FontoXML

Afstudeerverslag

Uitvoeren van onderzoek naar de performance van de rendering engines in browsers en voorstellen van verbetering bij FontoXML

Student:	T. N. Gravekamp
Studentnummer:	11023449
Onderwijsinstelling:	De Haagse Hogeschool
Opleiding:	Informatica
Afstudeerperiode:	09-02-2015 – 05-06-2015
Begeleidend examinator:	Dhr. E. M. van Doorn
Tweede examinator:	Dhr. G. M. Tuk
Bedrijf:	FontoXML
Opdrachtgever:	Dhr. B. Willems
Bedrijfsbegeleider:	Dhr. M. Middel
Inleverdatum:	05-06-2015

Referaat

Dit afstudeerverslag is geschreven ter afronding van de opleiding Informatica aan De Haagse Hogeschool te Den Haag. In dit afstudeerverslag worden de werkzaamheden beschreven die zijn uitgevoerd tijdens de afstudeerperiode van 9 februari 2015 tot en met 5 juni 2015. De titel van de uitgevoerde opdracht luidt: “Uitvoeren van onderzoek naar de performance van de rendering engines in browsers en voorstellen van verbetering bij FontoXML”. Het doel van dit afstudeerverslag is inzicht geven in de uitgevoerde werkzaamheden.

Tijdens de uitvoer van de afstudeeropdracht is onderzoek uitgevoerd naar optimalisaties op het gebied van HTML en CSS. Als onderdeel van dit onderzoek is een proof of concept gebouwd waarmee de effectiviteit van de gevonden optimalisatie gemeten kon worden.

Descriptoren:

- FontoXML
- XML
- HTML
- CSS
- DOM
- Optimalisaties

Voorwoord

Dit verslag heb ik geschreven met als doel de lezer te informeren over de uitvoer van mijn afstudeeropdracht. Tijdens de uitvoer van deze opdracht heb ik onderzoek gedaan naar optimalisatiestrategieën voor FontoXML.

Ik heb tijdens de uitvoer van deze opdracht een hoop geleerd op het gebied van de interne werking van browsers. Ook heb ik een voor mij totaal nieuwe taal geleerd.

Bij deze wil ik graag de volgende personen bedanken:

- Mijn opdrachtgever dhr. Bert Willems, voor het aanbieden van deze interessante opdracht
- Mijn bedrijfsbegeleider dhr. Martin Middel, voor de goede begeleiding tijdens deze opdracht en het bieden van nieuwe inzichten wanneer ik vast liep
- Mijn examinatoren dhr. E.M. van Doorn en dhr. G.M. Tuk, voor de feedback op het afstudeerverslag
- Mijn collega's bij Liones en FontoXML voor de gezellige werksfeer
- Mijn ouders, voor de steun tijdens de uitvoer van deze opdracht

Ik wens u als lezer veel plezier tijdens het lezen van dit verslag.

Inhoudsopgave

1 Inleiding

In dit document beschrijf ik de uitvoer van de afstudeeropdracht “Uitvoeren van onderzoek naar de performance van de rendering engines in browsers en voorstellen van verbetering bij FontoXML”. Deze afstudeeropdracht heb ik uitgevoerd in het kader van mijn afstuderen bij de opleiding Informatica aan De Haagse Hogeschool te Den Haag.

Tijdens deze afstudeeropdracht heb ik onderzoek gedaan naar optimalisaties op het gebied van HTML en CSS. Daarnaast heb ik als onderdeel van het onderzoek een proof of concept geïmplementeerd om te weten te komen of de voorgestelde optimalisatie een positief (of negatief) effect zal hebben op de applicatie.

Dit document volgt de volgorde waarin ik de werkzaamheden heb uitgevoerd. Per onderdeel worden de keuzes die ik gemaakt heb beschreven. Hierbij beschrijf ik de keuzemogelijkheden, de afwegingen en welke keuze ik uiteindelijk genomen heb. Tijdens het uitleggen van de werkzaamheden heb ik een aantal specifieke begrippen gebruikt. Deze begrippen worden uitgelegd in de woordenlijst aan het eind van dit document.

2 Organisatie

In dit hoofdstuk beschrijf ik het bedrijf waar ik de afstudeeropdracht heb uitgevoerd.

2.1 Bedrijf

FontoXML is een bedrijf dat een gelijknamig product ontwikkelt en verkoopt. Dit product is een gebruiksvriendelijke editor voor het maken en bewerken van gestructureerde content (XML). Daarnaast adviseert FontoXML partijen met uitgeefvraagstukken. FontoXML werkt onder andere voor Wolters Kluwer, Thieme Meulenhoff, IAEA en het RILM.

FontoXML is begin 2014 afgesplitst van Liones, een internetbureau dat al 12 jaar actief is in de uitgeverij. Daardoor is FontoXML zelf een jong bedrijf.

Begin 2014 is gestart met de ontwikkeling van FontoXML. Versie 1 wordt op dit moment afgerond en uitgeleverd aan een viertal early adoptors. Binnen het team wordt met SCRUM ontwikkeld.

2.2 Producten en diensten

FontoXML verkoopt de gelijknamige editor FontoXML aan klanten. Aan het basispakket kunnen daarnaast nog verschillende add-ons toegevoegd worden om de editor meer functionaliteit te geven. Denk hierbij aan add-ons voor spellingscontrole, wiskundige formules en het geautomatiseerd genereren van samenvattingen. Daarnaast is het mogelijk om klantspecifieke aanpassingen of add-ons te maken.

2.3 Omvang

Op dit moment werken er 10 fulltime werknemers, een aantal stagiair(e)s en afstudeerders bij FontoXML. Daarnaast werken er bij Liones nog eens 20 medewerkers.

Binnen het ontwikkelteam is geen sprake van een hiërarchische structuur. Jan Benedictus is algemeen directeur van het bedrijf. Bert Willems is product architect en eindverantwoordelijke voor het gehele product. De rest van het ontwikkelteam en ik vallen onder Bert Willems.

2.4 Klanten

- Wolters Kluwer
- Thieme Meulenhoff
- IAEA
- RILM

3 Probleemstelling

FontoXML is een webapplicatie specifiek gebouwd voor het bewerken van documenten met een DITA of TEI indeling. Het doel van FontoXML is om net zo gebruiksvriendelijk te werken als Microsoft Word of Google Docs, daarnaast is het doel van FontoXML om na elke bewerking in het document een valide XML-document te garanderen welke klopt volgens het gebruikte XML schema. Ook is het de bedoeling dat de applicatie, in tegenstelling tot concurrerende pakketten, om kan gaan met DITA- of TEI-documenten van enkele honderden pagina's.

De applicatie is in staat om XML-documenten met een DITA of TEI indeling te openen. Tijdens het bewerken van een XML-document van een document met enkele tientallen pagina's is er geen performanceverlies merkbaar. XML-documenten van enkele honderden pagina's zorgen ervoor dat tussen de toetsaanslag en het moment waarop het ingevoerde teken op het scherm verschijnt een merkbare vertraging zit. Deze vertraging is hinderlijk voor de gebruikers van de applicatie.

Vanuit de klanten van FontoXML is de behoefte ontstaan om documenten van enkele honderden pagina's te openen. Een voorbeeld van een klant met deze behoefte is het IAEA (International Atomic Energy Agency). Deze organisatie wil zogenaamde *nuclear saftey standard* documenten van typisch 80 tot 100 pagina's kunnen bewerken. Vanuit deze behoefte is deze afstudeeropdracht ontstaan.

4 Doelstelling

Het doel van deze afstudeeropdracht is het uitbrengen van een advies over de meest effectieve optimalisatiestrategieën voor de web-applicatie FontoXML. Dit advies wordt uitgebracht op basis van een onderzoek naar deze optimalisatiestrategieën. Als onderdeel van dit onderzoek is een proof of concept gebouwd, waarmee bepaald kon worden of de gevonden optimalisatiestrategie daadwerkelijk een snelheidswinst oplevert.

Aan het eind van deze afstudeeropdracht zal een adviesrapport opgeleverd worden waarin een advies zal worden uitgebracht over de meest effectieve optimalisatiestrategieën. Op basis van dit advies zal het ontwikkelteam van FontoXML de optimalisatiestrategie op de roadmap plaatsen. De optimalisatiestrategie zal uiteindelijk door het ontwikkelteam van FontoXML geïmplementeerd worden.

5 Plan van aanpak

Dit project begon ik met het schrijven van een plan van aanpak en het maken van de bijbehorende planning. Het plan van aanpak is in combinatie met de planning een fundament voor het uitvoeren van een project. Het plan van aanpak en de planning hebben tijdens de uitvoer van het project houvast geboden in de uitvoering van het project.

5.1 Scope en afbakening

In het plan van aanpak heb ik vastgelegd welke activiteiten uitgevoerd moesten worden om het project af te kunnen ronden. Deze activiteiten vormen samen de project scope. In het plan van aanpak heb ik de volgende activiteiten gedefinieerd:

Activiteit	**Specificatie**
Bepalen baseline profile (testrapport)	• Bepalen hoe de performance van FontoXML betrouwbaar gemeten kan worden • Benchmarks draaien met de bestaande situatie
Uitvoeren onderzoek	• Opstellen van een hoofdvraag en een deelvraag • Uitvoeren van vooronderzoek en vergaren van bronnen • Uitvoeren literatuuronderzoek • Verwerken bevindingen literatuuronderzoek • Uitvoeren experimenten voor het beantwoorden deelvragen • Conclusie trekken om de hoofdvraag te kunnen beantwoorden
Ontwerp proof of concept	• Door middel van verschillende diagrammen in kaart brengen hoe het proof of concept gebouwd zal worden
Bouw proof of concept	• Bouwen van het proof of concept op basis van de opgestelde ontwerpen
Opstellen testrapport	• Benchmarks draaien met het prototype • Vergelijken van benchmarks
Opstellen adviesrapport	• Opstellen adviesrapport • Opstellen advies over de te implementeren optimalisaties

Naast de uit te voeren activiteiten heb ik in het plan van aanpak een aantal punten opgesteld welke samen de afbakening van de opdracht vormen. De volgende punten vormen samen deze afbakening:

- Het onderzoek zal zich richten op optimalisaties welke betrekking hebben op het optimaliseren van HTML of CSS;
- Het proof of concept hoeft geen andere functionaliteit te bevatten dan de optimalisaties die uit het onderzoek zijn gekomen;
- De kwaliteit van het proof of concept hoeft niet van productieniveau te zijn.

5.2 Risicofactoren

Naast de scope van het project heb ik in het plan van aanpak ook een aantal risicofactoren beschreven die de uitvoer van het project konden hinderen. Door een wekelijks voortgangsgesprek te hebben met mijn bedrijfsbegeleiders heb ik de onderstaande risico's al deels kunnen beperken.

Risicofactor 1: Het onderzoek kan niet volledig worden afgerond binnen de geplande tijd vanwege de complexiteit van het onderwerp.
Dit is een risico omdat het project moeilijk af te ronden is met een incompleet onderzoek. Dit risico heb ik beperkt door het onderzoek af te bakenen en een planning te maken voor de uitvoer van het onderzoek. Wanneer het onderzoek toch langer zou gaan duren dan gepland, had ik het onderzoek minder uitgebreid moeten afronden door het onderzoek nog verder af te bakenen. Hierbij had ik rekening moeten houden met de tot dan toe vergaarde informatie.

Risicofactor 2: De te implementeren optimalisaties blijken te complex voor een proof of concept.
Dit is een risico omdat er twee keer performancetests uitgevoerd dienen te worden. Een keer aan het begin van het project en keer een aan het eind van het project. Wanneer een optimalisatie te complex blijkt om te kunnen implementeren in de geplande tijd, kan de tweede keer tests niet uitgevoerd worden. Hierdoor kan de nieuwe situatie niet worden vergeleken met de bestaande situatie. Dit risico heb ik deels kunnen beperken door gebruik te maken van een op SCRUM gebaseerde softwareontwikkelmethode. Wanneer een te implementeren optimalisatie toch te complex zou blijken voor het proof of concept, had ik moeten inschatten of de te implementeren optimalisatie op een simpelere of minder uitgebreide manier geïmplementeerd had kunnen worden.

Risicofactor 3: Het blijkt dat er geen literatuur beschikbaar is over het onderwerp.
Dit is een risico voor de uitvoer van het onderzoek. Dit risico was niet te beperken omdat de hoeveelheid literatuur vanzelfsprekend niet te beïnvloeden is. Als blijkt dat er geen literatuur beschikbaar is over het onderwerp, had ik het onderzoek voort moeten zetten aan de hand van het uitvoeren van kleine tests om te bepalen waarmee een performancewinst behaald kan worden.

5.3 Planning

Op basis van de globale planning heb ik een detailplanning gemaakt. Hieronder staat de detailplanning voor dit project:

Product	**Uitvoerdatum**	**Duur**
Plan van aanpak	09-02-2015 t/m 11-02-2015	3 dagen
Baseline profile - Testmethode bepalen - Benchmarks oude situatie	12-02-2015 t/m 19-02-2015	7 dagen - 5 dagen - 2 dagen
Onderzoeksrapport - Vooronderzoek - Literatuuronderzoek - Verwerken bevindingen - Conclusie	20-02-2015 t/m 26-03-2015	18 dagen - 4 dagen - 8 dagen - 4 dagen - 2 dagen
Ontwerp proof of concept	27-03-2015 t/m 03-04-2015	5 dagen
Bouw proof of concept	06-04-2015 t/m 08-05-2015	20 dagen
Testrapport - Benchmarks nieuwe situatie - Oude en nieuwe situatie vergelijken	11-05-2015 t/m 22-05-2015	8 dagen - 3 dagen - 5 dagen
Adviesrapport	25-02-2015 t/m 05-06-2015	9 dagen

5.4 Methode

In het plan van aanpak heb ik vastgelegd welke softwareontwikkelmethode ik zou gaan gebruiken tijdens de bouw van het proof of concept. Ik koos voor de Agile softwareontwikkelmethode SCRUM omdat ik van tevoren nog niet wist uit welke en hoeveel onderdelen het proof of concept zou gaan bestaan. Omdat ik dit hele project alleen werkte was het echter niet mogelijk om de softwareontwikkelmethode SCRUM te gebruiken. Daarom heb ik ervoor gekozen om een softwareontwikkelmethode te gebruiken welke gebaseerd is op SCRUM.

Omdat SCRUM is gebaseerd op het werken in teamverband, zijn ook de bijbehorende activiteiten gebaseerd op het werken in teamverband. Door met het ontwikkelteam van FontoXML mee te doen aan de dagelijkse standup en de tweewekelijkse retrospective heb ik deze twee activiteiten toch kunnen uitvoeren. Het voordeel hiervan was dat ik makkelijk en snel hulp kon vragen en wist het team waar ik op dat moment mee bezig was.

Omdat de gebruikte softwareontwikkelmethode gebaseerd is op SCRUM, heb ik tijdens de bouw van het proof of concept gewerkt met sprints. Het werken met sprints was voor de bouw van het proof of concept een zeer geschikte eigenschap, omdat ik van tevoren nog niet precies wist hoe complext de bouw van het proof of concept zou zijn. Ik wist van tevoren ook nog niet welke optimalisatiestrategie uiteindelijk uitgewerkt zou gaan worden in het proof of concept, waardoor het onmogelijk was om een precieze planning te maken.

Ik heb deze softwareontwikkelmethode alleen toegepast tijdens het bouwen van het proof of concept. De performancetests en het onderzoek heb ik niet ingedeeld in sprints. De performancetests heb ik opgenomen in de detailplanning. Voor het onderzoek heb ik een aparte fasering gemaakt. Deze fasering heb ik opgenomen in het onderzoeksplan.

6 Testrapport (baseline profile)

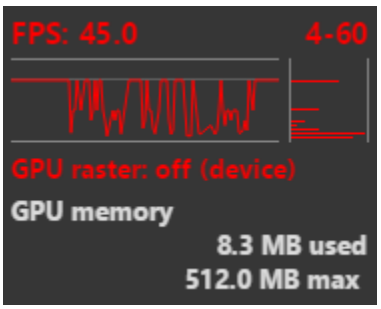
Ik begon dit project met het bepalen van de baseline profile. De baseline profile bestaat uit een set testresultaten waarmee de resultaten van de performancetests met het proof of concept vergeleken kunnen worden. Voordat ik de performancetests voor de baseline profile kon uitvoeren moest ik eerst de testmethode bepalen. De testmethode en de resultaten van de performancetest voor de baseline profile heb ik opgenomen in het testrapport. Het testrapport heb ik later aangevuld met de performancetests van het proof of concept.

De performancetests moeten inzicht geven in welke onderdelen van de applicatie of browser voor het grootste performanceverlies zorgen. Daarnaast moeten de performancetests van het proof of concept vergeleken kunnen worden met de baseline profile.

6.1 Testmethode

Om de performancetests uit te kunnen voeren begon ik met het bepalen van de te gebruiken testmethode voor de performancetests. Het doel van deze testmethode is het geven van inzicht in de performance van de applicatie tijdens het uitvoeren van de tests. Voorwaarde voor deze testmethode was dat deze reproduceerbaar zou zijn, zodat de testmethode in dezelfde situatie meerdere malen uitgevoerd kan worden.

In het afstudeerplan heb ik in overleg met de opdrachtgever beschreven dat 60 frames per seconde (FPS) een must zou zijn voor een soepele gebruikerservaring. Het renderen van 60 frames per seconde zorgt in games voor een soepele gebruikerservaring. Als 60 frames per seconde niet haalbaar is door hardware- of softwarelimitaties, richt men op 30 frames per seconde. Het doel is uiteindelijk om een constante FPS te behouden, omdat de gebruiker het direct merkt wanneer deze inzakt (Bithell, 2014) (Stanley, 2014).



Figuur 1: Schommeling van FPS in browser

Met meten van het aantal frames per seconde leek een goede meetwaarde voor het bepalen van de performance van de browser. Echter rendert een browser pas een nieuw frame wanneer een element op de pagina verandert. Hierdoor is het aantal frames per seconde in een browser nooit constant en zijn hieraan geen conclusies te verbinden. Daarom is het meten van het aantal frames per seconde geen betrouwbare manier van het meten van de performance van een webapplicatie. Daarnaast kan er ook niet gericht worden op het behalen van een bepaald aantal frames per seconde in een browser.

In Figuur 1: Schommeling van FPS in browserFiguur 1 is te zien dat tijdens het typen in de applicatie het aantal frames per seconde schommelt tussen 4 en 60 FPS, zonder dat performanceverlies merkbaar was.

Nadat bleek dat het aantal frames per seconde geen goede meetwaarde is, ben ik verder gaan kijken naar de mogelijkheden om de performance van de applicatie te meten met behulp van de profilers welke standaard in elke moderne browsers zitten[1][2][3][4][5]. Deze profilers zijn onder andere in staat om een overzicht te geven van de uitgevoerde scripts, de hoeveelheid tijd waarin de browser niets doet, de tijd die de garbage collector actief is en de tijd die de browser nodig heeft om de pagina (opnieuw) te renderen.

[1] Google Chrome: https://developer.chrome.com/devtools
[2] Firefox: https://developer.mozilla.org/en-US/docs/Tools
[3] Internet Explorer: https://msdn.microsoft.com/library/bg182326%28v=vs.85%29
[4] Opera: http://www.opera.com/dragonfly/
[5] Safari: https://developer.apple.com/safari/tools/

6.2 Software

Om de tests reproduceerbaar te maken besloot ik om de tests uit te voeren in een virtuele machine. Van een virtuele machine kan een snapshot gemaakt worden, waarmee de machine teruggebracht kan worden naar de staat waarin de machine was op het moment dat de snapshot gemaakt werd. Hierdoor kon ik de performancetests op een later moment in exact dezelfde testomgeving uitvoeren.

Op de virtuele machine heb ik Windows 8.1 geïnstalleerd. Ik heb hier voor Windows gekozen omdat Internet Explorer alleen werkt op Windows en om de testomgeving voor de drie grootste browsers (Google Chrome, Firefox en Internet Explorer) gelijk te houden. Daarnaast heb ik gekozen voor de op dat moment meest recente versie van Windows. Dit heb ik gedaan omdat Microsoft in elke nieuwe versie van Windows de performance verbetert ten opzichte van de vorige versie (Terkaly, 2013) (Walton, 2012). Hierdoor had deze versie van Windows op het gebied van performance het minste effect op de resultaten van de performancetests.

De profilers van de drie grootste browsers meten de performance van een webpagina allemaal op een andere manier. De waarden die uit deze metingen kwamen waren onderling moeilijk of niet te vergelijken. Daarnaast waren de waarden uit Internet Explorer en Firefox niet goed af te lezen. In overleg met het team van FontoXML heb ik de performancetests uiteindelijk uitgevoerd in Google Chrome. Google Chrome heeft een uitgebreidere profiler dan Internet Explorer en Firefox. Daarnaast wordt FontoXML hoofdzakelijk ontwikkeld voor het gebruik in Google Chrome, waardoor deze browser voor het ontwikkelteam van FontoXML het belangrijkst is. Ik heb de op dat moment meest recente versie van Google Chrome gebruikt.

Om de invoer van een gebruiker te simuleren heb ik gebruik gemaakt van het programma AutoHotkey[6]. Dit programma kan met constante snelheid toetsaanslagen simuleren, waardoor elke test met dezelfde snelheid uitgevoerd kon worden. Dit is belangrijk om de performancetests later te kunnen vergelijken en om de performancetests reproduceerbaar te maken.

[6] http://www.autohotkey.com/

6.3 Testapplicaties

Ik heb de performancetests uitgevoerd in twee verschillende applicaties. De eerste applicatie is de retail versie van FontoXML. Deze editie is de meest basale vorm van de applicatie. In deze applicatie zijn geen klantspecifieke toevoegingen verwerkt.

Naast de tests met de volledige applicatie heb ik ook tests uitgevoerd met de HTML versie van de applicatie. De HTML versie is geen bestaande versie van de applicatie. Ik heb de HTML versie zelf gemaakt door de broncode van de retail versie te nemen en hier de scripts uit te halen, zodat alleen een pure HTML versie overbleef. Doordat de editor is gebaseerd op invoer via het in HTML 5 geïntroduceerde `content-editable`-attribuut[7], was het ook mogelijk de performancetests uit te voeren in de HTML versie van de applicatie zonder dat hiervoor scripts nodig waren.

Door te testen met alleen de HTML en CSS van de applicatie kan de tijd die de browser besteedt aan het renderen van de webpagina zeer zuiver gemeten worden. Dit ten opzichte van de tests met de retail versie van FontoXML, waarin de aanwezige scripts ook tijd innemen. Het doel van de performancetests met de HTML versie van de applicatie is een beter inzicht geven in de performance van de browser tijdens het renderen van de webpagina.

[7] http://www.w3.org/TR/2008/WD-html5-20080610/editing.html

6.4 Testdocumenten

Om de twee verschillende applicaties te testen heb ik een aantal verschillende testdocumenten gemaakt. Deze testdocumenten heb ik tijdens het uitvoeren van de performancetests voor de baseline profile gebruikt. Later heb ik deze documenten nog een keer gebruikt voor de performancetests met het proof of concept.

6.4.1 DITA

De documenten die in de retail versie van FontoXML bewerkt kunnen worden, zijn opgebouwd volgende de Darwin Information Typing Architecture (DITA) standaard. DITA is gebaseerd op XML en wordt gebruikt voor het maken, produceren en publiceren van informatie.

Binnen DITA worden topic-elementen gebruikt voor het beschrijven van een onderwerp. Een topic-element bevat altijd een title-element welke de titel van het topic-element beschrijft en een body-element welke de inhoud van het topic-element beschrijft. Naast een title- en een body-element kan een topic-element zelf ook een topic-element bevatten.

Hieronder zijn de hiervoor beschreven elementen te zien zoals deze in een XML-document met DITA indeling worden opgeslagen:

DITA voorbeeld

```
<topic>
  <title>Windmills</title>
  <body>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut
nec lorem vestibulum, viverra justo quis, mollis nibh. Donec posuere
diam felis, non vehicula purus sodales non. Nulla congue, dui sed
aliquet condimentum, turpis ante tincidunt mi, in porttitor mi ante
vel est. Quisque a orci elementum velit eleifend facilisis. In congue
convallis sapien ac rutrum. Cras laoreet sapien nulla, ut molestie
nisi consequat ac. Donec fermentum ullamcorper nulla nec tristique.
Morbi vulputate luctus felis vitae condimentum.</p>
  </body>
</topic>
```

6.4.2 Documenten

De retail versie van FontoXML bevat een standaard document waarmee de editor door bijvoorbeeld een klant uitgeprobeerd kan worden. Omdat tijdens het werken in dit bestand geen merkbaar performanceverlies optreedt, heb ik dit document in de tests meegenomen als vergelijkingsmateriaal. Hierdoor hoopte ik later een (duidelijk) verschil te kunnen zien tussen de resultaten van de tests met het standaard document en de tests met de andere documenten.

Naast het standaard document heb ik een tweetal andere documenten gemaakt waarmee ik de applicatie getest heb. Het eerste document is een document met 600 DITA topic-elementen. Elke tien topics wordt een standaard set aan elementen herhaald om een gemiddelde inhoud van een DITA document te representeren. Deze gemiddelde inhoud is gebaseerd op het standaard document van de retail versie van FontoXML. Bij de opbouw van dit document was het vooral belangrijk dat het document in de applicatie een merkbaar performanceverlies veroorzaakte.

Het tweede document bevat tien geneste topic-elementen. Deze topic-elementen bevatten net als de topic-elementen in de andere testdocumenten een standaard set elementen. Het doel van dit document is het bepalen van de invloed van een diep genest document op de performance van FontoXML.

6.5 Testgevallen

De uit te voeren performancetests heb ik ingedeeld in testgevallen. Deze testgevallen richten zich op het wijzigen van de lay-out van de pagina door middel van het invoeren van tekst. Het wijzigen van de lay-out van de webpagina had ook gekund door het verwijderen van tekst uit het document. Ik heb hier echter gekozen voor het invoeren van tekst, omdat het performanceverlies tijdens het invoeren van tekst voor de gebruiker het meest hinderlijk is.

Om de performancetests uit te voeren heb ik de volgende testgevallen opgesteld:

- Testgeval zin 1: Invoeren zin aan het begin van de eerste paragraaf
- Testgeval zin 2: Invoeren zin aan het eind van de eerste paragraaf
- Testgeval paragraaf 1: Invoeren paragraaf na de eerste paragraaf
- Testgeval paragraaf 2: Invoeren paragraaf aan het eind van het document
- Testgeval paragraaf 3: Invoeren paragraaf na de eerste paragraaf met 600 aanslagen per minuut

Testgeval zin 1 en zin 2 heb ik opgesteld om een inzicht te krijgen in de performance (en het mogelijke performanceverschil) van het invoeren van een zin aan het begin en aan het eind van een paragraaf. Testgeval paragraaf 1 en paragraaf 2 moeten het mogelijke performanceverschil tussen het invoeren van een paragraaf aan het begin en aan het eind van een document duidelijk maken. Testgeval Paragraaf 3 moet inzicht geven in het verschil tussen 300 aanslagen per minuut en 600 aanslagen per minuut. Een ervaren typiste moet gemakkelijk 300 aanslagen per minuut kunnen behalen[8][9]. Dit laatste testgeval heb ik opgesteld om de invloed van de typesnelheid op de performance te kunnen meten.

[8] http://en.wikipedia.org/wiki/Words_per_minute
[9] https://www.ticken.nl/Typecursus/Typetest.html

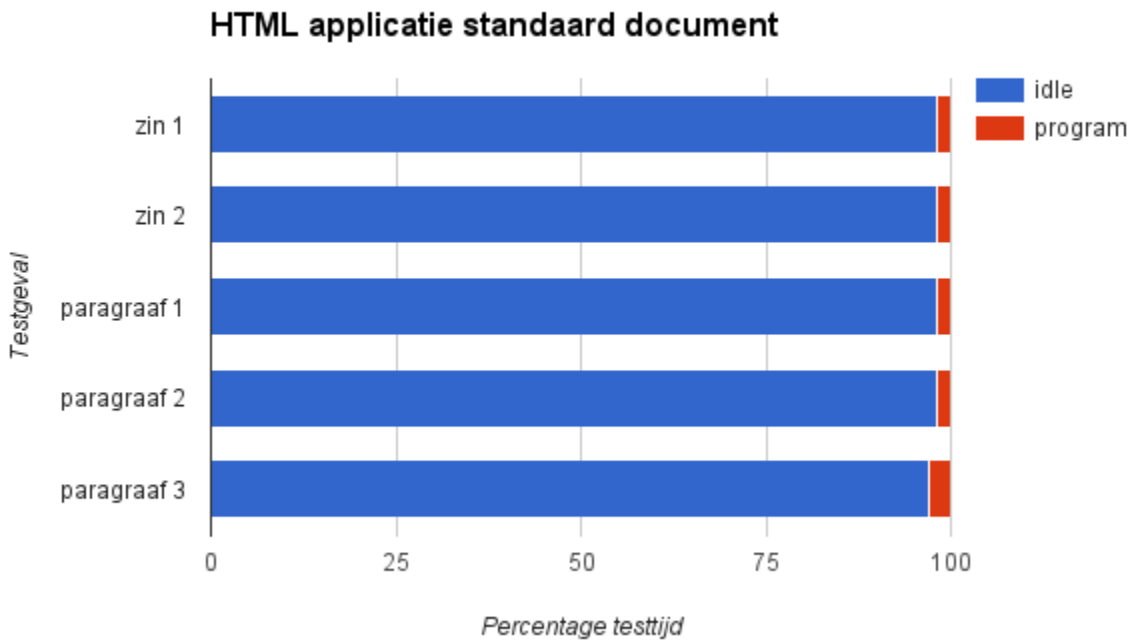
6.6 Baseline profile

De resultaten van de tests met de retail versie en de HTML versie van FontoXML vormen samen de baseline profile. Ik heb alle tests twee keer uitgevoerd en van deze resultaten het gemiddelde genomen. De tests heb ik niet vaker dan twee keer uitgevoerd, omdat de resultaten nooit meer dan 5% onderling verschilde in de HTML versie van de applicatie. In de resultaten van de performancetests met de retail versie van FontoXML kwam het vijf keer voor dat het verschil groter was dan 5%. Echter waren deze resultaten minder belangrijk voor het onderzoek omdat deze resultaten door de uitgevoerde scripts een vertekend beeld geven.

6.6.1 Testresultaten standaard document

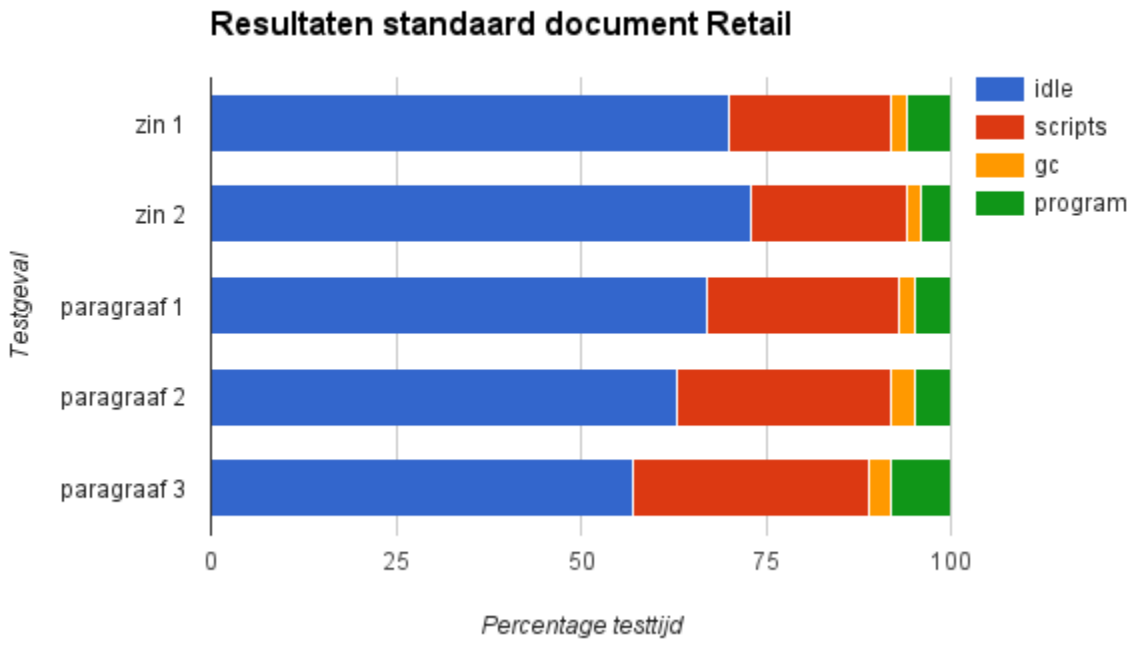
Uit de testresultaten voor het bepalen van de baseline profile konden al een aantal dingen opgemaakt worden. Tijdens de tests met het standaard document was de browser in zowel de retail versie als de HTML versie van de applicatie grotendeels inactief. Zichtbaar performanceverlies definieer ik als een zichtbare vertraging tussen het invoeren van een karakter en het daadwerkelijk verschijnen van het karakter op het scherm.

Het onderstaande diagram laat de testresultaten van de HTML versie van de applicatie met het standaard document zien. Tijdens het uitvoeren van deze tests was er geen zichtbaar performanceverlies. De browser was tijdens deze tests hoofdzakelijk inactief.



Figuur 2: Testresultaten baseline profile HTML applicatie; standaard document

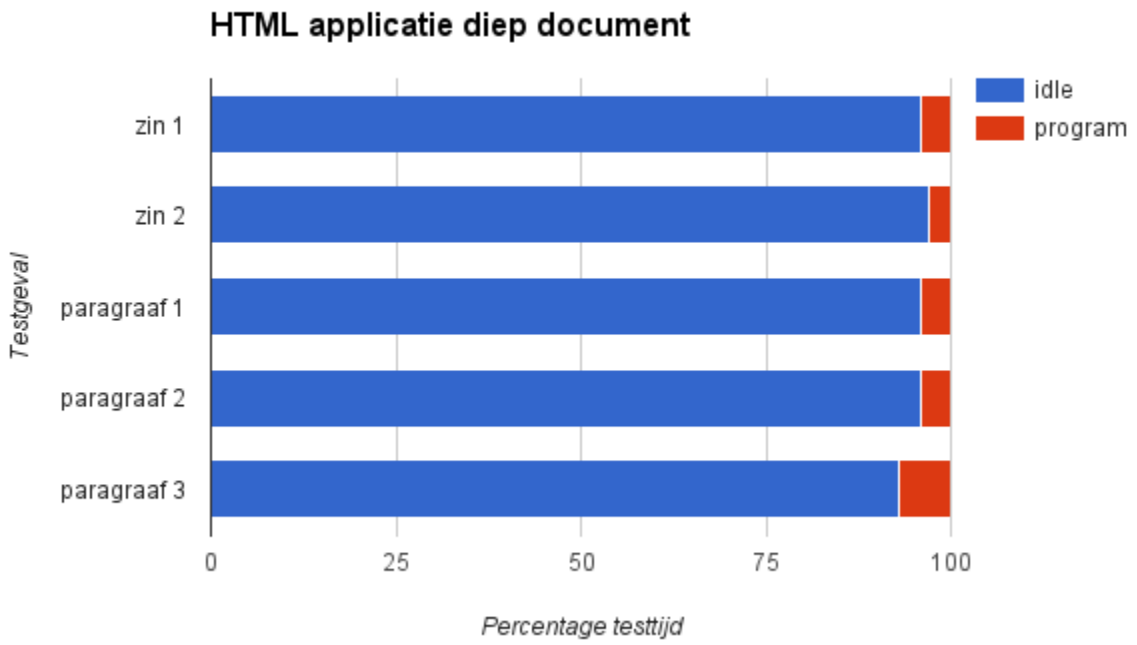
Het onderstaande diagram laat de testresultaten van de retail versie van de applicatie met het standaard document zien. Tijdens het uitvoeren van deze tests was er geen zichtbaar performanceverlies. De browser was tijdens deze tests minder lang inactief vergeleken met de tests met de HTML applicatie in hetzelfde document. Tijdens deze tests zorgden de scripts van de applicatie hiervoor.



Figuur 3: Testresultaten baseline profile retail applicatie: standaard document

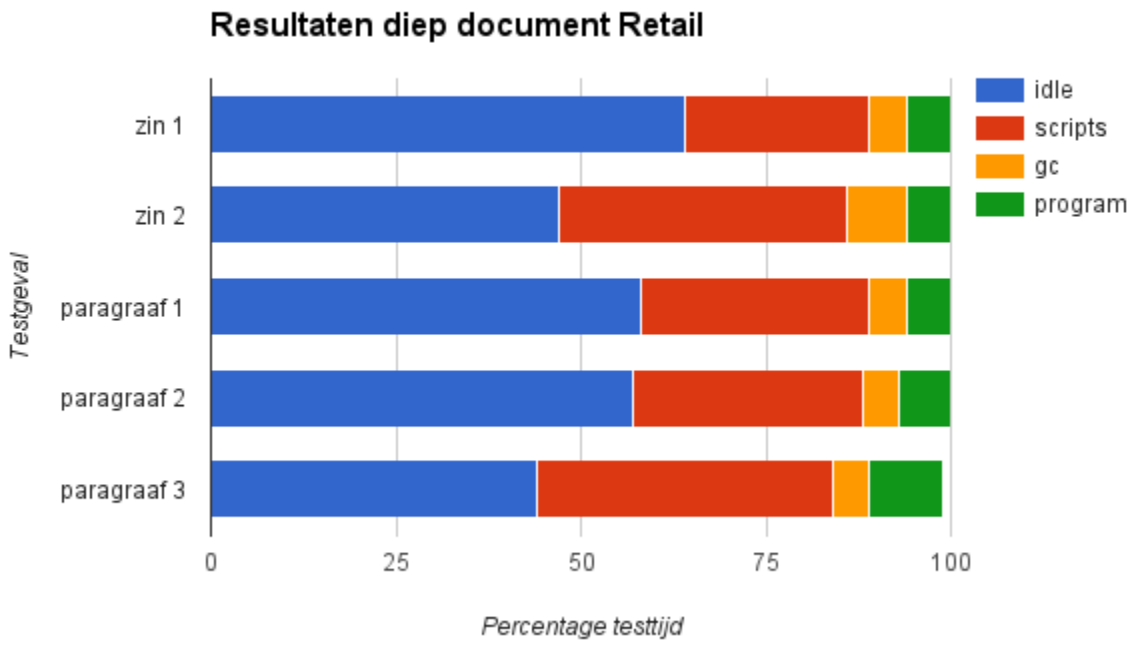
6.6.2 Testresultaten diep document

Het onderstaande diagram laat de testresultaten van de HTML versie van de applicatie met het diepe document zien. Tijdens het uitvoeren van deze tests was er geen zichtbaar performanceverlies. De browser was tijdens deze tests hoofdzakelijk inactief.



Figuur 4: Testresultaten baseline profile HTML applicatie; diep document

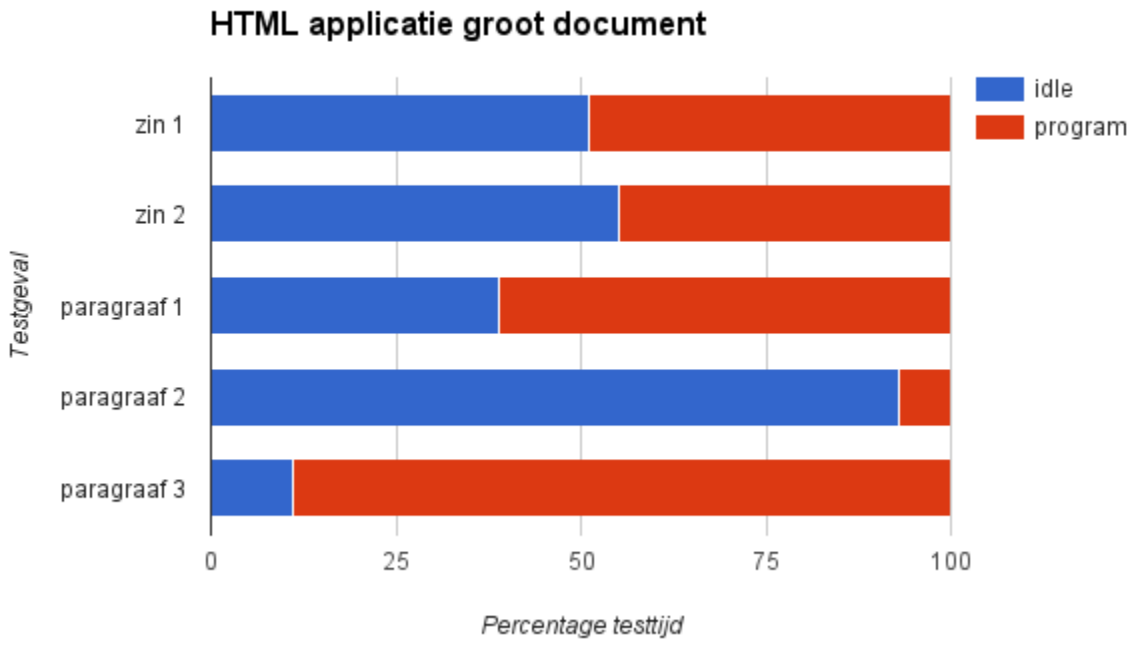
Het onderstaande diagram laat de testresultaten van de retail versie van de applicatie met het diepe document zien. Tijdens het uitvoeren van deze tests was er geen zichtbaar performanceverlies. De browser was tijdens deze tests minder lang inactief vergeleken met de tests met de HTML applicatie in hetzelfde document. Tijdens deze tests zorgden de scripts van de applicatie hiervoor.



Figuur 5: Testresultaten baseline profile retail applicatie; diep document

6.6.3 Testresultaten groot document

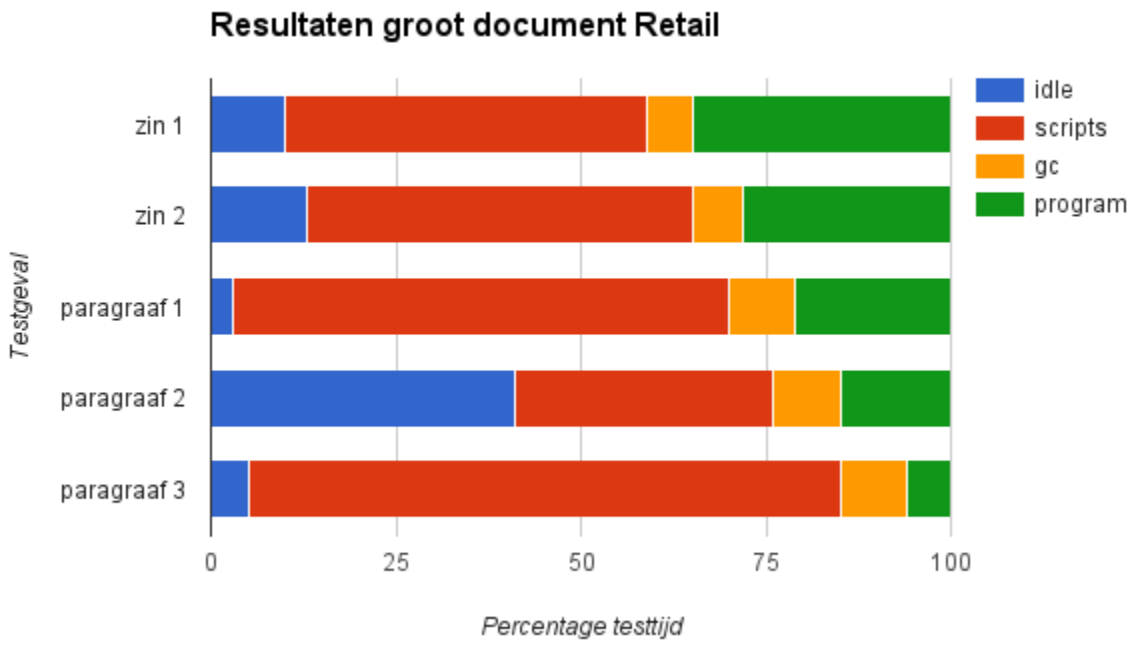
Het onderstaande diagram laat de testresultaten van de HTML versie van de applicatie met het grote document zien. Tijdens het uitvoeren van deze tests was er een goed zichtbaar performanceverlies.



Figuur 6: Testresultaten baseline profile HTML applicatie; groot document

Aan deze testresultaten is goed te zien dat de browser een groot deel van de totale testtijd besteedt aan het renderen van de webpagina. Opvallend is echter het testresultaat van testgeval paragraaf 2. Tijdens deze test was er, vergeleken met de overige tests, geen performanceverlies zichtbaar. Dit komt overeen met de bovenstaande testresultaten

Het onderstaande diagram laat de testresultaten van de retail versie van de applicatie met het grote document zien. Tijdens het uitvoeren van deze tests was er een goed zichtbaar performanceverlies. Testgeval paragraaf 2 valt net als bij de tests met de HTML applicatie in dit document op door de significant grotere tijd waarin de browser inactief is.



Figuur 7: Testresultaten baseline profile retail applicatie; groot document

7 Onderzoek

Het onderzoek begon ik met een vooronderzoek. Tijdens het vooronderzoek heb ik mij verder ingelezen in de werking van browsers en optimalisaties op het gebied van webpagina's. Op basis van de tijdens het vooronderzoek opgedane kennis heb ik het onderzoeksplan kunnen opstellen en het onderzoek kunnen indelen in fasen.

7.1 Onderzoeksplan

Aan de hand van het vooronderzoek heb ik het onderzoeksplan opgesteld. Hierin heb ik de aanleiding van het onderzoek, de probleemstelling van het onderzoek en de doelstelling van het onderzoek opgenomen. Daarnaast heb ik in het onderzoeksplan de bevindingen van het vooronderzoek vastgelegd.

Het volgende onderdeel van het onderzoeksplan is het onderzoeksontwerp. In het onderzoeksontwerp heb ik de afbakening van het onderzoek opgenomen. Uit het vooronderzoek bleek dat optimalisaties voor webpagina's in te delen zijn in vier categorieën: netwerk & webserver, Javascript, HTML en CSS. Het onderzoek heb ik afgebakend om het onderzoek alleen te richten op optimalisatiestrategieën op het gebied van HTML en CSS.

Het onderzoek heb ik niet gericht op het optimaliseren van het XML-document. Het XML-document is niet te optimaliseren, tenzij uit dit document content wordt verwijderd. Dit is niet wenselijk omdat hierdoor informatie verloren gaat.

Als laatste heb ik ook een planning opgenomen in het onderzoeksplan. De planning voor het onderzoek is als volgt ingedeeld:

Activiteit	Duur
Vooronderzoek + onderzoeksplan	6 dagen
Deelvraag 1	2 dagen
Deelvraag 2	5 dagen
Deelvraag 3	3 dagen
Conclusie	2 dagen

7.2 Onderzoeksrapport

De hoofdvraag van dit onderzoek luidt: "Welke optimalisaties op het gebied van het renderen van een webapplicatie hebben een positief effect op de performance van FontoXML?".

Bij deze hoofdvraag heb ik de volgende deelvragen gesteld:

- Deelvraag 1: Waardoor wordt het performanceverlies in FontoXML momenteel veroorzaakt?
- Deelvraag 2: Welke optimalisatiestrategieën kunnen worden toegepast op het gebied van HTML en CSS?
- Deelvraag 3: Welke optimalisatiestrategieën zullen (de meeste) performancewinst opleveren?

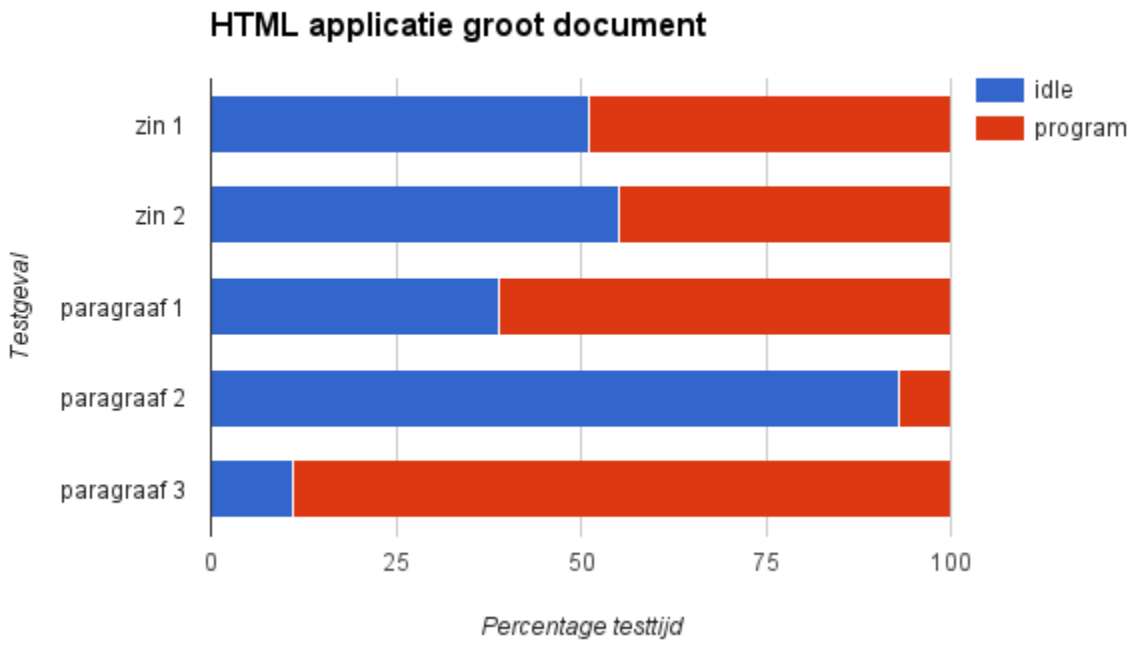
Omdat zeer weinig informatie beschikbaar is over optimalisaties van HTML en CSS en het toepassen van deze optimalisaties, is naast een literatuuronderzoek ook een experimenteel onderzoek uitgevoerd.

De eerste deelvraag is beantwoord aan de hand van de baseline profile. De tweede deelvraag heb ik beantwoord aan de hand van een literatuuronderzoek naar optimalisatiestrategieën op het gebied van HTML en CSS. De laatste deelvraag is beantwoordt aan de hand van een experimenteel onderzoek. Tijdens het beantwoorden van deze deelvraag heb ik het proof of concept gebouwd op basis van de gevonden optimalisatiestrategieën die ik gevonden had tijdens het beantwoorden van deelvraag 2.

7.2.1 Deelvraag 1: Waardoor wordt het performanceverlies momenteel veroorzaakt?

Om deze deelvraag te beantwoorden heb ik gekeken naar de resultaten van de performancetests die ik heb uitgevoerd voor het bepalen van de baseline profile. Daarnaast heb ik deze deelvraag beantwoord aan de hand van bronnen.

Tijdens het uitvoeren van de performancetests met het grote document was de browser een groot deel van de totale testtijd bezig met het opnieuw renderen van de webpagina. Dit is duidelijk te zien aan het diagram van de performancetests met de HTML versie van de applicatie, welke hieronder te zien is.

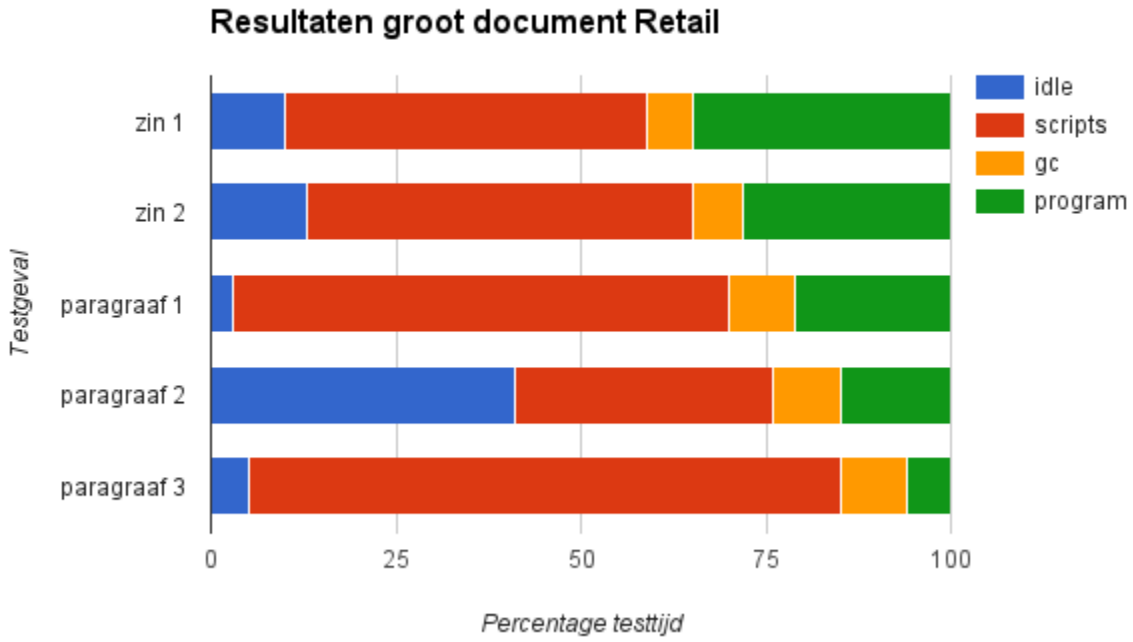


Figuur 8: Testresultaten baseline profile HTML applicatie; groot document

Opvallend aan het bovenstaande diagram is het testresultaat van testgeval paragraaf 2. Tijdens de tests met testgeval paragraaf 2 werd een paragraaf tekst ingevoerd aan het eind van het document. Tijdens deze test had de browser duidelijk veel minder tijd nodig voor het renderen van de webpagina dan tijdens het uitvoeren van testgeval paragraaf 1, waarbij een paragraaf aan het begin van het document wordt ingevoerd.

Een browser rendert een webpagina opnieuw wanneer de lay-out een element op de pagina veranderd. Wanneer tekst wordt ingevoerd in een paragraaf-element, verandert de grootte van het element. Hierdoor moet de browser opnieuw de positie en de grootte van het paragraaf-element opnieuw berekenen, maar ook de positie van alle onderliggende elementen. Tijdens testgeval paragraaf 1 moet de browser na elke toetsaanslag de positie van alle onderliggende elementen opnieuw berekenen. Dit hoeft de browser vanzelfsprekend niet te doen tijdens het uitvoeren van testgeval paragraaf 2, omdat de paragraaf aan het eind van het document wordt ingevoerd en onder deze paragraaf zich geen andere elementen meer bevinden.

Hetzelfde was in mindere mate terug te zien in de testresultaten met de retail versie van de applicatie. Tijdens het uitvoeren van de tests met de retail versie van de applicatie was de browser naast het renderen van de pagina ook bezig met het uitvoeren van scripts. Ondanks dat de browser ook de scripts uit moest voeren, is duidelijk te zien dat testgeval paragraaf 2 sneller uitgevoerd kon worden.



Figuur 9: Testresultaten baseline profile retail applicatie: groot document

7.2.2 Deelvraag 2: Welke optimalisatiestrategieën kunnen worden toegepast op het gebied van HTML en CSS?

Om deze deelvraag te beantwoorden heb ik eerst onderzocht hoe de browser omgaat met HTML- en CSS-bestanden nadat deze gedownload zijn van de webserver. Dit heb ik gedaan om een beter inzicht te krijgen in de werking van de browser. Deze kennis heb ik later kunnen gebruiken om een beter inzicht te krijgen in de werking van de optimalisatiestrategieën. Hierdoor heb ik een betere inschatting kunnen maken van de effectiviteit van de optimalisatiestrategieën.

Deze deelvraag heb ik opgedeeld in de volgende drie analysevragen:

- Hoe gaat de browser om met HTML- en CSS-bestanden?
- Hoe kan de HTML van een webpagina worden geoptimaliseerd?
- Hoe kan de CSS van een webpagina worden geoptimaliseerd?

7.2.2.1 Hoe gaat de browser om met HTML- en CSS-bestanden?

Nadat de browser de HTML- en CSS-bestanden opgehaald heeft van de webserver worden de bestanden verwerkt door de HTML parser of de CSS parser.

De HTML parser zet het HTML-bestand om in een HTML DOM (Document Object Model) Tree. De HTML DOM Tree is, zoals de naam al zegt, een boomstructuur. Deze boomstructuur wordt opgebouwd aan de hand van de in het HTML-bestand gedefinieerde elementen en opgeslagen in het DOM object van de browser.

De CSS parser zet de CSS-bestanden om in een CSSOM (Cascading Style Sheet Object Model) Tree. Deze boomstructuur bevat alle stijlregels die toegepast moeten worden op de elementen die zich de DOM Tree bevinden.

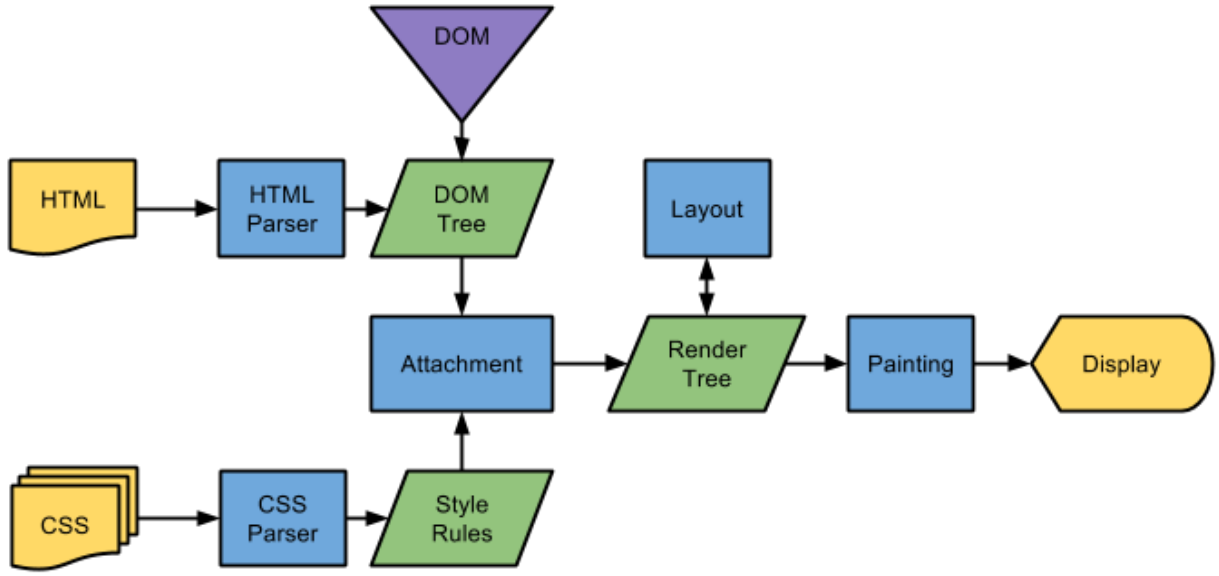
Na het opbouwen van de HTML DOM Tree en de CSSOM Tree wordt de Render Tree opgebouwd. De Render Tree bevat alleen de zichtbare elementen van een webpagina. De Render Tree wordt gebruikt tijdens het renderen van de pagina.

Het renderen van een webpagina bestaat uit twee stappen. In de eerste stap wordt de lay-out van de pagina berekend tijdens een zogenaamde (re)flow. Tijdens een (re)flow wordt het formaat en de positie van alle zichtbare elementen op de webpagina (opnieuw) berekend. Daarna wordt een (re)paint uitgevoerd. Tijdens de (re)paint van een webpagina worden alle elementen uit de render tree (opnieuw) getekend. Hierna kan de webpagina op het scherm worden getoond.

Wanneer er een wijziging in het formaat of de positie van een element in de DOM Tree plaatsvindt, wordt er een weer reflow uitgevoerd. Na een reflow volgt altijd een repaint van de webpagina. Dit gebeurt omdat de lay-out van de pagina niet meer overeen komt met wat op het scherm staat.

Wanneer alleen een visuele eigenschap wordt gewijzigd, zoals de tekstkleur of achtergrondkleur van een element, wordt alleen een repaint uitgevoerd. Hier hoeft geen reflow uitgevoerd te worden, omdat het wijzigen van een visuele eigenschap geen invloed heeft op de positie of het formaat van andere elementen in de Render Tree.

Het onderstaande stroomdiagram geeft het hiervoor besproken proces weer (Garsiel, 2011):



Figuur 10: Proces van het opbouwen van een webpagina

7.2.2.2 Hoe kan de HTML van een webpagina worden geoptimaliseerd?

Voor het optimaliseren van de HTML DOM heb ik een tweetal optimalisatiestrategieën gevonden. De eerste optimalisatiestrategie is het minimaliseren van de grootte van de HTML DOM. Het minimaliseren van de HTML DOM komt neer op het verminderen van het aantal aanwezige elementen in een HTML-bestand (Simon, 2015).

De tweede optimalisatiestrategie is het minimaliseren van de diepte van de HTML DOM. Het minimaliseren van de HTML DOM komt neer op het verminderen van het aantal geneste elementen in een HTML-bestand.

Beide optimalisatiestrategieën komen neer op het minimaliseren van de hoeveelheid elementen in de HTML DOM. Dit zijn, gezien de manier waarop de browser de HTML DOM opbouwt en gebruikt, logische optimalisaties.

7.2.2.3 Hoe kan de CSS van een webpagina worden geoptimaliseerd?

Voor het optimaliseren van de CSS van een webpagina heb ik eveneens twee optimalisatiestrategieën gevonden. Deze optimalisatiestrategieën hebben betrekking op de selector van een CSS stijlregel. Selectors worden gebruikt om HTML-elementen te selecteren. Wanneer een HTML-element voldoet aan de eisen van een selector wordt de stijl die is gedefinieerd in de stijlregel toegepast op dat HTML-element.

De eerste optimalisatiestrategie is het gebruiken van de meest efficiënte CSS selector. CSS selectors zijn te sorteren op efficiëntie. In onderstaande lijst zijn de CSS selectors van meest efficiënt tot het minst efficiënt te zien (Roberts, 2011) (Coyier, 2010):

- ID selector: `#header`
- Class selector: `.item`
- Type selector: `div`
- Adjacent selector: `h1 + p`
- Child selector: `ul > li`
- Descendant selector: `ul a`
- Universal selector: `*`
- Attribute selector: `[type="text"]`
- Pseudo-classes: `a:hover`

De tweede optimalisatiestrategie is het vermijden van complexe CSS selectors. Hieronder is een voorbeeld te zien van een simpele CSS selector en een complexe CSS selector. De eerste selector is simpel en selecteert alle div-elementen op een pagina. De tweede selector is complex en selecteert alle p-elementen welke een directe child zijn van een div-element en niet de eerste child van dat div-element zijn. Echter zijn nog veel complexere CSS selectors mogelijk.

CSS selector

```
div {
  color: green;
  border: 1px solid black;
}

div > p:not(:first-child) {
  color: red;
  font-weight: bold;
}
```

7.2.2.4 Conclusie

Tijdens het beantwoorden van de analysevragen van deze deelvraag heb ik niet veel optimalisatiestrategieën op het gebied van HTML en CSS kunnen vinden.

Op het gebied van HTML heb ik twee optimalisatiestrategieën kunnen vinden. De eerste optimalisatiestrategie is het minimaliseren van grootte de HTML DOM. De tweede optimalisatiestrategie is het minimaliseren van de diepte van de HTML DOM.

Op het gebied van CSS heb ik eveneens twee optimalisatiestrategieën kunnen vinden. De eerste optimalisatiestrategie is het gebruik maken van de meest efficiënte selector. De tweede optimalisatiestrategie is het vermijden van complexe selectors.

Browsers zijn tegenwoordig zodanig goed geoptimaliseerd dat het richten op optimalisaties op het gebied van CSS zeer weinig performancewinst zal opleveren (Sullivan, 2011) (Coyier, 2010). Hierdoor valt het optimaliseren van de CSS-bestanden van de applicatie af.

Het minimaliseren van de HTML DOM is moeilijk uit te voeren omdat een vast aantal elementen altijd nodig blijft om de pagina van een layout te voorzien. Daarnaast is het ook niet te voorkomen dat een gebruiker van de applicatie meerdere geneste elementen zal aanmaken. Hierdoor valt deze optimalisatiestrategie eveneens af.

Het minimaliseren van de grootte HTML DOM blijft als enige optimalisatiestrategie over om te kunnen verwerken in het proof of concept. Deze optimalisatiestrategie sluit ook aan bij de oplossingsrichting welke het ontwikkelteam van FontoXML zelf al had opgesteld als mogelijke optimalisatiestrategie. Deze optimalisatiestrategie is het toepassen van HTML DOM Culling.

7.2.3 Deelvraag 3: Welke optimalisatiestrategieën zullen (de meeste) performancewinst opleveren?

Omdat uiteindelijk maar één optimalisatiestrategie uit de vorige deelvraag is gekomen heb ik in deze deelvraag het proof of concept uitgewerkt waarin HTML DOM Culling is toegepast. Hiervoor heb ik een culling-algoritme moeten ontwikkelen welke de juiste HTML-elementen kan selecteren om te verwijderen en weer terug te plaatsen in de HTML DOM.

7.2.3.1 Conclusie

De conclusie van deze vraag en de hoofdvraag zijn geformuleerd in het adviesrapport aan de hand van de uitkomst van de performancetests met het proof of concept.

8 Bouw proof of concept

Tijdens de bouw van het proof of concept heb ik gebruik gemaakt van de op SCRUM gebaseerde softwareontwikkelmethode. Daarom heb ik de bouw van het proof of concept ingedeeld in sprints. Per sprint heb ik één of meer user story's geïmplementeerd. Hieronder is een overzicht te zien van de user story's per sprint:

Sprint	User story
Sprint 1	• Elementen buiten de viewport markeren • Dynamisch elementen buiten de viewport markeren
Sprint 2	• Dynamisch elementen buiten de viewport markeren door middel van een binary search
Sprint 3	• Elementen buiten de viewport uit de HTML DOM verwijderen • Elementen in de HTML DOM terugplaatsen • Vervangen verwijderde elementen door een opvul-element om de scrollbar op de juiste lengte te houden
Sprint 4	• Bijhouden cache/structuur met de grootte van elementen
Sprint 5	• Culling-algoritme uitbreiden naar geneste elementen
Sprint 6	• Culling-algoritme inbouwen in de testdocumenten

De user story's van sprint 1 tot en met 3 heb ik in deze volgorde uit moeten voeren om toe te werken naar een werkend culling-algoritme. Een andere volgorde was hierin niet mogelijk omdat deze user story's voort bouwen op de vorige user story's. De user story's van sprint 4 en 5 hadden als enige verwisseld kunnen worden. De user story van sprint 6 heb ik vanzelfsprekend pas als laatste kunnen uitvoeren.

8.1 Sprint 1

In sprint 1 ben ik begonnen met het implementeren van het eerste onderdeel van het culling-algoritme. Het eerste onderdeel van het culling-algoritme is het zoekalgoritme wat verantwoordelijk is voor het vinden van de HTML-elementen die buiten de viewport van de browser liggen. De viewport van de browser is het gedeelte van de pagina dat direct zichtbaar is voor de gebruiker.

Om het zoekalgortime te kunnen implementeren heb ik allereerst een HTML-document gemaakt met daarin 50 div-elementen. Ieder div-element bevat een h1-element en een p-element met beide een stuk opvultekst.

Div-element
`<div>` `  <h1>DIV 1</h1>` ` <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut nec lorem vestibulum, viverra justo quis, mollis nibh. Donec posuere diam felis, non vehicula purus sodales non. Nulla congue, dui sed aliquet condimentum, turpis ante tincidunt mi, in porttitor mi ante vel est. Quisque a orci elementum velit eleifend facilisis. In congue convallis sapien ac rutrum. Cras laoreet sapien nulla, ut molestie nisi consequat ac. Donec fermentum ullamcorper nulla nec tristique. Morbi vulputate luctus felis vitae condimentum.</p>` `</div>`

Het doel van deze eerste sprint was het implementeren van het zoekalgoritme en het bekend raken met Javascript. Het daadwerkelijk cullen (verwijderen en terugplaatsen van HTML-elementen) heb ik in deze sprint beperkt tot het veranderen van de tekstkleur van de “gecullde” elementen. Hieraan was duidelijk te zien welke HTML-elementen buiten de viewport lagen. Daarnaast kon ik hiermee controleren of het zoekalgoritme correct werkte.

Om de implementatie simpel te houden begon ik met het implementeren van een lineair zoekalgortime. Het ontwerp van dit zoekalgortime heb ik tevens simpel gehouden. Onderstaand een stuk pseudocode wat ik gebruikt heb als ontwerp voor deze sprint:

Pseudocode zoekalgortime sprint 1

```
function inView (element)
  return (element.bottom > window.top) ||
         (element.top < window.bottom)

function cull ()
  foreach element in DOM.children
    if inView(element)
      element.textColor = red;
```

Het bovenstaande algoritme controleert voor elk element dat zich in de HTML DOM bevindt of deze buiten de viewport van de browser ligt. Dit doet het algoritme door te controleren of de onderkant van het element boven de bovenkant van de viewport ligt of de bovenkant van het element onder de onderkant van de viewport ligt.

8.2 Sprint 2

Tijdens sprint 2 heb ik het lineaire zoekalgoritme vervangen voor een binary search. Een binary search is in de meeste gevallen sneller dan een lineaire search. Dit is een groot voordeel wanneer ik het culling-algoritme toe ga passen op het testdocument. Het grote testdocument bestaat uit 600 elementen waar in dat geval doorheen gezocht moet worden. Dit in tegenstelling tot de 50 elementen van het document waarin ik het culling-algoritme ontwikkel.

Een van de eigenschappen van een binary search is dat deze search uit gaat van een gesorteerde lijst. Deze eigenschap is niet beperkend voor de toepassing waarvoor ik de binary search gebruik. Doordat alle elementen onder elkaar liggen, zijn de posities van deze elementen ten opzichte van de bovenkant van het scherm (nulpunt) te allen tijde gesorteerd.

Het verschil tussen een lineaire en binary search is de manier waarop door de elementen gezocht wordt. Zoals eerder uitgelegd loopt een lineair zoekalgoritme door alle elementen heen totdat het gezochte element is gevonden.

Een binary search begint niet aan het begin van een lijst, maar in het midden van een lijst. Het middelste element wordt vergeleken met het element waarnaar gezocht wordt. Is het element groter dan het middelste element, gaat de binary search verder met de rechter kan van de lijst (in het geval van een lijst met van links naar rechts oplopende getallen). Is het element kleiner dan het middelste element, gaat de binary search verder met de linker kant van de lijst.

Hieronder staat een voorbeeld van een lijst met de stappen die uitgevoerd worden tijdens het zoeken naar het getal 45:

Stap 1: Selecteer het middelste element van de lijst

5	10	15	20	25	30	35	40	45	50

Stap 2: Vergelijk het middelste element van de lijst: 45 is groter dan 25

5	10	15	20	25	30	35	40	45	50

Stap 3: Selecteer het middelste element van de overgebleven lijst

5	10	15	20	25	30	35	40	45	50

Stap 4: Vergelijk het middelste element van de lijst: 45 is groter dan 40

5	10	15	20	25	30	35	40	45	50

Stap 5: Selecteer het middelste element van de overgebleven lijst

5	10	15	20	25	30	35	40	45	50

Stap 6: Vergelijk het middelste element van de lijst: 45 is gelijk aan 45

5	10	15	20	25	30	35	40	45	50

Het implementeren van de binary search duurde één dag langer dan deze sprint. Dit kwam doordat ik op de laatste dag tegen een bug aanliep waardoor de binary search in sommige gevallen in een oneindige recursie terecht kwam. Daardoor heb ik deze user story afgemaakt in de volgende sprint.

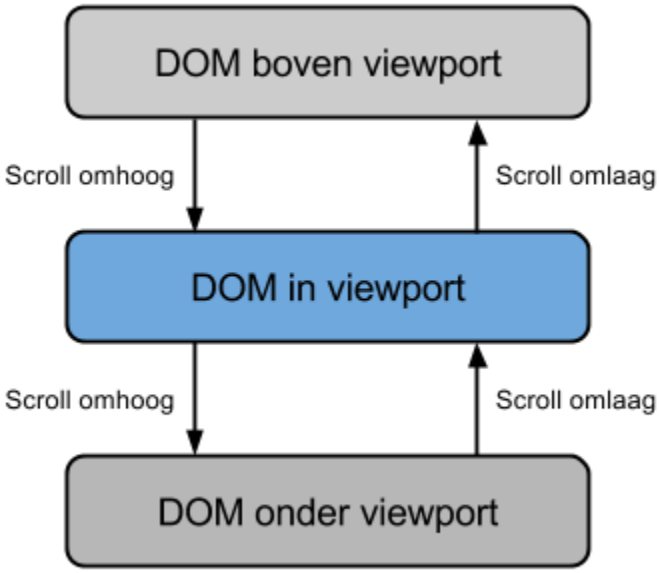
8.3 Sprint 3

In sprint 3 heb ik culling toegepast. Hiervoor moest ik een manier vinden om de verwijderde elementen op te slaan om deze elementen later weer terug te kunnen zetten in de HTML DOM.

Ik had twee verschillende opties voor het opslaan van de HTML-elementen buiten de DOM Tree van de browser. De eerste optie was het bijhouden van twee stukken DOM in het geheugen van de browser. Deze twee stukken DOM zouden de HTML DOM van boven de viewport en van onder de viewport moeten bevatten.

Wanneer een gebruiker in dit geval scrollt, moeten er twee bewerkingen uitgevoerd worden. De eerste bewerking is het verplaatsen van de HTML-elementen die buiten de viewport vallen. Deze HTML-elementen moeten verplaatst worden naar de juiste DOM in het geheugen van de browser, afhankelijk van de scrollrichting. De tweede bewerking is het verplaatsen van de HTML-elementen die zichtbaar worden door het scrollen van de gebruiker naar de DOM Tree van de browser. Deze HTML-elementen moeten verplaatst worden van de juiste DOM in het geheugen naar de DOM Tree van de browser. Deze bewerking is ook weer afhankelijk van de scrollrichting.

Het bovenstaande proces wordt geïllustreerd door de volgende afbeelding:



Figuur 11: Optie 1

Een voordeel van deze optie is dat de bewerkingen aan de DOM direct zichtbaar zijn voor de gebruiker. Dit komt doordat alle wijzigingen direct in de zichtbare DOM worden opgenomen.

Een nadeel aan deze optie is dat van een element meerdere kopieën zullen bestaan. Dit probleem treedt op wanneer een geneste structuur geculld wordt. Hieronder een voorbeeld om dit probleem te illustreren:

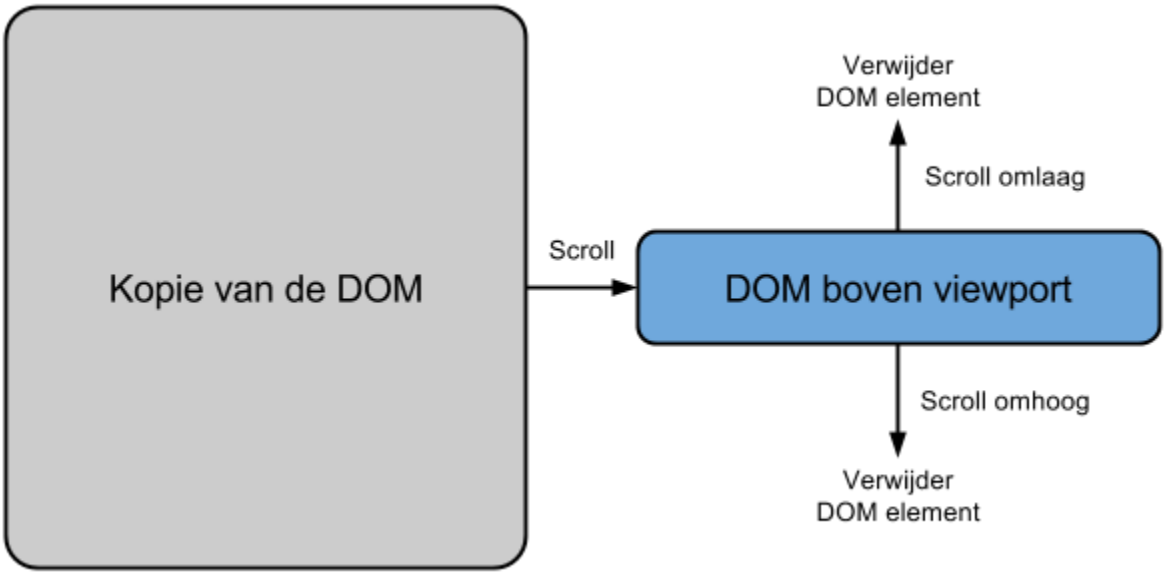
Originele DOM	DOM tijdens optie 1
DOM Tree: 	DOM boven viewport: DOM Tree: DOM onder viewport:

In de linker kolom is een normaal fragment van een DOM Tree te zien. Gegeven is dat het div-element met id="child2" de volledige viewport beslaat, waardoor de div-elementen met id="child1" en id="child3" buiten de viewport vallen. Hierdoor moeten deze twee div-elementen verplaatst worden van de DOM Tree van de browser naar een van beide DOM structuren in het geheugen van de browser. Wanneer dit gebeurt moeten van het div-element met id="parent" drie verschillende kopieën bestaan om de DOM in zowel de DOM Tree als de DOM in het geheugen kloppend te houden.

Doordat van verschillende elementen meerdere kopieën zullen bestaan wordt deze optie lastig om te implementeren. Daarnaast zal het bijwerken van de elementen lastiger worden omdat dit op drie plekken moet gebeuren.

De tweede optie was het hebben van een volledige kopie van de DOM Tree. Hierbij wordt na het inladen van het document een kopie gemaakt van de DOM Tree.

Wanneer een gebruiker scrollt, moeten er net als bij optie 1 twee bewerkingen uitgevoerd worden. De eerste bewerking is het verwijderen van de elementen uit de DOM Tree die buiten de viewport vallen. De tweede bewerking is het kopiëren van elementen die in de viewport komen te liggen uit de kopie van de DOM naar de DOM Tree. Dit wordt geïllustreerd door het volgende diagram:



Figuur 12: Optie 2

Het voordeel aan deze optie is dat er altijd een volledige kopie bestaat van de DOM. Hierdoor bestaan alleen de elementen die op dat moment in de viewport liggen twee keer. Het verwijderen van elementen uit de viewport is ook makkelijker dan bij optie 1 omdat de elementen simpelweg verwijderd kunnen worden en niet verplaatst hoeven worden naar een andere DOM.

Een nadeel aan deze optie is dat de zichtbare DOM altijd geüpdatet moet worden wanneer de kopie van de DOM gewijzigd wordt.

Uiteindelijk heb ik gekozen voor optie 2. Deze optie is makkelijker te implementeren en zal beter werken wanneer het culling-algoritme geïmplementeerd zal worden in FontoXML. FontoXML zal met deze optie de DOM kunnen blijven bewerken zoals dit nu ook al gebeurt.

In deze sprint heb ik een aantal stappen doorlopen om deze user story te implementeren. De eerste stap was het kopiëren van de HTML DOM en het verwijderen van de elementen die buiten de viewport lagen. Wanneer de pagina ingeladen wordt, wordt de pagina volledig gerendert. Na het renderen wordt een kopie gemaakt van de HTML DOM. Hierna wordt met behulp van de binary search gezocht naar het element wat op de bovengrens van de viewport ligt. Vanuit dit element wordt gezocht naar de elementen die in de viewport liggen. De elementen buiten de viewport worden hierna verwijderd, zodat alleen de zichtbare HTML-elementen overblijven. In deze sprint werkte deze functionaliteit alleen wanneer het eerste element bovenin de viewport lag. In een latere sprint heb ik ervoor gezorgd dat deze functionaliteit ook werkt wanneer het eerste element niet bovenin de viewport ligt.

De tweede stap was het toevoegen van een element onderaan de viewport wanneer naar beneden gescrollt wordt. Hiervoor moet gedetecteerd worden of er gescrollt wordt en of de onderkant van het laatste zichtbare element boven de onderkant van de viewport uitkomt. Als dit zo is moet het volgende element toegevoegd worden aan de DOM Tree van de browser. Dit element wordt gekopieerd vanuit de kopie van de originele HTML DOM.

De derde stap was het verwijderen van het element wat tijdens het naar beneden scrollen aan de bovenkant van de viewport uit het zicht verdwijnt. Hiervoor moet gecontroleerd worden of het bovenste element volledig buiten de viewport komt te liggen na het scrollen. Als dit zo is, moet het element verwijderd worden uit de HTML DOM.

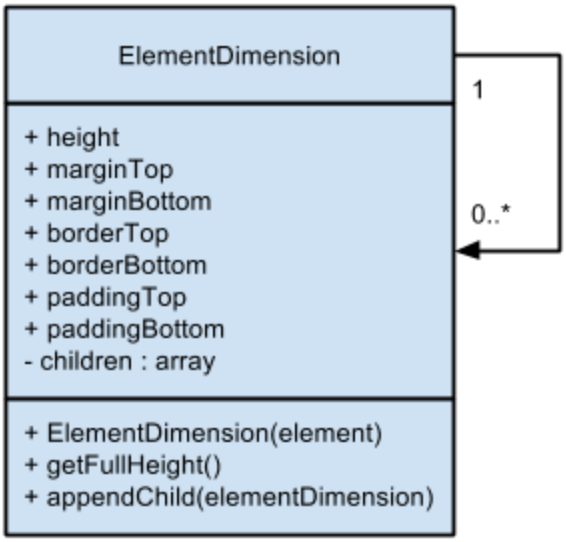
Om te voorkomen dat de elementen onder het verwijderde element naar boven springen door de vrijgekomen ruimte heb ik een opvul-element toegevoegd aan het document. Wanneer een element aan de bovenkant van de pagina verwijderd wordt, wordt net voor het verwijderen van het element de grootte van het element opgevraagd. De hoogte van het verwijderde element wordt toegevoegd aan de hoogte van het opvul-element. Hierdoor verspringen de elementen onder het verwijderde element niet en blijft de pagina even lang.

De vierde stap heb ik de functionaliteit van voorgaande stappen in tegengestelde richting geïmplementeerd. Hierdoor werd het ook mogelijk om naar boven te scrollen waarbij de elementen aan de bovenkant van de pagina toegevoegd werden en aan de onderkant verwijderd werden. Hiervoor heb ik ook een opvulelement aan de onderkant van de pagina toegevoegd.

8.4 Sprint 4

In sprint 4 heb ik een cachinglaag geïmplementeerd om de groottes van de elementen in op te slaan. Deze cachinglaag moet voorkomen dat de browser meerdere keren de grootte van een element moet opvragen. Het opvragen van de grootte van een element is duur omdat de browser de grootte van het element gaat herberekenen om de juiste waardes terug te kunnen geven (Cousineau, 2013).

Om de cache op te bouwen heb ik een algoritme geschreven welke door de boomstructuur van de DOM Tree loopt. Dit algoritme vraagt van elk element de grootte op en slaat deze op in de cache. De cache heeft na het opbouwen dezelfde structuur als de DOM Tree. Elk element in de DOM Tree heeft een bijbehorend cache-object.



Figuur 13: Ontwerp ElementDimension klasse

Onderstaand de pseudocode voor dit algoritme:

Pseudocode cache-algoritme

```
function buildDimensionCache(element)
  dimensionObject = new ElementDimension(element)

  if element.children.length > 0
    foreach child in element.children
      temp = buildDimensionCache(child)
      dimensionObject.appendChild(temp)

  return dimensionObject
```

Na het implementeren van het algoritme wat de cache opbouwt moest ik ervoor zorgen dat de cache daadwerkelijk gebruikt zou kunnen worden. In de code beschikte ik alleen over een verwijzing naar het HTML-element waarvan ik een grootte nodig had. Deze verwijzing moest ik gebruiken voor het vinden van het bijbehorende ElementDimension object. Hiervoor heb ik de volgende opties bedacht:

- Optie 1: Prototype van de Node-elementen uitbreiden met een verwijzing naar het bijbehorende cache-object
- Optie 2: Vertaal de positie van het DOM-element naar de positie van het bijbehorende cache-object

Optie 1 viel gelijk af. Het uitbreiden van de native prototype in Javascript wordt gezien als een slechte manier van programmeren. Daarnaast kan het uitbreiden van een native prototype voor problemen zorgen in browsers die dit niet goed ondersteunen (Allardice).

Optie 2 is de enige optie die over blijft. Ik heb de implementatie van deze optie uitgesteld naar de volgende sprint. Dit heb ik gedaan omdat dit uiteindelijk een onderdeel bleek van de manier waarop ik geneste culling zou gaan toepassen.

8.5 Sprint 5

In sprint 5 ben ik begonnen aan het implementeren van nested culling. Nested culling houdt in dat geneste elementen ook geculld kunnen worden. Tot deze sprint was het alleen mogelijk om te cullen op één niveau in het document.

Het implementeren van nested culling verliep moeizamer dan ik had verwacht. Dit kwam doordat ik moest bepalen welk element als volgende toegevoegd moest worden aan de HTML DOM. Om te bepalen welk element toegevoegd moest worden aan de HTML DOM heb ik een algoritme geschreven welke het volgende element in de HTML DOM zoekt en hiervan het pad als een array van indices teruggeeft. Het voordeel aan dit array is dat dit array ook gebruikt kan worden voor het opzoeken van het bijbehorende cache object.

Dit algoritme heb ik geïmplementeerd aan de hand van de volgende ontwerp in pseudocode:

Pseudocode

```
function getNextBottomElement(element, path)
  elementIndex = path.lastElement
  newPath = path

  if element.parent == null
    return []

  if elementIndex == element.parent.lastChildIndex
    newPath.pop

    return getNextBottomElement(element.parent, newPath)
  else
    newPath.pop
    newPath.push(elementIndex + 1)

    return
getNextBottomElementStep2(element.parent.children[elementIndex + 1],
newPath)

function getNextBottomElementStep2(element, path)
  newPath = path

  if element.children.length > 0
    newPath.push(0)

    return getNextBottomElementStep2(element.children[0], newPath)
  else
    return newPath
```

Het algoritme bestaat uit twee recursieve functies. Beide functies werken tijdens elke stap het pad naar het element bij. De eerste functie loopt terug naar de root van de DOM. Deze functie controleert allereerst of het meegegeven element een parent-element heeft. Wanneer het meegegeven element geen parent heeft, betekent dit dat de root van de DOM is bereikt en dat er geen volgend element is. Wanneer het de root van de DOM wordt bereikt, wordt een leeg array teruggegeven.

Wanneer het element wel een parent heeft, wordt gecontroleerd of het meegegeven element het laatste element van zijn parent is. Als dit zo is wordt dezelfde functie nog een keer aangeroepen. Hierdoor loopt het algoritme terug naar de root van de DOM. Als het element niet de laatste child van het parent-element is wordt de volgende child geselecteerd en meegegeven aan de tweede recursieve functie.

De tweede functie loopt weer verder van de root van de DOM weg om het eerstvolgende geschikte element te zoeken. Deze tweede functie controleert of het meegegeven element zelf children heeft. Als dit zo is wordt deze tweede functie opnieuw aangeroepen en wordt de child van het element meegegeven. Als het element zelf geen children heeft is het element zelf het laatste element. Als het laatste element gevonden is wordt het bijgewerkte pad teruggegeven.

Het array met indices kan gebruikt worden om het element op te zoeken in de DOM. Elke waarde in het array komt overeen met de index van de children van het parent element. Het element waarnaar verwezen wordt in het array kan worden gezocht met de volgende recursieve functie:

Pseudocode

```
function getElementFromPath(element, path)
  nextElement = element.children[path[0]]

  path.unshift()

  if path.lenght == 0
    return nextElement
  else
    return getElementFromPath(nextElement, path)
```

8.6 Sprint 6

In sprint 6 ben ik niet verder gegaan met het implementeren van de nested culling. Omdat dit de laatste sprint was heb ik ervoor gekozen om het implementeren van de nested culling achterwege te laten en mij in plaats daarvan te richten op het overzetten van het culling-algoritme naar de testbestanden.

De testbestanden bevatten naast de HTML-elementen ook een hoop HTML-comments. Deze comments worden door de applicatie geplaatst en gebruikt voor het bijhouden van data met betrekking tot de ingevoegde elementen.

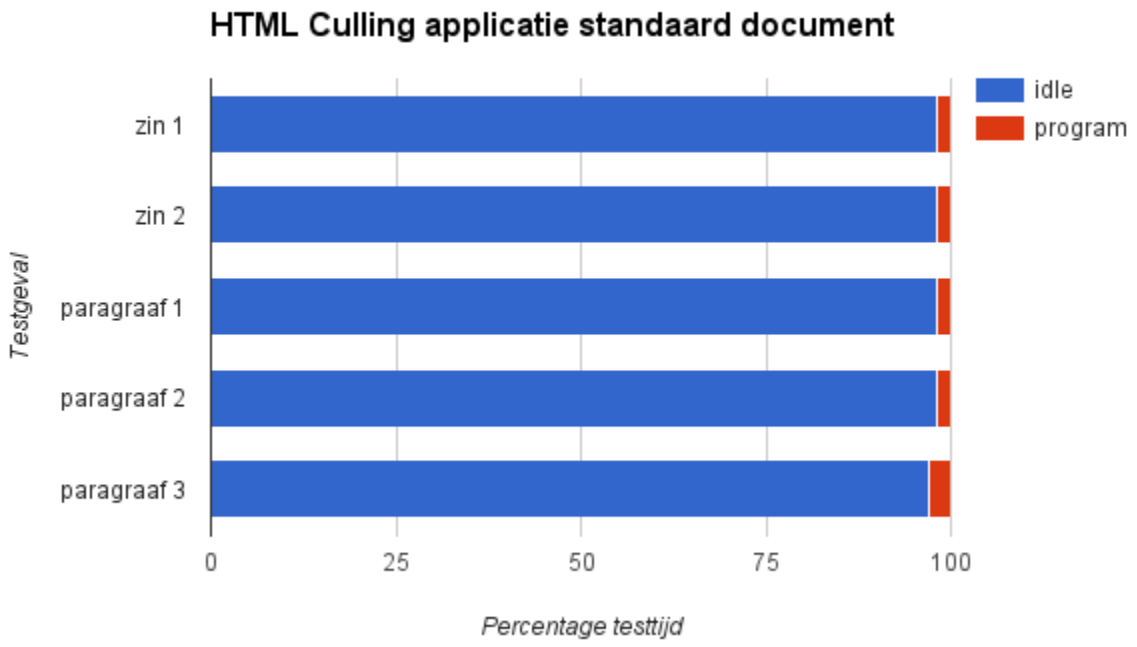
HTML-comments worden door de browser echter niet gezien als HTML-elementen, maar als nodes. Hierdoor werden de comments niet verwijderd uit de zichtbare DOM, waardoor de comments ophoopten. Dit zorgde ervoor dat de browser zeer traag werd. Dit probleem heb ik opgelost door de comments, net als de HTML-elementen, te cullen.

Nadat het cullen in de testdocumenten correct werkte, heb ik de performancetests opnieuw uitgevoerd.

9 Testrapport (proof of concept)

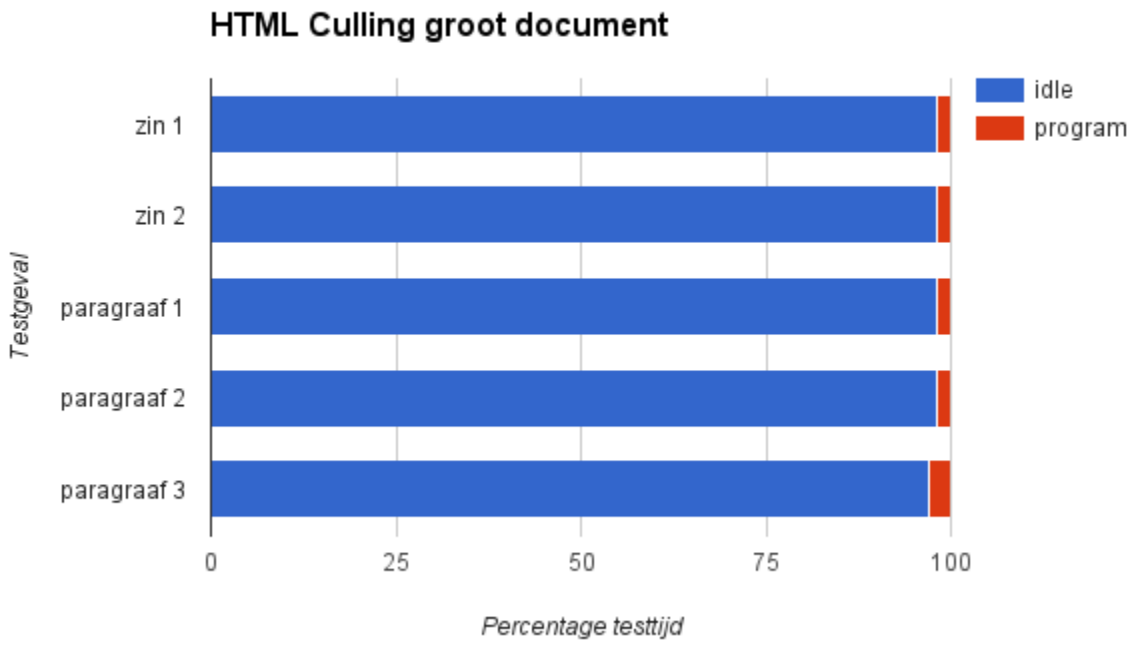
Na het afronden van het proof of concept, heb ik de eerder opgestelde performancetests uitgevoerd met het proof of concept. Ik heb de performancetests uitgevoerd met het standaard document en met het grote document. Het diep geneste document heb ik niet getest omdat ik de nested culling niet heb kunnen afronden. Het testen van culling op één niveau is niet nuttig in het diep geneste document, omdat in dit document op elk niveau maar 3 elementen zijn die geculld kunnen worden.

De resultaten van de performancetests met het standaard document in het proof of concept zijn hieronder te zien. De resultaten van deze performancetests zijn hetzelfde als de resultaten van de performancetests voor de baseline profile. Culling op het standaard document heeft dus geen effect op de performance.



Figuur 14: Testresultaten proof of concept; standaard document

De resultaten van de performancetests met het grote document in het proof of concept zijn hieronder te zien. De resultaten van deze performancetests zijn totaal anders dan de resultaten van de performancetests voor de baseline profile. Aan die resultaten was duidelijk te zien dat de browser lang bezig was met het renderen van de webpagina in tegenstelling tot onderstaande testresultaten.



Figuur 15: Testresultaten proof of concept: groot document

10 Adviesrapport

Als laatste heb ik het adviesrapport opgesteld. In dit rapport heb ik een advies uitgebracht over de te implementeren optimalisatie.

In het adviesrapport heb ik het advies gegeven om HTML DOM Culling toe te gaan passen in FontoXML. Dit advies kwam tot stand doordat HTML DOM Culling de enige nuttige optimalisatie bleek, waardoor er op het gebied van HTML en CSS geen alternatieven zijn.

11 Evaluatie aanpak en producten

In dit hoofdstuk evalueer ik de aanpak die ik tijdens de uitvoer van deze afstudeeropdracht heb gehanteerd. Daarnaast evalueer ik de producten die ik tijdens de uitvoer van deze afstudeeropdracht heb opgeleverd.

11.1 Planning

De planning die ik aan het begin van deze afstudeeropdracht gemaakt heb, heb ik voor het grootste gedeelte kunnen volgen. Voor het bepalen van de testmethode had ik meer tijd nodig, waardoor de planning in zijn geheeld opschoof. Daarnaast heb ik in de laatste drie weken net te weinig tijd ingepland voor het afronden van mijn afstudeerverslag. Een verbeterpunt op dit vlak is het toevoegen van een buffer aan de planning.

11.2 Performancetests en testrapport

De performancetests heb ik uitgevoerd in een virtuele machine in één browser. Om de testresultaten meer solide te maken had ik de tests liever uitgevoerd op meerdere besturingssystemen en in verschillende browsers. Dit is niet gelukt vanwege tijdsgebrek. Een verbeterpunt op dit vlak is het testen op meerdere besturingssystemen en browsers.

11.3 Onderzoek en onderzoeksrapport

Tijdens het onderzoek heb ik onderzoek gedaan naar de mogelijk optimalisatiestrategieën op het gebied van HTML en CSS. Het onderzoek had ik uitgebreider willen uitvoeren. Dit was echter niet mogelijk vanwege de beperkte hoeveelheid bronnen. Daarnaast was de afstudeerperiode te kort om naast het uitwerken van de HTML DOM culling ook uitgebreider onderzoek te doen naar CSS.

11.4 Bouw en ontwerp proof of concept

Als onderdeel van het onderzoek heb ik een proof of concept gebouwd. De bouw van dit proof of concept heb ik ingedeeld in sprints. Per sprint heb ik één of meer user story's geïmplementeerd. Het implementeren van het cullen van geneste elementen duurde langer dan ik had ingeschat. Ik had deze user story beter kunnen opsplitsen in twee of meer user story's. Hierdoor had ik de user story's effectiever kunnen plannen.

11.5 Beroepstaken

In dit hoofdstuk bespreek ik de beroepstaken die ik heb opgesteld voor mijn afstudeeropdracht en of ik deze beroepstaken behaald heb.

11.5.1 Beroepstaken afstudeerplan

Hieronder staan de beroepstaken zoals ik deze heb opgesteld in mijn afstudeerplan:

1.1 Selecteren methoden, technieken en tools

Deze beroepstaak wil ik binnen deze opdracht invullen door te onderzoeken welke optimalisatietechnieken er beschikbaar zijn en hoeveel performancewinst deze kunnen opleveren. Het is de bedoeling dat ik de technieken met de meeste performancewinst selecteer, zodat ik hierover een advies kan uitbrengen.

3.2 Ontwerpen systeemdeel

Deze beroepstaak wil ik binnen deze opdracht invullen door het ontwerpen van een proof of concept waarmee kan worden aangetoond dat de optimalisatietechniek daadwerkelijk werkt. Hierbij moet in het ontwerp rekening gehouden worden met de bestaande applicatie.

3.3 Bouwen applicatie

Deze beroepstaak wil ik binnen deze opdracht invullen door het bouwen van een proof of concept waarmee kan worden aangetoond dat de optimalisatietechniek daadwerkelijk werkt.

3.5 Uitvoeren van en rapporteren over het testproces

Deze beroepstaak wil ik binnen deze opdracht invullen door de performance van de bestaande situatie te vergelijken met de nieuwe situatie (proof of concept). Hiervoor dient de performance van de bestaande en de nieuwe situatie (namelijk die van het proof of concept) getest te worden. Hiermee kan worden aangetoond dat de optimalisatietechniek daadwerkelijk performancewinst oplevert en deze daarmee voldoet aan de eisen die aan de performance van de applicatie gesteld worden.

11.5.2 Evaluatie beroepstaken

In dit hoofdstuk evalueer ik de gekozen beroepstaken. De afstudeeropdracht is zelfstandig uitgevoerd. Dit betekent dat een beroepstaak op niveau 3 in een lastige context uitgevoerd moet worden en een beroepstaak op niveau 4 in een complexe context uitgevoerd moet worden.

	geleid	zelfstandig	sturend
simpel	1	2	3
lastig	2	3	4
complex	3	4	5

De definities van de beroepstaken komen uit het document Beroepstaken Informatica versie 1.1 van juli 2009.

1.1 Selecteren methoden, technieken en tools: niveau 4

Deze beroepstaak luidt in de complexe context als volgt:

> *"Er moet advies uitgebracht worden over de keuze(s) van methoden, technieken en tools in de verschillende omgevingen (OTAP), rekening houdend met twee of meer stakeholders."*

Deze beroepstaak heb ik ingevuld met het selecteren van een softwareontwikkelmethode. Daarnaast heb ik deze beroepstaak ingevuld met het uitbrengen van advies over de te toe te passen optimalisatiestrategie. Een optimalisatiestrategie kan gezien worden als een techniek.

3.2 Ontwerpen systeemdeel: niveau 4

Deze beroepstaak luidt in de complexe context als volgt:

> *"Het betreft het ontwerpen van een gedistribueerde applicatie of een applicatie met complexe algoritmiek. Er wordt rekening gehouden met niet-functionele kwaliteitsattributen en beveiligingsaspecten en gebruik gemaakt van design patterns."*

Deze beroepstaak heb ik ingevuld door het ontwerpen van het proof of concept. Voor het proof of concept zijn een aantal complexe algoritmes ontworpen. Tijdens het ontwerpen van het proof of concept het ik ook rekening gehouden met de bestaande applicatie bij de keuze van de manier waarop het culling-algoritme de niet zichtbare DOM zou bijhouden. Daarnaast is rekening gehouden met de niet functionele-kwaliteitsattribuut performance door een binary search in plaats van een lineaire search te implementeren.

3.3 Bouwen applicatie: niveau 3
Deze beroepstaak luidt in de lastige context als volgt:

"Het betreft het bouwen van een objectgeoriënteerde applicatie, waarbij geavanceerder concepten van de gebruikte programmeertaal aan de orde komen. Verder wordt rekening gehouden met toekomstige wijzigingen, testbaarheid en hergebruik. Het bouwen gebeurt in een ontwikkelomgeving."

Deze beroepstaak heb ik ingevuld met de bouw van het proof of concept. Hierbij heb ik gebruik gemaakt van Javascript. Tijdens de bouw heb ik rekening gehouden met toekomstige wijzigingen. Dit heb ik onder andere gedaan bij de implementatie van de cachelaag voor de HTML-elementen. Het bouwen heb ik gedaan in Sublime Text.

3.5 Uitvoeren van en rapporteren over het testproces: niveau 3
Deze beroepstaak luidt in de lastige context als volgt:

"Bij het opstellen van een logisch testontwerp wordt gebruik gemaakt van een testontwerptechniek. Er is aandacht voor de herhaalbaarheid van de testen. Het betreft hoofdzakelijk het testen van de functionaliteit. Testrapportage betreft het volledige systeem."

Deze beroepstaak heb ik ingevuld met het bepalen van de testmethode voor de performancetests. Hierbij heb ik rekening gehouden met de herhaalbaarheid van de tests. Ik heb echter geen functionaliteit getest maar de performance van de applicatie. De performance van de applicatie is een niet-functionele kwaliteitsattribuut. Het testen van een niet-functionele kwaliteitsattribuut is onderdeel van deze beroepstaak in de complexe context.

12 Bibliografie

A vocabulary and associated APIs for HTML and XHTML. (sd). Opgeroepen op 2015, van W3C: http://www.w3.org/TR/2008/WD-html5-20080610/editing.html

Allardice, J. (sd). *Extending prototype of native object: '{a}'*. Opgeroepen op 2015, van JSLint Error Explanations: https://jslinterrors.com/extending-prototype-of-native-object

AutoHotkey. (sd). Opgeroepen op 2015, van AutoHotkey: http://www.autohotkey.com/

Bithell, M. (2014, November 5). *Understanding the Importance of Frame Rate*. Opgeroepen op 2015, van IGN: http://www.ign.com/articles/2014/11/05/understanding-frame-rate-and-its-importance

Chrome DevTools Overview. (sd). Opgeroepen op 2015, van Chrome: https://developer.chrome.com/devtools

Cousineau, D. (2013, September 3). *High Performance JS Tip: Dimensions Are Not Your Friend*. Opgeroepen op 2015, van dcousineau.com: http://dcousineau.com/blog/2013/09/03/high-performance-js-tip/

Coyier, C. (2010, Mei 24). *Efficiently Rendering CSS*. Opgeroepen op 2015, van CSS-Tricks: https://css-tricks.com/efficiently-rendering-css/

Firefox Developer Tools. (sd). Opgeroepen op 2015, van Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Tools

Garsiel, T. (2011, Augustus 5). *How Browsers Work: Behind the scenes of modern web browsers*. Opgeroepen op 2015, van HTML5 Rocks: http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/

Internet Explorer F12 Tools. (sd). Opgeroepen op 2015, van Microsoft Developer Network: https://msdn.microsoft.com/library/bg182326%28v=vs.85%29

Opera Dragonfly. (sd). Opgeroepen op 2015, van Opera: http://www.opera.com/dragonfly/

Roberts, H. (2011, September 17). *Writing efficient CSS selectors*. Opgeroepen op 2015, van CSS Wizardry: http://csswizardry.com/2011/09/writing-efficient-css-selectors/

Safari for Developers. (sd). Opgeroepen op 2015, van Apple: https://developer.apple.com/safari/tools/

Simon, L. (2015, April 8). *Minimizing browser reflow* . Opgeroepen op 2015, van Google Developers: https://developers.google.com/speed/articles/reflow

Stanley, D. (2014, November 14). *Understanding Frame Rate – Uncovering The Truth Behind 30 VS 60 FPS*. Opgeroepen op 2015, van Technology X: http://www.technologyx.com/featured/understanding-frame-rate-look-truth-behind-30v60-fps/2/

Sullivan, N. (2011, December 29). *CSS Selector Performance has changed! (For the better)*. Opgeroepen op 2015, van Performance Calendar: http://calendar.perfplanet.com/2011/css-selector-performance-has-changed-for-the-better/

Terkaly, B. (2013, Januari 14). *Windows 8 is definitely faster than Windows 7*. Opgeroepen op 2015, van Microsoft Developer Network Blogs: http://blogs.msdn.com/b/brunoterkaly/archive/2013/01/14/windows-8-is-definitely-faster-than-windows-7.aspx

Typetest. (sd). Opgeroepen op 2015, van Ticken: https://www.ticken.nl/Typecursus/Typetest.html

Walton, S. (2012, Augustus 15). *Windows 8 vs. Windows 7 Performance*. Opgeroepen op 2015, van Techspot: www.techspot.com/review/561-windows8-vs-windows7/page2.html

Words per minute. (sd). Opgeroepen op 2015, van Wikipedia: http://en.wikipedia.org/wiki/Words_per_minute

13 Begrippenlijst

In deze begrippenlijst staat een groot deel van de in dit verslag gebruikte begrippen uitgelegd.

SGML (http://www.isgmlug.org/**)**
SGML staat voor Standard Generalized Markup Language. SGML is een zogenaamde metataal. Dit is een taal waarmee een andere gestructureerde taal wordt beschreven.

XML (http://www.w3.org/XML/**)**
XML staat voor Extensible Markup Language. XML is een subset van SGML. XML wordt gebruikt voor het omschrijven van gestructureerde data en is daarnaast ook leesbaar voor mensen.

XML Schema (http://www.w3.org/XML/Schema**)**
Een XML Schema wordt gebruikt om de structuur, inhoud en semantiek van een XML document te definiëren. Een XML Schema kan daardoor gebruikt worden om een XML document te valideren.

DITA (http://dita.xml.org/book/about-dita**,** http://dita.xml.org/**)**
DITA staat voor Darwin Information Typing Architecture. DITA definieert een XML structuur voor het ontwerpen, schrijven, beheren en publiceren van informatie in geprinte en digitale vorm.

Topic (http://docs.oasis-open.org/dita/v1.0/archspec/topicover.html**)**
Een topic is een element binnen de DITA standaard. Topics zijn de basis voor het indelen van inhoud van een document. Elk topic moet een enkel onderwerp beschrijven.

TEI (http://www.tei-c.org/index.xml**)**
TEI staat voor Text Encoding Initiative. TEI definieert net als DITA een XML structuur voor representeren van tekst in digitale vorm.

HTML (http://www.w3.org/html/**)**
HTML staat voor HyperText Markup Language. HTML is een opmaaktaal voor het specificeren van documenten, voornamelijk bedoeld voor het internet. Alle websites op het internet gebruiken HTML als opmaaktaal. De huidige versie is HTML5. HTML is geen subset van SGML, maar is er wel van afgeleid.

Tag (https://docs.webplatform.org/wiki/guides/the_basics_of_html#What_HTML_looks_like**)**
Een tag is een onderdeel uit een HTML of XML bestand. Een tag in een HTML bestand ziet er bijvoorbeeld zo uit: . Deze tag definieert een paragraaf element. In XML zien de tags er hetzelfde uit als in HTML.

Attribuut (https://docs.webplatform.org/wiki/guides/the_basics_of_html**)**
Een attribuut wordt toegevoegd aan een HTML of XML tag. Een voorbeeld van een attribuut: . Hier is het class-attribuut toegevoegd aan een paragraaf-tag. Het class-attribuut bevat de naam van de class.

CSS (http://www.w3.org/Style/CSS/**)**
CSS staat voor Cascading Style Sheet. In een CSS-bestand wordt de opmaak van een HTML pagina gedefinieerd.

Style rule
Een Style Rule beschrijft de opmaak van een HTML element. Deze Style Rules worden gedefinieerd in een CSS-bestand.

DOM Tree
De DOM Tree representeert de HTML die door de browser is verwerkt.

Element
Een element is een onderdeel in de DOM Tree binnen de browser. Een element wordt in het HTML bestand gerepresenteerd door een tag.

Child-element
Een child-element is een subelement van zijn parent element.

CSSOM Tree (http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/**)**
De CSSOM Tree representeert de CSS die door de browser is verwerkt.

Render Tree (http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/**)**
De Render Tree is een samenvoeging van de DOM en CSSOM Trees. De Render Tree wordt gebruikt voor het (opnieuw) renderen van de webpagina.

Reflow (http://www.phpied.com/rendering-repaint-reflowrelayout-restyle/**)**
Wanneer de layout van een pagina veranderd wordt er door de browser een reflow uitgevoerd. Tijdens de reflow berekent de browser de nieuwe grootte en positie van nieuwe of aangepaste elementen. De elementen die hierdoor beïnvloed worden, worden ook opnieuw gerendert.

Repaint (http://www.phpied.com/rendering-repaint-reflowrelayout-restyle/**)**
Wanneer er een reflow is uitgevoerd of wanneer alleen de opmaak van een element veranderd wordt er een repaint uitgevoerd. Tijdens een repaint tekent de browser de gehele of een deel van de webpagina opnieuw.

Renderen
Het proces van een pagina reflowen en repainten.

Profiler
Een profiler is een tool binnen een browser waarmee verschillende onderdelen op een webpagina beter bekeken en gemeten kunnen worden. Een aantal functionaliteiten van de profiler zijn: inspecteren van HTML elementen, profilen van webpagina, profilen van javascript en het bieden van een console waarop debuginformatie getoond kan worden.

Virtuele machine
Een omgeving welke een vaste pc simuleert waarop een OS geïnstalleerd kan worden binnen een OS wat op een vaste pc draait.

Snapshot
Van een virtuele machine kan een snapshot gemaakt worden waarmee de virtuele machine teruggebracht kan worden in de staat van het moment waarop de snapshot gemaakt werd. Dit kan onder andere gebruikt worden voor het opnieuw kunnen uitvoeren van tests op een virtuele machine, zonder dat de omgeving veranderd is