

Research Rapport Data Preview

Contents

Inleiding	2
Onderzoeksvragen en Opzet.....	3
Hoofdvraag	3
Deelvragen.....	3
Hoe kan ik ervoor zorgen dat mijn applicatie een bestand heen kan sturen naar de gebruiker zonder de gebruiker te lang te laten wachten.	4
Methods	4
Prototyping	6
Resultaat	8
Wat zijn mogelijke opties om een word document te laten previewen?	12
Methods	12
Wat zou ik moeten veranderen	13
Prototyping	15

Inleiding

Bij het gebruik van mijn applicatie is er belang dat er informatie goed wordt overgedragen aan de gebruiker. Dit komt aangezien het doel van mijn applicatie is om template/voorbeelden te geven aan gebruikers voor het opzetten van documenten (Bijvoorbeeld een voorbeeld/template voor een Analyse Document).

Hiervoor is het belangrijk dat de gebruiker goed de benodigde informatie binnen krijgt om te weten of het document dat ze krijgen voldoet aan hun eisen. Het doel van dit Research Rapport is dus om achter te komen hoe we dit zouden kunnen doen.

Onderzoeksvragen en Opzet

Hoofdvraag

Hoe kan ik ervoor zorgen dat gebruikers WORD- en PDF-bestanden snel en efficiënt de preview kunnen inzien.

Deelvragen

- Hoe kan ik ervoor zorgen dat mijn applicatie een bestand heen kan sturen naar de gebruiker zonder de gebruiker te lang te laten wachten.
 - Library: Literature Study
 - Met deze methode wil ik voornamelijk achter komen welke manieren er gebruikt worden door anderen
 - Workshop: Prototyping
 - Met deze methode wil een paar van de manieren die ik was tegengekomen om te prototypen om te kijken of ze weken voor mij applicatie of niet.
 - Lab: Non-Functional test
 - Met deze methode wil ik testen of de verandering die ik heb gemaakt daadwerkelijk sneller is geworden.
- Wat zijn mogelijke opties om een word document te laten previewen?
 - Library: Literature Study
 - Met deze methode wil ik voornamelijk kijken naar manieren die andere gebruiken om een Word document te previewen.
 - Field: Problem Analysis
 - Met deze methode wil ik achter komen wat er allemaal zou moeten veranderen in mijn applicatie om een word document te previewen
 - Workshop: Prototyping
 - Met deze methode wil ik manieren kijken welke manier het beste werkt voor gebruik in mijn applicatie
 - Lab: Usability testing
 - Met deze methode wil ik testen dat een gebruiker daadwerkelijk door deze veranderingen een Word Document kan previewen

Hoe kan ik ervoor zorgen dat mijn applicatie een bestand heen kan sturen naar de gebruiker zonder de gebruiker te lang te laten wachten.

Het versturen van bestanden vanuit een backend naar de frontend kan leiden tot lange wachttijden en hoge belasting op de server, vooral bij grotere bestanden. Door de juiste technieken toe te passen, kun je zorgen voor een **efficiënte bestandsoverdracht** en een **betere gebruikerservaring**.

Methods

Hier onder zijn methods die ik online ben tegen gekomen die ik denk dat zou kunnen werken voor mijn applicatie.

1. Stream het bestand direct naar de gebruiker

In plaats van het hele bestand in het geheugen te laden, kun je het direct als **stream** naar de gebruiker sturen. Dit voorkomt geheugenproblemen en maakt de overdracht efficiënter.

Waarom is dit belangrijk?

- Grote bestanden kunnen veel geheugen gebruiken.
- Directe streaming zorgt ervoor dat gebruikers direct beginnen met downloaden.

2. Gebruik HTTP Range Requests (voor grote bestanden)

Met **HTTP Range Requests** kunnen bestanden in delen worden opgehaald. Dit is vooral handig voor:

- Grote bestanden zoals video's of software.
- Downloads die kunnen worden hervat na een onderbreking.

Voordelen:

- De gebruiker kan een download hervatten zonder opnieuw te beginnen.
- Het versnelt de overdracht van media (streaming).

Spring Boot Ondersteuning

Spring Boot ondersteunt Range Requests automatisch wanneer je `InputStreamResource` of `Resource` gebruikt.

3. Maak gebruik van caching voor herhaalde downloads

Door gebruik te maken van caching voorkom je onnodige herhaalde downloads van hetzelfde bestand.

Hoe werkt dit?

- **ETag-headers:** Laat de client weten of het bestand is gewijzigd.
- **Last-Modified-headers:** Gebruikers krijgen alleen een nieuw bestand als de serverversie nieuwer is.
- **Browser caching:** Zet cachingregels in je backend zodat bestanden lokaal in de browser worden opgeslagen.

5. Optimaliseer netwerkverkeer met compressie

Voor bepaalde bestandstypen, zoals tekstbestanden of JSON-bestanden, kun je **compressie** toepassen voordat je het bestand naar de gebruiker stuurt.

Technieken:

- **GZIP-compressie:** Vermindert de bestandsgrootte en versnelt de overdracht.
- **HTTP Headers:** Gebruik Content-Encoding: gzip om gecomprimeerde bestanden te versturen.

Prototyping

Uit de lijst hierboven heb ik gekozen om voor het volgende een Prototype te maken: Stream en Compression.

Stream

Hier onder zie een voorbeeld van mijn applicatie dat gebruik maakt van een Stream om data heen te sturen. Dit gaat ook goed samen met het gebruik van mongoDB aangezien ik hier direct een InputStream kan halen.

```
54     }
55     public DocumentModel getDocumentModel(ObjectId id) throws IOException { 1 usage  Tjm van Dongen *
56         try{
57             GridFSFile file = gridFsTemplate.findOne(new Query(Criteria.where("key": "_id").is(id)));
58             Log.info(file.getFilename());
59             GridFSResource tempFile = operations.getResource(file);
60             DocumentModel documentModel = new DocumentModel();
61             LocalDate uploadDate = LocalDate.parse(file.getMetadata().get("uploadDate").toString());
62             documentModel.setFileSize(tempFile.getGridFSFile().getLength());
63             documentModel.setUploadDate(uploadDate);
64             documentModel.setFileName(tempFile.getFilename());
65             documentModel.setContentType(tempFile.getContentType());
66             documentModel.setFileStream( new InputStreamResource(tempFile.getInputStream()));
67             documentModel.setFileKey(id);
68             return documentModel;
69         }
70         catch(Exception e){
71             Log.error(ERROR_MESSAGE_SERVICE_LAYER + " getDocumentModel: {} ", e.getMessage());
72             throw e;
73         }
74     }
75
76 }
77
```

```
1     }
2     @GetMapping("/{fileKey}")  Tjm van Dongen *
3     public ResponseEntity<> getFileData(@PathVariable("fileKey") String fileKey) throws IOException {
4         try {
5             DocumentModel model = documentModelService.getDocumentModel(new ObjectId(fileKey));
6             if (model == null) {
7                 return ResponseEntity.badRequest().body("No document model found with the given file key.");
8             }
9             return ResponseEntity.ok()
10                 .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + model.getFilename() + "\"")
11                 .header(HttpHeaders.CONTENT_TYPE, model.getContentType() != null && !model.getContentType().isEmpty()
12                     ? model.getContentType()
13                     : "application/octet-stream")
14                 .header(HttpHeaders.CONTENT_LENGTH, String.valueOf(model.getFileSize()))
15                 .body(model.getFileStream());
16         }
17         catch (Exception e) {
18             Log.debug(e.getMessage());
19             return ResponseEntity.status(500).body("Error downloading the file: " + e.getMessage());
20         }
21     }
22 }
23
```

Compression

Hieronder zie wat ik heb gedaan om iets van compression toe te voegen aan mijn applicatie, ik heb hier voornamelijk gebruik gemaakt van GZIP. Het probleem dat ik hiermee opliep was dat ik in het begin hierbij het verkeerde idee had gekregen, ik dacht in het begin dat het soort van zip bestand maakte die doorgestuurd zou worden. Hoewel het dit soort van doet is het niet helemaal correct. Aangezien gzip ook met http werkt wordt die dus automatisch ge-unzippt wanneer die bij de bestemming aan komt. Dit werkte eigenlijk in voordeel aangezien ik dacht dat ik zelf een unzipping code moest schrijven voor gebruik in het front-end.

```
private InputStream compressedInputStream(InputStream inputStream) throws IOException {
    // Create a ByteArrayOutputStream to hold the compressed data
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

    // Set a larger buffer size (e.g., 8192 bytes)
    byte[] buffer = new byte[8192];

    // Wrap the ByteArrayOutputStream with GZIPOutputStream
    try (GZIPOutputStream gzipOutputStream = new GZIPOutputStream(byteArrayOutputStream, size: 1, syncFlush: true);
        BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream)) {

        int length;
        // Read the input stream in chunks and write to the GZIPOutputStream
        while ((length = bufferedInputStream.read(buffer)) > 0) {
            gzipOutputStream.write(buffer, 0, length);
        }
        gzipOutputStream.finish(); // Ensure data is written out to ByteArrayOutputStream
    }

    // Return the compressed data as an InputStream
    return new ByteArrayInputStream(byteArrayOutputStream.toByteArray());
}
```

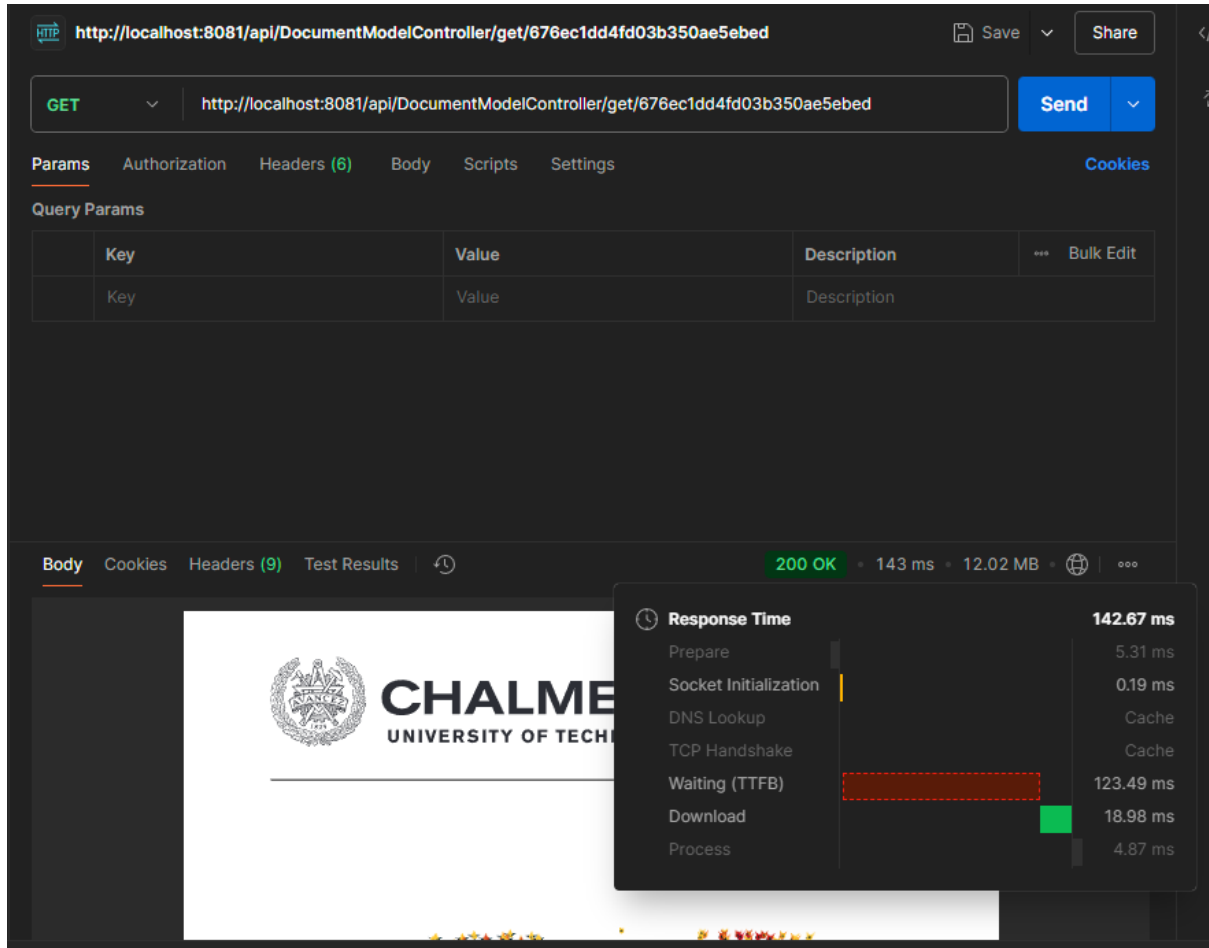
```
@GetMapping("/{fileKey}/compressed") new *
public ResponseEntity<?> getCompressedDocument(@PathVariable("fileKey") String fileKey) throws IOException {
    ObjectId id = new ObjectId(fileKey);
    DocumentModel model = documentModelService.getCompressedDocumentModel(id);

    // Return the compressed file with the correct headers
    return ResponseEntity.ok()
        .header(HttpHeaders.CONTENT_DISPOSITION, ...headerValues: "attachment; filename=\"" + model.getFileName() + "\"")
        .header(HttpHeaders.CONTENT_TYPE, model.getContentType() != null && !model.getContentType().isEmpty()
            ? model.getContentType()
            : "application/octet-stream")
        .header(HttpHeaders.CONTENT_ENCODING, ...headerValues: "gzip") // Indicating that the content is compressed with gzip
        .body(model.getFileStream()); // The compressed content as InputStreamResource
}
```

Resultaat

Om te zien of deze aanpassingen daad werkelijk een effect hebben gehad op laad snelheden heb ik gebruik gemaakt van postgress om de response time tussen de aanpassingen te meten

Als eerst laat ik de response time zien zonder het gebruik van streams bij een URL.



The screenshot shows a web browser interface with a GET request to the URL `http://localhost:8081/api/DocumentModelController/get/676ec1dd4fd03b350ae5ebed`. The response is `200 OK` with a total response time of `143 ms` and a body size of `12.02 MB`. A tooltip displays the detailed response time breakdown:

Step	Time
Prepare	5.31 ms
Socket Initialization	0.19 ms
DNS Lookup	Cache
TCP Handshake	Cache
Waiting (TTFB)	123.49 ms
Download	18.98 ms
Process	4.87 ms

Vervolgens laat ik de response time zien waar er gebruik van streams gemaakt werd om de data te verzenden.

HTTP

http://localhost:8081/api/DocumentModelController/get/676ec1dd4fd03b350ae5ebed

Save

Share

</>

GET

http://localhost:8081/api/DocumentModelController/get/676ec1dd4fd03b350ae5ebed

Send

Params

Authorization

Headers (6)

Body

Scripts

Settings

Cookies

Query Params



	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		


Body

Cookies

Headers (9)

Test Results

200 OK • 129 ms • 12.02 MB •  • 

 Response Time

128.08 ms

Prepare

7.05 ms

Socket Initialization

0.18 ms

DNS Lookup

Cache

TCP Handshake

Cache

Waiting (TTFB)


106.50 ms


Download

21.40 ms

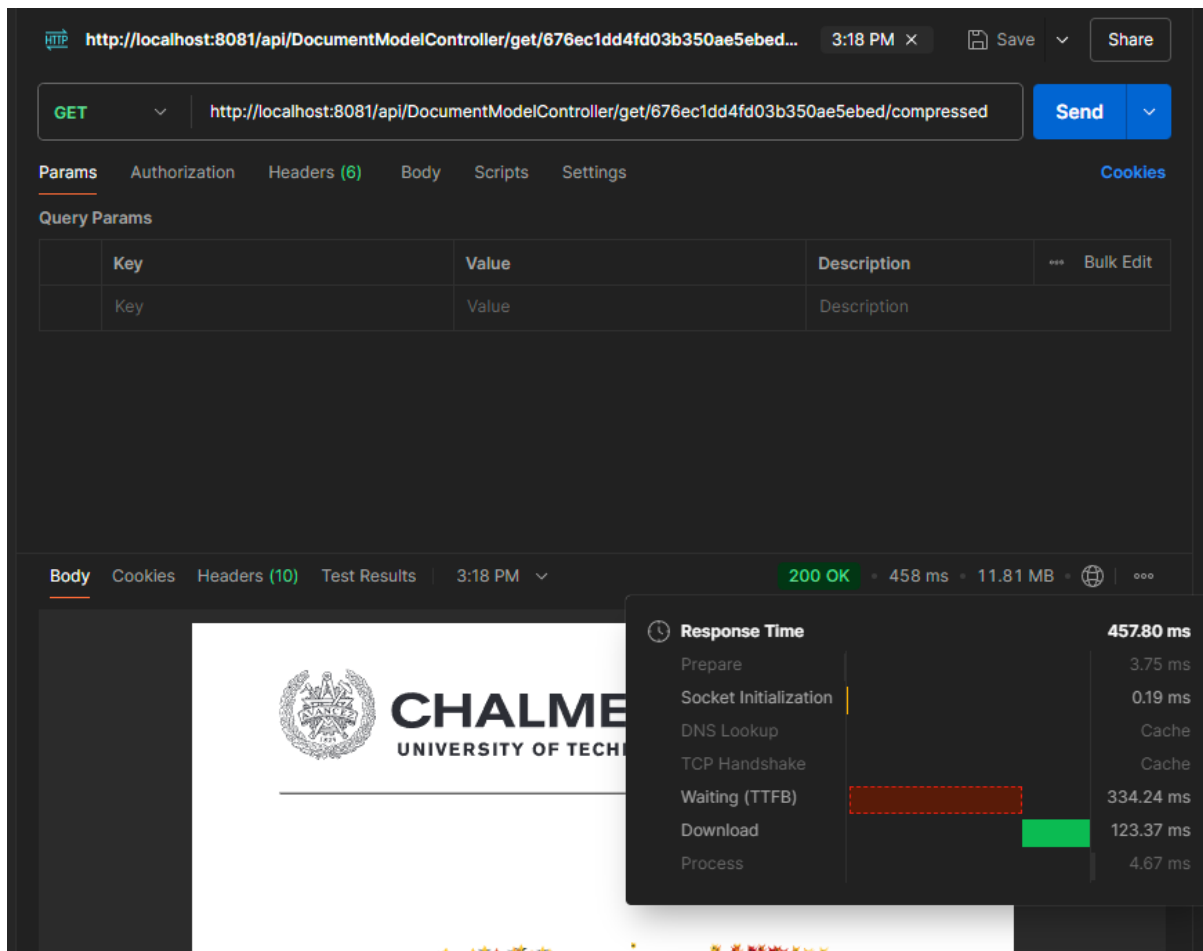
Process

4.41 ms

 **CHALMERS**
UNIVERSITY OF TECHNOLOGY



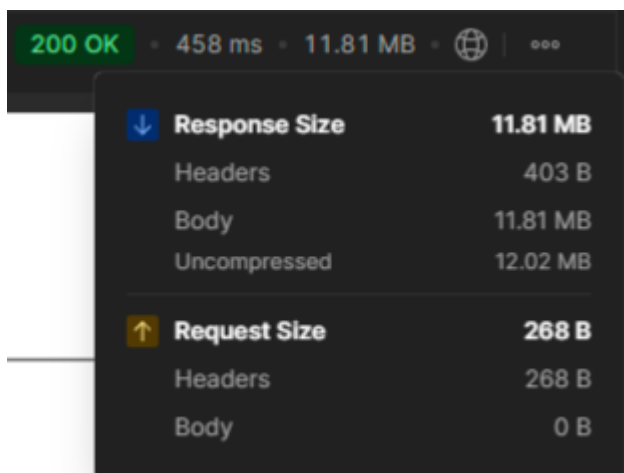
Als laatst kijk ik hoe snel het is om gebruik te maken van compressie zoals GZIP



The screenshot shows a web browser's developer tools interface. The top bar indicates a GET request to `http://localhost:8081/api/DocumentModelController/get/676ec1dd4fd03b350ae5ebd/compressed` at 3:18 PM. The response status is 200 OK, with a response time of 458 ms and a size of 11.81 MB. The 'Body' tab is selected, showing the response content, which includes the Chalmer University of Technology logo and name. A 'Response Time' panel is open, displaying a breakdown of the request process.

Step	Time
Prepare	3.75 ms
Socket Initialization	0.19 ms
DNS Lookup	Cache
TCP Handshake	Cache
Waiting (TTFB)	334.24 ms
Download	123.37 ms
Process	4.67 ms
Total	457.80 ms

GZIP maakt ook het bestand dat je doorstuurt kleinere door het te comprressen en dan vervolgens bij de client automatisch uncompressen.



The screenshot shows a web browser's developer tools interface. The top bar indicates a 200 OK response with a response time of 458 ms and a size of 11.81 MB. The 'Response Size' panel is open, displaying the size of the response components. The 'Request Size' panel is also open, displaying the size of the request components.

Component	Size
Response Size	11.81 MB
Headers	403 B
Body	11.81 MB
Uncompressed	12.02 MB
Request Size	268 B
Headers	268 B
Body	0 B

Wat ik hiervan heb gemerkt is dat streaming mijn de http Request sneller heeft gemaakt met ongeveer 15-20 ms. Dit lijkt niet veel maar kan veel schelen wanneer er veel request worden gestuurt naar de applicatie.

Ik was wel verast met hoe het compressen met GZIP ervoor heeft gezorgd dat de laad snelheden nog langer werden dan zonder compressie. Toen ik hier verder naar heb gekeken waarom dat zo kon zijn ben ik erachter gekomen dat het kan komen omdat de bestanden die ik gebruik te klein waren om echt te compressen of ze waren al in een gecompressed bestand type zoals PDF. Hierdoor zou het compressen er van het alleen maar langer maken zonder er veel waarde uit brengen.

Dit zou waarschijnlijk wel goed werken bij json formats en grote lijsten van data in json format. Het zou dus goed kunnen werken bij gebruik in een lijst van objecten.

Wat zijn mogelijke opties om een word document te laten previewen?

In mijn applicatie is het belangrijk dat gebruikers de bestanden die zijn ge-uploaden kunnen inzien zonder het bestand te downloaden. Dit zou ook het beste zijn als ze dit kunnen doen met wordt documenten zonder deze om te zetten naar foto's of iets anders dat gerenderd kan worden.

Methods

1. Gebruik van ingebouwde browser-ondersteuning

- Je kunt een Word-document rechtstreeks laden in een `<iframe>` of `<embed>` tag. Dit werkt echter alleen als de browser native ondersteuning biedt voor Word-documenten (zoals Google Chrome met Google Docs-integratie).
- **Voordelen:** Eenvoudig te implementeren, geen conversie nodig.
- **Nadelen:** Beperkte browserondersteuning en afhankelijkheid van online viewers zoals Google Docs.

2. Gebruik van een documentviewer-service

- Integreer met een externe service zoals **Google Docs Viewer**, **Microsoft OneDrive Viewer**, of **PDF.js** (voor geconverteerde documenten).
- **Voordelen:** Ondersteuning voor meerdere bestandstypen, geen serverbelasting.
- **Nadelen:** Afhankelijkheid van externe diensten, mogelijk privacy- en beveiligingsproblemen.

3. Server-side conversie naar HTML of PDF

- Converteer het Word-document naar een HTML-pagina of PDF op de server en toon het resultaat in de applicatie.
- Tools: **LibreOffice**, **Aspose.Words**, **Apache POI**.
- **Voordelen:** Volledige controle over het proces en weergave.
- **Nadelen:** Vereist serverinfrastructuur, hogere complexiteit.

4. Client-side rendering met JavaScript libraries

- Gebruik JavaScript libraries zoals **Mammoth.js** of **docx-preview** om Word-documenten rechtstreeks in de browser weer te geven.
- **Voordelen:** Geen serverinteractie nodig na upload, lage latency.
- **Nadelen:** Beperkte styling- en opmaakondersteuning.

5. Embed met cloudopslagintegratie

- Upload het bestand naar een clouddienst (bijvoorbeeld OneDrive of Google Drive) en gebruik de ingebouwde previewfunctionaliteit van deze diensten.
- **Voordelen:** Minimale ontwikkeling nodig, schaalbaar.
- **Nadelen:** Afhankelijkheid van externe clouddiensten, mogelijk minder controle over beveiliging.

Wat zou ik moeten veranderen

Hieronder een overzicht van welke technologieën gebruikt kunnen worden en de omvang van de veranderingen.

1. . Gebruik van ingebouwde browser-ondersteuning (HTML-embed of iframe)

- **Technologieën:** Vue, standaard HTML <iframe> tag.
- **Veranderingen:** Minimale wijzigingen in de frontend, maar mogelijk afhankelijkheid van een externe viewer zoals Google Docs.
- **Omvang van verandering:** Klein.

2. Gebruik van een documentviewer-service

- **Technologieën:** REST API (voor upload), Vue (frontend integratie), externe diensten zoals Google Docs Viewer.
- **Veranderingen:** Implementatie van links naar externe viewers en mogelijk wijzigingen in bestandsopslaglocatie.
- **Omvang van verandering:** Middelmatig (afhankelijk van integratie).

3. Server-side conversie naar HTML of PDF

- **Technologieën:** Spring Boot (REST-service), tools zoals LibreOffice (bijvoorbeeld via CLI of JODConverter).
- **Veranderingen:** Toevoegen van conversielogica op de backend en endpoints voor previewbestanden.
- **Omvang van verandering:** Groot (nieuwe backend-functionaliteit nodig).

4. Client-side rendering met JavaScript libraries

- **Technologieën:** Vue, libraries zoals Mammoth.js of docx-preview.
- **Veranderingen:** Toevoegen van nieuwe JavaScript-library en aanpassingen in de frontend voor het renderen van documenten.
- **Omvang van verandering:** Middelmatig.

5. Embed met cloudopslagintegratie

- **Technologieën:** REST API voor upload naar clouddiensten (bijv. Google Drive API, OneDrive API), Vue voor integratie van embed-links.

- **Veranderingen:** Vereist wijzigingen in bestandsopslag en autorisatie voor cloudintegratie.
- **Omvang van verandering:** Groot.

Prototyping

Voor het prototype heb ik gekozen om gerbuik te maken van een word to PDF Converter inplaats van een viewer die compatible is met word, dit komt omdat ik denk dat het meer bruikbaar is om het zo te doen aangezien meeste viewers meer compatible zijn met een pdf dan een word document. Daarnaast verwacht ik beter conversie te garanderen met een server-sided conversion.

Hier onder staat een Docx to PDF Converter door gebruik van Docx4j, ik had deze gekozen aangezien veel andere gerbuik maakte van externe services zoals openoffice en libreoffice.

Hoewel die mijn beter conversion kwaliteit zouden geven en ook meer ondersteuning voor andere word bestanden, waren ze niet geschikt voor de applicatie aangezien ze een bestand nodig hebben om aan te passen en dit niet in memory kunnen doen.

Dit is belangrijk aangezien de applicatie vaak opgeroepen wordt en het dus niet handig is om dan telkens een bestand aan te maken die dan wordt aangepast.

```
@Service 2 usages 1 Tijm van Dongen
@Slf4j
public class DocxToPdfService {

    public InputStream convertDocxToPdf(InputStream inputStream) throws IOException, Docx4JException, FOException, JAXBException { 1 usage 1 Tijm van Dongen
        ByteArrayOutputStream pdfOutputStream = new ByteArrayOutputStream();
        Log.info(inputStream.available() + " bytes available");
        try {
            String regex = ".*(calibri|camb|cour|arial|times|comic|georgia|impact|LANS|pala|tahoma|trebuc|verdana|symbol|webdings|wingding).*";
            PhysicalFonts.setRegex(regex);
            WordprocessingMLPackage wordMLPackage = WordprocessingMLPackage.Load(inputStream);

            FOSettings foSettings = Docx4J.createFOSettings();
            foSettings.setOpcPackage(wordMLPackage);

            Docx4J.toFO(foSettings, pdfOutputStream, Docx4J.FLAG_EXPORT_PREFER_NONXML);
            Log.info("Length of conversion " + pdfOutputStream.toByteArray().length);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        // Load DOCX file into WordprocessingMLPackage (docx4j model)
        byte[] pdfBytes = pdfOutputStream.toByteArray();
        Log.info("PDF ByteArray length: " + pdfOutputStream.size());
        return new ByteArrayInputStream(pdfBytes);
    }
}
```

Hier onder is de service laag voor het handelen van de DocumentModel er wordt gecheckt of het bestand dat opgeroepen wordt al een pdf is zo niet dan word het geconverteerd naar een pdf.

```
}  
}  
public DocumentModel getPDFDocumentModel(ObjectId id) throws IOException, FOPEException, Docx4JException, JAXBException {  
    try {  
        DocumentModel documentModel = getDocumentModel(id);  
        if("application/pdf".equals(documentModel.getContentType())) return documentModel;  
        InputStream compressedInputStream = docxToPdfService.convertDocxToPdf(documentModel.getFileStream().getInputStream());  
        byte[] pdfBytes = compressedInputStream.readAllBytes(); // Fully read into memory  
        Log.info("PDF byte array size: " + pdfBytes.length);  
  
        documentModel.setFileSize( Long.valueOf(pdfBytes.length));  
  
        // Ensure stream is reusable  
        compressedInputStream = new ByteArrayInputStream(pdfBytes);  
        documentModel.setFileStream(new InputStreamResource(compressedInputStream));  
        return documentModel;  
    } catch (IOException | Docx4JException | FOPEException | JAXBException e) {  
        Log.error(ERROR_MESSAGE_SERVICE_LAYER + " getPDFDocumentModel: {}", e.getMessage());  
        throw e;  
    }  
}
```

Dit is de Endpoint die de pdf verzendt naar de client

```
}  
@GetMapping("/{fileKey}/pdf")  
public ResponseEntity<> getFileDataPDF(@PathVariable("fileKey") String fileKey) throws IOException {  
    try {  
        DocumentModel model = documentModelService.getPDFDocumentModel(new ObjectId(fileKey));  
        if (model == null) {  
            return ResponseEntity.badRequest().body("No document model found with the given file key.");  
        }  
        return ResponseEntity.ok()  
            .header(HttpHeaders.CONTENT_TYPE, "application/pdf")  
            .header(HttpHeaders.CONTENT_LENGTH, String.valueOf(model.getFileSize()))  
            .body(model.getFileStream());  
    }  
    catch (Exception e) {  
        Log.debug(e.getMessage());  
        return ResponseEntity.status(500).body("Error downloading the file: " + e.getMessage());  
    }  
}
```


Vervolgens heb ik gekeken naar een manier om de PDF te displayen. Hiervoor had ik gekeken naar Library's zoals PDF.js, ook had ik gekeken om gebruik te maken van IFrame alleen ik las op het internet dat die niet altijd zou werken op elke browser. Vandaar dat ik keek of pdf.js zou werken

Het probleem dat ik hier mee op liep was dat pdf.js niet fijn werkte met de Framework die ik gebruikte (Vue3 js (Typescript)). Vandaar dat ik de library genaam VuePDF van TaTo30 heb gekozen om te prototypen, de library is een wrap van pdf.js die ervoor zou zorgen dat ik gebruik kan maken van een pdf.js in Vue.

Hier onder wordt de stream die de front-end binnen krijgt om gezet naar een byteArray die VuePDF gebruik van kan maken

```
32 const givenPDF = ref( value: {})  
33 const { pdf, pages } = usePDF(givenPDF)  
34 const fetchPDF = async (fileKey: string) => { Show usages  Tijn van Dongen *  
35   try {  
36     const responseFile = await apiClient.get( url: `/get/${fileKey}/pdf`, config: {responseType:'blob' });  
37     const arrayBuffer = await responseFile.data.arrayBuffer()  
38     const byteArray = new Uint8Array(arrayBuffer);  
39     givenPDF.value = byteArray  
40   } catch (error) {  
41     console.error('Error fetching data:', error);  
42   }  
43 }  
44 // Update loading state
```

Aangezien een VuePDF type alleen maar een enkele pagina van de PDF kan laten zien moet ik meerdere keren VuePDF instantiëren en die een ander paginanummer geven, hiervoor heeft VuePDF gelukkig een pages variabele die bij houdt hoeveel pagina's in de pdf zat voor juist deze functie.

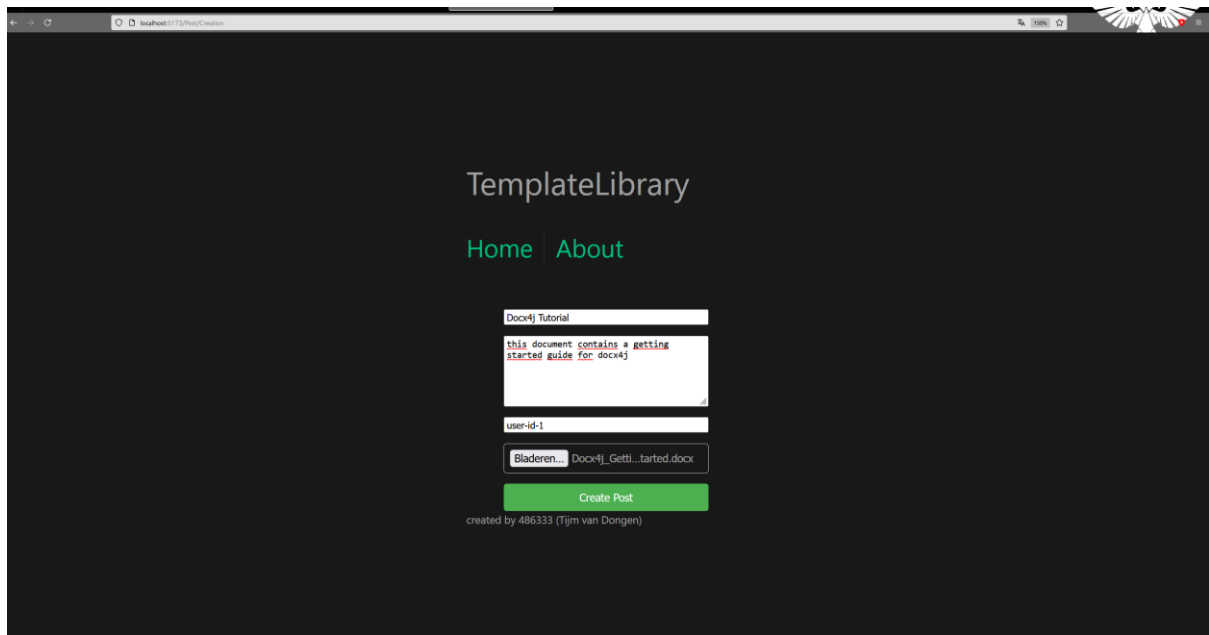
```
</div>  
<div class="pdfContainer OrderLeft" >  
  <div v-if="givenPDF" >  
    <div v-for="page in pages" :key="page">  
      <VuePDF :pdf="pdf" :page="page" :text-layer="true" :annotation-layer="true" fit-parent/>  
    </div>  
  </div>  
</div>
```

[illegible]

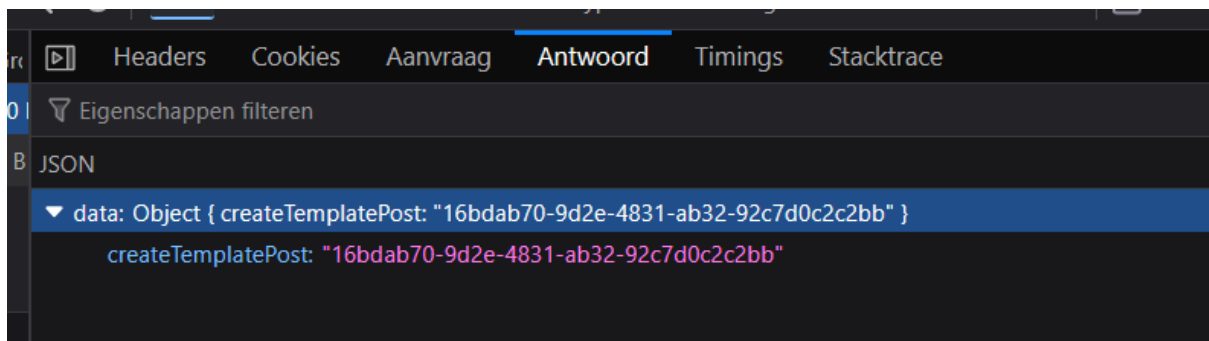
Situatie Test

Hier onder wil ik voornamelijk een test laten zien van de situatie waarbij de gebruiker een Word document uploadt bij zijn post en waarbij hij daarna ook zijn bestand kan inzien na het aanmaken van zijn post.

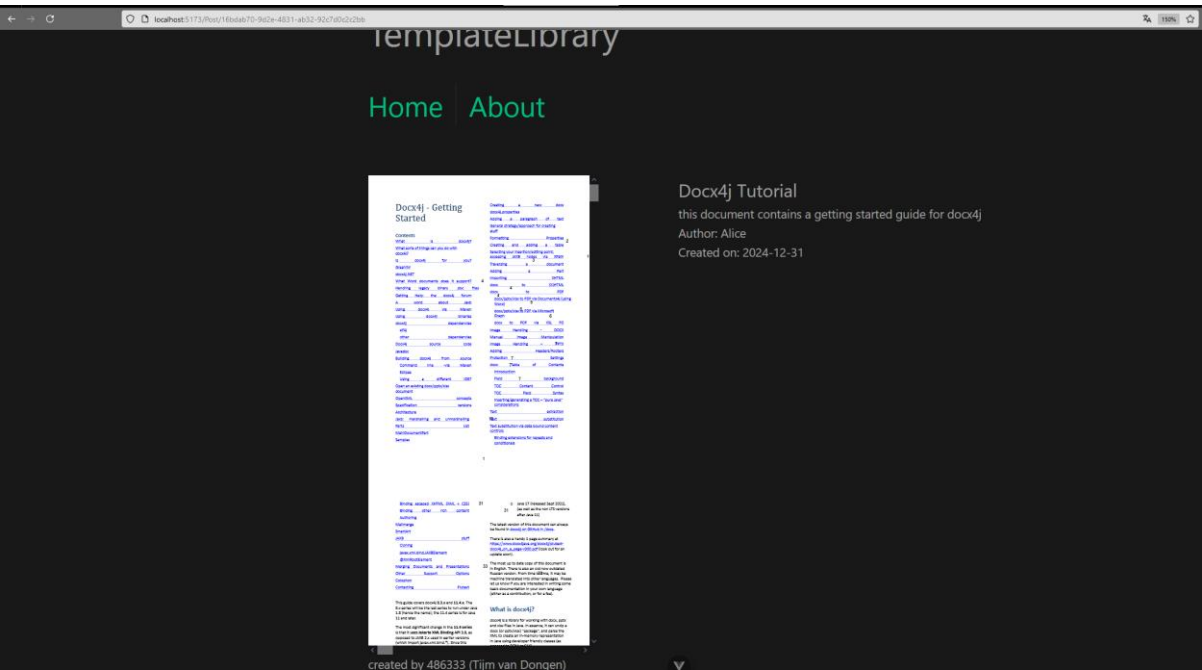
Als eerst gaat de gebruiker beginnen met een post aanmaken, zoals hieronder aangegeven.



In de response staat de ID die word terugggeven van de aangemaakte post



Vervolgens gaat de gebruiker naar de pagina van zijn post waar zijn document als PDF zichtbaar is.



Conclusie

Het onderzoek dat ik heb uitgevoerd, is naar mijn mening nuttig geweest voor mijn applicatie. Het heeft bijgedragen aan een verbeterde gebruikerservaring doordat de kans op fouten minder snel bij de gebruiker wordt gelegd.

Ik ben met name tevreden over de implementatie van een PDF-converter. Dit stelt mij in staat om het originele bestand later nog steeds aan gebruikers beschikbaar te stellen voor download. Dit maakt het eenvoudiger voor gebruikers om het originele bestand als template te gebruiken. Dit was niet mogelijk geweest als ik alleen PDF-bestanden had toegestaan, omdat gebruikers dan hun bestanden niet meer via de browser konden bekijken.

Een punt van kritiek is echter dat de converter momenteel beperkt is tot alleen .docx-bestanden en geen ondersteuning biedt voor .doc- en .odt-bestanden, wat wel mijn oorspronkelijke doel was. Daarnaast heb ik gemerkt dat de conversiekwaliteit nog niet optimaal is. Dit is een compromis dat ik heb gemaakt door geen gebruik te maken van externe diensten zoals OpenOffice of LibreOffice.

Wat betreft het versnellen van het verzenden van bestanden, heeft de implementatie van streaming technologie enkele milliseconden bespaard. Hoewel dit voordeel op kleine schaal nauwelijks merkbaar is, verwacht ik dat het meer impact zal hebben bij hogere verkeersvolumes op de applicatie.

Tot slot vond ik het jammer dat GZIP-compressie de laadsnelheid niet verbeterde, maar zelfs vertraagde. Mijn verwachting was dat het comprimeren van bestanden de snelheid zou verhogen. Helaas heeft het compressieproces zelf te veel tijd toegevoegd. Bovendien bleek de compressie niet significant genoeg om de moeite te rechtvaardigen, omdat de verzonden bestanden voornamelijk PDF's waren, die van nature al gecomprimeerd zijn.