Bachelor thesis

Responsive and Interactive Web Application for a Lawyer's Office

Host institution:

Law firm

MEYNIOGLU LAW

Responsible:

Maître Yusuf Meynioglu

Academic responsible:

Dr. Christian Franck

UNIVERSITÉ DU LUXEMBOURG

William Girgis

June 2021

Faculty of Science, Technology and Medicine

Bachelor in Applied Information Technology

Abstract

English:

In this thesis we will see how we can build a full website including its design , its functionalities, and the third-party APIs used. First, we will analyse users' interaction and needs using use case diagrams and see how we can build the services under a single roof. We will also compare two authentication systems, compare them and choosing one. We will then discuss on how we can improve a given outdated design from an old web site and transforming it in a much modern one. After having listed and analysed the needs, we will see how the full design has been made by having a look at the design structure hierarchy. We will then discuss and analyse codes that show how the authentication system has been built using Web Tokens and how the different services have been implemented. Finally, we will be looking at some screen shots and commenting them to visualise the final product including its design and the different services implemented.

Français:

Dans cette thèse, nous verrons comment construire une application web incluant son design et ses fonctionnalités, ainsi que les APIs de fournisseur tierce que nous utiliserons. Nous allons d'abord analyser l'interaction et les besoins des utilisateurs en utilisant des diagrammes de cas d'utilisation et ainsi voir comment nous pourrons implémenter ces services en arrière-plan. Ensuite nous analyseront deux systèmes d'authentification, nous les comparerons puis en choisirions un. Nous allons ensuite discuter sur comment potentiellement améliorer un vieux design d'un vieux site web et le rendre bien plus moderne. Après avoir listé et analysé les besoins, nous allons voir comment le site web complet a été fait en jetant un coup d'œil à la hiérarchie et la structure du design. Nous discuterons et analyseront les codes décrivent l'implémentation d'un système d'authentification et ceux décrivent l'implémentation des différents services implémenté. Finalement, nous jetterons un coup d'œil à quelques captures d'écran que nous commenterons et qui nous permettront de visualiser le produit final incluant le design et les différents servies mise en place.

Acknowledgements

I would like to thank my academic supervisor Dr.Christian Franck for giving me advice and supporting me through the organisation of my internship. I would like to thank my local supervisor Maître Meynioglu Yusuf for trusting and providing me this opportunity. Finally, I thank Mrs. Karola Theis for the time she spent correcting my text.

1 Table of contents

2 Introduction

As long as technology grows, the users' expectations grow, too. However, some websites nowadays have been built a long time ago and hence make their user experience a trial that is neither desired nor necessary. Being confronted with this problem, the law firm Meynioglu gave me the task of improving its online presence by creating a new website for the company.

The host institution is a law firm that owns a website which has been built several years ago which, hence, is not responsive on mobile devices, and has an old design. In addition, the host institution asked me to solve the following problem:

Quite often, the clients were requesting the legal files of their folder containing the elements related to their case. The idea was to provide an access to the clients such that they can see and download their files without having to request them to the institution. Therefore, they contacted me for building a new modern website with specifics requirements.

In this thesis we are going to realise a web application based on recent technologies that fulfill those requirements and provides a much better user experience. We use modern technologies such as Angular and MongoDB to implement a Single Page Application. We provide a fully functional prototype of the future website that includes all the features.

In Chapter 2 we analyse the requirements and the users' interactions. Then, in the Chapter 3, we will explain the implementation by looking at some code snippets. In the Chapter 4, we will show the resulting website and discuss the various features. Finally, we conclude by comparing the requirements stated in the introduction with the final result in Chapter 5. The rest of this chapter states the specifications and requirements.

2.1 Specifications and Requirements

The new web site must contain the **same features as the old web** site including:

- Allowing the admin to post publications by connecting himself to a given login route
- Allowing the clients to send a message via a contact page for finally receiving it as mail in the company's mailbox

- The new website must be fully responsive for mobile devices. Indeed, one of the main issues the company was facing regarding the website, was the non-responsive behavior of it.
- To answer the problem of clients requesting files, with the firm we agreed on implementing **new feature in the new website**: Allowing the clients to connect to a given route to access a folder where all the files related to their case are stored and allow them to download the files.
- Allowing the admin to add, modify, and delete users, as well as their files.

Let us summarize the final needs:

- **A responsive web application for mobile devices .**
- **A contact service for clients.**
- **A file download system for clients.**
- **A posting publication service for the admin user.**
- **A client and file management system for the admin user.**

We will be analysing the services that must implemented in the next chapter.

On request of the host institution, I will keep the **same routes as the old website** including:

- “Home” route that contains the law firm description.
- “Avocat” route that contains the list of avocats working at the law firm.
- “Publication” route where publications are displayed.
- “Skills” route where the expertise fields are described.
- “Contact” route where clients send mails to the company’s mailbox.
- “Login” route where the admin post modify and delete the publications.

Now that we have the needs well described, let us focus on the Design and Analyse chapter and see how the users will be interacting with the different services.

It is important to note that the host institution does not have a department of technology and therefore I had to work on my own. Most of the decisions taken result from several consultations with the law firm director.

3 Analyse and Design

In this chapter we will be analysing the choices made for designing the web site and analyse the user's interaction using actor models for a better understanding of the given problems. We will then analyse the hierarchy of the design components using an UML model. After having conceptualise the design, we will be looking at how storing the users' data, but also the publication list. This leads us to the finals and crucial topics which are the authentication system and the security.

3.1 Single Page Application (SPA) and Multiple Pages Application (MPA)

Regarding the conception of a new web application, the first question that comes up is: which type of web application is appropriate ? Well, there are in fact 2 main types of web application: Single Page Application (**SPA**) and Multiple Pages Application (**MPA**) Before giving my choice, let us get a small definition of both and consider their advantages and disadvantages.

MPAs are types of web application where pages are rendered multiple times based on user requests meaning that for each page requested by the user, a new resource is sent by the server to the client. MPAs main advantage is the SEO that makes highlighting the website on internet by giving you a higher rank in the search result page. They are however much slower and less secure since there are multiple pages to render on each user request and that we must secure each page.

SPAs are types of web application where pages are fully rendered once by the server, meaning that during the navigation, the user does not load any additional resources. This become evident that SPAs are faster and more user-friendly regarding the UX. SPAs are much better protected for the simple reason that we only have one page to secure. SPAs are built using a framework, which means that they have pre-defined features included and have some built-in protections against common web-application vulnerabilities.

It becomes obvious that my choice turned to a SPA application: because it is more secure and more user-friendly. Plus, SPAs are much nicer regarding the responsive behavior on mobile devices which will considerably improve the user experience.

3.2 Study Requirements and User Interactions

Let us now get into the user interactions for the file system by first illustrating the following use case diagram:

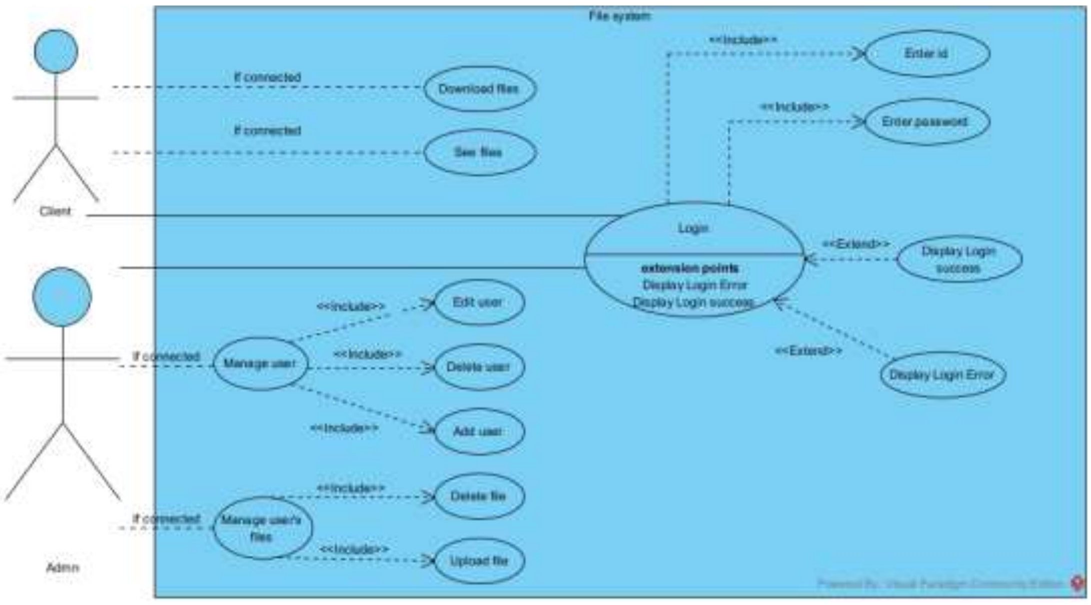


Figure 3-1 File system interactions

So, we get the following interactions:

- The client and admin user login by entering their credentials
- (Once connected) The user can see their files and download them.
- (Once connected) The admin manages users and users' files.

It is important to note that I could have split the users and files management system into two different systems, but for reason of implementation simplicity and because both data categories are directly related, it became logical to merge them into a same unique system.

Having now the global picture of the file system interactions, it is now time to analyse how the admin will be interacting with the publications and how the users will see it.

Let us have a look at the following diagram:

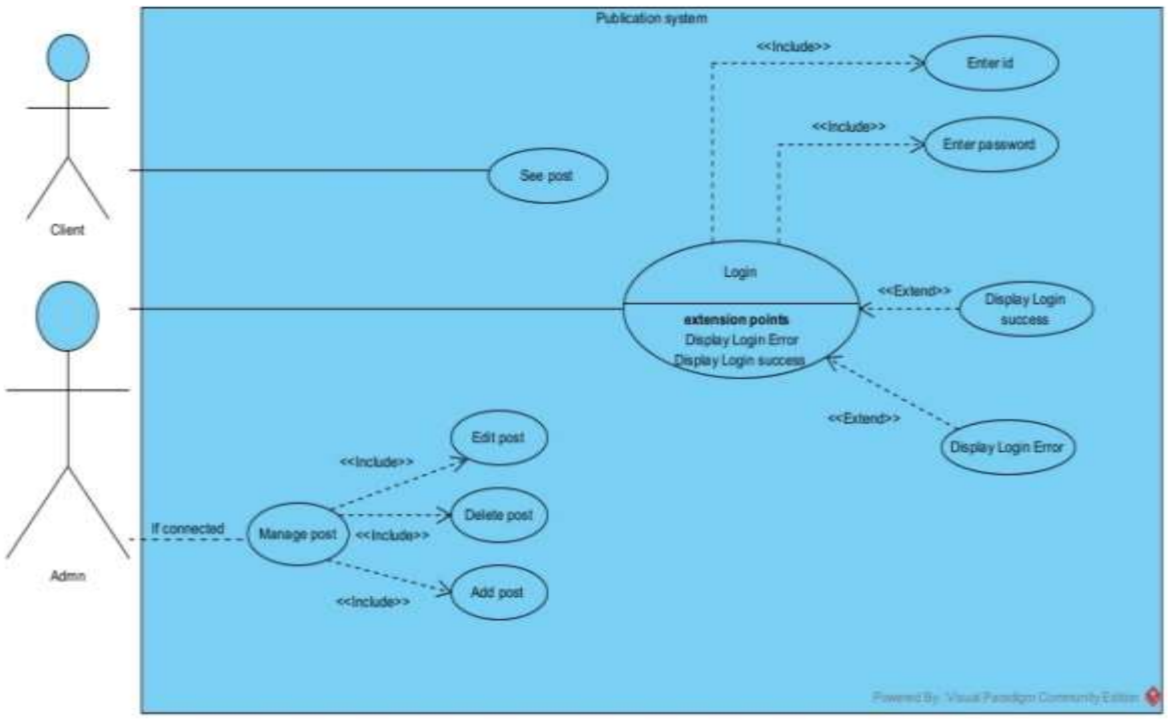


Figure 3-2 Post system interactions

It is as simple as that:

- (Once logged in) The admin adds, deletes, and modifies the publications displayed on the website.
- (Once logged in) The user sees and can download their files.

One last scenario we need to illustrate is the contact service. In that scenario, a secondary user comes into play: **the mail service provider.** The role of the mail service provider is to first store the mails, and then send them. We will discuss the one that I chose later in the implementation chapter , but first let us get into that interaction illustration:

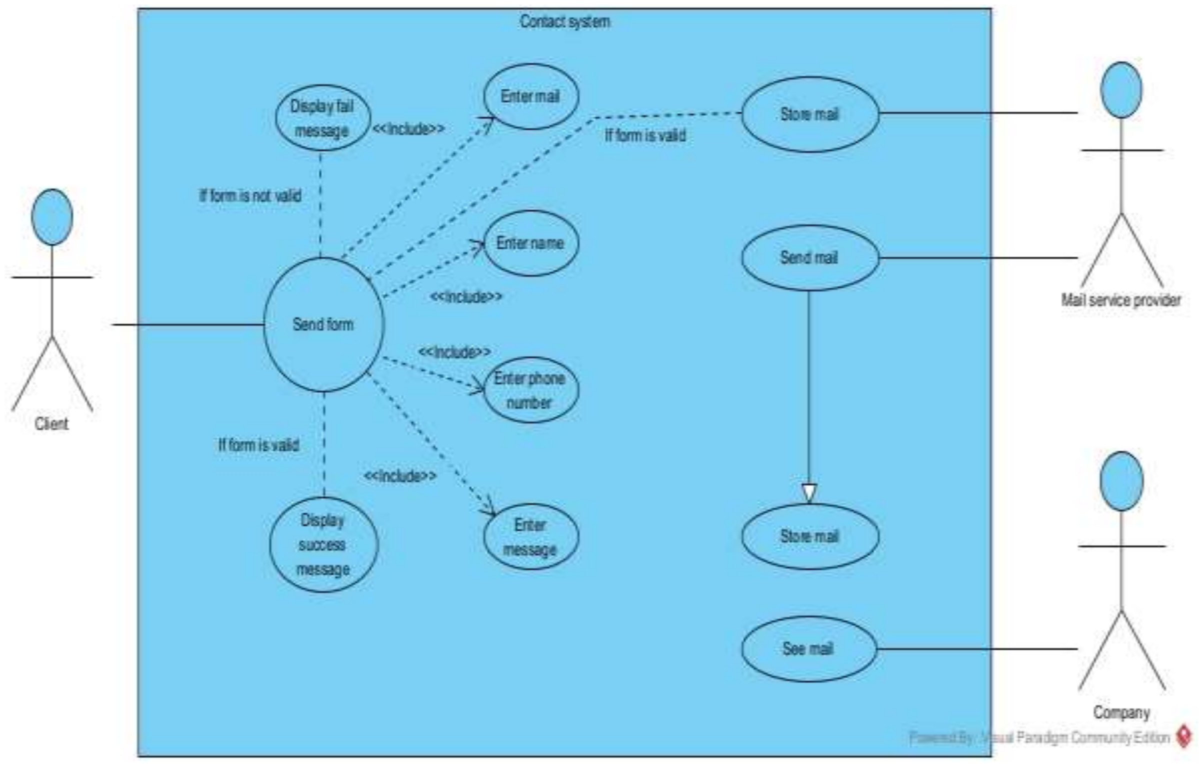


Figure 3-3 Contact system interactions

As illustrated, the client enters his **mail**, **name**, **phone number** and **message,** and send them all as a mail to the mail service provider which in turn sends them as a mail to the company's mailbox.

3.3 Data Model

Let us now analyse how the data will be structured by showing 2 class diagrams. The first diagram as follow illustrate the client credentials data and files relationship.

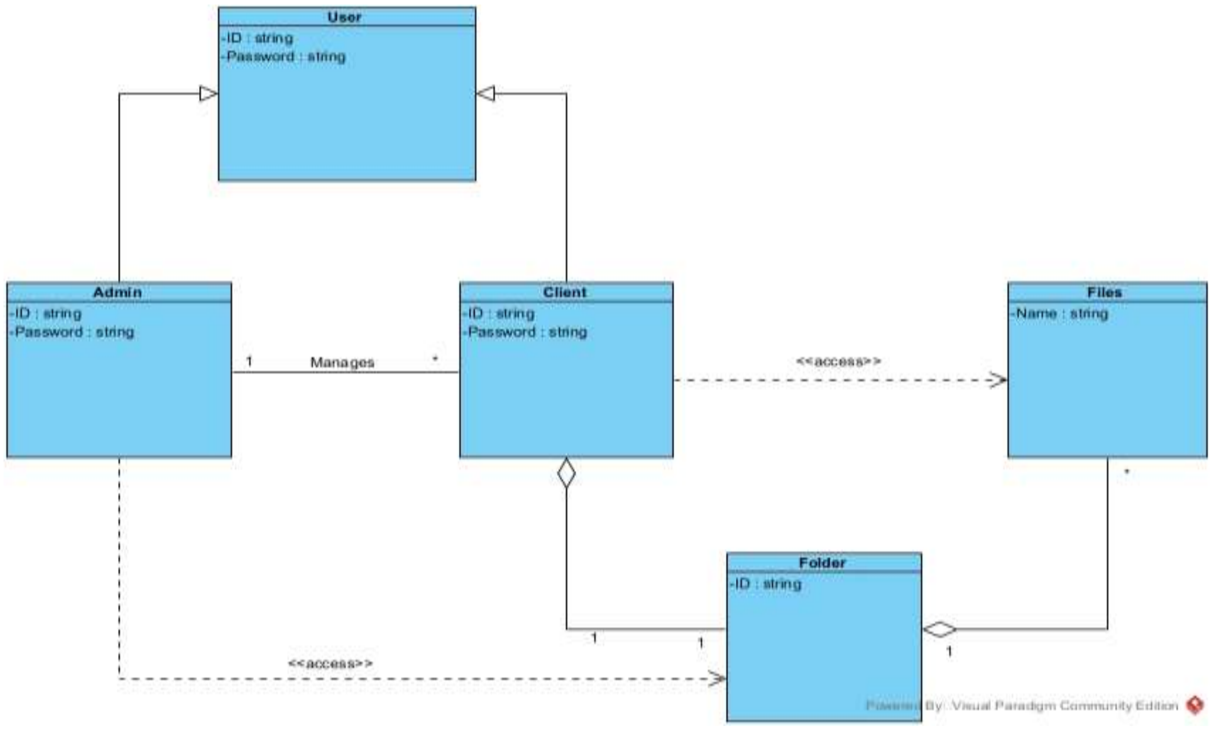


Figure 3-4 Credentials and files description

As shown above, the admin and the client both belong to the user parent class, which means that only one data type will be stored in the database: the user type. The files will be in a folder that in turn will be stored in the web server. As said in the introduction the final product will be a prototype. The reason is that the file service will not be implemented into the final product. My host institution came back on its decision during the project, and I therefore kept the file service just to enrich the thesis. This will be deleted when setting it in the production.

Let us now analyse the publication data model.

The model of each publication will be an JSON object. Here is the list of attributes for each publication:

- The title
- The date (that must be generated automatically after each post is added)
- The content

The reason I am using JSON, is to simplify the handling of publications. Modifying a publication for instance is easier in a structured JSON file, because each object property is accessible and manipulable. In addition, this reduces the number of queries sent to the database.

3.4 Client-Server Communication

One last important thing to think about, is what will we need for implementing such services regarding the communication between client and server. In the Client – Server architecture, the client corresponds to the front-end part and the server corresponds to the back-end part (the web server). The client (more precisely the admin) will need to perform CRUD (Create Read Update Delete) operations on users and publications. For implementing such communication, we will be using a RESTful API which is the most used interface that allows the client to access data from a server. A RESTful API is no more less than a piece of code that allows a client (in a client – server architecture) to make (http) requests to retrieve data from the server. For the full definition, here is the link [1] for the Wikipedia page.

There is one last communication way that comes into the picture: the SMTP (Simple Mail Transfer Protocol). Regarding the contact service, mails will be sent from the client via the contact sending form and therefore will be invoking the SMTP to send mails into the company's mailbox. To understand how it is going to work, let us count the steps that must be performed:

- First the client (front-end) sends a form via the contact view.
- Then a SMTP is invoked for sending the form as a mail to a mail server.
- Then the mail server will be sending it to the company's mailbox.
- Finally, the company's agent will be reading it.

Now the first question that comes up is: How can we implement such service model? We know that we need a SMTP server for it. I found in fact three ways of implementing this service:

- The first one is to use a personal mail address (from a third party) that will receive the message coming from the web site and send it. Even though it is the easiest way to send mails, it is also the less adapted for the current situation. Indeed, personal mailbox requires authentication system and except a real user to use it (like Gmail box for instance).

- The second one is to combine the webserver and a SMTP server. Even though it is technically possible it remains a bad option for security reasons. For instance, it does not automatically prevent the spam mails. Therefore, I will be using the third option.

- The third one is to use a mail service provider (from a third party) dedicated for companies. They often have pre-build secured system and can manage several mails at once. Obviously, in most of the cases and depending on the package you choose, it is not free of charge.

Now that we have a full description of users' and admin's interactions, and a good description of how data will be represented, it is now time to see how all of this has been implemented.

3.5 Authentication System Analysis

Regarding the systems that must be implemented, we will need a secure and simple authentication system. In this section we will be looking at two authentication systems, comparing them, and state which one is the more appropriate.

The first authentication method to analyse, is the **cookie-based authentication**. First, we need to define what cookies are. As well described in the **section.io** website [2]:

> *"A cookie is a small piece of data created by a server and sent to your browser when you visit a website.[…] A cookie-based authentication uses HTTP cookies to authenticate the client requests and maintain session information on the server over the stateless HTTP protocol."*

There are many advantages regarding this authentication system, you can look at the website for more details. However, there is one issue regarding this system: Cookie-based authentication is vulnerable to Cross Site Request Forgery (CSRF). As well described in the Owasp web site [3]:

> "The CSRF is an attack that forces an end user to execute unwanted action on a web application in which they are currently connected".

To fix this issue, extra security measures must be added meaning that I would need extra time. Therefore, it would not be a good choice in my case.

The second authentication system to analyse and the one I will pick up for the project, is the JWT-based authentication system. For the definition of JWT, the official **jwt.io** [4] website provides all the information we need. In short, the following definition explains well what it is:

"JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed."

The main advantages are the following:

- JWT is well flexible since a token can be generated from anywhere.
- JWT does not use any additional server resources since it is stateless, and the user's "session" is stored on the client-side
- JWT is easy to implement in Angular.

We will see how the system works exactly and how it has been implemented in the implementation chapter. Finally, the reason why I chose that system is that it is easy to implement in Angular and very flexible.

4 Development and Implementation

In this chapter, we will focus on the implementation methods of each part of the web site including:

- The design part and how to secure it
- The data storage implementation
- The authentication system

Before introducing this chapter, I would like to mention some important points:

- The IDE used along the project is **Visual Studio Code**. Built by Microsoft, it contains many features and provide a high flexibility in terms of formatting the code.
- Some of back-end code comes from a git repository [5] that I found by following a tutorial on a YouTube Channel [6]. This code and this tutorial helped me a lot for building the Authentication system and its security. The link to the YouTube channel and the git repository will be given in annex.
- The RESTful APIs have been implemented using the **Express** framework.

Now that these important points have been mentioned, let us turn to the actual implementation of the whole web site.

4.1 Front-End Development

4.1.1 Route Navigation

As mentioned in the previous chapter, SPA's application are built using a framework. The framework that I used to build the web application, is the Angular framework, developed by Google, it is always up to date and contains a lot of features. Before presenting the actual implementation of the application, I would like to define some of its important aspects.

Angular works using Node js, a set of APIs written in JavaScript that provides services such as building RESTful APIs, SMTP, communicating with different database, etc . Angular is a framework written in TypeScript that uses MVVM pattern for building web application. For the full definition of MVVM pattern, I read the **TutorialsPoint** [7] website. I will just keep the most important points:

- “The Model, View, ViewModel pattern is all about guiding you in how to organize and structure your code to write maintainable, testable and extensible applications”.
- *“Model* simply holds the data”
- *“ViewModel* acts as the link between the Model and the View”
- *“View* simply holds the formatted data and delegate everything to the model”

For building this pattern, Angular uses components. A component is a folder that handles a specific part of the application. Each of these parts (component) has a specific behavior, and/or view. Each component is a folder that contains four files but only three are important:

- The TypeScript file that represents the *ViewModel* and manages the view
- A HTML file that represents the *View*
- A CSS file.

Now that we got the first approach to the tool, let us now understand how the design has been drawn by illustrating the following components hierarchy:

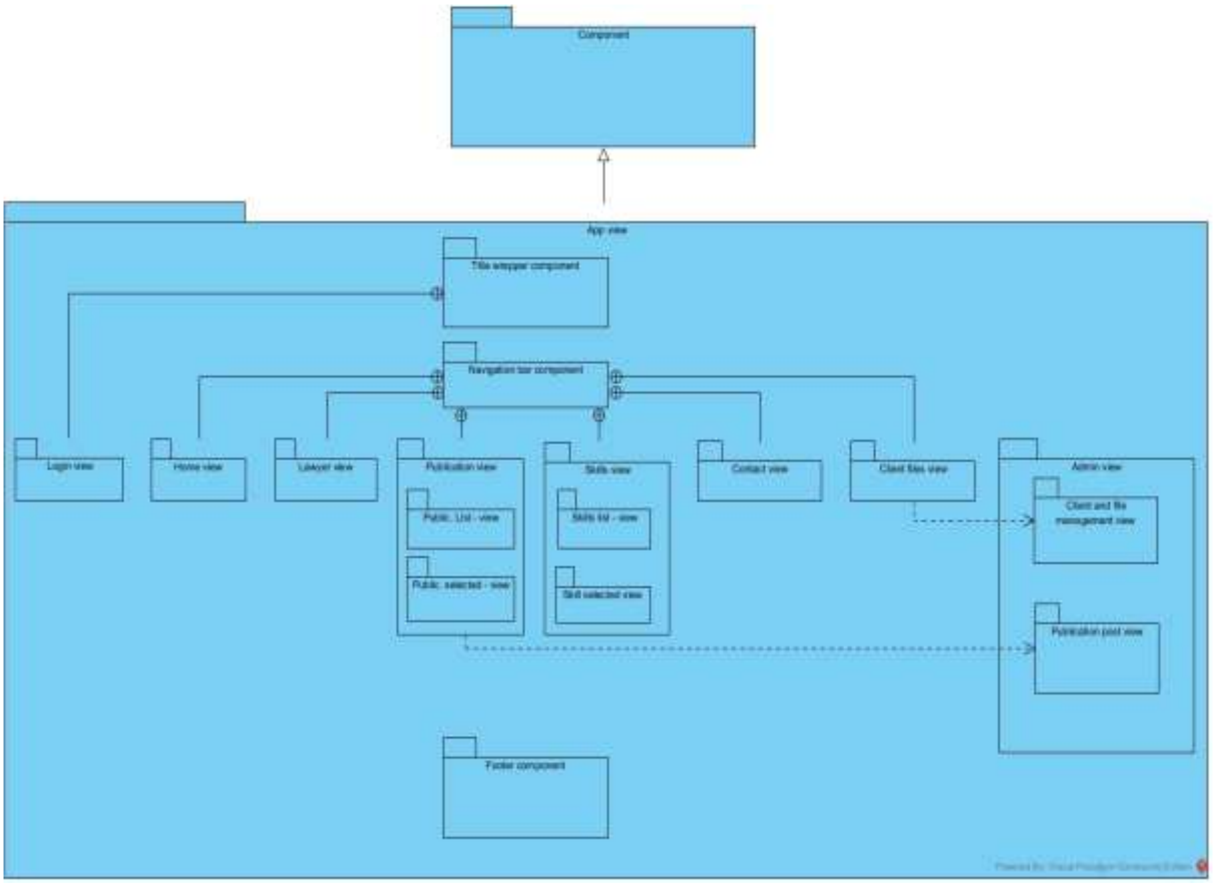


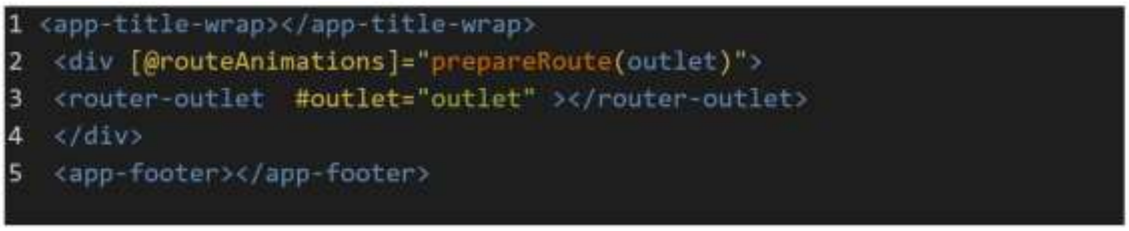
Figure 4-1 Components hierarchy

On this diagram, we see that the whole application is a component, that in turns includes other components. It is important to keep in mind that the views are also components and that only one view can be displayed at once. The diagram shows that all the navigation is done via the navigation bar component except for the login and the admin view. The login view is reached from the Title wrapper component and for obvious security reasons, the admin route is reached only via a specific route (not from the UI) that I am going to give to the admin user. At the end, the components have been classified as follows:

- Statics component (global components that are included in every view):
  - Title wrapper which is in fact the first page the user will see and that includes the login component which leads to the login page.
  - The navigation bar that leads to the home page, lawyer page, publication page, skills page, contact page, and if the user is connected, the files pages.
  - The footer that gives the company's coordinates.

- View's components (dynamically changed):
  - Lawyer view
  - Client's file view
  - Contact view
  - Publications' view
  - Skills view
  - Home view
  - Login view
  - Admin view
    - Post publication view
    - Client and Files management view

This following figure represents (as shown in the figure 3.1) the application view.:



```
1 <app-title-wrap></app-title-wrap>
2  <div [@routeAnimations]="prepareRoute(outlet)">
3  <router-outlet  #outlet="outlet" ></router-outlet>
4  </div>
5  <app-footer></app-footer>
```

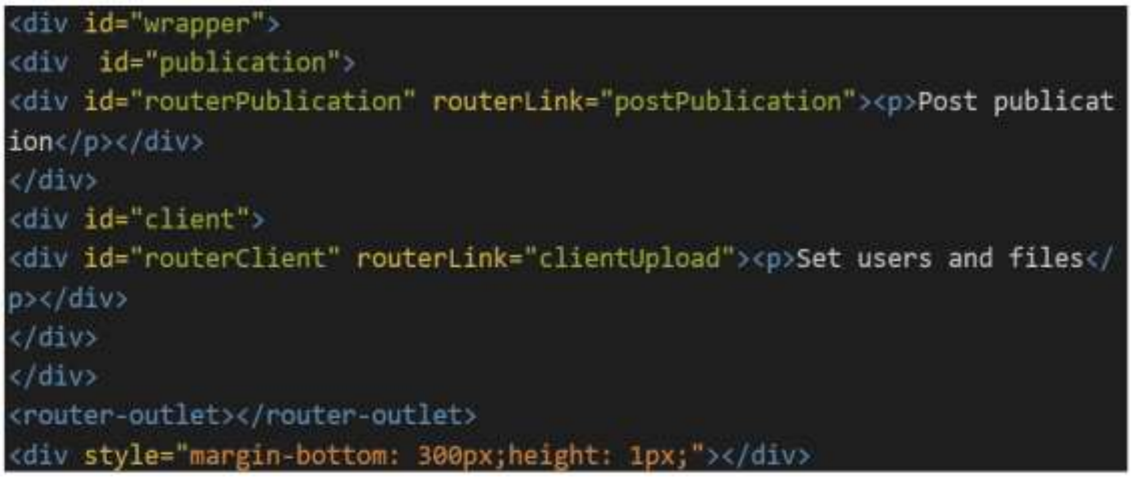
Figure 4-2 App component template

We see that we got our statics component: **app-title-wrap** line 1, and the **app-footer** line 5, displayed respectively, on the top and on the bottom of the user interface. You might now think: but where is the navigation bar? In fact, for design reason, the navigation bar has been placed inside the app-title-wrap component. It remains nevertheless an independent component. The **router-outlet** tag is what we previously illustrated in the figure 3.1, which is the view that changes during the user navigation. The div tag containing the **[@routeAnimations]** property, is just a wrapper allowing the views to trigger animations when swapping the views. The **routeAnimations** is the name I gave to the animation trigger implemented in the app.component.ts.

The most interesting part in the design implementation is the admin route. We have seen in the figure 3.1, that the admin route contains two other routes (views):

- The publication post view.
- The client and files management view.

The following figure shows us how the admin route looks like:



```
<div id="wrapper">
<div  id="publication">
<div id="routerPublication" routerLink="postPublication"><p>Post publicat
ion</p></div>
</div>
<div id="client">
<div id="routerClient" routerLink="clientUpload"><p>Set users and files</
p></div>
</div>
</div>
<router-outlet></router-outlet>
<div style="margin-bottom: 300px;height: 1px;"></div>
```

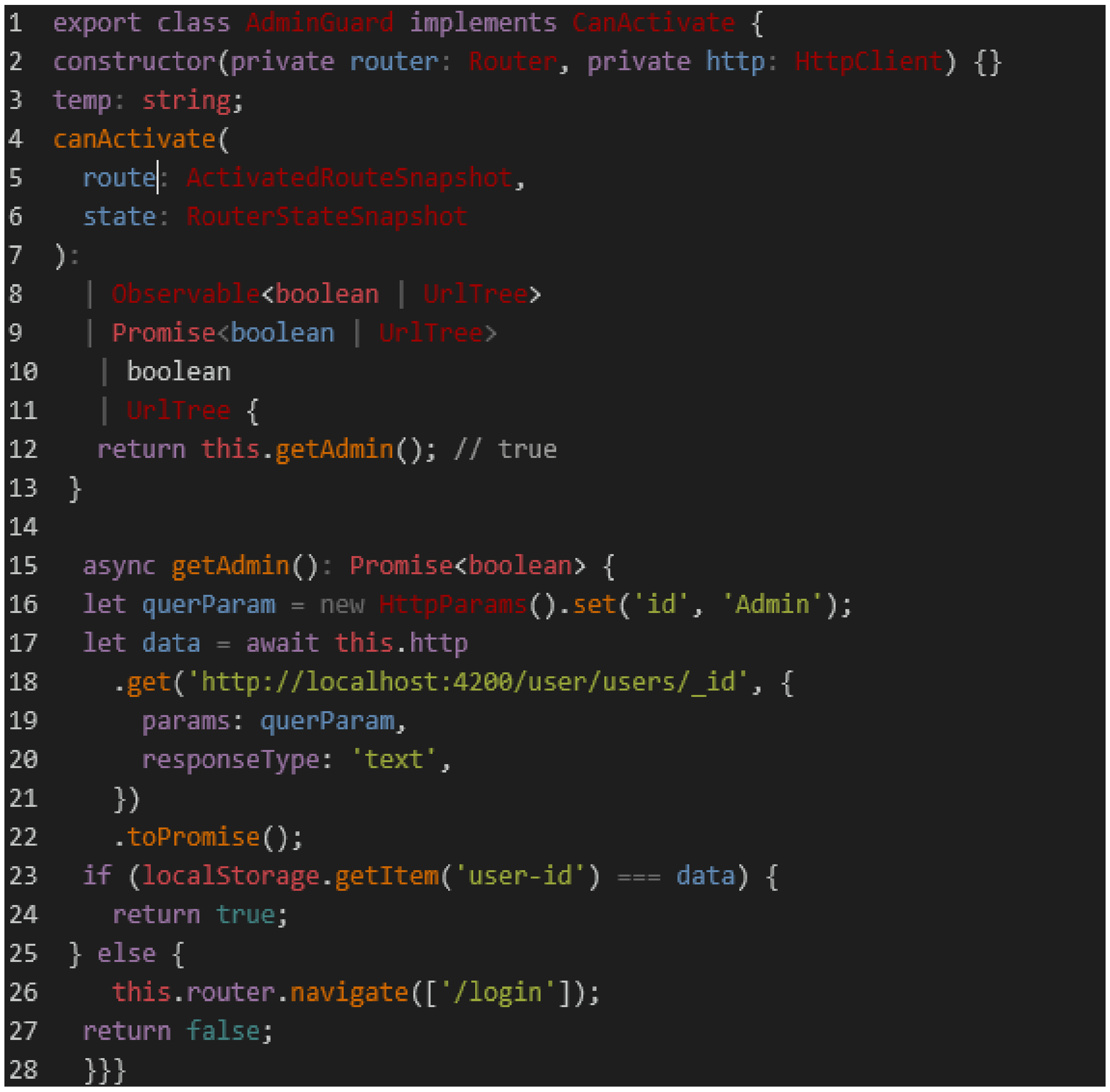
Figure 4-3 Admin component template

We see that like the figure 3.2, the admin component contains a **router-outlet** tag. This is where the views are switched. In addition, to allow the admin to navigate properly (without entering a new URL), I have added two links. The first one leads to the post publication view and the other one, to the client and file management view. Their system implementation will be seen in the back-end section.

4.1.2 Front-End Security

Let us now investigate how we can prevent unauthorized user to access the different routes such as the admin route or the file route (if not connected).

To reach that goal, Angular provides so-called route guard. As described in the official Angular’s website [8], guards are classes that implements the **CanActivate interface** that contains a **CanActivate method** returning a boolean. If the method returns true, the access is authorized, if false, the access is denied. For the project, I have been implementing two route guards. The first one is the basic Auth guard: “Files” route cannot be accessed if the user is not connected. The second one that we are going to see, a little bit trickier, is the admin guard. As its name suggests, this guard will be used for preventing any non-admin user, to access the admin route. Let us see how this route guard has been implemented:



```
export class AdminGuard implements CanActivate {
constructor(private router: Router, private http: HttpClient) {}
temp: string;
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
):
  | Observable<boolean | UrlTree>
  | Promise<boolean | UrlTree>
   | boolean
   | UrlTree {
   return this.getAdmin(); // true
 }

  async getAdmin(): Promise<boolean> {
  let querParam = new HttpParams().set('id', 'Admin');
  let data = await this.http
    .get('http://localhost:4200/user/users/_id', {
      params: querParam,
      responseType: 'text',
    })
    .toPromise();
  if (localStorage.getItem('user-id') === data) {
    return true;
 } else {
    this.router.navigate(['/login']);
  return false;
  }}}
```

Figure 4-4 Admin guard service

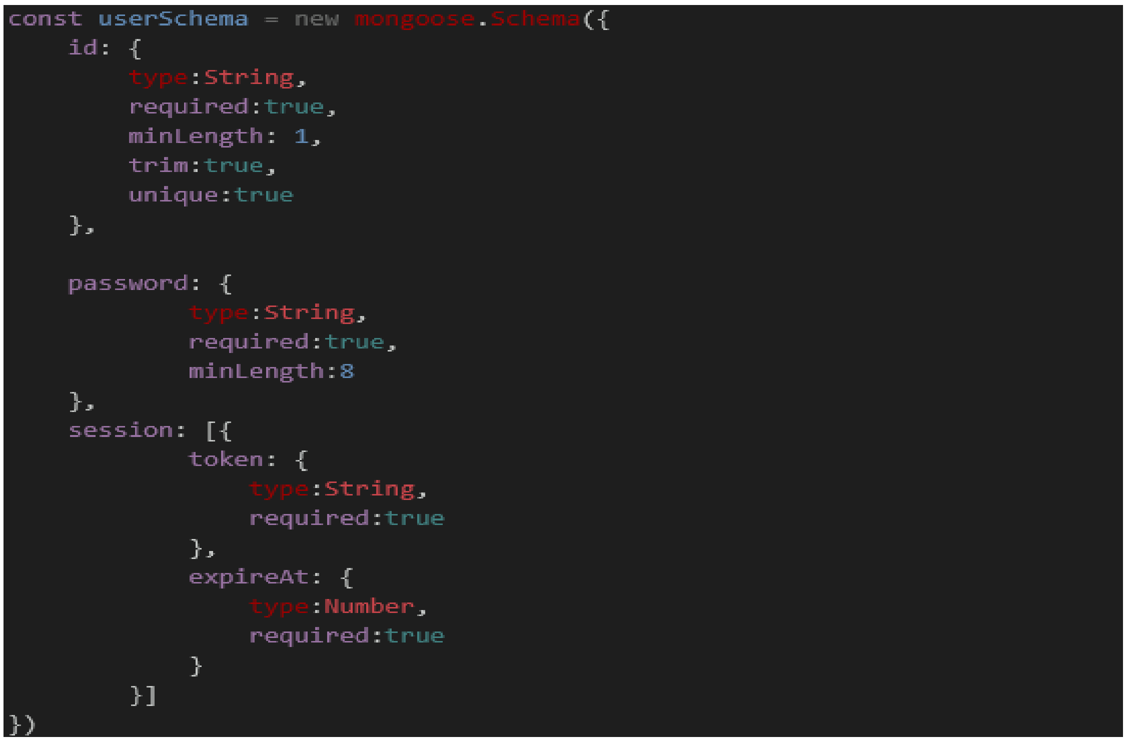
As shown in the figure, the **canActivate** function returns another function that in turns return a (promised) boolean. So, how this method works ? The method sends an http get request to the server, wait for a response, and then check if the user currently connected is the admin by comparing the current user ID on the computer (local storage) with the one in the DB. We will talk about the **local storage** in detail when starting the back-end development section.

But what does the ***async*** keyword mean- ? Async stands for asynchronous, which means that the current execution of the method is asynchronous and will block the current event-loop. In fact, ***async*** keyword goes hand in hand with the ***await*** keyword at line 17 which tells us that the thread will be blocking until a response is given (and then stored it in the ***data*** variable). The reason why I have been using this technique is that when I was executing the code, the http get request was triggered synchronously meaning that the execution code continued its execution and that the verification in the "if" statement was always returning false (no response was yet received from the http request).

One last security measure concerning the front-end is the implementation of an http Interceptor service. We will however talk about it when talking about the back-end security since it concerns more the server-side than the front-end side. But before going to the back-end side, it will be important to talk about the storage mediums regarding the clients' credentials first, the publications and finally the clients' files. Let us first start with the users' credentials.

4.2 Database Implementation

For storing user's credentials, I am using mongo DB for the simple reason that it works perfectly fine with the Angular framework, and it is free to use for the service I need (even though NoSQL databases are usually used for big and well-established companies). Therefore, I created a cluster in which I created a database. The database will in turn contain only one collection since the only data I am storing are the user's credentials. So how is it implemented concretely? To understand it, let us investigate the following figure representing a piece of my back-end code:



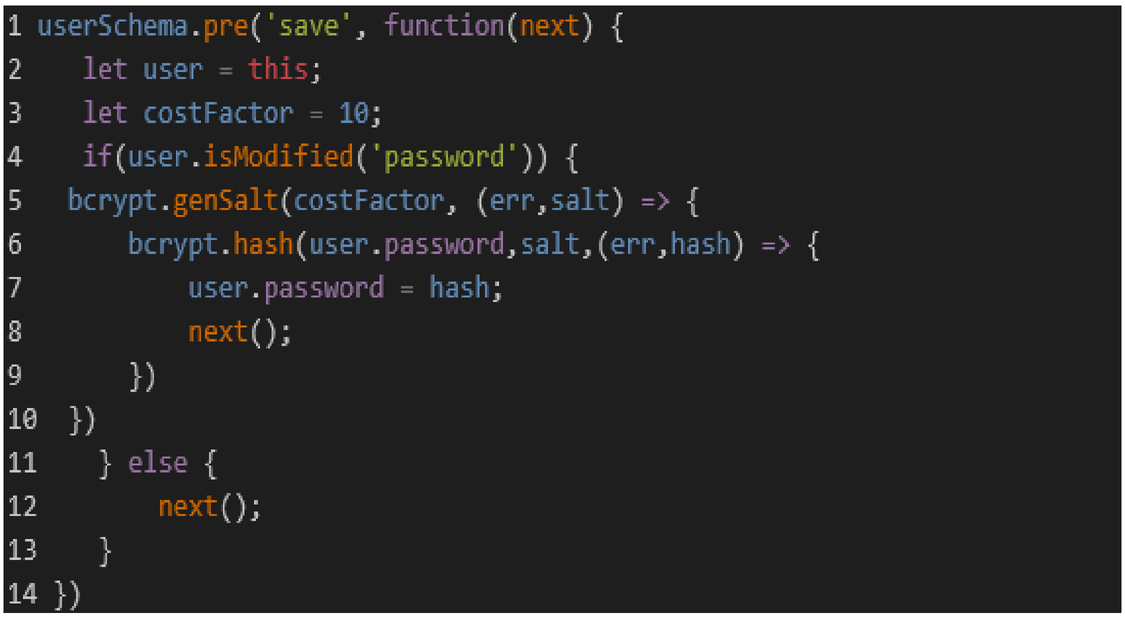
```
const userSchema = new mongoose.Schema({
    id: {
        type:String,
        required:true,
        minLength: 1,
        trim:true,
        unique:true
    },

    password: {
            type:String,
            required:true,
            minLength:8
    },
    session: [{
            token: {
                type:String,
                required:true
            },
            expireAt: {
                type:Number,
                required:true
            }
        }]
})
```

Figure 4-5User data definition

This figure describes a mongo schema [9] that defines the shape of the data that will be stored. It tells us what will be stored in the mongo database and how the user accounts will be defined. So, at the end we got:

- A user ID
- A password
- A session object

You might wonder now: why has a session to be added in the user´s definition and what does it represent? It is important to note, even though it seems obvious, that each user establishing a connection, will be connected only for predetermined lifetime. The session represents each connection established with the server which means that every time a user reconnects with its credential, it will update the session. However, this will be discussed later when talking about authentication. For the password, it is obvious that it should not be stored as plain text in the database, that would be a bad idea. To solve this, I have implemented a method that includes a library and that hashes a given user's password before storing it in the database. The library used for that goal is the ***bcrypt*** library. Here is the code that defines the function:



```
userSchema.pre('save', function(next) {
   let user = this;
   let costFactor = 10;
   if(user.isModified('password')) {
  bcrypt.genSalt(costFactor, (err,salt) => {
      bcrypt.hash(user.password,salt,(err,hash) => {
          user.password = hash;
          next();
      })
  })
    } else {
        next();
    }
 })
```

Figure 4-6 Hash password function

Mongo DB allows us to create methods (some statics too) belonging to this specific schema . In that case the function is predefined by mongo. The **pre** corresponds to a middleware that will be triggered every time a user's password is modified or a user is created.

Now, what about the users' files, how will we be storing them? For the user's files, I will be storing them into the web server directly. You might think that regarding the data sensitivity, it would be a bad idea, and you are right. However, as I mentioned in the introduction, the final product is a prototype and hence, will not be on production before all security issues are fixed. To delegate a folder to every single user, I have named each folder with the user's ID generated automatically by mongo with a hash function. In fact, mongo DB delegates to every single item (document) a specific hashed string **_id** even though we are not providing one. We will see how the file system has been implemented in the next section. But for now, it is time to get into the most interesting part of the project: the authentication system.

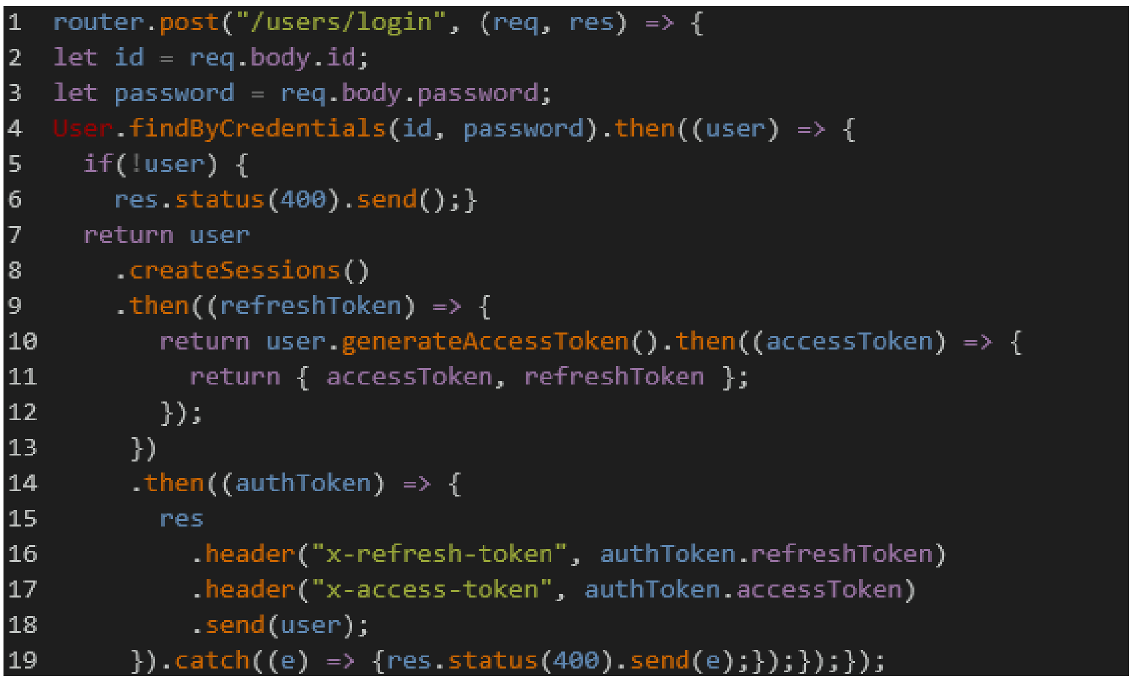
4.3 Back-End Development

Before introducing this section, I would like to mention that I have been following a tutorial for this part of the project, the Devstackr YouTube channel.

As previously said , for the authentication system I have been using the JWT (JSON Web Token) system. This system offers a high flexibility aspect and is easy to implement. So, here is how it works:

- The client connects to the server by sending an http request. The user's ID and password are checked, and if they correspond, the server will store the user in a new session, that in turns will be stored in the server memory. As described in the figure 3.5, a session is composed of a token, and an expiration time. The (refresh) token represents the key that will be used to refresh the expired time delay of the session. This means that, if the token has expired, the session cannot be refreshed and therefore, the user will have to reconnect (create a new session).
- After having stored in the server's memory the user's session, it will send back an access key (the access token) that will represent the key access for retrieving data on the server and for sending future http requests to the server.
- Once the client receives his access key, he can keep sending requests for the lifetime of that key.

Now let us investigate some pieces of code that illustrate most important parts of that system:



```
router.post("/users/login", (req, res) => {
let id = req.body.id;
let password = req.body.password;
User.findByCredentials(id, password).then((user) => {
  if(!user) {
    res.status(400).send();}
  return user
    .createSessions()
    .then((refreshToken) => {
      return user.generateAccessToken().then((accessToken) => {
        return { accessToken, refreshToken };
      });
    })
    .then((authToken) => {
      res
        .header("x-refresh-token", authToken.refreshToken)
        .header("x-access-token", authToken.accessToken)
        .send(user);
    }).catch((e) => {res.status(400).send(e);});});});
```

Figure 4-7 Login route

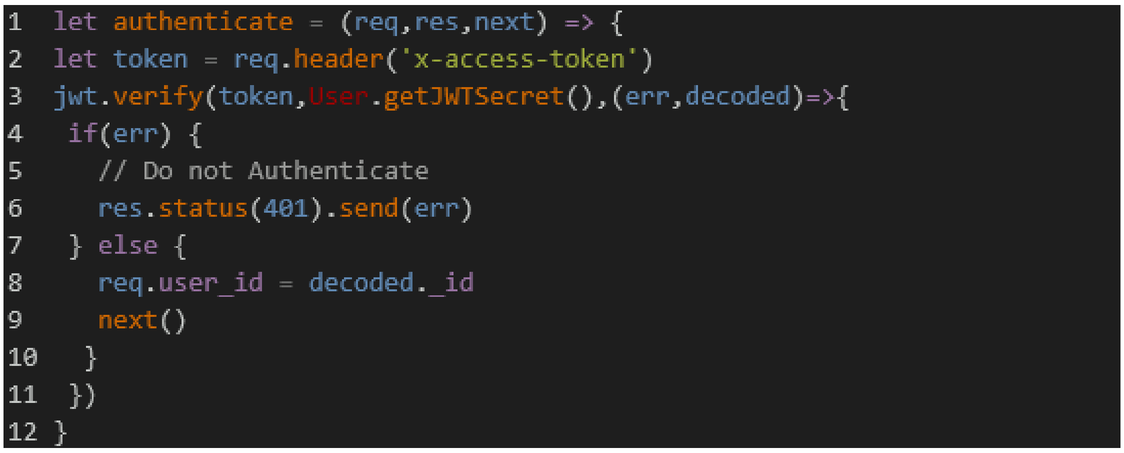
This piece of code represents the APIs gateway for the client's login. Here is what is happening when a request is sent to that route:

- The server takes the body's request including the user ID and password (line 2 and 3).
- Then the server checks whether there exists a user with these credentials (line 4).
- If a user is found, then a new session is generated (with a new refresh token) for that user (line 8).
- Then with that refresh token, a new access token (the actual key used to perform the future requests) is generated (line 10).
- The server finally sends both tokens by putting them into the headers of the response (line 15 – 18).

The question that comes up is: why do I need a refresh token ? Because at the end, only one token is required for checking if the user is the one he pretends to be. The reason is the following:

Imagine a third party that gets access to that access token, he would be able to connect with that account and get access to the data. The problem is that we cannot restrict the access (logout the user) of that third party only using that access token since it is a unique key.

Here is when the refresh token comes into the picture. On every client request, the server checks whether the access token is still valid (it has usually a short life) , if not, then the refresh token (that has a longer life) will be generating a new access token, storing it in the new session and will therefore restrict the access for others unauthorized users. The following figure represents the middleware that will check the access token validations:

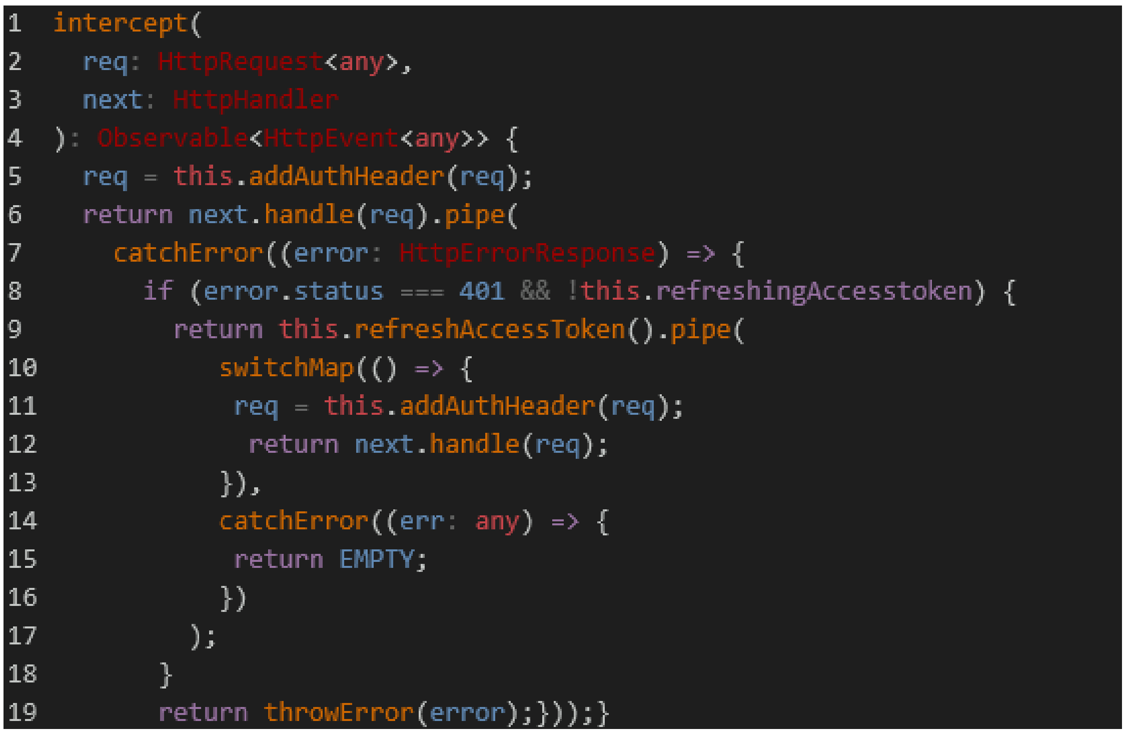


```
let authenticate = (req,res,next) => {
let token = req.header('x-access-token')
jwt.verify(token,User.getJWTSecret(),(err,decoded)=>{
 if(err) {
   // Do not Authenticate
   res.status(401).send(err)
 } else {
   req.user_id = decoded._id
   next()
  }
 })
}
```

Figure 4-8 Access token verify middleware

On every client's request, the access token is taken and checked by the server (line 2 -4). At the end, we got two tokens with a predefined lifetime. For the access token, I have set up his lifetime to 15 minutes and for the refresh token I have set up his lifetime to 1 hour.

As mentioned previously in the design section, an http interceptor has been created for checking the access token validity on every single request. Here is the following code that triggers the access token verification on the back-end:



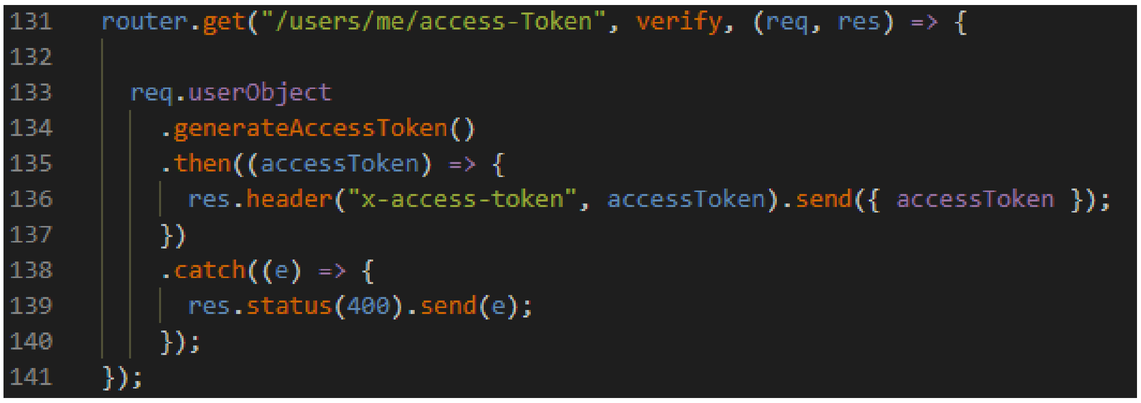
```
1  intercept(
2    req: HttpRequest<any>,
3    next: HttpHandler
4  ): Observable<HttpEvent<any>> {
5    req = this.addAuthHeader(req);
6    return next.handle(req).pipe(
7      catchError((error: HttpErrorResponse) => {
8        if (error.status === 401 && !this.refreshingAccesstoken) {
9          return this.refreshAccessToken().pipe(
10            switchMap(() => {
11             req = this.addAuthHeader(req);
12              return next.handle(req);
13            }),
14            catchError((err: any) => {
15             return EMPTY;
16            })
17          );
18        }
19        return throwError(error);}));}
```

Figure 4-9 Http interceptor (front-end)

This code performs the following:

- First the current access token is stored in the header of the request (line 5).
- Then we check whether an error has returned (meaning that the access token has expired) and if the token is not refreshing (line 8-9).
- Finally, once the token is valid, the request can continue.

To complete the picture, we need to see what the http interceptor triggers on the back-end side. Here is the code:



```
router.get("/users/me/access-Token", verify, (req, res) => {

  req.userObject
    .generateAccessToken()
    .then((accessToken) => {
      res.header("x-access-token", accessToken).send({ accessToken });
    })
    .catch((e) => {
      res.status(400).send(e);
    });
});
```

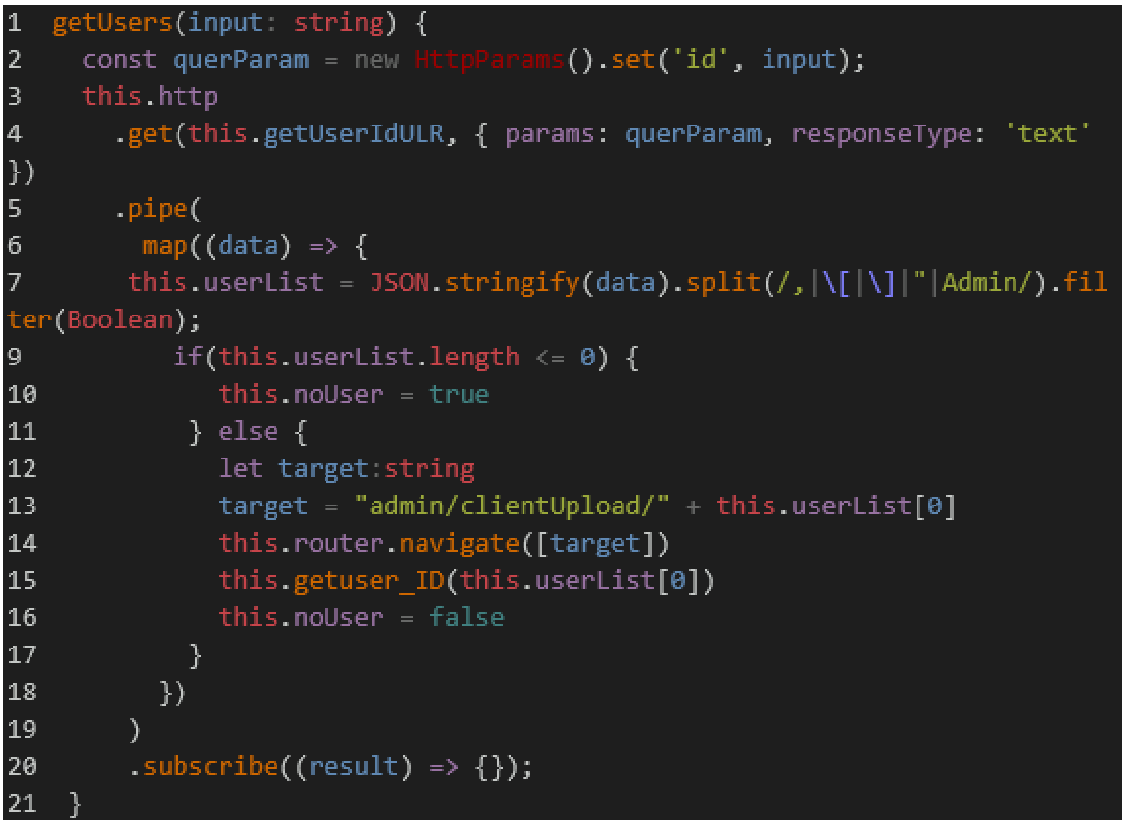
Figure 4-10 Http interceptor (back-end)

This code represents the server’s route that generates the access token. We can note that a middleware is implemented with the name **verify**. This middleware corresponds to the check of the refresh token validity. If the **verify** middleware returns an error (the refresh token has expired), then the access token will not be generated and therefore will disconnect the user. Otherwise, the server sends the new access token in the header’s response.

Now that we have seen the most important parts of the authentication system, it is time to see how the file system has been implemented including how the users will be downloading their files, and how the admin will be storing them.

4.3.1 Users / Files Management System Implementation

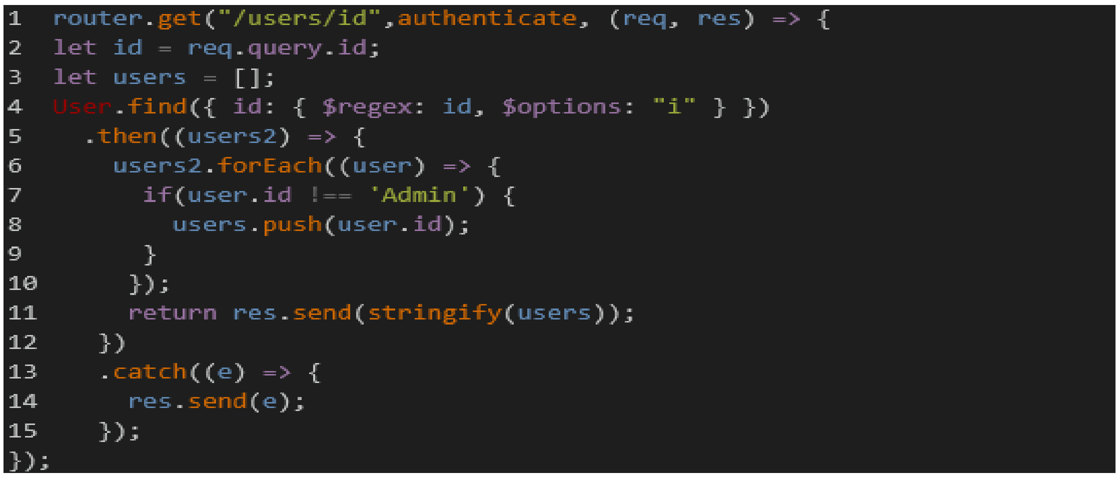
As previously mentioned, the admin will be able via the client and file management view, to perform CRUD operations on clients, and will be able to upload and delete their files. Now the question is: how all this parts has been implemented ? In the TypeScript file of the client-file component, functions have been defined for each single operation described below. Each of these functions contain an http request method for their dedicated functionality. Here is an example with the **getUser()** function that provide the list of the clients saved in the Mongo database :



```
getUsers(input: string) {
  const querParam = new HttpParams().set('id', input);
  this.http
    .get(this.getUserIdULR, { params: querParam, responseType: 'text'
})
    .pipe(
      map((data) => {
     this.userList = JSON.stringify(data).split(/,|\[|\]|"|Admin/).fil
ter(Boolean);
        if(this.userList.length <= 0) {
          this.noUser = true
         } else {
          let target:string
          target = "admin/clientUpload/" + this.userList[0]
          this.router.navigate([target])
          this.getuser_ID(this.userList[0])
          this.noUser = false
         }
       })
     )
     .subscribe((result) => {});
 }
```

Figure 4-11 Get user list function (front-end)

The function is triggered on the component (view) initialisation, after deleting a user, adding a user, after modifying a user and after searching for a user. The parameter that the function takes, is a string that will be passed as a regex expression on the server-side. This regex functionality has been implemented to let the admin finding a client by entering only the first characters. The **userList** variable corresponds to the list of users that has been found and returned by the server. For a better understanding of the function, we must analyse what the server does when this function is triggered. So let us continue with the next figure:



```
router.get("/users/id",authenticate, (req, res) => {
let id = req.query.id;
let users = [];
User.find({ id: { $regex: id, $options: "i" } })
  .then((users2) => {
    users2.forEach((user) => {
      if(user.id !== 'Admin') {
        users.push(user.id);
      }
    });
    return res.send(stringify(users));
  })
  .catch((e) => {
    res.send(e);
  });
});
```

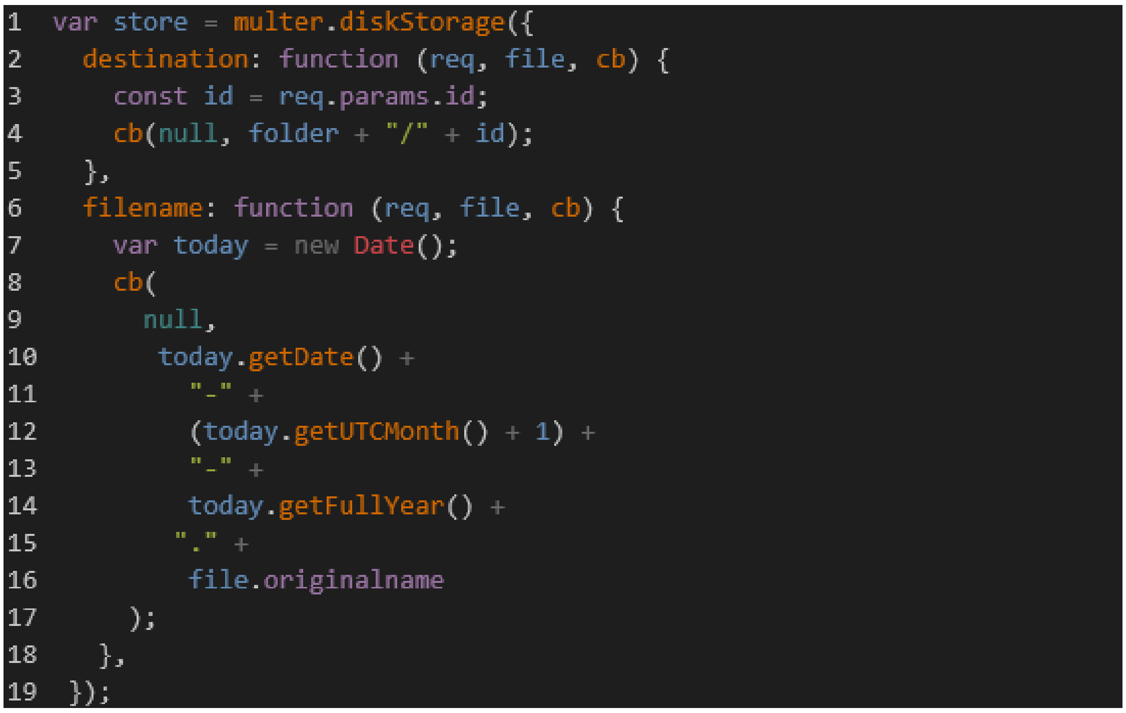
Figure 4-12 Get user list function (back-end)

We can note that the middleware described in the figure 3.8, is implemented in the method (line 1) and checks if the admin access token is still valid before continuing the code execution.

So, what does the server do when the admin requests the user's list? Of course, it first takes the parameter (string) sent from the admin UI and stores it in a variable line 2. Then it searches for a user (or list of users) corresponding to the given string by passing it as regex in the **User.find** function line 4. Then it checks for every single user found, whether it is the admin user or not and, if it is not the admin user, it pushes the user in an array that will be sent as response to the client. Why is that check required ? Simply because the admin user has the same data definition than other users and is placed in the same document collection. We do not want the "Admin" user displayed in the client list on the admin GUI.

Well, now that we have seen how users' ID is obtained by the admin, let us investigate how the admin will be storing the files for the users. As you remember, in the **Database Implementation** section, I said that a folder will be allocated to each client by naming them with their corresponding ID (automatically generated by mongo). The folders are created at the same time a user is created and deleted at the same time a user is deleted.

Let us first see how the admin is storing (in fact uploading) the file in the user's folder. For that, the admin user selects a user in the user list by clicking on it. Then the admin user clicks on the upload button, selects all the files to upload, and then upload them. The function triggered when the admin uploads the files is the following:



```
1  var store = multer.diskStorage({
2    destination: function (req, file, cb) {
3      const id = req.params.id;
4      cb(null, folder + "/" + id);
5    },
6    filename: function (req, file, cb) {
7      var today = new Date();
8      cb(
9        null,
10        today.getDate() +
11          "-" +
12          (today.getUTCMonth() + 1) +
13          "-" +
14          today.getFullYear() +
15         "." +
16          file.originalname
17      );
18    },
19  });
```

Figure 4-13 Upload files code

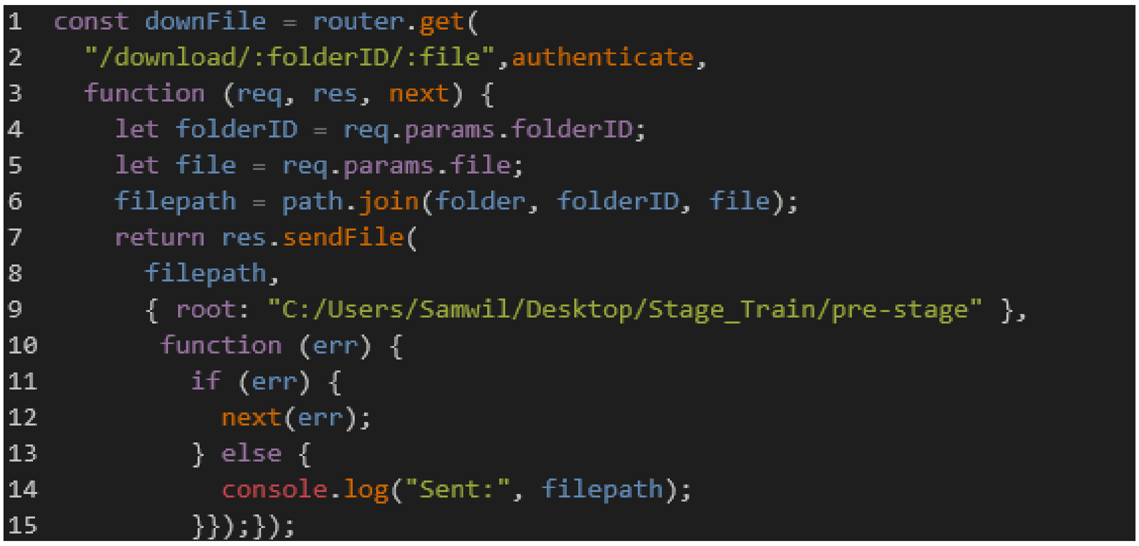
For uploading files, I am using a middleware provided by node.js that is mainly used for that purpose. Before describing how the codes works, it is important to note that the use of this middleware is only adapted when it is just a specific user that uploads the files (admin user in that case). The use of this middleware on a global scope might be dangerous since a user would be able to upload a malicious file in the web server.

The **disckStorage** keyword is an engine that provides a specific functionality of the middleware. In that case, the functionality is to store a file. The first parameters that the engine takes is the destination. You can see that an ID is provided as query parameter. This ID corresponds to the user's folder that the admin is requesting. After the destination set up, a filename must be provided at line 6. You can see here that the name I am giving to every file, is its authentic name file (the one already stored in the admin's computer), plus the current date in the format "dd/mm/yy". The reason why the date is added, is to let the user know, at which date the file has been uploaded.

Another important point regarding the implementation of the file system is how the clients are downloading their files. This works in three steps but before explaining them, I will just illustrate what local storage is, as I mentioned in the **security** section when we were dealing with the front-end development. **Local storage** is a Web API that maintains a separate storage area on your computer. It allows to keep data that might be useful during the user navigation. So here are the three steps:

- The client connects by sending a request to the server (as described in the section 1 of this chapter) that will in turn provide the user ID (the one generated by mongo DB) that will be stored on the user's **local storage**.
- Once this ID is stored, the user will access its file in the **client file** route by querying the server (with its ID) to get the list of the files. At this stage, the list of files is displayed on the UI.
- Finally, the client will be able to trigger a download file method by clicking on a button next to the file's name. Each download function triggered takes in parameters the corresponding file name.

For a better understanding of that process, I will just be illustrating the following back-end code:



```
1  const downFile = router.get(
2    "/download/:folderID/:file",authenticate,
3    function (req, res, next) {
4      let folderID = req.params.folderID;
5      let file = req.params.file;
6      filepath = path.join(folder, folderID, file);
7      return res.sendFile(
8        filepath,
9        { root: "C:/Users/Samwil/Desktop/Stage_Train/pre-stage" },
10        function (err) {
11         if (err) {
12           next(err);
13         } else {
14           console.log("Sent:", filepath);
15         }});});
```

Figure 4-14 Downloading files code (back-end)

We can note that once again, the authenticate middleware is implemented for checking the access token.

So here is what the server does on user's file requests:

- First the server stores in a variable the user ID provided by the user's local storage (line 4).
- Then the file name is taken from the request parameter and stored in a variable (line 5).
- Then a path is created by merging the **folder** variable (that defines the path to the folders list), the user's ID and then the file name (line 6).
- Finally, the server responds by sending the corresponding file to the client (line 7).

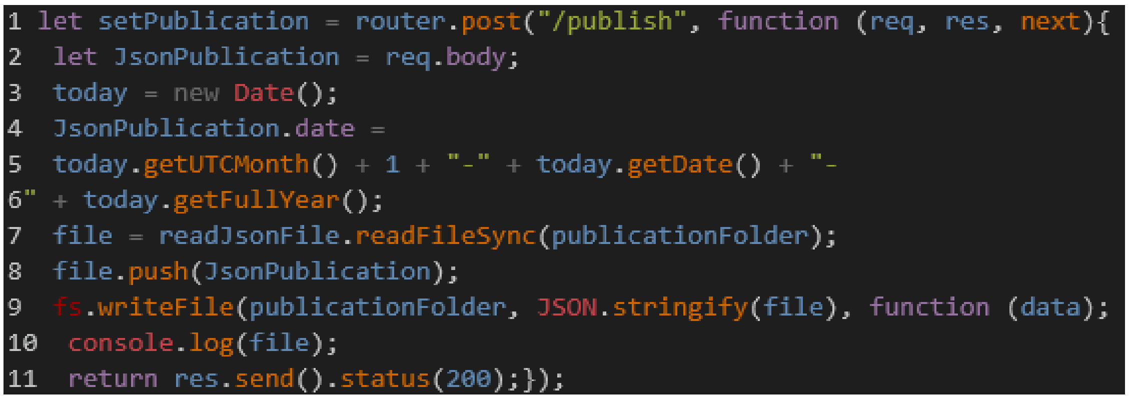
Now that we have seen how the clients and files system management works, it is time to get into the Publication post system implementation.

4.3.2 Post Publication System Implementation

As described in the Chapter 2, in the publication interaction's section, the data format of each publication will be the JSON object with its attributes : **title**, **date** and **content.** So here is how this system works:

- The admin once connected, goes into the publication post view.
- From there, the admins can add a post by sending to the server in a body's http request, the json object describing the publication
- The admin can modify a post, by sending in an http request, the new publication, plus the index corresponding to its emplacement in the publication list array.
- The user can delete posts by just providing the publication's index to the server.

For a better understanding of the system, let us investigate a piece of back-end code that represents an important part of the system:



```
1 let setPublication = router.post("/publish", function (req, res, next){
2  let JsonPublication = req.body;
3  today = new Date();
4  JsonPublication.date =
5  today.getUTCMonth() + 1 + "-" + today.getDate() + "-
6" + today.getFullYear();
7  file = readJsonFile.readFileSync(publicationFolder);
8  file.push(JsonPublication);
9  fs.writeFile(publicationFolder, JSON.stringify(file), function (data);
10  console.log(file);
11  return res.send().status(200);});
```

Figure 4-15 Add publication code (back-end)

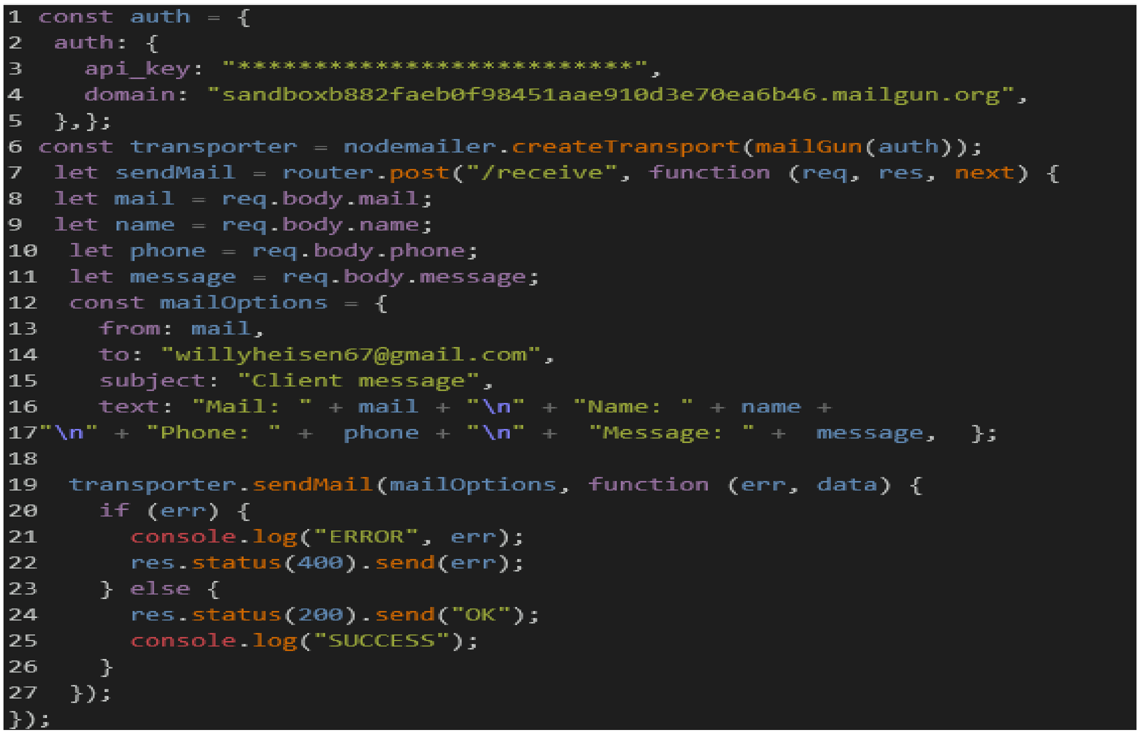
This code shows us the function triggered by the server when the admin user adds a publication. First the object is taken from the body's request, then stored in a variable (line 2). Then we generate a new date with respectively the current month, day, and year (line 4-6). Then we read the .json file containing all the publications and put the result in a variable name **file** (line 7). Then in that variable, we will be pushing the new publication previously stored (line 8). Finally, we will overwrite the old .json file with the content of the **file** variable (line 9). It is just as simple as that! As previously said in the Chapter 2, store the publications as json object avoid in first instance to reduce the number of queries sent to the database, but also provide a good flexibility in terms of data manipulation.

Having now the comprehensive idea of how the different systems have been implemented, it is now time to see how the users will be sending mails into the firm's mailbox.

4.3.3 Mail Service Development

As mentioned earlier in the Chapter 2, when disccusing about Client – Server communication, to implement the mail service, I am using a third-party mail service to send mails in the company's mailbox. The name of this third-party is **Mailgun**. They service is free for up to 200 mails daily and provides a free sandbox domain name for testing your application.

But how does it work exactly ? To understand it, I would like to illustrate the following piece of code:



```
1 const auth = {
2  auth: {
3    api_key: "***************************",
4    domain: "sandboxb882faeb0f98451aae910d3e70ea6b46.mailgun.org",
5  },};
6 const transporter = nodemailer.createTransport(mailGun(auth));
7  let sendMail = router.post("/receive", function (req, res, next) {
8  let mail = req.body.mail;
9  let name = req.body.name;
10  let phone = req.body.phone;
11  let message = req.body.message;
12  const mailOptions = {
13    from: mail,
14    to: "willyheisen67@gmail.com",
15    subject: "Client message",
16    text: "Mail: " + mail + "\n" + "Name: " + name +
17"\n" + "Phone: " +  phone + "\n" +  "Message: " +  message,  };
18
19  transporter.sendMail(mailOptions, function (err, data) {
20    if (err) {
21      console.log("ERROR", err);
22      res.status(400).send(err);
23    } else {
24      res.status(200).send("OK");
25      console.log("SUCCESS");
26    }
27  });
});
```

Figure 4-16 Send mail code (back-end)

To send a mail, we will need an **auth** object and a node module called **nodemailer** [10] that allows easy mail sending. The link to the official website will be put in the annex. A transporter is then created which will transport the message using SMTP. This is where the SMTP is actually used for the first time. The **api_key** line 3 is a unique secret key provided by Mailgun once an account is created. The **domain** name correspond to the server that will be sending the mail using SMTP. In that case, the domain name is just a sandbox for testing the app, the real domain name must be added in the Mailgun account and verified.

So here the explanation how it works:

- First the user sends the contact form to the server with his mail address, his phone number, and his name.
- The server then takes each credential and stores each of them into a variable line 8-11.
- Then an **mailOptions** is created (line 12) including the mail sender who is the **user's mail** sending the message, a recipient that is the company's mailbox, the **subject,** and the **content.** Note that the mail recipient is my own but will be replaced on deployment.
- Finally, the transporter will be sending the mail including the **mailOptions** variable as parameter.

Now that we got the full descriptions of how the application has been implemented including the global design implementation and the services, it is time to conclude this chapter by summarising it.

In this chapter we have seen:

- How the global design has been implemented including the different views.
- How the Client and File management system has been implemented.
- How the post publication system has been implemented.
- How the mail service has been implemented.
- How the authentication system has been implemented

In the next chapter we will see what the products looks like by showing some screen shots. We will then be commenting them.

5 New Website for Lawyer Office

Now you will be interested what the product looks like. In this chapter we will first be looking at the design part, by checking first the full screen size user interface, then the small devices user interface. Secondly we will be looking at what the admin user interface looks like and how the admin is interacting with. We will then be looking at how the clients download their files. Finally, we will be looking at the contact form route ,and see how the user sends a mail and how the mailbox receives it. Before introducing this chapter, it is important to note that the pictures used on the website such as the one in the Title wrap component are under CC0 licence, which means that they are completely free to use.

5.1 Global Design

The first screenshot that I would like to illustrate is the following:

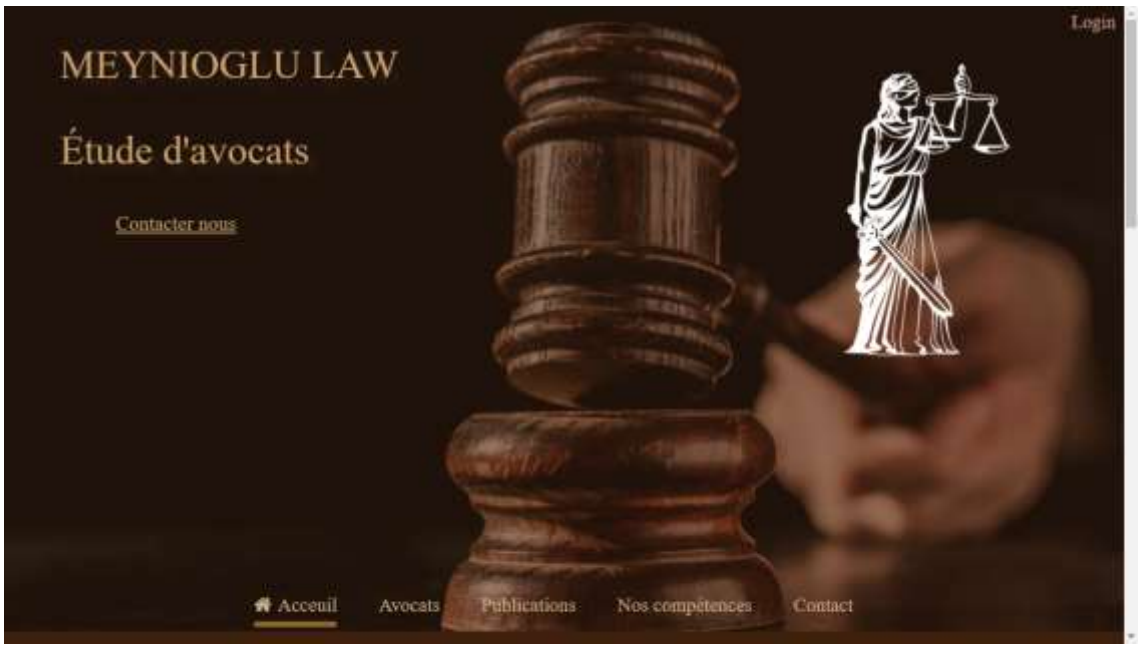


Figure 5-1 First view

The following screenshot shows us what the user will see when clicking on the web address. A full screen presentation. You can see that the screen navigation is from top to left and routes are displayed down this presentation screen. Regarding the component hierarchy described in the Chapter 2, this first screen is in fact the **Title Wrapper** component mentioned.

One of the honourable mention is the contact view, here is the screenshot:

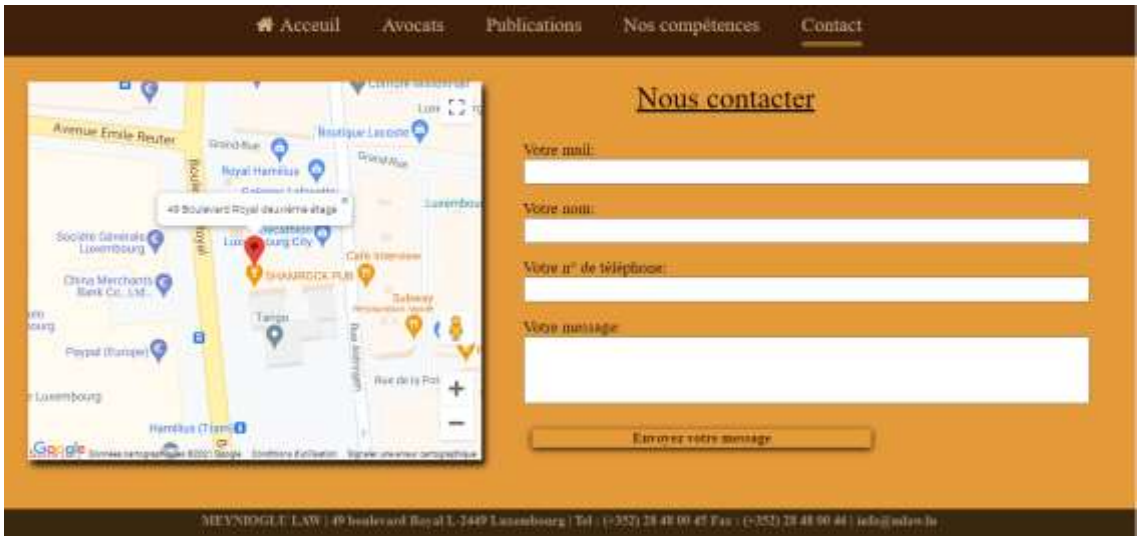


Figure 5-2 Contact page

We got on the left side, the Google map showing the location of the office and on the right-hand side the form to send the data. We will however see the sending message in the next section. For the Google map, I am using my personal account to get the API key to use it. I will probably replace the provider in a near future since many map providers offer free services. Google was just easier to implement (since angular itself is also build by Google).

One last screen I would like to show regarding the global design is the Home (Acceuil) view:

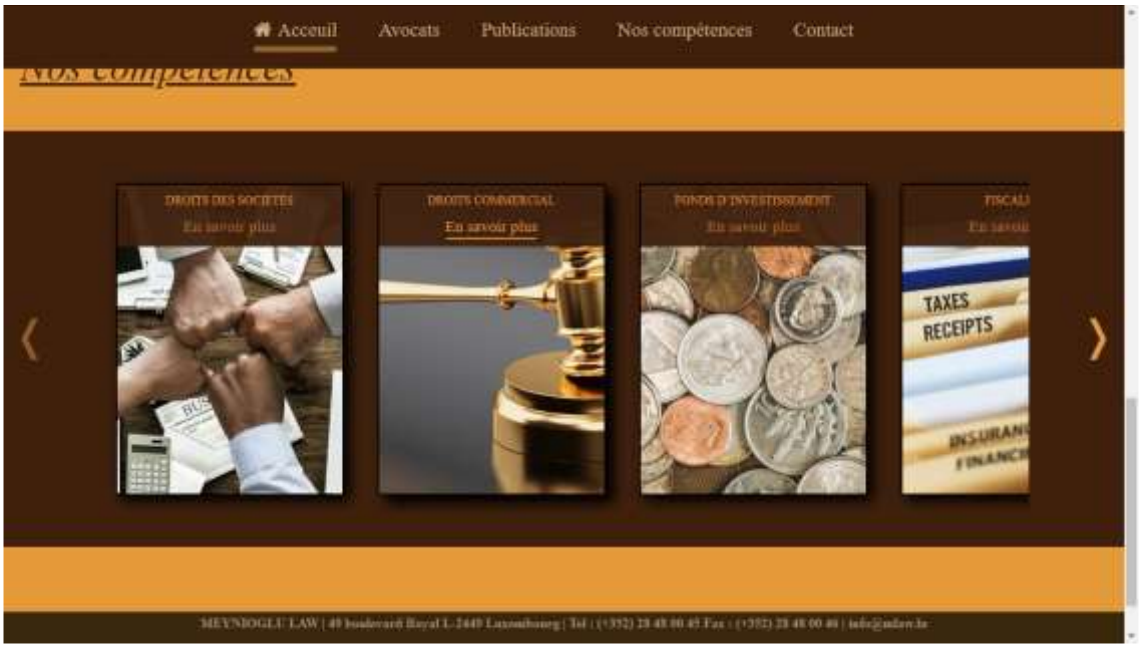


Figure 5-3 Home page

The following screenshot illustrates a list of items inside a wrapper that provides the route link to each specific skill description.

Now that we have seen what the design looks like on the full screen size, let us have a look at the small devices. Since at the end, it was one of the requested requirements:

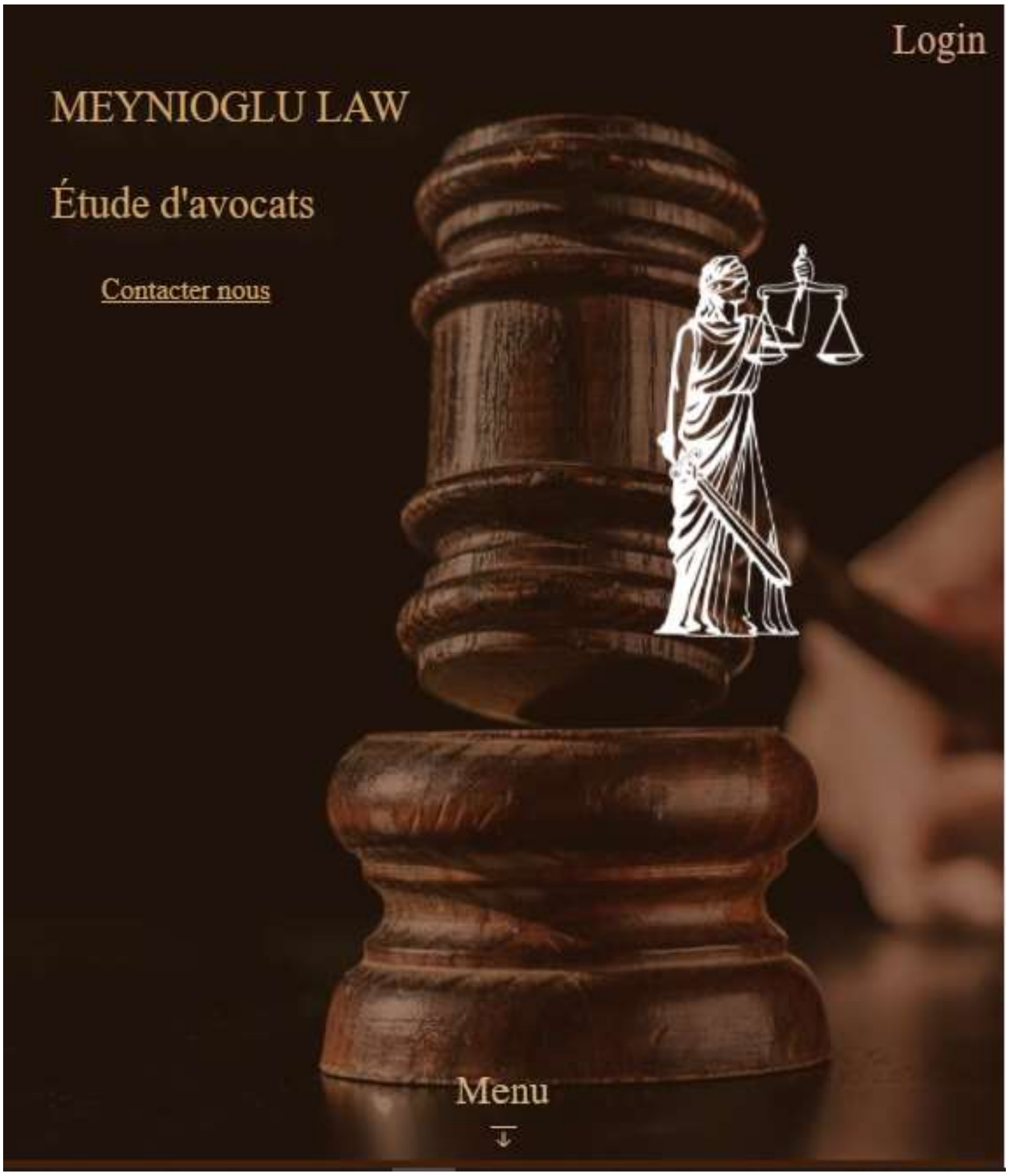


Figure 5-4 First view mobile device

Here is the main page on the phone device. As requested by the institution, the menu bar size has been reduced. Here is what it looks like when displaying the menu:

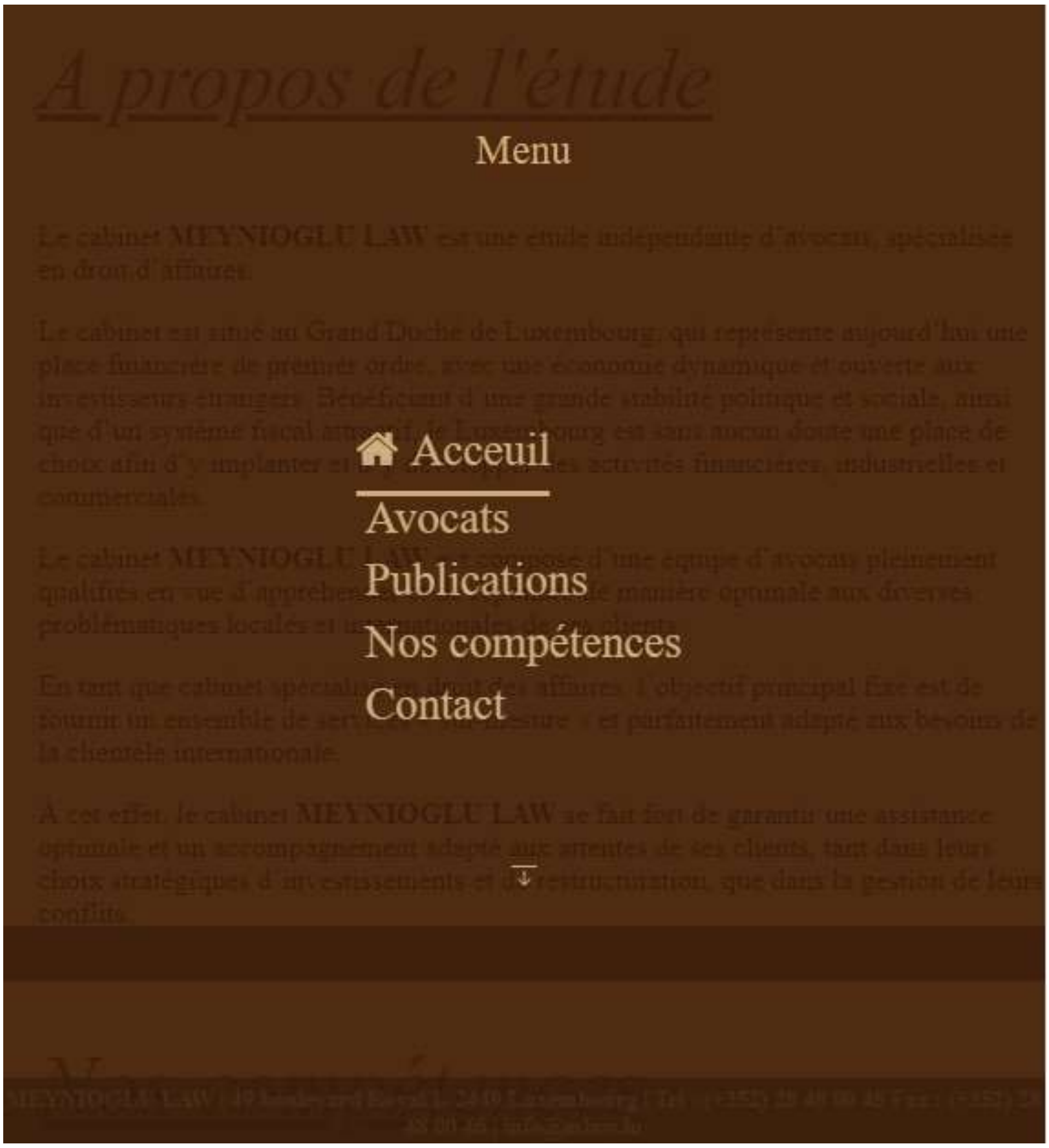
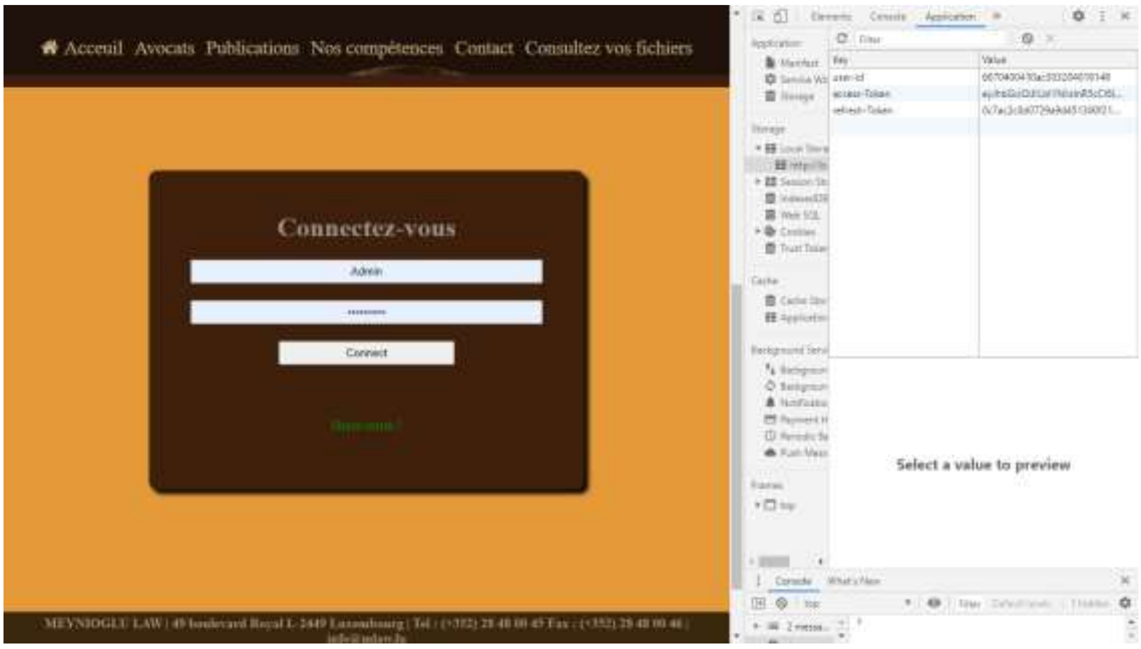


Figure 5-5 Menu mobile device

So far about global design, let us get into the most interesting part: The admin route.

5.2 Admin Route

As we have seen in chapter 2, the admin route is composed of two views. The first view is corresponding to the post publication view, on which the admin performs the create, read, update, and delete operations. The second view corresponds to the clients and files management view. I will keep the best for the end, so let us start with the post-publication view. But before this, the admin must connect! Here is the screen that shows the login route and the user's local storage after the admin successfully connected:



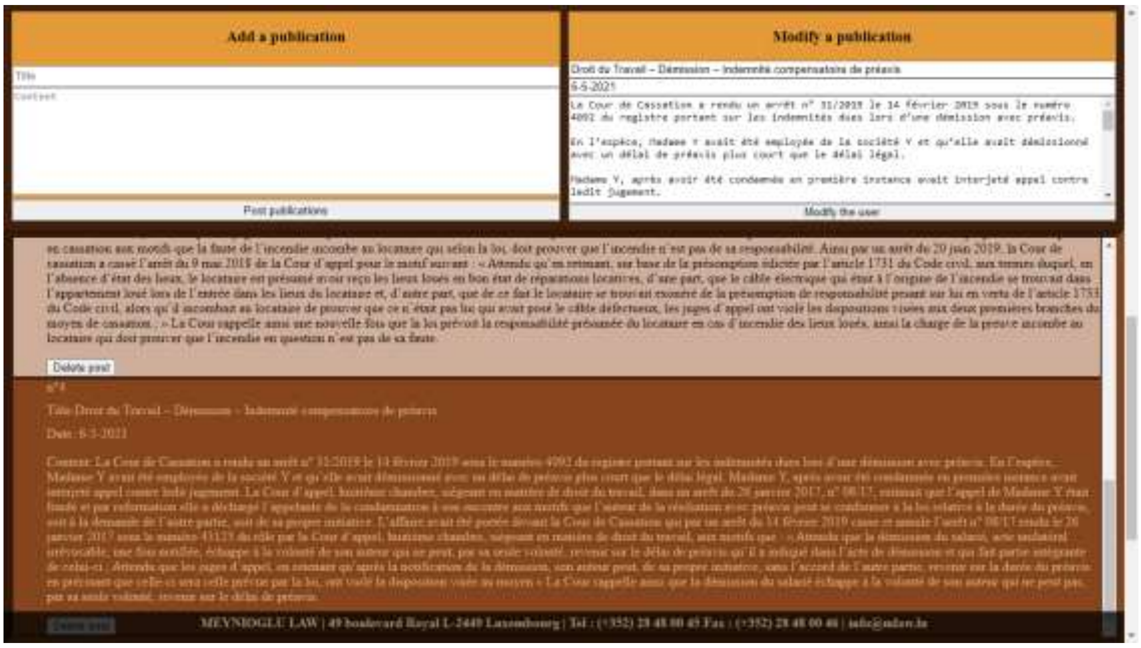
Key	Value
user-id	60704004 [illegible]
access-Token	eyJhbGciOiJ [illegible]
refresh-Token	0c7ac3c8a0729 [illegible]

MEYNIOGLU LAW | 49 boulevard Royal L-2449 Luxembourg | Tel : (+352) 28 48 00 45 Fax : (+352) 28 48 00 46 | info@mlaw.lu

Figure 5-6 Login view

Once the user connected, a welcome message is displayed. We see that we got our three essential data that the user needs to keep sending requests. In that case the user is the admin.

Let us now see what publication post route looks like:



Droit du Travail – Démission – Indemnité compensatoire de préavis
6-5-2021
La Cour de Cassation a rendu un arrêt n° 31/2019 le 14 février 2019 sous le numéro 4092 du registre portant sur les indemnités dues lors d'une démission avec préavis.

En l'espèce, Madame Y avait été employée de la société Y et qu'elle avait démissionné avec un délai de préavis plus court que le délai légal.

Madame Y, après avoir été condamnée en première instance avait interjeté appel contre ledit jugement.

en cassation aux motifs que la faute de l'incendie incombe au locataire qui selon la loi, doit prouver que l'incendie n'est pas de sa responsabilité. Ainsi par un arrêt du 20 juin 2019, la Cour de cassation a cassé l'arrêt du 9 mai 2018 de la Cour d'appel pour le motif suivant : « Attendu qu'en retenant, sur base de la présomption édictée par l'article 1731 du Code civil, aux termes duquel, en l'absence d'état des lieux, le locataire est présumé avoir reçu les lieux loués en bon état de réparations locatives, d'une part, que le câble électrique qui était à l'origine de l'incendie se trouvait dans l'appartement loué lors de l'entrée dans les lieux du locataire et, d'autre part, que de ce fait le locataire se trouvait exonéré de la présomption de responsabilité pesant sur lui en vertu de l'article 1733 du Code civil, alors qu'il incombait au locataire de prouver que ce n'était pas lui qui avait posé le câble défectueux, les juges d'appel ont violé les dispositions visées aux deux premières branches du moyen de cassation. » La Cour rappelle ainsi une nouvelle fois que la loi prévoit la responsabilité présumée du locataire en cas d'incendie des lieux loués, ainsi la charge de la preuve incombe au locataire qui doit prouver que l'incendie en question n'est pas de sa faute.

n°4

Title: Droit du Travail – Démission – Indemnité compensatoire de préavis

Date: 6-5-2021

Content: La Cour de Cassation a rendu un arrêt n° 31/2019 le 14 février 2019 sous le numéro 4092 du registre portant sur les indemnités dues lors d'une démission avec préavis. En l'espèce, Madame Y avait été employée de la société Y et qu'elle avait démissionné avec un délai de préavis plus court que le délai légal. Madame Y, après avoir été condamnée en première instance avait interjeté appel contre ledit jugement. La Cour d'appel, huitième chambre, siégeant en matière de droit du travail, dans un arrêt du 26 janvier 2017, n° 08/17, estimant que l'appel de Madame Y était fondé et par réformation elle a déchargé l'appelante de la condamnation à son encontre aux motifs que l'auteur de la résiliation avec préavis peut se conformer à la loi relative à la durée du préavis, soit à la demande de l'autre partie, soit de sa propre initiative. L'affaire avait été portée devant la Cour de Cassation qui par un arrêt du 14 février 2019 casse et annule l'arrêt n° 08/17 rendu le 26 janvier 2017 sous le numéro 43123 du rôle par la Cour d'appel, huitième chambre, siégeant en matière de droit du travail, aux motifs que : « Attendu que la démission du salarié, acte unilatéral irrévocable, une fois notifiée, échappe à la volonté de son auteur qui ne peut, par sa seule volonté, revenir sur le délai de préavis qu'il a indiqué dans l'acte de démission et qui fait partie intégrante de celui-ci ; Attendu que les juges d'appel, en retenant qu'après la notification de la démission, son auteur peut, de sa propre initiative, sans l'accord de l'autre partie, revenir sur la durée du préavis en précisant que celle-ci sera celle prévue par la loi, ont violé la disposition visée au moyen » La Cour rappelle ainsi que la démission du salarié échappe à la volonté de son auteur qui ne peut pas, par sa seule volonté, revenir sur le délai de préavis.

MEYNIOGLU LAW | 49 boulevard Royal L-2449 Luxembourg | Tel : (+352) 28 48 00 45 Fax : (+352) 28 48 00 46 | info@mlaw.lu

Figure 5-7 Post publication view

On the screen shot we see that the selected publication (brown one) is set to be modified on the top right wrapper. This means that the admin has just to modify the characters in the input fields and then pressing the “modify the publication” button to modify it. In the next step we see the client and files management view, here is the screenshot:

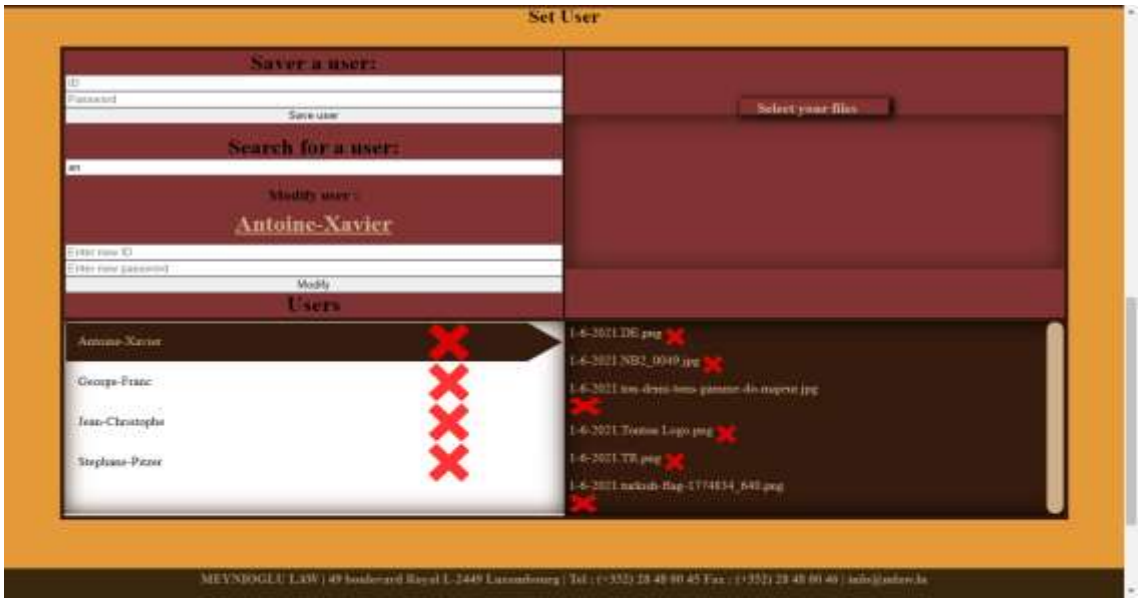


Figure 5-8 Client and File management view

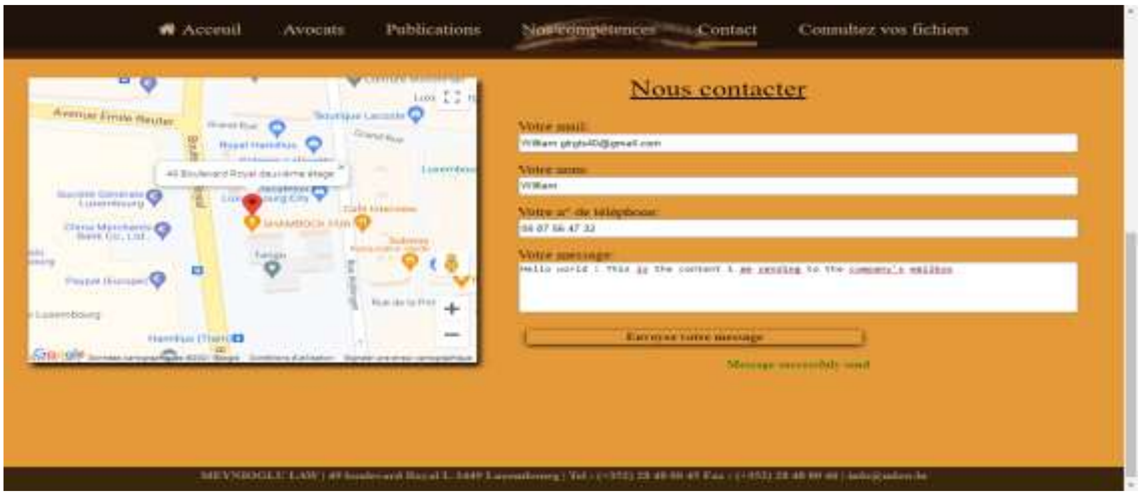
So, in this view we got a table. This table is split into two section:

- The left section that concerns the client management system.
- The right section that concerns the file management system.

On the bottom left, the list of user is displayed and, on the bottom right the list of its files is displayed. When a user is selected, it is ready to be modified and gets his files being uploaded. You can see that the list of user displayed is the one corresponding to the list of characters entered in the “Search for a user” input field, this is where the regex is used.

5.3 Mail Service

I would like to show now how the mails are displayed when the user sends the form. Here is a screenshot taken right after having sent the form:



Nous contacter

Votre mail: William.girgis40@gmail.com

Votre nom: William

Votre n° de téléphone: 06 87 56 47 32

Votre message: Hello world ! This is the content I am sending to the company's mailbox

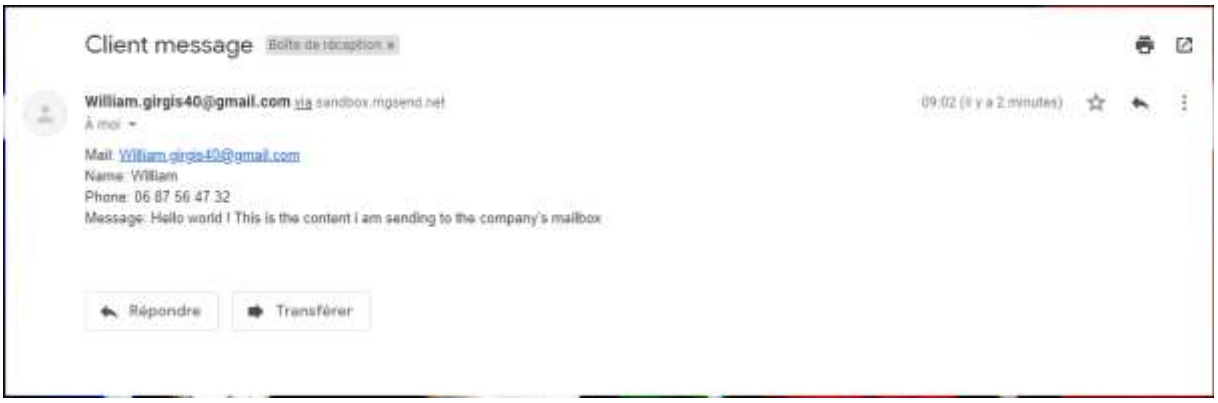
Envoyer votre message

Message successfully sent

MEYNSOGLU LAW | 49 boulevard Royal L-2449 Luxembourg | Tel : (+352) 28 48 00 45 Fax : (+352) 28 48 00 46 | info@mlaw.lu

Figure 5-9 Mail sending illustration 1/2

Now here is the received message:


Client message Boîte de réception x

William.girgis40@gmail.com via sandbox.mgsend.net 09:02 (il y a 2 minutes)

À moi

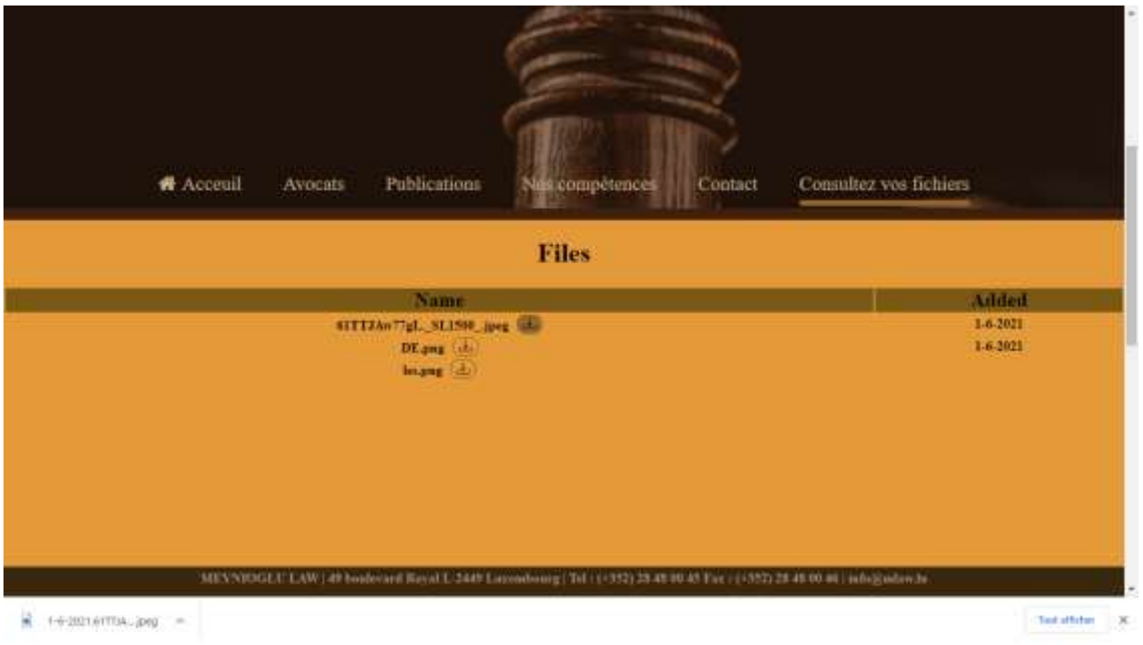
Mail: William.girgis40@gmail.com
Name: William
Phone: 06 87 56 47 32
Message: Hello world ! This is the content I am sending to the company's mailbox

Figure 5-10 Mail sending illustration 2/2

As you can see, this works perfectly fine! The mail contents includes: the mail entered (even though we already get it as the sender address), the client's name, the client's phone number and finally the client's message. Of course, the destination address will be changed by the company's one.

5.4 Client File Route

Let us now conclude this chapter by illustrating the scenario where the client downloads his files. The following screenshot illustrates the client file view, and has been taken right after having clicked on the first download button:



Files

Name	Added
61TT2An77gL._SL1500_.jpeg	1-6-2021
DE.png	1-6-2021
lei.png	

MEYNIOGLU LAW | 49 boulevard Royal L-2449 Luxembourg | Tel : (+352) 28 48 00 45 Fax : (+352) 28 48 00 46 | info@mlaw.lu

1-6-2021.61TTJA....jpeg

Figure 5-11 Client file view

Two sections are shown on the view: on the left we got the file’s name, and on the right, we got the date on which it has been uploaded. We can note that the actual name of the file downloaded, is the current date plus the original file name.

Having now seen the most important parts of the product we can get into the conclusion chapter and compare the product with the initial specifications and discussing about the several issues that must (or at least might) be fixed. Finally, I will conclude by giving my personal experience on the project.

6 Conclusion

Before concluding this paper, you must know that I will keep working on the website until I will have fixed the few problems regarding the design. These problems will be quotes in this chapter.

6.1 Comparison with the Initial Specifications

The first question that comes up is: are the initial specifications stated in the introduction fulfilled ? To answer this, we must first quote them back. Remember, the company needed a new (modern) web site that must include:

- A post publication system (as the old web site).
- A contact form and its mail service (as the old web site).
- A client and file management system (even though it will be deleted).
- A responsive behavior for small devices.

As described in the previous chapter, all the services requested by the host institution are working fine and the website is fully responsive on small devices. We can conclude that the specifications are fulfilled. However, there are still some deployment issues that we are going to illustrate right now.

6.2 Deployment Issues

The first deployment issue regarding the final product, is the domain name used for sending the mail to the company's mailbox. Indeed, and as I already said, the current domain name is a free sandbox provided by my mail service provider and must be replaced with the company's domain name.

The second deployment issue, even though it is a small one, is the Google map API in the contact view. Why is that an issue you might think? Simply because it adds extra costs. Indeed, the current API key used comes from my personal Google account, and therefore I will be deleting it and replace it by a free provider before deploying it.

Another major deployment issue is the translation of the website in three languages. Indeed, the old website contains a feature that I was not able to add in the new website which the translation into Turkish, German, and English languages. The reason is that it requires a lot of work, and this could not be done in the time frame given for the project. Plus, with the institution, we agreed on not implementing it when the specifications have been listed.

The last deployment issue that I already mentioned concerns the security aspect. The files stored in the webserver are in fact too sensitive and that is the reason why the host institution decided not to keep it.

Now let us think on how we could improve the product in the future.

6.3 Potential Improvement

The first improvement that might be done concerns the design globally but mostly, the post publication admin's view. Even though it is working perfectly fine, the system is not intuitive and must be improved. As second potential improvement, I would quote the languages translation. Adding the languages from the old website would considerably improve the website quality. Another improvement could be the adding of a search input on the publication's view to ease the user publication finding. As a last improvement, I would quote the client and file system. As previously said, these data are too sensitive but however we could fix that issue, like for instance by renting a file server and secure it, this service might be kept. But still, this would add considerable extra costs.

To conclude, I will present my experience in the company.

6.4 Personal Experience

My general experience in the company was good and really enriching. I learned a lot during my practical work. The company allowed to implement the services with a maximum flexibility. I learned a lot regarding the organisation of a full-stack project and in addition, all the research done gave me a considerable amount of knowledge especially of the web token authentication system.

However, as I have already mentioned when introducing this thesis, I have been working on my own for the whole project. This has some advantages as I have just described, but also some disadvantages. The fact that there was nobody for monitoring me during the code production made it hard for me to find the solutions to the given problems.

Bibliography

[1] Wikipedia, «Wikipedia RESTful definition,» [En ligne]. Available: https://fr.wikipedia.org/wiki/Representational_state_transfer.

[2] W. Gichuhi, «section.io Cookie-based vs token-based authentication,» [En ligne]. Available: https://www.section.io/engineering-education/cookie-vs-token-authentication/.

[3] KirstenS, «CSRF definition,» [En ligne]. Available: https://owasp.org/www-community/attacks/csrf.

[4] jwt, «jwt introduction,» [En ligne]. Available: https://jwt.io/introduction.

[5] Devstackr, «task-manager-mean-stack,» [En ligne]. Available: https://github.com/Devstackr/task-manager-mean-stack/blob/master/api/db/models/user.model.js.

[6] Devstackr, «YouTube,» [En ligne]. Available: https://www.youtube.com/channel/UCbwsS1m4Hib6R-9F1alus_A.

[7] Tutorial points, «Tutorials point MVVM Tutorial,» [En ligne]. Available: https://www.tutorialspoint.com/mvvm/index.htm.

[8] Google, «Angular canActive,» [En ligne]. Available: https://angular.io/api/router/CanActivate.

[9] Mongo, «Mongo Schema,» [En ligne]. Available: https://mongoosejs.com/docs/guide.html.

[10] Nodemailer, «Nodemailer about,» [En ligne]. Available: https://nodemailer.com/about/.

[11] Cambrige dictionnary, «SEO,» [En ligne]. Available: https://dictionary.cambridge.org/fr/dictionnaire/anglais/seo.

[12] R. Raghuwanshi, «Dzone advantages of JWT,» [En ligne]. Available: https://dzone.com/articles/jwtjson-web-tokens-are-better-than-session-cookies.

[13] npm, «npm multer,» [En ligne]. Available: https://www.npmjs.com/package/multer.

[14] TechTerms, «Techterms.com regular expression,» [En ligne]. Available: https://techterms.com/definition/regular_expression.

[15] Google, «Angular built-in directive,» [En ligne]. Available: https://angular.io/guide/built-in-directives.

Table of figures