

**Bachelor Project**



**Czech  
Technical  
University  
in Prague**

**F3**

**Faculty of Electrical Engineering  
Department of Computer Science**

## **Semantic Document Manager User Interface**

**Valeryia Chyzhova**

**Supervisor: Ing. Martin Ledvinka  
Subfield: Software Engineering and Technology  
May 2021**



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Chyzhova** Jméno: **Valeryia** Osobní číslo: **478038**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/ústav: **Katedra počítačů**  
Studijní program: **Softwarové inženýrství a technologie**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Uživatelské rozhraní Sémantického správce dokumentů**

Název bakalářské práce anglicky:

**Semantic Document Manager User Interface**

Pokyny pro vypracování:

Seznam doporučené literatury:

- [1] M. Ledvinka, P. Křemen, L. Saeeda, M. Blaško: Termlt: A Practical Vocabulary Manager, Proceedings of the 22nd International Conference on Enterprise Information Systems, 2020
- [2] M. Jaroš, Semantic Document Manager, Bachelor's thesis, 2020
- [3] J. J. Garrett, The Elements of User Experience: User-Centered Design for the Web and Beyond, New Riders, 2010

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **15.02.2021**

Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **19.02.2023**

\_\_\_\_\_  
Ing. Martin Ledvinka  
podpis vedoucí(ho) práce

\_\_\_\_\_  
podpis vedoucí(ho) ústavu/katedry

\_\_\_\_\_  
prof. Mgr. Petr Páta, Ph.D.  
podpis děkana(ky)

## III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studentky

## Acknowledgements

I would like to thank my supervisor Ing. Martin Ledvinka for providing guidance and feedback throughout this project.

## Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May , 2021

## Abstract

This bachelor thesis continues the development of a document management system, which is called the Document Manager. The Document Manager is an application that can structure and store files.

It was created as a part of the bachelor thesis written by Martin Malinov. Then the application back-end was extended by Marek Jaroš in his bachelor thesis.

The purpose of this work is to create the front-end of the Document Manager application, which will have intuitive and user-friendly interface.

The result of the bachelor thesis is a React application, which allows a user to easily manipulate the data within the application, namely, to create the entities, upload Files to a file system, change the metadata of the entities, manage user access rights, etc.

**Keywords:** document manager, web application, front-end, software, REST API, JavaScript, React, UI design, UX design

**Supervisor:** Ing. Martin Ledvinka

## Abstrakt

Tato bakalářská práce navazuje na vývoj správce dokumentů s názvem Document Manager. Document Manager je aplikace, která umí strukturovat a ukládat soubory.

Document Manager byl vytvořený jako součást bakalářské práce Martina Malinova. Poté backendová část aplikace byla rozšířena v rámci bakalářské práce Marka Jaroše.

Cílem této práce je vytvořit front-endovou část aplikace Document Manager, která bude mít intuitivní a uživatelsky přívětivé rozhraní.

Výsledkem bakalářské práce je React aplikace, která umožňuje uživateli snadno manipulovat s daty v aplikaci, vytvářet entity, nahrávat soubory do souborového systému, měnit metadata entit, spravovat přístupová práva uživatelů, atd.

**Klíčová slova:** document manager, webová aplikace, frontend, software, REST API, JavaScript, React, UI design, UX design

# Contents

<b>1 Introduction</b>	<b>1</b>	4.2 Technical approaches to the SPA development	22
1.1 Main functions of the Document Manager	1	4.2.1 Vue.js	23
1.2 Current situation	1	4.2.2 Angular	23
1.3 Aim of the work	2	4.2.3 React	24
<b>2 Back-end analysis</b>	<b>3</b>	4.2.4 Conclusion	24
2.1 Back-end overview	3	4.3 Application structure	25
2.1.1 Architecture	4	4.3.1 Assets	25
2.1.2 Types of the Document Manager entities	4	4.3.2 Helpers	26
2.1.3 File versioning	5	4.3.3 Features	26
2.1.4 User roles	5	4.3.4 Components	27
2.1.5 Access rights	5	<b>5 UI design</b>	<b>29</b>
2.2 Document Manager API	6	5.1 Authorization	29
2.2.1 REST API interface	6	5.2 File system display	30
2.2.2 Data formats	6	5.3 Operations implementation	30
2.2.3 API endpoints for the front-end implementation	7	5.3.1 Document and Folder operations	30
2.2.4 Weaknesses in the REST API implementation	9	5.3.2 File operations	33
2.3 Document Manager functionalities	10	5.4 Access rights management	34
2.4 Use Case Diagram	10	5.5 User management	34
<b>3 UX research</b>	<b>13</b>	<b>6 Technical solution</b>	<b>37</b>
3.1 Motivation	13	6.1 Authorization	37
3.2 Document Manager functionality	13	6.1.1 External authorization service	37
3.3 Usability	13	6.1.2 Token validity	37
3.4 The importance of a clear visual hierarchy	14	6.2 Routing	38
3.4.1 Main traits of a clear visual hierarchy	14	6.3 Hooks	38
3.4.2 Adding a Document	15	6.3.1 Custom hooks	38
3.4.3 Place of a logotype in a visual hierarchy	15	6.3.2 useState	39
3.5 Web navigation	15	6.3.3 useEffect	39
3.5.1 Home page button	16	6.4 Client/Server state synchronization	40
3.5.2 Search button	17	6.4.1 React Query concepts	40
3.5.3 Operation icons	17	6.5 React.memo	41
<b>4 Front-end plan</b>	<b>19</b>	6.6 Styling	41
4.1 Concepts of front-end development	19	<b>7 Usability testing</b>	<b>43</b>
4.1.1 Single-page application vs Multi-page application	19	7.1 Usability Testing Method	43
4.1.2 Server-side rendering vs Client-side rendering	20	7.2 Test scenarios	43
4.1.3 Conclusion	21	7.2.1 Basic functionality test	43
		7.2.2 Access rights test	44
		7.3 Results	44
		7.3.1 Folder user rights management UI/UX problem	45
		7.3.2 WRITE access level does not work	45

7.3.3 Filename does not update in a file system . . . . .	45
7.3.4 File version does not update after a new version is added . . . . .	46
7.3.5 A new folder display . . . . .	46
7.3.6 Problem with changing a filename . . . . .	46
7.3.7 File information window does not close after the File is deleted . . . . .	46
7.4 Changes . . . . .	47
<b>8 Conclusion</b>	<b>49</b>
<b>A Bibliography</b>	<b>51</b>
<b>B List of abbreviations</b>	<b>55</b>
<b>C Instructions how to start the Document Manager application</b>	<b>57</b>
C.1 Back-end part . . . . .	57
C.2 Front-end part . . . . .	58
<b>D Content of the electronic attachment</b>	<b>59</b>

## Figures

3.1 An example of a clear visual hierarchy in the list of File versions	14
3.2 An example of a clear visual hierarchy in the file system	15
3.3 An example of placing a logotype on the page	16
3.4 The initial state of the application	16
3.5 An example of placing a search field on the page	17
4.1 A server-side rendering schema (taken from [3])	21
4.2 A client-side rendering schema (taken from [3])	22
4.3 The popularity scale of the six most well-known Javascript frameworks	23
4.4 The structure of the Document Manager front-end (taken from Gitlab FEL)	26
5.1 The page shown to an unauthorized user	29
5.2 KBSS Authorization Service, which is used in the Document Manager application	30
5.3 The file system in the Document Manager application	31
5.4 The operations that can be performed on a Document or Folder	31
5.5 The modal window for adding a new Document or Folder	31
5.6 The modal window for adding a new File (before the File is chosen)	32
5.7 The modal window for adding a new File (after the File is chosen)	32
5.8 The operations that can be performed on a File	33
5.9 The window with the information about the File and its versions	34
5.10 The modal window, which contains the user management section	35

## Tables

2.1 The possible operations that can be performed in the Document Manager application	12
7.1 The problems identified while the usability testing and their solutions	47





# Chapter 1

## Introduction

This work is a front-end<sup>1</sup> part of the application, which is called the Document Manager. Back-end<sup>2</sup> development was a part of the bachelor thesis written by Martin Malinov [27], who created a basic application for storing files. Then, the back-end was extended within the bachelor thesis of Marek Jaroš [18], who added the possibility of user management and the Semantic Web technology<sup>3</sup>.

The work describes the stages of building a front-end part of a modern web application, including the back-end analysis, the comparison of technologies for web development and UX<sup>4</sup> research. The results of analysis are then applied to the application design and implementation.

### 1.1 Main functions of the Document Manager

The Document Manager(DM) is an application that can structure and store files. A user can manipulate the data within the application, i.e. add, change or delete it. The application saves file versions (that are called file mementos) and allows the user to get access to them. Additionally, the Document Manager has access rights management and several access levels.

### 1.2 Current situation

Due to the fact that the Document Manager has only the back-end part, it is not convenient to use it. It is much simpler to understand the application possibilities, when it has User Interface to interact with.

---

<sup>1</sup>Everything a user may interact with to use a digital product or service

<sup>2</sup>Any part of a website or software that users do not see

<sup>3</sup>The goal of the Semantic Web is to structure the information on the Web so that it was understandable not only to a human but also to a machine [18]

<sup>4</sup>UX is an acronym that stands for User eXperience. It is the study of the interaction between users and a system [28]

## ■ 1.3 Aim of the work

The aim of the work is to analyze the back-end part of the Document Manager application and its REST API; to study approaches to front-end development and to choose the appropriate one; to design and implement the front-end of the Document Manager and to conduct usability testing.

## Chapter 2

### Back-end analysis

The analysis of the already existing back-end is an integral research before starting front-end development. The back-end and front-end work together to create a full user experience. The data generated on the back-end side is passed to the front-end and then presented to the user. That is why it is necessary to study and analyze the back-end part thoroughly to efficiently use gained information while implementing the front-end.

#### 2.1 Back-end overview

The back-end part of the DM is written in Java using the Spring framework. It is responsible for storing and organizing the DM data, uploading files to a file system and managing user roles and their access rights.

Figure 2.1 illustrates how the back-end works.

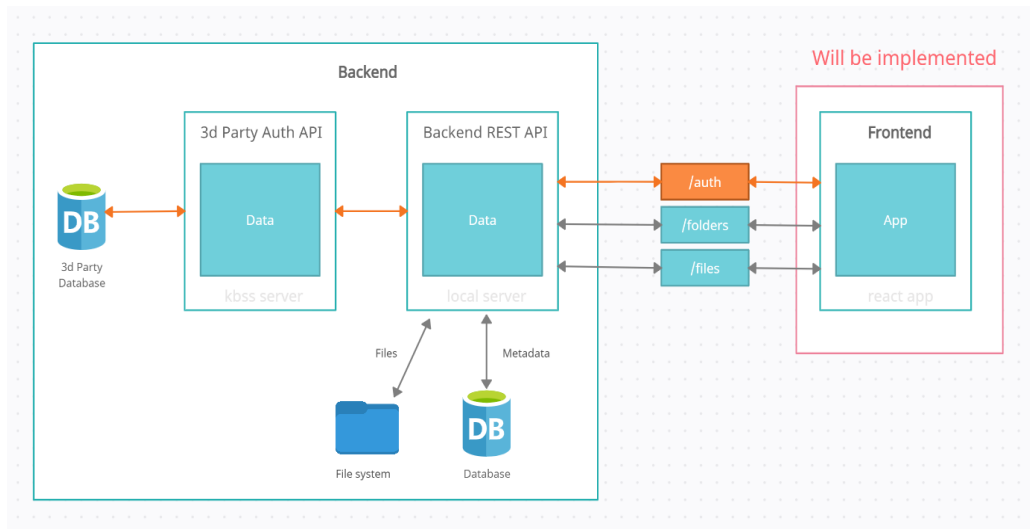


Figure 2.1: The back-end overview (Made in Creately<sup>1</sup>)

<sup>1</sup>Creately is an easy to use diagram and flowchart software built for team collaboration. Supports over 40+ diagram types and has 1000's of professionally drawn templates [26].

### 2.1.1 Architecture

The architecture is designed using a four-layered pattern, specifically it contains **Controller**, **Service**, **Persistence** and **Data** layers. A layered pattern allows the division of the responsibilities of individual layers by separating the client interface from the business logic and the business logic from the database one [30]. The diagram of the system architecture is shown in Figure 2.2

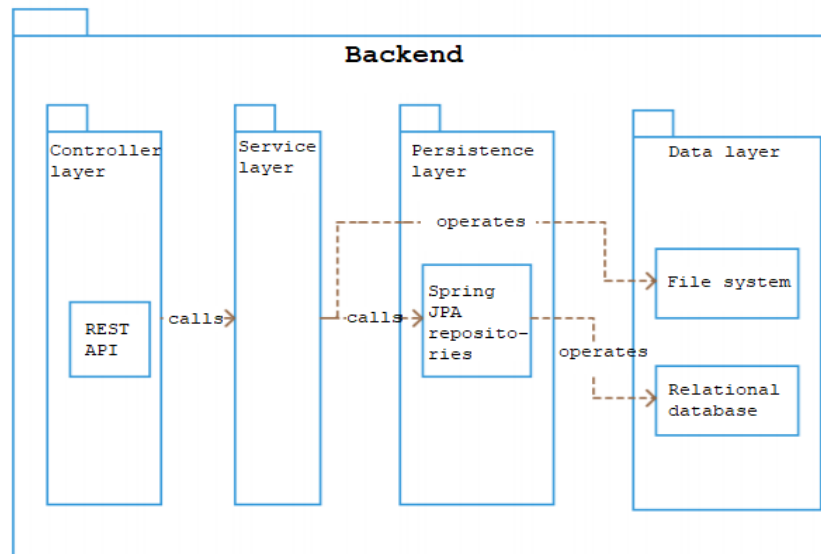


Figure 2.2: The Document Manager architecture [27]

### 2.1.2 Types of the Document Manager entities

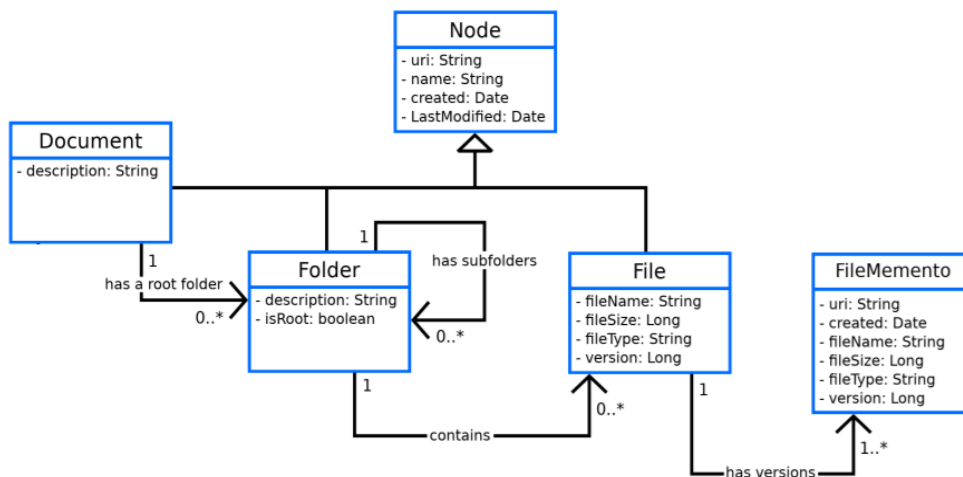
The majority of entities is of a type **Node**. Nodes can be divided into three groups: **Documents**, **Folders** or **Files** - based on their properties. Nodes have unique IDs within the whole application and associations with each other (for example, a parent-child association between a Folder and its content).

The highest position in the DM hierarchy has a Document. It acts as a container for Folders and Files and it always contains a **root** Folder. The root Folder is the only Folder, which does not have a parent Folder.

A **basic** Folder has a lower-level position in the DM hierarchy. It can contain other Folders and Files, but it always has a parent Folder.

The lowest position is held by a File. It can be uploaded by a user to a Folder. A physical File is uploaded to the DM file system and its metadata, which contain a path to the File, is saved to a database. Files can also have several versions.

Figure 2.3 shows the Document Manager entities, their parameters and relationships between entities.



**Figure 2.3:** The Document Manager entities and relationships between them [18]

### ■ 2.1.3 File versioning

When a user wants to update File content, a new version of the File is created. The initial upload is marked as a version zero, the second upload – version one, etc. All File versions are saved in the DM file system and the user has a possibility to download them.

### ■ 2.1.4 User roles

There are two main roles in the DM application: **User** and **Admin**. A User can create Documents, Folders and Files. An Admin is responsible for managing users: only he has the rights to create, update or delete users [18]. Also the Admin can create groups, add users to them or remove users. For example, in Listing 2.1 you can see the endpoint to add a user to a group. This operation can be done only by the Admin.

```

1 POST groups/TestGroup/users?namespace=http://example.cz/
  UserGroup
2 Body: http://onto.fel.cvut.cz/ontologies/uzivatel/user-name
  
```

**Listing 2.1:** The endpoint to add a user to a group

### ■ 2.1.5 Access rights

Access rights can be assigned only to Documents. The access rights are divided into four groups:

- **None** - a user has no rights to access this entity and it is not shown for him in the DM hierarchy;
- **Read** - a user can read the content of this entity, but he has no rights to manipulate (change, delete, etc.) with it;



## ■ JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. JSON is a text format that is completely language-independent. These properties make JSON an ideal data-interchange language [12].

```

1 {
2   "uri": "http://example.cz/Document/TestDocument",
3   "name": "Test document",
4   "description": "A document created for testing purposes"
5 }
```

**Listing 2.2:** The body of a POST request in the JSON format

## ■ JSON-LD

JSON-LD (JavaScript Object Notation for Linked Data) is a JSON-based format to serialize Linked Data<sup>2</sup>.

According to Google Search Central<sup>3</sup> JSON-LD better describes the content of the website and makes a Google search engine to understand the content more effectively, therefore the content of the website will be featured more relevantly.

The Document Manager is an internal application that is why it is not important to increase its visibility for relevant searches. For this reason, I will use the JSON data format in front-end development.

```

1 {
2   "@id": "http://example.cz/Document/TestDocument",
3   "@type": [
4     "http://example.cz/Document"
5   ],
6   "http://example.cz/name": "Test document",
7   "http://example.cz/description": "A document created for
8     testing purposes"
```

**Listing 2.3:** The body of a POST request in the JSON-LD format

### ■ 2.2.3 API endpoints for the front-end implementation

The goal of this subsection is to study and analyze the REST API of the Document Manager and to describe the endpoints needed to create a dynamic web service.

<sup>2</sup>Linked Data is simply about using the Web to create typed links between data from different sources. Technically, Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets [5].

<sup>3</sup>Google Search Central helps the right people to view the content with resources to make the website discoverable to Google Search [8]



## ■ Document management

As the application name suggests, the main function of the Document Manager is managing documents. In our case, the data can be divided into three groups: **Documents**, **Folders** and **Files**. A user can browse them if he has access, create a new one, update or delete the existing ones. The DM API provides the endpoints that enable the above-mentioned functionality for all the types of data.

An example of the Document management is given below.

- get all Documents to which a user has access:

```
1 GET /documents
```

- create a new Document and add it to the list of user's Documents:

```
1 POST /documents
```

Listings 2.2 and 2.3 (the only difference between them is the data format) illustrate the structure of the request body.

- update the specific document (**namespace** is an URL parameter):

```
1 PUT /documents/documentName?namespace=http://example.cz/  
Document
```

The request body is the same as in the POST request.

- delete the specific document (**namespace** is an URL parameter):

```
1 DELETE /documents/documentName?namespace=http://example.cz/  
Document
```

## ■ Version control

Previous File versions are called File mementos. A user can get all File mementos or a specific memento. It depends on the endpoint that will be requested:

- get all mementos of a File:

```
1 GET /files/TestFile/versions?namespace=http://example.cz/  
File
```

- get the specific memento:

```
1 GET /files/TestFile/versions/0?namespace=http://example.cz/  
File
```

The version is specified using a number (in this case, the version is 0).

## 2.2.4 Weaknesses in the REST API implementation

During the analysis of the REST API endpoints I have found some things that were not carefully considered in the back-end implementation. These weak places in the back-end architecture caused problems in front-end development.

### Not convenient solution for a file system display

The only way to display a file system is to get the list of Documents as a first step and then to make separate requests for getting the content of every single Document. The more convenient way is to return all Folders with defined relationships between entities.

### Different endpoints for getting Folder data

Imagine the situation when a Folder has both subfolders and Files. I have to make two requests to get at first all the subfolders and then all the Files.

- get the list of the Folder subfolders:

```
1 GET /folders/folderName/subfolders?namespace=http://
   example.cz/Folder
```

- get the list of the Folder Files:

```
1 GET /folders/folderName/files?namespace=http://example.
   cz/Folder
```

The better solution is to make one request that gets all the items stored in the current folder.

### Different structure of a File and a File memento

There is an option for a user to change a filename when he uploads a File to the file system. It does not matter if the user wants to upload a new File or a new File version, he always has such a possibility.

The difficulties start when I want to get the name the user gave to the File version, because unlike the File, the File version contains only the initial file name. There are two API responses in the listings below: when I request a File (Listing 2.4) and when I request one of its versions (Listing 2.5).

```
1 {
2   "@id": "http://example.cz/File/Test.html",
3   "@type": [
4     "http://example.cz/File",
5     "http://example.cz/Node"
6   ],
7   "http://example.cz/version": 1,
8   "http://example.cz/fileType": "image/png",
9   "http://example.cz/created": 1620116546771,
10  "http://example.cz/fileSize": 74176,
11  "http://example.cz/lastModified": 1620116560959,
12  "http://example.cz/name": "Test",
```

```

13   "http://example.cz/fileName": "schema.PNG"
14 }

```

**Listing 2.4:** The API response when a File is requested

```

1 {
2   "@id": "http://example.cz/FileMemento/Test.html_0",
3   "@type": [
4     "http://example.cz/FileMemento"
5   ],
6   "http://example.cz/fileType": "image/png",
7   "http://example.cz/version": 0,
8   "http://example.cz/created": 1620116546771,
9   "http://example.cz/fileSize": 74176,
10  "http://example.cz/fileName": "schema.PNG"
11 }

```

**Listing 2.5:** The API response when a File version 0 is requested

The File version response does not have the "http://example.cz/name" key, therefore I cannot get the name of the version given by the user.

## 2.3 Document Manager functionalities

Based on the analysis of the API endpoints, I defined the main operations that can be performed in the Document Manager application. All of them are represented in the Table 2.1.

## 2.4 Use Case Diagram

To map the functionalities of the DM application, I have created a use case diagram, which demonstrates the different ways a user might interact with the application.

A use case diagram consists of use cases, actors and relationships between them. Actor is a role that communicates with individual use cases. A user or an external system can participate in this role [6].

I can specify two actors in the DM application: an authorized user and an admin. All the use cases of the Document Manager application can be found in Figure 2.4

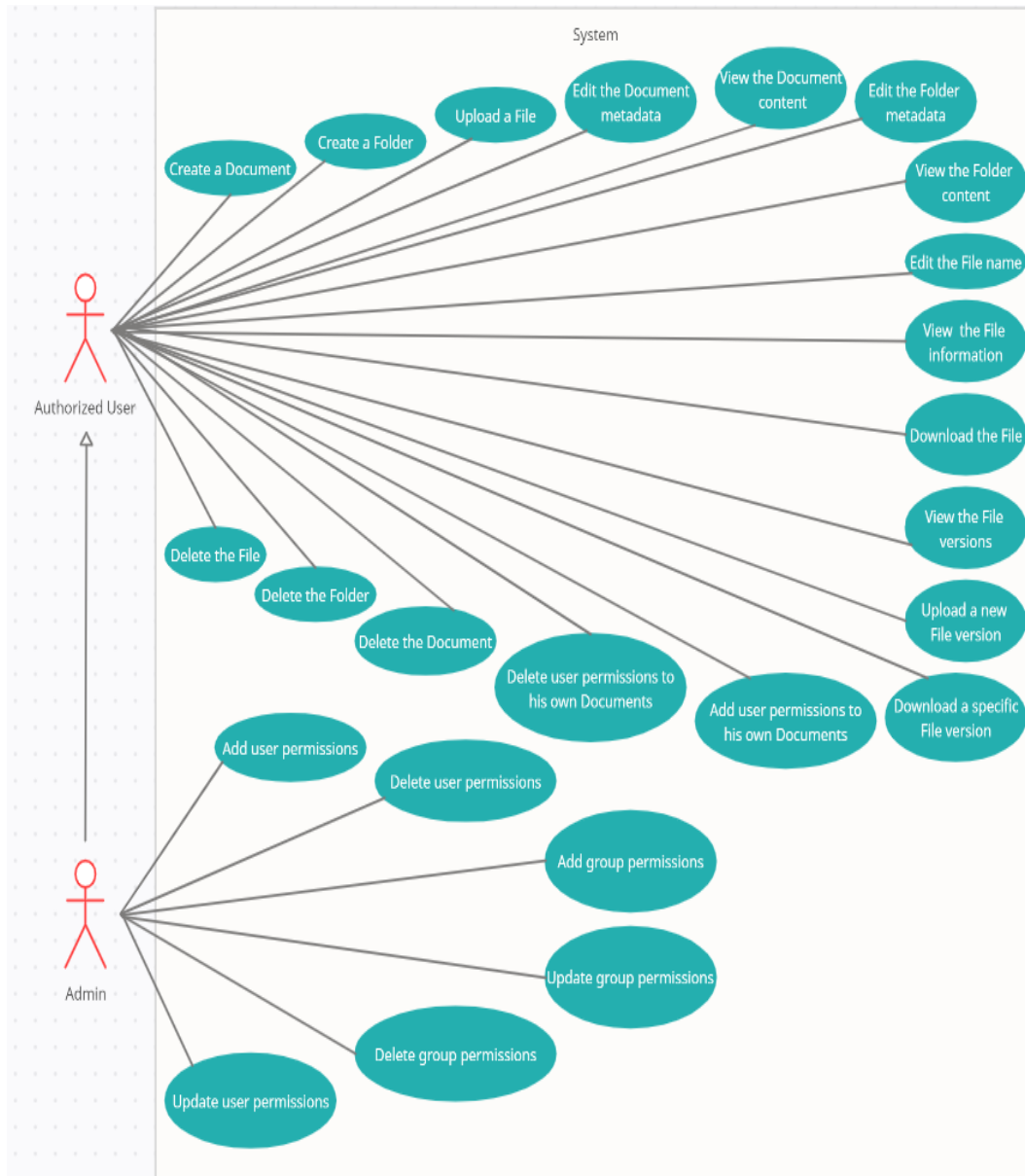


Figure 2.4: A Use Case Diagram

<b>Entity</b>	<b>Operations</b>
Document	create a Document view the Document content edit the Document metadata delete the Document set the user permissions to the Document delete the user permissions to the Document
Folder	create a Folder view the Folder content edit the Folder metadata delete the Folder
File	create a File edit the File name view the File information download the File view the File versions upload a new File version download a specific File version delete the File

**Table 2.1:** The possible operations that can be performed in the Document Manager application

## Chapter 3

### UX research

#### 3.1 Motivation

In my opinion, it is very important to do the UX research before starting front-end development, because it can potentially save me from redoing the parts of the application that were not sufficiently considered.

Also, as the UX is the study of the interaction between users and a system [28], the UX research will give me an idea how to make the application easy and intuitive from a user's point of view.

#### 3.2 Document Manager functionality

The success of any computer application is dependent on providing the appropriate facilities for the task in a manner that enables users to exploit them effectively [11].

The provision of facilities is an issue of functionality. In case of the Document Manager, its functionality fully depends on the implemented API endpoints and accordingly, the UI/UX<sup>1</sup> will be also built based on them.

I need to study how well the application functionality accommodates users' needs, and here I am directly concerned with one of the main components of UX design – usability.

#### 3.3 Usability

Usability is a measure of how well a user in a specific context can use a product to achieve a defined goal effectively, efficiently and satisfactorily [15]. To reach a high level of usability the page is supposed to load fast and be easy for understanding. To make it possible I need to create a clear visual hierarchy and transparent web navigation.

---

<sup>1</sup>UI (User Interface) is anything that facilitates the interaction between a user and a machine [28]. UX is an acronym that stands for User eXperience [28]

## 3.4 The importance of a clear visual hierarchy

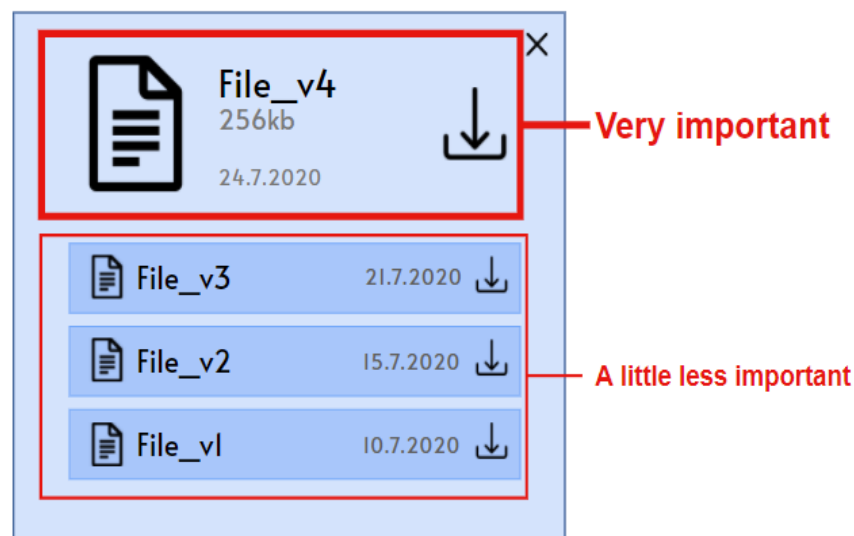
It is necessary to make sure that the appearance of the elements on the page clearly and accurately portray the relationships between them: which elements are more important, which are similar and which are a part of other elements. In other words, each page should have a **clear visual hierarchy** [22].

### 3.4.1 Main traits of a clear visual hierarchy

A good visual hierarchy saves the work by "preprocessing" the page: organizing and prioritizing its contents in a way that can be grasped almost instantly [22]. Pages with a clear visual hierarchy have three traits:

- **The more important something is, the more prominent it is**

For example, the last version of a File in the list of versions is larger and bolder because it is more important than the older ones (Figure 3.1).



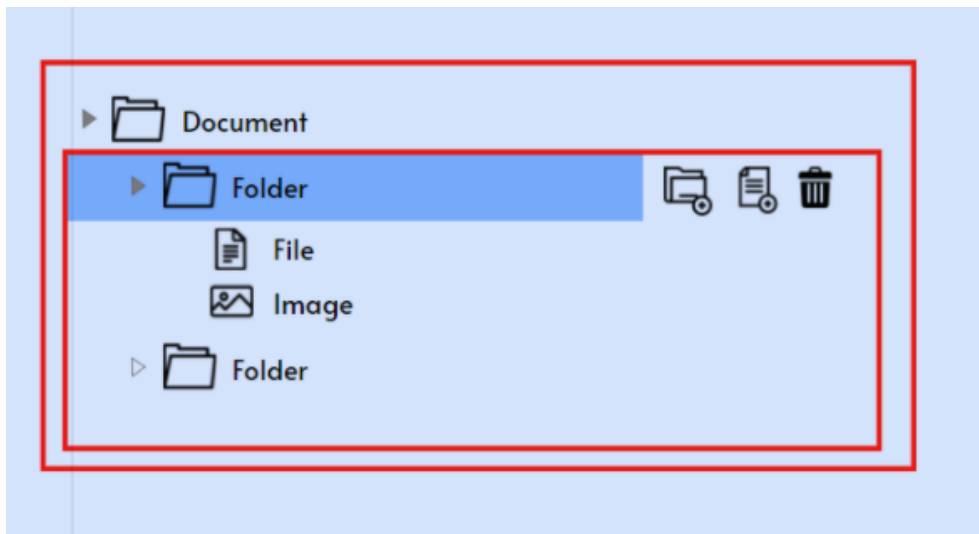
**Figure 3.1:** An example of a clear visual hierarchy in the list of File versions

- **The things that are related logically are also related visually**

Documents, Folders and Files are related logically, that is why they should be displayed in a similar visual style.

- **Things are “nested” visually to show what is a part of what**

For example, a Document should appear above a Folder because the Folder is a part of that Document (Figure 3.2).



**Figure 3.2:** An example of a clear visual hierarchy in the file system

### ■ 3.4.2 Adding a Document

Due to the fact that Documents have the highest position in the DM hierarchy and they act as a starting point of a file system, the location of the Document add-icon should be different from the others. Probably, it should be completely moved out of a file system bounds.

### ■ 3.4.3 Place of a logotype in a visual hierarchy

The application name or logotype represents the whole site, which means that it is the highest element in the logical hierarchy of the site. In the same way as we expect to see the name of a building over the front entrance, we expect to see the logotype at the top of the page.

The best place for the logotype is in the upper left corner where it frames the entire page and where it can be found without thinking (Figure 3.3) [22].

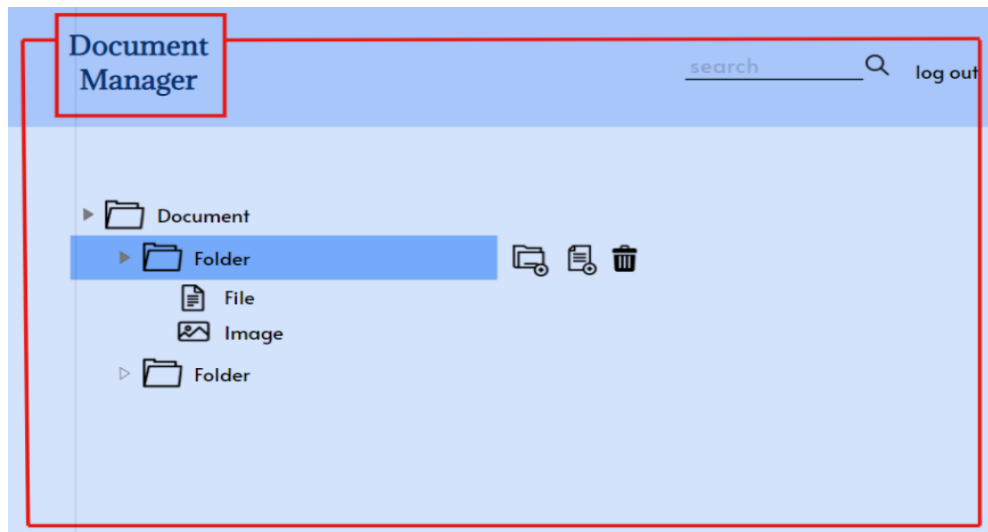
## ■ 3.5 Web navigation

A visitor needs to know that he does not get lost on the website. That he easily figures out where to find what he wants. Otherwise, the visitor will leave the website.

To encourage a visitor to stay and subsequently create a positive user experience it is necessary to make an organized and transparent web navigation that will act as a road map to direct a visitor to various information on your website.

The Navigation should explain how to use the website. If the navigation fulfills its task it implicitly tells to the visitor where to start and what options the website provides.





**Figure 3.3:** An example of placing a logotype on the page

### ■ 3.5.1 Home page button

One of the most crucial items in the persistent navigation is a button or a link that takes a visitor to the website's Home page. There is an emerging convention that the logotype doubles as such a button. Having a Home button in sight at all times offers reassurance that no matter how lost the visitor is, he can always start over. It is a useful idea that every site should implement [22].

In Document Manager, when clicking on the logotype, the file system closes and the visitor sees only the list of his documents, i.e. the initial state of the application (Figure 3.4).

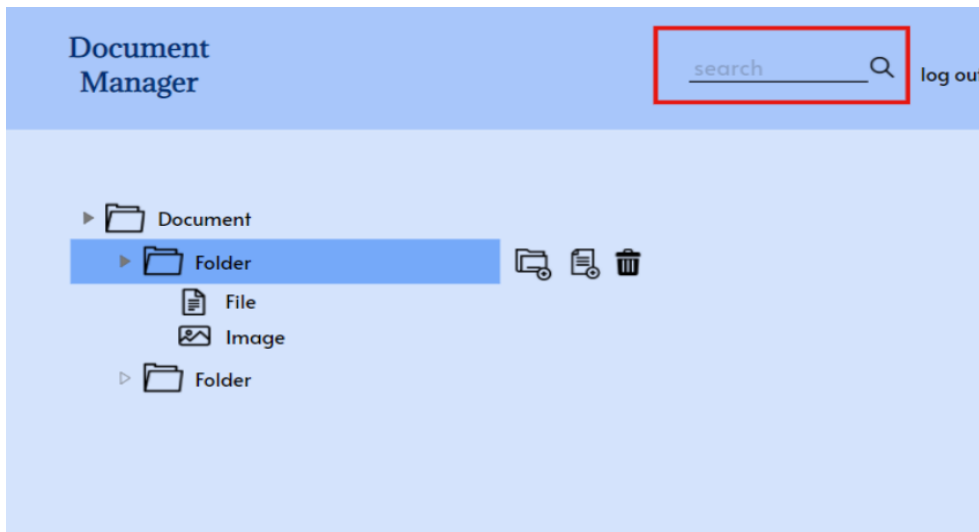


**Figure 3.4:** The initial state of the application

### 3.5.2 Search button

According to Econsultancy<sup>2</sup> report [21], 30 percent of visitors use a search field. Therefore, it is important to remember that when a large number of users visit a page for the first time they look for the search field (Figure 3.5).

From the UX side it is necessary to have a search field, especially in a file system, where a lot of folders and files are stored. But due to the fact that the API does not have the endpoints for searching on the server, the search field is not implemented on the front-end side.



**Figure 3.5:** An example of placing a search field on the page

### 3.5.3 Operation icons

A user can perform lots of operations within the Document Manager. For instance, add, change, download, delete entities.

All these operations should be easily-recognizable and displayed on a page in such a way that the user will have no problems with understanding how to operate with them.

In case of an existing entity, the operation-icons should be situated near it, to make it clear that the operations relate exactly to this entity.

---

<sup>2</sup>a source of intelligence for digital marketers [2]



## Chapter 4

### Front-end plan

Front-end is the presentation layer of an application. The front-end includes building intuitive and pleasant interfaces, as well as efficiently storing, presenting, and updating data received from the back-end or API [34].

There are a lot of tools and techniques used to create the front-end of a website. It is important to choose the most appropriate way of front-end development based on the idea of the application and taking into account the data provided by the DM API.

#### 4.1 Concepts of front-end development

##### 4.1.1 Single-page application vs Multi-page application

A key success factor for a web application is the design and architecture choices a developer makes. There are two main design patterns for web applications: multi-page application (MPA) and single-page application (SPA). Both models have their pros and cons.

##### Single-page application (SPA)

A single-page application is an app that works inside a browser and does not require page reloading during use. The SPA performs most of the user interface logic in a web browser [20].

It communicates with the server using web APIs to retrieve data. Initially, all UI resources (HTML, CSS, JavaScript) are retrieved by a single page load once. Any update to the page happen dynamically in response to the user's actions. The page does not reload (or loads other pages) at any point in the life span of the application[20].

- **Pros** - the SPA is fast, as most resources (HTML, CSS, Scripts) are only **loaded once** throughout the lifespan of an application.

The SPA provides **better caching capabilities**. It sends a request to the server and then stores the received data locally in the client-side. Therefore, the SPA can cache any local data efficiently and users can work even offline [20].

- **Cons** - Since the SPA loads the entire application at once, the initial load can often take **much longer** than an MPA-loaded page at a time. It happens, because heavy client frameworks are required to be loaded to the client.

It **requires JavaScript** to be present and enabled. If a user disables JavaScript in a browser, it will be impossible to present an application and its actions correctly.

## ■ Multi-page application (MPA)

A Multiple-page applications work in a “traditional” way. Every change, e.g. displaying the data or submitting the data back to server requests rendering a new page from the server in the browser [4].

- **Pros** - It is the perfect approach for users who need a **visual map** of where to go in the application. Solid, few level menu navigation is an essential part of traditional Multi-Page Applications [4].

Very good and easy for proper **SEO management**. It gives better chances to rank for different keywords since an application can be optimized for one keyword per page [4].

- **Cons** - In the MPA, a browser reloads most of the UI resources with every user interaction. It **lowers speed and performance** [20].

The development becomes **quite complex**. The developer needs to use frameworks for both the client and the server side. This results in a longer time of application development [4].

## ■ 4.1.2 Server-side rendering vs Client-side rendering

Earlier, websites and web applications had a common strategy to follow. They prepared HTML content to be sent to the browser at the server-side. Then, this content was rendered as HTML with CSS styling in the browser [32].

JavaScript frameworks came in with a completely different approach to web development. They allowed to render dynamic content right from the browser [32].

That is why over recent years a lot of developers are into the dilemma of having the two approaches to rendering websites: client-side and server-side.

## ■ Server-side rendering (SSR)

In server-side rendering when a user makes a request to a webpage, the server prepares the HTML page by fetching user-specific data and sends it to the user’s machine over the internet. The browser then constructs the content and displays the page. This entire process of fetching data from the database, creating the HTML page and sending it to the client happens in milliseconds (Figure 4.1) [3].

SSR can speed up page load time. Search engines can crawl the website for better SEO<sup>1</sup> and rank it better in Google<sup>2</sup> search results [3].

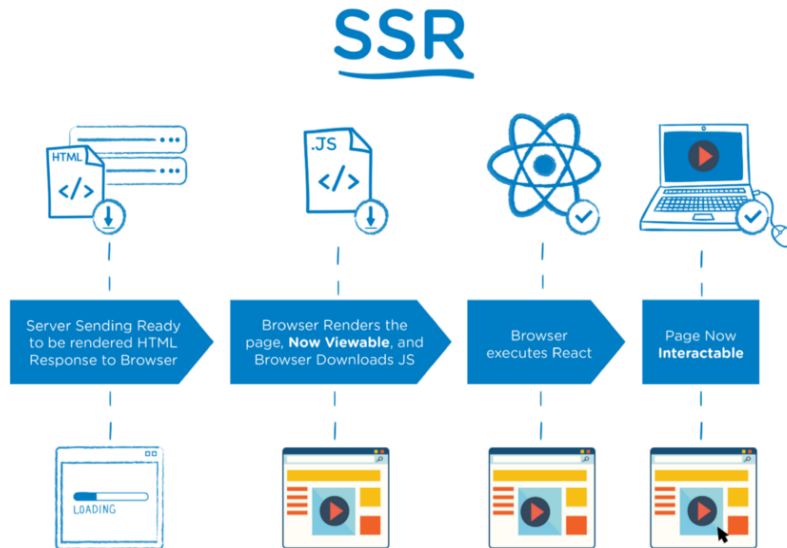


Figure 4.1: A server-side rendering schema (taken from [3])

### ■ Client-side rendering (CSR)

When we talk about client-side rendering, it is about rendering the content in the browser using JavaScript. So instead of getting all the content from the HTML document itself, the HTML document with a JavaScript file in initial loading itself is received, which renders the rest of the site using the browser (Figure 4.2) [3].

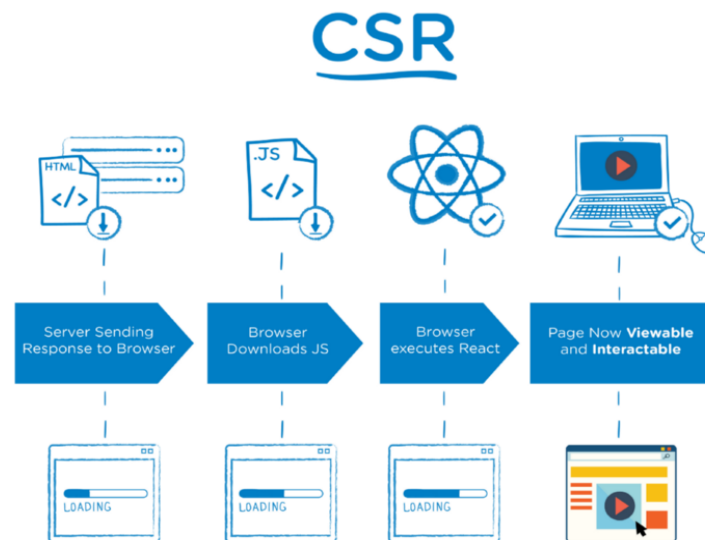
### ■ 4.1.3 Conclusion

Among all the concepts of front-end development I have chosen the Single-Page Application and Client-Side Rendering for the Document Manager application development. When making the final decision, I based it on the following factors:

- **Good SEO is not important** - The Document Manager is an internal application, so it is unnecessary to optimize the website and its content to increase its visibility for relevant searches.
- **JavaScript** - the SPA cannot work correctly without JavaScript instead of the MPA. But it is not an essential problem, because by 21.04.2021

<sup>1</sup>SEO - Search engine optimization: the process of making your site better for search engines [8]

<sup>2</sup>Google is a fully-automated search engine that uses software known as "web crawlers" that explore the web on a regular basis to find sites to add to our index [8]



**Figure 4.2:** A client-side rendering schema (taken from [3])

according to [9] 92.11 percent of web users use JavaScript.

- **CSR ensures fast website rendering after the initial load** - with client-side rendering, the initial page loading is naturally a bit slow. However, after that, every subsequent page loading is very fast [3].
- **No need to reload the entire UI** - In the CSR approach, communication with the server happens only to get run-time data. It will be more convenient for a user to see that only parts of the page are updated and the entire UI is not reloaded after every call to the server [3].
- **Complexity** - CSR is arguably easier to do.

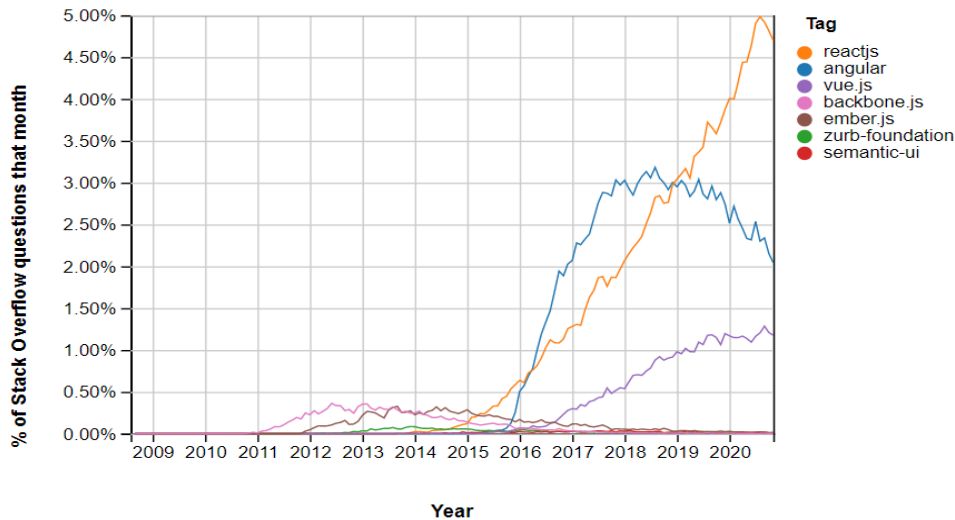
## 4.2 Technical approaches to the SPA development

Modern web applications, due to the functionalities they provide in their user interfaces, have a complex program structure. Because of the complexity of the entire application, writing code manually can result in uneven quality and content of individual application parts [19]. The maintenance of such applications is also more difficult.

Because of this, web applications are often developed using different frameworks. A framework allows structuring, simpler and more uniform program script writing, and thus easier web application maintenance [19].

According to Stack Overflow<sup>3</sup> trends [1] the three most popular JavaScript frameworks in 2020 are Vue.js, Angular and React (Figure 4.3).

<sup>3</sup>the largest, most trusted online community for developers to learn, share their programming knowledge, and build their careers [34]



**Figure 4.3:** The popularity scale of the six most well-known Javascript frameworks

### 4.2.1 Vue.js

Vue.js is a progressive framework used to build web interfaces and one-page applications. It is small in size and offers two major advantages – it is component-based and renders the UI using a virtual DOM [10].

This framework is increasingly popular in China, and a significant part of its content and discussions are in Chinese. Moreover, most Vue.js plugins are also written in Chinese [13]. This language difference can cause problems during implementation.

Vue.js has a relatively low market share, which means that information exchange in this framework is only in the early stages [37].

### 4.2.2 Angular

Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps [35].

Angular offers a wide range of features and benefits for developers. For example, Angular is cross-platform, so it can be used for desktop-installed, native or progressive web applications (PWA<sup>4</sup>).

It uses different design patterns, such as MVC<sup>5</sup>, Singleton, DI<sup>6</sup> and others,

<sup>4</sup>PWA - Progressive Web Apps: web apps that bring a native app-like user experience to cross-platform web applications [28].

<sup>5</sup>MVC - Model-View-Controller: an application design pattern comprised of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input) [7].

<sup>6</sup>Dependency Injection: a design pattern in which a class requests dependencies from external sources rather than creating them [35].



which provide easily recognizable solutions to common problems.

Angular also has tools for improving speed and performance. Notably, these include a new Component Router, which delivers automatic code-splitting, and Angular Universal technology for near-instant rendering in just HTML and CSS (see 4.1.2 for more information about this type of rendering) [35].

In other words, Angular is a full-fledged framework for application development. There is no need to import additional libraries, because all the above-mentioned functions can be implemented by means of the Angular package.

Taking into account the fact that the DM application is not very complex, does not contain a lot of business logic, does not need a mobile application or specific optimization, I can make a conclusion that Angular is a sophisticated solution for the DM application front-end development.

### ■ 4.2.3 React

React is a JavaScript library for building user interfaces [33].

React makes it painless to create interactive UIs. It allows to design simple views for each state in an application, and then efficiently updates and renders just the right components when the data changes [33].

React requires additional libraries for optimization, routing, styling, animations, etc. Therefore, a developer has to install the libraries and modules he needs for the application.

### ■ JSX syntax

JSX is a JavaScript Extension Syntax used in React to easily write HTML and JavaScript together.

For example, this tag syntax is neither a string nor HTML:

```
1 const element = <h1>Hello, world!</h1>;
```

**Listing 4.1:** An example of JSX syntax

React embraces the fact that rendering logic is tightly coupled with other UI logic. That is why instead of separating technologies by putting markup and logic in different files, React separates concerns with the units called “components” that contain both [33].

Capitalized types indicate that the JSX tag is referring to a React component (For instance, the component `CustomButton` in Listing 4.2). These tags get compiled into a direct reference to the named variable [33].

```
1 return <CustomButton color="red" />;
```

**Listing 4.2:** The React component in JSX

### ■ 4.2.4 Conclusion

Based on the comparison of the JavaScript frameworks, I can conclude that the most suitable framework for the Document Manager front-end development

is React. The key factors of such a choice are:

- **flexible development** - I use only these necessary modules and libraries, I need directly for the DM;
- **reusable components** - this feature is very useful, for example, for the implementation of modal windows, which have the same structure, but are used in different parts of the application;
- **thriving community** - React has a community of millions of developers. There are many online forums, where I can find the answers to my questions and discuss the best practices;
- **a lot of useful libraries** - for instance, I use React Query library to manage the data within the application (see the section 6.4 for more details);
- **my own experience in React** - and last but not least, I have already written some small applications in React.

## 4.3 Application structure

I use a **feature-based approach** to React development. Feature folders help eliminate some common development problems (for example, when a project widens) by placing everything that is unique to one area of the code together [24].

Benefits of this approach:

- **Code is structured in a way that reflects purpose**  
This makes it easier to find files and to understand where the things belong to. Additionally, it helps to clarify how a bit of code contributes to the overall goals of an application.
- **Code is easier to refactor**  
If a feature has one entry point, it is clear that removing the feature will have limited impact on the rest of the system.

You can see the structure of the Document Manager front-end in Figure 4.4.

### 4.3.1 Assets

Assets folder should contain the resources, which will be required by the application. For example, files like images, icons, fonts, etc.

In my case, the assets folder contains only icons (Folder, File, add Folder, delete Folder, etc. icons) in SVG<sup>7</sup> format.

---

<sup>7</sup>Scalable Vector Graphics

Name
..
assets
components
features/document
helpers
DocumentList.js
Routes.js
index.js
styled.js

**Figure 4.4:** The structure of the Document Manager front-end (taken from Gitlab FEL)

### 4.3.2 Helpers

Helpers, as the name suggests, help with different tasks. Each helper file is simply a collection of functions in a particular category [14].

Helpers are not written in an Object-Oriented format. They are simple, procedural functions. Each helper function performs one specific task, with no dependence on other functions [14].

I have one helper function for making API requests. It takes several arguments: **url** - an API endpoint, **method** - GET, POST, PUT or DELETE, **body** and **headers** (Listing 4.3).

```

1 export const request = async (
2   url,
3   method = "GET",
4   body = null,
5   headers = {}) => {
6   const response = await fetch(url, {method, body, headers});
7   const data = await response.json();
8   return data;
9 }

```

**Listing 4.3:** The helper function for making an API request

### 4.3.3 Features

My feature folder contains the following files:

- **api.js** - the functions, which perform CRUD operations are situated in `api.js`. For example, `getDocuments()`, `addFolder()`, `updateFile()` and `deleteUserPermission()`;
- **hooks.js** - this file contains my custom hooks (you can find more information about custom hooks in the section 6.3.1) with application logic processing functions;
- **utils.js** - here I place small snippets, which are used throughout the application. For instance, the function that parses URLs (Listing 4.4).

```

1 export const getLinkInfo = (link, part) => {
2   if (!link) return null;
3   const url = new URL(link);
4   return `${url.pathname.split("/") [part]}`;
5 };

```

**Listing 4.4:** The URL parser function

#### 4.3.4 Components

React allows to create encapsulated components and make complex UIs by composing them [33]. In the DM application I have several main components:

- **Document List component** - it is a starting point of the application. It shows the list of Documents;
- **Folder List component** - it receives props from the Document List and recursively shows the Folders and Files, which the Documents contain;
- **Modal component** - it is the component, which helps to create dialogues. It is used when a user wants to add a Document, Folder or File;
- **File Info component** - this component is rendered when a user wants to see the information about the File. It contains the list of File versions and the basic File information.

Components are reusable, so it is enough to create a component once and then just call it when it is needed. For example, the Modal component is called when a user wants to add a Document, Folder or File. The structure of these modals is different because of the different props that were passed, but in fact it is the same Modal component.



# Chapter 5

## UI design

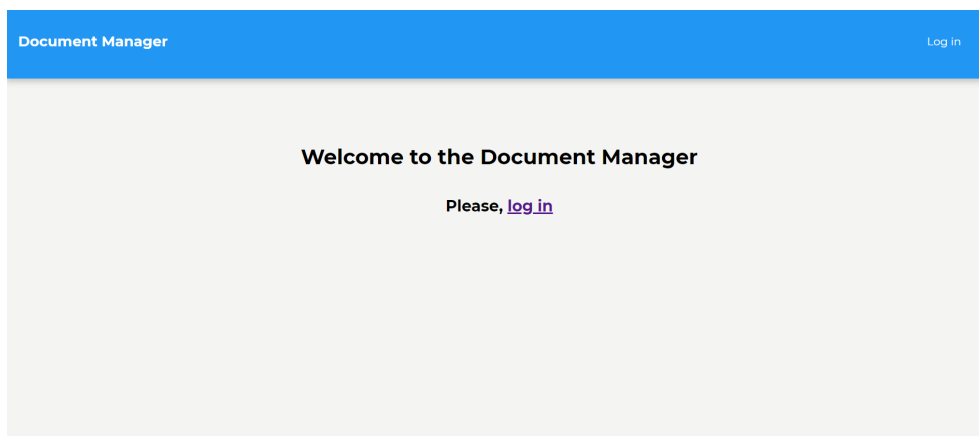
In this chapter the application flow is described in detail. To illustrate the application functionalities the chapter contains the screenshots of the Document Manager front-end, which was implemented based on the use cases from the section 2.4.

The recommendations from the UX research (chapter 3) were also taken into account.

### 5.1 Authorization

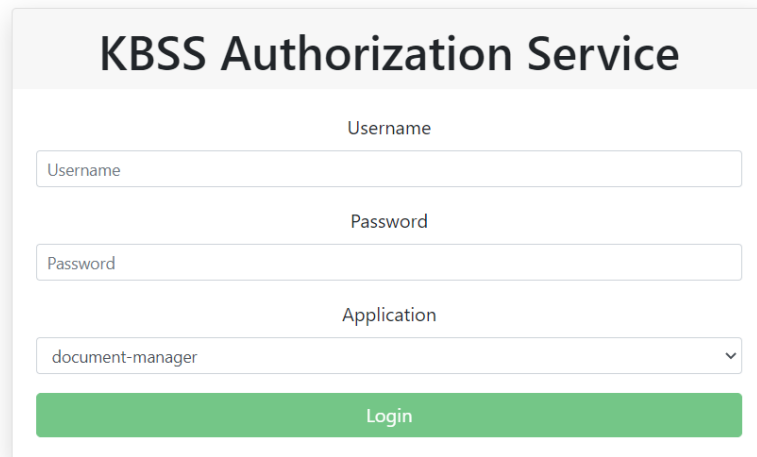
The Document Manager does not have its own internal authorization system and uses an external one provided by the Knowledge-based and Software Systems CTU FEE group.

A user cannot use the application without authorization. When the user is unauthorized he is offered to log in by clicking on a link on a screen (Figure 5.1)



**Figure 5.1:** The page shown to an unauthorized user

Then, the user is redirected to the page with a log in form, where he needs to fill in two inputs: "username" and "password" (Figure 5.2). After successful authorization, the user is returned to the main page of the Document Manager application, where the list of available for him Documents is situated.



The image shows a login form titled "KBSS Authorization Service". It contains three input fields: "Username", "Password", and "Application". The "Application" field is a dropdown menu with "document-manager" selected. Below the fields is a green "Login" button.

**Figure 5.2:** KBSS Authorization Service, which is used in the Document Manager application

Authorization is needed to define the user's access level to show him only these Documents he has the rights to interact with.

An authorized user can log out any time by clicking on the corresponding button.

## ■ 5.2 File system display

Documents are the highest entities in the DM hierarchy, therefore they should be the first elements a user sees on the page. Documents contain Folders and Files, which should be accessible by clicking on the Document, where they are stored. The Folders inside Documents in turn can contain other Folders and Files and so on (Figure 5.3)

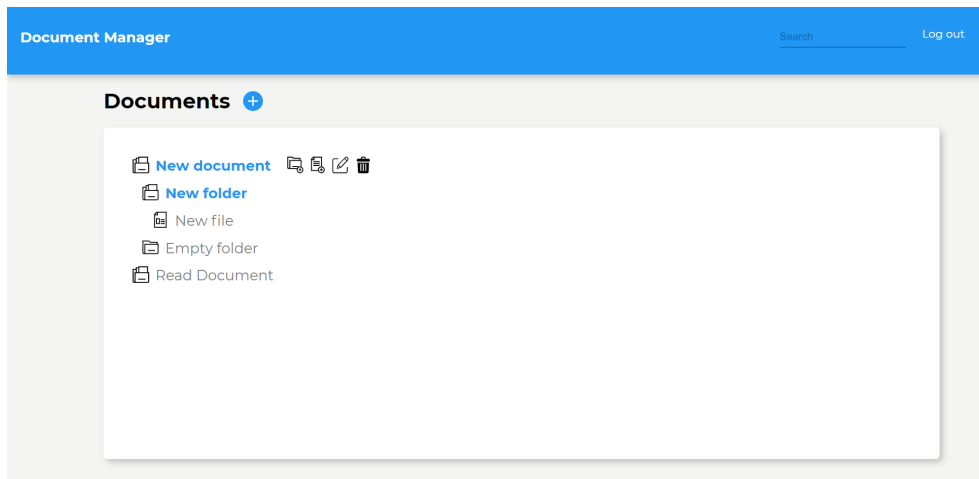
This is called a recursive structure. The specific meaning of "recursive" in this context is "operating on a directory and its contents, including the contents of any subdirectories".

## ■ 5.3 Operations implementation

As I inspected in the section 2.3, there is a set of operations that can be done with all the DM entities. These operations can be divided into two groups: Document or Folder operations and File operations.

### ■ 5.3.1 Document and Folder operations

Documents and Folders act as containers for Folders and Files, that is why first of all they need to have "Add a new Folder" and "Add a new File"



**Figure 5.3:** The file system in the Document Manager application

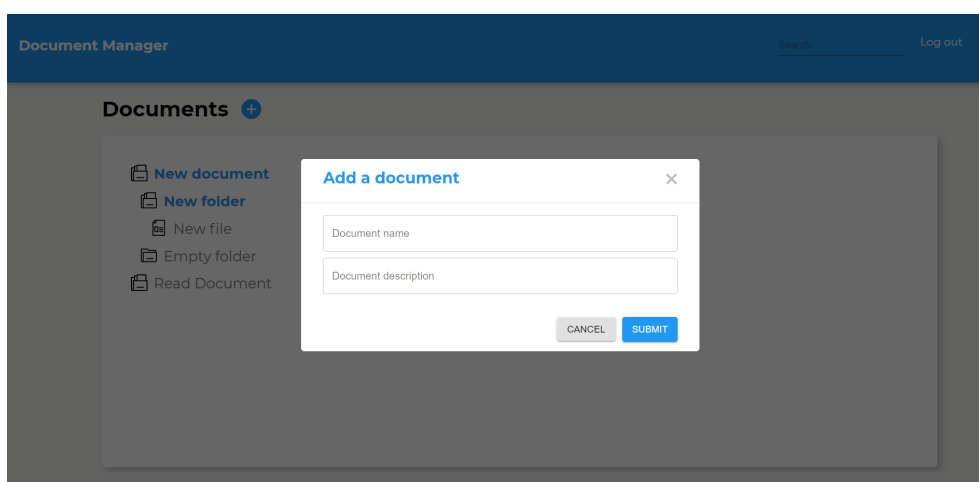
operations (Figure 5.4). Also, they can be edited and deleted.



**Figure 5.4:** The operations that can be performed on a Document or Folder

### ■ Add a Document or Folder

To add a new Document or Folder it is enough to fill in the field "name" in a modal window, which will open after clicking on the add-icon (Figure 5.5). The field "description" is optional. If a Document or Folder with such a name already exists, a user will see an error message and will be offered to change the name and try again.

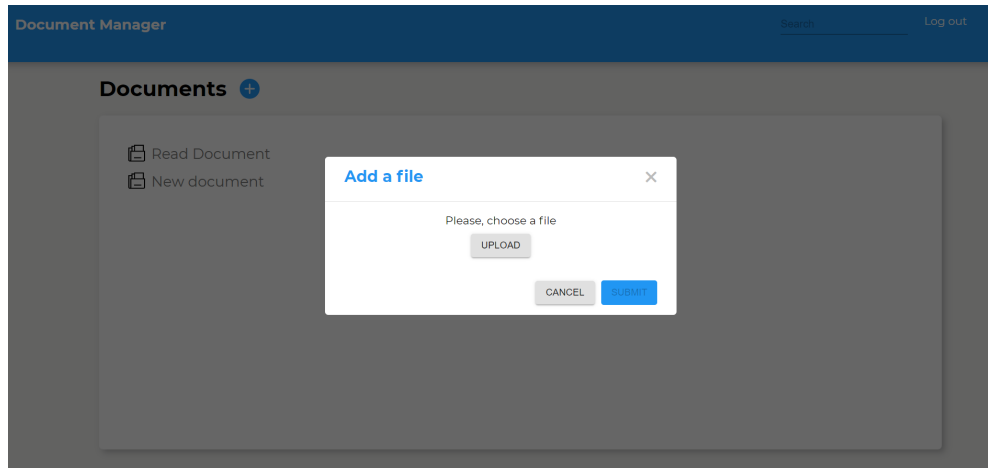


**Figure 5.5:** The modal window for adding a new Document or Folder

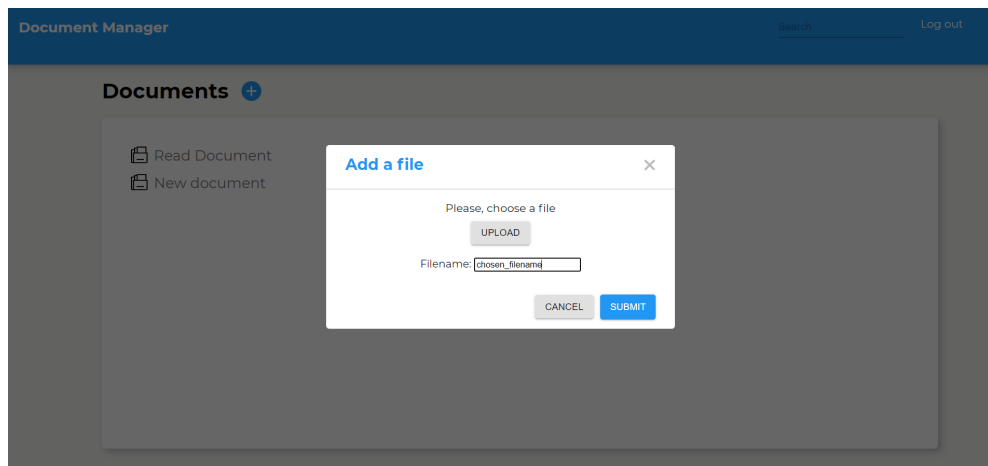


## ■ Add a File

Files can be added to any Folder. A modal window, where a user is able to choose a File to upload, also contains the "filename" input. It is hidden before the File is chosen (Figure 5.6) and after it a user can change the filename (Figure 5.7). By default, the File name corresponds to the File name on user's local device, but it can be changed before the uploading process starts. An application checks if a File name is unique and if yes, the File will be added.



**Figure 5.6:** The modal window for adding a new File (before the File is chosen)



**Figure 5.7:** The modal window for adding a new File (after the File is chosen)

## ■ Edit

Documents and Folders have only two parameters, which can be edited: "name" and "description". When a user wants to change the name or description, a modal window with prefilled values opens. In case the new name coincides

with the name of the already existing Document or Folder, an error message is shown and the changes are not saved.

### ■ Delete

If a user wants to delete a Document or Folder, an application will display a delete confirmation dialog box to make sure that the delete-icon was not clicked by mistake. Then, a Document or Folder will be deleted without a possibility of being restored.

### ■ 5.3.2 File operations

In comparison with Documents and Folders, Files have some unique operations, such as "Download", "View File information" and "View File versions". Accordingly, due to its complex properties, a single File is shown separately on the page to provide a user with the full File information (including all File versions) in a user-friendly way.



**Figure 5.8:** The operations that can be performed on a File

### ■ View the File information

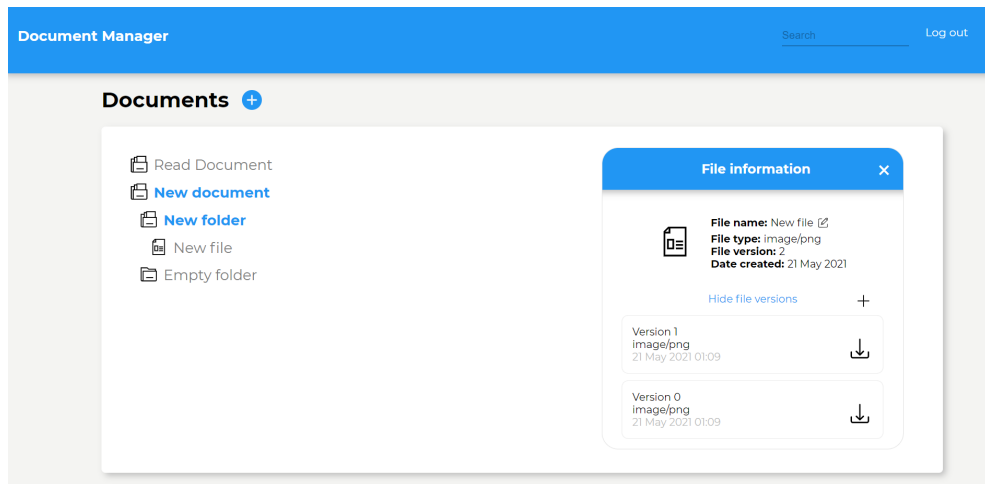
If a user clicks on the information-icon, all the information will be displayed on the right side of a page (Figure 5.9). The structure of a File info section will be the following: the last version of the File with the information about file size, the date when it was created, etc., the list of previous File versions in reversed order - from the newest version to the oldest. Also there will be a possibility to change the file name, to add a new file version and to download any of the versions.

### ■ Edit

A user can change both the File name and File content. To change the File content means to upload a new File version. To make it understandable for a user, an option to add a new version should be situated near the list of File versions. To change the File name it is enough to enter the name that is unique to prevent an error.

### ■ Download

A user can download a File by clicking on the download-icon, which is situated on the right-hand side of it (Figure 5.8). Moreover, it is possible to download all File versions.



**Figure 5.9:** The window with the information about the File and its versions

## ■ Delete

Files can be deleted in the same way like Documents and Folders.

## ■ 5.4 Access rights management

As I mentioned in 2.1.5 it is possible to set the access rights only in Documents. This automatically applies to the content of a Document. The Document is not visible to the user with the insufficient access level, consequently the content of the Document is also unavailable.

The access rights are managed in the Document modal window, which opens when a user wants to create a new Document (Figure 5.10). This user can set the access level for other users by clicking "Add user permissions" in the Document modal window. To set the access rights it is necessary to choose the user's URI in the list of users and one of the four (None, Read, Write, Security) access levels.

Access rights can also be managed by clicking on the edit-icon near the already existing Documents.

## ■ 5.5 User management

The API contains some endpoints related to user management (for instance, Listing 2.1), which are not used in the DM application. User management implementation is not a part of my bachelor thesis, but it can be considered as a future extension. Now user management is regulated by the authorization service provided by CTU FEE.

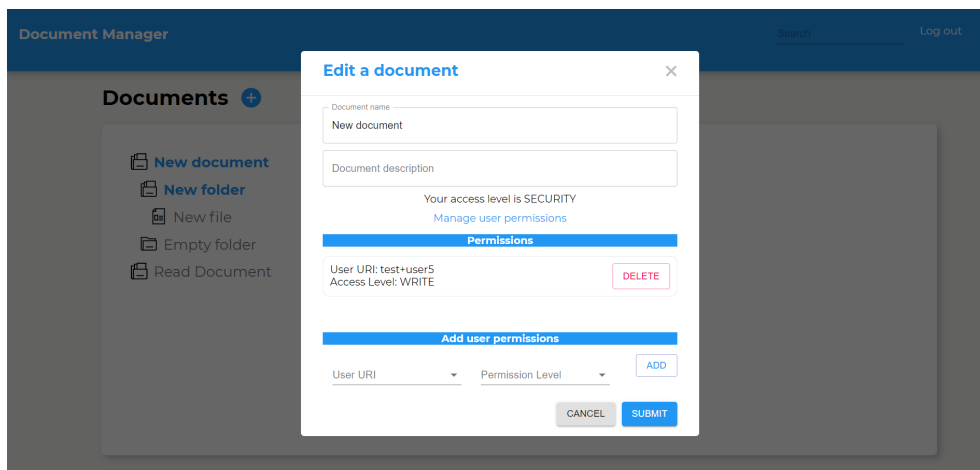


Figure 5.10: The modal window, which contains the user management section



## Chapter 6

### Technical solution

#### 6.1 Authorization

##### 6.1.1 External authorization service

I am using an external service provided by Knowledge-based and Software Systems CTU FEE group for user authorization. This service uses JSON Web Tokens, which are able to transfer the information in JSON format. These tokens can be digitally signed or even encrypted. Due to these properties they are mostly used for authorization.

##### Working principle

The Document Manager application only needs to know the address of the authorization service to redirect a user who wants to log in there. Then, the user fills in his username and password, chooses 'document-manager' in the list of KBSS applications and presses login. The rest of the work, namely validation, checking user credentials and setting the user access level, is done by the authorization service.

After a successful authorization the service returns a unique token, which I process and set it in Local Storage. The token contains the information about the user's access rights and is used in server requests.

##### 6.1.2 Token validity

A token is valid for some period of time. The case of the token expiration should be handled to prevent a user to see the error unknown to him.

When the token is not valid anymore, the request to the server fails and returns the error with the status code 502 - **Bad Gateway**. My solution to this issue is to check the status code of the request for getting the Documents and once I get 502 - **Bad Gateway**, the user is automatically logged out. Then, the user is redirected to a page with information that he needs to log in again to continue.

## 6.2 Routing

The routing is organized by the React Router library. React Router is a collection of navigational components that compose declaratively with the application [36].

The Document Manager application has two routes: the main route, where the file system can be found and the authorization route, where a user can log in. You can see the routes in the listing 6.1

```

1 export const Routes = () => {
2   return (
3     <Switch>
4       <Route path="/" exact component={DocumentTree} />
5       <Route path="/auth" exact component={TokenHandler} />;
6     </Switch>
7   );
8 };

```

**Listing 6.1:** The Document Manager routes

## 6.3 Hooks

Hooks are a new addition to the React 16.8. They allow for using state and other React features without writing a class [33].

With Hooks, it is possible to extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components [33].

### 6.3.1 Custom hooks

A custom Hook is a JavaScript function, which name starts with "use" and that may call other Hooks [33].

Unlike a React component, the custom Hook does not need to have a specific signature. We can decide what it takes as arguments, and what it should return. In other words, it is just like a normal function [33].

I have built custom hooks to extract the DM logic into reusable functions. For example, the listing 6.2 is a hook, which contains the logic of adding a new Document. A function `handleAddFolderFetcher` can be simply reused. On lines 22 and 23 I call `useQueryClient` and `useMutation` hooks to manage adding a new Document to the list of Documents displayed on the page. These React Query hooks are described in section 6.4 in more detail.

```

1 export const useAddRootFolder = ({ onClose, modals }) => {
2   const handleAddFolderFetcher = async (data) => {
3     const { isRoot, parentFolderId, isOpen, isEdit } = modals.
4       folder;
5     if (!isOpen || isEdit) return;
6     const parentFolderName = getLinkInfo(parentFolderId, 2);

```

```

7
8   try {
9     const addedFolderResponse = await addFolder(
10      parentFolderId ? "Folder" : "Document",
11      data.name,
12      data.description,
13      isRoot,
14      parentFolderName
15    );
16    }catch (error) {
17      throw(error);
18    }
19
20    const onSuccess = (response) => { ... }
21
22    const queryClient = useQueryClient();
23    const { mutateAsync, isLoading, error } = useMutation(
24      handleAddFolderFetcher, {
25      onSuccess,
26    });
27    return { addRootFolder: mutateAsync, isLoading, error };
28  };

```

**Listing 6.2:** The hook, which contains the logic of adding a Document

### 6.3.2 useState

useState is called inside a function component to add some local state to it. React will preserve this state between re-renders [33]. In the listing 6.3 you can see that useState returns a stateful value and a function to update it.

```
1 const [state, setState] = useState(initialState);
```

**Listing 6.3:** useState hook

I use this hook, for example, to manage the state of a Folder: if it has children it can be opened.

```
1 const [isOpen, setOpen] = useState(false);
2 if (folderChilds.length > 0) setOpen(!isOpen);
```

**Listing 6.4:** An example of the useState hook in the Document Manager

### 6.3.3 useEffect

useEffect adds the ability to perform side effects from a function component. It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but are unified into a single API [33].

For instance, I use this hook to get the current user immediately after render.

```
1 useEffect(() => {
```



```

2   getCurrentUser().then(user => localStorage.setItem('
3     currentUserUri', user['uri']));
  }, [])

```

**Listing 6.5:** An example of the `useEffect` hook in the Document Manager

## 6.4 Client/Server state synchronization

The client and server states should be synchronized to display the DM file system correctly. The client state should include the last changes made on the server and should be immediately updated after a successful server request, that changes the structure of the file system. It is very important from the UX side to show the user the result of his interaction with the application.

Let me give an example. A user wants to add a Folder or a File to the file system. It is assumed he will see a new entity on the page right after he successfully finishes the process of adding it. So, he will not need to refresh the page to see the changes.

For this purpose, I have chosen the React Query library, which ideally fits the requirements of the application. Namely, to update the file system in case of adding, editing or deleting an entity.

### 6.4.1 React Query concepts

React Query makes fetching, caching, synchronizing and updating the data in React applications easier. It can be customized as the application grows and need zero configuration [25].

The three core concepts of React Query are queries, mutations and query invalidation.

#### Queries

A query is a declarative dependency on an asynchronous source of data that is tied to a unique key. A query can be used with any Promise-based method (including GET and POST methods) to **fetch** data from a server [25].

```

1 export const useDocuments = () => {
2   const { data } = useQuery("Documents", getDocumentList);
3   return { documents: data };
4 };

```

**Listing 6.6:** An example of getting the Documents from the server

#### Mutations

Unlike queries, mutations are typically used to create, update or delete data. For this purpose, React Query exports a `useMutation` hook [25].

I use it for adding, updating or deleting the DM entities. For example, if a user successfully adds a new Folder he will immediately see it in the Folder list.

```

1  const { mutateAsync } = useMutation(handleAddFolderFetcher, {
    onSuccess});
2  return {handleAddFolderFetcher : mutateAsync}

```

**Listing 6.7:** An example of adding a Folder to the Folder list

OnSuccess function will be called when the mutation is successful and the mutation's result will be passed [25].

## ■ Query Invalidation

The QueryClient<sup>1</sup> has an invalidateQueries method that lets mark queries as stale when the query's data are out of date [25].

```

1  const queryClient = useQueryClient()
2  queryClient.invalidateQueries('Documents')

```

**Listing 6.8:** An example of invalidation a 'Documents' query

## ■ 6.5 React.memo

React.memo is a higher order component<sup>2</sup>.

If a component renders the same result given the same props, it can be wrapped in a call to React.memo for a performance boost by memorizing the result. This means that React will skip rendering the component and reuse the last rendered result [33].

This method exists for performance optimization purposes [33]. For example, when I add a new Folder or File to the list, the other items do not rerender because their props have not been changed.

```

1  const FolderItem = memo(({
2    folder,
3    isRoot,
4    setModals,
5    setOpenFileInfoModal,
6    setFileInfo,
7    parentId
8  }) => { ... })

```

**Listing 6.9:** An example of a memo usage in the DM application

## ■ 6.6 Styling

There are a lot of ways to style components in React. In the Document Manager application, I use Styled-Components library for this purpose.

<sup>1</sup>The QueryClient can be used to interact with a cache[25]

<sup>2</sup>A higher-order component (HOC) is an advanced technique in React for reusing component logic[33]

Styled-Components is a CSS-in-JS library, that bridges the gap between components and styling, offering numerous features to style React components in a functional and reusable way.

It removes the mapping between components and styles. This means that when a developer defines the styles, he actually creates a normal React component, that has the styles attached to it (Figure 6.10) [16].

```

1 export const StyledIcon = styled.img `
2   height: 30px;
3   width: 30px;
4   position: absolute;
5   cursor: pointer;
6   right: ${props => props.right};
7   top: ${props => props.top};
8   position: ${props => props.position};
9   transform: ${props => props.transform};
10 `;

```

**Listing 6.10:** The CSS styles of the StyledIcon component

```

1 import StyledIcon from "./styled";
2 <StyledIcon
3   src={addVersionIcon}
4   title="Add a new file version"
5   position="absolute"
6   right="0"
7   top="-3px"
8   transform="scale(0.6)"
9   onClick={() => { ... }}
10 \>

```

**Listing 6.11:** The usage of the StyledIcon component

The reasons why I use Styled-Components are:

- **Automatic critical CSS** - styled-components keep track of which components are rendered on the page and injects their styles fully automatically [16];
- **Simple dynamic styling** - adapting the styling of a component is simple and intuitive, because a developer do not have to manually manage lots of classes [16];
- **Developer experience** - all of the core library and most popular pieces of functionality are available right in the main import. This makes the library easier to refactor, teach and understand [17].

## Chapter 7

### Usability testing

To design the user interface, which will meet the users' needs, a developer should understand what tasks the users will use their system for and how those tasks will be performed. An understanding of tasks that the users will perform gives developers an insight into the functionality which should be provided and how it will be used [29].

The aim of usability testing is not to solve problems, or to enable a quantitative assessment of usability. It provides the means of identifying problem areas, and the extracting of information concerning problems, difficulties, weaknesses and areas for improvement [29].

#### 7.1 Usability Testing Method

For the Document Manager application, I have chosen a formal testing method: data are collected as typical users are observed interacting with the system, using predefined tasks. I prepared two test scenarios for three test users to complete.

#### 7.2 Test scenarios

Scenarios describe the stories and context behind why a specific user comes to your site. They note the goals and questions to be achieved and sometimes define the possibilities of how a user can achieve them on the site [31].

##### 7.2.1 Basic functionality test

The goal of this test is to check if a user can easily understand how to add, change and delete the basic Document Manager entities.

1. Add a new Document with the name "New document" and the description "This is my first document";
2. Add a new Folder with the name "New folder" and the description "This is my first folder" to this Document;

3. Change the name of this Folder to "Changed folder" ;
4. Delete this Folder;
5. Upload a new File with the name "New file" to the Document;
6. Change the name of this File to "Changed file";
7. View the File versions;
8. Add a new File version;
9. Download this File;
10. Delete the File;
11. Delete the Document;

### 7.2.2 Access rights test

The goal of this test is to check if a user can easily understand how to manage the access rights of his own Documents and be able to distinguish the documents with different access levels.

For this purpose, I have created the two Folders as the administrator and gave the test user the READ and WRITE permission levels to these Folders respectively.

1. Add a new Document with the name "New document" and the description "This is my first document";
2. View user permissions for this Document;
3. Add a new user permission with the userURI "test+user1" and the permission level "READ";
4. Add a new user permission with the userURI "test+user2" and the permission level "SECURITY";
5. Delete a permission for the user with the URI "test+user2";
6. Add a new Folder to the document "Read document";
7. Add a new Folder to the document "Write document";

## 7.3 Results

Practically all the instructions were done by the test users **without serious problems**. I received **positive application reviews** and the UI/UX of the Document Manager was highly valued.

But of course, there were some weak places in the application, where the users did not understand if they act right. Some of these problems were not

critical and did not keep the test users from completing the scenario, but one of the problem was really severe.

The detailed information about discovered application problems you can find below.

### ■ 7.3.1 Folder user rights management UI/UX problem

- **Problem description** - when a test user added a wrong user permission, clicked cancel in a modal window and opened user management section again, he found out that the wrong user is still there. It was not clear that there is no option to reset changes a user made in this section.
- **Where** - the front-end side;
- **Severity level** - 1/5<sup>1</sup>;
- **Solution** - A possible solution is to change the UI/UX by separating the user management section from the editing section, for example, by displaying a new modal window with the list of permissions above the opened one. Another solution is to add "submit" and "cancel" buttons to the user management section, but in my opinion it is not an elegant way to solve this issue, because it will cause the duplication of buttons.

### ■ 7.3.2 WRITE access level does not work

- **Problem description** - As an administrator I have created a folder with WRITE access level for test users to check if test users easily recognize the difference between access levels.

Test users found out that adding a folder or a file to a folder with access level WRITE was impossible and returned an error 403 - forbidden.

It means that something went wrong on a back-end side and back-end was not able to process this action.

- **Where** - the back-end side;
- **Severity level** - 4/5;
- **Solution** - this issue should be solved on the back-end side.

### ■ 7.3.3 Filename does not update in a file system

- **Problem description** - when a user changes a filename in the file information window, the filename does not change in the file system immediately. The filename is updated only after the page reloads.
- **Where** - a front-end side;
- **Severity level** - 3/5;
- **Solution** - update the filename in the file system using React Query.

---

<sup>1</sup>1 - minor, 5 - critical

#### ■ 7.3.4 File version does not update after a new version is added

- **Problem description** - when a user uploads a new file version, the number is not changed immediately in the file information window.
- **Where** - the front-end side;
- **Severity level** - 2/5;
- **Solution** - update the file version number in a file information window using React state.

#### ■ 7.3.5 A new folder display

- **Problem description** - when a user adds a new folder it is not displayed in the file system, it remains hidden inside its parent Folder. A user needs to click on the parent Folder to see the Folder he has just added.
- **Where** - the front-end side;
- **Severity level** - 1/5;
- **Solution** - open the parent Folder immediately after a user adds a new Folder.

#### ■ 7.3.6 Problem with changing a filename

- **Problem description** - when a user changes a filename he supposes that the changes can be saved on enter, not on blur.
- **Where** - the front-end side;
- **Severity level** - 2/5;
- **Solution** - change the way of saving a new filename.

#### ■ 7.3.7 File information window does not close after the File is deleted

- **Problem description** - when a user has the window with the File information opened and then deletes this File, the information window remains open.
- **Where** - the front-end side;
- **Severity level** - 3/5;
- **Solution** - handle closing the information window when a user deletes the File.

## 7.4 Changes

I solved some of the above-mentioned problems to improve the usability of the Document Manager application. I described it in detail in the table 7.1

Problem	Solution
Filename does not update in a file system (subsection 7.3.3)	I changed the filename in cache using the <code>useQueryClient</code> hook from the React Query library.
File version does not update after a new version is added (subsection 7.3.4)	I solved this problem by changing the states in the <code>FileInfo</code> component.
Problem with changing a file-name (subsection 7.3.6)	I added the event listener for a global <code>window</code> , checked if the event key equals <code>Enter</code> and called the blur event for the focused element.
File information window does not close after the File is deleted (subsection 7.3.7)	I change the state of the File information window before the request to delete the File is sent.

**Table 7.1:** The problems identified while the usability testing and their solutions







## Chapter 8

### Conclusion

In my opinion, the main goal of my bachelor thesis – to implement the user interface for the Document Manager application – was reached. It is also important to mention that the application can be considered not only as a base for the bachelor thesis, but as a functional tool for practical use as well.

I had some experience with React application development beforehand, but during the Document Manager front-end development I came across with much more difficult tasks and certainly improved my knowledge. Moreover, I gained experience as a UI/UX designer and did research how to make the application intuitive and user-friendly.

All planned functionalities were implemented, but there is also a space for improvement. For example, the admin panel, where it will be possible to manage users, namely to add them to groups or to manage user permissions in a more UX-friendly way, can be a future application extension. Also, a search field is a necessity for such a type of application, where lots of files are stored.



# Appendix A

## Bibliography

- [1] Stack overflow trends. <https://insights.stackoverflow.com/trends>, 2020. [Online; accessed 19-March-2021].
- [2] Econsultancy. <https://econsultancy.com/>, 2021. [Online; accessed 17-January-2021].
- [3] Yudhajit Adhikary. Client side rendering vs server side rendering in react js and next js. <https://yudhajitadhikary.medium.com/client-side-rendering-vs-server-side-rendering-in-react-js-next-js-b74b909c7c51>, April 6th 2020. [Online; accessed 19-March-2021].
- [4] Neoteric Software Development Agency. Single-page application vs. multiple-page application. <https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58>, 09 2017. [Online; accessed 25-April-2021].
- [5] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. *International Journal on Semantic Web and Information Systems*, 5:1–22, 07 2009.
- [6] David Capka. Lesson 2 - uml - use case diagram. "<https://www.ict.social/software-design/uml/uml-use-case-diagram>", 2021. [Online; accessed 27-April-2021].
- [7] Per Christensson. *TechTerms The Computer Dictionary*. TechTerms, 2020. Available at <https://techterms.com>, [Online; accessed 22-February-2021].
- [8] Google Developers. *How to get your website on Google Search*. Google, 2021. Available at <https://developers.google.com/search>, [Online; accessed 15-March-2021].
- [9] Alexis Deveria. Can i use javascript. <https://caniuse.com/?search=javascript>, 2021. [Online; accessed 23-March-2021].
- [10] Hiren Dhaduk. Best frontend frameworks of 2020 for web development. <https://www.simform.com/best-frontend-frameworks>, 02 2020. [Online; accessed 15-March-2021].

- [11] Andrew Dillon. *User Interface Design*. 01 2006.
- [12] ECMA International. *Introducing JSON*, 2017. Available at <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/1>, 2nd edition [Online; accessed 15-March-2021].
- [13] AltexSoft Software engineering. The good and the bad of vue.js framework programming. <https://medium.com/styled-components/why-styled-components-2deeed757cfa>, 09 2019. [Online; accessed 15-March-2021].
- [14] CodeIgniter Foundation. Helper functions. [https://codeigniter.com/user\\_guide/general/helpers.html](https://codeigniter.com/user_guide/general/helpers.html), 02 2021.
- [15] The Interaction Design Foundation. What is usability? <https://www.interaction-design.org/literature/topics/usability>, 2020. [Online; accessed 17-February-2021].
- [16] Phil Pluckthun Glen Maddern, Max Stoiber. *Styled-components documentation*, 2021. Available at <https://styled-components.com/doc>, version 5.3.0 [Online; accessed 25-April-2021].
- [17] Evan Jacobs. Why styled-components? - medium. <https://medium.com/styled-components/why-styled-components-2deeed757cfa>, 04 2020. [Online; accessed 25-April-2021].
- [18] Marek Jaroš. *Sémantický správce dokumentů*. May 2020. [Online; accessed 14-February-2021].
- [19] Marin Kaluža and Bernard Vukelic. Comparison of front-end frameworks for web applications development. *Zbornik Veleučilišta u Rijeci*, 6:261–282, 01 2018.
- [20] Zeinab Khalifa. Multi-page, single-page, or a hybrid? - medium. <https://medium.com/swlh/spa-mpa-or-a-hybrid-42fdf6b3415c>, Jun 2020. [Online; accessed 23-March-2021].
- [21] Ornaith Killen. Four reasons why site search is vital for online retailers. <https://econsultancy.com/four-reasons-why-site-search-is-vital-for-online-retailert>, 11 2013. [Online; accessed 17-January-2021].
- [22] Steve Krug. *Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition)*. New Riders Publishing, USA, 2005.
- [23] Chaitanya Kulkarni and Mukta Takalikar. Cseit183535 | analysis of rest api implementation. 01 2020.
- [24] Ryan Lanciaux. A feature based approach to react development. <https://ryanlanciaux.com/blog/2017/08/20/a-feature-based-approach-to-react-development/>, 08 2017. [Online; accessed 04-May-2021].

- [25] Tanner Linsley. *React Query documentation*, 2020. Available at <https://react-query.tanstack.com/quick-start>, version 3.16.0 [Online; accessed 20-April-2021].
- [26] Cinergix Pty Ltd. *About Creately*. Creately, 2021. Available at <https://creately.com>, [Online; accessed 08-March-2021].
- [27] Malinov Martin. *Systém pro správu dokumentů, souborů a datových zdrojů*. May 2019. [Online; accessed 14-February-2021].
- [28] Mozilla and individual contributors. *Web technology for developers*. MDN Web Docs, 2021. Available at <https://developer.mozilla.org/en-US/docs/Web>, [Online; accessed 19-March-2021].
- [29] Christian Osterbauer, Monika Köhle, Manfred Tscheligi, and Thomas Grechenig. Web usability testing - a case study of usability testing of chosen sites (banks, daily newspapers, insurances). 07 2014.
- [30] Mark Richards. *Software Architecture Patterns*. 2015.
- [31] Toni Bonitto Sara Cope. Scenarios. <https://www.usability.gov/how-to-and-tools/methods/scenarios.html>, 2021. [Online; accessed 07-May-2021].
- [32] Karan Shah. Client-side v/s server-side rendering: What to choose when? "<https://dzone.com/articles/client-side-vs-server-side-rendering-what-to-choose>", January 10th 2020. [Online; accessed 19-March-2021].
- [33] Facebook Open Source. *React official documentation*. Facebook, 2021. Available at <https://reactjs.org>, version 17.0.2 [Online; accessed 15-March-2021].
- [34] Stack overflow. *Stack Overflow. For developers, by developers*, 2021. Available at <https://stackoverflow.com>, [Online; accessed 15-March-2021].
- [35] Google Angular Core team. *Angular features and benefits*. Google, 2021. Available at <https://angular.io>, version 11.2.12 [Online; accessed 15-March-2021].
- [36] React Training and hundreds of contributors. *React Router documentation*, 2021. Available at <https://reactrouter.com/>, version 5.2.0 [Online; accessed 25-April-2021].
- [37] Vinay Hegde Vinuta Hutagikar. Analysis of front-end frameworks for web applications. 07(4), April 2020.
- ..





## Appendix B

### List of abbreviations

<b>DM</b>	Document Manager
<b>UI</b>	User Interface
<b>UX</b>	User Experience
<b>REST</b>	Representational State Transfer
<b>API</b>	Application Programming Interface
<b>HTTP</b>	Hypertext Transfer Protocol
<b>CRUD</b>	Create, Read, Update, and Delete
<b>JSON</b>	JavaScript Object Notation
<b>JSON-LD</b>	JavaScript Object Notation for Linked Data
<b>KBSS</b>	Knowledge-Based and Software Systems
<b>CTU</b>	Czech Technical University
<b>FEE</b>	Faculty of Electrical Engineering
<b>URI</b>	Uniform Resource Identifier
<b>MPA</b>	Multi-Page Application
<b>SPA</b>	Single-Page Application
<b>HTML</b>	Hypertext Markup Language
<b>CSS</b>	Cascading Style Sheets
<b>SSR</b>	Server-Side Rendering
<b>CSR</b>	Client-Side Rendering
<b>SEO</b>	Search Engine Optimization
<b>DOM</b>	Document Object Model
<b>PWA</b>	Progressive Web Application
<b>MVC</b>	Model, View, Controller
<b>DI</b>	Dependency Injection
<b>JSX</b>	JavaScript Extension Syntax
<b>SVG</b>	Scalable Vector Graphics
<b>URL</b>	Uniform Resource Locator





## Appendix C

### Instructions how to start the Document Manager application

#### C.1 Back-end part

The instructions how to run the back-end part were taken from the bachelor thesis of Marek Jaroš.

To run the Document Manager back-end you need to have installed the following technologies:

- Java 8
- Apache Maven 3 (or newer)

Before compiling, you need to change the application configuration file, which is located in the `application\semantic-document-manager\src\main\resources` directory.

- To connect your own SQL database, you need to uncomment the rows under the *External database* section and comment the rows under the *Local file database* section. Then, you need to set your SQL database address, username and password.
- To connect your own RDF database, you need to change the `repositoryUrl` in the *Spring DATASOURCE - SEM* section.
- The profile where the application will run is defined in `spring.profiles.active`. The *sem* profile means that the application will use the RDF database and the *jpa* profile – the SQL database respectively.
- To define the `security.authorization-point` you need to use this value  
`https://kbss.felk.cvut.cz/authorization-service/api/v1/users/current`

Then, compile the application and run it in your IDE. Alternatively, follow these steps:

1. Open a command prompt in the main project directory and enter the command  
`mvn clean package`

### C. Instructions how to start the Document Manager application

2. A new *document-manager-0.0.1-SNAPSHOT.jar* file will be created in the *target* directory

3. The file can be started with the command

```
java -jar document-manager-0.0.1-SNAPSHOT.jar
```

## C.2 Front-end part

To run the application front-end you need to have installed the following technologies:

- Node 12.13 (or newer)

The front-end part can be started with the commands:

1. `npm install` to install the needed packages
2. `npm start` to start the web application



## **Appendix D**

### **Content of the electronic attachment**

The electronic attachment contains the reference to the Gitlab repository where the source code of the Document Manager front-end is stored and the hash of the final commit.