# Report

## Table of Contents

- Introduction
- Literature Review
  - Full-Text Search Engine
    - Precision and Recall
    - Inverted Index
  - Semantic Search
  - Apache Tika
  - DocFetcher
  - Gantt Chart
  - Rationale
- Methodology & Design
  - API Design
  - UI Design
  - Progress
- Implementation
  - Tools & Third-party Libraries
    - Tools
    - Languages
    - Frameworks
    - Third-party Libraries
  - CustodianAPI Implementation
    - Document Indexing
    - Searching
  - Web API Implementation
- Testing
  - API
    - Unit Tests for Custodian API:
    - Unit Tests for Word documents:
    - Unit Tests for Excel documents:
    - Unit Tests for PowerPoint documents:
    - Unit Test for PDF documents:
    - Unit Test for Text documents:

  - Web API
    - Web API Overview
    - Web API Index Folder
    - Web API Get Indexed Folders
    - Web API Search
  - UI
- Conclusion & Future Work
  - Conclusion
  - Future Work
- References
- Appendix

# Introduction

[TBC]

# Literature Review

Following paragraphs are the literature review around the chosen topic.

## Full-Text Search Engine

Full-Text search engines perform linguistic searches against text data in target document or collection of text data based on the rules of a particular language (Microsoft Docs, 2020). Full-Text search engines consider the basic unit of Full-Text search as a token rather than a word (Melton & Buxton, 2006). Full-Text queries can include simple words and phrases or multiple forms of a word or phrase. A Full-Text query returns any documents that contain at least one match (also known as a hit). A match occurs when a target document contains all the terms specified in the Full-Text query, and meets any other search conditions, such as the distance between the matching terms. It widely used in modern technology like web search(Google Search, Bing Search, etc.), desktop search(search for local files) and even email search.

As the need of semantic-aware Full-text search and indexing against databases and documents keeps increasing in the past decade, full-text search systems became available to no-expert users who aren't familiar with documents they are searching, don't know the exact vocabulary used to index relevant documents, or don't know how to formulated advanced search queries (Tekli, Chbeir, et al., 2018). All of those issues mentioned above result in noisy or irrelevant search results and false positives & false negatives in the search result (See Precision and Recall).

### Precision and Recall

Precision and recall are often used to measure the performance of information retrieval or extraction systems, precision (aka. positive prediction value) is the fraction of relevant results among all results returned (in other words, how many returned results are relevant?), while recall (aka. sensitivity) is the fraction of the amount of returned relevant results among all relevant results in the dataset (in other words, how many relevant results are returned?) (Makhoul, Kubala, et al., 1999) (see Appendix a.).

In general, there is a trade-off between "precision" and "recall". High precision means that fewer irrelevant results are returned as a result of the query (fewer false positives), while high recall means that fewer relevant results are missing (fewer false negatives). traditional text-matching systems such as the LIKE operator in SQL gives you 100% precision with no concessions for recall. A full text search facility gives you a lot of flexibility to tune down the precision for better recall (Erickson, 2008).

For example, the SQL LIKE operator can be extremely inefficient. If you apply it to an un-indexed column, a full scan will be used to find matches (just like any query on an un-indexed field). If the column is indexed, matching can be performed against index keys, but with far less efficiency than most index lookups. Therefore, most full text search implementations use an "inverted index" system to improve performance (see Inverted Index) (Erickson, 2008).

### Inverted Index

Inverted Index is an index where the keys are individual terms, and the associated values are sets of records that contain the term. Full text search is optimized to compute the intersection, union, etc. of these record sets, and usually provides a ranking algorithm to quantify how strongly a given record matches search keywords (Erickson, 2008).

As mentioned earlier, non-expert users of full-text search system often don't know how to formulate advanced queries or don't know the vocabularies used when indexing the documents (Tekli, Chbeir, et al., 2018). To address said issue the authors designed and constructed a extended version of inverted index called $SemIndex$, which empowered standard inverted index with a semantic network and a general keyword query model, in order to yield semantic-aware results (ibid.).

## Semantic Search

"Semantics" means the concepts or ideas expressed by words, when people speak to each other, words are not just words but a medium that conveys semantics. Therefore when it comes to web searches, semantic analysis plays a huge part in making search queries easier for a machine to understand (Search Engine Watch, 2019), which then provides users with semantic enabled web searching services (i.e. search with meaning).

Semantic search engine tries to understand the user's intent of a search query and also the contextual meaning of search keywords. Search engine providers (e.g. Google and consequently

SEOs) are dealing with two main concepts behind semantic search, which are semantic mapping (i.e. finding the relationships or connections between words / phrases) and semantic coding (i.e. embedding meaning into webpages using code, to tell search engine what types of information can be found on this page) (ibid.).

## Apache Tika

Tika was formally a sub-project of Apache Lucene. Lucene is a Java library for indexing, searching. Lucene also has many other powerful features (e.g. spellchecking, analysis/tokenization, etc.) (Apache Lucene, 2020).

Apache Tika is an open-source toolkit that detects and extracts metadata and text from over a thousand different file types (such as PPT, XLS, and PDF). All of these file types can be parsed through a single interface, making Tika useful for search engine indexing, content analysis, translation, and much more. You can find the latest release on the download page. Please see the Getting Started page for more information on how to start using Tika (Apache Tika, 2020).

## DocFetcher

DocFetcher is an existing open-source desktop search application written in Java programming language and Apache Tika framework (see Apache Tika). DocFetcher provides very powerful searching capabilities and as well as a wide range of support document formats (DocFetcher, 2020).

## Gantt Chart

The planning of this project is done by utilizing Gantt chart (See Appendix b.).

There're several complications result in a shift of priority to develop a functional cross-platform full-text search API which return search result either from CLI (command-line interface) or `JSON` via REST Web API, instead of building both cross-platform UI and API.

The main processes of this project were requirements analysis, implementation, testing and reporting.

## Rationale

This project idea was formed based on specific needs for customisable full-text searching against a huge amount of local documents, and was then inspired by the open-source desktop search application - DocFetcher (see DocFetcher).

As powerful as DocFetcher is, it lacks the full-support for macOS 64-bit platform, also has some minor performance issues. Same thing goes with other existing desktop search applications. So, I decided to build a local document search engine to address said issues and to fit my specific needs.

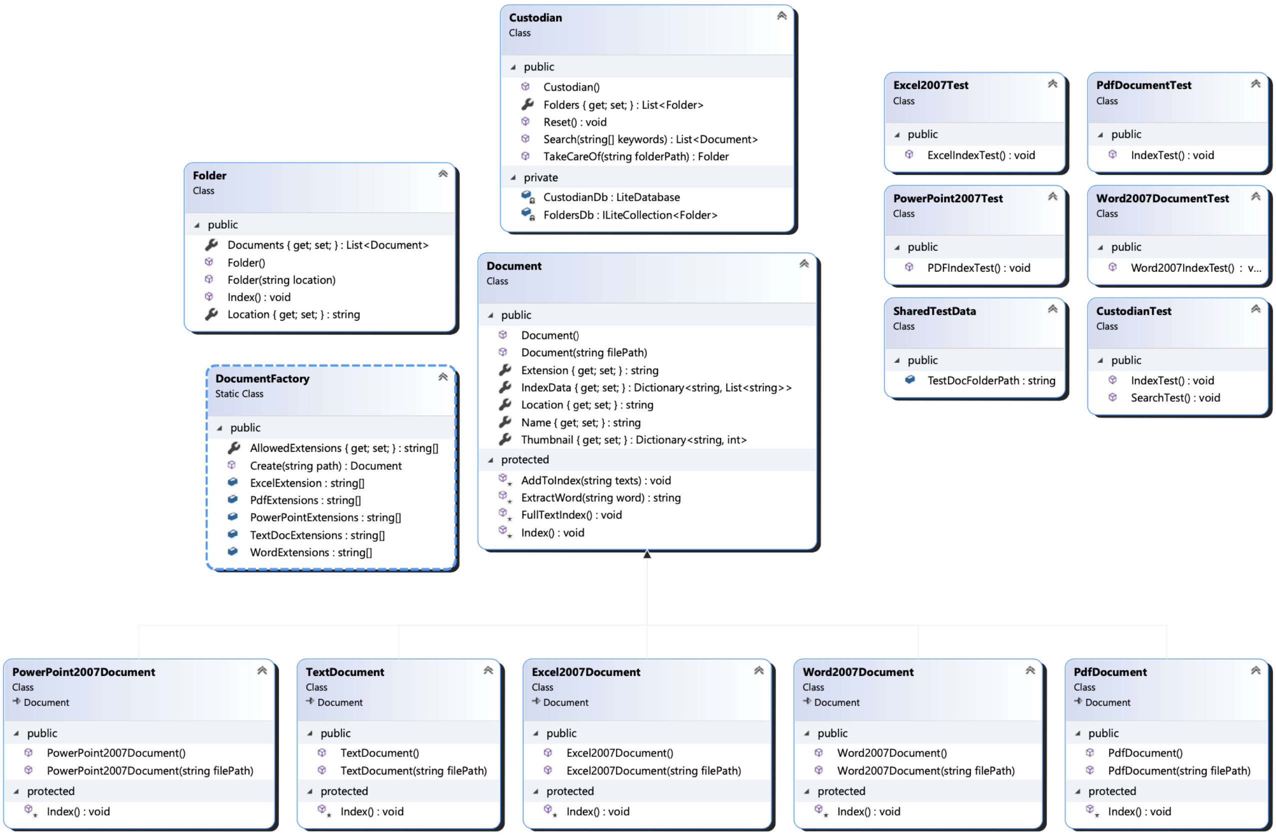
At first I planed to build both GUI (Graphic User Interface) and API (Application Programming Interface), which later turns out to be not feasible enough given the complexity of implementing GUI document previewer and the time constrain for this project. That said, I decided to shift my focus onto API alone.

Tika came into the picture, as I was going through DocFetcher's source code try to understand the implementations for its powerful features. Given the fact that there're no good alternatives to Tika in C# ecosystem, then it occurs to me that i should implement a similar API for text-extraction or content-analysis in C#.

# Methodology & Design

This part will demonstrate and explain the design of API and UI.

## API Design



Custodian
Class
public
Custodian()
Folders { get; set; } : List<Folder>
Reset() : void
Search(string[] keywords) : List<Document>
TakeCareOf(string folderPath) : Folder
private
CustodianDb : LiteDatabase
FoldersDb : ILiteCollection<Folder>
Folder
Class
public
Documents { get; set; } : List<Document>
Folder()
Folder(string location)
Index() : void
Location { get; set; } : string
DocumentFactory
Static Class
public
AllowedExtensions { get; set; } : string[]
Create(string path) : Document
ExcelExtension : string[]
PdfExtensions : string[]
PowerPointExtensions : string[]
TextDocExtensions : string[]
WordExtensions : string[]
Document
Class
public
Document()
Document(string filePath)
Extension { get; set; } : string
IndexData { get; set; } : Dictionary<string, List<string>>
Location { get; set; } : string
Name { get; set; } : string
Thumbnail { get; set; } : Dictionary<string, int>
protected
AddToIndex(string texts) : void
ExtractWord(string word) : string
FullTextIndex() : void
Index() : void
Excel2007Test
Class
public
ExcelIndexTest() : void
PdfDocumentTest
Class
public
IndexTest() : void
PowerPoint2007Test
Class
public
PDFIndexTest() : void
Word2007DocumentTest
Class
public
Word2007IndexTest() : v...
SharedTestData
Class
public
TestDocFolderPath : string
CustodianTest
Class
public
IndexTest() : void
SearchTest() : void
PowerPoint2007Document
Class
Document
public
PowerPoint2007Document()
PowerPoint2007Document(string filePath)
protected
Index() : void
TextDocument
Class
Document
public
TextDocument()
TextDocument(string filePath)
protected
Index() : void
Excel2007Document
Class
Document
public
Excel2007Document()
Excel2007Document(string filePath)
protected
Index() : void
Word2007Document
Class
Document
public
Word2007Document()
Word2007Document(string filePath)
protected
Index() : void
PdfDocument
Class
Document
public
PdfDocument()
PdfDocument(string filePath)
protected
Index() : void

The class diagram above showed the structure and relationship between classes in the API project.

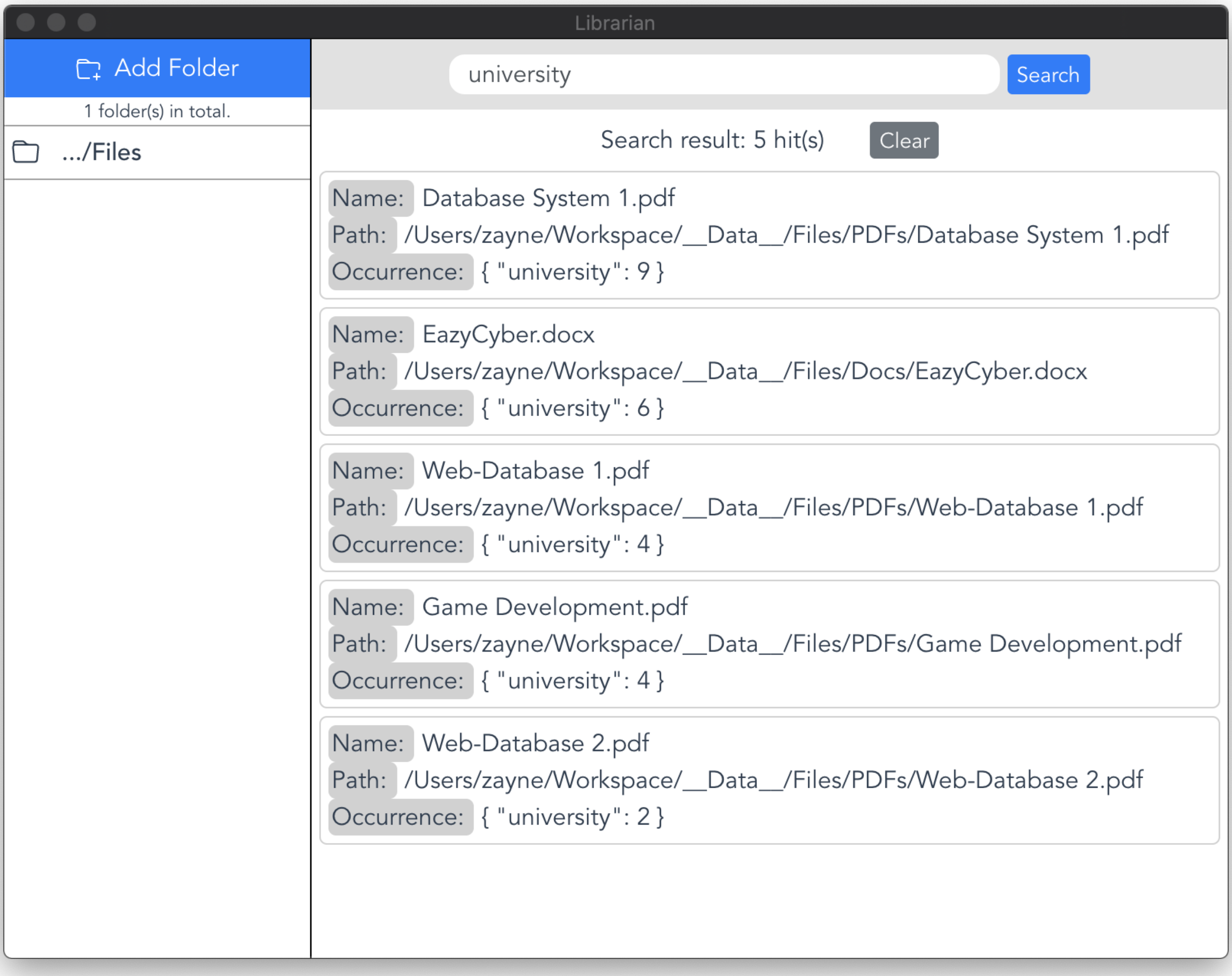
Where `Custodian` class is the entry point of the API, which then initialise a `Folder` instance with a given folder path to index containing supported documents.

`Folder` will then go through the given directory to find supported documents, and `DocumentFactory` will instantiate the correct document class for each supported document based on its file extension. Upon instantiation, each `Document` instance wil begin indexing process.

## UI Design

I'm using Vue.js + NW.js for the cross-platform UI implementation.

As of now, the UI is still rather simple and act only as a visual user interface for demonstration of `Custodian API` 's functionalities.



1 folder(s) in total.

.../Files

university

Search result: 5 hit(s)

Name: Database System 1.pdf
Path: /Users/zayne/Workspace/__Data__/Files/PDFs/Database System 1.pdf
Occurrence: { "university": 9 }

Name: EazyCyber.docx
Path: /Users/zayne/Workspace/__Data__/Files/Docs/EazyCyber.docx
Occurrence: { "university": 6 }

Name: Web-Database 1.pdf
Path: /Users/zayne/Workspace/__Data__/Files/PDFs/Web-Database 1.pdf
Occurrence: { "university": 4 }

Name: Game Development.pdf
Path: /Users/zayne/Workspace/__Data__/Files/PDFs/Game Development.pdf
Occurrence: { "university": 4 }

Name: Web-Database 2.pdf
Path: /Users/zayne/Workspace/__Data__/Files/PDFs/Web-Database 2.pdf
Occurrence: { "university": 2 }

## Progress

| Icon | ☐ | ☑ | 🚧 |
|---|---|---|---|

| Icon | ☐ | ☑ | 🚧 |
|---|---|---|---|
| Meaning | Planed | Done | WIP |

- UI
  - ☑ Index folders.
  - ☑ Retrieve indexed folders.
  - ☑ Search function.
  - ☐ Advance search query.
  - ☑ Display search result.
- Web API
  - ☑ API Documentation.
  - ☑ Indexing.
  - ☑ Searching.
  - ☐ Advanced search query.
- API
  - ☑ 64-bit multi-platform support.
  - ☑ Indexing.
  - ☑ Searching.
  - ☐ Semantic Search. !!!
- File type support
  - ☑ Microsoft Office 2007+ documents support.
    - ☑ `.docx`, `.docm` | Word 2007+ documents.
    - ☑ `.pptx`, `.pptm` | PowerPoint 2007+ documents. ⚠
    - ☑ `.xlsx`, `.xlsm` | Excel 2007+ documents. ⚠
  - ☐ Plain-text documents.
    - ☑ `.txt` | Text documents.
    - ☐ `.md` | Markdown documents. 🚧
    - ☐ `.rtf` | Rich text format documents.
  - ☐ `.pdf` | PDF documents.
    - ☑ Text-based PDF.
    - ☐ Image-only PDF. 🚧
  - ☐ `.epub`, `.mobi`
- Testing

| Icon | Meaning |
|---|---|
| ✅ | Fully Tested |
| ✓ | Partially Tested<br>( Basic functionalities ) |

| File Type | Testing Status |
|---|---|
| `.txt` | ✅ |
| `.docx` , `.docm` | ✅ |
| `.pptx` , `.pptm` | ✓ |
| `.xlsx` , `.xlsm` | ✓ |
| `.pdf` | ✓ |

- Report 🚧

# Implementation

## Tools & Third-party Libraries

This part contains general background information regarding tools, languages and third-party libraries used in this project.

### Tools

#### Visual Studio Code

Visual Studio Code is an open-source cross-platform code editor developed by Microsoft, with great extensibility and support for syntax highlighting, debugging, IntelliSense (code completion tool developed by Microsoft), etc. And VSCode supports a large variety of programming languages by installing dedicated extensions (VSCode, 2020).

### Languages

#### C Sharp

`C#` is a modern, general-purpose, object-orient programming language which was released by Microsoft in 2000 as a part of `.NET Framework` (ECMA, 2017). C# is type-safe and the syntax of it is highly expressive. In addition, C# also benefits from Language-Integrated Query (LINQ) which provides built-in query capabilities across many different kinds of data sources (Microsoft, 2015).

### TypeScript

TypeScript (abbr. TS) is an open-source language that is built on top of JavaScript. TypeScript empowers JavaScript with static type checking and type inference capabilities by adding static type definitions (Microsoft, 2020).

## Frameworks

### .NET Core

.NET Core is an open-source, general-purpose development platform, which can be used to create .NET Core applications for Windows, macOS, and Linux for x64, x86, ARM32, and ARM64 processors using `C#`, `Visual Basic`, and `F#` programming languages (Microsoft Docs, 2020).

### ASP.NET Core

ASP.NET Core is a high-performance cross-platform open-source framework for building web applications / cloud services, IoT, and mobile backends (Microsoft Docs, 2020).

### NW.js

NW.js is previous called node-webkit, it's an application runtime based on `Chromium` (i.e. open-source chrome engine) and `node.js`. NW.js provides developers the ability to build cross-platform native applications with Web technologies (Nw.js, 2020).

### Vue.js

Vue.js is a progressive framework that focuses on data binding for view layer in web development. And Vue.js also has the concept of component which is perfectly for powering sophisticated SPA (i.e. Single Page Application) (Vue.js, 2020).

## Third-party Libraries

This project also utilizes several third-party libraries to implement certain functionalities.

### LiteDB

LiteDB is a embedded No-SQL single-file Database. It's used in this project as an object storage for indexed data.

### Open XML SDK

Open-XML SDK is an SDK that provides tools for working with Office (2007+) Word, Excel, and PowerPoint documents developed by OfficeDev Team at Microsoft.

Office Open XML (aka. OpenXML or OOXML) is an XML-based format for office documents, including word processing documents, spreadsheets, presentations, as well as charts, diagrams, shapes, and other graphical material. Since Office 2007 the file formats and structure of all office documents has changed. Instead of older binary formats (i.e. `.doc`, `.xls`, and `.ppt`), they have adopted Open XML to be the default format of all Microsoft Office documents (`.docx`, `.xlsx` and `.pptx`) (Office Open XML, 2012).

OpenXML files can be decompressed into many `.xml` files by compression software or tools.

### iText Core

`iText Core` is a versatile programmable PDF solution for both .NET (C#) and Java languages. iText provides developers with the ability to create, read, modify PDF documents (iText, 2020).

### axios

Axios is a promise based HTTP client for the browser and node.js (Axios, 2020). Axios supports Promise API which is a built-in javascript API used to represent the eventual completion (or failure) of an asynchronous operation, and its resulting value (MDN, 2020), promise API also supports method chaining to add more asynchronous actions.

```
// Promise method chaining.
const promise2 = doSomething().then(successCallback, failureCallback);
```

# CustodianAPI Implementation

Detailed implementations regarding Indexing, Searching and WebAPI functionalities of this project will be given in the this section.

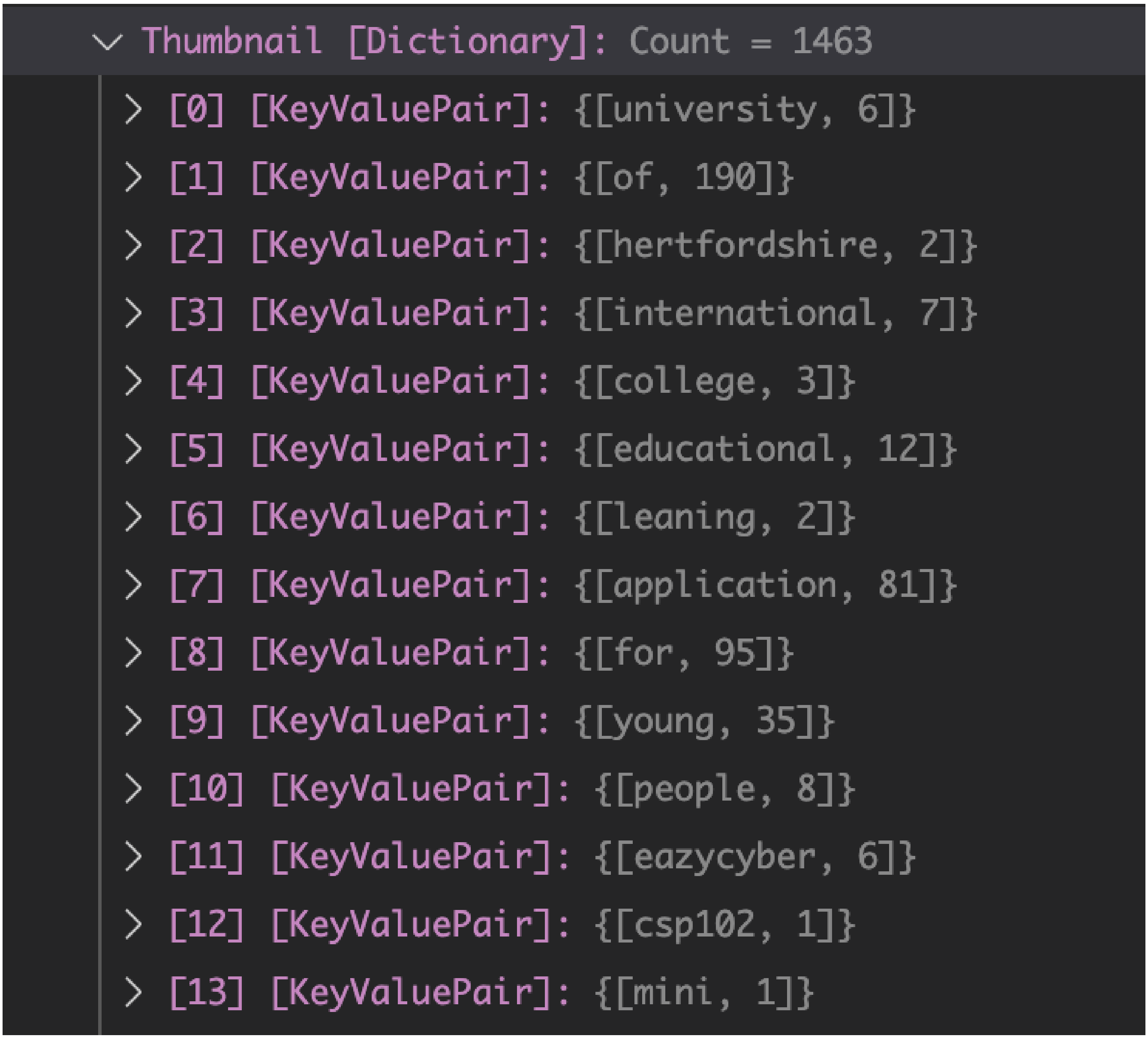
## Document Indexing

### Inverted Index

Code below is responsible for extract words from pre-processed paragraphs or sentences, and add each word into the Dictionary where inverted index is implemented.

```
protected void AddToIndex(string texts)
{
    using var words = texts.Split(" ", StringSplitOptions.RemoveEmptyEntries).As
    while (words.MoveNext())
    {
        var word = words.Current;
```

```
            var processedWord = ExtractWord(word);
            if (processedWord == null) continue;

            if (Thumbnail.ContainsKey(processedWord))
            {
                Thumbnail[processedWord]++;
                continue;
            }
            Thumbnail.Add(processedWord, 1);
        }
    }
```



```
Thumbnail [Dictionary]: Count = 1463
  > [0] [KeyValuePair]: {[university, 6]}
  > [1] [KeyValuePair]: {[of, 190]}
  > [2] [KeyValuePair]: {[hertfordshire, 2]}
  > [3] [KeyValuePair]: {[international, 7]}
  > [4] [KeyValuePair]: {[college, 3]}
  > [5] [KeyValuePair]: {[educational, 12]}
  > [6] [KeyValuePair]: {[leaning, 2]}
  > [7] [KeyValuePair]: {[application, 81]}
  > [8] [KeyValuePair]: {[for, 95]}
  > [9] [KeyValuePair]: {[young, 35]}
  > [10] [KeyValuePair]: {[people, 8]}
  > [11] [KeyValuePair]: {[eazycyber, 6]}
  > [12] [KeyValuePair]: {[csp102, 1]}
  > [13] [KeyValuePair]: {[mini, 1]}
```

The figure above shows the partial structure of implemented inverted index in the memory. Based on current implementation each file type will eventually be processed into Dictionaries like this.

---

### Word 2007

The structure of composing `.xml` files for `.docx` documents is similar to the picture (see Appendix c.).

The main document story of the simplest WordprocessingML ( `.docx` ) document consists of the following XML elements (Microsoft Docs, 2017):

| Element | Description |
|---|---|
| document | The root element for a WordprocessingML's main document part, which defines the main document story. |
| body | The container for the collection of block-level structures that comprise the main story. |
| p | A paragraph. |
| r | A run. |
| t | A range of text. |

Following is the code to extract text from Word documents. `Word2007Document.cs`

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    #region Index Word Document
    // ReadFiles
    using var doc = WordprocessingDocument.Open(path: Location, isEditable: fals
    var body = doc.MainDocumentPart.Document.Body;
    // Get all <w:p> elements.
    using var paragraphParts = body.Descendants<Paragraph>().GetEnumerator();
    while (paragraphParts.MoveNext())
    {
        var currentParagraph = paragraphParts.Current;
        // Find all <w:t> elements inside each <w:p> element.
        var textParts = currentParagraph.Descendants<Text>().AsQueryable();
        var paragraphTemp = from text in textParts
                            where text.Text != ""
                            // Last char is number, possibly page number in Tab
                            select char.IsNumber(text.Text, text.Text.Length -
        var paragraph = string.Concat(paragraphTemp);

        if (new[] { "" }.Contains(paragraph)) continue;
        if (paragraph == null) continue;

        this.AddToIndex(texts: paragraph);
    }
    #endregion
    Console.Write(
        $" >==> {Thumbnail.Count} unique words. {(DateTime.Now - startTime).Tota
}
```

There are still some special formats or cases haven't been taken into consideration, which will be covered in future work.

---

### Excel 2007

The document structure of a SpreadsheetML document consists of the `workbook` element that contains `sheets` and `sheet` elements that reference the worksheets in the workbook (Microsoft Docs, 2017). (see Appendix d.)

The main parts of a SpreadsheetML document:

| Name | Element | Description |
|---|---|---|
| Workbook | workbook | The root element for the main document part. |
| Worksheet | worksheet | A type of sheet that represent a grid of cells that contains text, numbers, dates or formulas. |
| Table | table | A logical construct that specifies that a range of data belongs to a single dataset. |
| Shared Strings Table | sst | A construct that contains one occurrence of each unique string that occurs on all worksheets in a workbook. |

`sheet1.xml`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<worksheet xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" >
    <sheetData>
        <row r="1" spans="1:3" x14ac:dyDescent="0.2">
            <c r="A1" t="s">
                <v>0</v>
            </c>
            <c r="B1" s="1">
                <v>123</v>
            </c>
            <c r="C1" s="3">
                <v>36078</v>
            </c>
        </row>
        <row r="2" spans="1:3" x14ac:dyDescent="0.2">
            <c r="A2" t="s">
```

```
                <v>1</v>
            </c>
            <c r="B2" s="1">
                <v>123</v>
            </c>
            <c r="C2" t="s">
                <v>3</v>
            </c>
        </row>
    </sheetData>
</worksheet>
```

However, based on the implementation standard of SpreadsheetML documents, strings sometimes can be stored in `sharedStringTable`, which is similar to an Array in programming languages.

Example content of `sharedStrings.xml`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sst xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" count="4"
    <si>
        <t>university</t>
    </si>
    <si>
        <t>of</t>
    </si>
    <si>
        <t>hertfordshire</t>
    </si>
</sst>
```

In the example of `sheet1.xml` above, each `<c>` element with `t="s"` attribute mean that it's a reference to shared strings table, with the value inside `<v>` element as the index of the string it's referencing.

Hence in the implementation below, there's a part code for retrieving the `sharedStringTable`. And in the text extraction process,

`Excel2007Document.cs`

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    #region Excel
    var xlsx = SpreadsheetDocument.Open(path: Location, isEditable: false);
    var workSheets = xlsx.WorkbookPart.WorksheetParts.GetEnumerator();
    # region Get sharedStringTable
    IQueryable<string> sharedStringTableList = null;
    using (var parts = xlsx.RootPart.Parts.GetEnumerator())
    {
        while (parts.MoveNext())
        {
            var b = parts.Current.OpenXmlPart;
            if (b.GetType() == typeof(SharedStringTablePart))
            {
                var part = (SharedStringTablePart)parts.Current.OpenXmlPart;
                sharedStringTableList = from item in part.SharedStringTable.Eler
                                        select item.InnerText;
            }
        }
    }
    #endregion

    while (workSheets.MoveNext())
    {
        var sheet = workSheets.Current;
        using var sheetData = sheet.Worksheet.Elements<SheetData>().GetEnumerato
        while (sheetData.MoveNext())
        {
            using var rows = sheetData.Current.Elements<Row>().GetEnumerator();
            while (rows.MoveNext())
            {
                var row = rows.Current;
                using var cells = row.Elements<Cell>().GetEnumerator();
                while (cells.MoveNext())
                {
                    var cell = cells.Current;
                    var text = "";
                    // TODO: Potential issue with the cell type
                    // FIXME: The value the cells with type `Date`
                    if (cell.DataType == null)
                        text = cell.CellValue.InnerText;
                    else if (cell.DataType == CellValues.SharedString)
                        text = sharedStringTableList.ElementAt(int.Parse(cell.Ce
                    else
                        text = cell.CellValue.InnerText;

                    this.AddToIndex(text);
                }
            }
        }
    }
    #endregion

    Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
}
```

However, there's still an issue with retrieving formatted date value. Pending for further fixes.

### PowerPoint 2007

`PowerPoint2007Document.cs`

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    # region PowerPoint
    var ppt = PresentationDocument.Open(path: Location, isEditable: false);
    // Get all slides in current presentation.
    using var slides = ppt.PresentationPart.SlideParts.GetEnumerator();

    while (slides.MoveNext())
    {
        var slide = slides.Current;
        // Get all Text elements inside current slide
        using var text = slide.Slide.Descendants<TextBody>().GetEnumerator();
        while (text.MoveNext())
        {
            this.AddToIndex(texts: text.Current.InnerText);
        }
    }
    #endregion

    Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
}
```

There are already known issues with the implementation above which will be fixed in the future, thus no detailed documentation is provided at this point.

### Plain-text Documents

`TextDocument.cs`

```
protected override void Index()
{
    if (Location is null)
        return;

    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    #region txt
    // Get encoding of current document.
    var encoding = new StreamReader(Location, true).CurrentEncoding;
    // Get all lines in `.txt` file as an Enumerator.
    using var lines = File.ReadAllLines(Location, encoding).AsEnumerable().GetEn
    while (lines.MoveNext())
    {
        var currentLine = lines.Current;
        if (currentLine == null) break;

        this.AddToIndex(texts: currentLine);
    }
    #endregion

    Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
}
```

In all the indexing implementations mentioned above, i'm using `Enumerator` instead of `for` / `foreach` loops for performance and memory consumption concerns. `using` keyword is also heavily used for garbage collection (i.e. disposal of no-longer need objects).

### PDF Documents

`PdfDocument.cs`

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    # region PDF
    // Get the PDF document from a `FileStream` via `PdfReader`.
    var pdfDocument =
        new Pdf.PdfDocument(new Pdf.PdfReader(new FileStream(Location, FileMode.
    var totalPageNumber = pdfDocument.GetNumberOfPages();

    for (var i = 1; i <= totalPageNumber; i++)
    {
        var text = PdfTextExtractor.GetTextFromPage(pdfDocument.GetPage(i));

        this.AddToIndex(texts: text);
    }
    #endregion

    Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
}
```

---

## Searching

`Custodian.cs`

```
public List<Document> Search(string[] keywords)
{
    var result = new List<Document>();
    using var folders = Folders.GetEnumerator();
    try
    {
        while (folders.MoveNext())
        {
            var current = folders.Current;
            if (current == null)
                throw new Exception("no folders indexed.");

            using var docs = current.Documents.GetEnumerator();
            while (docs.MoveNext())
            {
                var currentDoc = docs.Current;
                if (currentDoc == null)
                    throw new Exception("no documents indexed.");
                // Calculate the intersection of search keywords and index data
                var intersection = currentDoc.Thumbnail.Keys.Intersect(keywords
                // Check if there is any intersected items.
                if (intersection.Any())
                {
                    result.Add(currentDoc);
                }
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
    }
    finally
    {
        folders.Dispose();
    }

    // Preliminary ranking based on the occurrences of search keyword.
    result.Sort((a, b) => b.Thumbnail[keywords[0]].CompareTo(a.Thumbnail[keywor
    return result;
}
```

# Web API Implementation

### CustodianApiController.cs:

```
[ApiController]
[Route("/")]
public class CustodianApiController : Controller
```

```
{
    public static readonly Custodian Custodian = new Custodian();
    public static readonly char PathDelimiter = Path.DirectorySeparatorChar;

    // Method `GET`
    // URL: "/folders"
    /// <summary>
    /// Get indexed folders.
    /// </summary>
    /// <returns>List of indexed folders</returns>
    [HttpGet("folders")]
    public ActionResult<List<FolderResult>> GetIndexedFolders()
    {
        var result = new List<FolderResult>();

        var folders = Custodian.Folders;
        folders.ForEach(
            folder => result.Add(
                new FolderResult
                {
                    FolderName = folder.Location.Split(PathDelimiter).Last(),
                    FolderPath = folder.Location
                })
        );
        return result;
    }

    // Method `POST`
    // URL: "/folders"
    /// <summary>
    /// Index given folder.
    /// </summary>
    /// <param name="folderPath">Path to the folder to be indexed.</param>
    /// <returns>Indexed folder info.</returns>
    [HttpPost("folders")]
    public IActionResult IndexGivenFolder(string folderPath)
    {
        if (folderPath == null) return new JsonResult(new { msg = "folderPath ca

        var result = Custodian.TakeCareOf(folderPath);
        // Console.WriteLine();
        var folderName = result.Location.Split(PathDelimiter).Last();

        return new JsonResult(
            new
            {
                folder = new { location = result.Location, files = result.Docume
                msg =
                    $"folder {folderName} is indexed with {result.Documents.Cour
            }
        );
    }

    // Method `POST`
    // URL: "/search"
    [HttpPost("search")]
    public ActionResult<List<DocumentResult>> Search(string keyword)
    {
        if (keyword == null)
        {
            return new List<DocumentResult>();
        }
        keyword = keyword.ToLower();
        var result = new List<DocumentResult>();
        var keywords = keyword.Split(" ", StringSplitOptions.RemoveEmptyEntries

        Console.WriteLine(Custodian.Search(keywords));
        using var resultDocList = Custodian.Search(keywords).GetEnumerator();
        while (resultDocList.MoveNext())
        {
            var currentDoc = resultDocList.Current;
            if (currentDoc == null)
                throw new Exception("No Doc!");

            var resultDict = keywords.ToDictionary(kw => kw, kw => currentDoc.Th
            result.Add(
                new DocumentResult()
                {
                    Name = currentDoc.Name,
                    Path = currentDoc.Location,
                    Result = resultDict
                }
            );
        }
        return result;
    }
    // Return type for schemas in Swagger Doc
    public class FolderResult
    {
        [Required] public string FolderName { get; set; }

        [Required] public string FolderPath { get; set; }
    }
    // Return type for schemas in Swagger Doc
    public class DocumentResult
    {
        /// <summary>
        /// File name of the document.
        /// </summary>
        [Required]
        public string Name { get; set; }

        /// <summary>
        /// Path of the document.
        /// </summary>
        [Required]
        public string Path { get; set; }
        /// <summary>
        /// Key-value pairs of each keyword and its occurrences.
        /// </summary>
        [Required]
        public Dictionary<string, int> Result { get; set; }
    }
}
```

---

# Testing

## API

Unit testing was used to test the behaviours of each corresponding class for different file types.

### Unit Tests for Custodian API:

The code below is a simple unit test for Indexing a given folder and Searching capabilities of `Custodian` class.

```
using Xunit;

namespace CustodianAPI.Test
{
    public class CustodianTest
    {
        [Fact]
        public void IndexTest()
        {
            // Given
            var folderPath = SharedTestData.TestDocFolderPath;
            var custodian = new Custodian();

            // When
            custodian.Reset();
            var result = custodian.TakeCareOf(folderPath: folderPath);

            // Then
            Assert.Equal(10, result.Documents.Count);
        }
        [Fact]
        public void SearchTest()
        {
            //Given
            var keywords = new string[] { "university" };
            var folderPath = SharedTestData.TestDocFolderPath;
            var custodian = new Custodian();
            //When
            custodian.TakeCareOf(folderPath);
            var result = custodian.Search(keywords);
            var resultSize = result.Count;

            //Then
            Assert.Equal(expected: 6, resultSize);
        }

    }
}
```

## Unit Tests for Word documents:

The code below is a simple unit test for `Index` functionality of `Word2007Document` class.

```
using CustodianAPI.Utils;
using Xunit;

namespace CustodianAPI.Test
{
    public class Word2007DocumentTest
    {
        [Fact]
        public void Word2007IndexTest()
        {
            // Given
            var docxPath = SharedTestData.TestDocFolderPath + "test.docx";
            var docmPath = SharedTestData.TestDocFolderPath + "test.docm";

            // When
            var docx = new Word2007Document(docxPath);
            var docm = new Word2007Document(docmPath);

            // Then
            Assert.Equal(6, docx.Thumbnail["university"]);
            Assert.Equal(6, docm.Thumbnail["university"]);
        }
    }
}
```

## Unit Tests for Excel documents:

The code below is a simple unit test for `Index` functionality of `Excel2007Document` class.

```
using CustodianAPI.Utils;
using Xunit;
```

```
namespace CustodianAPI.Test
{
    public class Excel2007Test
    {
        [Fact]
        public void ExcelIndexTest()
        {
            //Given
            var xlsmPath = SharedTestData.TestDocFolderPath + "test.xlsm";
            var xlsxPath = SharedTestData.TestDocFolderPath + "test.xlsx";

            //When
            var xlsx = new Excel2007Document(xlsxPath);
            var xlsm = new Excel2007Document(xlsmPath);

            //Then
            Assert.Equal(1, xlsx.Thumbnail["university"]);
            Assert.Equal(3, xlsx.Thumbnail["123"]);

            Assert.Equal(1, xlsm.Thumbnail["university"]);
            Assert.Equal(3, xlsm.Thumbnail["123"]);
        }
    }
}
```

## Unit Tests for PowerPoint documents:

The code below is a simple unit test for `Index` functionality of `PowerPoint2007Document` class

```
using CustodianAPI.Utils;
using Xunit;

namespace CustodianAPI.Test
{
    public class PowerPoint2007Test
    {
        [Fact]
        public void PDFIndexTest()
        {
            //Given
            var pptxPath = SharedTestData.TestDocFolderPath + "test.pptx";
            var pptmPath = SharedTestData.TestDocFolderPath + "test.pptm";

            //When
            var pptx = new PowerPoint2007Document(pptxPath);
            var pptm = new PowerPoint2007Document(pptmPath);

            //Then
            //FIXME: Check for `tx:body` issue
            // Find another test doc?
            Assert.Equal(expected: 1, actual: pptx.Thumbnail["harikala"]);
            Assert.Equal(expected: 44, actual: pptx.Thumbnail["programming"]);
            Assert.Equal(expected: 28, actual: pptx.Thumbnail["paradigm"]);

            Assert.Equal(expected: 1, actual: pptm.Thumbnail["harikala"]);
            Assert.Equal(expected: 28, actual: pptm.Thumbnail["paradigm"]);
            Assert.Equal(expected: 44, actual: pptm.Thumbnail["programming"]);
        }
    }
}
```

The unit test above found some inconsistencies where some words inside header or other non-standard elements in `.pptx` / `.pptm` files are not indexed properly. Pending for further fixes.

## Unit Test for PDF documents:

The code below is a simple unit test for `Index` functionality of `PdfDocument` class

```
using CustodianAPI.Utils;
using Xunit;

namespace CustodianAPI.Test
{
    public class PdfDocumentTest
    {
        [Fact]
        public void IndexTest()
        {
            // Given
            var filePath = SharedTestData.TestDocFolderPath + "Database System 

            // When
            var testDoc = new PdfDocument(filePath);

            // Then
            Assert.Equal(expected: 9, actual: testDoc.Thumbnail["university"]);
        }
    }
}
```

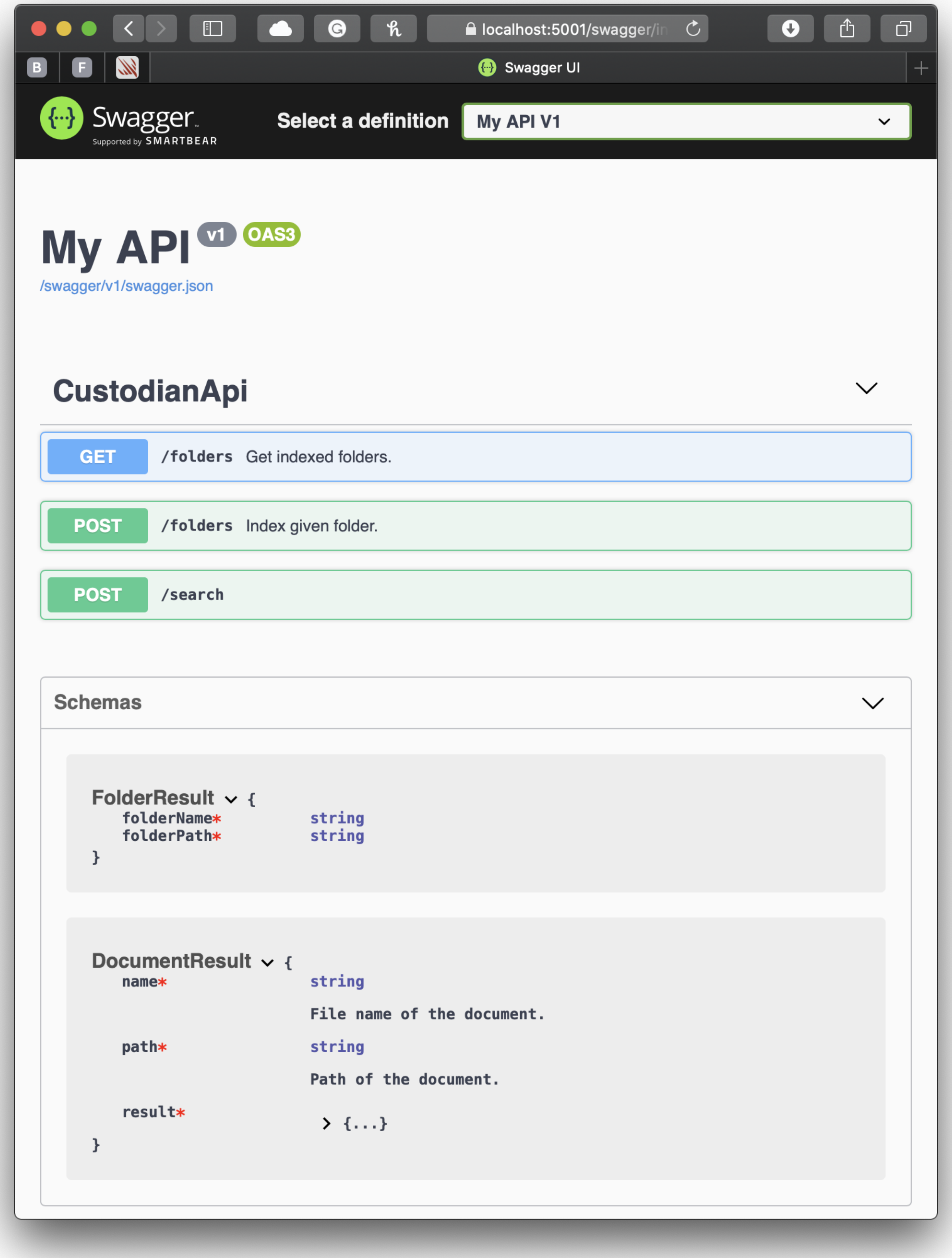
## Unit Test for Text documents:

```
using Xunit;
using CustodianAPI.Utils;

namespace CustodianAPI.Test
{
    public class TextTest
    {
        [Fact]
        public void IndexTest()
        {
            //Given
            var txt = new TextDocument(SharedTestData.TestDocFolderPath + "test.
            //When

            //Then
            Assert.Equal(30, txt.Thumbnail["castle"]);
        }
    }
}
```

# Web API

Web API was documented and tested with `SwaggerUI`.

## Web API Overview



My API v1 OAS3

/swagger/v1/swagger.json

CustodianApi

GET /folders Get indexed folders.

POST /folders Index given folder.

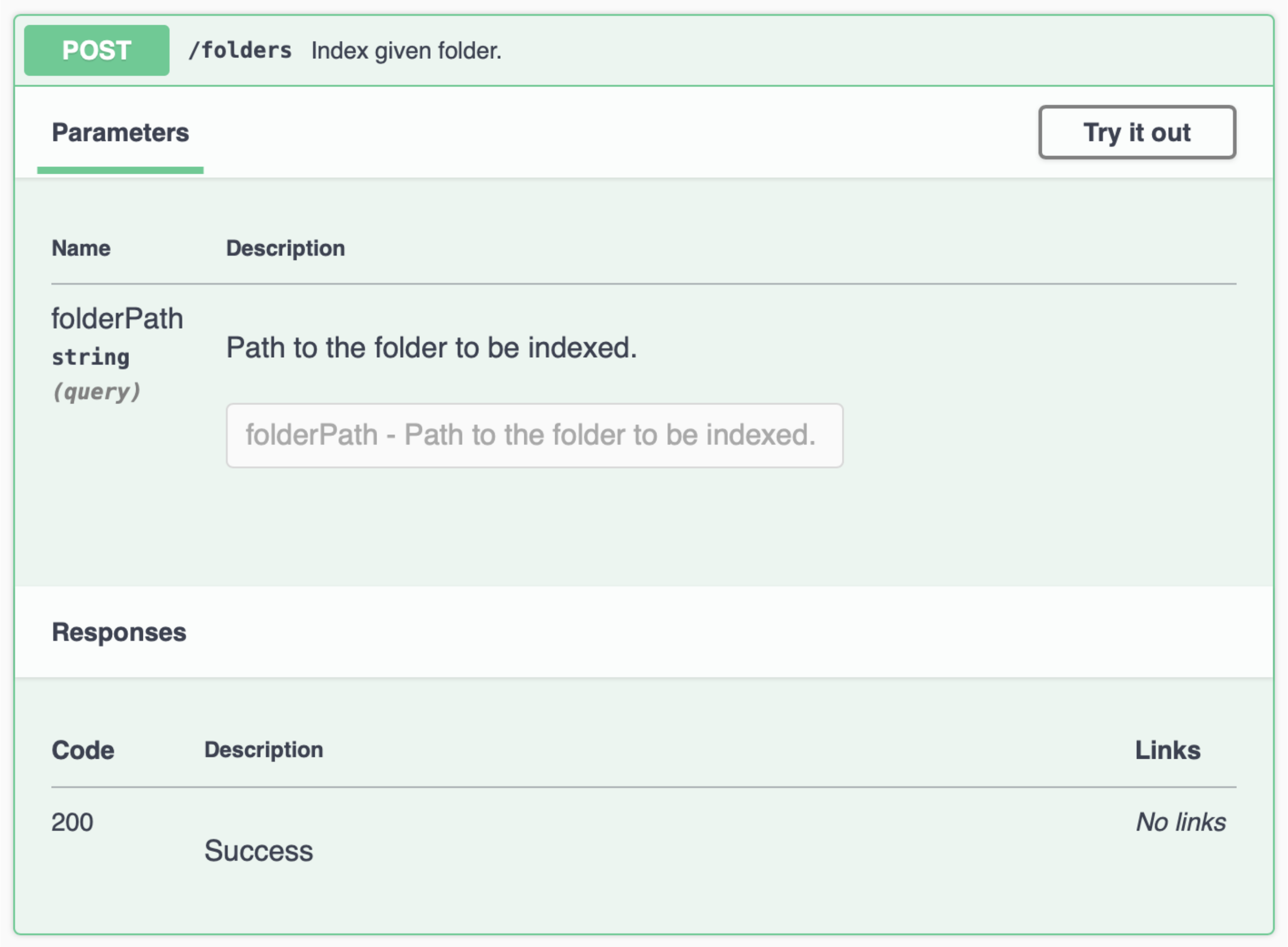
POST /search

Schemas

```
FolderResult ⌄ {
   folderName*        string
   folderPath*        string
}
```

```
DocumentResult ⌄ {
   name*              string

                      File name of the document.

   path*              string

                      Path of the document.

   result*             > {...}
}
```

## Web API Index Folder

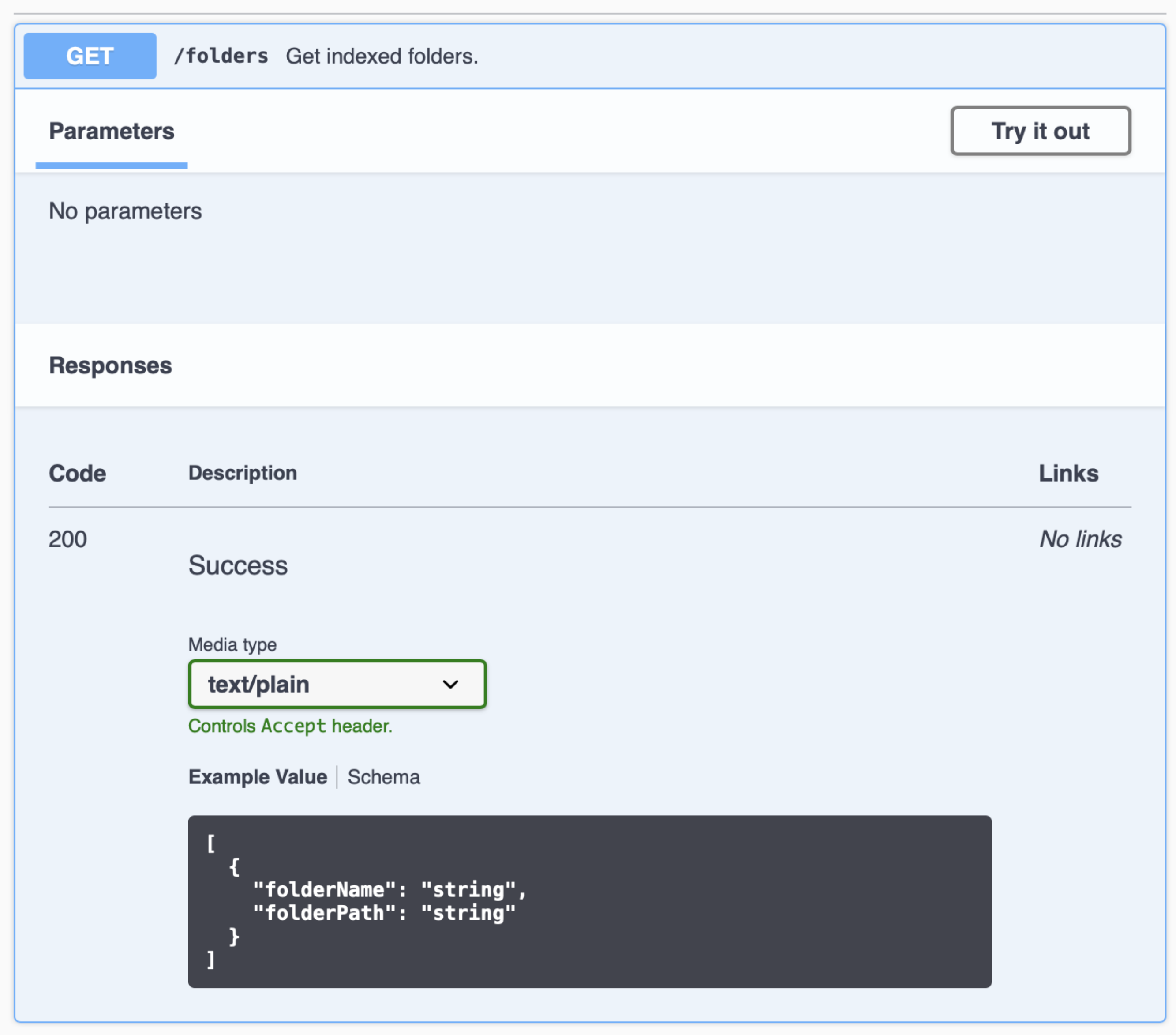


`/folders` Index given folder.

| Name | Description |
|---|---|
| folderPath<br>string<br>*(query)* | Path to the folder to be indexed.<br>folderPath - Path to the folder to be indexed. |

| Code | Description | Links |
|---|---|---|
| 200 | Success | *No links* |

## Web API Get Indexed Folders



No parameters

Responses

| Code | Description | Links |
|---|---|---|
| 200 | Success | *No links* |

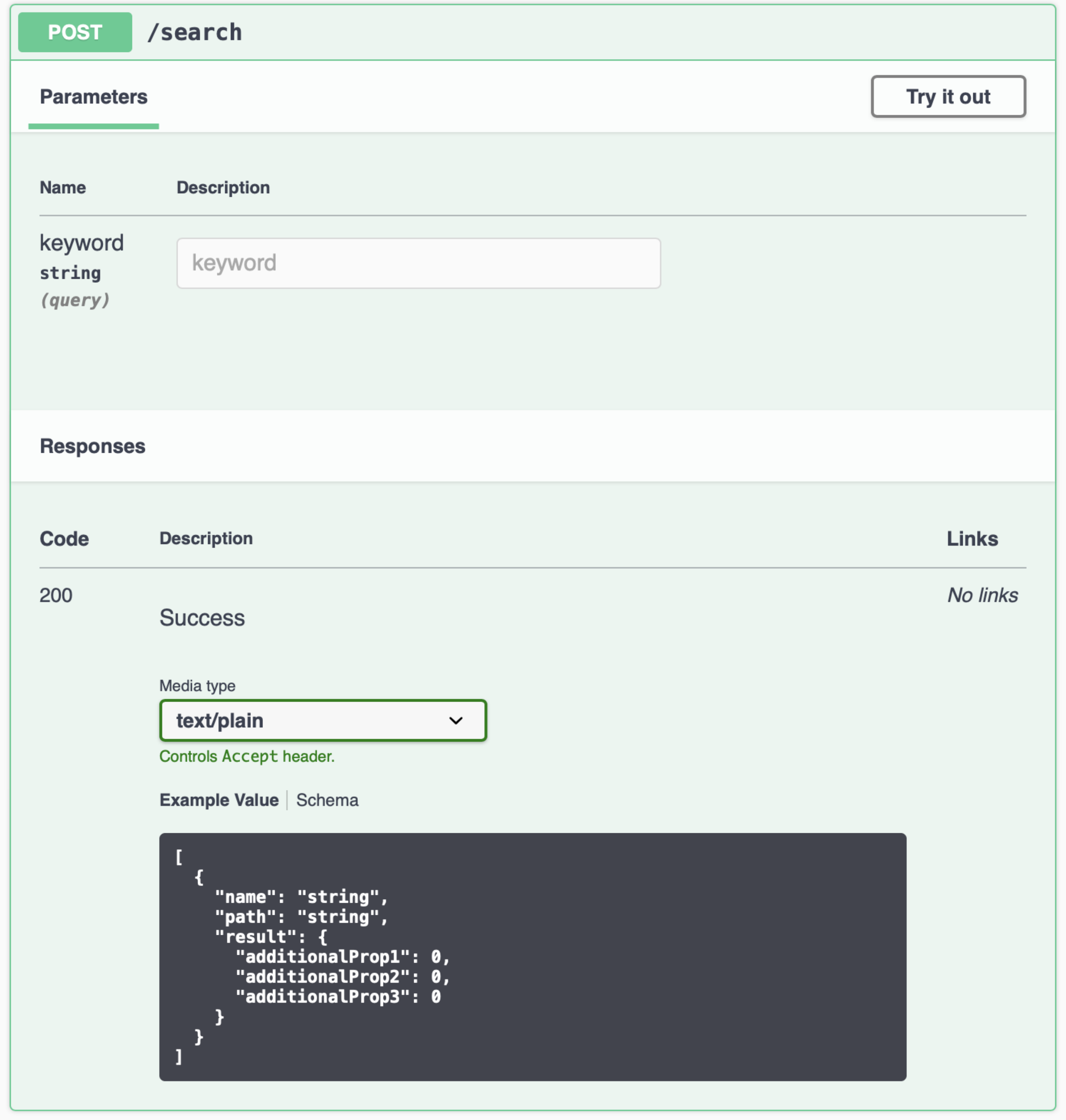
Media type

text/plain

Controls Accept header.

```
[
  {
    "folderName": "string",
    "folderPath": "string"
  }
]
```

## Web API Search



| Name | Description |
|---|---|
| keyword<br>string<br>*(query)* | keyword |

| Code | Description | Links |
|---|---|---|
| 200 | Success | *No links* |

Media type

text/plain

Controls Accept header.

```
[
  {
    "name": "string",
    "path": "string",
    "result": {
      "additionalProp1": 0,
      "additionalProp2": 0,
      "additionalProp3": 0
    }
  }
]
```

## UI

Basic functionality of cross-platform UI was tested manually. Some styling issue was found.

# Conclusion & Future Work

## Conclusion

The development progress can be found here.

Given the time constraint, unfortunately, i was unable to finish all planed features for this project, however most planned functionalities are well-implemented and tested.

Indexing and searching capabilities for most supported file types are fully functional. However some potential bug fixes are needed for some file type support (which are mentioned in the implementation chapter).

Web API was originally planned to use `Flask` (Python), instead of `ASP.NET Core`. That was changed due to performance concerns, as HTTP requests themselves are much slower than native function calls, and given the fact that `Python` as an interpreted language is relatively slower than `C#` (compiled language).

As for Cross-platform UI, it was originally planned to use `Electron`. Unfortunately, due to local file accessibility issues and flexibility concerns, `NW.js` was adopted working along with `Vue.js` serve as the cross-platform UI of this project. The UI at this stage is not fully completed as other features (e.g. Document Preview, etc.) are still pending to be implemented in future work.

During the development process, I was struggle with many tech stacks, i've also done many demos to find suitable stacks for my purpose. The API is always planned to used `.NET Core` for it's great performance and cross-platform capabilities.

Due the fact that native function calls is way faster than HTTP requests, I dropped planned web api + electron / nw.js structure and started exploring `Xamarin` framework (open-source cross-platform application framework using .NET and C#) for a possible work-around which later turned out to have some compatibility issues, as Xamarin.Mac is based on `.NET Standard` framework while the API is built with `.NET Core`. Thus I reverted back into web api + NW.js approach. However, there is a possibility where Custodian API is built with a CLI (Command-Line Interface) which GUI application can run shell commands to interact with Custodian API, this would possibly solve the latency issues with HTTP requests, and also improve usability to fit other use cases, this solution is for future reference.

# Future Work

This project was originally considered only to be a tool to do full-text searches against local file system. However, along the development process of the project, I discovered that there's a gap for full-text search (or text extraction, content analysis) framework in .NET ecosystem (even though Apache Tika has a .NET wrapper API, it's still not a pure C# implementation which might result in issues with a non-C# origin). Thus, I decided to focus mainly on API, try to push this into next steps as a library written in C# for full-text searching. Furthermore, as mentioned earlier Custodian API as a library should also have a CLI to improve usability.

Up until this point, there're still many dependencies with various licences which can be tricky for potential use cases in the future. That said, until all planned features of this project is implemented and fully tested, I'll then look into replacing some dependencies that have licence issues with other licence compatible libraries or possibly my own implementations of the dependencies.

Hopefully, this project can be an alternative to Apache Tika in C# ecosystem.

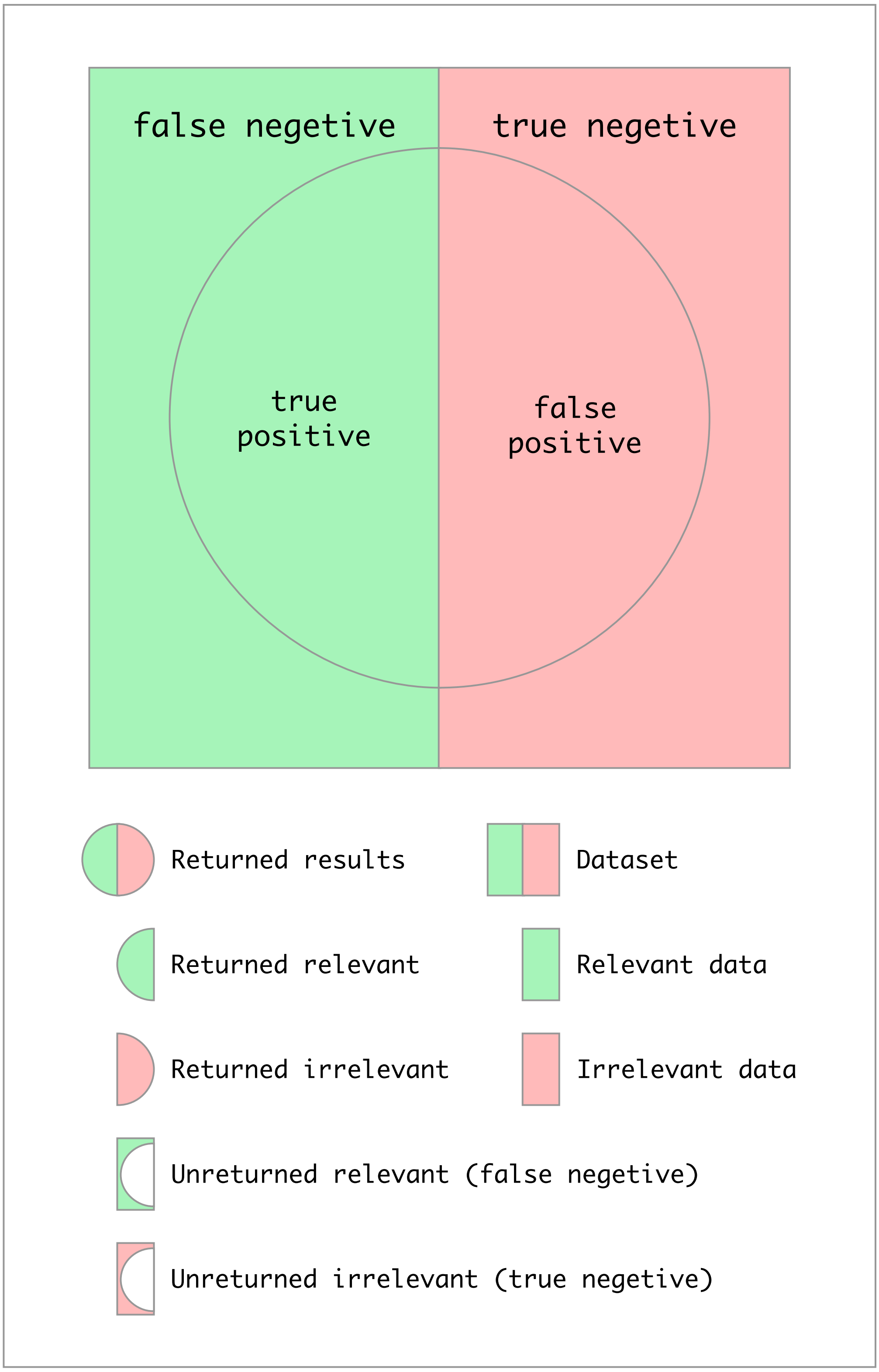
# References

- Apache Lucene, 2020. *Apache Lucene - Welcome to Apache Lucene*. Available at https://lucene.apache.org. Accessed on 7th Aug 2020.
- Apache Tika, 2020. *Apache Tika - Apache Tika*. Available at https://tika.apache.org. Accessed on 7th Aug 2020.
- Axios, 2020. *axios/axios: Promise based HTTP client for the browser and node.js | GitHub*. Available at https://github.com/axios/axios. Accessed on 12th August 2020.
- DocFetcher, 2020. *DocFetcher - Fast Document Search*. Available at http://docfetcher.sourceforge.net/en/index.html. Accessed on 7th Aug 2020.
- ECMA, 2017. *C# Language Specification | ECMA International*. Available at https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf. Accessed on 13th August 2020.
- Erickson, 2008. *sql - What is Full Text Search vs LIKE - Stack Overflow*. Available at https://stackoverflow.com/a/224726/8702601. Accessed on July 1st 2020.
- iText, 2020. *iText 7 Core: an open-source PDF development library for Java and .NET*. Available at https://itextpdf.com/en/products/itext-7/itext-7-core. Accessed on 13 August 2020.
- Makhoul, J., Kubala, F., Schwartz, R. and Weischedel, R., 1999, February. Performance measures for information extraction. In Proceedings of DARPA broadcast news workshop (pp. 249-252).
- MDN, 2020. *Promise - JavaScript | MDN*. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed on 12th August 2020.
- Melton, J. Buxton, S., 2006. *Chapter 13 - What's Missing? - Querying XML | ScienceDirect*. Available at https://doi.org/10.1016/B978-155860711-8/50014-9. Accessed on July 1st 2020.
- Microsoft Docs, 2015. *Introduction to the C# Language and the .NET Framework | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework. Accessed on 13th August 2020.
- Microsoft Docs, 2017. *Structure of a PresentationML document (Open XML SDK) | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/office/open-xml/structure-of-a-presentationml-document. Accessed on 12th August 2020.
- Microsoft Docs, 2017. *Structure of a SpreadsheetML document (Open XML SDK) | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/office/open-xml/structure-of-a-spreadsheetml-document. Accessed on 12th August 2020.
- Microsoft Docs, 2017. *Structure of a WordprocessingML document (Open XML SDK) | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/office/open-xml/structure-of-a-wordprocessingml-document. Accessed on 5th August 2020

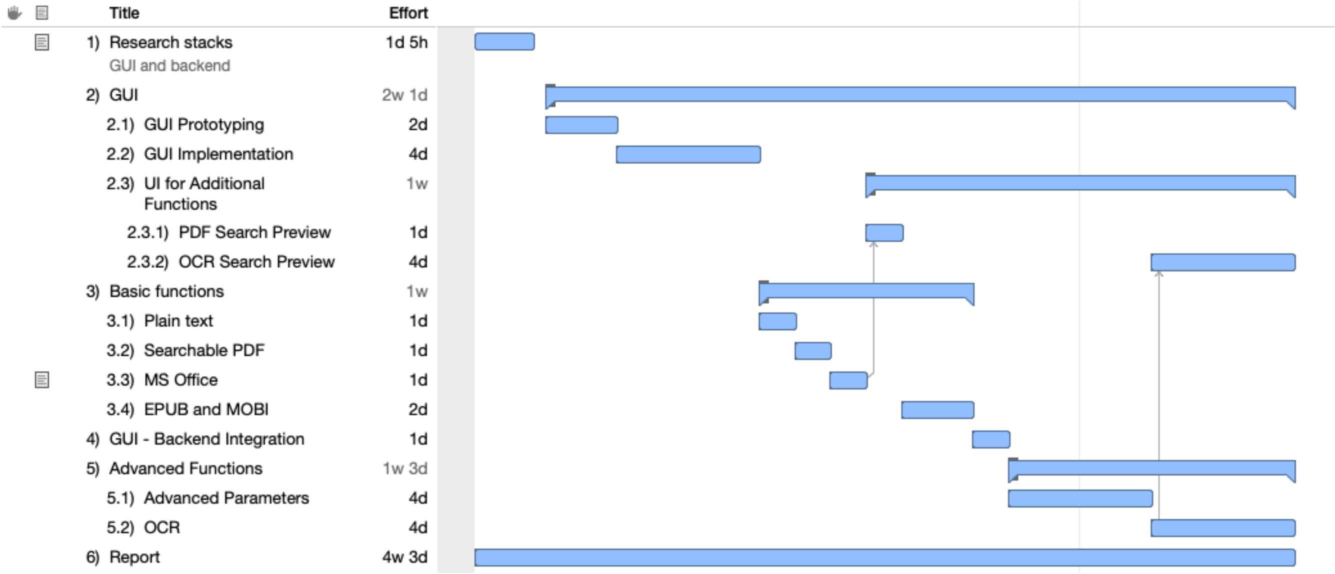
- Microsoft Docs, 2018. *Full-Text Search - SQL Server | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver15. Accessed on July 1st 2020.
- Microsoft Docs, 2020. *.NET Core intro and overview | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/dotnet/core/introduction. Accessed on 13th August 2020.
- Microsoft Docs, 2020. *Introduction to ASP.NET Core | Microsoft Docs*. Available at https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-3.1. Accesed on 13th August 2020.
- Microsoft, 2020.*TypeScript: Typed JavaScript at Any Scale*. Available at https://www.typescriptlang.org. Accessed on 13th August 2020.
- NW.js, 2020. *nwjs/nw.js: Call all Node.js modules directly from DOM/WebWorker and enable a new way of writing applications with all Web technologies*. Available at https://github.com/nwjs/nw.js. Accessed on 13th August.
- Office Open XML, 2012. *Office Open XML - What is OOXML?*. Available at https://docs.microsoft.com/en-us/office/open-xml/structure-of-a-wordprocessingml-document. Accessed on 5th August 2020.
- Reiner, K. Chichao, C. Farzin, M. Ravi, K.,2006. *Searching with context | ACM Digital Library*. Available at https://dl.acm.org/doi/abs/10.1145/1135777.1135847. Accessed on July 1st 2020.
- Search Engine Watch, 2019. *The beginner's guide to semantic search: Examples and tools*. Available at https://www.searchenginewatch.com/2019/12/16/the-beginners-guide-to-semantic-search/. Accessed on 13th August 2020.
- Tekli, J., Chbeir, R., Traina, A., Traina, C Jr., Yetongnon, K., Ibanez, C., Assad, M., Kallas, C., 2018. *Full-fledged semantic indexing and querying model designed for seamless integration in legacy RDBMS*. Available at https://doi-org.ezproxy.herts.ac.uk/10.1016/j.datak.2018.07.007. Accessed on July 2nd 2020.
- VSCode, 2020. *Visual Studio Code - Code Editing. Redefined*. Available at https://code.visualstudio.com. Accessed on 12th August 2020.
- Vue.js, 2020. *Vue.js*. Available at https://vuejs.org. Accessed on 12th August 2020.

# Appendix

a. Precision & Recall | Back to content

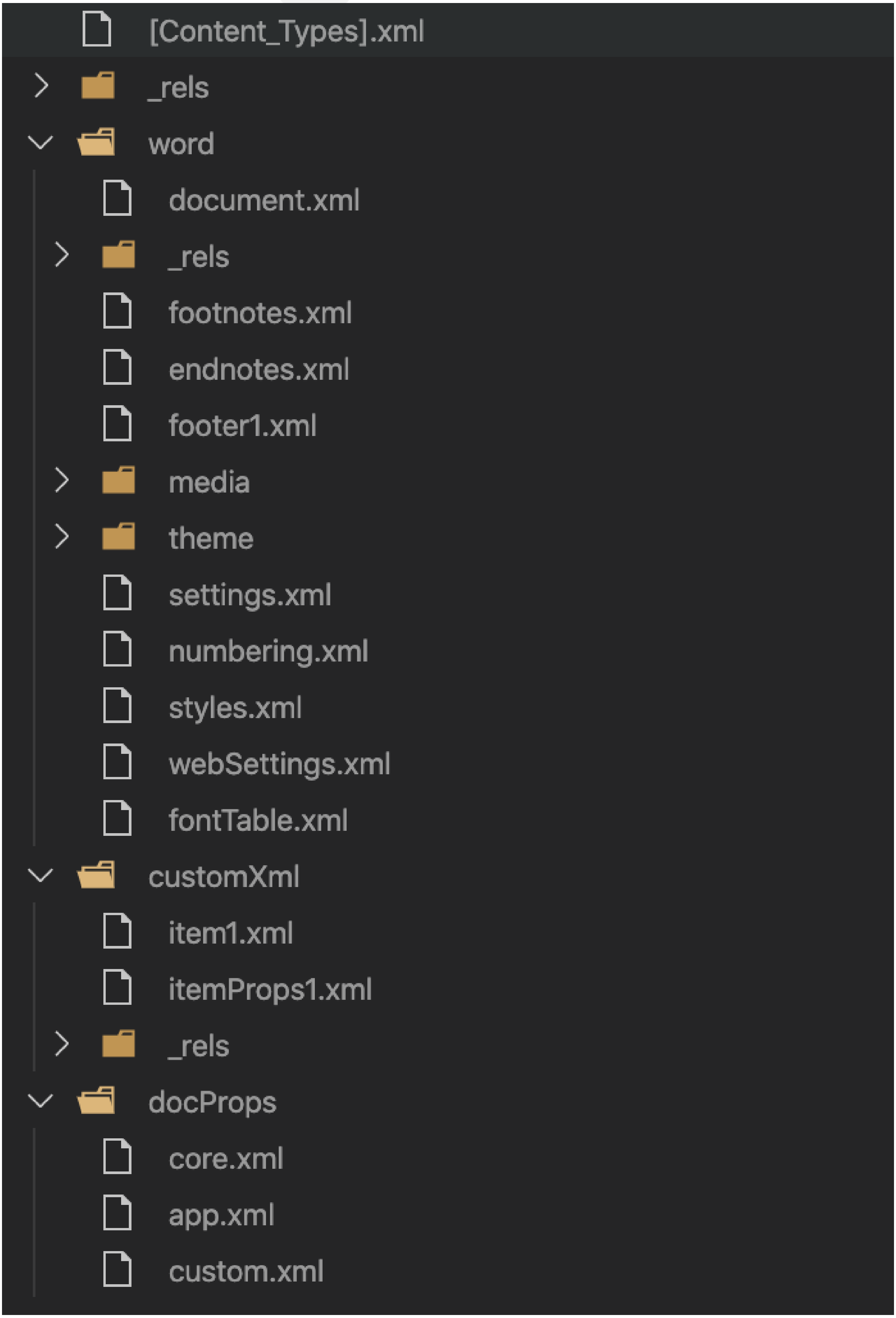


## b. Gantt Chart | Back to content



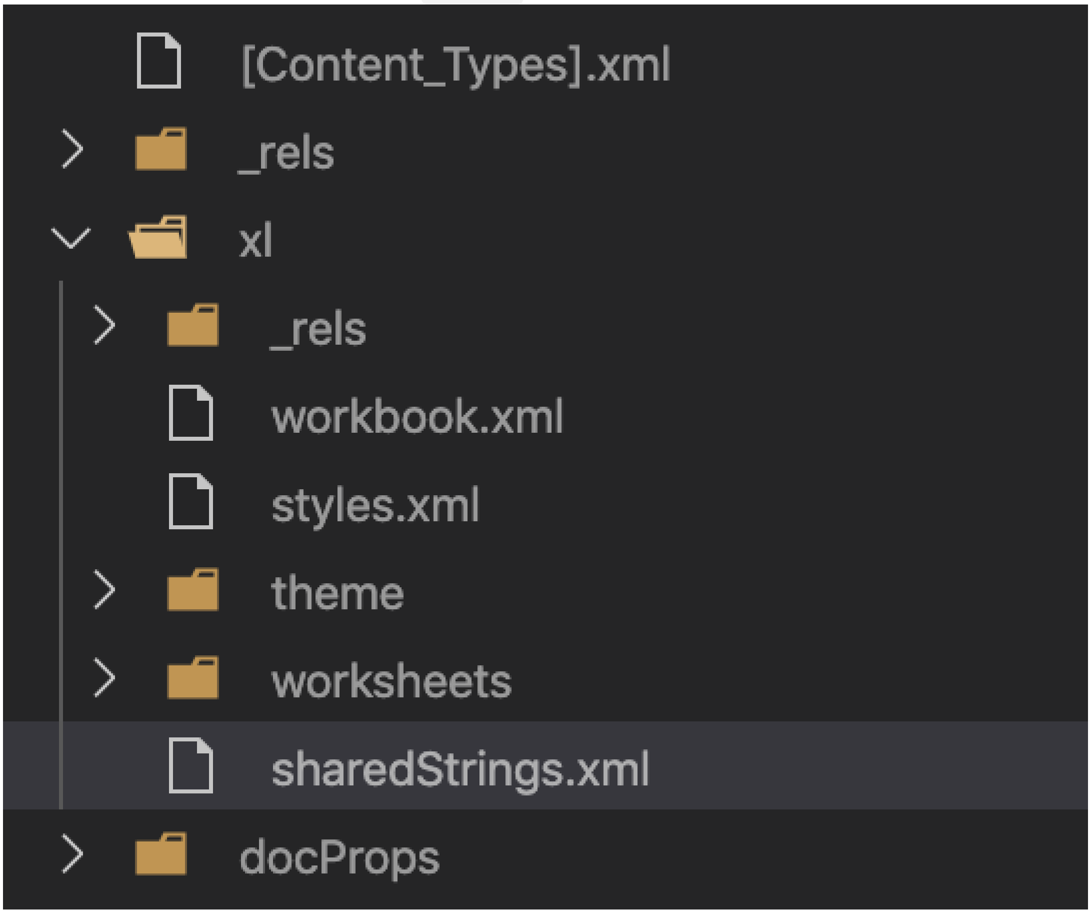
| Title | Effort |
|---|---|
| 1) Research stacks<br>GUI and backend | 1d 5h |
| 2) GUI | 2w 1d |
| 2.1) GUI Prototyping | 2d |
| 2.2) GUI Implementation | 4d |
| 2.3) UI for Additional Functions | 1w |
| 2.3.1) PDF Search Preview | 1d |
| 2.3.2) OCR Search Preview | 4d |
| 3) Basic functions | 1w |
| 3.1) Plain text | 1d |
| 3.2) Searchable PDF | 1d |
| 3.3) MS Office | 1d |
| 3.4) EPUB and MOBI | 2d |
| 4) GUI - Backend Integration | 1d |
| 5) Advanced Functions | 1w 3d |
| 5.1) Advanced Parameters | 4d |
| 5.2) OCR | 4d |
| 6) Report | 4w 3d |

c. Structure of decompressed `.docx` file | Back to content



```
[Content_Types].xml
> _rels
v word
    document.xml
  > _rels
    footnotes.xml
    endnotes.xml
    footer1.xml
  > media
  > theme
    settings.xml
    numbering.xml
    styles.xml
    webSettings.xml
    fontTable.xml
v customXml
    item1.xml
    itemProps1.xml
  > _rels
v docProps
    core.xml
    app.xml
    custom.xml
```

d. Structure of decompressed `.xlsx` file | Back to content



```
[Content_Types].xml
> _rels
∨ xl
    > _rels
      workbook.xml
      styles.xml
    > theme
    > worksheets
      sharedStrings.xml
> docProps
```