POLITECHNIKA KRAKOWSKA im. T. Kościuszki
Wydział Mechaniczny
Instytut

Kierunek studiów: Inżynieria przemysłowa

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

José Moliner Galbis

Opracowanie strony internetowej do udostępniania prac CAD

Development of a webpage to share CAD works

Promotor:
dr inż. Paweł Lempa

Kraków, rok akad. 2019/2020

Autor pracy: ……………………………………

Nr pracy: ……………………………………

OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa magisterska/inżynierska*) została napisana przeze mnie samodzielnie. Jednocześnie oświadczam, że ww. praca:

1) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2017 r. poz. 880 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am* w sposób niedozwolony,
2) nie była wcześniej podstawą żadnej innej procedury związanej z nadawaniem tytułów zawodowych, stopni lub tytułów naukowych.

Jednocześnie wyrażam zgodę na:

1) poddanie mojej pracy kontroli za pomocą systemu antyplagiatowego oraz na umieszczenie tekstu pracy w bazie danych uczelni, w celu ochrony go przed nieuprawnionym wykorzystaniem. Oświadczam, że zostałem/am* poinformowany/a* i wyrażam zgodę, by system antyplagiatowy porównywał tekst mojej pracy z tekstem innych prac znajdujących się w bazie danych uczelni, z tekstami dostępnymi w zasobach światowego Internetu oraz z bazą porównawczą systemu antyplagiatowego,
2) to, aby moja praca pozostała w bazie danych uczelni przez okres, który uczelnia uzna za stosowny. Oświadczam, że zostałem poinformowany i wyrażam zgodę, że tekst mojej pracy stanie się elementem porównawczej bazy danych uczelni, która będzie wykorzystywana, w tym także udostępniana innym podmiotom, na zasadach określonych przez uczelnię, w celu dokonywania kontroli antyplagiatowej prac dyplomowych/doktorskich, a także innych tekstów, które powstaną w przyszłości.

Jednocześnie przyjmuję do wiadomości, że w przypadku stwierdzenia popełnienia przeze mnie czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzej pracy, lub ustalenia naukowego, właściwy organ stwierdzi nieważność postępowania w sprawie nadania mi tytułu zawodowego (art. 193 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym, Dz.U. z 2017 r., poz. 2183, z późn. zm.).

…………………………………………..

podpis

3) Wyrażam zgodę na udostępnianie mojej pracy dyplomowej w Archiwum PK do celów naukowo-badawczych z poszanowaniem przepisów ustawy o prawie autorskim i prawach pokrewnych (Dz.U. z 2017 r. poz. 880 z późn. zm.).

TAK/NIE*

…………………………………………..

podpis

Uzgodniona ocena pracy: ………………………………………………………………..

……………………………… ....

podpis promotora podpis recenzenta

* niepotrzebne skreślić

TABLE OF CONTENTS

1. PURPOSE AND SCOPE OF WORK 6

2. INTRODUCTION 7

3. HOW THE WEB APPLICATION WORKS 10

3.1. How does the Internet works? ... 10

3.2. What is a web server? ... 12

4. METHODOLOGY 13

4.1. Server-side web frameworks ... 14

4.2. Database system ... 16

4.3. Template processor ... 17

4.4. Cloud computing ... 18

5. HOW IS THE WEB APPLICATION ORGANIZED? 20

5.1. Functional and non-functional requirements ... 20

5.2. Class diagram ... 21

5.3. Use case ... 22

5.4. Activity diagrams ... 30

5.5. Database diagram ... 37

6. APPLICATION SOURCE CODE 39

6.1. Directory structure ... 39

6.2. package.json ... 40

6.3. app.js ... 41

6.4. WWW Server File ... 45

6.5. database.js ... 46

6.6. Routes ... 47

6.7. Controllers ... 49

6.8. Views ... 59

6.9. Public ... 63

6.10. Helpers ... 66

7. FUTURE WORK AND CONCLUSIONS 74

REFERENCES 75

SUMMARY IN POLISH 78

LIST OF FIGURES 79

CODE LISTING 81

1. Purpose and scope of work

This project will consist of the development of a website with basic functionality, such as registration and login of different users, upload and download of 3D objects files with their corresponding extension, and file searching and filtering system. It will also include features such as a file rating scale system or storage of customer statistics to improve the user experience.

This web page, as previously mentioned, will be oriented to the upload of downloadable files that can be understood and compiled by 3D printers, so that they are capable of replicating aforementioned designs, creating volumetric parts or models from an external file made by computer software.

By doing a little browsing on the Internet, we can already find applications that meet these requirements [1]. However, most of them are pages developed in English and highly globalized. Therefore, it would probably not be very difficult to identify a sector (either geographical or productive) in which a market niche can be found, where the application could focus its attention. Anyway, this is not the purpose of this thesis and the system will not be focused on a specific environment, but this was considered an important aspect to mention.

On the other hand, the major purpose of the project is to understand the foundations of web development, one of the most attractive and demanded topics today, since it is in constant evolution and it has a great future projection. And to achieve this aim, the work has been divided into four main steps.

As an industrial engineer student, the acquired knowledge regarding web development is quite limited, being almost inexistent. This has been one of the principal reasons which encourage the beginning of this project. In a world where web applications dominate our lives, a minimum understanding of their functioning is necessary.

In the first chapters there will be carried out a slight (and humble) study about the behaviour of these applications. The objective will be to understand how it is possible that an ordered series of characters created by a developer, can be translated to some information requested by a client from any part of the world.

The second part will be based on a small study about the different existing tools used to develop a project like this. To do so, some of these tools will be analysed, as well as explained the reasons for their preferred usage against other types.

In the third one, an attempt will be made to explain the relationships between the different parts of the application, in addition to specifying how and why each of the possible actions intended by the client works.

Finally, in the fourth step, a great part of the written code will be analysed. It will be the most extensive and, probably, the most interesting part of the thesis, in order to understand the operation of the web application.

2. Introduction

This project is based on a design-publishing platform. It is a good way to share pieces, figures, or copies of other designs to be able to print oneself at home and it is completely open. This means that all the designs that are uploaded are accessible to the public without any restriction.

To upload models to this platform, users must register and, then, they will be allowed to provide physical printable solutions to many daily problems. For example, if they need a spare part of an appliance or, furthermore, it does not exist, they can design and upload it, and they will help many people with the same problem.

Models can be uploaded in two different ways. There are 3D format models that are widely used in design and render programs, but other formats can also be found (like the '*.stl*'). Therefore, there can be found either files which are prepared to print directly or files that are modifiable with some design software to be able to use them later to print them as a mock-up.

In short, this platform is a good way to obtain what we are looking for if we have a 3D printer. Let us see its basic functionality.

To start, when visiting the web page for the first time, we are faced with a screen like the one shown in figure 2.1.

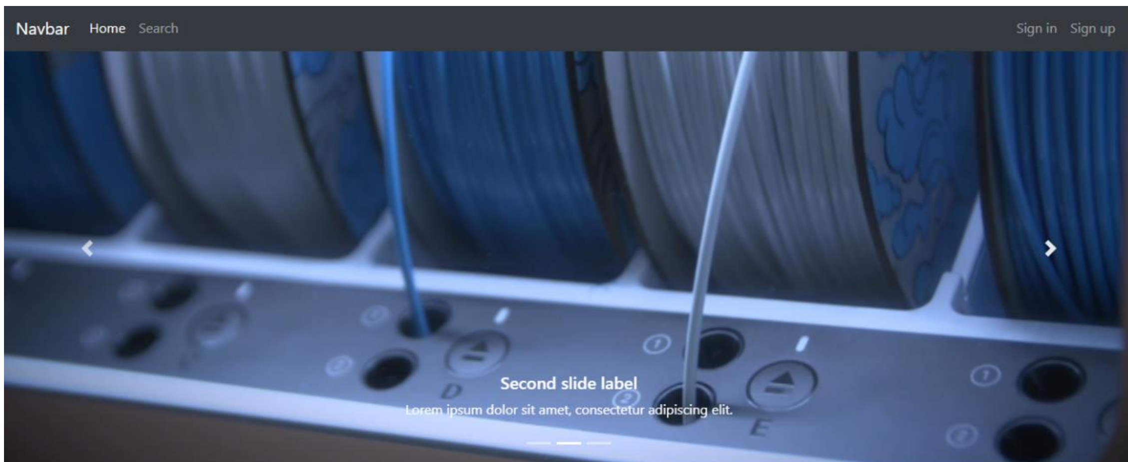

Fig. 2.1. Home page of web application [1]

As it can be noticed, at the top of the web page there is a navigation bar. This bar will be found in each one of the pages of the application. In the left corner, there can be seen the links that will lead to both the home page (the current one) and the search page. On the right, there are the links for registering as a new user and to log in.

Apart from this, on the home page will be found a series of images and texts that would offer a small welcome and description of the website.

The next step that can be taken before logging in, is to visit the *search* page. On it, can be queried the database to find a specific file or to search for one that meets specific requirements. However, the web will not allow to download them unless the user log in, so, without being registered, the website will provide with very little useful services. Figure 2.2 reflects how it looks.

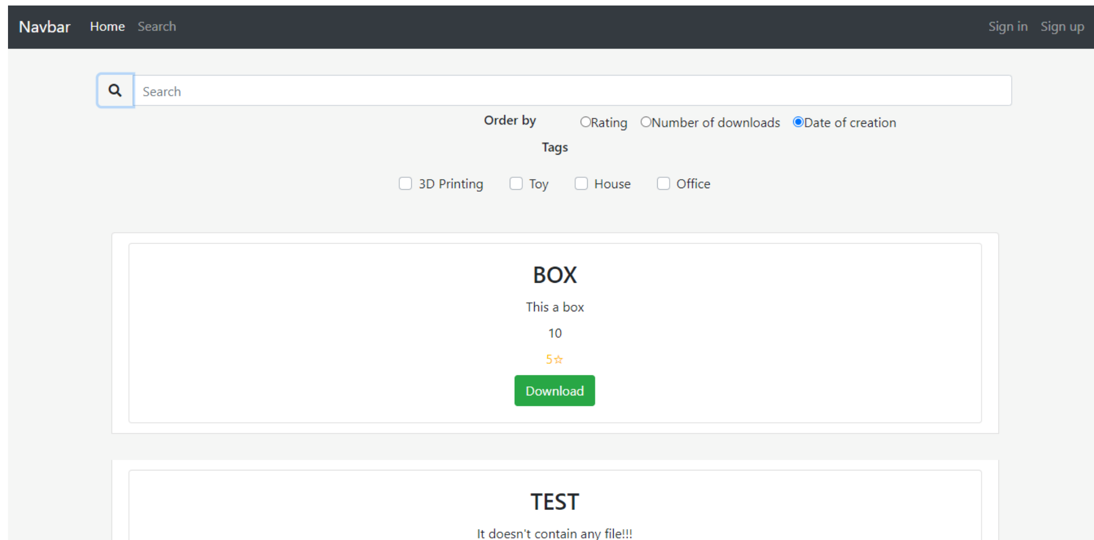

Fig. 2.2. Search page of web application [2]

In this case, a simple query has been made, without any special parameters required, to show all the results stored in the database.

However, as previously mentioned, the website has been designed with the intention of being able to interact with the files uploaded on it: upload solutions to problems, download them, rate them according to whether they fulfil their function or not ... For all this, the user must open an account, *clicking* on *'sign up'*, which will redirect them to a page with a box like the one in figure 2.3.

Sign Up
Name
Surname
Birthday
dd/mm/aaaa
Country
Afghanistan
Username
E-mail
Password
Sign up

Fig. 2.3. Sign up box of web application [3]

On it, the user must fill in all the fields and, once finished, *click* the button. However, this will not be over, since when registering, he will be sent a message to his personal email with a link that will allow him to activate his account. Once activated, all the functionalities that the website offers will be available.

At this moment, apart from being able to download 3D files from the system, the client will be able to upload their own designs, rate those of others, or modify their account data, in addition to receiving welcome points to buy files. Above, figure 2.4 shows a user's page and, figure 2.5, the details box of a file.

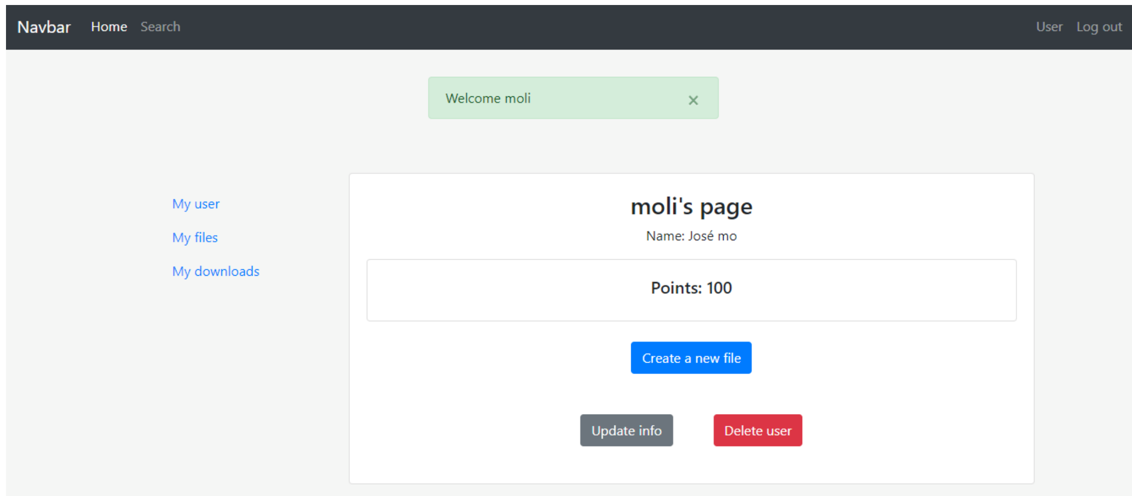

Fig. 2.4. User's page of web application [4]

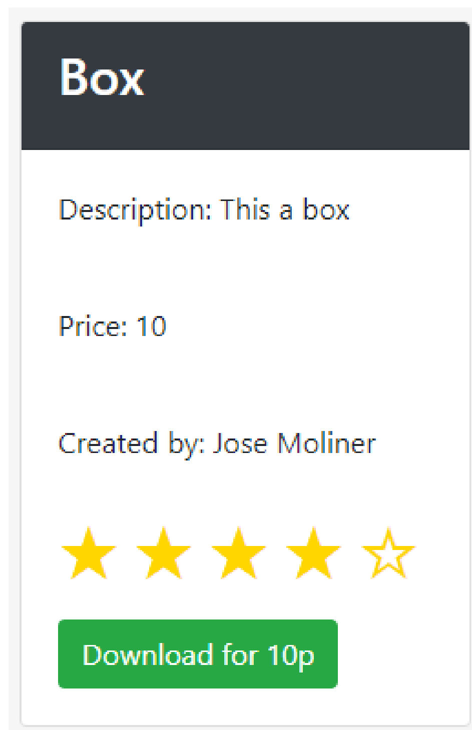

Fig. 2.5. File detail box of web application [5]

3. How the web application works

The Internet is the backbone of the Web, the technical infrastructure that makes it possible. At its most basic, the Internet is a large network of computers that communicate simultaneously.

The history of the internet started in the 1960s as a research project funded by the US Army, and then became a public infrastructure in the 1980s with the support of many public universities and private companies. The different technologies that the internet supports have evolved over time, but the way it works has not changed much: the Internet is a way to connect computers to each other and ensure that, whatever happens, they find a way to stay connected. [2]

3.1. How does the Internet works?

Most of us know how to use the Internet without actually understanding how it works. Sort of like electricity in our home, we use it every day but may not understand the mechanics behind it.

Whenever most people think of the Internet, something like a bubble cloud comes to mind. However, the Internet is more likely a wire buried in the ground. There might be fibre optics, copper or occasionally beamed to satellites or through cell phone networks. But, basically, the Internet is simply a wire that connects two computers directly, as shown in figure 3.1.1.

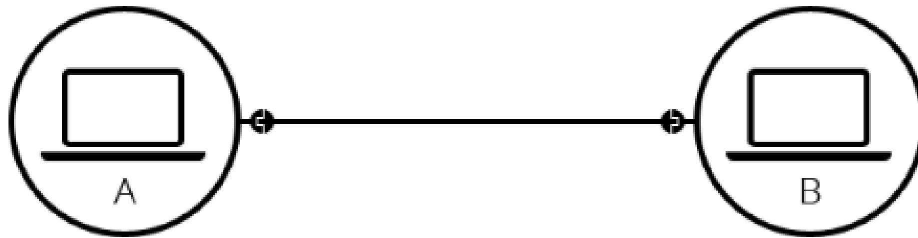

Fig. 3.1.1. Computers are connected by a wire [6]

Practically every time we make an Internet query and receive a response, the data that accompanies it travels thousands of kilometres to reach our computer or our mobile phone. This happens because this data is stored in a data centre.

A solution to send this data could be the following: sending a signal to a satellite through an antenna and then, from the satellite, another signal is sent to the device through another antenna close to it. However, this way of sending information would have a great latency, since until reaching the satellites, there is a great distance. This would make it unusable for most applications that it currently has.

That is why the help of a network of these cables tis needed. Ultimately, to not always depend on cables, they can be connected remotely, once the information is near, through antennas (but the large distance always will be travelled by these wires).

Previously it was said that all the information is in a data centre. To be more specific, it is located in a solid-state device in a data centre, which acts as the internal memory of a server. The server is simply a more powerful computer whose job is to provide data when requested.

But how can some information be transmitted between a server and, for example, a specific mobile phone? Each device that is connected to the internet is uniquely identified by a string of numbers known as an IP address. Thanks to this, can be ensured that the information will reach the correct destination.

Nonetheless, this series of numbers is difficult for people to remember, so domain names associated with these addresses are used, which are easier to remember. These domain names will be stored together with their corresponding IP addresses on the DNS server.

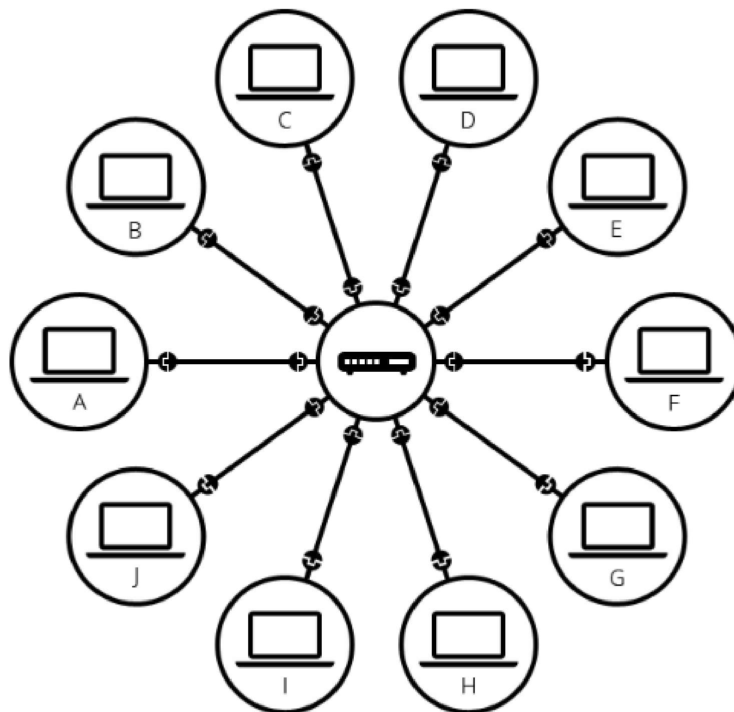

Fig. 3.1.2. Computers are connected together [7]

Let us make a summary of the whole operation. When entering the domain name into a browser, it sends a request to the DNS server to obtain the corresponding IP address. After obtaining it, the browser simply forwards the request to the data centre (more specifically, to the corresponding server). Once the server receives a request to access a particular website, the data flow starts. Data is transmitted in digital format via fibre optic cables, more concretely in form of light pulses. These pulses of light sometimes must travel great distances through wires to reach their destination where they are connected to a router. The router converts these light signals into electrical signals and an ethernet cable is used to transmit electrical signals to the final receiver. However, if the device is accessing the internet using cellular data from the optical cable, the signal must be sent to a cell tower and from the cell tower the signal reaches the device in form of electromagnetic waves.

3.2. What is a web server?

Regarding software, a web server is the piece of Internet in charge of controlling how users access files. For example, an HTTP server is a piece of software that includes URLs (web addresses) and HTTP (the protocol that your browser uses to view web pages).

When a user performs a search in a browser or directly inserts the name of the domain to which he tries to access, immediately a request is sent. It is accepted by the web server and consequently sends a response to this request (Fig. 3.2.1). Nothing apart from the page he is trying to access is dispatch.

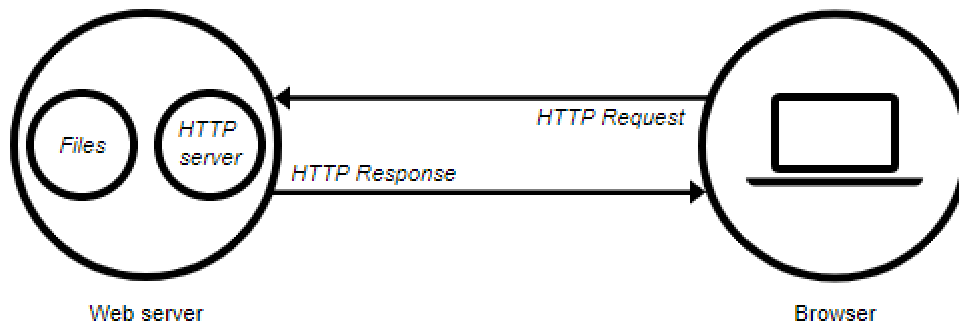

Fig. 3.2.1. HTTP request and response [8]

Both, hardware and software, work together to maintain server functionality.

The hardware side stores all the components of a web page such as image files, programming, CSS design styles…, in short, all the files related to a web page. This hardware is kept connected to the internet and stores data that is exchanged with other media when a request is made.

The software used will allow to maintain control of all the stored files and will allow ingress to users who try to access these hosted files.

The HTTP server will oversee understanding the URLs and the transfer protocol used by the browser where this query is made.

If it does not find the address indicated by the client, the server returns a *404-error* indicating that the content has not been found and sends the request back to the browser from the HTTP protocol.[5]

Apart from this type of servers, we can also find database servers. It is a type of server software that allows the organization of information using tables, indexes and records.

At the hardware level, a database server is a computer team specialized in serving queries to remote or local clients that request information or make modifications to the records and tables that exist within the system's databases (in many cases from a web or application server).

The databases that exist inside, serve to manage and administer immense amounts of information, as happens in cases of companies, institutions, universities or banks, which store user/client data such as addresses, telephone numbers, emails, income…

Therefore, these types of servers are going to be totally necessary for the development of the final application since they fulfil just what we want to achieve. Very briefly, sending responses to clients and storing information.

4. Methodology

As previously mentioned, the *web* works according to a really simple principle: web servers host content and clients request it via HTTP, understood by browsers (Mozilla Firefox or Google Chrome) installed on the system of the user, where they run. As it has been explained before, it acts like it is shown in Figure 4.1.

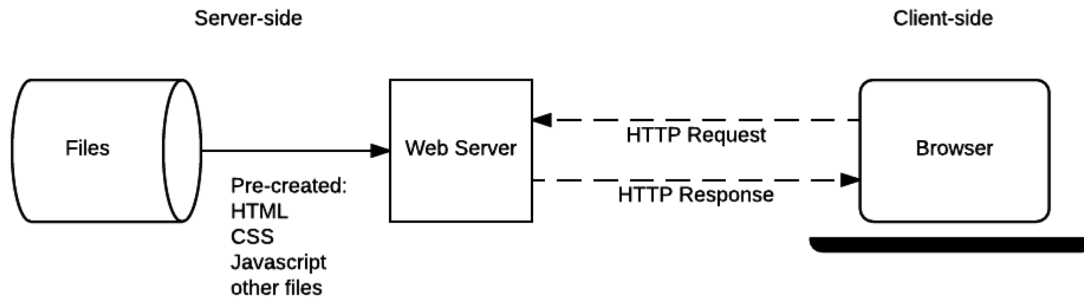

Fig. 4.1. Running of a static web [9]

This is true for a static web, that is, a web that returns the same hard-coded content from the server whenever a particular resource is requested. But this project is not about creating a static web page, but a dynamic one. Server-side programming comes into play in developing web pages with dynamic elements and web applications. This web development technology is based on the use of scripts that are executed by the web server, with the help of an appropriate programming language when a client requests the content. A common scripting task is to extract the needed data from a database and integrate it into the web project.

A scheme much closer to would be the next:

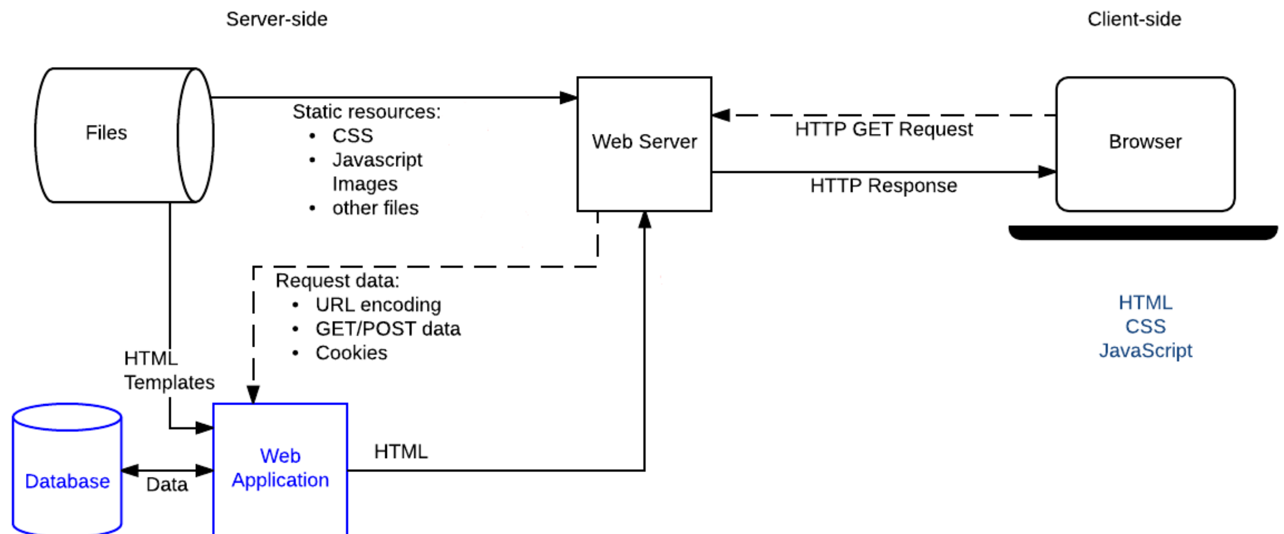

Fig. 4.2. Running of a dynamic web [10]

The main difference observed is that dynamic resource requests are forwarded to server-side code (‘*Web application*’). The server interprets the request, reads the required information from the database, combines the retrieved data with HTML templates, and sends a response that contains the generated HTML(Fig 4.2).

Therefore, it will take a wide variety of different tools to build the final web page.[1]

4.1. Server-side web frameworks

A framework is a set of tools, libraries and good practices to solve a set of problems. In backend development, all frameworks have the main task of creating a web server. The libraries are fragments of code created by third parties that solve a specific problem in order to extend the functionality of the code.

That is, the difference between these is that, while a framework gives you a workspace to develop, a library solves a problem by making their application code easier and more readable.

But... why use a framework or a library? First, these tools can help to have a faster development using good practices and avoiding writing repetitive code. But most importantly, focus the time on solving problems of development, instead of solving problems that other people have already solved.

Therefore, it will be really useful to select one of the existing frameworks. To do this, we should look at the availability and quality of the documentation, the size of the community that uses it, the problems it can solve, its flexibility, its complexity and its compatibility with the other tools used. Some that meet the characteristics we require are:

- **Python**, an interpreted programming language based on the readability of its code. They say it is a multi-paradigm programming language because it supports object orientation, imperative and functional programming.
- **Node.js**, a multiplatform, useful in many different cases. It is open source and it has a runtime environment for the server based on the JavaScript programming language. What is more, it is asynchronous (very important in many cases), and has an event-oriented architecture, based on Google's V8 engine.
- **Php**, which is called Hypertext Pre-processor. It is usually used for server-side code programming and it is actually one of the most important.
- **Java**, a computing environment, capable of executing applications developed using Java language or other languages that have a set of development tools and compile bytecode.
- **Ruby**, a general-purpose language. With Ruby can be developed all kinds of different applications: web service applications, email clients… It is dynamic and flexible language and it is open source and multiplatform framework.

[1] Among other reasons, one of the most important requirements to select them was the learning curve so, finally, practically all chosen tools work with the same language. This greatly facilitated the development stage, since there was not almost any knowledge about the different programming languages before starting it.

After doing a little study between the different environments, we opted for *Node.js*.[6][7][8][9]

Node.js is a JavaScript runtime environment. This real-time runtime environment includes everything needed to run a program written in JavaScript.

It was created by the original JavaScript developers. They transformed it from something that could only be run in the browser to something that could be run on computers as if they were separate applications. Thanks to *Node.js* you can go a step further in programming with JavaScript not only by creating interactive websites, but by having the ability to do things that other script languages like *Python* can create.

Where *Node.js* really shines is in creating fast network applications, as it is capable of handling a large number of simultaneous connections with a high level of performance, which equates to high scalability.

Between its numerous advantages, there is having JavaScript incorporated, being an easy language to learn. Moreover, its event driven model helps a lot in simultaneous handling of requests.

Node.js is the most widely used software platform at the moment, being above in runtime environments and programming languages like PHP and C++.

But, although it is true it is not a very complex code, it requires many more lines of coding and greater understanding than other languages.

Anyway, *Node.js* is one of the most widely used technologies today and has become one of the most popular platforms used for the development of web applications, desktop applications, and services.

Next, within Node.js the Express framework will be used [10]. Espress.js (Fig 4.1.1), according to its creators, is *'a flexible and minimalist web application development framework for Node.js'*. It allows you to create APIs and web applications easily and it provides a set of features such as route management (*routing*), static files, use of template engine, integration with databases, error handling, middleware, etc. among others.

Fig. 4.1.1. Node.js and Express logo [11]

4.2. Database system

It will be necessary for this project to store information belonging to the same context, systematically ordered for subsequent retrieval, analysis and transmission. The database system on which to work for a web project is very important, since each of them has its peculiarities. The most suitable depends on what type of projects are going to be created.

We have many different options to choose from, but the first question that arises is the following: is a SQL or a NoSQL[2] database more convenient?

The fundamental difference between both types of databases is that NoSQL databases do not use the relational model. In this type of database, its scalability and its decentralized nature are the most valuable. They tend to be much more open and flexible databases. In addition, they allow adapting to project needs more easily than SQL models. Another advantage will be their horizontal scalability, that is, they are able to grow in number of machines, instead of having to reside in large machines.

On the other hand, SQL databases have their most adapted use and the profiles that use them are majority and cheaper. Due to their long time on the market, these tools have more support and better product suites to manage. In addition, the data must meet integrity requirements in both data type and compatibility.

Finally, we opted for an SQL database, specifically *MySQL*(Fig. 4.2.1). It is true that there are other types of non-relational databases that, a priori, might seem easier to learn, having a language closer to JavaScript, such as *MongoDB*. But *MySQL* was chosen for two different reasons. Firstly, because it is a much more widely used language and, therefore, answers to possible problems will be found easier. On the other hand, there is the stock of *MySQL Workbench*, a visual database design tool that integrates software development, database administration, database design, management and maintenance for the *MySQL* database system. Thanks to that, designing the database will become a simple task.

Fig. 4.2.1 MySQL logo [12]

[2] *Structured Query Language* and *non Structured Query Language*

4.3. Template processor

When rendering a web page, it is needed some HTML to log information. HTML is static, which means that if we want to display dynamic data (fetched from a database, for example), we invariably end up with a disarray of HTML strings inside JavaScript. This can be very difficult to debug and to maintain. And here is where template engines come in.

A template engine is a program which is responsible for compiling a template (that can be written using any one of several languages) into HTML. The template engine will normally receive data from an external source, which will be injected into the template it is compiling. Its performance is always similar to the one of figure 4.3.1.

This approach allows to reuse static web page elements, while defining dynamic elements based on your data. It also facilitates a separation of concerns, keeping the application logic isolated from the display logic.

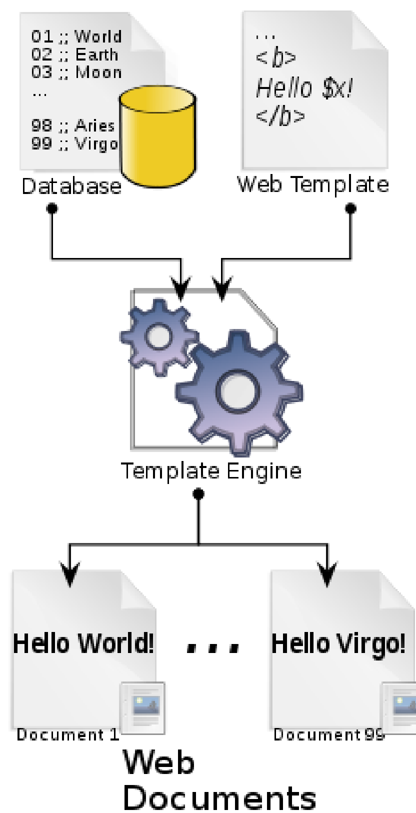

Fig. 4.3.1 Templating engine performance [13]

There are a great number of popular view/templating engines, as `Ejs`, `Hbs`, `Pug`, `Twig` or `Vash`, although Express itself can also support many other templating languages.

Generally speaking, should be selected a templating engine that delivers all the functionality needed and allows to be productive sooner. Some of the things to consider when comparing template engines are:

- **Time to productivity:** to be more productive using that language.
- **Popularity and activity:** review if it has an active community. It is important to be able to get support when problems arise throughout the lifetime of the website.

- **Style:** some template engines use specific mark-up to indicate inserted content within "ordinary" HTML, while others construct the HTML using a different syntax (for example, using indentation and block names).
- **Performance/rendering time.**
- **Features:** should be considered whether the engines have the following features available:

1. *Layout inheritance:* allows to define a base template and then 'inherit' just the parts of it different for a particular page. This is typically a better approach than building templates by including several required components or building a template from scratch each time.
2. *'Include' support:* allows to build up templates by including other templates.
3. *Concise variable and loop control syntax.*
4. *Ability to filter variable values at template level* (e.g. making variables upper-case or formatting a date value).
5. *Ability to generate output formats other than HTML* (e.g. JSON or XML).
6. *Support for asynchronous operations and streaming*.
7. *Client-side features*. If a templating engine can be used on the client this allows the possibility of having all or most of the rendering done client-side.

For this project, will be used the *Pug* templating engine, as this is one of the most popular *Express/JavaScript* templating.

Thus, *Pug* is a template engine for *Node.js* and for the browser. It compiles to HTML and has a simplified syntax, which can make us more productive and our code more readable. *Pug* makes it easy both to write reusable HTML, as well as to render data pulled from a database or API.[11]

Fig. 4.3.2 Pug logo [14]

4.4. Cloud computing

Cloud computing is, according to *Wikipedia*, the availability upon request of the resources of the computer system, especially data storage and some computing capacity, without direct active management by the client. The term is generally used to describe data centres available from anywhere to many users over the Internet from any mobile device. So, if we want other users to access to the web application, a cloud computing service is needed.

Heroku is one of the most widely used PaaS (Platform as a Service)[3] today in business environments due to its strong focus on solving the deployment of an application. It also allows to manage the servers and their configurations, escalation, and administration. In *Heroku*, only it is needed to tell what backend language and database is being used and the developer only has to take care about the development of his application.[12]

And, another time, looking for simplicity and efficiency, this development tool was chosen.

Fig. 4.3.2 Heroku logo [15]

[3] Cloud computing offering that provides users with a cloud environment in which they can develop, manage, and deliver applications.

5. How is the web application organized?

The requirements of any project must be well thought out, balanced and clearly understood by all people involved on it, and perhaps most importantly, they do not have to be discarded or compromised in the middle of the project. For this purpose, some descriptions and diagrams of the final product must be designed before writing any code.

In one hand, we have the functional and non-functional requirements. A functional requirement is the one that essentially specifies something the system must do, and the non-functional requirements describe how the system works. It essentially specifies how the system should behave, that is, the constraints on the behaviour of the system and the quality attributes it has.[13][14][15]

On the other hand, UML diagrams were created to forge a common visual language in the software development world that is also understandable by business users and whoever wants to understand a system. In this type of diagrams, the components of web projects are defined and exemplified, in addition to their internal structure. That is why they will be used below to explain what the created project consists of. [16][17]

5.1. Functional and non-functional requirements

5.1.1. *Functional Requirements*

Functional process requirements

- The website will have 3 main pages: Home page, User page and Search page.
- In addition, there will be one page for the login of users. The other 3 pages will be inaccessible until the client log in.
- In the user registration, the client must fill in all the required fields to complete the request.
- Once the user is registered, he will be redirected to the home page.
- Once logged in as a user, you should not re-enter your password unless you leave the website.
- From any page you can access the Home page, the Search page, and the User page.
- Each user can upload and download files, rate every file, modify his personal data, delete and modify his uploaded files data. All this data must be filled.
- Each user can track his uploaded files or create new ones from his User page.
- The registration of files with incomplete data will be allowed, which can be completed later by modifying the file.
- The web application will allow the creation, deletion and modification of files.
- The web application will allow the client modification and deletion of his user.
- Home page will consist of a presentation of the web page. It will be a static page.
- Search page first will show the most recent uploaded files.
- Search page will have a box with statistics of the web page.
- The user will be able to filter files by tags or name.
- The user will be able to order files by rating or number of downloads.

Graphical Interface functional requirements

- The name field only accepts alphabetic characters.
- The username will only accept alphanumeric characters.
- The password will only accept alphanumeric characters.
- Dates introduced in the client data will only be accepted in one specific format.
- The country and tags field will consist of a pre-selection list.

Security functional requirements

- The web application will control access and allow it only to authorized users. Users must enter the web application with a username and password.
- Members of the user group can create, modify and delete their files, but not those from other users.
- Any data exchange via the Internet made by the software will be done through the https encrypted protocol.

5.1.2. *Non-functional requirements*

Efficiency

- No efficiency requirements will be asked of the web application, but it must be scalable in order to improve it in the future.

Usability

- The web application must be very intuitive, so that any user can perform any action without prior knowledge.
- The web application must provide error messages that are informative and orientate the users' actions.
- The web application must have a "Responsive" design in order to guarantee an adequate display on multiple personal computers, tablet devices and smartphones.
- The web application should be usable by 98% of Internet users, depending on their browser.
- The navbar should always be accessible.

5.2. Class diagram

It is the most commonly used UML diagram, and the main foundation of any object-oriented solution. Here are located the classes within a system, attributes and operations, and the relationship between each class. Classes are grouped together to create class diagrams when creating large system diagrams.[18]

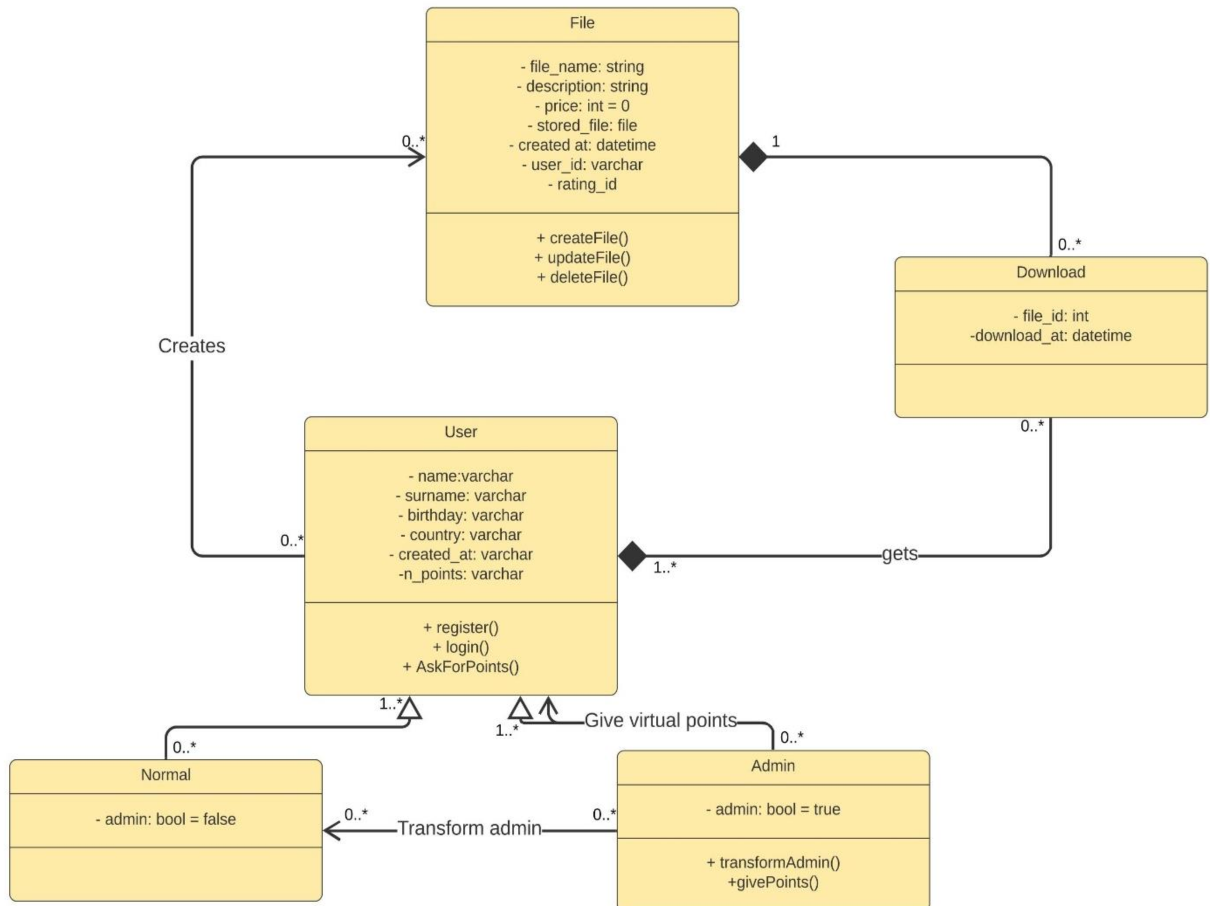

Fig. 4.3.2 Classes diagram [16]

The previous diagram is very self-explanatory. Each of the boxes is divided into three sections, with their names, their properties and their methods, respectively. For example, the *admin* class has a boolean property called *admin* with a value of *true* (in addition to all those that come from the *user* class, since it inherits all of them), and two unique methods: `transformAdmin()` and `givePoints()`.

Thanks to the diagram, we can also check that there are three main classes: *user*, *file* and *download*. The *user* will be in charge of creating files. For the existence of a *download*, there will be needed to be both a *user* who creates it and a *file* that is downloaded.

Then, *users* can be separated into two different ones: *normal users* and *admin*, each with some properties in common and others different.

5.3. Use case

A use case diagram can summarize the details of the system's users (also known as actors) and their interactions with the system. Here, can be located the different scenarios in which the system or application interacts with people, organizations, or external systems, the goals that the system or application helps those entities (actors) achieve and the scope of the application.[19]

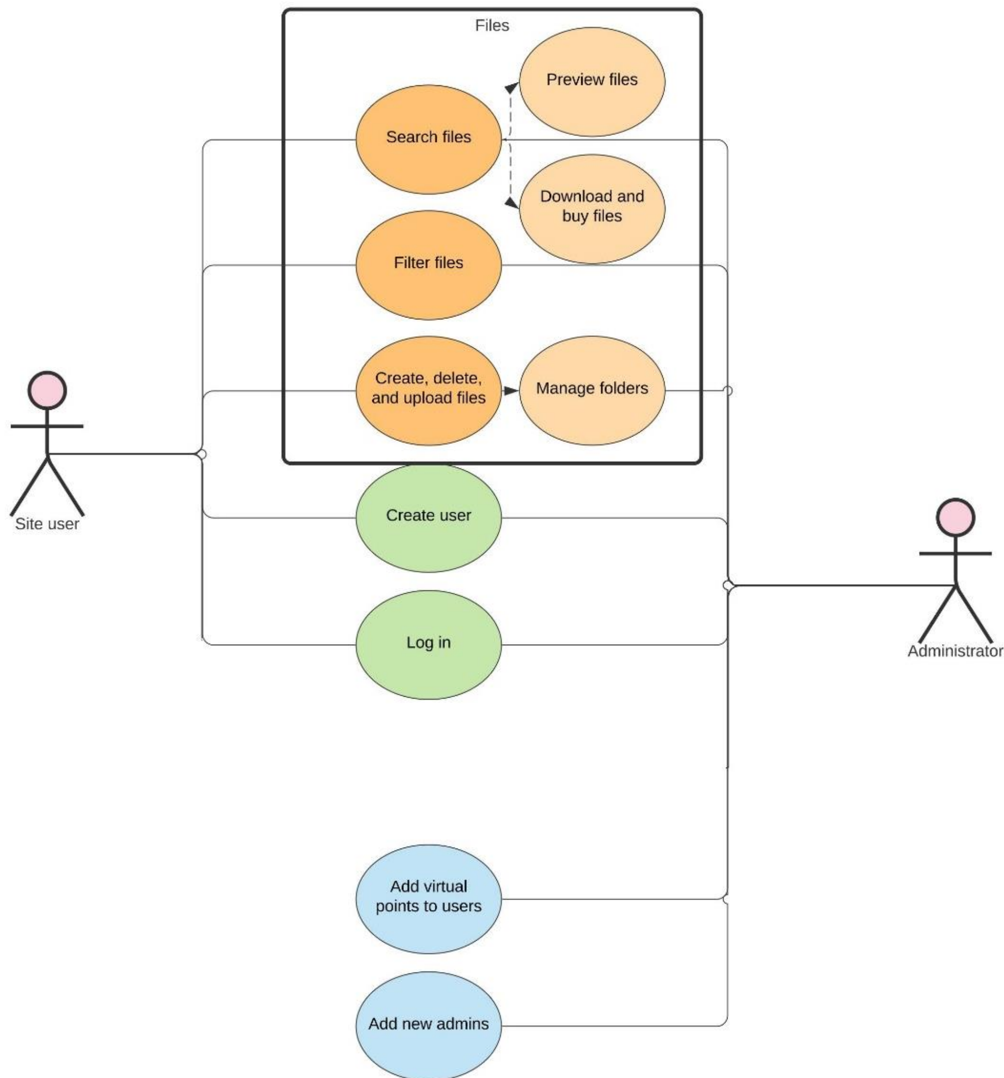

Fig. 4.3.2 Use case diagram [17]

In figure 4.3.2, can be identified the different uses offered by the website. As can be observed, all users will be able to search, filter, create, download, etc. files, in addition to performing actions such as logging in or viewing statistics for their account. On the other hand, there are the use cases of an administrator. These are, apart from any action that a normal user can perform, add virtual points to any user on the page and add new administrators (that is, transform a normal user into an administrator).

Next each of the use cases are showed up, explained by means of a use case table, which shows a short description of it, some of the necessary conditions, and the usual workflow:

5.3.1. Search files

Use case name	Search files	
Use case id	*1*	
Use case description	It helps the actor to find the files by name/description that he will preview or download later.	
Actors	User, admin	
Pre-condition	• Actor has logged in	
Post-condition	Success	1. The files with the name searched are displayed. 2. The actor can enter to the files' page.
	Failed	1. An error message is displayed.
Paths		
Ordinary workflow	**Step**	**Action**
	1	Actor *clicks* on the Search hyperlink on the nav box.
	2	The system navigates to the *Search* page.
	3	Actor *clicks* on the Search box and type the string he wants to find.
	4	Actor *clicks* on the *Search* button.
	5	The system filters all the files with a name or a description that contains the word typed.
	6	The system displays these files.
Alternative workflow	**Step**	**Action**
	5b	The system does not find any file with the string typed in the name or in the description.
	6b	The system displays an error message.

5.3.2. Preview files

Use case name	Preview files	
Use case id	*2*	
Use case description	It displays a *file's page*	
Actors	User, admin	
Pre-condition	• Actor has logged in • Actor can select a file's box (he is on the *Search page* or on the *User page*)	
Post-condition	Success	1. The *file's page* is displayed.
	Failed	1. A message error is displayed

Paths		
Ordinary workflow	**Step**	**Action**
	1	Actor *clicks* on the file's box with the name of the file he wants to preview.
	2	The system navigates to the *file's page*.
	3	The *file's* page is displayed.
Alternative workflow	**Step**	**Action**
	2b	The system does not find the file's page.
	6b	The system displays an error message.

5.3.3. Download and buy files

Use case name	Download and buy files	
Use case id	*3*	
Use case description	It lets the actor download a file and store a file's id in his *User page*.	
Actors	User, admin	
Pre-condition	• Actor has logged in • Actor is on a file's page	
Post-condition	Success	1. The files with the name searched are downloaded. 2. The file's box is stored in the *User's page*.
	Failed	1. An error message is displayed.

Paths		
Ordinary workflow	**Step**	**Action**
	1	Actor *clicks* on the *Download button* on the *file's page*.
	2	The system checks if the number of virtual points the actor has is bigger than the price of the file.
	3	The condition resolves to true.
	4	The download of the file starts.
	5	A success message is displayed.
	6	An instance of the file's box is stored in the *User's page*.
	7	The system navigates to the *User's page*.
Alternative workflow	**Step**	**Action**
	3b	The condition resolves to false.
	4b	The system displays an error message.
	5b	The system asks the actor if he wants to add points and how many.
	6b	The actor enters the required data and press the submit button.
	7b	This data is stored in the admin's *User page*.
	8b	The system displays a success message.

5.3.4. Filter files

Use case name	Filter files	
Use case id	*4*	
Use case description	It helps the actor to find the files by tags or order them by some criteria in the *Search page*.	
Actors	User, admin	
Pre-condition	• Actor has logged in • Actor is on the *Search page*.	
Post-condition	Success	1. The files with the tags' id searched are displayed. 2. The files are order by the specified criteria.
	Failed	1. An error message is displayed.
Paths		
Ordinary workflow	**Step**	**Action**
	1	Actor *clicks* on the *Tags* button.
	2	The system displays all the available tags.
	3	Actor *clicks* on every tag he wants to search for.
	4	Actor *clicks* on the criteria *Order by* button that he wants.
	5	The system filters all the files with the tag id stored on it.
	6	The system displays these files order by the criteria specified.
Alternative workflow	**Step**	**Action**
	5b	The system does not find any file with the specified tag.
	6b	The system displays an error message.

5.3.5. Create, update and delete files

Use case name	Create, update and delete files	
Use case id	*5*	
Use case description	It let the actor create, update information or delete his files.	
Actors	User, admin	
Pre-condition	• Actor has logged in • Actor is on the *User page*.	
Post-condition	1. The files are created or updated or deleted.	
Paths		
Create	**Step**	**Action**
	1	Actor *clicks* on the *Create file* hyperlink.

	2	The system navigates to the *Create file* page.
	3	Actor types and adds all the required data.
	4	Actor *clicks* on the *Create file* button.
	5	A new file is created and stored in the database.
	6	The file is stored in the *User page* of the actor.
	Alternative workflow	
	Step	**Action**
	3b	Actor does not fill all the required data.
	4b	Actor *clicks* on the *Create file* button.
	5b	An error message is displayed.
	6b	The system returns to step 2.
Update	**Step**	**Action**
	1	Actor navigates to one of his created *file's pages*.
	2	Actor *clicks* on the *Update file* hyperlink.
	3	The system navigates to the *Update file* page.
	4	Actor types and adds all the required data.
	5	Actor *clicks* on the *Update file* button.
	6	The file is updated and stored in the database.
	Alternative workflow	
	Step	**Action**
	4b	Actor does not fill all the required data.
	5b	Actor *clicks* on the *Update file* button.
	6b	An error message is displayed.
	7b	The system returns to step 2.
Delete	**Step**	**Action**
	1	Actor navigates to one of his created *file's pages*.
	2	Actor *clicks* on the Dele*te file* hyperlink.
	3	A warning message is displayed
	4	Actor confirms he want to delete the file by *clicking* the *Yes* button
	5	The file is deleted from the database and the *User page*.

5.3.6. Create user

<table>
<tr><td>Use case name</td><td colspan="2">Create user</td></tr>
<tr><td>Use case id</td><td colspan="2">6</td></tr>
<tr><td>Use case description</td><td colspan="2">Create a new user to be able to do other actions in the web application</td></tr>
<tr><td>Actors</td><td colspan="2">Client</td></tr>
<tr><td>Pre-condition</td><td colspan="2">• Not being logged in the web application</td></tr>
<tr><td rowspan="2">Post-condition</td><td>Success</td><td>1. A new user is created.
2. Actor logs in the web application.</td></tr>
<tr><td>Failed</td><td>2. An error message is displayed.</td></tr>
<tr><td colspan="3">Paths</td></tr>
<tr><td rowspan="10">Ordinary workflow</td><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>Actor navigates from any browser to the web Home page.</td></tr>
<tr><td>2</td><td>Actor clicks on Register button.</td></tr>
<tr><td>3</td><td>The system displays the Create user page.</td></tr>
<tr><td>4</td><td>Actor fills all the required data.</td></tr>
<tr><td>5</td><td>Actor clicks on the Create new user button.</td></tr>
<tr><td>6</td><td>The system checks if all the required data is filled and if it is valid.</td></tr>
<tr><td>7</td><td>The required data is valid.</td></tr>
<tr><td>8</td><td>A new user is created and stored in the database.</td></tr>
<tr><td>9</td><td>The system navigates to the Home page.</td></tr>
<tr><td rowspan="4">Alternative workflow</td><td>Step</td><td>Action</td></tr>
<tr><td>7b</td><td>The required data is not filled or valid.</td></tr>
<tr><td>8b</td><td>The system displays an error message.</td></tr>
<tr><td>9b</td><td>The system returns to the step 3.</td></tr>
</table>

5.3.7. Log in

<table>
<tr><td>Use case name</td><td colspan="2">Log in</td></tr>
<tr><td>Use case id</td><td colspan="2">7</td></tr>
<tr><td>Use case description</td><td colspan="2">Log in as a user or as an admin.</td></tr>
<tr><td>Actors</td><td colspan="2">User, admin</td></tr>
<tr><td>Pre-condition</td><td colspan="2">• Not being logged in the web application</td></tr>
<tr><td rowspan="2">Post-condition</td><td>Success</td><td>1. Actor logs in the web application</td></tr>
<tr><td>Failed</td><td>1. An error message is displayed.</td></tr>
</table>

Paths		
	Step	**Action**
Ordinary workflow	1	Actor navigates from any browser to the web *Home* page.
	2	Actor *clicks* on *Log in* button.
	3	The system displays the *Log in* page.
	4	Actor type his username and password.
	5	Actor *clicks* on the *Log in* button.
	6	The system checks if the required data is valid.
	7	The required data is valid.
	8	Actor logs in the web application.
	9	The system navigates to the *Home* page.
Alternative workflow	**Step**	**Action**
	7b	The required data is not filled or valid.
	8b	The system displays an error message.
	9b	The system returns to the step 3.

5.3.8. Add virtual points to users

Use case name	Add virtual points to users		
Use case id	*8*		
Use case description	The admin can add more *virtual points* to the users that require them.		
Actors	Admin		
Pre-condition	• Admin is on the *User* page.		
Post-condition	Success	1. The user selected has more points than before.	
Paths			
	Step	**Action**	
Ordinary workflow	1	Actor *clicks* on the *Add virtual points* hyperlink.	
	2	The system navigates to the *Add points* page.	
	3	Actor *selects* the user to whom add *virtual points* and how many of them add.	
	4	Actor *clicks* on the *Confirm* button.	
	5	The system adds *virtual points* to the user selected.	
	6	The system displays these files.	
	7	The data of this user is deleted from the *Add virtual points* section of the admin page.	

5.3.9.Add new admins

Use case name	Add new admins		
Use case id	*9*		
Use case description	The admin can convert normal users in admins.		
Actors	Admin		
Pre-condition	• Admin is on the *User* page.		
Post-condition	Success	1. The user is converted into an admin,	
Paths			
Ordinary workflow	**Step**	**Action**	
	1	Actor *clicks* on the *Add admin* hyperlink.	
	2	The system navigates to the *Add admin* page.	
	3	Actor *selects* the user he wants to convert to admin.	
	4	Actor *clicks* on the *Confirm* button.	
	5	The system converts the user into an admin.	

5.4. Activity diagrams

For now, it has been explained how different users will act when they want to perform a certain action, but it is not known how the system will act yet. The activity diagrams are considered behaviour diagrams because they describe what should happen in the modelled system.[20]

Stakeholders have many issues to handle. Activity diagrams help anyone interested to integrate to understand the application's process and behaviour. There are a set of specialized symbols (for starting, ending, merging steps ...) that will allow each activity to be correctly exemplified.

In every diagram will be located the following elements:

- *Action:* A step in the activity in which users or software perform a given task.
- *Decision Node:* A conditional branch in the stream that is represented by a diamond. It indicates that the flow can take two different paths.
- *Control Flows:* The connectors that show the flow between steps in the diagram.
- *Initial node:* Symbolizes the start of the activity. The starting node is represented by a black circle.
- *Terminal node:* Represents the final step in the activity, represented by a black circle with a white outline.

5.4.1. *Sign up*

The first action performed by any user, if he wants to access the functionalities of the web application, will be to register on the web page. There is a link enabled for this on the navigation

page, which will allow registration. Once registered, the user will not be able to log in until they confirm their email address, accessing it and *clicking* on the URL provided.

Its performance is:

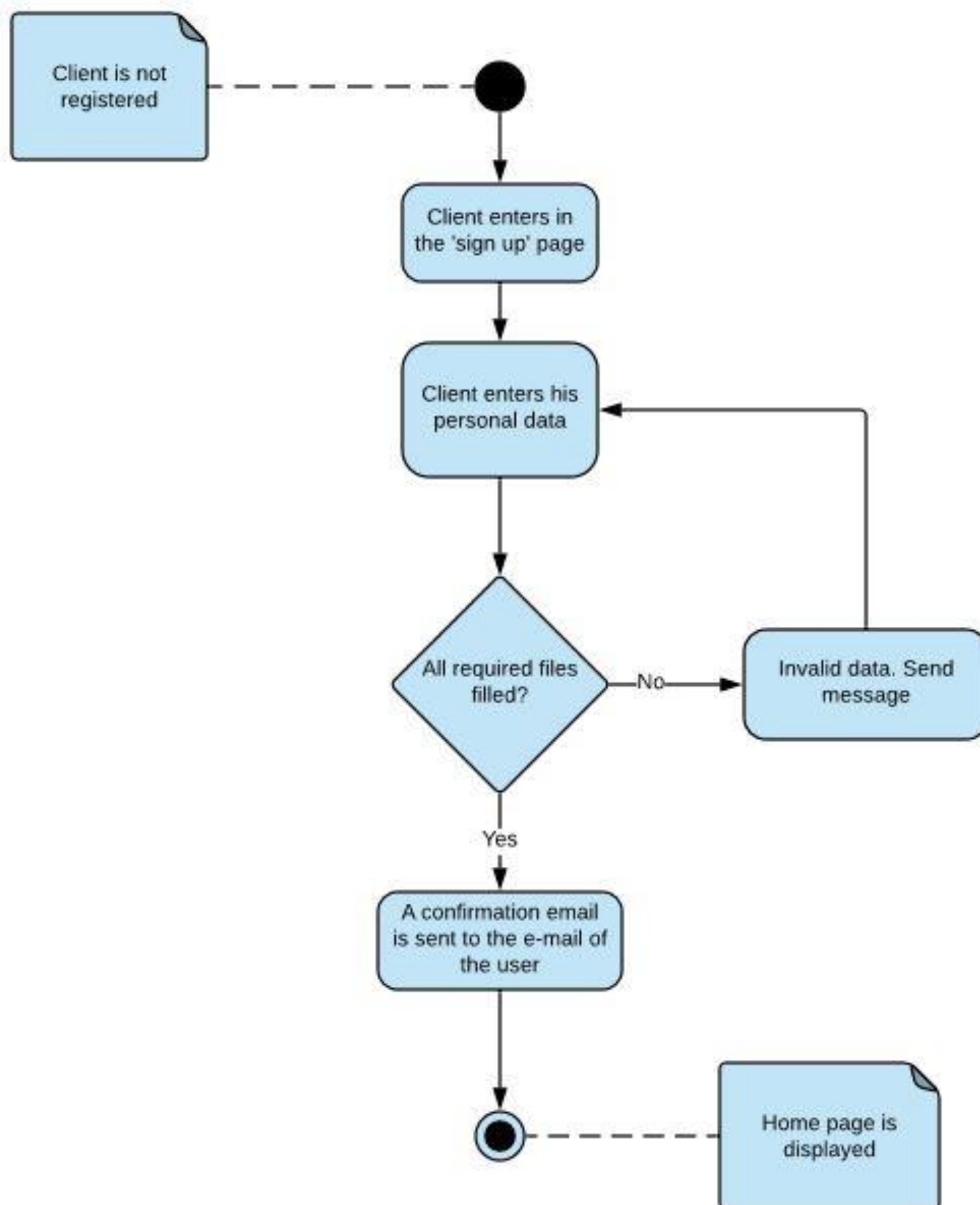

Fig. 5.3.1 Activity diagram for signing up [18]

5.4.2. *Sign in*

Once the user has been registered and has confirmed his email address, the user will be able to log in. He only must *click* on the link enabled for it in the navigation bar and correctly fill in his credentials. If any error occurs by the user when filling in the fields, it will be shown on the screen.

Once the session has started, the user will be redirected to their profile page.

Basically, it works as shown in figure 5.3.2.

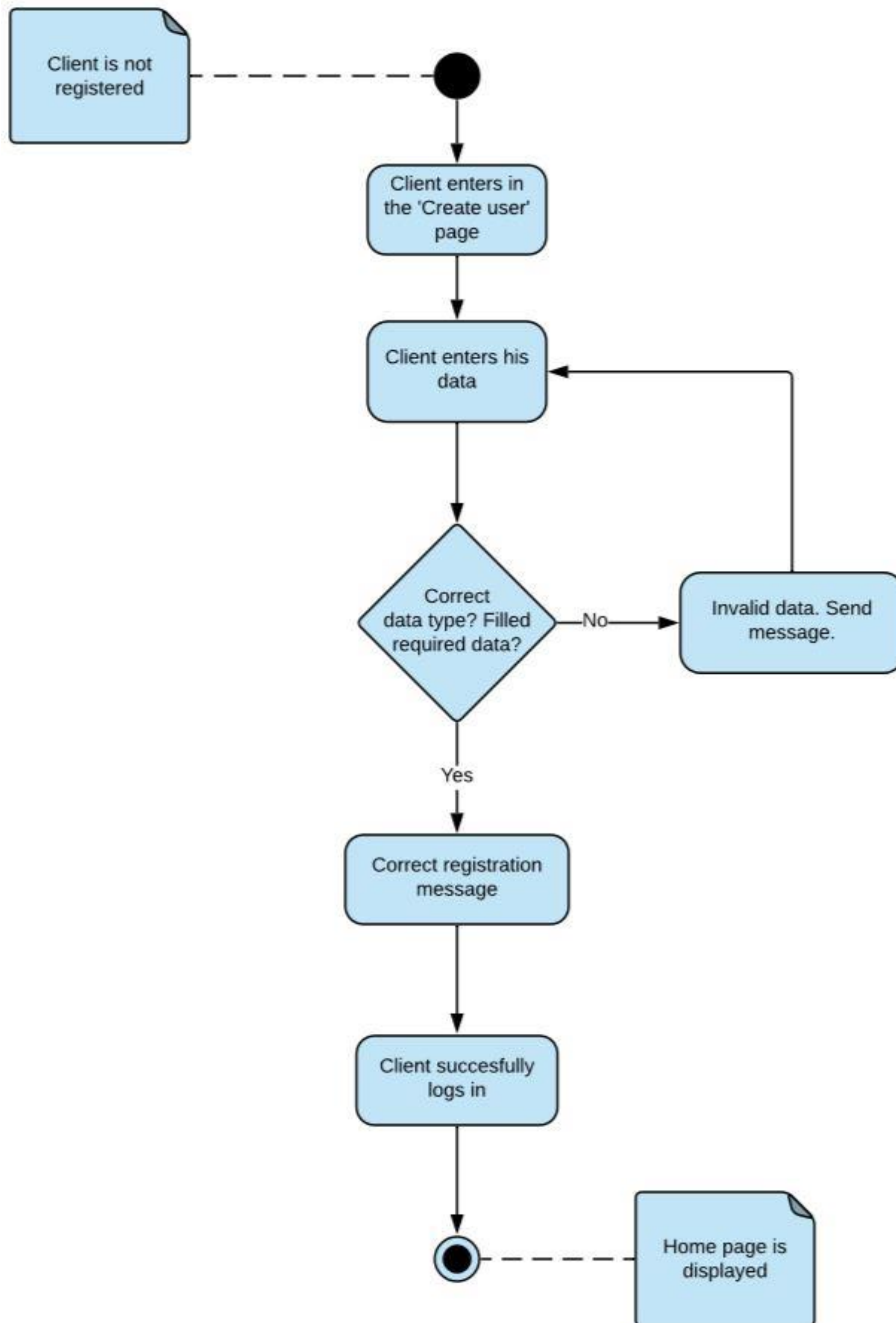

Fig. 5.3.2 Activity diagram for signing in [19]

5.4.3. *Create a file*

The main function of the website is file sharing. Therefore, the participation of users in it is necessary and the best way for them to contribute is to create files.

Once users are registered in the application, they can create a file from the *users* page, giving it a name, a description and a price.

If the file has been created correctly, it will be saved, in addition to the database, on the page of the current user, so that you can access it whenever you consider it necessary.

Its activity diagram is the next:

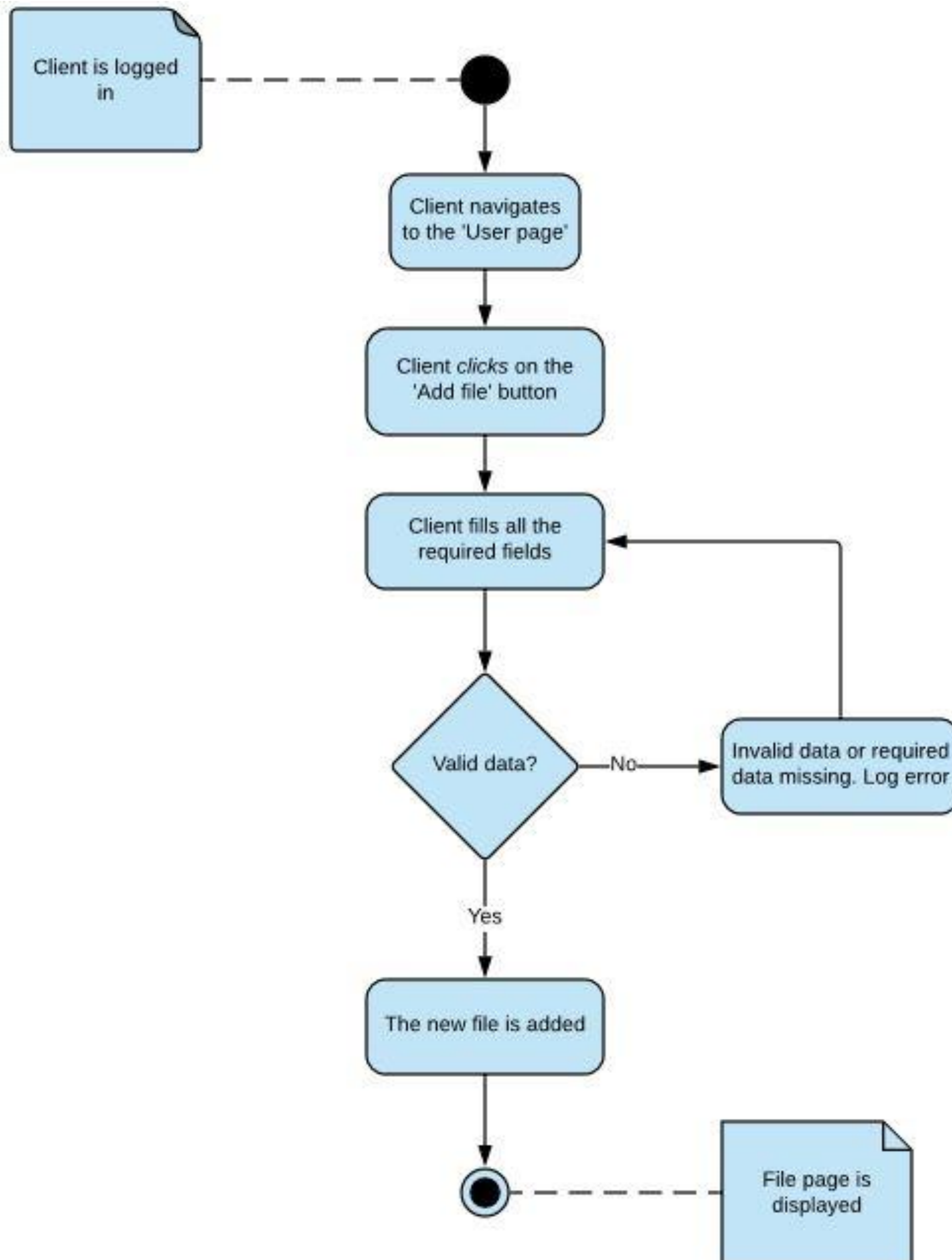

Fig. 5.3.3 Activity diagram for creating a file [20]

5.4.4. *Download a file*

By downloading a file, content will be obtained through the server that receives the data that is accessed by clients through the applications. This will be possible once the user has searched for a specific file on the searching page. Subsequent viewing of the content (available locally on the device) will be possible once the download is complete. Once the user clicks on the download button, the file will be saved on the user profile page, making it accessible at any time.

It can be understood thanks to figure 5.3.4.

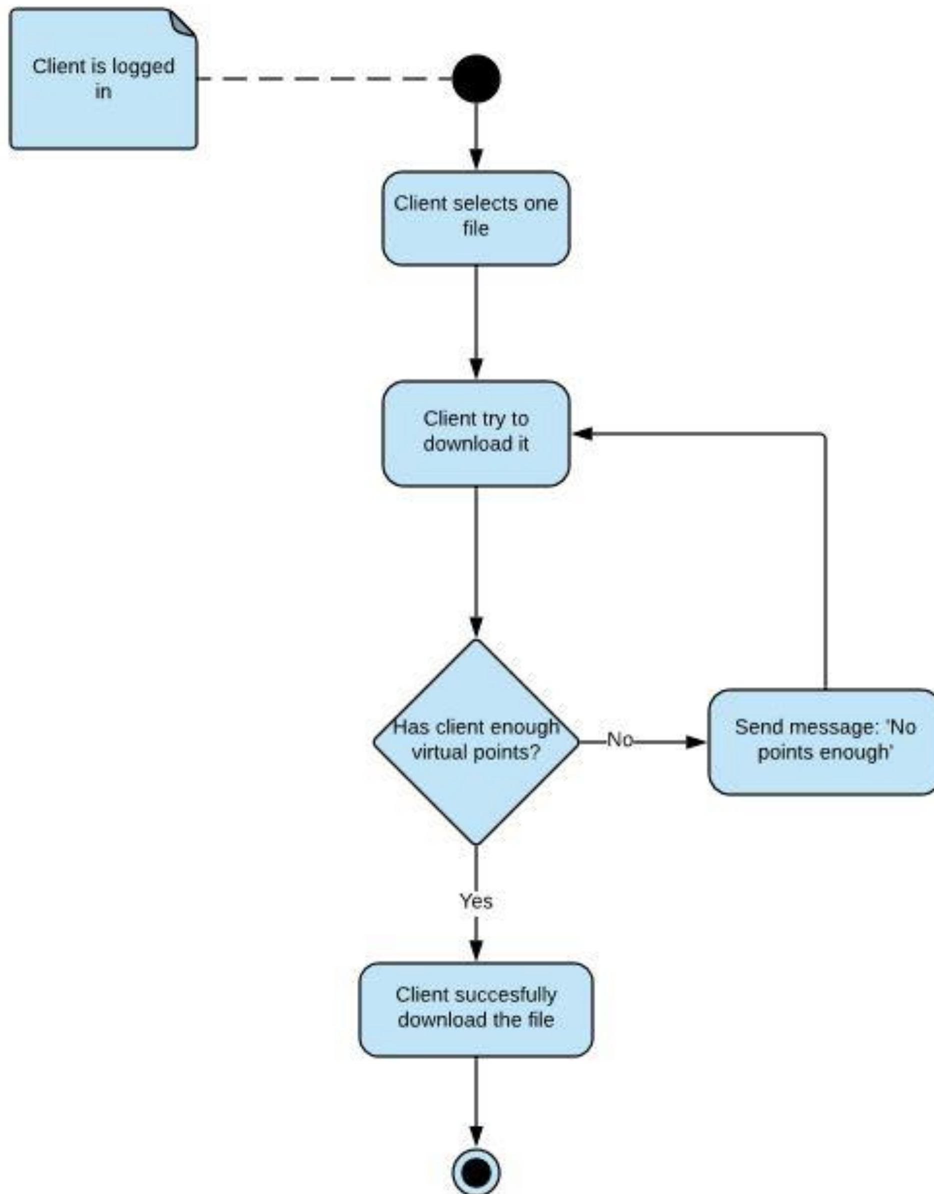

Fig. 5.3.4 Activity diagram for downloading a file [21]

5.4.5. *Searching files*

A search box is a graphical control element very important in this application. Files can be filtered by name and by tags. In addition, files can be sorted by creation date, by number of downloads and by score. Therefore, it will allow users to enter a query to be submitted to the web search engine server-side script, where an index database is queried for entries that contain one or more of the user's keyword research.

The search box will be accompanied by a search button to submit the search and the query will start when it is clicked. After that, the results will be rendered by screen.

The search box is very important as it is an integral part of the site search functionality and very useful in content-rich websites.

The activity diagram for searching files is the next:

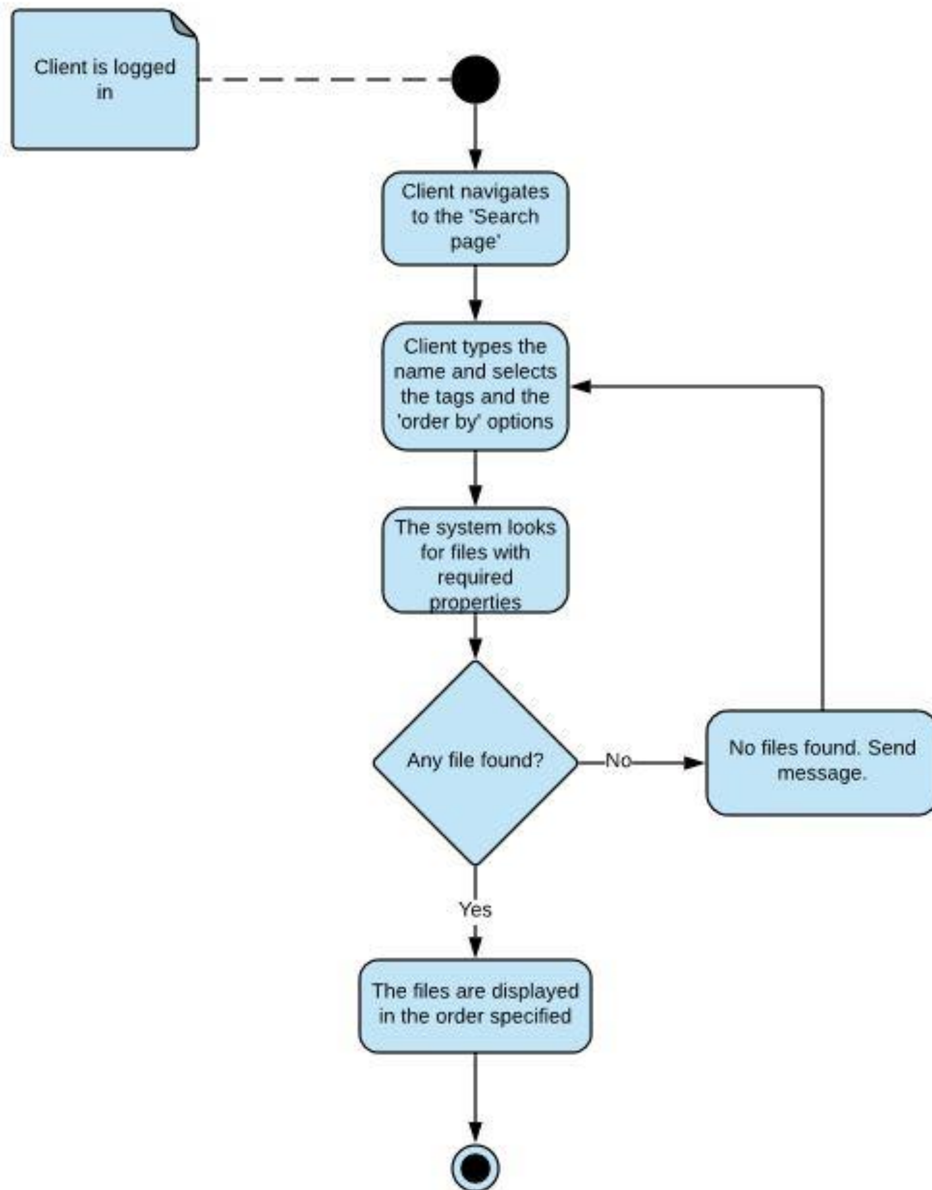

Fig. 5.3.5 Activity diagram for searching files[22]

5.4.6. *Add new admin* and *add points*

When an administrator-type user is logged in, a special option will appear on their profile page, which cannot be accessed by a *normal* user. Thanks to it, you can perform two special actions: *add new users* and *add points* to other users.

To add a user as administrator, you must select this user from a list and press the '*Add admin'* button.

To add virtual points to the account of a certain user, it will be necessary to simply select the user at which adding points to and the number of points to add, with 50 points being the default amount.

The way these functions work is shown in figures 5.3.6 and 5.3.7.

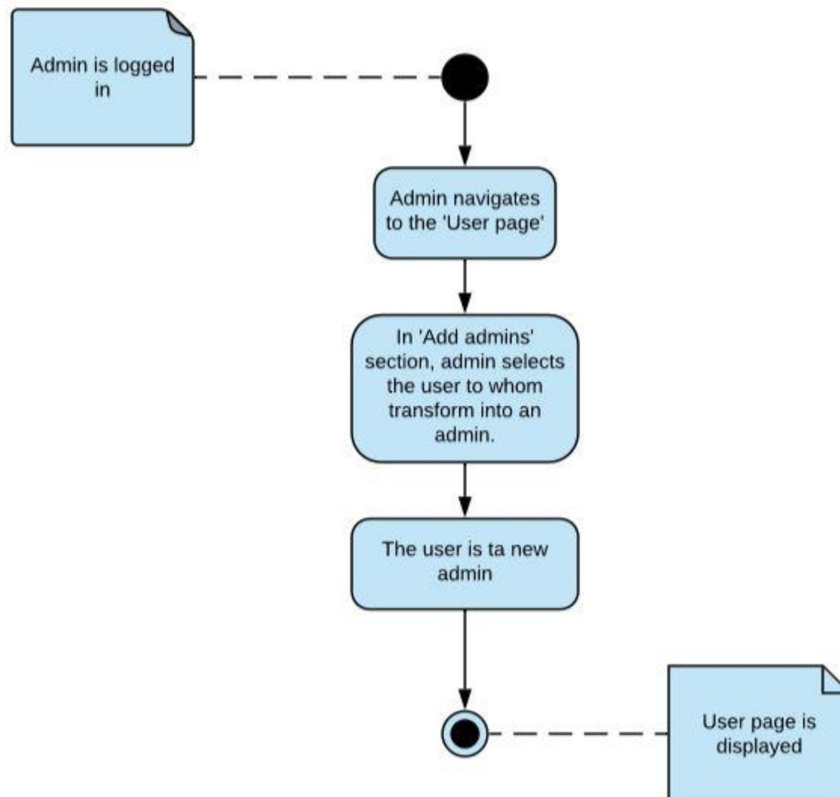

Fig. 5.3.6 Activity diagram for adding new admins [23]

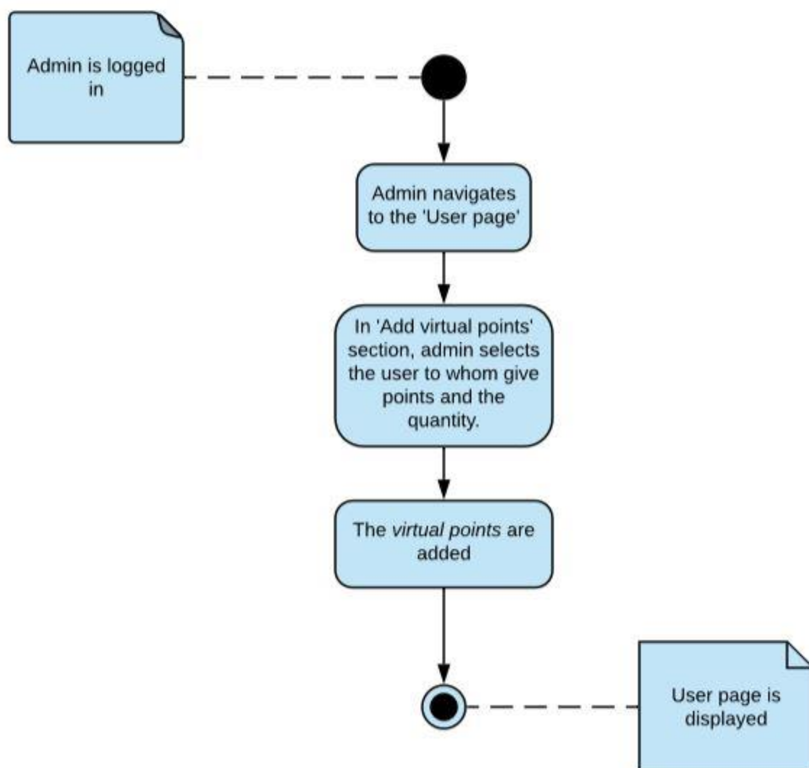

Fig. 5.3.7 Activity diagram for adding points to users [24]

5.5. Database diagram

This diagram will be a visual tool that allows to design and view a database to which the application is connected. A database model shows the logical structure of the database, including the relationships and limitations that determine how data is stored and how it is accessed.

- Key column This characteristic can take, in addition to a null value, the value of PK (*primary key*) or FK (*foreign key*). PK will be used only once per table and this will be the number that will identify a specific element within the table. The FK will be used to relate a certain element to another in a different table.
- Title column. Displays the name of each of the properties of an element.
- Data type column It will show us how each of the properties is to be stored

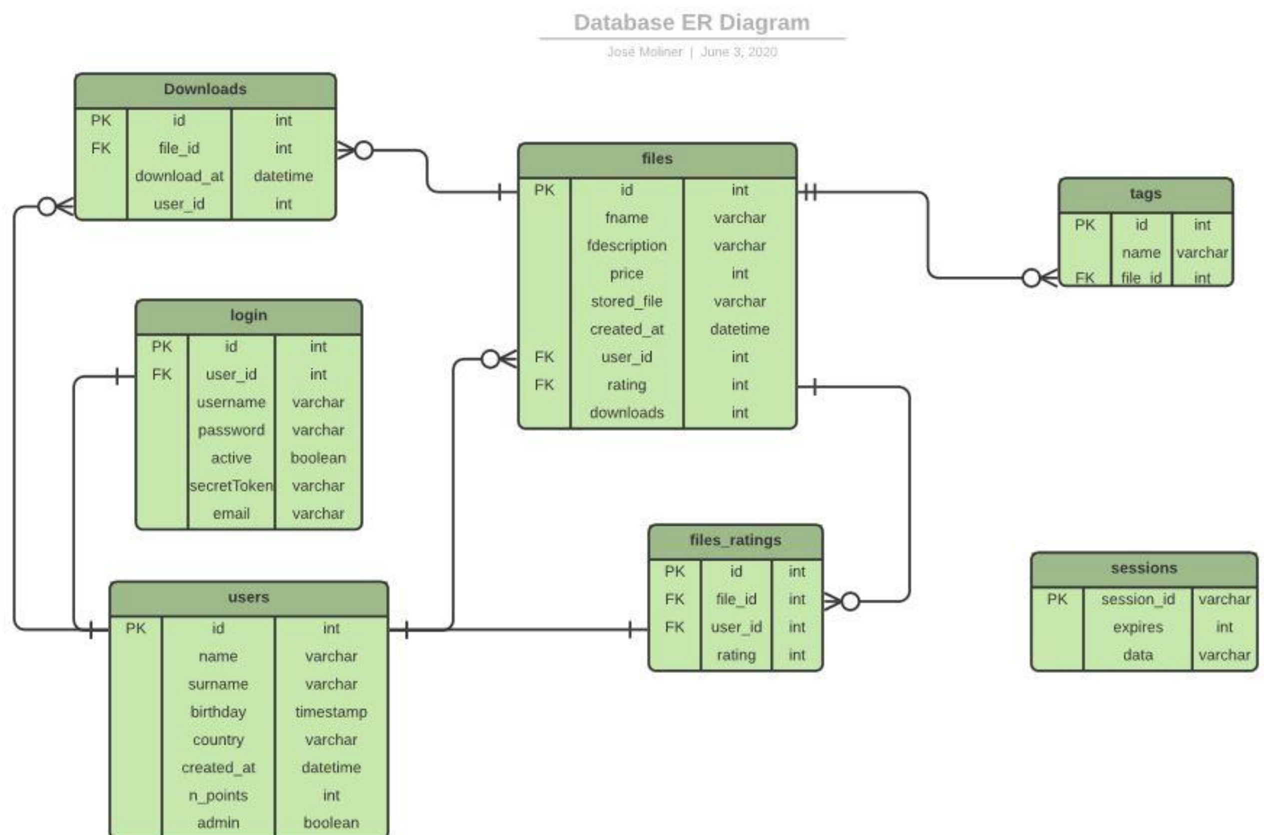

Fig. 5.4.1 Relational Entities database diagram [25]

In the diagram (figure 5.4.1), the relationships between the different tables can be easily identified. Among them are located different tables for *files*, *users*, *login*, *downloads*, *tags* and ratings (*'files_ratings'*), in addition to a table called *sessions* that apparently is isolated from the rest. This last table is created by a *Node.js* module [4]and is in charge of managing the different sessions created by the web page. The workflow to understand the structure of this database, following all the steps would be as follows:

[4] This module is *express-sessions-mysql*. It will be mentioned in next chapter.

A new user is created, and his personal data will be stored in the *users* table and his credentials and other data necessary for logging in will be saved into the *login* table. Once the user is registered, they can log in, storing his session in the *sessions* table. The main function of this web page is the creation of files, so the user decides to create one, storing it in the *files* table. If he decides to add a tag to the file, it would be stored by in the *tags* table, instead of in the *files* table, to facilitate the searching function later. Once several files exist in the database, the user can decide to download one. If he finds a file with the requirements he wants and downloads it, this download will be stored in the *downloads* table. In addition, he can rate the file, storing this valuation in the *files_ratings* table.

Arguably, the preceding paragraph is a short summary of how the web application created works. However, it is still not understood how it can perform the mentioned actions. In the next chapter an extensive exposition of the application structure will be carried out.

6. Application source code

The source code makes up the entire body of the web application. It is a set of lines of text with the steps that the computer must follow to run a program.

The source code is many times written in some programming languages[5], but in this first state it is not directly executable by the computer. It must be translated into another language or binary code. This will make it easier for the machine to interpret it.

In the following sections, each part of the source code will be analysed. Many of the functions that have been developed are similar, so only a few will be taken as an example.

6.1. Directory structure

The project has the following file structure:

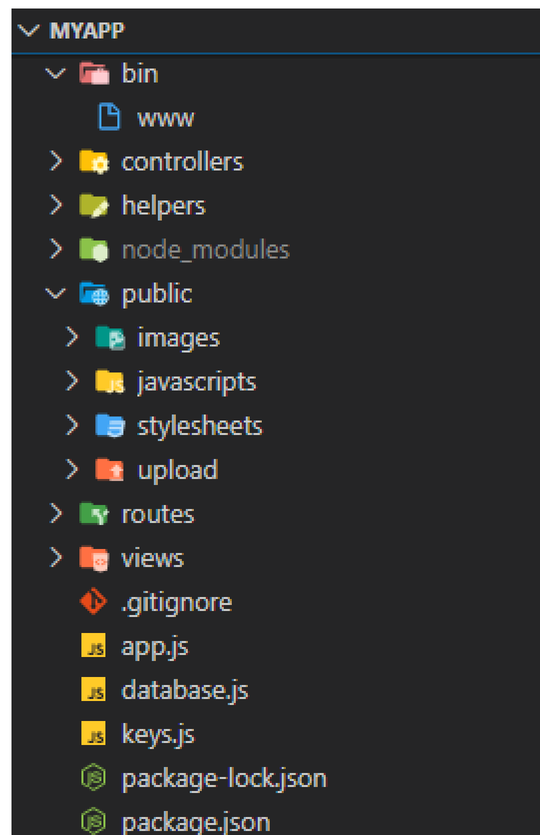

Fig. 6.1.1 directory structure [26]

In general terms, the '*package.json*' file is where are saved various relevant metadata from the project. It is the letter of introduction of the web application. The entry point is the '*/bin/www*' file, which sets up some error handling and then loads '*app.js*' to do the rest of work (all the

[5] Go to chapter 4 for more information about the programming languages selected

functions inside these files will be define later, as they are very important for the correct performance of the web application).

Then, we can see a '*node_modules*' folder where are stored the dependencies of the project, a *'views'* folder, where the pug templates will be saved, the '*routes*' folder, where the references of our files will be defined, the '*controllers*' folder, where have been saved the functions that will be called in the *'routes'* folder, and a '*public*' folder, where are located the static files that we will serve in our web site (*images*, *javascripts*, *stylesheets* and *upload*). Also, a *'helpers'* folder was created, where will be stored some of the needed functions to have more order.

Finally were written a '*database.js*' file, which will be the file that will have access to the database, and a '*keys.js*', where will be saved many of the keys that will be used in the project to create connections.

The *'gitignore'*, although not being essential, it is useful if we want to *'git'* our project. Here can be written the files that we do not want to *'git'* and the program will ignore them.

6.2. package.json

In this file we can find general information about the web application [21], like the name or the version. It also defines a start-up script that will call the application entry point which, in this case, the '*/bin/www*' file (Lis. 6.1.1).

```
{
    "name": "myapp",
    "version": "0.0.0",
    "private": true,
    "scripts": {
    "start": "node ./bin/www",
    "devstart": "nodemon -e js,pug ./bin/www"
},
```

Lis. 6.1.1 package.json [1]

Apart from this, here is where information is given to the *node package manager (npm)* to identify the project dependencies that will be needed to make the application useful, among which we find:

- `bcryptjs`: is a module to encrypt user passwords before storing them in the database.
- `connect-flash:` is a module to show erroneous or successful messages when the user performs an action.
- `cookie-parser`: is used to parse the cookie header and populate `req.cookies`.
- `debug`: is a tiny *node* debugging utility modelled after *node* core's debugging technique.
- `express`: express is the most popular *node.js* framework for creating back-end applications.
- `express-mysql-session`: is a module that will store the sessions in the database.
- `express-session`: it manages sessions of apps. It is required for user authentication.

- `express-validator`: is a module to validate the data that the user sends us from the client application.
- `fs-extra`: is an important module as it adds file system methods that are not included in the native `fs` module and adds promise support to the `fs` methods.
- `http-errors`: is a module that creates HTTP errors for Express.
- `morgan`: it will help to show by console what the client is asking for to the server.
- `multer`: is a module that will help with the uploading of different files to the web application.[22]
- `mysql`: is an *npm* module to connect and make queries to the database.
- `nodemailer`: is a module useful when it is needed sending automated mails from the web application.
- `passport`: a module to authenticate and manage the login process for the application.
- `passport-local`: is a passport add-on to authenticate users with our own database.
- `pug`: is the package for the template engine.
- `randomstring`: helps to create random strings with a specific length.
- `timeago.js`: is a module that will help to convert the date formats into a different one that will be used later in our application.
- `nodemon`: is a developer module which will be used to automatically restart the execution of the file every time the application is saved during development.

The scripts section defines a 'start' script that will invoke the server to start. Two scripts have been defined to start the '*./bin/www*' file with '`start`' but also with `devstart`. This last starts it with `nodemon` instead of with *node*. It will be only used in the development stage.

6.3. app.js

In this JavaScript file is where it is created an *express* application object, whose name is *app*. Here it will be taken care of configuring a large part of the application, using some of the middleware. Finally, the *app* will be exported from the module.

```
const createError = require('http-errors');
const express = require('express');
const path = require('path');
const cookieParser = require('cookie-parser');
const morgan = require('morgan');
const multer = require('multer');
const flash = require('connect-flash');
const session = require('express-session');
const MySQLStore = require('express-mysql-session');
const passport = require('passport');
```

Lis. 6.3.1 app.js libraries [2]

At the top of the file (Lis.6.3.1), first are imported all the required libraries that will be used along the configuration and that have been downloaded and stored before in the *package.json* file (*http-errors, express, cookie-parser, morgan…*). Apart from them, it is imported *path*, a Node library for parsing file and directory paths.

```
const indexRouter = require('./routes/index');
const authenticationRouter = require('./routes/authentication');
const searchRouter = require('./routes/search');
const userRouter = require('./routes/user');


const { database } = require('./keys');
```

Lis. 6.3.2 app.js modules [3]

After that, the modules are imported from the *routes* directory which will handle of the different URL paths, that is, the different routes. Then, the *'keys'* file, which will save some private data to access to different platforms (Lis. 6.3.2).

```
// initialitzations
const app = express();
require('./helpers/passport');

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

Lis. 6.3.3 Some app.js initializations [4]

Then, the *app* object can be created, and the *passport* file is initialized.

After that, the engine template is set up, in this case '*pug'*. First, it is defined where will be stored the different *'pug'* files and, then, it is defined *'pug'* as the default view engine (Lis. 6.3.3).

```
// Middlewares and their configuration
app.use(session({
      secret: 'finalproject',
      resave: false,
      saveUninitialized: false,
      store: new MySQLStore(database)
}))
```

Lis. 6.3.4 Some app.js initializations [5]

```
app.use(flash());
app.use(morgan('dev'));
app.use(express.json());
app.use(multer({dest: path.join(__dirname, './public/upload/temp')}
).single('file'));
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(passport.initialize());
app.use(passport.session());
```

Lis. 6.3.5 Some app.js initializations [6]

Next comes the calling of `app.use` to add many of our middleware modules (Lis. 6.3.5).[23] The first middleware used is *'session'*. This is a function that receives an object with various properties:

- `Secret` is a property that is used by a cryptographic algorithm that takes the string and uses it as a *salt* to encrypt the session ID cookies sent to the client.
- The `resave` property forces the session information to be saved in the database for each call made to the server, regardless of whether there was a change on it or not.
- `SaveUninitialized` forces the server to save the sessions into the database even if no changes have been made to them.
- Finally, `store` is the property that will allow to save the session in the app database. With *'new MySQLStore*', the sessions database table should be automatically created, when using default options.

Then are used some middlewares for showing messages to the client (*'flash'*) or to developers (*'morgan'*), for parsing incoming requests with JSON payloads or with urlencoded payloads (*'express.json'* and *'express.urlencoded'*, based on '*body-parser'*), for helping with the uploading of files to the application (adding a body object and a file or files object to the request object, *'multer'*), for handling cookies(*'cookie-parser'*) and for creating sessions (*'passport.initialize'* for creating them and *'passport.session'* for saving them in the previously created session).

```
// public
app.use(express.static(path.join(__dirname, 'public')));
```

Lis. 6.3.6 app.js public folder configuration [7]

After them, `express.static` is used for configuring where will be found the static files (which were saved in the *public* folder) (Lis. 6.3.6).

```
// global variables (thanks to them, this variables can be accessed
from any view)
app.use((req, res, next) => {
	app.locals.success = req.flash('success');
	app.locals.message = req.flash('message');
	app.locals.user = req.user;
	next();
});

// routes
app.use('/', indexRouter);
app.use('/authentication', authenticationRouter);
app.use('/search', searchRouter);
app.use('/user', userRouter);
```

Lis. 6.3.7 Global variables and routes' configuration [8]

Now, the global variables are set up and, thanks to that, the application will be able to access some variables from any part of the code. The local variable *user*, for example, will be very useful in the project, as it will be needed for making queries to the database for the current user.

Then are used some modules which handle with routes[6] and they define the direction at which each route points. In *'/'* will be found the routes to pages related with the index (*Home page*), in *'/authentication'* the routes to the authentication pages… and so on (Lis. 6.3.7).

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
	next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
	// set locals, only providing error in development
	res.locals.message = err.message;
	res.locals.error = req.app.get('env')==='development'?err:{};

	// render the error page
	res.status(err.status || 500);
	res.render('error');
});
```

[6] The *routes* files are introduced in chapter 6.6

Lis. 6.3.8 app.js errors' handling [9]

Finally, have been created methods to take care of some errors (for example, when the server does not find some route), creating some local variables for handling them, and *rendering* the *'error.pug'* template if an error occurs (Lis. 6.3.8).

```
module.exports = app;
```

Lis. 6.3.9 app.js is exported [10]

When the *app* configuration is finished, the *app* object is exported (Lis. 6.3.9).

6.4. WWW Server File

To start writing code for the web application with *Node.js*, it can be useful use the *Express* application generator that it contains (Lis. 6.4.1).

```
$ npm install express-generator -g
```

Lis. 6.4.1 express-generator installation [11]

It creates a very basic app structure that will help developers to organize their work. One of the most important files created, is the *'www'* file, created inside the *'bin'* folder.[24]

The *bin/www* file is the application entry point for *Express*. It serves as a location where define the start-up scripts.

```
const app = require('../app');
const debug = require('debug')('myapp:server');
const http = require('http');
```

Lis. 6.4.2 www libraries and modules [12]

The first thing it does is to import the *app* object from *app.js* file (where have been made previously all the configuration and routing), set the server debug and name (*'debug'* exposes a function and pass it the name of your module, and it will return a decorated version of '*console.error'* for passing debug statements) and loads the *http* module (Lis. 6.4.2).

```
const port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
```

Lis. 6.4.3 www port definition [13]

After that, we assign the value of the application port (if it has not been set before, we assign the value of *3000,* for development) and set it as the one the application should use to listen on. The `normalizePort` function receives a value and it ensures that the received value is a valid number port (Lis. 6.4.3).

```
const server = http.createServer(app);

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

Lis. 6.4.4 www server initialization [14]

After that, the server is created (in a very easy way, thanks to *Express),* and finally we set up the listening on provide port as well as the error and listening event, with function created for this purpose (Lis. 6.4.4).

In summary, this file is the one that we will use to start the express app as a web server.

6.5. database.js

In this file we find the configuration of the database, in this case, *MySql.* [25][26][27]

```
const mysql = require('mysql');
const { promisify } = require('util');

const { database } = require('./keys');
```

Lis. 6.5.1 database.js libraries and modules [15]

First, can be found the importation of the tools that will be needed: `mysql` (a driver for our database) and `promisify` (a function that will allow to make promises with elements that, a priori, could not) (Lis. 6.5.1).

After that, the requirement of a file called `keys.js`, which is simply the place where will be stored the access keys to the database.

```
const pool = mysql.createPool(database);
```

Lis. 6.5.2 database initialization [16]

Then `createPool` is used. Rather than creating and managing connections one-by-one, the pool module also provides built-in connection pooling using `mysql.createPool(config)`. Connection pooling is a mechanism to maintain cache of database connection so that connection

can be reused after releasing it. In Node, pooling can be used directly to handle multiple connection and reuse the connection. It is used here because all queries on the MySQL connection are made one after the other. This means that if you want to do 10 queries and each query takes 2 seconds, it will take 20 seconds to complete the entire execution. The solution is to create 10 connections and run each query on a different connection (Lis. 6.5.2).

```
pool.getConnection((err, connection) => {
  if(err) {
      if(err.code === 'PROTOCOL_CONNECTION_LOST'){
        console.error('DATABASE CONNECTION WAS CLOSED');
      }
      if(err.code === 'ER_CON_COUNT_ERROR') {
        console.error('DATABASE HAS TOO MANY CONNECTIONS');
      }
      if(err.code === 'ECONNREFUSED'){
        console.error('DATABASE CONNECTION WAS REFUSED');
       }
  }

  if (connection) connection.release();
  console.log('DB is connected');
  return;
});
```

Lis. 6.5.3 Database connection [17]

Next, we use the connection with `getConnection` and then we will handle with some errors that can occur to the database connection (these possible errors, when occur, are described in the console). And, after that, the connection is *released* (Lis. 6.5.3).

```
pool.query = promisify(pool.query);

module.exports = pool;
```

Lis. 6.5.4 Database exportation [18]

The last step is to use `promisify`, so promises will could be used when a query is done, as explained before (Lis. 6.5.4).

Finally, the *pool* object is exported.

6.6. Routes

Previously, in *app* file, we talked about routing. And in this folder will be found a great part of it.

This is the folder where will be stored all the route's files. Routing refers to determining how an application responds to a client request at a certain endpoint, which is a URI[7] (or path) and a specific HTTP request method (GET, POST, etc.). Each route can have one or more handler functions, which are excluded when the route is mapped. In this folder there are four different files: *authentication*, *index*, *search,* and *user*.

The function of each file is very similar. In fact, these files are not necessary. All the routing could be have done in the *app* file. But, anyway, this configuration is going to e very useful for the app production, as the code is more organized, and it is more intuitive where can be found each part of code.

As their functionality is similar, only one file will be explained: *user.js* file.

```
const express = require('express');
const router = express.Router();
```

Lis. 6.6.1 Routes' libraries [19]

First, *express* is imported and then, *router* is initialized. The `express.Router` class is used to create modular, mountable route handlers. A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a 'mini-app' (Lis. 6.6.1).

```
const filesController = require('../controllers/filesController');
const userController = require('../controllers/userController');
const { isLoggedIn } = require('../helpers/auth');
const { isAdmin } = require('../helpers/helpers');
```

Lis. 6.6.2 Routes' modules [20]

Then are required two objects from the *controllers'* folder (*filesController* and *userController*[8]), which contains the methods that are going to work once the pertinent route is called. The *isLoggedIn* function and the *isAdmin* function[9] will work in a very similar way, with the main difference that they are called in more than one route (Lis. 6.6.2).

After that, can be found similar structure for each route. Let us see the admin page as an example:

[7] URI: is a character string that uniquely identifies the resources of a network. [Wikipedia, URI]

[8] `filesController` and `userController` are introduced in chapter 6.7.

[9] `isLoggedIn` and `isAdmin` functions are introduced in chapter 6.10.

```
//GET request for admin page
router.get('/admin',isLoggedIn, isAdmin, userController.admin_get);

// POST request for adding admins
router.post('/admin/add', isLoggedIn, isAdmin, userController.
admin_add_post);
```

Lis. 6.6.3 Some routes' examples [21]

Route paths, in combination with a request method, define the endpoints at which requests can be made .Route paths can be strings, string patterns, or regular expressions and each one is unique, so each one has one or more actions associated (Lis. 6.6.3).

In the first route, the request method is *get*. But what is it, a request method? First, it is needed understand what *HTTP* is.

Looking for a definition in *Wikipedia 'the Hypertext Transfer Protocol (HTTP) is the communication protocol that enables information transfers on the World Wide Web'*[10], that is, the Internet language.[11]

The first HTTP method visible in the code is GET. The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. This method with its route path (*'/admin'*) calls to three different functions sequentially: *isLoggedIn*(checks if the client is a user logged in), *isAdmin* (checks if the user is an admin) and *userController.admin_get* (method of the *userController*[12] object that will render the requested page).

After this route, a new request is given, with a POST method in this case. The functionality is similar, with the main difference being that it sends data to be processed by the resource identified in the request line URI. The data will be included in the body of the request. This method is useful in several cases, whenever it is necessary to save data.

6.7. Controllers

In the *routes* folder, one or more functions are called for each route. And these functions handle with the purpose of the appeal to that specific route. Many of these functions will be located here.

As before, many of these files are very similar so only are going to be explained the interesting parts of the code.

[10] [Wikipedia]

[11] This is not totally correct, as we call Internet to the Network of Networks, and WWW is the system used to access the Internet, but this is a good analogy for the reader understanding.

[12] Go to chapter 6.7 for more information about controllers.

First, let us start with the basic structure of all files.

6.7.1. Controllers' Structure

Here is presented the format of the *filesController* file, which is very similar to the other controllers' files:

```
const path = require('path');
const fs = require('fs-extra');

const {randomName,timeago,lastItem }=require('../helpers/helpers');
const pool = require('../database');

const controller = {};

controller.file_detail_get = async (req, res, next) => {

      . . .

};

controller.file_detail_post = async (req, res, next) => {

      . . .

};

. . .


module.exports = controller;
```

Lis. 6.7.1 Controllers' files structure [22]

First, we require two modules: *'path'* and *'fs-extra'*. Both will help when working with directories and file paths (Lis. 6.7.1).

After them, as in the *routes* files, some functions from the *helpers* folder are require, which will help with many different tasks. Then it is called the object that was created to let make queries to the database (*'pool')*.

Next, can be created a new object called controller. As can be seen, at the bottom of the file, this object is exported. That is because it is the object that will be called for the different routes of the web application. Each function that is called, is a method of this object (*controller.file_detail_get*, *controller.file_detail_post…*).

In the following sections, some of the different controllers will be presented.

6.7.2. File detail

Many of the methods present on *controllers'* files, have a couple with which they complement each other. This is the case of the previous example: the *file detail methods*.

Here we have a *controller.file_detail_get* method and a *controller.file_detail_post method*. The first, will oversee rendering the requested information and, the second, will handle with posting the information sent by the client on the database.

Let us start with *controller.file_detail_get* method, that will cause the rendering of a page like this:

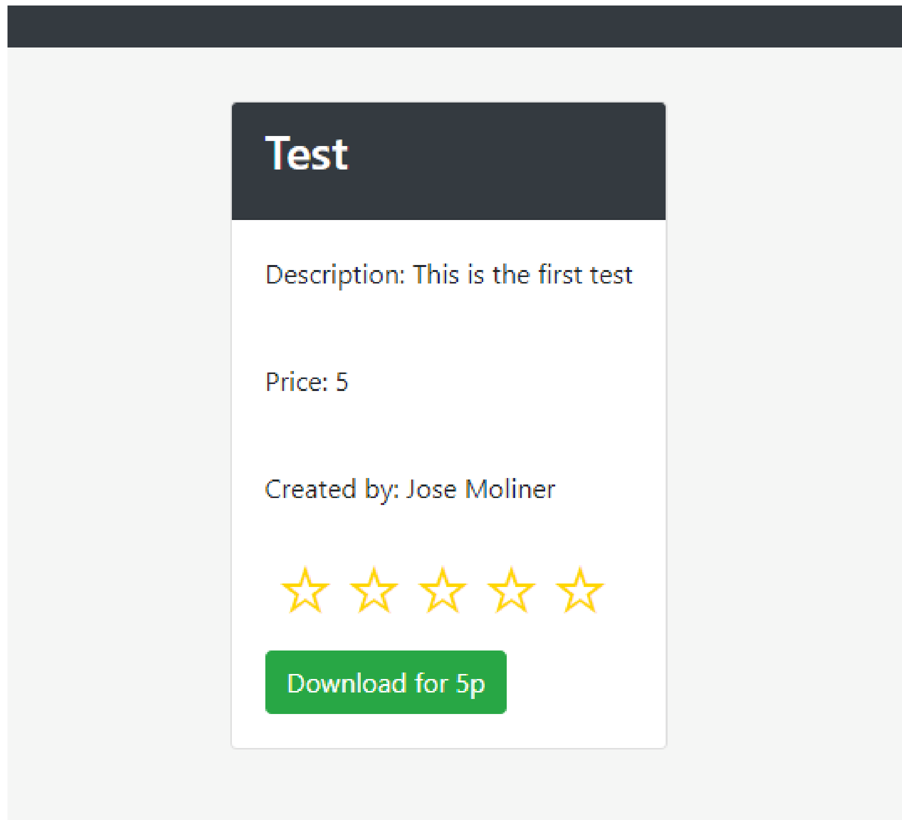

Fig. 6.7.1 File detail of the web page [23]

```
controller.file_detail_get = async (req, res, next) => {
  // First we search for the file we want to
  const { id } =req.params;
  const file = await pool.query('SELECT * FROM files WHERE id = ?',
                                                              [id]);
  console.log(file[0].id);
  console.log(req.user.user_id);
  // Then we search the rating that the user logged in put to
                                                           this file
  const rating = await pool.query('SELECT * FROM files_rating WHERE
      file_id = ? AND user_id = ?', [file[0].id, req.user.user_id]);
  console.log(rating);
```

```
        // We look for who is the user that created the file
        const createdBy = await pool.query('SELECT * FROM users WHERE
                                           id = ?', [file[0].user_id]);
        console.log(createdBy);
        // We look for who is the user that is logged in
        const userParams = await pool.query('SELECT * FROM users WHERE
                                           id = ?', [req.user.user_id]);
        // We save all this data in an object
        const fileData ={
            id: file[0].id,
            name: file[0].name,
            description: file[0].description,
            user: `${createdBy[0].name} ${createdBy[0].surname}`,
            price: file[0].price,
            stored_file: file[0].stored_file,
            userPoints: userParams[0].n_points,
        };
        if(rating.length>0){
            fileData.rating = rating[0].rating;
        };
        res.render('file_detail', { file: fileData});
      };
```

Lis. 6.7.2 file_detail_get controller [23]

The route path that takes the application to this function is something like: *'/user/file/:id'*, where *':id'* is a number that identifies the file searched for. So, the first thing done, is to get this id and save it in a variable.

Then, this *id* is searched in the database and, if it is found, the file data associated to it is saved in the new *file* variable.

Then we do the same for the ratings table of the database but, in this case, we have one more condition: *req.user.user_id*. This is one of the global variables that were created in the *app.js* file. Inside it, a number that identifies the user is stored and with its value and the *file id* value, the rating data will be found in the appropriate table (if it already exists). It is useful for rendering how many stars the user put to the file the last time he visited it.

Next, the system looks and saves in different variables for the user that created this and the data of the user that is logged in file in the *users* table.

When all data required has been stored, a new object is created with all of it.

The function will finish rendering a page of the *views*[13] folder, that will show the client all the data stored when it passes the new object created (Lis. 6.7.2).

---

[13] *Views* folder is introduced in chapter 6.8.

After this controller, there is a new POST controller. Its main function is going to be saving the rating from the user logged in:

```
controller.file_detail_post = async (req, res, next) => {
  //We use this post method specially for the new rating
  //First we get the file id and the new rating
  const { id } = req.params;
  const urlParams = new URLSearchParams(req.params.rating);
  const rating = urlParams.get('rating');
  // And we save them in an abject
  const newRating = {
      file_id: id,
      user_id: req.user.user_id,
      rating:rating
  };
   // We seach if the rating already exists and, if it exists, we update
                                     the rating and, if it doesn't, we insert it
  const existsRating = await pool.query('SELECT * FROM files_rating
              WHERE file_id = ? AND user_id = ?', [id, req.user.user_id]);
  if(existsRating.length>0){
      await pool.query('UPDATE files_rating set ? WHERE file_id = ? AND
                            user_id = ?', [newRating, id, req.user.user_id]);
  }else{
      await pool.query('INSERT INTO files_rating SET ?', [newRating]);
  };
  // Then we calculate the average of the ratings stored in
                                    'files_rating' table for this file
  const totalRatings = await pool.query('SELECT * FROM files_rating
                                                WHERE file_id = ?', [id]);
  let ratingAverage = 0;
  for(i=0; i<totalRatings.length; i++){
      ratingAverage += parseInt(totalRatings[i].rating);
  };
  ratingAverage/=totalRatings.length;
  // We save the average in an object
  const newFileRating = {
      rating: Math.round(ratingAverage)
  };
  // And we update the average(by default, it is 0)
  await pool.query('UPDATE files set ? WHERE id=?',[newFileRating,id]);
  res.send('rated');
};
```

Lis. 6.7.3 file_detail_post controller [24]

How this controller's route is called is a bit different as the others but the way it works is almost the same.

The main difference is found at the start of the code: instead of getting the necessary data to work from the body, it is obtained from the URL (*param*[14]). After saving this data in different variables, it is stored in a new object called *newRating*.

The next step is to make sure that this user had not rated this file before. If he did it, the rating is updated and, if not, it is saved as a new rating.

The last part of code basically updates the average rating of the file. It takes all the ratings of *files_rating* the table where the *file_id* corresponds the one of the file and calculates its average[15]. When it is done, the *round* method is used to have an integer and show how many *complete* stars each file has (Lis. 6.7.3).

6.7.3. File create POST

An interesting part of code is the POST method for creating a file, as it has the main structure of any basic POST method and it contains other interesting functionality:

```
controller.file_create_post = (req, res, next) => {
    const saveFile = async() => {
    const fileUrl = randomName();
    const sqlCount = await pool.query(`SELECT COUNT(*) AS count FROM
                         files WHERE stored_file LIKE "%${fileUrl}%"`);
    if(sqlCount[0].count > 0) {
        saveFile();
    } else {
      const fileTempPath = req.file.path;
      const ext = path.extname(req.file.originalname).toLowerCase();
      const targetPath=path.resolve(`public/upload/${fileUrl}${ext}`);
      if(ext === '.dwg' || ext === '.blend' ||...){
          await fs.rename(fileTempPath, targetPath);
          const { filename, description, price } = req.body;
          const name = filename;
          const stored_file = lastItem(targetPath);
          const user = await pool.query('SELECT * FROM users WHERE
                                           id=?', [req.user.user_id]);
```

[14] How this data si created and sent is explained in chapter 6.9.

[15] Obviously, as in other parts of the code, there are many ways to do this function more efficiently, but this is the most understandable.

```
            const newFile = {
                name,
                description,
                price,
                stored_file,
                user_id: user[0].id
            };
            await pool.query('INSERT INTO files set ?', [newFile]);
            const newFileId = await pool.query('SELECT * FROM files ORDER
                                                    BY id DESC LIMIT 1');
                for(let i=0; i<10; i++){
                    if(req.body['cbox'+i]!=undefined){
                        const newTagInstance = {
                            name: req.body['cbox'+i],
                            file_id: newFileId[0].id
                        };
                        await pool.query('INSERT INTO tags set ?',
                                                       [newTagInstance]);
                }};
                req.flash("success", "File created succesfully");
                res.redirect('/user/file/list');
            } else {
                await fs.unlink(fileTempPath);
                res.status(500).json({error: 'Invalid file'});
            }
        }
    }
    saveFile();
};
```

Lis. 6.7.4 file_create_post controller [25]

In this part of code, a new asynchronous function called `saveFile` is created and then, a new random name (created by an internal function of the *helpers* folder]) is given to the file uploaded to store the file. But… what happen if this random name is already taken? This is the reason why the `saveFile` function was created. If this random name is already on use, the function is called another time and it try to take a new random name.

Once the file has a unique name, the application saves it in the *'/public/upload/temp'* folder and starts to create a new path to store the new file. If the file type is valid (it is checked by comparing its extension with the valid ones [28][29]), the function continues executing. If not, a new file with a valid extension is required and the file is removed from the *'temp'* folder.

Finally, the controller has two more basic actions to do: creating a new file instance in the *'files'* table of the database and adding the pertinent tags to it in the *'tags'* table (Lis. 6.7.4).

6.7.4. Files get

This type of controller is a very basic example to rendering database information in a web page:

```
controller.files_get = async(req, res, next) => {
   const files= await pool.query('SELECT * FROM files WHERE user_id = ?
                 ORDER BY created_at DESC LIMIT 10',[req.user.user_id]);
   res.render('file_list', { files: files, timeago: timeago });
};
```

Lis. 6.7.5 files_ get controller [26]

Basically, the function does the following: first it saves in an array variable called *files* all the data of the last 10 files the user created and then it passes to the engine template this array and a function `timeago`[16], required before and useful to show the date in a custom format (Lis. 6.7.5).

6.7.5. Search controller

Another interesting controller is the one used in the *search* page:

```
controller.search_filtering = async (req, res, next) => {
  // Transform the url into an object with its values
  const urlParams = new URLSearchParams(req.params.restrictions);
  const name = urlParams.get('name');
  const orderBy = urlParams.get('orderBy');
  //We get the tags
  let tags='';
  let j=0;
  //First we create the query for the tags table
  for(let i=0; i<10; i++){
      if(urlParams.get('tag'+i)){
          if(j===0){
              tags+='name = "'+urlParams.get('tag'+i)+'"';
          }else{
              tags+=' OR name = "'+urlParams.get('tag'+i)+'"';
          }
          j++;
      };
  };
  // We initialize the query for the files table
  let idFilesTagged ='';
```

[16] Function created in *helpers* folder. It uses the *timeago.js* module, introduced in chapter 6.2.

```
    //If we have created a tags query, we do the query, looking for files
                                                 with the tags selected
    if(tags.length>0){
        const queryTags= 'SELECT DISTINCT file_id FROM tags WHERE '+tags;
        const taggedFile = await pool.query(queryTags);
        //Once we have the different file_id, we create the query to look
                                          for these id in the files table
        let l=0;
        for(let k=0; k<taggedFile.length; k++){
            if(l===0){
                idFilesTagged+='AND (id = "'+taggedFile[k].file_id+'"';
            }else{
                idFilesTagged+=' OR id = "'+taggedFile[k].file_id+'"'
            };
            l++;
        };
        idFilesTagged+=')';
    };
    // Querying data ordered by the restriction
    const result = await pool.query('SELECT * FROM files WHERE name LIKE
     "%'+name+'%"'+ idFilesTagged +' ORDER BY '+orderBy+' DESC LIMIT 10');
    res.send(result);
};
```

Lis. 6.7.6 search_filtering controller [27]

First, the restrictions for filtering from every file in the database are taken from the URL path (how these restrictions are stored in the URL will be explained later). The *name* and the *order* are taken simply with the get method, and the tags too, but with the small difference that, since there can be more than one tag through which to filter, there will have to iterate through each one of them (with a maximum of 10 different tags). Then, with the addition of each tag, a string is created with a format valid to make a MySQL query later (‘*name = tag1 OR name=tag2 OR ... OR name=tagN*’). [30]

But the *tags* string for querying is not ready yet. If the previous string was created, it can be stated that the user wants to filter also by tags. So, the application decides to prepare the final *tags* string to query.

First of all, it selects all the files that have one of the tags required, making a query to the database using the string created in the loop. And, once it has stored these files, it creates a new string that will be used later to filter in the *files* table by their *id* (*‘AND (id = file-with-required-tag-1 OR id = file-with-required-tag-2 OR ... OR id = file-with-required-tag-N)’*).

After that, the new query can be executed, using all the restrictions provided by the client (filtering by name and tags, and ordering the results by the order provided) (Lis. 6.7.6).

Subsequently, some controllers from the *usersController* folder will be studied, since they are very different from the controllers seen until now.

6.7.6. User new password POST

```
controller.user_newpassword_post = async(req, res, next) => {
  const {email} = req.body;
  const newPassword = randomstring.generate(10);
  const newEncryptPassword = await encryptPassword(newPassword);
  await pool.query('UPDATE login SET password = ? WHERE email = ?',
                                            [newEncryptPassword, email]);
  const html = `This is your new password:
  </br>
  ${newPassword}`
  // Send email
  await mailer.sendEmail('admin@finalproject.com', email, 'Password
                                                        recovery', html);
  res.redirect('/authentication/signin');
};
```

Lis. 6.7.7 user_new_password controller [28]

The web application calls this part of code when the user needs to log in but he do not remember his password. This controller receives from the body an e-mail and it triggers two actions, that can be identified by two new functions that had not appeared yet: `randomstring.generate()` and `mailer.sendEmail()`. First, a new password is created by the `randomstring` module and encrypted by the `encryptPassword`[17] function. Then, the encrypted function is stored in the database.

After that, a constant called *html* is created, with a string that contains the value of the password before it was encrypted. This html will be the body of an e-mail that will be sent by the `sendEmail` method of the `mailer` object (Lis. 6.7.7).

6.7.7. User create and signin POST

This is the part of code that will handle of the *sign up* functionality of the web application.

```
controller.user_create_post = passport.authenticate('local.signup', {
      successRedirect: '/authentication/logout',
      successMessage: true,
      failureRedirect: '/authentication/signup',
      failureFlash: true
});
```

Lis. 6.7.8 user_create_post controller [29]

[17] Go to chapter 6.10. for more information about the function performance

The first difference that can be noted here, is the fact the fact that it is not a function that takes the `req`, `res` and `next` objects as arguments. It executes a passport method directly (Lis. 6.7.8).

```
controller.user_signin_post = (req, res, next) => {
  passport.authenticate('local.signin', {
      successRedirect: '/user/profile',
      failureRedirect: '/authentication/signin',
      failureFlash: true
  })(req, res, next);
};
```

Lis. 6.7.9 user_signin_post controller [30]

The structure for the sign in of the client has a similar structure, and it could be coded in the same way of the previous method, as his internal functionality is very similar.

In both, can be appreciated the calling of the authenticate method, with a different strategy in each case. For the first one, it is required the *'local.signup'* strategy and, for the second one, the *'local.signin'*[18].

Once the correspondent strategy execution has finished, two scenarios can arise: success or failure. If there are no errors and it finish with success, the method will redirect to one route and, in the opposite case, it will redirect to another route combined with flash error messages in order to display status information to the user[19] (Lis. 6.7.9).

6.8. Views

The views (templates) are stored in the */views* directory (as specified in *app.js*) and are given the file extension *'*`.pug`*'*. These templates will be called in the routes files and will be the ones which handle with the rendering of our information in the screen of the client.[31]

A template processor is software designed to combine templates with a data model to produce result documents. The language in which templates are written is known as a template language or template language[20].

All these files can be used thanks to the *app* configuration[21]. Previously, *app* configuration was told that, in case that it was needed to search for an engine template, it looked in this folder among the '`.pug`' extension files.

There are some *'*`.pug`*'* templates in this project much more difficult to understand than others. In order not to excessively lengthen this section, only two templates will be explained in detail.

---

[18] Strategies of the `passport` function, explained in chapter 6.10.

[19] Using *flash messages* requires a `req.flash()` function, which was configured in the *app.js* file.

[20] This language was selected before. Go to chapter 4.3. for more information.

[21] Go to chapter 6.3. for *app.js* configuration.

6.8.1. User profile

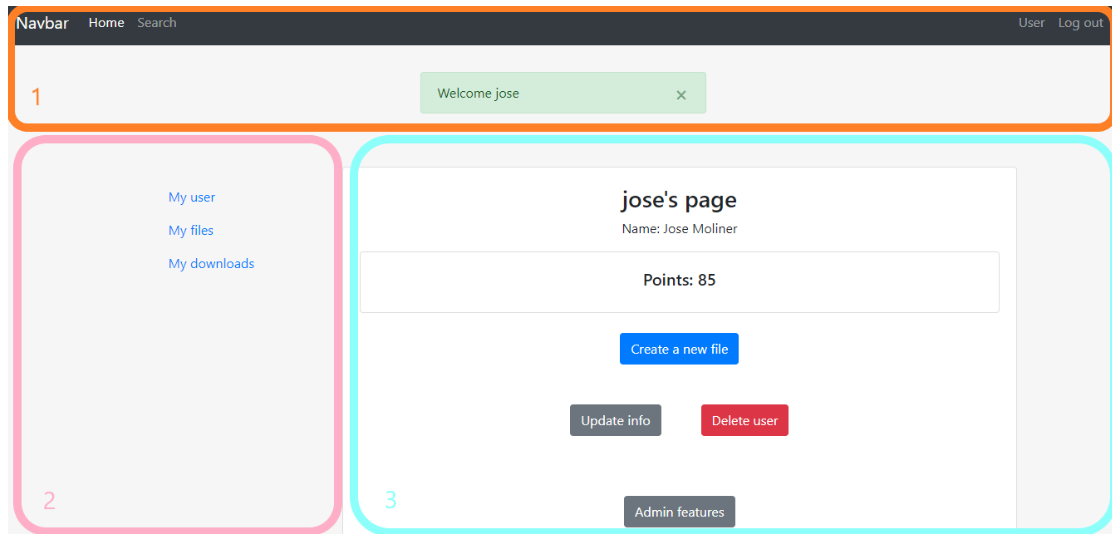

Fig. 6.8.1 Users' profile web page [28]

```
extends layout

block content
  .container.p-4
     .row
        .col-md-3
           include menu.pug
        .col-md-9.mx-auto
           .card.text-center
             .card-body
               h3 #{user.username}'s page
               p Name: #{userdata.name} #{userdata.surname}
               .card.text-center
                 .card-body
                    h5 Points: #{userdata.n_points}
               a.btn.btn-primary.m-4(href="/user/file/create")
                                                    Create a new file
               br
               a.btn.btn-secondary.m-4(href="/user/update") Update info
               a.btn.btn-danger.m-4(href="/user/delete"
                             onclick=`confirmDeleteUser();`) Delete user
               if(admin)
                  br
                  br
                  a.btn.btn-secondary.m-4(href="/user/admin") Admin
                                                               features
```

Lis. 6.8.1 user profile pug file [31]

Here we have both, the result, and the code of the *users' profile* web page. Let us study the code and watching what part of the web page corresponds to it.

First, the `extends layout` statement appears. This is going to call to another template, the one with the name *layout*. This *layout* is like the main template. It is going to be called by any other one, and is going to fulfil three functions: rendering the navbar(identified with number 1 in the image) and the footer, displaying messages when it is required (in this case *'Welcome jose'*) and finally, configuring, calling and requiring all the scripts and other files needed to display the web page correctly, that is to say, configuring the *head* of the web page, as any other web page *html* file.

Then, the `block content` statement is used to indicate the start of the body of the file.

Then, lots of classes can be identified. These classes are *Bootstrap classes* and their unique function is to order the data in the page and make it look better [32]. But the interesting part comes in line 7, with the `include menu` statement. It works like `extends layout`, but inside of the body of the file. It calls another template file and render it in the correspondent column. It is identified in the image by the number 2.

After that, in the web page, we can see a card with some basic data of the user, like the username or the number of points he owns. This data can be accessed thanks to the controller that make this template renders. The controller passes to the template an *userdata* object (the *user* object can be accessed globally[22]) and it takes the necessary data and logs it in the page.

At the bottom of the file, we can see an *if* statement. It is saying to the template that he can only execute (and render) this part of code in case of the *admin* property (passed from the controller) is true (Lis. 6.8.1).

6.8.2. Search

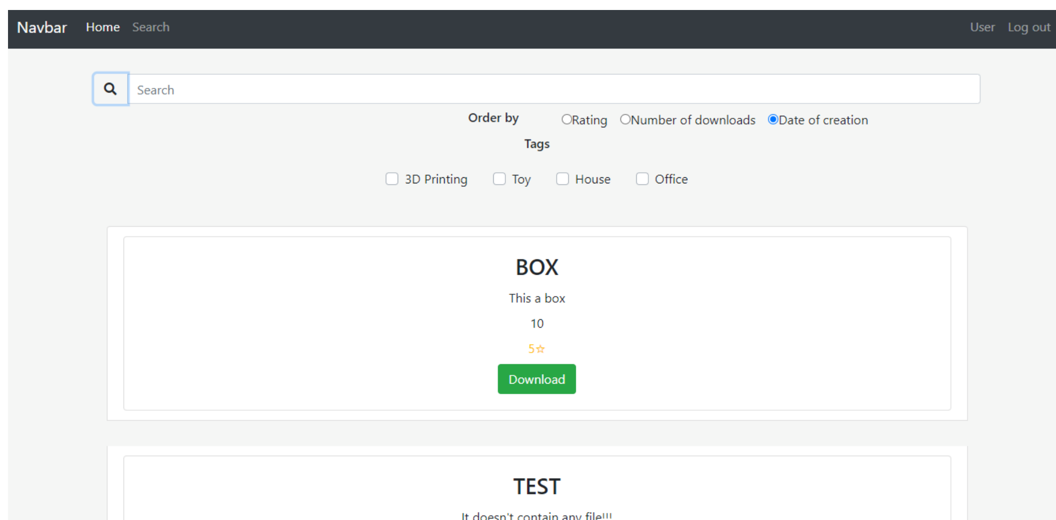

Fig. 6.8.2 Searching web page [29]

[22] As explained in the *app* configuration (chapter 6.3), the application can access to the login data of the user at any time

Probably, the search page was the hardest part of the project to implement. It combines many of the technologies used in other sections to render something like the previous image.

To understand how it works, it is necessary study not only the *'pug'* template, but also all the scripts related to it. Let us start with the template rendered and in the next section we will crumble the dynamic part of the page.

```
extends layout

block content
  .container.p-4
    .row
      .input-group
        .input-group-prepend.mx-auto
          button.btn#searchButton
            i.fas.fa-search(aria-hidden="true")
          input.form-control.w-100.my-auto(type="text" name="search" id
           ="search" placeholder="Search" aria-label="Search" autofocus)
    .row
      .col.m-2.text-right.my-auto
        h6 Order by
      .col.m-2.my-auto#order
        label.radio-inline.m-2
          input(type='radio', name='order', value='rating', checked='')
          | Rating
          label.radio-inline.m-2
          input(type='radio',name='order',value='downloads',checked='')
          | Number of downloads
          label.radio-inline.m-2
          input(type='radio',name='order',value='createdat',checked='')
          | Date of creation
    .row
      .col.m-8.mx-auto.text-center
        h6.d-block Tags
        -let i=0
          each val in tags
            .custom-control.custom-checkbox.d-inline-block.m-3
              input.custom-control-input(type="checkbox" id=`cbox${i}`
                                                    value=val.name)
              label.custom-control-label(for=`cbox${i}`)=val.name
              -i++
    .row
      ul.list-group#list.w-100.mx-auto.my-3
```

Lis. 6.8.2 searching pug file[32]

The first thing done is, like in any other template, *extend layout*[23].

Next, it can be appreciated that the page is structured in four rows.

The first three have the same function. They are used to let the user select the restrictions he wants to make the query. First row is for selecting the name by which the client wants to filter, second one to select the order of displaying these files and third one to look for files that had associated some tags. All these works as any other *input button*. Some data is given by the user and it is sent to some place where it can be used. But this place is not only a controller. Before sending the data to it, *jQuery* is used in the *JavaScripts* files.[32][33]

The last row is the one that is going to handle with the rendering of the different files that accomplish with the restrictions given (Lis. 6.8.2).[24]

6.9. Public

The public folder is where will be saved almost all the files that handle with the front-end part of the web application. Here we can find the *images* folder, which will contain all the images used in the website, the *styles* folder, which will contain the CSS code used to style the content[25], the *upload* folder[26] and the *JavaScripts* folder which will contain all the JavaScript code used to add interactive functionality to the site (e.g. buttons that load data when clicked). We will pay special attention to the latter, since it is necessary to know the operation of the files in this folder to understand what happens to the website and why.

6.9.1. *javascripts/search*

To continue with the comprehension of the previous section and all the *search* page, can be met something like the following:

```
//Functionality of Search Button when clicked
$('#searchButton').click(function(){
  //We get the data in the input field
  const restrictions = {
      name: $('#search').val(),
      orderBy: $('input[name=order]:checked', '#order').val()
  };
  // For tags, we iterate only through 10 because there will be a
                                                  maximum of 10 tags
```

[23] That is, include the actual template in a main template, as explained in this chapter before.

[24] How it works is explained in *JavaScripts* section.

[25] Although, in this case, this is not going to be very large since, as previously mentioned, bootstrap will mainly be used to stylize the project.

[26] Go to chapter 6.7 for more information

```
    for(i=0; i<10; i++){
        if($(`input[id=cbox${i}]:checked`).val()!=undefined){
            restrictions['tag'+i]=$(`input[id=cbox${i}]:checked`).val();
        };
    };
    // We transform the data into a string
    const urlRestrictions = $.param(restrictions);
    console.log(urlRestrictions);
    // We redirect to a route where it is used
    $.get('/search/'+urlRestrictions)
    //When finished, we get the data
        .done(data => {
            console.log(data);
            // And display it in the 'search' view
            setList(data);
        });
});
```

Lis. 6.9.1 javascript search file [33]

This type of language is a little bit different from the JavaScript we are used to. That is because it is jQuery. jQuery is a multi-platform JavaScript library that allows us to simplify the way of interacting with HTML documents, manipulate the DOM tree, handle events, develop animations and add interaction with the AJAX technique to web pages. And this is what it is used for in the code. [34][35]

This part of code is executed once the element with the *searchButton* id is *clicked*. First, it takes all the information given by the user in the *search* page and saves it in an object (the *name*, the *orderBy* and the *tags*, in an array inside the object). After that, it uses the *param* method to transform this object into a string, which will be part of the URL path. This path is passed with the GET method and provokes the calling of the appropriate controller. Once the controller has finished and has sent the response (*data*), the function calls the `setList` function (Lis. 6.9.1).

```
//Getting the container where my searched items will be displayed
const list = document.getElementById('list');
// Create the list that match the criteria given
function setList(group){
  //First we clear the possible previous list
  clearList();
  //If there are no items with the requirements, send 'No results' item
  if(group.length === 0){
      setNoResults();
  //Creating the list
  }else{
```

```
        for(i=0; i<group.length; i++){
            createItems(group);
        }
    }
  }
  // Remove all items of the list after a search
  function clearList(){
    while(list.firstChild){
        list.removeChild(list.firstChild);
    }
  }
  // If there are no results... ==>
  function setNoResults(){
    const item = document.createElement('li');
    item.classList.add('list-group-item');
    const text = document.createTextNode('No results found');
    item.appendChild(text);
    list.appendChild(item);
  }
  //Function to create the items of the list
  function createItems(group){
    //Creating elements
    const item = document.createElement('li');
    . . .
    const buttonItem = document.createElement('a');
    //Adding classes and attributes to elements
    item.classList.add('list-group-item', 'm-3');
    . . .
    //Adding text to elemets
    const name = document.createTextNode(group[i].name);
    . . .
    //Nesting elements
    nameItem.appendChild(name);
    . . .
  };
```

Lis. 6.9.2 javascript search file [34]

The `setList` function is the *'mother'* of other three: `clearList`, `setNoResults` and `createItems`. The functionality of each one is very easy.

The `clearList` function only has one action. It is going to use the method `removeChild` to delete all files as long as one exists. It is called every time the `setList` function is called.

The `setNoResults` function will execute if any file is found with the specific requirements. It is going to create a list item and add it some classes to communicate the user about the inexistence of files with the restrictions provided.

Finally, the `createItems` function oversees creating the correspondent elements, and giving him the necessary classes, attributes, and texts (Lis. 6.9.2).

This function was called in the '*search.pug*'[27] file, and it will fill the web page with the information located in the database.

6.9.2. *javascripts/rating* and *javascripts/scripts*

These files are very similar to the previous since a great part of code uses jQuery to.

Here can be found functions that will handle with some specific tasks of many pages.

For example, the *rating* file is the part of code that will let the showing of the rating run [35][36]. This JavaScript complements the *file detail* page, taken as an example for the controllers to display the stars that the current user gives to a specific file.

Inside the *scripts* file, can be met some function like:

```
function confirmDeleteFile(id){
  const response = confirm('Are you sure you want to delete it?');
  if(response){
      location.href='/user/file/'+id+'/delete';
  }
};
```

Lis. 6.9.3 javascript confirmDeleteFile function [35]

It is going to show a new confirm window when the user wants to delete a file. It is used to be sure that any user does not delete a file by error.

Therefore, as can be seen, this type of files will not interact with the database or with the internal operation of the application. However, these scripts will be really useful when obtaining and managing certain information provided by the user and when displaying other data at certain times (Lis. 6.9.3).

6.10. Helpers

The *helpers* folder is the place where can be found many of the code that will help with the correct performance of some functions. The functions that are here will be required from other files and will be part of their code. The reason for the existence of the *helpers* folder is only to keep the code organized, since these functions would fit perfectly into other existing files.

6.10.1. *helpers.js*

This file is the sibling of the scripts file, inside the public folder. On it will be found many kinds of reusable functions in different places of our code.

For example, we have some code that will work with passwords, which is an important section of the web application.

---

[27] Introduced in chapter 6.8.

As you might imagine, user passwords cannot be stored directly in the database. This would put their security in danger since they would be very easily accessible by hackers. It is for this reason that when storing passwords and private information, it must be encrypted, and the maximum number of barriers must be putted to prevent them from being accessible by external agents.

To understand why these functions are necessary, we must know that encryption is the practice of scrambling information so that only someone with a corresponding key can decrypt and read it. To encrypt data, an algorithm is used, in other words, a series of well-defined steps that can be followed procedurally. The algorithm can also be called the encryption key. But… how this algorithm works?

There are infinite algorithms to encrypt passwords, but basically what they do is to take the password without encryption and through a series of steps, change the set of all the characters for totally different ones, so that, if you do not know what operations have been submitted to the initial string, it is practically impossible to reach the end (encrypted string).[36]

Of course, if you need to get the original password from ciphertext, you need to know the first algorithm or, at least, know the correspondent method to transform this particular encryption to the initial string in plain text.

Fig. 6.10.1 Encryption schema [30]

The problem is that we do not need our application to return the initial password, which was entered by the user. We only need to verify that if the user wants to log in, he enters the password that was entered when signing up. Therefore, we do not need to decrypt it. This is where the concept of hashing comes into play.

Hashing is the practice of using an algorithm to assign data of any size to a fixed length. This is called a hash value. And although it is possible to return to the initial string after hashing, it is not usually profitable. Therefore, we say that hashing is a one-way function.

Now, whereas encryption is meant to protect data in transit, hashing is intended to verify that a file or piece of data has not been altered, that it is authentic. In other words, it serves as a checksum. The interesting thing of hashing is that no two files can create the same hash value, so any alteration – even the tiniest tweak – will produce a different value[28].

---

[28] With this, what is meant is that if two different files produce the same unique hash value this is called a collision and it makes the algorithm essentially useless.

Fig. 6.10.2 Hashing schema [31]

And then, to help us improve security, we have salting.

Salting is a concept that generally belongs to password hashing. Essentially, it is a unique value that can be added to the end of the password to create a different hash value. This adds a layer of security to the hashing process, specifically against brute force attacks[29]. By salting your password, you are essentially hiding its real hash value by adding an additional bit of data and altering it.

Now it will be very easy to understand, for example, what this part of code does:

```
// Helper function to encrypt passwords
helpers.encryptPassword = async (password) => {
  const salt = await bcrypt.genSalt(10);
  const hash = await bcrypt.hash(password, salt);
  return hash;
};

// Function to verify if the password matches
helpers.matchPassword = async (password, savedPassword) => {
  try {
      return await bcrypt.compare(password, savedPassword);
  } catch (e) {
      console.log(e);
  }
};
```

Lis. 6.10.1 Helper functions [36]

The `encryptPassword` basically uses the `genSalt` and the `hash` methods to transform the password given in a string that is secure to stored, as has been explained before.

The `matchPassword` function only compares the password given when the user logs in, with the password of the current user(`savedPassword`) and returns the result. In case there is any error, the application logs it by the console.

[29] A brute force attack is when a computer or botnet tries every possible combination of letters and numbers until the password is found.

And, as these functions, there are others used for creating a random name for the files, for working with dates, for taking and splitting some parts of an array or for checking if the user logged in is an admin user (Lis. 6.10.1).

6.10.2. *auth.js*

This file contains an object with methods used to allow access to certain routes of some controllers. For example, we do not want to allow a user to access the sign in page if they are already logged in. Basically, they are barriers that only the users that we decide can cross.[37][38]

```
// Method to verify if the user is active
  async isActive(req, res, next) {
      const { username } = req.body
      const login= await pool.query('SELECT * FROM login WHERE username
                                                        = ?', [username]);
      if(login.length !=0){
          if(login[0].active===1){
              return next();
          }else{
              req.flash('message', 'Verify your e-mail');
              return res.redirect('/authentication/signin');
          };
      }else{
          req.flash('message', 'This username does not exist');
          return res.redirect('/authentication/signin');
      };
  }
```

Lis. 6.10.2 Authentication functions [37]

This method will be useful to verify if the user has already verified his e-mail account. This function basically accesses the database and queries the *users* table, to check if the user who has tried to log in has already been activated or not. Once verified, we have three possible scenarios: that the user cannot be found, that it is not active or that it is. For the first two situations, the relevant messages will be displayed on the screen. In case it is an already activated user, it will be allowed to continue with the following function (*next*) (Lis. 6.10.2).

```
  //Function to verify if the user is logged in
  isLoggedIn(req, res, next) {
      if(req.isAuthenticated()){
          return next();
      } else {
          return res.redirect('/authentication/signin');
      }
  },
```

Lis. 6.10.3 isLoggedIn function [38]

Apart, two functions can be accessed also to check whether users are logged in or not, since some routes are protected only for users who are logged in and others for unlogged clients. The previous one is the `isLoggedIn` function. Basically it uses the `req.isAuthenticated` method, which will return a Boolean value, depending on whether or not he is logged in (Lis. 6.10.3).

6.10.3. *mailer.js*

Nodemailer is a module for Node.js applications that allows sending e-mails easily. It is the solution that most Node.js users turn to by default.[39][40]

```
const nodemailer = require('nodemailer');
const { mailer } = require('../keys');

const transport = nodemailer.createTransport({
  service: 'Mailgun',
  auth: {
      user: mailer.MAILGUN_USER,
      pass: mailer.MAILGUN_PASSWORD
  },
  tls: {
      rejectUnauthorized: false
  }
});

module.exports = {
  sendEmail(from, to, subject, html) {
      return new Promise((resolve, reject) => {
          transport.sendMail({from, to, subject, html}, (err, info)=>{
              if(err) reject(err);

              resolve(info);
          });
      });
  }
};
```

Lis. 6.10.4 Mailer functions [39]

First, the transport object is created. With the `createTransport` method, the basic characteristics of our service provider for sending e-mails are defined (Lis. 6.10.4).

For this project, it was decided to use *mailgun*. Its free version only allows sending emails to a maximum of 5 users who must be selected previously. However, it is enough to show how it works, and it provides a solid technical service to help with possible problems.

The next step is exporting the `sendEmail` function, which will use the `sendMail` method of *nodemailer* to send the emails with the appropriate remitter, addressee, subject and body. Before it, the *new Promise* statement is used to be able to use it in an asynchronous function[30].

6.10.4. *passport.js*

Passport is an authentication middleware for *Node.js*. It is very flexible and modular and can be unobtrusively dropped in to any *Express-based* web application. There are a comprehensive set of strategies, supporting authentication using a username and password (which is the one used by this project) or other methods.[41][42]

In this file, basically two new strategies are created for login and user creation, which will be used later in the corresponding controllers[31]. Let us try to understand how the `local.signup` strategy works.

```
passport.use('local.signup', new LocalStrategy({
  usernameField: 'username',
  passwordField: 'password',
  passReqToCallback: true
}, async (req, username, password, done) => {
  const { email, name, surname, birthday, country } = req.body;
  const newUserLogin = {
      username,
      password
  };
  newUserLogin.password = await helpers.encryptPassword(password);
  const newUser = {
      name,
      surname,
      birthday,
      country
  };
  const resultUser = await pool.query('INSERT INTO users SET ?',
                                                              [newUser]);
  newUserLogin.user_id = resultUser.insertId;
  newUserLogin.email = email;
  // Generate random token
  const secretToken = randomstring.generate();
  newUserLogin.secretToken = secretToken;
```

[30] The `promissify` module could also be used for this purpose

[31] Go to chapter 6.7. for more information.

```
    // Active equals 0
    newUserLogin.active = 0;
    // Compose email
    const html = `Welcome to my project!
    </br>
    Thank you for registering
    </br>
    </br>
    Please, verify your account by clicking <a href="http://final-
project-pk.herokuapp.com/authentication/verify/${secretToken}">here</a>`
    // Send email
    await mailer.sendEmail('admin@finalproject.com', email, 'Email
                                                   verification', html);
    const resultLogin = await pool.query('INSERT INTO login SET ?',
                                                        [newUserLogin]);
    newUserLogin.id = resultLogin.insertId;
    return done(null, newUserLogin);
  }));
```

Lis. 6.10.5 Passport functions [40]

First, the local strategy configuration is set up. The local authentication strategy authenticates users using a username and password. The strategy requires a verify call-back function, which accepts these credentials and calls done providing a user.

After that, the password passed by the user is encrypted with the function previously created and the user data is inserted in the *users* table, in a new row.

Then, for the *login* table we are going to insert a new row, for the new user. Apart from the *username*, the *password,* and the e-*mail* fields, it will contain the *active* field (stored with a value of 0 at the beginning) and a *secretToken* field. This token is created to let the user verify his email, as a unique URL is going to be sent to his email, with this *secretToken* on it.

Finally, the email for verifying the account is created and sent, the user's credentials are saved in the *login* table, and the id of the new user is stored as a property in the *newUserLogin* object, which will be passed as the function result later. This last step is used to serialize and deserialize the user. The user id (provided as the second argument of the done function) is saved in the session and is later used to retrieve the whole object via the `deserializeUser` function (Lis. 6.10.5).

```
passport.serializeUser((user, done) => {
  done(null, user.id);
});
passport.deserializeUser(async (id, done) => {
  const rows = await pool.query('SELECT * FROM login WHERE id = ?',
                                                              [id]);
  done(null, rows[0]);
});
```

Lis. 6.10.6 Passport serialize and deserialize user functions [41]

Basically, the `passport.serializeUser` determines which data of the user object should be stored in the session, and `passport.deserializeUser` is getting id from the cookie, which is then used in a call-back function to get user info, based on that id or some other piece of information from the cookie (Lis. 6.10.6).

7. Future work and conclusions

To sum up, it has been proved that the web application is fully functional, as it is capable of performing all initial actions that were proposed during the design stage. With this project, we have obtained a web application with the following main actions:

- User registration and login, using a username and a password, which will be stored securely in the database.
- 3D files uploading, assigning their corresponding name, description and price, in addition to having the option of adding tags, depending on the users' preferences.
- Downloading of uploaded files.
- A payment system through virtual points created by the web application itself.
- Searching and filtering of files through a search box.

The process to acquire the required knowledge to achieve this aim and to have an appropriate background to start coding was the hardest part. As it has been shown, it is not a fully optimized code. However, it is very didactic and, with it, the performance of each one of its sections is perfectly understandable.

The steps to follow the whole project were basically four:

- Study of how websites work, specifically those for uploading and downloading files, how information is stored on them and how do the different functions they need work.
- Compilation of information on the different tools that can be found on the market for free, and selection of the most suitable for this project.
- Organization of ideas and the structure of the different parts into which the web application can be divided.
- Development of the source code of the web application, studying each one of the main actions and deciding the best way to implement them.

However, like any web development project, it is not fully finished. It is true that it has all the required functional part satisfied, yet the result cannot be considered the final application, so further work will be necessary for its improvement.

To do so, a professional team should work on it in order to improve its final design and security before users can make use of it. This would be of utmost importance since now, both, the files' and the users' information, are being unsafely uploaded to the database and it would not be very difficult for an external agent to access them.

Furthermore, as stated in the first chapters, a profound research on the existing competition, obstacles and possible development paths should be undertaken. Once studied, the best possible sector could be selected, and the brand could be adapted to its preferences.

References

[1] *Thingiverse* application (online – accessed: April 4th, 2020), Available online: http://blascarr.com/thingiverse/

[2] How the Internet Works? (online – accessed: May 1st, 2020), Available online: https://www.youtube.com/watch?v=7_LPdttKXPc

[3] How the Internet Works? (online – accessed: April 29th, 2020), Available online: https://developer.mozilla.org/es/docs/Learn/Common_questions/How_does_the_Internet_work

[4] What is a web server? (online – accessed: April 29th, 2020), Available online: https://developer.mozilla.org/es/docs/Learn/Common_questions/Que_es_un_servidor_WEB

[5] *HTTP* protocol (online – accessed: April 30th, 2020), Available online: https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto

[6] *Node.j*s (online – accessed: April 10th, 2020), Available online: https://www.w3schools.com/nodejs

[7] *Node.js* (online – accessed: April 12th, 2020), Available online: https://developer.mozilla.org/es/docs/Learn/Server-side/Express_Nodejs

[8] *Node.js* (online – accessed: April 17th, 2020), Available online: https://platzi.com/blog/lenguajes-frameworks-librerias-backend-2019/

[9] *Node.js* (online – accessed: June 3rd, 2020), Available online: https://openwebinars.net/blog/que-es-nodejs/

[10] *Express* (online – accessed: May 13th, 2020), Available online: https://expressjs.com/en/guide/routing.html

[11] *Pug* (*Jade*) (online – accessed: May 5th, 2020), Available online: https://www.sitepoint.com/a-beginners-guide-to-pug/

[12] Deploy to *Heroku* (online – accessed: June 5th, 2020), Available online: https://bezkoder.com/deploy-node-js-app-heroku-cleardb-mysql/

[13] Functional Requirements (online – accessed: May 22nd, 2020), Available online: https://en.wikipedia.org/wiki/Functional_requirement

[14] Non-Functional Requirements (online – accessed: May 22nd, 2020), Available online: https://en.wikipedia.org/wiki/Non-functional_requirement

[15] Functional and Non-Functional Requirements (online – accessed: May 22nd, 2020), Available online: https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/

[16] ER Diagrams (online – accessed: May 17th, 2020), Available online: https://www.guru99.com/er-diagram-tutorial-dbms.html

[17] UML Diagrams (online – accessed: May 18th, 2020), Available online: https://www.lucidchart.com/pages/es/que-es-el-lenguaje-unificado-de-modelado-uml

[18] UML Class Diagrams (online – accessed: May 18th, 2020), Available online: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

[19] UML Use Case Diagram (online – accessed: May 18th, 2020), Available online: https://sparxsystems.com/resources/tutorials/uml2/use-case-diagram.html

[20] UML Activity Diagram (online – accessed: May 19th, 2020), Available online: https://www.tutorialspoint.com/uml/uml_activity_diagram.html

[21] What is the *package.json* (online – accessed: May 27th, 2020), Available online: https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/

[22] Uploading files in Node.js (online – accessed: May 5th, 2020), Available online: https://www.geeksforgeeks.org/file-uploading-in-node-js/

[23] What is *express.json* and *express.urlencoded* (online – accessed: May 1st, 2020), Available online: https://stackoverflow.com/questions/23259168/what-are-express-json-and-express-urlencoded/51844327

[24] *Express* tutorial (online – accessed: May 23rd, 2020), Available online: https://vegibit.com/express-js-beginner-tutorial/

[25] *mysql* package (online – accessed: May 6th, 2020), Available online: https://www.npmjs.com/package/mysql

[26] *Node.js* and *MySQL* tutorial (online – accessed: May 5th, 2020), Available online: https://codeforgeek.com/nodejs-mysql-tutorial/

[27] Delete and upload rows in *MySQL* (online – accessed: May 8th, 2020), Available online: https://www.guru99.com/delete-and-update.html

[28] 3D CAD file formats (online – accessed: May 25th, 2020), Available online: https://all3dp.com/2/overview-of-3d-cad-file-formats/

[29] What files can be converted to *g-code* (online – accessed: May 5th, 2020), Available online: https://all3dp.com/2/g-code-converter-what-can-be-converted-to-from-g-code/

[30] Simple Search in *JavaScript* (online – accessed: May 20th, 2020), Available online: https://youtu.be/SWkPXbQXArk

[30] Simple Search in *JavaScript* (online – accessed: May 6th, 2020), Available online: https://html-to-pug.com/

[31] Forms example (online – accessed: May 16th, 2020), Available online: https://gist.github.com/danrovito/977bcb97c9c2dfd3398a

[32] Bootstrap (online – accessed: April 29th, 2020), Available online: https://getbootstrap.com/

[33] Buttons for *html* forms (online – accessed: May 21st, 2020), Available online: https://study.com/academy/lesson/adding-radio-buttons-checkboxes-lists-for-user-input-to-html-forms.html

[34] *Ajax* search-box using *Node.js* and *MySQL* (online – accessed: May 10th, 2020), Available online: https://codeforgeek.com/ajax-search-box-using-node-mysql/

[35] *Ajax* search-box using *Node.js* (online – accessed: May 10th, 2020), Available online: https://www.youtube.com/watch?v=6xEmZGHIHP4

[36] The difference between encryption, hashing and salting (online – accessed: May 6th, 2020), Available online: https://www.thesslstore.com/blog/difference-encryption-hashing-salting/

[37] *express-session* packages (online – accessed: May 14th, 2020), Available online: https://www.npmjs.com/package/express-session

[38] *Node.js* y *MySQL*, aplicación completa (online – accessed: May 1st, 2020), Available online: https://www.youtube.com/watch?v=qJ5R9WTW0_

[39] Site Authentication with *Node* (online – accessed: May 25th, 2020), Available online: https://www.youtube.com/watch?v=gzDB0ZGOjA0

[40] Nodemailer (online – accessed: May 29th, 2020), Available online: https://nodemailer.com/about/

[41] Building a *Node.js* web app using *passport.js* (online – accessed: May 25th, 2020), Available online: https://dev.to/gm456742/building-a-nodejs-web-app-using-passportjs-for-authentication-3ge2

[42] Passport authenticate (online – accessed: May 25th, 2020), Available online: http://www.passportjs.org/docs/authenticate/

[43] Bases of Web Development, online course (online - accessed: March 7th, 2020 – April 29th, 2020), Available online: https://www.codecademy.com/learn/paths/web-development

Summary in Polish

Projekt polegał na stworzeniu strony internetowej z podstawowymi funkcjami, takimi jak rejestracja i logowanie różnych użytkowników, przesyłanie i pobieranie plików obiektów 3D z odpowiednim rozszerzeniem oraz system wyszukiwania i filtrowania plików. Objął również takie funkcje, jak system skali oceny plików i przechowywanie statystyk klientów w celu poprawy komfortu użytkowania.

Ta strona internetowa, jak wspomniano wcześniej, skupiła się na przesyłaniu plików do pobrania, które mogą być rozpoznawane i wykorzystywane przez drukarki 3D, dzięki czemu są w stanie powielać wyżej wymienione projekty, tworzyć części lub modele wolumetryczne z zewnętrznego pliku wykonanego przez różnorodne oprogramowanie komputerowe.

Z drugiej strony głównym celem projektu jest zrozumienie podstaw tworzenia stron internetowych, jednego z obecnie najbardziej atrakcyjnych i pożądanych tematów, ponieważ są one w ciągłej ewolucji i mają świetną prognozę na przyszłość.

Najtrudniejszym procesem było zdobycie wiedzy niezbędnej do osiągnięcia tego celu i posiadania odpowiedniego zaplecza do rozpoczęcia kodowania. Prawdą jest, że nie jest to w pełni zoptymalizowany kod. Jest to jednak bardzo dydaktyczne, a dzięki temu każda części zostało wykonana w zrozumiały sposób.

Istniały cztery główne kroki przy tworzeniu projektu:

- Badanie, w jaki sposób działają strony internetowe, w szczególności te do przesyłania i pobierania plików, w jaki sposób przechowywane są na nich informacje i jak działają różne potrzebne im funkcje.
- Kompilacja informacji o różnych narzędziach dostępnych na rynku za darmo i wybór najbardziej odpowiednich dla tego projektu.
- Organizacja pomysłów i struktura różnych części, na które można podzielić aplikację internetową.
- Opracowanie kodu źródłowego aplikacji internetowej, badanie każdego z głównych działań i wybór najlepszego sposobu ich wdrożenia.

Jednak, jak każdy sieciowy projekt, wciąż można go rozwinąć. Spełnił on wszystkie założone elementy funkcjonalne, ale wciąż możliwe jest jego dalszy rozwój.

List of Figures

Fig. 2.1. Home page of web application [1], Screenshot from the developed webpage.
Fig. 2.2. Search page of web application [2], Screenshot from the developed webpage.
Fig. 2.3. Sign up box of web application [3], Screenshot from the developed webpage.
Fig. 2.4. User's page of web application [4], Screenshot from the developed webpage.
Fig. 2.5. File detail box of web application [5], Screenshot from the developed webpage.
Fig. 3.1.1. Computers are connected by a wire [6] (online – accessed: May 25th, 2020), Available online: https://mdn.mozillademos.org/files/8441/internet-schema-1.png.
Fig. 3.1.2. Computers are connected together [7] (online – accessed: May 25th, 2020), Available online: https://mdn.mozillademos.org/files/8445/internet-schema-3.png
Fig. 3.2.1. HTTP request and response [8] (online – accessed: May 26th, 2020), Available online: https://mdn.mozillademos.org/files/8659/web-server.svg
Fig. 4.1. Running of a static web [9] (online – accessed: May 26th, 2020), Available online: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction
Fig. 4.2. Running of a dynamic web [10] (online – accessed: May 26th, 2020), Available online: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction
Fig. 4.1.1. Node.js and Express logo [11] (online – accessed: May 27th, 2020), Available online: https://medium.com/@aarnlpezsosa/introducci%C3%B3n-a-express-js-a1ebe16dbcf4
Fig. 4.2.1 MySQL logo [12] (online – accessed: May 27th, 2020), Available online: https://en.wikipedia.org/wiki/MySQL
Fig. 4.3.1 Templating engine performance [13] (online – accessed: May 27th, 2020), Available online: https://en.wikipedia.org/wiki/File:TempEngWeb016
Fig. 4.3.2 Pug logo [14] (online – accessed: May 27th, 2020), Available online: https://en.wikipedia.org/wiki/File:TempEngWeb016
Fig. 4.3.2 Heroku logo [15] (online – accessed: May 27th, 2020), Available online: https://gluonhq.com/mobile-cloud-and-heroku-100-pure-java/
Fig. 4.3.2 Classes diagram [16]
Fig. 4.3.2 Use case diagram [17]
Fig. 5.3.1 Activity diagram for signing up [18]
Fig. 5.3.2 Activity diagram for signing in [19]
Fig. 5.3.3 Activity diagram for creating a file [20]
Fig. 5.3.4 Activity diagram for downloading a file [21]
Fig. 5.3.5 Activity diagram for searching files [22]
Fig. 5.3.6 Activity diagram for adding new admins [23]
Fig. 5.3.6 Activity diagram for adding points to users [24]
Fig. 5.4.1 Relational Entities database diagram [25]
Fig. 6.1.1 directory structure [26]. Screenshot from Visual Code
Fig. 6.7.1 File detail of the web page [27], Screenshot from the developed webpage.
Fig. 6.8.1 Users' profile web page [28], Screenshot from the developed webpage.
Fig. 6.8.2 Searching web page [29]
Fig. 6.10.1 Encryption schema [30] (online – accessed: May 30th, 2020), Available online: https://www.thesslstore.com/blog/difference-encryption-hashing-salting/

Fig. 6.10.2 Hashing schema [31] (online – accessed: May 30th, 2020), Available online: https://www.thesslstore.com/blog/difference-encryption-hashing-salting/

Code Listing

Lis. 6.1.1 package.json code [1]
Lis. 6.3.1 app.js libraries [2]
Lis. 6.3.2 app.js modules [3]
Lis. 6.3.3 Some app.js initializations [4]
Lis. 6.3.4 Some app.js initializations [5]
Lis. 6.3.5 Some app.js initializations [6]
Lis. 6.3.6 app.js public folder configuration [7]
Lis. 6.3.7 Global variables and routes' configuration [8]
Lis. 6.3.8 app.js errors' handling [9]
Lis. 6.3.9 app.js is exported [10]
Lis. 6.4.1 express-generator installation [11]
Lis. 6.4.2 www libraries and modules [12]
Lis. 6.4.3 www port definition [13]
Lis. 6.4.2 www server initialization [14]
Lis. 6.5.1 database.js libraries and modules [15]
Lis. 6.5.2 database initialization [16]
Lis. 6.5.3 Database connection [17]
Lis. 6.5.4 Database exportation [18]
Lis. 6.6.1 Routes' libraries [19]
Lis. 6.6.2 Routes' modules [20]
Lis. 6.6.3 Some routes' examples [21]
Lis. 6.7.1 Controllers' files structure [22]
Lis. 6.7.2 file_detail_get controller [23]
Lis. 6.7.3 file_detail_post controller [24]
Lis. 6.7.4 file_create_post controller [25]
Lis. 6.7.5 files_ get controller [26]
Lis. 6.7.6 search_filtering controller [27]
Lis. 6.7.7 user_new_password controller [28]
Lis. 6.7.8 user_create_post controller [29]
Lis. 6.7.9 user_signin_post controller [30]
Lis. 6.8.1 user profile pug file [31]
Lis. 6.8.2 searching pug file [32]
Lis. 6.9.1 JavaScript search file [33]
Lis. 6.9.2 JavaScript search file [34]
Lis. 6.9.3 JavaScript `confirmDeleteFile` function [35]
Lis. 6.10.1 Helper functions [36]
Lis. 6.10.2 Authentication functions [37]
Lis. 6.10.3 `isLoggedIn` function [38]
Lis. 6.10.4 Mailer functions [39]
Lis. 6.10.5 Passport functions [40]
Lis. 6.10.6 Passport serialize and deserialize user functions [41]