou hebben op het klantenportaal. Aan de hand van de resultaten van de tussenproducten heb ik een gegrond advies kunnen geven.

Deze beroepstaak is op de best mogelijke manier uitgevoerd door een advies te schrijven aan Nalta en het ontwikkelteam van het klantenportaal welke is gebaseerd op feiten. Ik vind dat ik dit onderwerp op de juiste manier heb uitgevoerd tijdens mijn afstudeeropdracht doordat er uiteindelijk een gegrond advies uit is het gekomen. In mijn vervolg carrière ben ik dan ook van plan om dit op dezelfde manier aan te pakken.

D 14 - Realiseren van software
Deze beroepstaak heeft betrekking op het ontwikkelen van het prototype welke staat beschreven in hoofdstuk 8. De beroepstaak is door mij op niveau 4 uitgevoerd. Het realiseren van software is op niveau 4 uitgevoerd doordat ik meerdere taken heb uitgevoerd die hebben geleid tot kwalitatieve software. Tijdens het realiseren van de software ben ik aan de gang gegaan met een nieuwe techniek. Deze techniek heb ik eerst onderzocht door middel van de documentatie om een beeld te krijgen van de mogelijkheden die deze techniek met zich meebrengt. Vervolgens is de software modulair opgebouwd zodat bepaalde stukken makkelijk hergebruikt konden worden. Als laatste is de logica geëxtraheerd om het testen van de logica makkelijker te maken.

Ik heb deze beroepstaak op de juiste manier uitgevoerd tijdens mijn stage. Er heeft zich geen moment voorgedaan waarbij ik het vermoeden had dat ik geen correct prototype zal opleveren aan Nalta. In de toekomst zal ik mijn eigen taken op dezelfde manier aanpakken.

Echter zou ik dit liever in een team uitvoeren zodat de geschreven code ook ge-peer-reviewed kan worden door collega's. Op deze manier kan ik een nog hogere kwaliteit van software garanderen.

D 15 - Testen
Deze beroepstaak houdt in dat de geschreven software ook op de juiste manier getest wordt. De testen die verifiëren of geschreven software ook daadwerkelijk voldoet aan de verwachtingen staan beschreven in hoofdstuk 9. In hoofdstuk 9 staat dat ik drie verschillende vormen van testen heb uitgevoerd: integratie tests, unit tests en performance tests. Door middel van een combinatie van deze resultaten is de onderzoekshypothese getest waarbij deze niet is verworpen.

Deze beroepstaak heb ik goed uitgevoerd. Ik heb verschillende vormen van testen gebruikt om de onderzoekshypothese te testen. Daarbij heb ik voldaan aan het requirement dat het prototype automatisch getest kan worden. In het vervolg is het misschien praktischer om een testplan op papier te zetten. Op deze manier kan er nog gerichter getest worden en zullen alle verschillende scenario's die getest dienen te worden van tevoren bekend zijn.

G c - Kritisch, onderzoekend & methodisch werken
Vooral tijdens het schrijven van mijn onderzoek, maar ook tijdens de ontwikkelfase, heb ik veel te maken gehad met deze beroepstaak. Dit was nodig om er zeker van te zijn dat alle informatie die ik heb gevonden ook daadwerkelijk juist was. Zo ben ik tijdens het uitvoeren van het onderzoek kritisch op zoek gegaan naar de oorsprong van de bronnen. Hierbij gebruikte ik zowel fysieke- als onlinebronnen om een uitgebreider onderzoek te doen. Het methodisch werken heeft er onder andere voor gezorgd dat de verschillende documenten die zijn geschreven tijdens de afstudeeropdracht met de juiste structuur zijn opgezet.

Voor mijn vervolg carrière is het voor mij desalniettemin verstandig om nog grondiger onderzoek te doen naar de wijze van het opzetten van de verschillende documenten. Tijdens de feedback momenten heb ik aanwijzingen gekregen om de structuur van mijn verslag aan te passen. Deze aanwijzingen heb ik dan ook met beide handen aangepakt om hier zo veel mogelijk van te leren.

G f - Leren leren: voorbereiden op volgende studiefase en beroep
De laatste beroepstaak is leren leren. Dit houdt in dat je jezelf wilt blijven doorontwikkelen tijdens het uitvoeren van je afstudeeropdracht. Tijdens elk tussenproduct ben ik hier wel mee in aanmerking gekomen. Zo heb ik mijzelf aangeleerd hoe je op een zo duidelijk mogelijke manier open vragen kan stellen tijdens het werven van requirements om zoveel mogelijk informatie in te winnen. Ook in het onderzoek heb ik te maken gehad met deze beroepstaak. Door de verschillende nieuwe methoden en technieken, die gevonden zijn in het onderzoek, is het makkelijk om verdwaald te raken. Gezien je over het onderwerp meer te weten wil komen. Door het methodisch werken van beroepstaak G c bleef ik gericht werken en deed ik alsmaar nieuwe kennis op. Ook het testverslag is een goed voorbeeld voor deze beroepstaak. Mijn afstudeeropdracht is met name gericht op het front-end van het klantenportaal, maar toch heb ik mij ook op specifieke delen van de back-end gericht waarbij ik het vermoeden had dat er een grote optimalisatie te halen was met minimale wijzigingen.

Ik heb deze beroepstaak goed opgepakt. Ik bleef mijzelf uitdagen tijdens mijn afstuderen door niet altijd de makkelijkste weg te kiezen. Hierdoor heb ik niet alleen veel nieuwe kennis opgedaan, maar ben ik er ook achter gekomen dat er nog zoveel is waar ik nog geen kennis van heb. Om deze reden kijk ik erg uit naar het werk als Software Engineer om elke dag weer uitgedaagd te worden door nieuwe problemen. Een belangrijk punt dat ik uit deze

afstudeeropdracht heb gehaald is dat ik het prachtig vind om nieuwe kennis op te doen. Om dit te blijven doen wil ik mijzelf aanleren om elk kwartaal een lijstje te maken met onderwerpen of leerpunten waarin ik wil verdiepen.

Literatuurlijst

Ahmad, M. O., Markkula, J., & Oivo, M. (2013).
Kanban in software development: A systematic literature review (Vol. 39). Santander, Spain: IEEE.

Amazon. (z.d.). *Static file hosting* [Foto].
Geraadpleegd van https://d1.awsstatic.com/Projects/v1/AWS_StaticWebsiteHosting_ArchitEcture_4b.da7f28eb4f76da574c98a8b2898af8f5d3150e48.png

Arabi, K. (2014, 4 juni).
IEEE DAC 2014 Keynote: Mobile Computing Opportunities, Challenges and Technology Drivers
http://www2.dac.com/events/videoarchive.aspx?confid=170&filter=keynote&id=170-103--0&#video

Asthana, A. (2019, 10 mei).
Introducing .NET 5. Geraadpleegd op 5 maart 2020, van https://devblogs.microsoft.com/dotnet/introducing-net-5/

Balaji, S., & Sundararajan Murugaiyan, M. (2012).
WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC (1).
Geraadpleegd van https://pdfs.semanticscholar.org/8d2a/ad803781a34a43af447cea410cd710b9f1f5.pdf

Beal, V. (z.d.).
Release candidate. Geraadpleegd op 4 mei 2020, van https://www.webopedia.com/TERM/R/release_candidate.html

Bland, J. M., & Altman, D. G. (1996).
Measurement error. BMJ (Clinical research ed.), 312(7047), 1654.
https://doi.org/10.1136/bmj.312.7047.1654

Boks, A. M. (2015, 14 augustus).
Projectsucces | Projectsucces - eisen en randvoorwaarden aan projecten.
Geraadpleegd op 26 februari 2020, van https://www.projectsucces.nl/nieuws/projectsucces-eisen-en-randvoorwaarden-aan-projecten/

Burge, S. (2014, 31 januari).
Why and How We Use Pingdom for Uptime Monitoring. Geraadpleegd op 5 maart 2020, van https://www.ostraining.com/blog/tools/pingdom/

Chapel, J. (2020, 11 februari).
AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows.
Geraadpleegd op 12 mei 2020, van https://medium.com/@jaychapel/aws-vs-azure-vs-google-cloud-market-share-2020-what-the-latest-data-shows-9afd4accf8d7

de Swart, N. (2010).
Handboek Requirements. Amsterdam, Nederland: Reed Business Education.

Driessen, K. (2018, 29 januari).
Structuur plan van aanpak (PvA) voor het hbo. Geraadpleegd op 19 september 2018, van https://www.scribbr.nl/scriptie-structuur/structuur-plan-van-aanpak-pva-voor-het-hbo/

ISO25000. (z.d.).
InfoCamere obtains the ISO/IEC Data Quality certificate for their database Registro Imprese. Geraadpleegd op 5 maart 2020, van https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Filippo, C. (2012, 5 september).
Scriptieschrijven in fasen…stap voor stap kom je er wel! | Studietips Universiteit Leiden. Geraadpleegd op 27 februari 2020, van https://studietips.weblog.leidenuniv.nl/2012/09/05/scriptieschrijven-in-fasen-stap-voor-sta/
Fowler, M., Scott, K., Kobryn, C., & Safari Tech Books Online. (2004).
UML Distilled (3de editie, Vol. 2004). New Jersey, United States: Addison-Wesley.
Google. (2019, 28 mei).
Developer Guide | Protocol Buffers | . Geraadpleegd op 15 mei 2020, van https://developers.google.com/protocol-buffers/docs/overview
Gucinsky, E. (z.d.).
Wat is functioneel testen? welke soorten zijn er en hoe pak ik dit aan? Geraadpleegd op 15 mei 2020, van https://www.technosoft.nl/qa-testing/functioneel-testen
Krul, A. (2017, 20 april).
Zo doe je een literatuuronderzoek of literatuurstudie. Geraadpleegd op 12 maart 2020, van https://www.scribbr.nl/scriptie-structuur/hoe-doe-je-literatuuronderzoek/
Microsoft. (2020a, 25 maart).
Introduction to ASP.NET Core Blazor. Geraadpleegd op 23 april 2020, van . https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-3.1
Microsoft. (2020b, 26 maart).
Server-to-server storage replication. Geraadpleegd op 13 mei 2020, van https://docs.microsoft.com/en-us/windows-server/storage/storage-replica/server-to-server-storage-replication
Mittelmeijer, M., & van Stratum, R. (2014). *Kijk op bedrijfsprocessen* (3de editie). Groningen, Nederland: Noordhoff.
Nalta, 2020
Nalta.com: Your Partner in Platform Perfection. (z.d.). Geraadpleegd op 12 februari 2020, van https://www.nalta.com/
Oloruntoba, S. (2015, 15 maart).
S.O.L.I.D: The First 5 Principles of Object Oriented Design. Geraadpleegd op 23 april 2020, van https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
Peng, G. (2018).
CDN: Content Distribution Network. Geraadpleegd van https://arxiv.org/pdf/cs/0411069.pdf
Plug Things In. (z.d.).
What is Latency - How is Latency Different from Bandwidth.
Geraadpleegd op 21 februari 2020, van http://www.plugthingsin.com/internet/speed/latency/
R. (2019, 5 april).
How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study).
Geraadpleegd op 15 mei 2020, van https://www.smashingmagazine.com/2019/04/webassembly-speed-web-app/
Rising, L., & Janoff, N. S. (2000).
The Scrum software development process for small teams (4de editie, Vol. 17).
Santander, Spain: IEEE. https://doi.org/10.1109/52.854065
Static Content Hosting pattern - Cloud Design Patterns. (2020, 24 februari).
Geraadpleegd op 3 maart 2020, van https://docs.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting

Souders, S. (2007).
High Performance Web Sites. Culemborg, Nederland: Van Duuren Media.
Swaen, B. (2020, 15 januari).
Conclusie scriptie. Geraadpleegd op 24 maart 2020, van
https://www.scribbr.nl/scriptie-structuur/conclusie-scriptie/

Bijlage

A. Beroepstaken

- A 01 Analyseren probleemdomein & opstellen probleemstelling
- B 04 Gemotiveerd selecteren van ICT gerelateerde oplossingen
- B 05 Adviseren over inrichting van ICT gerelateerde oplossingen en processen
- D 14 Realiseren van software
- D 15 Testen
- G c Kritisch, onderzoekend & methodisch werken
- G f Leren leren: voorbereiden op volgende studiefase en beroep

Bijlageopgave

Plan van Aanpak

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
18-02-2020
Versie 3.0

Inhoudsopgave

1. Inleiding

Dit is het plan van aanpak van mijn afstudeeropdracht bij Nalta. Het onderwerp van mijn afstudeeropdracht is advies leveren voor performance optimalisatie van het klantenportaal van PostNL bij Nalta. PostNL vindt namelijk dat het klantenportaal, welke gebruikt wordt door de zakelijke klanten van PostNL, een slechte performance heeft. Tijdens het afstuderen lever ik hiervoor verschillende producten op.

Allereerst voer ik een onderzoek uit naar de methoden die gebruikt kunnen worden om de performance van het klantenportaal te verbeteren. Aan de hand van dit resultaat probeer ik de effectiviteit van de meest geschikte oplossing aan te tonen door het ontwikkelen van een prototype. Op het moment zijn de methoden en technieken die hiervoor gebruikt gaan worden nog niet bekend. Tijdens het onderzoek zal hier meer duidelijkheid over komen.

2. Aanleiding en context

De opdrachtgever van mijn afstudeeropdracht is Nalta. Nalta is een internet bedrijf. Zij ontwerpt en ontwikkelt responsive websites, online platformen en apps. Een aantal voorbeeld applicaties die Nalta heeft ontwikkeld zijn Digital Angel, Datumprikker en het bedrijven klantenportaal van PostNL. Nalta heeft 40 werknemers verdeeld over 2 filialen. Deze zijn gevestigd in Almere en Wateringen.

De opdracht waar ik mij op focus tijdens het afstuderen is performance optimalisatie voor het klantportaal van PostNL (hier zal verder naar gerefereerd worden als het klantportaal). Een aantal bedrijven klaagt namelijk over de snelheid van het portaal. Dit geldt voornamelijk voor bedrijven die vanaf buiten Europa het portaal benaderen. Deze gebruikers vinden dat het te lang duurt voordat de schermen getoond worden. PostNL heeft Nalta eerder gevraagd dit op te lossen. Onder andere hierom heeft Nalta de front-end van het klantenportaal omgezet van .NET Framework naar .NET Core. Helaas gaf dit nog niet de verwachte en gehoopte verbetering. PostNL wil graag dat Nalta uitzoekt welke mogelijkheden er nog meer gebruikt kunnen worden om de performance te verbeteren.

2.1 Opdrachtgever

De opdrachtgever van mijn afstudeeropdracht is de heer Geert Merkelbach. De functie van mijn opdrachtgever is CEO en mede-oprichter van Nalta. De heer Martijn Beenes is mijn bedrijfsmentor binnen Nalta. Hij bekleedt de functie van developer lead.

3. Probleemanalyse en probleemstelling

Het klantportaal van PostNL is momenteel aan het groeien in het aantal gebruikers. Naast Europa wordt de applicatie nu ook veelvuldig gebruikt buiten Europa. Deze gebruikers vinden dat het te lang duurt voordat de schermen van het klantenportaal getoond worden. Er bestaan vele verschillende manieren om dit te verbeteren, maar welke de meest geschikte oplossing voor PostNL blijkt te zijn moet nog worden onderzocht. Daarbij wil Nalta aan de hand van een prototype zien dat de gevonden oplossing ook daadwerkelijk effectief is.

4. Doelstelling en eindresultaat

De primaire doelstelling van mijn afstudeeropdracht is aantonen dat ik aan de hand van vakliteratuur zelfstandig een project kan uitvoeren. Dit demonstreer ik aan de hand van 7 majorcompetenties. Hiervan zijn er drie verplichte competenties en vier andere gekozen. Deze competenties staan beter beschreven in het al eerder opgeleverde afstudeerplan. Deze majorcompetenties heb ik verwerkt in de afstudeeropdracht. De producten die hieruit volgen staan verder beschreven in hoofdstuk 5 ‘Uitvoering’ en hoofdstuk 7 ‘Planning’. Door aan te tonen dat ik deze competenties vaardig ben laat ik zien dat ik competent ben binnen het Software Engineering vakgebied.

Om de competenties aan te tonen voer ik die opdracht uit bij Nalta. Gedurende deze opdracht lever ik een advies over hoe Nalta voor PostNL de performance van het klantenportaal kan optimaliseren. Zoals beschreven in de probleemanalyse moet hier eerst onderzoek naar worden gedaan. Dit zal dan ook een aanzienlijk deel van mijn afstudeeropdracht worden. Daarnaast is gewenst dat de meest geschikte oplossing uitgewerkt wordt door middel van een werkend prototype om de onderzoekshypothese aan te tonen.

Als eindresultaat van de opdracht die ik uitvoer bij Nalta lever ik in eerste instantie een onderzoek op waarin ik opzoek ga naar verschillende oplossingen om de performance van het klantportaal te verbeteren. Daarnaast lever ik een werkend prototype op.

5. Uitvoering

Zoals beschreven in hoofdstuk 4 ‘Doelstelling en eindresultaat’ laat ik mijn competenties zien aan de hand van de eerder beschreven majorcompetenties welke zijn verwerkt in de producten. Ik begin met mijzelf te oriënteren binnen het bedrijf. In de eerste weken ga ik verschillende gesprekken voeren met interne stakeholders. Zodoende kom ik meer de weten over het bedrijf, wat vereisten zijn aan geproduceerde software en hoe het klantenportaal in elkaar steekt. Gedurende deze oriëntatiefase lever ik de volgende producten op.

- Plan van Aanpak
- Document Huidige Situatie
- Requirements Rapport

Na de oriëntatiefase begin ik aan de onderzoeksfase. Tijdens deze onderzoeksfase voer ik het onderzoek uit naar de verschillende methoden en technieken die gebruikt kunnen worden voor het optimaliseren van de performance. Dit onderzoek leidt tot een onderzoekshypothese welke vervolgens getest kan worden. Gedurende de onderzoeksfase lever ik het volgende product op:

- Onderzoeksrapport

Als de onderzoeksfase is voltooid gaat de ontwikkelfase van start. Tijdens deze ontwikkelfase maak ik een prototype aan de hand van de meest geschikte oplossing die is gekomen uit het onderzoeksrapport. Daarbij creëer ik in deze fase ook een testverslag waarin ik de effectiviteit van het prototype laat zien. Gedurende de ontwikkelfase lever ik de volgende producten op:

- Ontwerp
- Prototype
- Testverslag

Na deze verschillende fases sluit ik mijn afstudeeropdracht af door het schrijven van een adviesrapport. Tijdens deze gehele afstudeerperiode beschrijf ik in mijn afstudeerdossier alle genomen stappen en onderbouw ik ze met argumenten.

Tijdens mijn afstudeeropdracht maak ik gebruik van de huidige ontwikkelomgeving binnen Nalta. De documentatie wordt allemaal gemaakt in de applicaties van Google (Google Docs) en wordt bewaard op Google Drive. Hierdoor staat alle documentatie in de cloud en raak ik bij een eventuele crash geen documentatie kwijt.

De code van het klantportaal van PostNL staat opgeslagen in Bitbucket. Ook BitBucket is een cloud oplossing waardoor ik bij een mogelijke crash alleen het laatste stukje geschreven code verloren gaat. Tijdens het maken van het prototype creëer ik hiervoor een extra repository aan binnen BitBucket. Gedurende mijn opdracht laat ik de code door mijn afstudeerbegeleider controleren door middel van pull requests. Dit is de standaard manier van werken binnen Nalta en hier sluit ik mij bij aan.

6. Randvoorwaarden en Risico’s

In dit hoofdstuk ga ik allereerst in op de randvoorwaarden die aan de opdracht zitten. Vervolgens beschrijf ik de risico’s die mij kunnen hinderen tijdens het werken aan de afstudeeropdracht.

6.1 Randvoorwaarden

Om mijn afstudeeropdracht succesvol te kunnen afronden, is er een aantal randvoorwaarden opgesteld. De volgende punten zijn nodig voor de uitvoering:

- Een werkplaats op het kantoor van Nalta.
- Een laptop met toegang tot internet.
- De opdracht moet aansluiten op het beroepsspecifieke deel van de Software Engineering major.
- Het aantonen van de majorcompetenties.
- 85 dagen de tijd om deze opdracht tot een goed einde te brengen.

6.2 Risicoanalyse

In tabel 4.2 zijn de risico's uiteen gezet die tijdens het project kunnen ontstaan. Hiervoor is er gekeken naar de eerdere algemene risico's van soortgelijke projecten. Deze risico's zijn verder gedefinieerd aan de hand van een projectrisico tabel waarin de kans x effect is aangeven als de waarde van het risico. Er is voor deze aanpak met een dergelijke tabel gekozen doordat hier in een eerder stadium van de opleiding ervaring mee is opgedaan en deze erg effectief bleek te zijn.

Nr.	Risico	Gevolg	Tegenmaatregel	Kans (1-5)	Effect (1-5)	Risico (K*E)
1	De gevonden mogelijke oplossing blijkt geen optimalisatie te leveren.	De herziene versie van het klantenportaal is langzamer dan dat het voorheen was.	Meerdere opties onderzoeken en oplossingen bieden zodat mogelijk een andere oplossing geadviseerd kan worden.	3	4	12
2	Het gekozen ontwerp van het prototype blijft onjuist te zijn.	Het prototype kan niet ontwikkeld worden zoals dat van te voren was gedacht.	Genoeg tijd inplannen om het gekozen ontwerp te controleren bij de infrastructuur architect en hem waar nodig aanpassen.	2	4	8
3	Onvolledige dataverzameling van huidige bedrijfsvoering/ processen/systemen.	Onvolledig beeld van de huidige bedrijfsvoering/processen/systemen waardoor een onjuist advies naar boven kan komen.	Goede voorbereiding voor dataverzameling en juist in kaart brengen van de bedrijfsvoering/processen/systemen	2	5	10

Tabel 4.2 - Risico's

7. Planning

In hoofdstuk 5 ‘Uitvoering’ heb ik beschreven welke producten ik in de verschillende fasen oplever. Om een duidelijk beeld te schetsen over wanneer deze producten precies opgeleverd worden heb ik onderstaande planning gemaakt.

Product	Datum van oplevering
Plan van Aanpak	12-02-2020
Document Huidige Situatie	27-02-2020
Requirements Rapport	13-03-2020
Onderzoeksrapport	03-04-2020
Prototype	24-04-2020
Testverslag	08-05-2020
Adviesrapport	22-05-2020
Afstudeerdossier	05-06-2020

Huidige situatie

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
10-02-2020
Versie 2.0

Inhoudsopgave

1. Inleiding

In dit document staat de huidige situatie van het klantenportaal van PostNL beschreven. De huidige situatie is beschreven om een beter beeld te geven aan de huidige problemen die zich in het portaal bevinden. Het klantenportaal van PostNL wordt gebruikt door de zakelijk klanten van PostNL, waar verder in het document naar gerefereerd worden als het klantenportaal. Al snel bleek uit intern overleg met het ontwikkelteam dat er niet één maar twee verschillende versies van het klantenportaal bestaan. Aan elk van deze versies is in dit verslag een apart hoofdstuk toegewijd. Naast deze verschillende versies van het klantenportaal staat er ook een API (Application Programming Interface). Deze API wordt door de verschillende klanten aangeroepen om de processen te automatiseren. Hierdoor hoeven zij zelf niet in te loggen op het klantportaal.

2. Klantenportaal .NET Framework 4.8

In dit hoofdstuk staat het klantenportaal beschreven welke gecodeerd is met behulp van .NET Framework 4.8. Deze versie wordt momenteel gebruikt op de live omgeving. Dat houdt in dat alle huidige klanten gebruik maken van deze versie. In de paragraaf '.NET Framework 4.8' wordt eerst uitgelegd wat dit precies is vervolgens wordt ook de koppeling tussen de back-end en de front-end beschreven en is er een architectuurdiagram gemaakt.

2.1 .NET Framework 4.8

Microsoft heeft het .NET Framework geschreven om applicaties te laten draaien op Windows machines. Van dit framework is versie 4.8 de laatste versie. Binnen dit framework kan er geprogrammeerd worden met een verzameling van programmeertalen. Een aantal voorbeelden hiervan is: C#, F#, VB, en Windows Powershell. Het huidige klantenportaal is ontwikkeld in C#. Ook voor C# geldt dat er verschillende versies zijn, deze lopen op van C# 1.0 tot en met C# 8.0. De laatste versie van het .NET Framework kan maximaal C# 7.3 gebruikten en dat is ook het geval bij dit project.

Echter, in het voorjaar van 2019 kwam Microsoft met het nieuws naar buiten dat .Net Framework 4.8 de laatste versie zal zijn van het .NET Framework (Asthana, 2019). In november van 2020 zal .NET 5 namelijk uitkomen welke de opvolger is van .NET Core 3.1. Hiervoor is gekozen om mogelijke verwarring van .NET Framework en .NET Core in de versie 4 range te voorkomen.

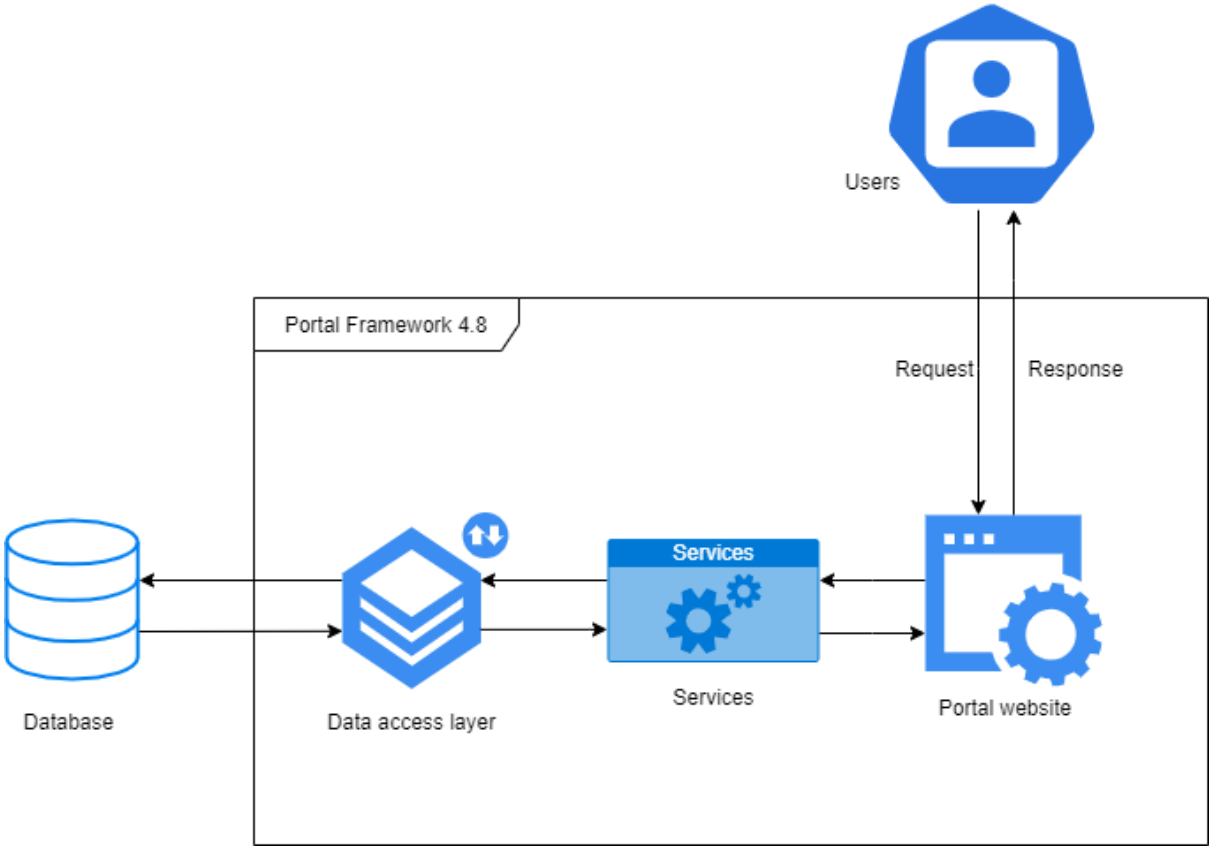
Om vooruit te blijven streven zou Nalta een belangrijke beslissing moeten maken. Blijven ze het .NET Framework gebruiken of schakelen ze over naar .NET Core?

2.2 Koppeling

De heer M. Beenes, de lead-developer binnen Nalta, liet blijken dat er nog een groot probleem was waar het development team mee kampte. De front-end van het klantenportaal is namelijk te erg verweven met de back-end van het platform. Hierdoor kunnen ze niet los van elkaar gepubliceerd worden. Op ten duur zorgt dit voor onnodig lange development cyclus.

2.3 Architectuurdiagram

Om een beeld te geven van het huidige proces van het klantenportaal is het onderstaande diagram opgesteld. Het diagram, welke te zien is in figuur 2.1, is een versimpelde versie van de daadwerkelijke architectuur. In dit diagram is te zien dat de gebruikers eerst een aanvraag doen aan het klantenportaal. Nadat de aanvraag bij het portaal is aangekomen wordt deze verwerkt door de services en de data access layer om informatie op te halen vanuit de database. Vervolgens wordt deze data weer teruggestuurd naar het portaal en de opgevraagde pagina opgebouwd op de server.



Figuur 2.1 - Architectuurdiagram .NET Framework 4.8

2.4 De overstap

Het team van het klantenportaal van PostNL heeft onder andere door de twee hierboven beschreven problemen de beslissing genomen om de overstap te maken. Het volledige front-end onderdeel van het portaal moest omgeschreven worden naar .NET Core. Hierdoor blijft het team vooruitstrevend en up-to-date met de laatste technieken. Daarbij hebben ze hierbij de mogelijkheid gecreëerd om de koppeling tussen de front-end en de back-end te verminderen.

3. Klantenportaal .NET Core 2.2

Zoals in de inleiding beschreven staat, bestaan er twee verschillende versies van het klantenportaal. Na het nieuws van Microsoft in het begin van 2019 (Asthana, 2019) heeft Nalta er voor gekozen om de overstap te gaan maken. Deze versie is ontwikkeld naast de versie die momenteel live staat. Echter, omdat deze nog niet volledig is getest, is er voor gekozen om deze nog niet live te zetten. In dit hoofdstuk staat meer informatie over .NET Core en wordt beschreven hoe de koppeling gedeeltelijk is opgelost. Om dit te verduidelijken is er een architectuurdiagram toegevoegd om de werking van dit portaal aan te tonen.

3.1 .NET Core

In 2016 heeft Microsoft .NET Core uitgebracht. Het grootste verschil van .NET Core ten opzichte van .NET Framework is dat .NET Core niet platform afhankelijk is. Deze kan naast de standaard Windows machine ook draaien op zowel Linux als Unix. Daarbij zijn er ook grote stappen gemaakt in het optimaliseren van de snelheid van het platform.

3.2 Koppeling

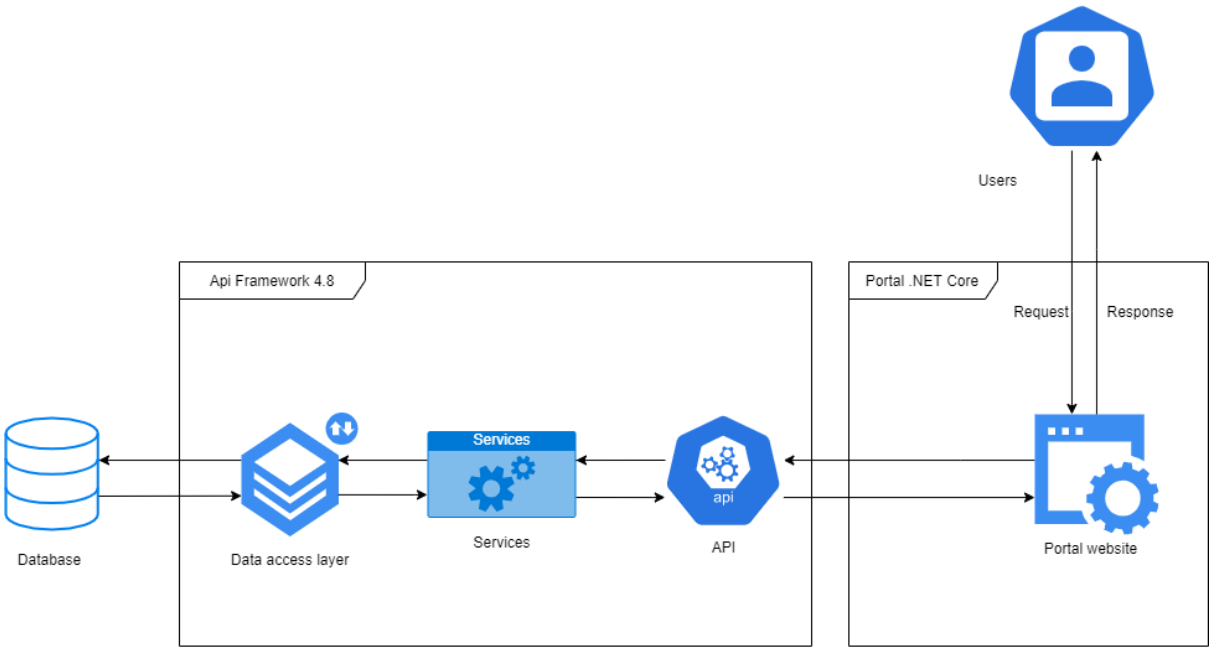
Zoals beschreven in hoofdstuk 2.2 bestaat er een hechte koppeling tussen de .NET Framework versie van klantenportaal en de back-end hiervan. Na verschillende opties te hebben bekeken heeft Nalta besloten om het klantenportaal van PostNL ook via de API data te laten ophalen. Een groot gedeelte van de functionaliteit was al aanwezig in de huidige API. Tijdens het omzetten zijn er een aantal methoden en functies aan de API toegevoegd. Hierdoor is het mogelijk om een volledige overstap te maken naar de nieuwe versie van het klantenportaal.

Echter, is er voor gekozen om deze versie nog niet live te zetten. Volgens stakeholders en het ontwikkelteam binnen Nalta en PostNL is deze versie nog niet volledig getest (interne communicatie, 2020). Om problemen met klanten voorkomen, is er besloten om hier nog mee te wachten. Wel hebben ze de webpagina’s al geprobeerd om op te halen buiten Europa. Dit leverde PostNL helaas nog steeds niet het gewenste resultaat op qua performance.

3.3 Architectuurdiagram

Ook voor de .NET Core variant is er een versimpeld architectuur diagram gemaakt van de daadwerkelijke architectuur. Dit diagram is te zien in figuur 3.1 Het grootste verschil ten opzichte van de .NET Framework versie is dat het portaal communiceert met de API in plaats van dat het allemaal op dezelfde server staat. Hierdoor hoeft het portaal niet op hetzelfde moment gepubliceerd te worden als de API en wordt downtime ook voorkomen.

Verder verloopt het proces hetzelfde als in de .NET Framework versie. Nadat de aanvraag bij de API is aangekomen wordt deze verwerkt door de services en de data access layer om informatie op te halen vanuit de database. Vervolgens wordt deze data weer teruggestuurd naar het portaal en de opgevraagde pagina gebouwd op de server.



Figuur 3.1 Architectuurdiagram .NET Core

4. Het Probleem

Het probleem zoals eerder al beschreven is dus dat performance van het klantenportaal niet optimaal is. Dit geldt voornamelijk voor klanten vanuit buiten Europa. Klanten vanuit buiten Europa zouden met relatief gelijke snelheid data moeten kunnen ophalen vanaf het klantenportaal. De reden dat dit zo belangrijk is voor PostNL, is dat het merendeel van de bedrijven die zijn aangesloten bij PostNL vanuit buiten Europa komt.

Doordat PostNL hier al een tijd mee kampt, zijn er al een aantal mogelijke oplossingen bedacht om de performance van het klantenportaal van PostNL te verbeteren. Een van de gebruikte oplossingen die uitgevoerd is door het team, was het omzetten van het klantenportaal naar .NET Core. Dit resulteerde in een kleine verbetering op in performance. In tabel 1 staan de laadsnelheid weergeven, welke duidelijk laat zien dat er nog een groot verschil zit tussen binnen en buiten Europa. Hierin is ook het verschil te zien tussen .NET Framework en .NET Core.

Regio	.NET Framework	.NET Core
Europa, Duitsland, Frankfurt	503 ms	**489 ms**
Europa, Engeland, Londen	506 ms	498 ms
Noord-Amerika, USA, San Francisco	3.02 s	2.76 s
Azië, Japan, Tokyo	5.19 s	4.28 s
Oceanië, Australië, Sydney	**6.41 s**	5.21 s

Tabel 1 (Laadsnelheid)

Literatuurlijst

Asthana, A. (2019, 10 mei).
Introducing .NET 5. Geraadpleegd op 5 maart 2020, van https://devblogs.microsoft.com/dotnet/introducing-net-5/

Requirements rapport

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
10-02-2020
Versie 1.0

Inhoudsopgave

1. Inleiding

Dit rapport bevat de wensen en eisen die gesteld worden aan de implementatie van mijn afstudeeropdracht bij Nalta. Deze opdracht heeft betrekking op het zakelijke klantenportaal van PostNL, waar verder in het document naar gerefereerd worden als het klantenportaal. De requirements zijn gebaseerd op de versie die op dit moment op de productieomgeving staat en fieldresearch. Om te beginnen wordt de scope van het systeem en de bijbehorende stakeholders gedefinieerd. Vervolgens worden de requirements opgesteld en worden hierbij de use cases weergegeven. Later in het document worden de technische beperkingen beschreven en de requirements geprioriteerd.

2. Systeem scope en stakeholders

Het huidige klantenportaal van PostNL moet worden herzien omdat het met performance problemen kampt voor bezoekers vanuit buiten Europa. In sommige regio's kan het 5 a 6 seconden duren voordat ze resultaat krijgen van de inlogpagina, welke één van de minst grootste pagina's is. Deze gebruikers ervaren hierdoor dat het te lang duurt voor ze de schermen van het portaal te zien krijgen. Hierbij is vanuit de opdrachtgever gegeven dat er geen aanpassingen aan de back-end zullen komen en dat deze dus buiten de scope valt.

2.1 Gebruikers van het systeem

In deze paragraaf worden de verschillende gebruikers van het klantenportaal beschreven en waarom ze het klantenportaal gebruiken.

Het klantenportaal

- Zakelijke klanten van PostNL
    - Zakelijke klanten van PostNL bevinden zich voornamelijk buiten Europa. Deze klanten vragen op regelmatige basis informatie op van het klantenportaal of informatie van de status van de verstuurde pakketten.
- Werknemers van PostNL
    - Naast de zakelijke partners van PostNL wordt het klantenportaal ook frequent gebruikt door werknemers van PostNL. De informatie van de status van de verstuurde pakketten worden opgehaald en het systeem wordt steekproefsgewijs getest via het portaal.

2.2 Functionaliteit

Het opgeleverde prototype moet voldoen aan de basis functionaliteit van het huidige klantenportaal. Pas wanneer de gekozen basis functionaliteit representatief is voor de functionaliteit van het huidige portaal kunnen beide platformen met elkaar worden vergeleken. Vervolgens kan er een uitspraak worden gedaan of er daadwerkelijk een optimalisatie heeft plaatsgevonden op het klantenportaal.

3. Business requirements

In dit hoofdstuk staan de business requirements beschreven die betrekking hebben op het prototype van de gegeven opdracht. Bij elke business requirement staat ook de belangrijkste stakeholder en de bron waar het requirement uit gehaald is.

id	Beschrijving	Stakeholder	Bron
BR1	De werknemers van PostNL willen dezelfde functionaliteit hebben als in het originele portaal.	Werknemers	Opdracht beschrijving
BR2	PostNL wil dat de geografische locatie weinig effect heeft op de laadsnelheid van het portaal.	PostNL	Opdracht beschrijving
BR3	PostNL wil dat het portaal sneller wordt.	PostNL	Opdracht beschrijving

Tabel 3.1 - Requirements

4. Use cases

In de eerder beschreven business requirements heeft alleen BR1 betrekking op functionaliteit. Deze functionaliteit wordt in de tabel 4.1 beschreven door een aantal use cases. De uses cases gaan alleen in op de basisfunctionaliteit die in het prototype moet zitten zoals in hoofdstuk 2.2 ‘Functionaliteit’ staat. Deze use cases zijn gekozen omdat deze representatief staan voor de volledige functionaliteit van het huidige platform. De kern van de opdracht zit voornamelijk in BR2 en BR3, welke meer betrekkingen hebben op de non-functionele requirements. Om aan te tonen dat aan deze requirements is voldaan moet dit getest worden op een prototype. Naast de opsommingen van de uses cases worden deze in hoofdstuk 4.2 visueel laten zien aan de hand van een use case diagram.

4.1 Use case opsomming

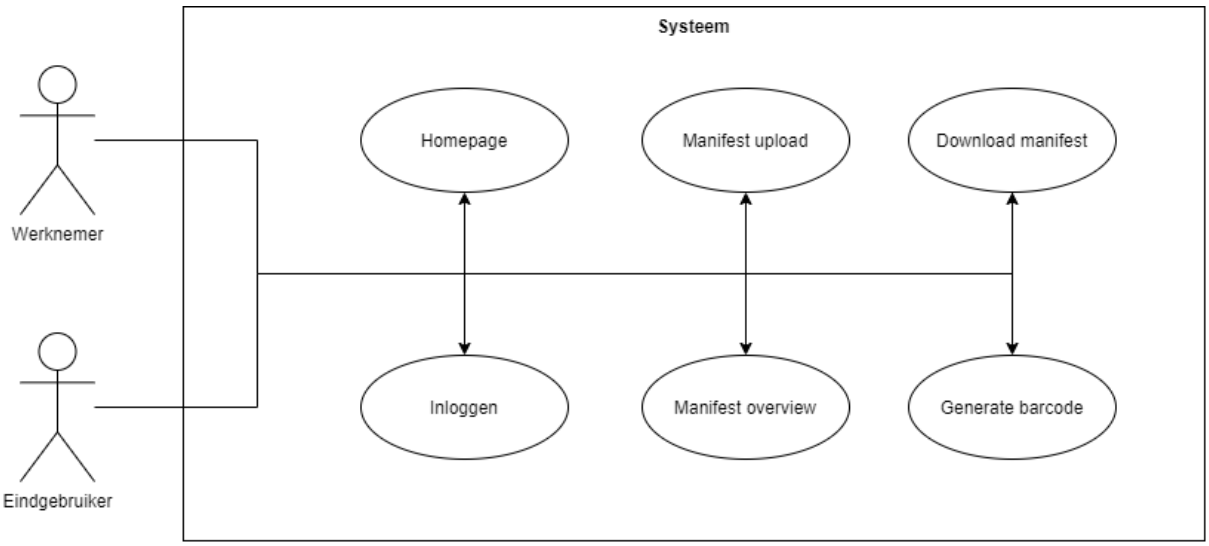
Zoals in het vorige hoofdstuk beschreven staat hebben de use cases betrekking op BR1. Om deze reden wordt de kolom ‘Business requirement’ niet meegenomen in de tabel. Gezien het prototype een groot deel van reeds bestaande functionaliteit moet bevatten, heeft het weinig zin om aan de hand van MoSCoW te prioriteren. In dat geval had het overgrote deel een ‘must have’ geweest. De priorisering is gedaan aan de hand van de meest gebruikte methodes binnen het portaal. Deze priorisering is besproken met de de heer M. Beenes ,lead developer binnen Nalta. De priorisering loopt op vanaf nummer 1.

#U	Use case	Beschrijving	Prioritisering
U1	Beginscherm	De klant moet het beginscherm kunnen laden	2
U2	Inloggen	De klant moet kunnen inloggen op het platform	1
U3	Manifest upload	De klant moet manifesten kunnen uploaden	3
U4	Manifest overview	De klant moet het manifest overzicht kunnen bekijken	6
U5	Download manifest	De klant moet een manifest kunnen downloaden	4
U6	Genereer assist label	De klant moet de mogelijkheid hebben om een assist label te genereren	5

Tabel 4.1 - Use cases

4.2 Use case diagram

In figuur 3.1 zijn de eerder beschreven use cases in kaart gebracht. Aan de linkerzijde staan de actoren. In dit diagram worden er connecties gemaakt tussen de actoren en de use cases waar zij betrekking op hebben.



Figuur 3.1 - Use case diagram

5. Niet-functionele requirements

In de volgende tabel, tabel 5.1, staan de niet-functionele requirements voor de opdracht beschreven. Naast een identifier staat het requirement ook beschreven. Het requirement wordt met een kwaliteitskenmerk gecategoriseerd vervolgens de ISO 25010. ISO 25010 is de opvolger van ISO 9126 en wordt gebruikt om kwaliteitskenmerken te geven aan software. Daarbij wordt de bron van het niet-functionele requirement aangegeven.

id	Requirement	ISO 25010 Kwaliteitskenmerk	Bron
NF1	Een webpagina moet buiten Europa sneller resultaat geven dan in de huidige situatie	Time behaviour	Opdracht omschrijving
NF2	De systeem taal is engels	Usability	Opdracht omschrijving
NF3	Binnenkomende requesten moeten via een secure connection worden verwerkt	Functionality	Opdracht omschrijving
NF4	De UI van applicatie moet automatisch getest kunnen worden	Testability	Opdracht omschrijving

Tabel 5.1 - Niet-functionele requirements

6. Use case beschrijving

Hieronder staan de verschillende use cases beschreven in meer detail om hier een duidelijker overzicht van te krijgen.

Naam	Beginscherm
ID	U1
Requirement	R1
Beschrijving	De klant moet het beginscherm kunnen laden
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker bestaat in de database 2. De gebruiker zit in een rol
scenario	1. De gebruiker logt in het portaal 2. Het systeem laat de toelaatbare acties zien
postconditie	1. De gebruiker ziet acties die hij kan uitvoeren
Alternatieve flows	

Naam	Inloggen
ID	U2
Requirement	R1
Beschrijving	De klant moet kunnen inloggen op het platform
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker bestaat in de database 2. De gebruiker zit in een rol
scenario	1. Vult zijn gebruikersnaam en wachtwoord in 2. De gebruiker drukt op inloggen 3. Het systeem controleert de gegevens [1][2]
postconditie	1. De gebruiker ziet dat hij is ingelogd
Alternatieve flows	[1] De gebruiker krijgt een foutmelding dat de gebruikersnaam of wachtwoord fout is. [2] De gebruiker krijgt een melding dat hij geblokkeerd is.

Naam	Manifest upload
ID	U3
Requirement	R1
Beschrijving	De klant moet manifesten kunnen uploaden
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker is ingelogd 2. De gebruiker zit in een rol 3. De gebruiker zit gekoppeld aan een klant
scenario	1. De gebruiker upload een manifest 2. Het systeem controleert het bestand [1] 3. Het systeem slaat het bestand op [2]
postconditie	1. Een manifest is geupload en klaar om verwerkt te worden door de API
Alternatieve flows	[1] Het geuploade bestand is geen manifest en het systeem geeft een error terug [2] Het opslaan ging mis en het systeem geeft een error code terug aan de gebruiker

Naam	Manifest overview
ID	U4
Requirement	R1
Beschrijving	De klant moet het manifest overzicht kunnen bekijken
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker is ingelogd 2. De gebruiker zit gekoppeld aan een klant
scenario	1. De gebruiker gaat naar de manifest overzichtspagina 2. Het systeem laat de manifesten voor de klant zien [1]
postconditie	1. Manifesten voor de gebruiker zijn klant worden weergeven
Alternatieve flows	[1] Er wordt een foutmelding weergeven dat er geen manifesten zijn voor de gebruiker zijn klant

Naam	Download manifest
ID	U5
Requirement	R1
Beschrijving	De klant moet een manifest kunnen downloaden
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker is ingelogd 2. De gebruiker zit gekoppeld aan een klant 3. Er zijn manifesten geupload voor de gekozen klant
scenario	1. De gebruiker gaat naar de manifest overzichtspagina 2. Het systeem laat de manifesten voor de klant zien 3. De gebruiker download een manifest [1]
postconditie	1. Het gekozen manifest wordt gedownload
Alternatieve flows	[1] Er wordt een foutmelding weergeven dat er een fout heeft opgetreden tijdens het downloaden van een manifest

Naam	Genereer barcode
ID	U6

Requirement	R1
Beschrijving	De klant moet de mogelijkheid hebben om een barcode te genereren
Primaire actors	Klant
Secundaire actors	-
preconditie	1. De gebruiker is ingelogd 2. De gebruiker zit gekoppeld aan een klant
scenario	1. De gebruiker gaat geneer barcode pagina 2. De gebruiker genereert een barcode 3. Het systeem haalt een barcode op vanuit de API [1]
postconditie	1. Er wordt een gegenereerde barcode laten zien
Alternatieve flows	[1] Er wordt een foutmelding weergeven dat er een fout heeft opgetreden tijdens genereren van een barcode

7. Technische beperkingen

In dit hoofdstuk worden de technische beperkingen beschreven die betrekking hebben op de opdracht. Deze zijn toegevoegd in tabel 7.1. Deze beperkingen moeten in het achterhoofd gehouden worden terwijl het onderzoek wordt uitgevoerd en er een prototype wordt ontwikkeld.

id	Beperking
TB1	De front-end en de back-end moeten apart van elkaar gepubliceerd kunnen worden.
TB2	Autorisatie aan de hand van JWT technieken.
TB3	Het moet cloudplatform agnostisch zijn.
TB4	Er moet gebruikgemaakt worden van binnen het bedrijf bekende programmeertalen (C#, JavaScript).
TB5	Er worden geen aanpassingen gedaan aan de back-end.
TB6	Geografische replicatie van de back-end is geen optie.

Tabel 7.1 - Technische beperkingen

8. Conclusie

Na het opstellen van de de verschillende hoofdstukken in dit requirements rapport kan er nu gericht begonnen worden aan het onderzoek. De verschillende requirements zijn in kaart gebracht en hierbij ook de use cases die daar betrekking op hebben. Als laatste zijn ook de technische beperkingen beschreven zodat duidelijk is waar er rekening mee gehouden moet worden tijdens het doen van het onderzoek of het ontwikkelen van het prototype.

Performance onderzoek

Performance optimalisatie bij PostNL

K.C.J van Zeijl
De Haagse Hogeschool
Software Engineering
16067614
Leerjaar 4 (2019 - 2020)

Wateringen
10-02-2020
Versie 2.0

Inhoudsopgave

Begrippenlijst

.NET is een open source software framework dat is ontwikkeld door Microsoft. .NET kan worden gebruikt om applicaties te ontwikkelen die platform onafhankelijk zijn.

API staat voor Application Programming Interface. Een API wordt gebruikt om informatie te verschaffen via een link zonder dat hierbij schermen getoond worden.

Artificial Intelligence ook wel bekend als AI, is een complex begrip dat vaak wordt gebruikt om de intelligentie van machines te beschreven en hoe zij berekeningen uitvoeren.

Cachen houdt in dat statische data voor een ingestelde tijd bewaard kan blijven. Op deze manier hoeft het niet elke keer van de database opgehaald te worden.

CMS staat voor Content Management System. Een CMS wordt gebruikt om het bijhouden van een applicatie te versimpelen. Een goed voorbeeld hiervan is WordPress.

Comprimeren
Tijdens comprimeren worden bestanden kleiner gemaakt zodat er minder data gestuurd hoeft te worden naar de eindgebruiker. Vervolgens wordt het bij de eindgebruiker weer uitgepakt.

DDOS staat voor Distributed Denial-of-service. Malafide hackers maken vaak gebruikt van DDOS aanvallen om webpagina's offline te halen.

DOM staat voor Document Object Model. Het HTML DOM is een standaard voor het ophalen,veranderen, toevoegen en verwijderen van HTML elementen op een webpagina.

ICMP staat voor Internet Control Message Protocol. Dit is een bericht van een standaard grootte die veelal gebruikt wordt om snelheid van internet te testen.

Latency is het tijdsinterval tussen een stimulatie die een bepaalde reactie verwacht en de daadwerkelijke reactie. In de IT wereld wordt dit veelal gebruikt bij internetconnecties (The Linux Information Project, 2005).

MVC staat voor Model, View en Controller. Een architectuur techniek die wordt gebruikt om de logica van een webapplicatie te scheiden in drie componenten.

1. Inleiding

De performance van een website kan een enorme impact hebben op een bedrijf. Volgens een onderzoek dat is gedaan door google (ThinkwithGoogle, 2019) is de kans dat bezoekers je website na bezoek direct verlaten 32% groter wanneer de laadtijd van 1 naar 3 seconden wordt verhoogd. Wanneer de website pas binnen 6 seconden is geladen ligt dit percentage zelfs op 106%. Door deze performance degradatie raken mensen sneller gefrustreerd en gaan mogelijk klagen.

Dit is ook het geval bij het zakelijke klantenportaal van PostNL. Wanneer klanten vanaf buiten Europa het klantenportaal proberen te benaderen vinden zij dat het te lang duurt voor zij hiervan resultaat zien. Uit een test van het vooronderzoek is gebleken dat dit tot wel 10 maal langer kan duren wanneer de schermen vanuit oceanië worden opgehaald.

Het onderzoek is uitgevoerd om verschillende oplossingen te zoeken die het probleem voor PostNL kunnen verhelpen. In hoofdstuk 2 wordt allereerst de aanleiding van het onderzoek besproken met hierbij een uitleg over de huidige situatie. Vervolgens beschrijft hoofdstuk 3 de onderzoeksopzet met hierin de onderzoeksvragen en volgens methoden hieraan gewerkt gaat worden tijdens dit onderzoek. In de daaropvolgende hoofdstukken worden de verschillende deelvragen beantwoord om als laatste een conclusie te geven waaruit een onderzoekshypothese volgt van een geschikte oplossing.

2. Aanleiding van het onderzoek

In dit hoofdstuk staat de aanleiding tot het onderzoek beschreven. Beginnend met de aanleiding tot de afstudeeropdracht. Vervolgens wordt dit aangevuld met de achtergrond van het project en daarbij een korte beschrijving over welke technieken er al zijn toegepast voor het probleem. Als derde wordt de scope van het probleem bepaald. Het laatste deelhoofdstuk beschrijft de probleemstelling.

2.1 Aanleiding van de afstudeeropdracht

De opdracht heeft betrekking op de versie van het zakelijke klantenportaal van PostNL (hier zal verder naar gerefereerd worden als het klantenportaal). Een aantal bedrijven klaagt over de snelheid van het portaal. Dit geldt voornamelijk voor bedrijven die vanaf buiten Europa het portaal benaderen. Deze gebruikers vinden dat het te lang duurt voordat de schermen van het portaal geladen worden. Om deze reden heeft PostNL aan Nalta, wie het portaal heeft gemaakt, gevraagd wat ze eraan kan doen om de snelheid hiervan te verbeteren.

2.2 Achtergrond

Zoals beschreven in het document ‘Huidige situatie’ werd bij aanvang van de opdracht al snel duidelijk dat er twee verschillende versies van het klantenportaal bestaan. De reden hiervoor is dat huidige versie welke live staat een aantal problemen heeft. Een belangrijk probleem hiervan is de performance. Zakelijke klanten buiten Europa ervaren dat te lang duurt voordat de schermen van het klantenportaal getoond worden. Deze versie van het klantenportaal is geschreven met behulp van .NET Framework 4.8.

In de laatste maanden is er ook al hard gewerkt aan een vernieuwde versie van het klantenportaal. Deze nieuwe versie is geschreven met behulp van .NET Core 2.2. De nieuwe versie is ontwikkeld om een aantal redenen. Allereerst is .NET Framework 4.8 de laatste versie in de .NET Framework reeks (Hunter, 2019). Hierna wordt er alleen nog verder ontwikkeld aan .NET Core. Met een blik op de toekomst raadt Microsoft dan ook aan om alleen nog .NET Core te gebruiken. Daarnaast heeft Nalta in 2018 besloten dat de standaard architectuur bestaat uit een back-end en front-end project welke los van elkaar gepubliceerd kunnen worden. Hierbij haalt de front-end data op van de back-end door middel van een API. Door de software in .NET Core opnieuw te schrijven kon Nalta de code gedeeltelijk ontkoppelen van de back-end code. Als laatste was het doel om de performance te verbeteren met deze overstap. Helaas leverde deze overstap niet de gewensten optimalisatie op.

Zoals hierboven beschreven, heeft de transitie naar .NET Core een aantal oplossingen geleverd voor PostNL. Zo blijven de gebruikte technieken vooruitstrevend en zijn de back-end en front-end gescheiden van elkaar. Het probleem dat nu nog opgelost moet worden is de slechte performance voor gebruikers buiten Europa. Gezien zij ervaren dat het te lang duurt voor de schermen van het klantenportaal worden geladen.
Tijdens deze transitie is er echter wel voor gekozen om het aloude MVC, Model-view-controller, te gebruiken waarbij het volledige scherm opgebouwd wordt op de server. Door

het volledige scherm te sturen naar de gebruiker in plaats van alleen de opgevraagde data kan dit ervoor zorgen dat het langer duurt voor eindgebruiker het scherm te zien krijgen.

2.3 Project scope

De projectscope is beschreven om te bepalen tot waar het onderzoek afgebakend moet worden. Om dit goed te bepalen zijn er verschillende gesprekken gevoerd met de stakeholders binnen PostNL en Nalta. Aan de hand van deze gesprekken is duidelijk geworden dat er geen aanpassingen aan de back-end zullen worden doorgevoerd. Hiermee valt de back-end dus buiten de scope.

2.4 Probleemstelling

Na de achtergrond van het onderzoek te hebben beschreven is de volgende probleemstelling opgesteld. De zakelijke klanten van PostNL vinden dat het te lang duurt voordat de schermen van het klantenportaal getoond worden.

2.5 Doel van het onderzoek

Het doel van dit onderzoek is om te onderzoeken wat de oorzaak van het klantenportaal performance probleem is en om hiervoor een geschikte oplossing voor te vinden. Hierna zullen de zakelijke klanten buiten Europa moeten merken dat het klantenportaal sneller reageert dan voorheen.

3. Onderzoeksopzet

In dit hoofdstuk staat de opzet van het onderzoek beschreven. In deelhoofdstuk 3.1 ’Onderzoeksvragen’ worden als eerste de onderzoeksvragen beschreven. Hierbij is een hoofdvraag opgesteld met de bijbehorende deelvragen. Vervolgens staat in hoofdstuk 3.2 ‘Methodologie’ beschreven hoe er aan de deelvragen wordt gewerkt.

3.1 Onderzoeksvragen

Uit de aanleiding en probleemstelling van hoofdstuk 2 is de volgende centrale hoofdvraag geformuleerd:

“Hoe kan Nalta de performance van het klantenportaal van PostNL in regio’s buiten Europa significant verbeteren?”

Om antwoord te kunnen geven op de centrale hoofdvraag dienen de onderstaande deelvragen beantwoord te worden:

1. Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa?
2. Welke bestaande methoden kunnen worden toegepast om het serveren van een webapplicatie te versnellen?

Na het beantwoorden van de hoofdvraag kan er een aanbeveling worden gedaan over wat de meest geschikte oplossing voor PostNL is binnen de huidige infrastructuur.

3.2 Methodologie

In de methodologie staat belangrijke informatie beschreven met betrekking tot het onderzoek. Zo staat onder andere het soort onderzoek beschreven, maar ook de methoden en technieken die zijn gebruikt tijdens het doen van het onderzoek.

3.2.1 Aard van het onderzoek

Het onderzoek heeft een exploratief aard, van te voren is nog niet duidelijk wat het resultaat gaat zijn dat uit dit onderzoek komt. Daarbij is er geen hypothese die getoetst kan worden of wordt er een bepaalde groep beschreven. Tijdens het onderzoek wordt er door middel van kwalitatieve onderzoeksmethoden gezocht naar een antwoord op de hoofdvraag. Hierdoor is het onderzoek geclassificeerd als een exploratief onderzoek in de vorm van een kwalitatief onderzoek.

3.2.2 Literatuuronderzoek

Literatuuronderzoek is gebruikt om meer kennis te vergaren voor het theoretisch kader van het onderzoek. Hiervoor hebben zijn verschillende ebooks en internetbronnen gelezen. De kennis is gebruikt tijdens het beantwoorden van de verschillende deelvragen.

3.2.3 Desk research

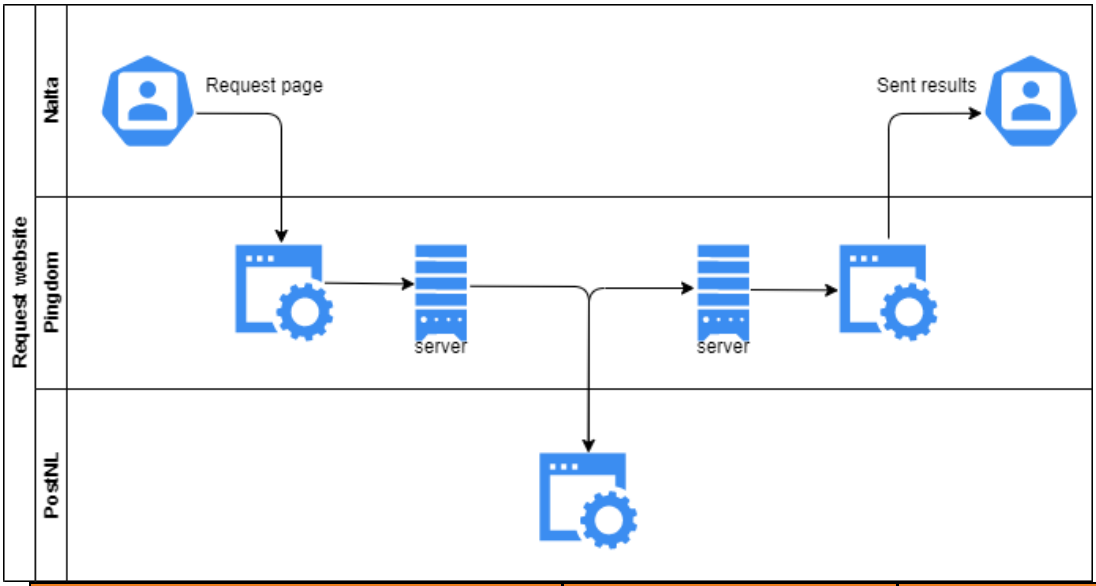
Om correct antwoord te kunnen geven op de verschillende deelvragen is er veel gebruikt gemaakt van deskresearch. Allereerst om te onderzoeken of er niet al eerder onderzoek is gedaan naar vergelijkbare problemen, maar ook om te valideren of gevonden methoden juist zijn.

4. Deelvraag 1

Om duidelijk te krijgen wat momenteel het probleem is met het klantenportaal is de volgende vraag opgezet: **“Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa”**. Nalta heeft vanuit PostNL begrepen dat het voor gebruikers buiten Europa te lang duurt voor ze de schermen van het klantenportaal te zien krijgen. In het document ‘Huidige situatie’ is er al een test uitgevoerd om te controleren of er daadwerkelijk verschil zit tussen de verschillende continenten.

Tijdens deze test is er gebruik gemaakt van de service van Pingdom, een betrouwbare partij voor het testen van websites vanuit Nalta. Pingdom heeft een groot aantal servers verspreid over de wereld. Via de website kan een snelheidstest worden aangevraagd voor een specifieke website vanuit een van Pingdoms servers. Op dat moment wordt de website vanuit die server opgehaald en wordt bijgehouden hoe lang het duurt voor de website volledig geladen is, wat zichtbaar is gemaakt in figuur 4.1. De test is vanuit de verschillende continenten tien maal uitgevoerd via Pingdom.com (https://www.pingdom.com/). Van de testresultaten staat het gemiddelde qua laadsnelheid in Tabel 4.1. Achter de gemiddelden wordt de standaarddeviatie aangegeven.

Figuur 4.1 - Pingdom test



Regio	.NET Framework	.NET Core
Europa, Duitsland, Frankfurt	595 ms (σ 93)	**540 ms** (σ 97)
Europa, Engeland, Londen	607 ms (σ 97)	565 ms (σ 101)

Noord-Amerika, USA, San Francisco	3021 ms (σ 314)	2766 ms (σ 267)
Azië, Japan, Tokio	5196 ms (σ 480)	4286 ms (σ 445)
Oceanië, Australië, Sydney	**5974 ms** (σ 557)	5214 ms (σ 521)

Tabel 4.1 -Laadsnelheid

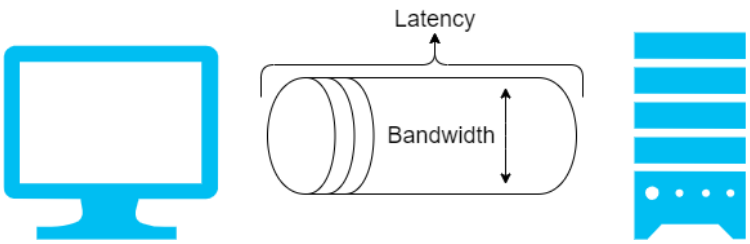
Deze tabel laat duidelijk zien dat er inderdaad een groot verschil in laadsnelheid is tussen de verschillende continenten. Ook laat deze tabel zien dat de .NET Core versie wel al een kleine verbetering biedt in de laadsnelheid van de pagina's. Er is dus vastgesteld dat het probleem zich inderdaad voordoet. Hierom riep dit de volgende vraag op 'Wat is de reden achter de grote verschillen tussen de continenten?'.

4.1 Latency

Kijkend naar de resultaten uit tabel 4.1, lijkt er vrijwel zeker een latency probleem op te treden. Latency is een gevolg van de beperkte snelheid waarmee elke fysieke interactie zich kan verspreiden. In de ICT wereld wordt latency vaak gemeten in milliseconden. Een herkenbaar meet voorbeeld hiervan is het doen van een snelheidsmeting. Tijdens een snelheidsmeting wordt er een ICMP (Internet Control Message Protocol) gestuurd en beantwoord. De latency hiertussen wordt uitgedrukt in ms en naar gerefereerd als ping.

Latency wordt vaak verward met bandbreedte, welke in hoofdstuk 4.2 wordt behandeld. Om het verschil aan te duiden wordt het volgende voorbeeld gebruikt:

- Bandbreedte heeft er mee te maken hoe wijd of smal een pijp is; hoeveel er door de pijp heen kan, weergeven in figuur 4.2
- Latency heeft met de inhoud van de pijp te maken; hoe snel het van de ene kant naar de andere kant van de pijp beweegt, weergeven in figuur 4.2



Figuur 4.2 - Latency & bandwidth

De definitie van goede latency wordt vaak relatief gezien. Over het algemeen is een latency van minder dan 100 ms gewenst. Wanneer er online spellen worden gespeeld is de latency erg belangrijk en wordt er eerder een latency van 30 tot 50 ms verwacht. De factoren die invloed hebben op de latency zullen in de volgende paragraaf worden behandeld.

4.1.1 Invloeden op latency

In dit hoofdstuk worden een aantal factoren besproken die invloed kunnen hebben op de latency. Uit recent onderzoek van 'Plug Things In' (Plug things in, z.d.) blijkt dat vooral het

connectie type en de afstand invloed hebben op de latency. Deze worden hieronder beschreven.

Connectie type
Het type connectie tussen de twee systemen kan van grote invloed zijn op de latency hiertussen. Internet via een satelliet is hier een goed voorbeeld van. Satellieten staan honderden kilometers van ons vandaan. Wanneer je met een satelliet probeert te verbinden wordt er eerst een aanvraag gedaan naar de satelliet. Hierna wordt de aanvraag doorgestuurd naar een ISP (Internet Service Provider) en naar het internet. Vervolgens gaat het antwoord via dezelfde weg weer terug naar de gebruiker. Vanzelfsprekend duurt het bij zo een connectie erg lang voor de gebruiker resultaat ziet. Een ping langer dan 500 ms is in dit geval geen uitzondering (Ground Control, z.d.). Dit is extreem hoog wanneer dit wordt vergeleken met een ping van 4ms over het ethernet bij het afstudeerbedrijf.

Afstand
Naast het connectie type heeft de afstand tussen de twee systemen ook een een grote invloed op de latency. Uit een onderzoek van Noction (Noction, 2015) blijkt dat vooral tussenliggende routers een grote invloed hebben op de latency. De snelheid binnen kabels waarmee zij verbonden zijn kunnen oplopen tot wel ⅔ van de lichtsnelheid (200.000 km/s). Hierdoor kan een pakket binnen milliseconden aan de andere kant van de wereld zijn. Echter, elke router die tussen de systemen ligt heeft tijd nodig om het verstuurde pakket te verwerken voor deze naar de volgende router verstuurd kan worden. Een kabel heeft een maximale lengte waardoor er op een grote afstand een groot aantal routers tussen de eindgebruiker en de server staan. Dus hoe groter de afstand tussen deze systemen is, hoe meer routers er tussen de server en de eindgebruiker staan. Om deze verwerkingstijd van de routers duurt het langer voor de informatie bij de eindgebruiker aankomt.

4.1.2 Invloeden bij PostNL
Zowel het connectie type als de afstand kunnen een invloed hebben op het klantenportaal van PostNL. De test waarvan de resultaten te zien zijn in tabel 4.1 zijn allemaal uitgevoerd via een ethernet verbinding. Hierom is het connectie type tijdens deze test weinig van invloed op de latency. De invloed van de afstand op het klantenportaal is wel duidelijk te zien. Hoe groter de afstand tussen de verschillende continenten en de server in Europa hoe hoger de latency.

4.2 Bandbreedte

Het begrip bandbreedte komt origineel vanuit de telecommunicatie wereld. Hierbij gaf de bandbreedte het verschil aan tussen de hoogste en laagste doorgelaten frequentie binnen een transmissiekanaal. Deze frequentie wordt gemeten in hertz (Hz). Pas later is er voor gekozen om het begrip bandbreedte ook te gebruiken binnen de datacommunicatie wereld. Hierbij wordt deze oneigenlijk gebruikt om de transportsnelheid aan te duiden. Over het algemeen wordt hiervoor de eenheid bits per seconde (bps) voor gebruikt (WikiBooks, z.d.). Niet te verwarren met Bytes, bestaande uit acht bits, die worden gebruikt voor dataopslag.

4.2.1 invloeden op bandbreedte

In dit hoofdstuk worden een aantal onderwerpen besproken die invloed kunnen hebben op de bandbreedte. Het medium is samen met congestie het meest voorkomende probleem voor een lage bandbreedte.

Medium

Binnen datacommunicatie is de bandbreedte vooral afhankelijk van het medium waar de data over gestuurd wordt. Zo werden er vroeger modems gebruikt die een maximale snelheid aankonden van 14.4 kbit/s, 28.8 kbit/s of 56 kbit/s. Deze zijn onderling vaak weer verbonden met Coaxiale kabels die een maximale snelheid van 10 Mbit/s konden bereiken en maximaal 185 meter lang konden zijn. Tegenwoordig wordt er binnen Nederland veelal glasvezel gebruikt welke een snelheid kan bereiken van 10 Gbit/s. Het gekozen medium zal dus een groot effect hebben op de gemeten bandbreedte.

Congestie

Congestie staat voor ophoping of verstopping. Congestie treedt op wanneer een netwerkknooppunt meer informatie binnenkrijgt dan dat het kan verwerken. Een aantal neven effecten die hierbij kunnen optreden zijn lange wachtrijen van berichten, pakketten die verloren raken of blokkades van nieuwe binnenkomende connecties. Wanneer er meer data wordt aangevraagd dan dat deze knooppunten aankunnen kan het dus zo zijn dat de eindgebruiker deze langzamer binnenkrijgt dan hij / zij initieel aankan. Deze limitatie wordt veelal door hackers gebruikt om bepaalde websites of servers te overbelasten en worden ook wel DDoS aanvallen genoemd.

4.2.2 Invloeden bij PostNL

Tijdens de test die hoort bij tabel 4.1 is het type ethernet verbinding onveranderlijk gebleven, hiervoor is glasvezel gebruikt. Hierdoor tonen de testresultaten van tabel 4.1 niet aan of het bandbreedte medium een impact heeft op de performance. Wanneer eindgebruikers een minder snel medium gebruiken dan glasvezel kan dit effect hebben op de snelheid waarmee de schermen worden opgehaald. Echter, voor PostNL of Nalta is het niet mogelijk om dit aan te passen voor haar klanten. Doordat er in de test gebruikt is gemaakt van Pingdom via een glasvezel internetverbinding is de kans klein dat er congestie heeft opgetreden tijdens de test. Dit sluit net als bij het medium niet uit dat klanten buiten Europa wel met dit probleem te maken hebben.

4.3 Tussentijdse conclusie

Om antwoord te geven geven op de eerste deelvraag **“Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa?”** is er onderzocht er daadwerkelijk een performance probleem optreedt en wat hiervan de oorzaak is. De resultaten van tabel 4.1 laten zien dat er merkbaar verschil in laadsnelheid per continent aanwezig is. Zo is een simpele pagina binnen Europa al in 500 ms geladen, maar duurt het in Oceanië minstens tien keer langer om dezelfde simpele pagina te laden. Wanneer er na het uitleggen van de begrippen latency en bandbreedte nogmaals naar de resultaten van tabel 4.1 wordt gekeken kan geconcludeerd worden dat het klantenportaal van PostNL veel last heeft van latency. Dit komt voornamelijk doordat de afstand tussen de eindgebruiker en de server erg groot is. Daarnaast is het ook mogelijk dat de schermen langzamer laden

vanwege congestie of het gebruikte internet medium. Echter, dit is niet te herleiden uit de uitgevoerde test.

In het volgende hoofdstuk worden verschillende oplossingen beschreven die kunnen bijdragen aan het verbeteren van de performance. Hiermee zal de derde deelvraag beantwoord worden.

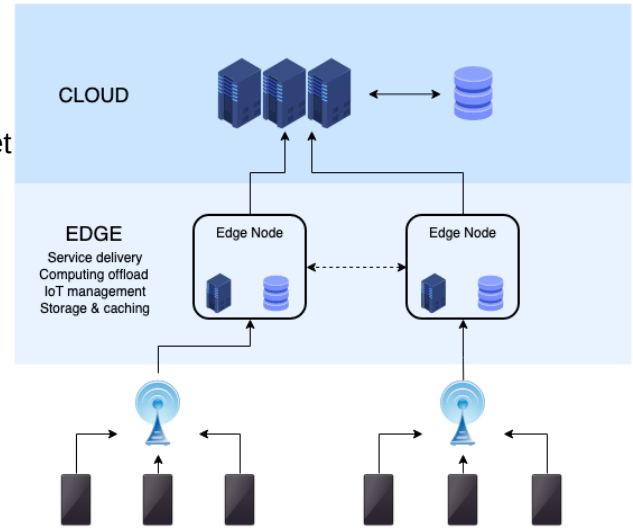
5. Deelvraag 2

Aan de hand van het onderzoek dat is uitgevoerd in hoofdstuk 4 is gebleken dat er daadwerkelijk een performance probleem optreedt met het klantenportaal buiten Europa. Om er achter te komen wat er gedaan kan worden om de performance te verbeteren is volgende deelvraag opgesteld: **“Welke bestaande methoden kunnen worden toegepast om het serveren van een webapplicatie te versnellen?”**.

Er bestaan verschillende manieren om de latency van een website te verlagen. Door middel van grondig onderzoek zijn er een aantal veel gebruikte methoden gevonden. Deze methoden worden in de volgende paragrafen beschreven.

5.1 Edge computing

Edge computing is een van de oplossingen die volgde uit het vooronderzoek van dit hoofdstuk. Letterlijk vertaald staat het voor berekenen op het randje. Strikt gezien is dit ook wat het doet. Een deel van de software draait hierbij op een server zo dicht mogelijk bij de klant, dit wordt on-premise genoemd (Arabi, 2014). Om deze reden is de opgevraagde data ontzettend snel terug bij de klant. Daarbij scheelt het bandbreedte voor de centrale server omdat niet alle data teruggestuurd hoeft te worden naar deze server. Dit is duidelijk te zien in figuur 5.1. Edge computing is het meest toepasbaar voor software waar real-time data noodzakelijk is. Een goed voorbeeld van een edge computing provider is de Azure IoT Edge (https://azure.microsoft.com/en-us/services/iot-edge/). Momenteel wordt de Azure IoT Edge vaak gebruikt voor Artificial Intelligence berekeningen. Deze geeft het bedrijf dan snel inzichten die gebruikt kunnen worden voor kritieke bedrijfsbeslissingen.



Figuur 5.1 - Edge computing (Edge computing infrastructure, 2019)

5.1.1 Voor- en nadelen

Elke gevonden oplossing heeft natuurlijk zijn voor- en nadelen. In dit hoofdstuk worden eerste de voordelen beschreven die zijn onderzocht voor edge computing. Vervolgens worden ook de nadelen beschreven die edge computing met zich meebrengt.

Voordelen

Lagere latency

Doordat de afstand tussen de eindgebruiker en de computing server kan worden beperkt tot enkele meters kan kost het minder tijd voor de eindgebruiker resultaat ziet van zijn of haar request. Dit lost met name het probleem op dat is beschreven in paragraaf 4.1.1 (Invloeden op latency).

Beschikbaarheid

Door gebruikt te maken van edge computing zorgt het bedrijf ervoor dat zij beter bestendig is tegen aanvallen van hackers. Voor hackers is het lastiger is om een DDOS aanval op alle verspreide servers uit te voeren. Hierdoor kan een betere beschikbaarheid gegarandeerd worden (Gyarmathy, 2019). Een ander bijkomend voordeel voor de beschikbaarheid van edge computing is dat een on-premise server niet perse een internetverbinding hoeft te hebben om de data van de klant te verwerken. De server kan direct worden aangesloten op de benodigde apparaten. Een werkende internetverbinding is pas weer nodig wanneer de verwerkte data naar de centrale server moet worden gestuurd. Op deze manier kun je garanderen dat de data altijd verwerkt wordt.

Nadelen

Prijzig

Zoals figuur 5.1 laat zien, staan er naast de centrale server ook nog verschillende edge nodes verspreid. In een desbetreffende edge node staan naast de software logica ook storage en caching mechanisms. Al deze services kosten weer extra geld bovenop de centrale server.

Server per klant

Naast de extra kosten dat edge computing met zich meebrengt, zullen er ook edge computing servers geïnstalleerd moeten worden per klant. Dit is vooral erg handig wanneer je een aantal klanten hebt die erg intensieve berekeningen doen. Als een bedrijf een hoop klanten heeft is het lastig om overal een server te plaatsen.

5.1.2 Toepasbaar voor PostNL

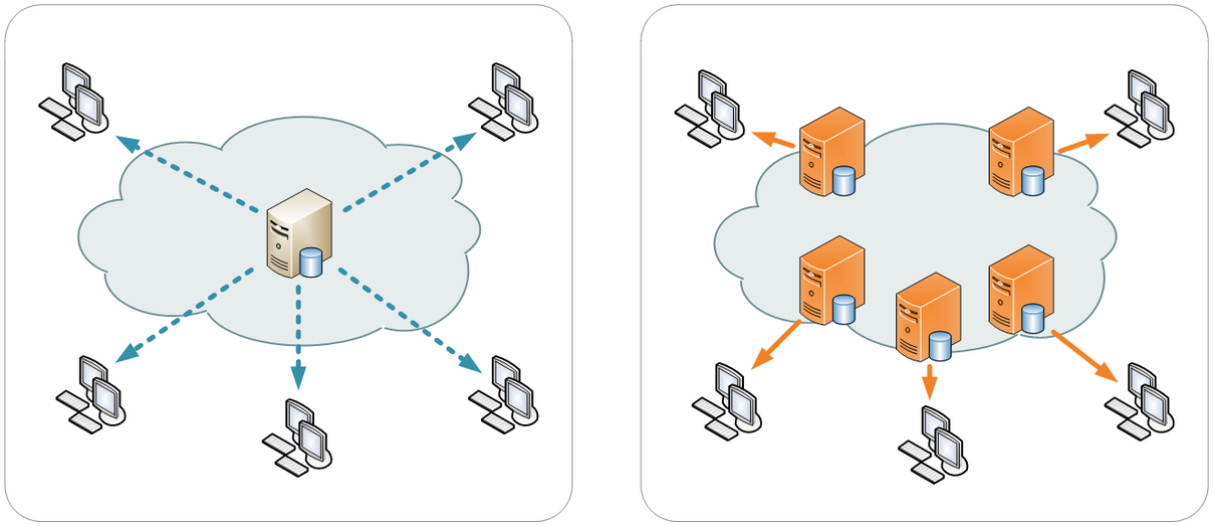
De gevonden edge computing oplossing is niet toepasbaar voor PostNL. Dit komt voornamelijk door het laatste nadeel. PostNL heeft een groot aantal klanten verspreid over de hele wereld. Voor haar is het nagenoeg niet mogelijk om voor elke klant een server op haar terrein te plaatsen. Naast dit nadeel zou het PostNL een hoop geld kosten om al deze servers te plaatsen en in te stellen dat alles onderling goed met elkaar kan communiceren.

Dat de afstand tussen de eindgebruiker en de server zo klein is had wel ideaal geweest voor PostNL. Hierdoor is de latency nagenoeg 0.

5.2 CDN

Internetgebruikers krijgen steeds meer bandbreedte door de steeds nieuwe technieken die worden ontwikkeld. De verwachting was dat gebruikers dezelfde data zouden opvragen door de vergroting in bandbreedte maar dan sneller. Dit bleek niet het geval. Door de vergroting van de snelheid gingen gebruikers steeds meer data opvragen. Denk hierbij aan video's in de eerdere jaren van Youtube met 240p werden opgevraagd worden ze nu in 4k opgevraagd. Hierdoor wordt het beschikbare internet overweldigd bij alle requests. Om deze reden ervaren gebruikers het vaak als onvoorspelbaar en onbevredigend.

Een handige methode om dit te verbeteren is het gebruiken van een Content Distribution Network (CDN) (Peng, 2018). Een CDN repliceert de originele inhoudt er plaatst een kopie hiervan op verschillende servers rond de wereld. Wanneer er een aanvraag naar de data wordt gedaan, wordt er gekeken welke server het dichtst bij de gebruiker staat. Vervolgens worden de bestanden vanaf die server gehaald. Dit verschil is de zien in figuur 5.2. Alle bestanden die vanaf een CDN geleverd worden moeten echter wel cachebaar zijn. Wat inhoudt dat ze 1 dag of 1 uur gecached mogen worden. Deze bestanden worden ook wel statische bestanden genoemd. Dit is het grote verschil te opzichte van cloud computing. Bij het gebruik van een CDN is computing van de data niet mogelijk.



Figuur 5.2 - CDN (NCDN - CDN, 2019)

5.2.1 Voor- en nadelen

Een CDN is een van de gevonden oplossingen die zijn ontdekt uit het onderzoek dat staat beschreven in de inleiding van dit hoofdstuk. In dit hoofdstuk worden eerste de voordelen beschreven die zijn onderzocht voor een CDN. Vervolgens worden ook de nadelen beschreven die een CDN met zich meebrengt.

Voordelen

Een CDN is een van de gevonden oplossingen die zijn ontdekt uit het onderzoek dat staat beschreven in de inleiding van dit hoofdstuk. In dit hoofdstuk worden de voordelen beschreven die zijn onderzocht voor een CDN.

Lagere latency

Net als bij Edge computing en back-end replicatie is de afstand tussen de afstand tussen de eindgebruiker en de server een stuk kleiner. Om deze reden duurt het minder lang voor de statische data bij de eindgebruiker is. Het lost het probleem op dat is beschreven in paragraaf 4.1.1 (Invloeden op latency).

Minder bandbreedte

Naast een lagere latency zorgt een CDN ook voor een lager bandbreedte verbruik. Uit onderzoek (Akamai, 2013) dat 65% van het internetverkeer bestaat uit afbeeldingen. Een CDN zorgt ervoor dat deze, samen met andere cachebare bestanden zoals javascript bestanden, op een slimme wijze worden gecached zodat ze niet steeds opgehaald moeten worden vanaf de server.

Nadelen

Naast de voordelen moeten ook de nadelen kritisch bekeken worden om een duidelijk beeld te krijgen. Deze nadelen worden hieronder beschreven.

Geen uitvoering van code

Het is niet mogelijk om code uit te voeren op een CDN. Zoals eerder beschreven levert een CDN alleen statische bestanden. Alle berekeningen die in de code van de statische bestanden staan zullen dus op het apparaat van de eindgebruiker worden uitgevoerd. Dit betekend dat je vooraf code moet genereren die per browser vervolgens een eigen ervaring aan de eindgebruiker levert (Keycdn, 2018). Hiermee wordt de connectie met de server in Europa tot een minimum beperkt.

Beschikbaarheid

Een ander groot nadeel bij het gebruiken van een CDN is de afhankelijkheid in beschikbaarheid van de CDN leverancier. Wanneer het CDN zelf down gaat is je website direct onbereikbaar. Dit is Cloudflare overkomen in 2013. Hierdoor waren 800.000 websites twee uur lang down.

5.2.2 Toepasbaar voor PostNL

Het gebruiken van een CDN lijkt een goede toepassing voor PostNL. De statische bestanden kunnen op deze manier snel geleverd kunnen worden aan de eindgebruikers. Vervolgens wordt de dynamische data opgehaald vanaf een centrale API welke het zware rekenwerk en de connectie met de database afhandelt. De bandbreedte is dan beperkt tot enkel de benodigde data en niet het hele scherm wordt gestuurd zoals met mvc wordt gedaan. Echter, om dit mogelijk te maken moet de software kunnen draaien op een static file hosting server (Keycdn, 2018) zoals beschreven bij de nadelen in paragraaf 5.2.2.

5.3.3 Static file hosting

Uit de onderzochte vakliteratuur, welke gevonden is voor de CDN oplossing, is gebleken dat de geschreven code moet kunnen draaien via een static file hosting aanbieder. Static file hosting, vertaald naar statische bestanden aanbieder, houdt in dat bestanden aangeboden worden aan een gebruiker zonder dat hier extra computaties voor nodig zijn (“Static Content Hosting pattern - Cloud Design Patterns”, 2020). Om te kijken of static file hosting een goede oplossing voor PostNL blijkt te zijn moet dit eerst verder onderzocht worden.

5.3 Back-end replicatie

Back-end replicatie wordt vaak gebruikt om de servers dichter bij de klant te plaatsen. Hierbij wordt dezelfde back-end geplaatst in verschillende continenten met ieder een eigen url. Op deze manier wordt de latency verlaagd doordat de afstand tussen de eindgebruiker en de server wordt verminderd. Het kan gezien worden als een zelf gehoste CDN, hierover meer uitleg in de volgende paragraaf, welke in tegenstelling tot een CDN wel computaties kan uitvoeren. Hierom wordt het vaak gezien als een tussenweg voor edge computing en een CDN.

Om deze techniek optimaal te laten werken wordt er ook een replicatie van de database geplaatst naast de back-end. Wanneer dit niet gedaan wordt moet de back-end nog steeds terug naar continent waar de database in staat.

5.3.1 Voor- en nadelen

Naast edge computing is ook back-end replicatie een gevonden oplossing die gebruikt kan worden om de latency te verlagen. In dit hoofdstuk worden als eerste de voordelen beschreven die zijn onderzocht voor back-end replicatie. Vervolgens worden ook de nadelen beschreven die back-end replicatie met zich meebrengt.

Voordelen

Lagere latency
Net als bij Edge computing is de afstand tussen de eindgebruiker en de server een stuk kleiner. Om deze reden duurt het minder lang voor de volledige website bij de eindgebruiker is. Het lost het probleem op dat is beschreven in paragraaf 4.1.1 (Invloeden op latency).

Beschikbaarheid
Een groot voordeel van het gebruiken van back-end replicatie is dat de beschikbaarheid hoger ligt. Doordat elke replicatie een eigen url heeft kan een andere server worden aangeroepen wanneer de server die normaal gebruikt wordt offline is. Op deze manier hoeft er geen downtime te zijn wanneer een server wordt geupdate.

Nadelen

Prijzig
Vergelijkbaar met edge computing worden er verschillende servers verspreid geplaatst. Per replicatie van de back-end samen met een database server en synchronisatie mechanisme zullen de kosten bijna x aantal keer hoger zijn voor het x aantal replicaties.

5.2.2 Toepasbaar voor PostNL

De gevonden back-end replicatie oplossing is niet toepasbaar voor PostNL. Dit komt voornamelijk door twee redenen. Technische beperkingen 5 en 6, beschreven in het requirements rapport. Technische beperking 5 zegt “Er worden geen aanpassingen gedaan aan de back-end.”. Daarbij staat in technische beperking 6 “Geografische replicatie van de back-end is geen optie.”. Naast deze beperkingen zou het PostNL een hoop geld kosten om de software om te zetten dat het onderling met elkaar kan synchroniseren en zouden de maandelijkse lasten voor het hosten van de software omhoog gaan.

5.4 Tussentijdse conclusie

Om antwoord te geven geven op de tweede deelvraag **“Welke bestaande methoden kunnen worden toegepast om het serveren van een webapplicatie te versnellen?”** is in eerst instantie deskresearch uitgevoerd. Uit dit onderzoek zijn drie mogelijke oplossingen gekomen om de performance te verbeteren, edge computing, back-end replicatie en het gebruiken van een CDN.

Zoals in paragraaf 5.1 is beschreven geeft edge computing wel de gewenste latency optimalisatie. Echter, is het niet toepasbaar voor PostNL. Dit komt voornamelijk door het laatste nadeel. PostNL heeft een groot aantal klanten verspreid over de hele wereld. Voor PostNL is het nagenoeg onmogelijk om voor elke klant een server op haar terrein te plaatsen. Naast dit nadeel zou het PostNL een hoop geld kosten om al deze servers te plaatsen en in te stellen dat alles onderling goed met elkaar kan communiceren.

Een tweede gevonden oplossing is om de back-end te repliceren in de verschillende continenten. Hierdoor komt de server dichter bij de eindgebruiker te staan. Echter, dit is geen optie voor PostNL doordat het niet voldoet aan technische beperkingen 5 en 6.

Een andere gevonden oplossing is het gebruiken van een CDN. Naar aanleiding van de gevonden vakliteratuur lijkt dit een geschikte oplossing voor het klantenportaal van PostNL. Een CDN biedt een aantal goede oplossingen voor het klantenportaal. Zo wordt de latency verlaagd omdat de statische bestanden dichter bij de klant staan. Het is een kleinere verlaging van de latency dan bij edge computing doordat de server nu niet op het terrein van de klant zelf staat. Hierdoor is de afstand dus groter dan bij edge computing. Een ander verschil tussen een CDN en edge computing is dat een CDN in de meeste gevallen een dienst is die je afneemt terwijl edge computing meestal zelf ingesteld moet worden.

Een CDN heeft echter ook een aantal nadelen. De grootste hiervan is dat er geen dynamische code op uitgevoerd kan worden. Dit betekent date software moet worden

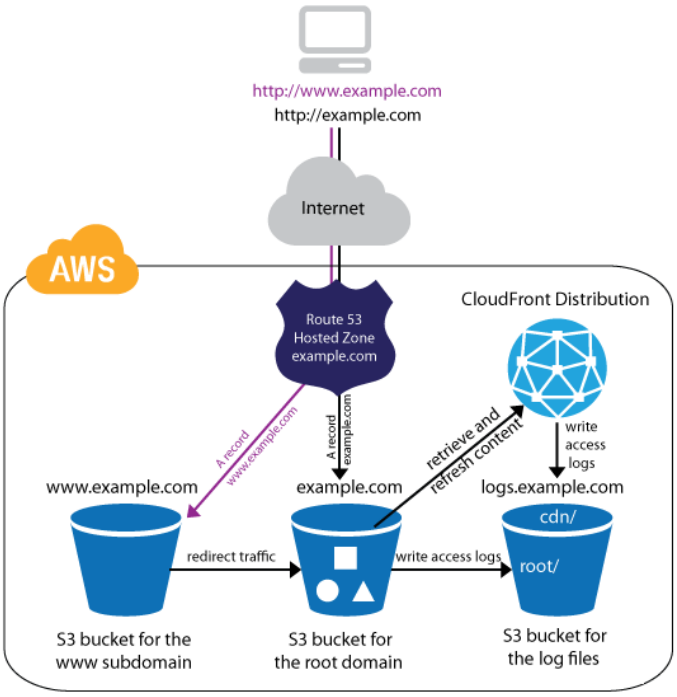
omgezet zodat het werkt vanaf een CDN. In het volgende hoofdstuk worden de manieren onderzocht om een website te hosten vanaf een static file hosting server.

6. Static file hosting

In hoofdstuk 5 is er antwoord gegeven op de tweede deelvraag “Welke bestaande methoden kunnen worden toegepast om het serveren van een webapplicatie te versnellen?”. Een van de oplossingen was het gebruiken van een CDN. Echter, om software beschikbaar te maken op een CDN moet deze kunnen draaien op een static file hosting server. Toen de eerste versie van het internet werd ontwikkeld bestonden webpagina’s alleen uit HTML en CSS. Voor deze websites werd er geen database of server code gebruikt. Daardoor kon het altijd draaien op een static file hosting server. In dit hoofdstuk wordt in de eerste paragraaf uitleg gegeven over wat static file hosting inhoudt. Vervolgens worden er verschillende manieren gegeven om dit mogelijk te maken.

6.1 Wat is static file hosting?

In hoofdstuk 5.3.4 is er al een korte beschrijving gegeven van static file hosting. In deze paragraaf staat er uitgebreidere beschrijving. Zoals in hoofdstuk 5.3.4 stond dat static file hosting inhoudt dat bestanden aangeboden worden aan een gebruiker zonder dat hier extra computaties voor nodig zijn (“Static Content Hosting pattern - Cloud Design Patterns”, 2020). Hierdoor kunnen deze files op een cloud opgeslag worden opgeslagen en vanaf daar worden opgehaald. De werking hiervan is beter te zien in figuur 6.1.



Figuur 6.1 - Static file hosting (Amazon, z.d.)

Statische sites worden als een belangrijk ingrediënt gezien van van de serverless technologieën. Bij de naam serverless treedt echter vaak verwarring op. Serverless houdt niet in dat de software zonder server kan draaien, wat het wel letterlijk vertaling ervan is. Bij serverless software kunnen de berekeningen in de cloud worden gedaan op een server van

de cloud provider. Hierdoor hoeft er geen eigen server aan toegewezen worden. De reden dat static file hosting als belangrijk ingrediënt van serverless wordt gezien van serverless, is dat de statische website bestanden in een cloud opslag kunnen staan en dat de dynamische berekeningen serverless in de cloud kunnen worden gedaan.

Echter, het is niet noodzakelijk om static file hosting te combineren met serverless functionaliteiten. De dynamische data kan ingeladen worden vanaf een externe API. In het geval van PostNL is dit de meest geschikte oplossing gezien zij al een centrale API hebben. In de volgende paragrafen worden er twee methoden besproken, static site generators en moderne front-end frameworks, die het mogelijk maken om statische bestanden te genereren en te combineren met de centrale API.

6.2 Static site generators

Zoals in de inleiding van dit hoofdstuk stond beschreven, bestonden de webpagina's aan het begin van het internet alleen uit HTML en CSS. Hierdoor kunnen die webpagina's gehost worden vanaf een static file hosting server. Vervolgens is het internet zich blijven ontwikkelen en zijn er dynamische webpagina's ontwikkeld. Deze dynamische websites zorgen daarentegen wel voor twee problemen. Ze kunnen niet goed geïndexeerd worden door zoekmachines omdat de content van de pagina kan veranderen terwijl de pagina's maar een keer worden geïndexeerd. Daarbij wordt bij elke aanvraag de data dynamisch ingeladen wat langer duurt dan het inladen van statische data.

Gaande tijd hebben verschillende personen gedacht aan een oplossing voor deze problemen. Uiteindelijk zijn hiervoor static site generators bedacht (Vardalos, 2017). Static site generators zijn de ideale oplossingen voor websites met dynamische content welke nagenoeg niet wijzigt. Een blog website gemaakt met behulp van een CMS is hier een goed voorbeeld van. Wanneer er elke week één nieuwe blog wordt geschreven kan de volledige site omgezet worden naar statische HTML / CSS. Deze kan vervolgens geplaatst worden op cloud storage om hem zo te laten repliceren via een CDN en hierdoor goedkoop te hosten.

Naast CMS static site generators bestaan er ook static site generators die JavaScript frameworks kunnen omzetten naar statische bestanden. Hierbij wordt een applicatie die geschreven is met een JavaScript framework omgezet naar statische HTML / CSS bestanden. Ook voor dit type static site generators geldt dat de de content van de website per gebruiker hetzelfde moet zijn. Hieronder staan een aantal static site generators voor beide types.

Jekyll
Een van de meest populaire static site generators is Jekyll. Dit komt mede doordat Jekyll de eerste static file generator was waarmee gebruikers konden werken. Jekyll is gemaakt met de programmeertaal Ruby en zit geïntegreerd in GitHub Pages (https://jekyllrb.com/). Naast dat het de eerste static site generator was is het mede om de integratie met GitHub pages erg populair geworden onder de developers. Het heeft een groot aantal gebruikers en is uitbreidbaar met vele plugins. Jekyll is een static site generator dat werkt als een CMS

Hugo
Tussen Hugo en Jekyll bevinden zich een groot aantal vergelijkingen. Hugo is gemaakt met de programmeertaal Go wat enorme snelheid met zich meebrengt. Dat is waar Hugo zich in onderscheid. Daarbij is het gebruiksvriendelijker en hoeft er minder geconfigureerd te worden om een eerste simpele website te genereren (https://gohugo.io/). Net als Jekyll is Hugo een static site generator dat werkt als een CMS.

Gatsby
Gatsby (https://www.gatsbyjs.org/) is gevonden in 2015 maar pas echt populair geworden sinds 2017. Gatsby maakt gebruik van React.js en zet gemakkelijk single page applications die geschreven zijn in React.js met statische content om naar statische HTML en CSS.

Scully
Scully (https://github.com/scullyio/scully/tree/master/schematics/scully) is een static site generator dat gebruikt maakt van Angular en zet Angular single page applicatie met statische content om naar statische HTML / CSS. Deze static site generator is nog erg nieuw en bevat hierdoor ook nog de benodigde kinderziektes.

6.2.1 Toepasbaar voor PostNL
Zoals eerder beschreven kan een static site generator gebruikt worden voor websites met een dynamische content die niet wijzigt. Helaas is het om deze reden niet mogelijk om deze techniek toe te passen op het klantenportaal van PostNL. Het klantenportaal van PostNL heeft teveel variabele welke afhankelijk zijn van de input die een gebruiker levert, waardoor er een ontzettend groot aantal pagina's gegenereerd moeten worden. Hierbij komen er door het gebruik van de applicatie ook steeds weer nieuwe pagina's bij, bijvoorbeeld door het genereren van barcodes. Nadat een barcode gegenereerd is zou de volledige website opnieuw gegenereerd moeten worden. Er zal dus een gezocht moeten worden naar een andere oplossing om er static files van te maken.

6.3 Statische bestanden cachen
Zoals in hoofdstuk 4 is beschreven, is er via Pingdom.com een test uitgevoerd om verschillende laadtijden met elkaar te vergelijken. Hieruit bleek dat er inderdaad een groot verschil bestond tussen de verschillende continenten. Deze testen lieten ook zien hoelang de verschillende bestanden er over deden om geladen te worden en hoeveel bandbreedte de bestanden in beslag namen. Hieruit bleek dat er een aantal javascript bestanden waren die er lang over deden om geladen te worden.

Om te controleren of het mogelijk is om alleen deze bestanden te laten cachen op een CDN is bij het ontwikkelteam nagevraagd hoe vaak de desbetreffende bestanden veranderen en of ze gecacht mogen worden. Uit deze gesprekken is gebleken dat de JavaScript gecachet mogen worden omdat deze bestanden eens in de twee weken gepubliceerd worden (Persoonlijke communicatie, 2020).

6.3.1 Toepasbaar voor PostNL

Zoals hierboven staat beschreven is het voor PostNL mogelijk om een deel van de bestanden te laten cachen op een CDN. Om de schermen aan de klanten te tonen moet het volledige scherm nog steeds opgebouwd worden op de centrale server. Hierdoor is er maar een klein deel opgelost en zal de verwachte performance optimalisatie minimaal zijn.

6.4 Moderne front-end frameworks

Een andere mogelijke oplossing is om een modern front-end framework te gebruiken voor het ontwikkelen van het klantenportaal. De meeste front-end frameworks kunnen gehost worden vanaf een static file hosting server. Moderne front-end frameworks creëren vaak single page applications, om deze reden ligt hier de focus op tijdens deze paragraaf. Een single page application is een website dat interacties met de webbrowser uitvoert door de webpagina dynamisch te herschrijven aan de hand van data van de server. Hierdoor hoeven er alleen delen van de website te worden herladen wat ervoor zorgt dat de gebruiker minder data binnen hoeft te halen. Het concept hiervan is in 2003 al geïntroduceerd (Galli, Soares, & Oeschger, 2003).

Het duurde hierna nog lang voordat er een efficiënte methode was gevonden om deze applicaties te programmeren en voor ze de gewenste snelheid hadden. In 2010 kwam angular uit met AngularJS (Angular, 2010), een JavaScript framework waarmee single page applications geschreven konden worden. Vanaf dat moment begonnen er steeds meer single page application op het internet te verschijnen en daarbij ook steeds meer JavaScript frameworks om deze single page applications te maken. Deze JavaScript frameworks maken gebruik van de JavaScript interpreters van de browsers. Om deze reden is de maximaal haalbare snelheid gelijk aan de snelheid van deze interpreters.

Tijdens de ontwikkelingen van de JavaScript frameworks bleven de browsers zich ook verder ontwikkelen. Een van deze ontwikkelingen is WebAssembly. WebAssembly, afgekort Wasm, staat programmeurs toe om in meer verschillende programmeertalen te programmeren die efficiënter gebruik maken van de resources op het systeem van de eindgebruiker. Daarbij staat Wasm het ook toe om applicaties te ontwikkelen die werken als een single page application. Alle berekeningen worden gedaan door Wasm en de aanpassingen worden doorgegeven aan het javascript in de browser.

6.4.1 JavaScript

JavaScript bestaat al sinds 1995 nog voordat het internet echt populair is geworden. Programmeurs wilde aanpassingen aan het DOM doorvoeren zonder dat de hele pagina opnieuw opgehaald zou moeten worden. Met behulp van JavaScript code kunnen webapplicaties ontwikkeld worden waarbij kleine DOM aanpassingen worden gedaan in tegenstelling tot het volledig herladen van de pagina zoals bij traditionele websites.

Deze techniek is zich door de jaren heen zo blijven ontwikkelen dat er op een gegeven moment JavaScript frameworks zijn ontwikkeld, een schil om JavaScript heen. Deze JavaScript frameworks staan het toe om single page applications te ontwikkelen. Hierbij hoeft de pagina nog maar eenmalig opgehaald te worden en wordt de rest van de content dynamische ingeladen. Na het ontstaan van AngularJS is het creëren van single page

applications steeds populairder geworden. Om deze reden zijn er hierna modernere JavaScript frameworks ontwikkeld die het programmeurs makkelijker maken om een applicatie te ontwikkelen.

Veelal maken deze frameworks gebruik van AJAX (Asynchronous JavaScript and XML (Garrett, 2005)) requesten om de dynamische data van de website in te laden. Vervolgens worden de resultaten verwerkt en waar nodig het DOM aangepast. Doordat er maar delen van het DOM worden aangepast is deze techniek erg populair geworden onder developers. Maar omdat iedereen zijn eigen mening heeft over hoe bepaalde software moet werken ontstonden er al snel een hoop verschillende JavaScript frameworks. In deze paragraaf worden alleen Angular en Vue.js behandeld gezien deze het meest gebruikt worden binnen Nalta. Andere alternatieve die niet worden uitgewerkt zijn bijvoorbeeld React.js en Ember.js.

Angular
Google is zoals hierboven beschreven begonnen met het ontwikkelen van AngularJS. Na vier jaar heeft het angular team echter besloten dat de initiële aanpak toch niet de manier was waarop zij verder wilde werken. Om deze reden hebben ze in 2014 Angular uitgebracht, maar gezien dit nog steeds een JavaScript framework is leverde het de benodigde verwarring op. Vervolgens heeft Google besloten de vernieuwde versie Angular 2 te noemen (https://angular.io/). Dit is ook de variant waar Nalta mee ontwikkeld (Interne communicatie, 2020). Angular 2, en de daarop volgende versies (verder naar gerefereerd als Angular), zijn gebaseerd op Typescript (een uitbreiding op JavaScript).

De reden dat Nalta heeft gekozen voor Angular is omdat ze een aantal jaar geleden de ontwikkelingen in de front-end wilde blijven volgen. Op dat moment waren er drie frameworks het meest prominent: Angular, React en Vue.js. De eerste twee worden door grote partijen ondersteund Angular door Google en React door Facebook terwijl Vue.js wordt ondersteund door een community. Hierom heeft Nalta toen gekozen om nog even te wachten met Vue.js tot dat wat verder ontwikkeld was. Hierna was het om het even en heeft Nalta ervoor gekozen om tijd te besteden aan het leren van Angular om hier vervolgens applicaties mee te ontwikkelen.

Vue.js
Vue.js (https://vuejs.org/) is als eerste ontwikkeld door Evan You, oud medewerker van google waar hij werkte met Angular (Hermans, 2018). Hij vond dat Angular te veel functies met zich meebracht die veelal niet gebruikt werden en wilde een snel en licht front-end framework ontwikkelen. Om deze reden is het ook anders opgebouwd dan Angular. Het is van zichzelf een erg klein framework en kan in elk web project worden toegevoegd. Hierdoor heeft het minder standaard bouwblokken die worden meegeleverd en moet je die zelf toevoegen of ontwikkelen. Dit kan bijvoorbeeld handig zijn wanneer er behoefte is om een aantal pagina's van een website in vue te schrijven. Naast deze optie is het ook mogelijk om de volledige front-end te schrijven in vue.

Na een aantal jaar ervaring te hebben met Angular vond ook Nalta dat het gebruik van een framework als angular niet altijd even praktisch is. Vue heeft zich in de tijd door blijven ontwikkelen en is een erg betrouwbaar framework geworden. Om deze reden heeft Nalta besloten ook tijd en energie te steken in het leren van Vue.js. Ondertussen worden beide varianten binnen de projecten van Nalta gebruikt. Zo is een klein gedeelte van de PostNL

applicatie in vue geschreven om de complexiteit in te perken. Nalta heeft voor vue gekozen omdat de leercurve van vue erg laag ligt zodat ook onervaren ontwikkelaars de code makkelijk kunnen begrijpen.

6.4.2 WebAssembly

Zoals beschreven in paragraaf 6.3 is het web zich in de laatste jaren blijven ontwikkelen. Een van deze ontwikkelingen is WebAssembly (Wasm). Wasm staat programmeurs toe om in een verschillende programmeertalen te ontwikkelen welke vervolgens efficiënter gebruik maken van de resources op het systeem van de eindgebruiker. In april 2015 (WebAssembly, 2015) is de WebAssembly Community Group begonnen met het ontwikkelen van Wasm en in juni datzelfde jaar hebben ze deze ontwikkelingen publiekelijk gemaakt. In 2017 werd WebAssembly gepubliceerd voor het gehele web. Ondertussen wordt dit door alle moderne browsers ondersteund. Hier hoort Internet Explorer niet bij.

Microsoft heeft toegegeven dat de aanpak van Internet Explorer niet de juiste aanpak was en dat ze hierdoor vanaf het begin af aan altijd technical debt opgebouwde (Microsoft, 2019). Onder andere om deze reden heeft Microsoft aangekondigd Internet Explorer niet verder te ontwikkelen en dat het volledige support eindigt op 14 oktober 2025 (Microsoft, 2020).

Een ander bijkomend voordeel bij het gebruik van WebAssembly is dat het de kosten van het hosten ook erg kan inperken. De Wasm code kan gehost worden op een static file hosting server (Latham, Stropek, & Roth, 2020) wat een stuk goedkoper is dan de huidige MVC hosting. De code wordt tijdens het bouwen gecompileerd naar statische bestanden. In paragraaf 5.3 ‘CDN’ staat beschreven dat deze statische bestanden vervolgens gerepliceerd kunnen worden met behulp van een CDN waardoor ze dicht bij de eindgebruiker komen te staan. Daarbij zijn de bestanden daardoor voor elke gebruiker hetzelfde en staan deze bestanden los van de rechten van de desbetreffende gebruiker.

Rust

Rust (https://www.rust-lang.org/) is een ontzettend snelle programmeertaal. De syntax van Rust is te vergelijken met die van C++. Rust is gestart door Graydon Hoare (Cassel, 2019) en werd in 2010 voor het eerst gepubliceerd. Na deze initiële publicatie is Rust steeds populairder geworden door de snelheid dat het levert in combinatie met veiligheidsgaranties die de compiler levert op het alloceren en gebruik van geheugen. Hiermee worden veel voorkomende memory bugs van C / C++ voorkomen. Een applicatie kan snel vanaf de command line worden opgezet en vervolgens gecompileerd worden naar verschillende soorten machine code. Voor de ontwikkelingen van WebAssembly werd de rust code vooral omgezet naar assembly code om op een laag level code te draaien. Hierna heeft Rust grote stappen gemaakt binnen de WebAssembly wereld en kan de Rust code hier nu ook naar gecompileerd worden.

Blazor

Blazor (https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor) is een nieuwe ontwikkeling van Microsoft. Blazor maakt het mogelijk om een single page application te schrijven in C#. Hiervoor bieden zij twee opties, Blazor server side en Blazor WebAssembly.

Bij een project dat Blazor server side gebruikt wordt er een open connectie gemaakt tussen de server en de browser van de eindgebruiker, ook wel de client genoemd, waarover alle DOM aanpassingen naar de client worden gestuurd. De zware berekeningen worden hierbij nog wel op de server uitgevoerd.

Zoals hierboven beschreven heeft Blazor ook een WebAssembly variant, Blazor WebAssembly. Blazor Wasm staat het toe om geschreven C# code te laten uitvoeren in de browser met behulp van WebAssembly. In tegenstelling tot Blazor server side worden nu de berekeningen uitgevoerd binnen de browser van de eindgebruiker. Hierdoor is het ontzettend snel en worden alleen de DOM aanpassingen door middel van JavaScript uitgevoerd. Blazor WebAssembly wordt in tegenstelling tot Rust niet gecompileerd naar WebAssembly code, alleen de .NET runtime is geschreven in WebAssembly terwijl de standaard .NET dll's hierdoor worden uitgevoerd. Deze standaard dll's zijn niet anders dan voor een .NET Core applicatie.

6.4.3 Toepasbaar voor PostNL

Het gebruik van een modern front-end framework voor het ontwikkelen van het klantenportaal van PostNL lijkt een goede oplossing te zijn. Dit geldt voor zowel de gevonden JavaScript als WebAssembly oplossingen. Nalta heeft in het verleden al eens single page applicaties ontwikkeld met JavaScript frameworks Angular en Vue.js. De andere gevonden oplossingen is door gebruik te maken van WebAssembly.

De twee besproken WebAssembly oplossingen zijn Rust en Blazor.
Rust is een efficiënte programmeertaal die gecompileerd kan worden naar Wasm code waardoor het snel intensieve berekeningen op de browser van de eindgebruiker uitvoeren. Het is voor Nalta zeker mogelijk om het klantenportaal om te zetten naar Rust echter heeft geen enkele ontwikkelaar binnen het Natla ontwikkelteam hier ervaring mee en staat dit ook niet op de planning (Interne communicatie, 2020).

Nadat deze informatie bekend is geworden blijkt Blazor een uitstekende oplossing te zijn voor het klantenportaal. Blazor WebAssembly biedt net als Rust de optie op ontzettend snel intensieve berekeningen op de browser van de eindgebruiker uitvoeren.
Zoals eerder aangegeven is de implementatie wel anders, bij Blazor WebAssembly is alleen de .NET runtime is geschreven in WebAssembly terwijl de standaard .NET dll's hierdoor worden uitgevoerd. Deze dll's zijn geschreven in C#, een wens van het development team, en alle kunnen files omgezet worden naar statische files die gehost kunnen worden op een static file hosting server.

6.5 Tussentijdse conclusie

Uit de tweede deelvraag is gebleken dat het gebruik van een CDN een geschikte oplossing lijkt te zijn om de performance van het klantenportaal te verbeteren. Echter, om een website via een CDN beschikbaar te maken moet de website gehost kunnen worden via een static file hosting server. In dit hoofdstuk zijn de verschillende manieren onderzocht om een website te kunnen hosten door middel van static file hosting.

De eerste gevonden oplossing is om gebruik te maken van een static site generator. Een static site generator genereert statische HTML en CSS files aan de hand van een layout met content. Dit is een ideale oplossing voor websites waarbij de inhoud hetzelfde blijft per gebruiker, een voorbeeld hiervan is een blog. Echter, wanneer er per gebruiker gecontroleerd moet worden of inhoud wel of niet zichtbaar moet zijn is dit geen optie. Deze conditionele rendering van de content kan alleen gedaan worden met dynamische data. De gevonden oplossing is dus geen optie om te gebruiken voor het klantenportaal van PostNL.

De tweede gevonden oplossing is het laten cachen van de statische bestanden van het klantenportaal. Het klantenportaal bevat een aantal grote JavaScript bestanden die mogelijk gecachet kunnen worden zodat ze dichter bij de klant komen te staan. Echter, de schermen moeten nog steeds worden opgebouwd op de centrale server in Europa. Door het cachen van deze JavaScript bestanden is er maar een klein deel opgelost en zal de verwachte performance optimalisatie minimaal zijn. Het ontwikkelteam heeft hierom ook aangegeven dat zij liever een oplossing zien waarbij de volledige applicatie op een CDN gehost kan worden.

Een andere gevonden oplossing is door gebruik te maken van moderne front-end frameworks. Met name omdat de meeste hiervan het toestaan om een single page application te ontwikkelen. Single page applications kunnen ook gehost worden vanaf een static file hosting server. In paragraaf 6.3 zijn twee verschillende manieren onderzocht die gebruikt kunnen worden voor het ontwikkelen van applicaties met moderne front-end frameworks, namelijk door het gebruik van JavaScript frameworks of met behulp van WebAssembly. Nalta heeft in het verleden al eens gebruik gemaakt van de JavaScript frameworks Angular en Vue.js, deze kunnen hierom beide gebruikt worden om het klantenportaal om te zetten naar static files.

Echter, Nalta heeft laten weten dat zij zeer geïnteresseerd zijn in de techniek van WebAssembly. WebAssembly lijkt een net zo dan wel niet meer geschikte oplossing te zijn, WebAssembly is ontzettend snel doordat de berekeningen worden uitgevoerd met nagenoeg maximale browser snelheid. Rust en Blazor zijn twee technieken die al ver ontwikkeld zijn en gebruik maken van WebAssembly. Beide technieken zijn in paragraaf 6.3.2 besproken en lijken mogelijke oplossingen te zijn voor het performance probleem. Ervaring krijgen met Rust en het ontwikkelen van applicaties hierin staat op het moment nog niet op de roadmap van Nalta en lijkt hierdoor minder gepaste oplossing voor PostNL. Blazor daar en tegen lijkt wel een goede kandidaat te zijn voor PostNL.

7. Conclusie

Het doel van het onderzoek is om de oorzaak van het performance probleem van het klantenportaal te vinden en hiervoor mogelijke oplossingen te zoeken. Om hierachter te komen was de volgende centrale hoofdvraag opgesteld:

“Hoe kan Nalta de performance van het klantenportaal van PostNL in regio’s buiten Europa significant verbeteren?”

In hoofdstuk 4 is de eerste deelvraag onderzocht, waar het performance probleem van het klantenportaal vandaan komt. Hieruit is gebleken dat het klantenportaal kampt met een latency probleem en dan voornamelijk de afstand tussen de server en de eindgebruiker. In het volgende hoofdstuk is onderzocht hoe deze afstand verminderd kan worden.

In hoofdstuk 5 is de tweede deelvraag beantwoord. Hierin zijn er oplossingen gezocht om de afstand tussen de server en de eindgebruiker te verminderen. Verschillende oplossingen vielen af door vooraf opgestelde technische beperkingen of door de complexiteit van de infrastructuur drastisch te veranderen. Gebruik maken van een CDN lijkt wel een geschikte oplossing. Hiervoor moet een website echter gehost kunnen vanaf een static file hosting server. Hoe dit gedaan kan worden staat uitgelegd in hoofdstuk 6.

In hoofdstuk 6 wordt static file hosting besproken. Hieruit bleek dat static site generators niet gebruikt kunnen worden voor het klantenportaal door de complexiteit van het klantenportaal. Moderne front-end frameworks kunnen wel goed gebruikt worden voor het klantenportaal. Door het creëren van een single page application kan de volledige website gehost worden op een static file hosting server en zo mogelijk de performance te verbeteren.

Uit dit onderzoek volgt een hypothese die vervolgens bewezen moet worden aan de hand van een prototype, de hypothese luidt: Door middel van een modern front-end framework, in combinatie met een CDN, wordt de performance van het klantenportaal van PostNL in regio’s buiten Europa verbeterd.

8. Aanbeveling

In dit hoofdstuk wordt een aanbeveling gedaan die ervoor kan zorgen dat de performance van het klantenportaal van PostNL wordt verbeterd.

Zoals de conclusie heeft uitgewezen komt het performance probleem van het klantenportaal voornamelijk door de afstand tussen de server en de eindgebruiker. Het gebruik van een CDN lijkt een geschikte oplossing om dit probleem op te lossen. Echter, om een website vanaf een CDN beschikbaar te maken moet de code gehost kunnen worden op een static file hosting server. Het omzetten van het klantenportaal naar een single page application lijkt in deze situatie een geschikte oplossingen.

Zowel JavaScript frameworks als WebAssembly zijn geschikte oplossingen om single page applications te schrijven. Tijdens interne vergaderingen heeft Nalta laten weten dat ze de WebAssembly technieken erg interessant vinden en deze graag op de voet blijven volgen. Ze hebben desalniettemin nog geen project of prototype ontwikkeld met behulp van een WebAssembly techniek. Hierom past deze methode goed in deze situatie.

In paragraaf 6.3.2 staat beschreven dat WebAssembly niet meer ondersteund wordt in Internet Explorer. Dit kan een mogelijke beperking zijn voor het prototype en advies. Om deze reden is er een korte meeting aangevraagd om deze mogelijke belemmering te bespreken. Uit deze bespreking is duidelijk geworden dat dit geen beperking is. Een eerste argument dat hierbij is gegeven is dat Microsoft heeft aangekondigd Internet Explorer niet meer te laten doorontwikkelen. Daarbij neemt Nalta hierbij een voorbeeld aan grote bedrijven als GitHub, YouTube en Spotify welke ook hebben besloten Internet Explorer niet meer te ondersteunen.

Zoals in paragraaf 6.4 is beschreven lijken zowel Rust als Blazor zeer geschikte WebAssembly oplossingen te zijn voor het performance probleem van het klantenportaal. Het Nalta development team van het klantenportaal heeft aangegeven dat zij de front-end het liefst ook in C# willen ontwikkelen. Nalta is een Microsoft georiënteerd bedrijf en alle back-end applicaties worden reeds geschreven in C#. Wanneer de front-end ook geschreven is in C# kunnen bepaalde modellen en functionaliteiten hergebruikt worden en hoeven ze maar eenmalig geschreven te worden.

Nadat deze informatie bekend is geworden blijkt Blazor een uitstekende oplossing te zijn voor het klantenportaal. Blazor WebAssembly biedt net als Rust de optie om ontzettend snelle intensieve berekeningen op de browser van de eindgebruiker uitvoeren. Blazor staat het developers toe om in C# te programmeren en kunnen alle files omgezet worden naar statische files die gehost kunnen worden op een static file hosting server. Om deze reden wordt op basis van de hypothese aanbevolen om Blazor als techniek te gebruiken voor het prototype.

Literatuurlijst

Akamai. (2013, 18 november).
Extreme Image Optimization: WebP & JPEG XR in Aqua Ion - The Akamai Blog. Geraadpleegd op 17 maart 2020, van https://blogs.akamai.com/2013/11/extreme-image-optimization-webp-jpeg-xr-in-aqUa-ion.html

Amazon. (z.d.). *Static file hosting* [Foto].
Geraadpleegd van https://d1.awsstatic.com/Projects/v1/AWS_StaticWebsiteHosting_ArchitEcture_4b.d
a7f28eb4f76da574c98a8b2898af8f5d3150e48.png

Angular. (2010, 21 september).
Angular/angular.js. Geraadpleegd op 6 maart 2020, van https://github.com/angular/angular.js/releases?after=v0.9.4

Cassel, D. (2019, 17 juni).
Rust Creator Graydon Hoare Talks About Security, History, and Rust. Geraadpleegd op
17 maart 2020, van https://thenewstack.io/rust-creator-graydon-hoare-talks-about-security-history-and-r
ust/

Edge computing infrastructure. (2019). [Foto].
Geraadpleegd van https://upload.wikimedia.org/wikipedia/commons/b/bf/Edge_computing_infrastruct
ure.png

Galli, M., Soares, R., & Oeschger, I. (2003).
Inner-browsing extending the browser navigation paradigm. Geraadpleegd van https://developer.mozilla.org/en-US/docs/Archive/Inner-browsing_extending_the_br
owser_navigation_paradigm

Garrett, J. J. (2005).
Ajax: A New Approach to Web Applications. Geraadpleegd van https://www.scriptol.fr/ajax/ajax_adaptive_path.pdf

Ground Control. (z.d.).
Low Satellite Internet Latency. Geraadpleegd op 21 februari 2020, van https://www.groundcontrol.com/Satellite_Low_Latency.htm

Gyarmathy, K. (2019, 4 oktober).
The Benefits and Potential of Edge Computing. Geraadpleegd op 12 maart 2020, van https://www.vxchnge.com/blog/the-5-best-benefits-of-edge-computing

Hermans, L. (2018, 6 februari).
Vue on 2018 — Interview with Evan You, author of the Vue.js framework. Geraadpleegd op 17 maart 2020, van https://blog.hackages.io/https-blog-hackages-io-evanyoubhack2017-cc5559806157

Arabi, K. (2014, 4 juni).
IEEE DAC 2014 Keynote: Mobile Computing Opportunities, Challenges and Technology Drivers
http://www2.dac.com/events/videoarchive.aspx?confid=170&filter=keynote&id=170
-103--0&#video

Hunter, S. (2019, 9 mei).
.NET Core is the Future of .NET . Geraadpleegd op 19 februari 2020, van https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/

Keycdn. (2018, 4 oktober).
Static Site Hosting With a CDN - KeyCDN Support. Geraadpleegd op 3 maart 2020, van
https://www.keycdn.com/support/static-site-hosting-with-a-cdn

Latham, L., Stropek, R., & Roth, D. (2020, 19 februari).
Host and deploy ASP.NET Core Blazor WebAssembly. Geraadpleegd op 10 maart 2020,
van
https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/blazor/webassembly?view=aspnetcore-3.1

Microsoft. (2019, 28 november).
The perils of using Internet Explorer as your default browser. Geraadpleegd op 10 maart 2020, van
https://techcommunity.microsoft.com/t5/windows-it-pro-blog/the-perils-of-using-in

ternet-explorer-as-your-default-browser/ba-p/331732

Microsoft. (2020, 29 januari).
Windows lifecycle fact sheet. Geraadpleegd op 10 maart 2020, van https://support.microsoft.com/en-us/help/13853/windows-lifecycle-fact-sheet

NCDN - CDN. (2019). [Foto].
Geraadpleegd van https://upload.wikimedia.org/wikipedia/commons/thumb/f/f9/NCDN_-_CDN.png/1200px-NCDN_-_CDN.png

Noction. (2015, 7 augustus).
Network latency and its effect on application performance. Geraadpleegd op 25 februari 2020, van https://www.noction.com/blog/network-latency-effect-on-application-performance

Peng, G. (2018).
CDN: Content Distribution Network. Geraadpleegd van https://arxiv.org/pdf/cs/0411069.pdf

Plug Things In. (z.d.).
What is Latency - How is Latency Different from Bandwidth.
Geraadpleegd op 21 februari 2020, van http://www.plugthingsin.com/internet/speed/latency/

Static Content Hosting pattern - Cloud Design Patterns. (2020, 24 februari).
Geraadpleegd op 3 maart 2020, van https://docs.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting

The Linux Information Project. (2005 september).
Latency Definition. Geraadpleegd op 20 februari 2020, van http://www.linfo.org/latency.html

ThinkwithGoogle. (2019, 15 januari).
Mobile Page Load 02. Geraadpleegd op 5 maart 2020, van https://www.thinkwithgoogle.com/data/page-load-time-statistics/

Vardalos, S. (2017, 19 augustus).
The best static website generators, and when you should choose them over a CMS.
Geraadpleegd op 4 maart 2020, van https://www.freecodecamp.org/news/static-sites-are-back-24d01a01f11a/

WebAssembly. (2015 april).
Roadmap - WebAssembly. Geraadpleegd op 10 maart 2020, van https://webassembly.org/roadmap/

WikiBooks. (z.d.).
Datacommunicatie in informatica/Inleiding - Wikibooks. Geraadpleegd op 25 februari 2020, van https://nl.wikibooks.org/wiki/Datacommunicatie_in_informatica/Inleiding

Testverslag

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
17-07-2020
Versie 2.0

Inhoudsopgave

1. Inleiding

Testen is en blijft een belangrijke stap in het ontwikkelingsproces van nieuwe websites en software. Software dient goed getest te worden voor het in gebruik genomen wordt om de kans van fouten in de software te verminderen. Er zijn veel verschillende manieren om geschreven software te testen. In dit document ga ik in op de verschillende manieren die gebruikt zijn voor het testen van het ontwikkelde prototype.

2. Laadsnelheid

In dit hoofdstuk worden er verschillende manieren van laadsnelheden getest. Als eerste is de test uit het ‘huidige situatie’ document uitgebreid. Vervolgens is Blazor tegen de .NET Core variant opgezet. Ook is er een vergelijking gemaakt met JavaScript.

2.1 Verschillende continenten

In het ‘huidige situatie’ document staat er een test beschreven die is uitgevoerd door middel van Pingdom. Uit deze test bleek dat het ophalen van de website vanuit buiten Europa dermate langer duurt dan wanneer dezelfde website binnen Europa wordt opgehaald. De resultaten van gemiddelden van deze test staan nogmaals weergeven in tabel 2.1.

Regio	.NET Framework	.NET Core
Europa, Duitsland, Frankfurt	503 ms	**489 ms**
Europa, Engeland, Londen	506 ms	498 ms
Noord-Amerika, USA, San Francisco	3.02 s	2.76 s
Azië, Japan, Tokio	5.19 s	4.28 s
Oceanië, Australië, Sydney	**6.41 s**	5.21 s

Tabel 2.1 - Initiële Laadsnelheid

Na het uitvoeren van het onderzoek is er aanbevolen om een prototype te ontwikkelen met behulp van het moderne front-end framework Blazor. Na het ontwikkelen van het prototype in blazor kon deze geplaatst worden op een static file server. Na het plaatsen van het prototype op deze server kon ook hier de test via pingdom op uitgevoerd worden. De resultaten van de gemiddelden van deze test zijn te zien in tabel 2.2 kolom ‘Blazor’. Opmerkelijk is dat het binnen Europa iets langer duurt om de website op te halen dan met .NET Framework en .NET Core. Dit is mogelijk te verklaren doordat het blazor framework nu ook gedownload wordt op de client wat groter is dan de normale versie. Wat ook opvalt is dat de website sneller laadt buiten Europa. Dit is te verklaren doordat Blazor WebAssembly direct snelle computaties kan uitvoeren bij de klant.

Regio	.NET Framework	.NET Core	Blazor	Blazor + CDN
Europa, Duitsland, Frankfurt	503 ms	489 ms	510 ms	**158 ms**
Europa, Engeland, Londen	506 ms	498 ms	540 ms	174 ms
Noord-Amerika, USA, San Francisco	3.02 s	2.76 s	2.65 s	218 ms
Azië, Japan, Tokio	5.19 s	4.28 s	3.87 s	264 ms
Oceanië, Australië, Sydney	**6.41 s**	5.21 s	3.21 s	294 ms

Tabel 2.2 - Laadsnelheid

Om de gevonden hypothese te testen is er CDN geplaatst voor de static file hosting server. Opnieuw zijn er met behulp van Pingdom een aantal testen uitgevoerd waarvan de gemiddelden te zien zijn in tabel 2.2 kolom ‘Blazor + CDN’. Voordat deze testen zijn uitgevoerd is er eerst een warm-up run uitgevoerd waardoor de website gecached kon worden op het CDN. Daarna kon de volledige website direct van het CDN gehaald worden. De resultaten laten duidelijk zien dat er een grote snelheids verbetering plaatsvindt door deze oplossing te gebruiken. Dat het buiten Europa nog steeds langer duurt voordat de website is geladen is te verklaren doordat er een klein aantal dynamische assets zijn die niet gecached konden worden en dus alsnog via Europa opgehaald moesten worden. Desalniettemin is een laadtijd onder de 500 ms alsnog sneller dan de huidige versie die vanuit binnen Europa wordt opgehaald.

2.2 Blazor langzamer dan .NET Core

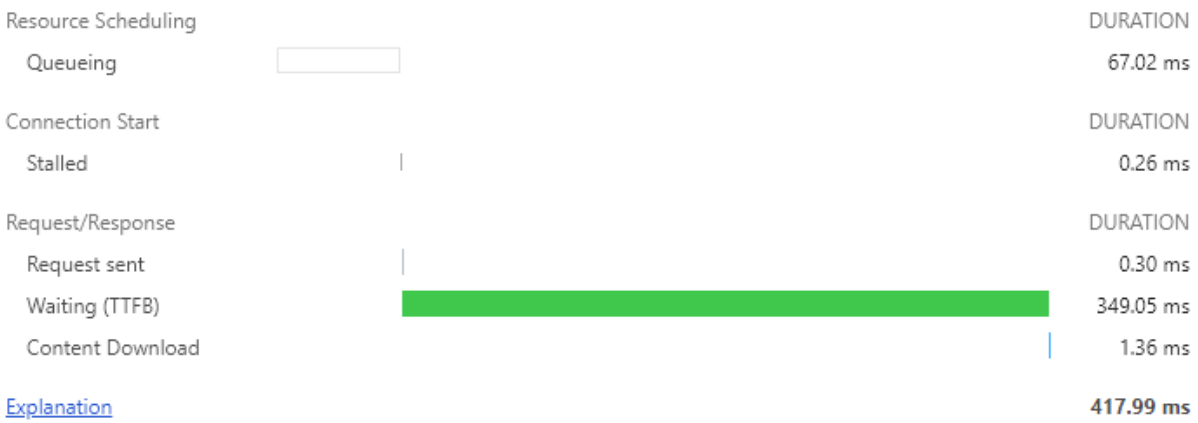
Na het uitvoeren van de Pingdom testen zijn er een aantal verschillende pagina’s in de verschillende versies met elkaar vergeleken. Op de meeste pagina’s was de Blazor variant sneller geladen, wat aan de verwachtingen voldeed, maar niet op elke pagina. Om erachter te komen wat dit verschil veroorzaakt diende er meer onderzoek gedaan te worden.

Om te testen waar het probleem vandaan kwam, is er gebruik gemaakt van de stopwatch class van .NET met behulp van de volgende code:

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
codeToRun
Console.WriteLine("Milliseconds ellapsed:" +
stopWatch.Elapsed.TotalMilliseconds);
```

De stopwatch staat het toe om te timen hoe lang verschillende methoden bezig zijn. Om uit te sluiten hoelang het duurt voor de applicatie data ontvangt vanaf de API is er gebruik gemaakt van de Chrome Devtools.

De network tab is gebruikt om te na te gaan hoelang het duurt voordat de browser de data vanuit de API terugkrijgt. Dit is terug te zien in figuur 2.1. Tijdens het ‘Queueing’ gedeelte zijn er andere taken die een hogere prioriteit hebben. Deze tijd wordt dan ook niet meegeteld met de laadtijd van de data. Het ophalen van de data duurt in beide projecten even lang. Dit komt doordat dezelfde API wordt gebruikt en de afstand even groot is.



Figuur 2.1 - API request

Figuur 2.1 toont aan dat het request naar de api 350 ms duurt. Dit is gebaseerd op een test die tienvoudig is uitgevoerd waarvan het gemiddelde is genomen. Het ophalen van de data werd met de stopwatch als 400 ms weergeven wat laat zien dat Blazor met het request zelf ongeveer 50 ms bezig. Het response wordt vervolgens omgezet naar een byte array wat 1-3 ms duurt. In de laatste stap wordt deze byte array door middel van de “System.Text.Json” serializer omgezet naar een model, wat standaard is met het gebruik van C# en API’s. Dit omzetten duurde ongeveer 150-250 ms. Hier zit dus de bottleneck. Een groot deel van het laden van de pagina is Blazor bezig met het omzetten van het response naar het model.

Dezelfde testen zijn gedaan voor de .NET Core versie. Hieruit bleek dat het omzetten van hetzelfde response daar binnen 0.72 ms is afgehandeld. Dit is meer dan 250 keer sneller dan wat bij Blazor het geval is. Uit de GitHub issues van Blazor is gebleken dat ze inderdaad een performance probleem hebben bij het deserializen van grote JSON (JavaScript Object Notation) objecten en dat er wordt gezocht naar een oplossing. Voor dit prototype is onderzocht of andere JSON deserializers gebruikt konden worden. Hieruit volgde een aantal verschillende libraries; Newtonsoft.Json, Utf8Json, Jil en de al eerder gebruikte System.Text.Json.

Eenzelfde test is met deze verschillende libraries waarvan de resultaten in tabel 2.3 te zien zijn.

Tijdens het uitvoeren van deze test is de eerste laadtijd meegenomen en het gemiddelde., welke tussen de haakjes staat. In de Blazor resultaten staat de eerste laadtijd tussen haakjes en het gemiddelde van de daaropvolgende daarvoor

	System.Text.Json	Newtonsoft.Json	Utf8Json	Jil
Blazor	113 (196)	63 (334)	10 (260)	12 (2725)
.NET Core	0.68 (478)	0.92 (483)	0.14 (317)	0.25 (882)

Tabel 2.3 - JSON vergelijking in millisecondes

Tijdens het testen leek er een groot verschil te zitten tussen de eerste keer dat het model werd omgezet en de daaropvolgende keren. Dit is te verklaren doordat Blazor en .NET Core

gebruik maken van JIT (Just-In-Time) compilatie, hierbij wordt de code pas gecompileerd wanneer het gebruikt gaat worden. Wanneer de models tijdens het laden eenmalig worden gecompileerd kan applicatie deze manier onthouden. De eerste keer dat een JSON object wordt deserialized was “System.Text.Json” het snelste, maar in alle daarop volgende deserialisaties was Utf8Json het snelste. Tijdens het opstarten van Blazor kan een dergelijk model al een keer in de achtergrond gedeserialiseerd worden waardoor altijd de snelle variant wordt gebruikt, waarin Utf8Json ruim elf keer sneller is dan “System.Text.Json”. Wel is Blazor zelfs met Utf8Json langzamer in het deserialiseren dan .NET Core.

Gezien de resultaten van de test is Utf8Json de eerte keer minder snel maar daarna een stuk sneller, doordat de verschillende pagina’s meerdere keren per gebruiker worden opgehaald wordt Utf8Json in het prototype. De resultaten van deze test hebben geen invloed op de test die met Pingdom is uitgevoerd gezien daar geen JSON data wordt omgezet.

Andere mogelijke oplossingen die volgde uit het onderzoek waren Vue.js en Angluar, welke beide gebruik maken van JavaScript. Om de vergelijking compleet te maken is ook in JavaScript de performance van het deserializen van het JSON object getest. Hiervoor is onderstaande code gebruikt:

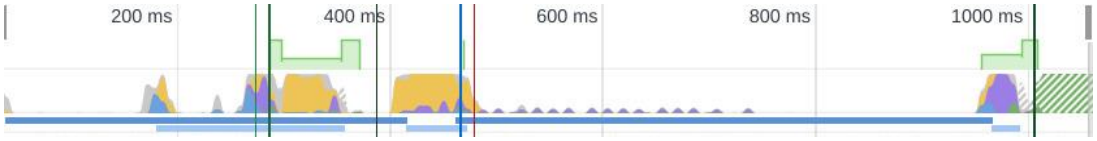
```
var startTime = performance.now();

var parsedObject = JSON.parse('{ Json object }');

var endTime = performance.now();
console.log("Parsing took " + (endTime - startTime) + " milliseconds.");
```

Hieruit bleek dat het in javascript ongeveer 0.45 milliseconde duurde. Dit is ongeveer drie keer langzamer dan .NET Core maar sneller dan Blazor. Door gebrek aan tijd kan geen volledig prototype uitgewerkt worden in JavaScript waardoor er geen volledige vergelijking gemaakt kan worden.

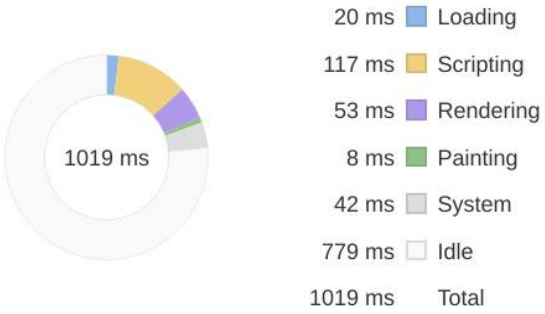
2.3 Paginas

Een pagina die veelvuldig aangevraagd wordt door de zakelijke klanten is de ‘manifest overview’ pagina. Om deze reden heb ik deze pagina in de verschillende versies met elkaar vergeleken. Het is echter niet mogelijk om deze pagina te testen via Pingdom gezien de gebruiker ingelogd dient te zijn. Tijdens deze test is er gebruik gemaakt van de Chrome Devtools Performance tab.

De performance tab is gebruikt om na te gaan hoe lang het duurt voordat het scherm is volledig is opgebouwd. In figuur 2.2 staat het laadproces weergeven. Een uitleg van de kleuren staat vervolgens in figuur 2.3 waarin de laadtijden gecategoriseerd zijn.



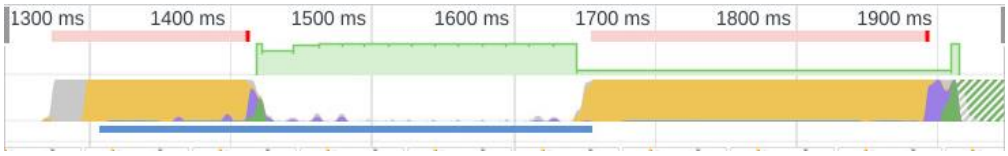
Figuur 2.2 - .Net Core loading



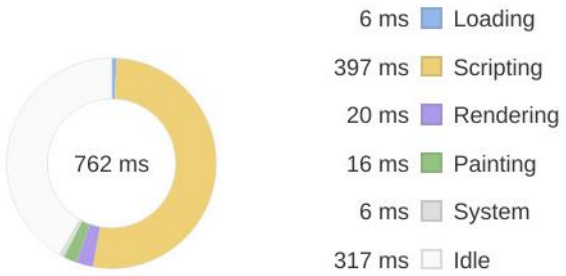
Figuur 2.3 - .NET Core

In deze figuren is te zien dat de browser meer dan 70% van de tijd stil 'idle' lijkt te staan. Dit is te verklaren doordat een deel van het scherm via JavaScript wordt aangevraagd en op de server wordt opgebouwd. Vervolgens wordt het gedeelte van het scherm terug gestuurd naar de client en met behulp van JavaScript in het dom gezet. In figuur 2.3 is te zien dat het 1019 ms duurt voordat het scherm volledig getoond wordt.

In figuur 2.4 staat het laadproces weergeven van de Blazor variant. Een uitleg van de kleuren staat vervolgens in figuur 2.5 waarin de laadtijden gecategoriseerd zijn.



Figuur 2.4 - Blazor loading



Figuur 2.5 - Blazor

Ook in deze figuren is de zien dat de browser even stil staat. Dit is te verklaren doordat de browser bijna geen verdere acties kan ondernemen terwijl het staat te wachten op de data van de API. In figuur 2.5 is te zien dat het scherm in 762 ms duurt voordat het scherm volledig getoond wordt.

Wanneer bovenstaande data met elkaar vergeleken wordt kan geconcludeerd worden dat de Blazor variant sneller is dan de .NET Core variant. Dit is te verklaren doordat webassembly

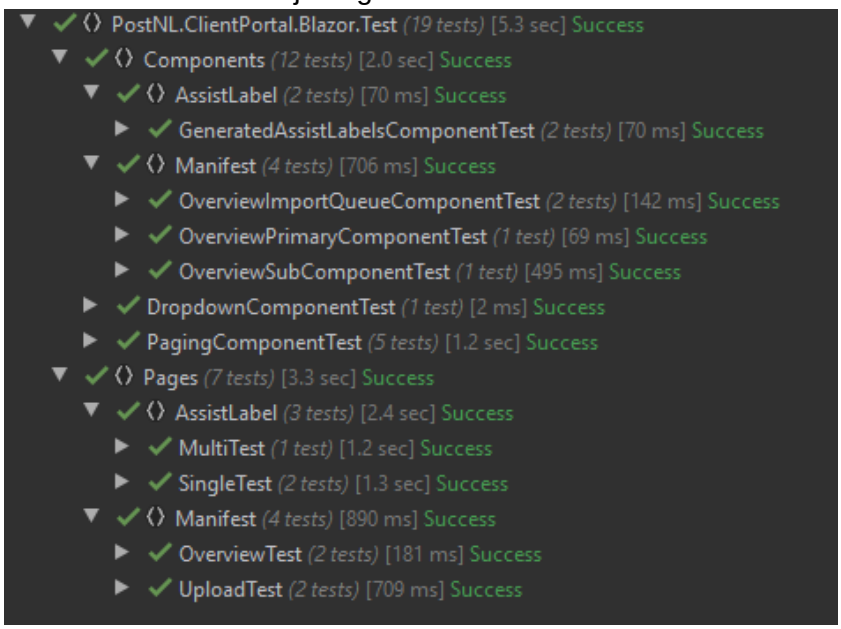
beter gebruik maakt van lokale resources en dat niet het volledige scherm opnieuw opgebouwd hoeft te worden.

3. Unit testing

Goede code dient ook getest te worden om aan te tonen dat de geschreven code daadwerkelijk doet wat verwacht wordt. Voor dit project is er gekozen om unit tests te schrijven gezien ze snel feedback geven of de geschreven code werkt zoals verwacht. Het is hierom ook een best practice om zoveel mogelijk unit tests te schrijven en later pas te kijken naar integration testen, deze doen er langer over om uit te voeren waardoor je ook pas later feedback ontvangt. Door deze unit tests wordt ook voldaan aan het vooropgestelde niet-functionele requirement NF4: “De UI van applicatie moet automatisch getest kunnen worden”.

Voor dit project is gebruik gemaakt van het test framework bUnit (https://github.com/egil/bunit). bUnit is een uitbreiding op xUnit waardoor niet alleen ‘gewone’ C# code getest kan worden, maar ook Blazor componenten. In het geheugen kunnen de Blazor componenten, het UI, opgebouwd worden terwijl hier verschillende testen op worden uitgevoerd. In .NET core werden deze testen uitgevoerd door middel van een JavaScript UI framework genaamd NightmareJS gezien deze manier van testen niet mogelijk was.

In figuur 3.1 is te zien dat de verschillende componenten getest worden en hoe lang het duurt voor de testen zijn uitgevoerd.



Figuur 3.1 - Unit tests

4. Back-end aanpassingen

In paragraaf 2.2 ben ik er achter gekomen dat door het aanpassen van de JSON library de verschillende modellen sneller gedeserialiseerd kunnen worden. Na dit onderzoek ben ik me gaan afvragen wat ik mogelijk nog meer kan aanpassen om objecten sneller te laten

deserialiseren. Hierdoor ben ik opzoek gegaan naar andere methoden en heb ik verder gekeken dan alleen de front-end aanpassingen. De back-end aanpassingen die in dit hoofdstuk zijn onderzocht, zijn minder ingrijpend dan de aanpassingen die gevonden zijn in het onderzoek zoals de back-end replicatie oplossing. De back-end aanpassingen hebben alleen te maken met het deserialiseren van de data, hier is voor gekozen om zo een volledigere advies te kunnen geven.

4.1 Benodigde data

Na het bekijken van het JSON-response bleek dat deze veel meer informatie bevat dan dat het klantenportaal daadwerkelijk nodig heeft. Dit is te verklaren doordat het volledige object wordt teruggestuurd zonder dat de benodigde data eruit wordt gefilterd. Om deze reden heb ik gecontroleerd welke data er daadwerkelijk nodig is in het klantenportaal. Van de 18 kB aan data, wat verstuurd wordt door de API, wordt er slechts 3 kB gebruikt. Het deserialiseren van enkel de benodigde data duurt 1,36 ms in Blazor. Dit is nog eens meer dan zeven keer sneller dan de situatie in Blazor met de Utf8Json library. Tijdens dit onderzoek ben ik er ook achter gekomen dat er naast het versturen van JSON-objecten de objecten ook in een binair formaat opgestuurd zouden kunnen worden. In de volgende paragraaf ga ik hier verder op in.

4.2 Binair

Het opsturen van het object in een binair formaat levert mogelijk een verbteringen in snelheid op gezien JSON is geoptimaliseerd voor JavaScript terwijl C# een binair formaat makkelijker kan deserialiseren. Om te controleren of dit ook daadwerkelijk een performance optimalisatie oplevert voor het klantenportaal heb ik hier onderzoek naar gedaan. Uit dit onderzoek zijn twee veel voorkomende methoden naar voren gekomen die data versturen in een binair formaat, Protocol buffers en MessagePack.

Protocol buffers
Protocol buffers, ook wel ProtoBuf genoemd, is een techniek welke is ontwikkeld door Google. Protocol buffers zetten de data om in een kleine en gestructureerde binaire data. Bij het gebruik van protocol buffers worden de verschillende protocol buffer messages types opgeslagen in .proto bestanden. Deze bestanden zijn zo opgesteld dat vanuit deze bestanden gemakkelijk de bijbehorende requesten gegenereerd kunnen worden. Deze requesten kunnen voor zowel voor de versturende als de ontvangende kant gegenereerd worden. Het grootste nadeel van protocol buffers is dat de data die wordt opgestuurd niet leesbaar is en er is geen manier om mee te geven van welke .proto file het komt om het te ontcijferen.
MessagePack
Een andere manier om modellen in een binair formaat op te sturen is door gebruik te maken van MessagePack. Zoals MessagePack zelf zegt: “It's like JSON. but fast and small.”. MessagePack verschilt van protocol buffers doordat in het model zelf wordt toegevoegd wat voor type elke property heeft. MessagePack geeft de gebruiker twee opties om de data op te zetten. De eerste manier is om net als bij JSON alle velden een naam te geven en hier een value bij plaatsen, deze wordt in tabel 4.2 op de derde rij aangegeven. Hierdoor is het request compacter, maar nog niet optimaal. Een tweede manier is om alle waardes van het

object in een array achter elkaar te plaatsen, deze wordt in tabel 4.2 op de vierde rij aangegeven. Dit is nog compacter dan de eerste manier.

Test
Om te bepalen welke manier de meeste optimalisatie oplevert is er een test uitgevoerd voor elk van de gevonden methoden. Zoals in hoofdstuk 2 is beschreven maakt Blazor gebruik van JIT, om deze reden heb ik ook de eerste laadtijd in de tabel meegenomen. De test is na de eerste keer nog tien keer uitgevoerd. Het gemiddelde van de testen is opgenomen in tabel 4.2

Naam	Eerste laadtijd	Opvolgende laadtijden
Utf8Json	260 ms	1.36 ms
Protocol buffers	190 ms	2.76 ms
MessagePack (string)	240 ms	1.1 ms
MessagePack (array)	220 ms	0.4 ms

Tabel 4.2 - deserialisatie tijden

De resultaten van tabel 4.2 wijzen uit het opsturen van het model in een binair formaat niet altijd sneller hoeft te zijn. Wanneer protocol buffers gebruikt worden is de eerste laadtijd het snelst, maar is deze in het vervolg langzamer dan Utf8Json. Dit is mogelijk te verklaren doordat Google de libraries voor C++ heeft geschreven en ik een library voor C# dien te gebruiken. MessagePack is langzamer dan protocol buffers in de eerste laadtijd, maar sneller dan Utf8Json. Wanneer de manier gebruikt wordt om alle waardes in een array achter elkaar te zetten blijkt dat MessagePack na de eerste keer tot wel 3.4 keer sneller deserialiseert dan Utf8Json.

Adviesrapport

Performance optimalisatie bij PostNL

K.C.J van Zeijl
16067614
De Haagse Hogeschool
Software Engineering
Leerjaar 4 (2019 - 2020)

Wateringen
30-04-2020
Versie 1.0

Inhoudsopgave

1. Inleiding

De aanleiding van dit adviesrapport is de afstudeeropdracht vanuit Nalta om de performance van het klantenportaal te verbeteren. In de huidige situatie duurt ophalen van de website in europa ongeveer 500 ms terwijl het buiten Europa soms tot wel 6 seconden duurt. Het klantenportaal bestaat uit een .NET Core applicatie welke data ophaalt door middel van een API. Dit rapport bevat de aanbevelingen en adviezen die gedaan zijn op basis van de resultaten van het onderzoek en testverslag.

In het eerste hoofdstuk wordt het onderzoek nog eens behandeld waarin ik ook de onderzoekshypothese nog eens benoem die uit het onderzoek is gevolgd. Vervolgens ga ik in op de verschillende alternatieven die Nalta kan gebruiken om mogelijk de performance van het klantenportaal te verbeteren. Voor ik een definitief advies geef, benoem ik ook nog een aantal kleine wijzigingen die aan de back-end kunnen worden doorgevoerd maar mogelijk een grote performance optimalisatie kunnen opleveren. Als laatste wordt er een definitief advies geleverd aan Nalta.

2. Onderzoek

In het performance onderzoek is het performance probleem van het klantenportaal van PostNL onderzocht door middel van de volgende hoofdvraag: *“Hoe kan Nalta de performance van het klantenportaal van PostNL in regio’s buiten Europa significant verbeteren?”.*

Deze hoofdvraag is beantwoord door antwoord te geven op de volgende twee deelvragen:

1. *“Wat is momenteel de oorzaak van de lage performance van het klantenportaal in regio's buiten Europa?”*
2. *“Welke bestaande methoden kunnen worden toegepast om het serveren van het klantenportaal te versnellen?”*

In hoofdstuk 4 van het onderzoek is de eerste deelvraag onderzocht, waar het performance probleem van het klantenportaal vandaan komt. Hieruit is gebleken dat het klantenportaal kampt met een latency probleem en dan voornamelijk de afstand tussen de server en de eindgebruiker. In het volgende hoofdstuk is onderzocht hoe deze afstand verminderd kan worden.

In hoofdstuk 5 van het onderzoek is de tweede deelvraag beantwoord. Hierin zijn er oplossingen gezocht om de afstand tussen de server en de eindgebruiker te verminderen. Verschillende oplossingen vielen af door vooraf opgestelde technische beperkingen of door de complexiteit van de infrastructuur drastisch te veranderen. Gebruik maken van een CDN lijkt wel een geschikte oplossing. Hiervoor moet een website echter gehost kunnen vanaf een static file hosting server. Hoe dit gedaan kan worden staat uitgelegd in hoofdstuk 6 van het onderzoek.

In hoofdstuk 6 van het onderzoek wordt static file hosting besproken. Hieruit bleek dat static site generators niet gebruikt kunnen worden voor het klantenportaal door de complexiteit van het klantenportaal. Moderne front-end frameworks kunnen wel goed gebruikt worden voor het klantenportaal. Door het creëren van een single page application kan de volledige website gehost worden op een static file hosting server en zo mogelijk de performance te verbeteren.

Uit het onderzoek is de volgende hypothese gekomen: *“Door middel van een modern front-end framework in combinatie met een CDN wordt de performance van het klantenportaal van PostNL verbeterd.”*. Vervolgens is in het testverslag te lezen dat deze hypothese is bewezen met een prototype welke is geschreven in Blazor. Blazor is een interessante ontwikkeling en zeker in de toekomst een goede optie. Echter, nu is het nog in preview en zullen er nog relatief veel kinderziektes in zitten waardoor dit meegenomen moet worden in de keuze.

3. Alternatief Rust

In het onderzoek is er aangeraden om Blazor te gebruiken doordat het gebruik maakt van WebAssembly en geschreven kan worden in C#. Echter, het gebruik van Blazor is niet de enige optie om de snelheid van WebAssembly te benutten. Ook Rust is hier een goede kandidaat voor.

Rust (https://www.rust-lang.org/) is een ontzettend snelle programmeertaal. De syntax van Rust is te vergelijken met die van C++. Rust is gestart door Graydon Hoare (Cassel, 2019) en werd in 2010 voor het eerst gepubliceerd. Na deze initiële publicatie is Rust steeds populairder geworden door de snelheid dat het levert in combinatie met veiligheidsgaranties die de compiler levert op het alloceren en gebruik van geheugen. Hiermee worden veel voorkomende memory bugs van C / C++ voorkomen. Een applicatie kan snel vanaf de command line worden opgezet en vervolgens gecompileerd worden naar verschillende soorten machine code. Voor de ontwikkelingen van WebAssembly werd de rust code vooral omgezet naar assembly code om op een laag level code te draaien. Hierna heeft Rust grote stappen gemaakt binnen de WebAssembly wereld en kan de Rust code hier nu ook naar gecompileerd worden.

Het grootste nadeel dat Nalta bij rust zal ondervinden is dat C# niet gebruikt kan worden en ervaring in Rust moet worden opgedaan. Volledig een nieuwe taal leren kost vaak veel tijd en daarom ook veel geld.

4. Alternatief modern JavaScript framework

Naast de WebAssembly techniek is het ook mogelijk om het klantenportaal om te zetten naar een single page application met behulp van een modern JavaScript framework. In het onderzoek zijn Angular en Vue.js als voorbeelden aangegeven omdat Nalta al ervaring heeft met deze technieken. Beide opties kunnen goed gebruikt worden om de website vanaf een static file hosting server te laten hosten.

Angular richt zich hierbij op grote applicaties die behoefte hebben aan de werking van een single page application. Hierbij brengt Angular een hoop functionaliteiten met zich mee die direct geïmplementeerd kunnen worden. Vue.js, ook wel Vue genoemd, is op zichzelf een klein framework en kan gebruikt worden voor grote en kleine projecten. Doordat het een klein framework is moet een groot deel van de functionaliteit die Angular biedt wel nog worden toegevoegd.

Het grootste voordeel is dat het omzetten van JSON naar een JavaScript object geoptimaliseerd is en hierdoor ontzettend snel wordt afgehandeld. Een nadeel is echter dat de logica moet worden herschreven van C# naar JavaScript en er hierdoor in twee verschillende talen wordt ontwikkeld.

5. Alternatief back-end aanpassingen

Naast het omzetten van het klantenportaal naar een single page application heb ik ook andere manieren gevonden die gebruikt kunnen worden om de performance te verbeteren. Deze methoden zullen echter niet dezelfde impact op de performance leveren maar zouden ook gecombineerd kunnen worden met de hiervoor beschreven oplossingen.

5.1 Benodigde data terugsturen

In het testverslag van het Blazor prototype is er een onderzoek gedaan naar de achterliggende reden van het performance probleem voor het deserialiseren. Uit dit onderzoek bleek dat de aangeraden deserialisatie methode van Microsoft nog niet geoptimaliseerd was voor grote JSON objecten. Door een andere deserialisatie library te gebruiken is het probleem al minder groot geworden echter was ik nog niet tevreden en heb ik gekeken naar een manier om het probleem nog meer in te perken.

Het bleek dat het JSON response veel meer informatie bevat dan dat daadwerkelijk nodig is. Van de 18 kb aan data, wat verstuurd wordt van de API, wordt er slechts 3 kb gebruikt. Het deserialiseren van enkel de benodigde data duurde 2 ms in Blazor. Dit is nog eens drie keer sneller dan de huidige situatie in Blazor. Het nadeel is echter dat er aanpassingen aan de back-end moeten worden doorgevoerd om dit te bewerkstelligen, wat ingaat tegen de vooropgestelde requirements. Om deze reden is het niet als officiële oplossingen aangegeven.

5.2 Binair protocol

Een andere oplossing die mogelijk gebruikt kan worden, is door het request vanuit de API in een binair formaat terug te sturen. JSON is geoptimaliseerd voor JavaScript terwijl C# een binair formaat makkelijker kan omzetten. In paragraaf 9.3 van het afstudeerverslag staat meer informatie over het onderzoek waarin deze manier wordt onderzocht. Hieruit blijkt dat er inderdaad manieren zijn om data in een binair formaat op te sturen waardoor het sneller wordt gedeserialiseerd dan JSON-objecten. Echter, net als bovenstaande methode moeten er aanpassingen aan de backend worden doorgevoerd om dit te bewerkstelligen. Hierom is dit niet verder onderzocht en niet als officiële oplossingen aangegeven.

6. Aanbeveling

Uit het onderzoek is een hypothese gekomen welke vervolgens is bewezen aan de hand van een Prototype. De gekozen techniek om het prototype te bouwen was Blazor, want Blazor maakt gebruik van het ontzettend snelle WebAssembly en er kan ontwikkeld worden in C#. Het nadeel van Blazor is echter dat het op moment van schrijven nog in preview is.

Op basis van de onderzoeksresultaten en een zorgvuldige afweging van de voor- en nadelen van de beschreven alternatieven is het advies om het klantenportaal in Blazor te maken. Echter wanneer Nalta vindt dat de voordelen van Blazor minder waard zijn dan het werken met preview software is, dan wordt geadviseerd om het klantenportaal te schrijven in Vue.js of Angular. Dit advies wordt gegeven gezien Nalta niet van plan is om aan de slag te gaan met Rust en er met behulp van Vue.js en Angular ook een single page application ontwikkeld kan worden die gehost kan worden op een static file hosting server.

Naast de techniek voor het klantenportaal raad ik Nalta ook aan om nog meer onderzoek te doen naar de gegeven back-end aanpassingen omdat deze de performance van het klantenportaal nog meer kunnen verbeteren.

Literatuurlijst

Cassel, D. (2019, 17 juni). *Rust Creator Graydon Hoare Talks About Security, History, and Rust*. Geraadpleegd op 28 april 2020, van https://thenewstack.io/rust-creator-graydon-hoare-talks-about-security-history-and-rust/