myFacebook

Leandro Salgado
A70949

Departamento de Informática
Universidade do Minho

1 Introduction

Nowadays, people use more and more digital repositories to store files, photos, and other things, because by having them on a digital platform they can access them everywhere and share them easily with other people, as long as they have their credentials to authenticate on said platform. If we take a closer look on the social media websites nowadays such as Facebook and Instagram, they can be considered an evolution of a digital repository because they allow the user to store information and to share it continuously with it's friends and family.

This project, in the area of web development, has as a main objective the development of a web application that allows it's users to store ideas and items like photos in a digital repository. In this application, the users can also share with other users the content that they stored (like in a social media website) and they can as well view said content later.

In terms of privacy, the users are free to store any content they want in the repository but they may not want every single one of their items exposed to the other users. Regarding this situation, the application allows the user to define privacy settings for his personal information and for every single item that he chooses to store. As for the privacy settings that the users can choose there are three settings: public, semi-private and private. For example, a user can have a public profile, which means that his profile information is public, but have items that are private and vice-versa. In the case of a semi-private profile, only the user and his friends can see its content, yet a friend of a user's friend will not be able to see the user's content nor the possible comments that the friend in common made on the user item.

In the development of this application it was decided that by default when a user creates an account it will be automatically set to private. This setting can be changed later on after the account is created.

In terms of structure, this report will be structured in the following way: firstly the functional and nonfunctional requirements that were defined for the correct development of this web application will be presented. Secondly an explanation of how the web application was implemented in terms of database models and business logic will be provided. To conclude this report, some considerations about this project in terms of the challenges faced and future work will be discussed.

2 Requirements

In this section, the functional and non-functional requirements needed for the correct development and behavior of the web application will be presented.

2.1 Functional requirements

2.1.1 Authentication

1. The application must allow a user to register.
2. The application must allow the user to login and logout.
3. The application must tell the user when the inserted credentials were wrong.
4. The application must allow the user to try to login again when the credentials were wrong.
5. The application must allow the user to change its password.

2.1.2 Privacy

1. The application must allow users to choose how private their profile is.
2. The application must allow users to choose how private their content is.
3. The application must not allow public access to private content or profiles.
4. The application must allow the user to request its account deletion.
5. The application must request confirmation for the account deletion process.

2.1.3 Profile

1. The user must have access to his personal user profile.
2. The application must allow the user to edit the following fields of the profile:
   (a) e-mail
   (b) password
   (c) name
   (d) gender
   (e) birth date
   (f) contacts
3. The user must be able to view the profiles of other users.
4. The user profile will show the user's information according to how private their profile is.

2.1.4 Friends

1. The application must allow the user to request other users to be added to their friends list.
2. The application must allow the user to remove friends from their friends list.

2.1.5 Items

1. The application must allow the user to post an item.
2. The application must allow the user to edit an item, including adding and removing files.
3. The application must allow the user to remove an item.

2.1.6 Comments

1. The application must allow the user to comment on items.
2. The application must allow the user to reply to comments.
3. The application must allow the user to remove their comments.

2.1.7 Search

1. The application must allow the user to search items by ”hashtags”.
2. The application must allow the user to search users by name.
3. The application must allow the user to search users by e-mail.

2.1.8 System

1. The application must allow the user to import and export data.

2.2 Non-functional requirements

2.2.1 Authentication

1. The system must store information regarding the authentication, to allow the user to stay logged in.

2.2.2 Privacy

1. The system must delete the content related to the user when requested.

2.2.3 System

1. The system must allow multiple concurrent users.

2.2.4 Data persistence

1. The system must keep record of all information pertaining users in a database.

2.2.5 Interface

1. The interface must be simple to use.

3 Implementation

The application was implemented with *Node.js* and the *Express* package. The *Express* package is a fast and flexible web application framework that allows the creation of web applications in a short amount of time.

To allow for a more focused development, the application was split in three main components: the data model, the interface, and the business logic. In this section, each component is explored as a contained unit, and how it connects to other components.

3.1 Model

The chosen database engine for this project was *MongoDB*, since it uses a relatively easy syntax, allowing for fast direct database querying for testing purposes, but also due to the *mongoose* package.

The package *mongoose* allows for easy object modeling, providing a schema-based solution to model application data. With it, the data was split into three models: the **user model**, the **item model**, and the **file model**. For each model, a schema was created, which defines what kind of information the model holds.

The **user model**, as the name implies, contains information regarding users, namely, their personal information, their friends list, a list of their items, and their profile privacy settings. In this case, since this model will store the user password, a pre hook on the save function of the schema was added, which encrypts the password of the user, using the package *bcrypt*, before saving it in the database.

The **item model** contains information regarding the items users create. It contains all information users provided when creating the item, as well as a list of files the user attached to the item. It also contains the item privacy setting, allowing users with public profiles to restrict access to specific items. In addition to that, the **item model** also stores the comments made on the item, in a way suggested by figure 1. A distinction is made between a *comment* and a *reply*, in which a *comment* is a direct reply to the item, and a *reply* is an answer to a comment. The main difference in the database is that *replies* are the 'leaves' on the 'tree' (see figure 1), and as such, replying to a *reply* will actually store it as a *reply* to the 'root' *comment*.

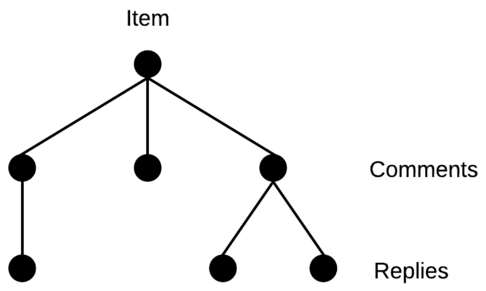

Figure 1: Visual representation of how comments are stored.

The **file model** contains information about the files uploaded by the users. The files are uploaded to the application as *multipart/form-data*, which is handled by the middleware package called *multer*. After being uploaded, the files are stored in a directory, with a name generated by *multer*. The original filename, the *id* of the user who uploaded the file, and the filename generated by *multer* will be stored in the **file model**, with the latter being the *id* of the file.

The models suffered many iterations, and the current stage was the preferred one, as it allows easy access to any information, both for reading and for writing, and allows the application to verify privacy settings for each individual piece of information.

Afterwards, a controller for each model was developed, in which all needed queries were implemented, always keeping in mind the privacy settings.

3.2 Interface

The interface was designed and implemented using *Pug* template engine, with *jQuery Mobile* for client-side Javascript and CSS.

Pug, formerly known as *Jade*, was chosen as the template engine for this project due to having a syntax easy to understand. Having a seamless integration with *Express* was also a great bonus.

Due to a lack of designing capabilities on the part of the developer of this project, *jQuery Mobile* was chosen. *jQuery Mobile* is a user interface system, with its own set of CSS and client-side Javascript, that allows the development of responsive apps available in multiple platforms. On account of having a defined style to all page elements, the burden of creating good looking pages was shifted away from developer, allowing the interface designing process to be even faster.

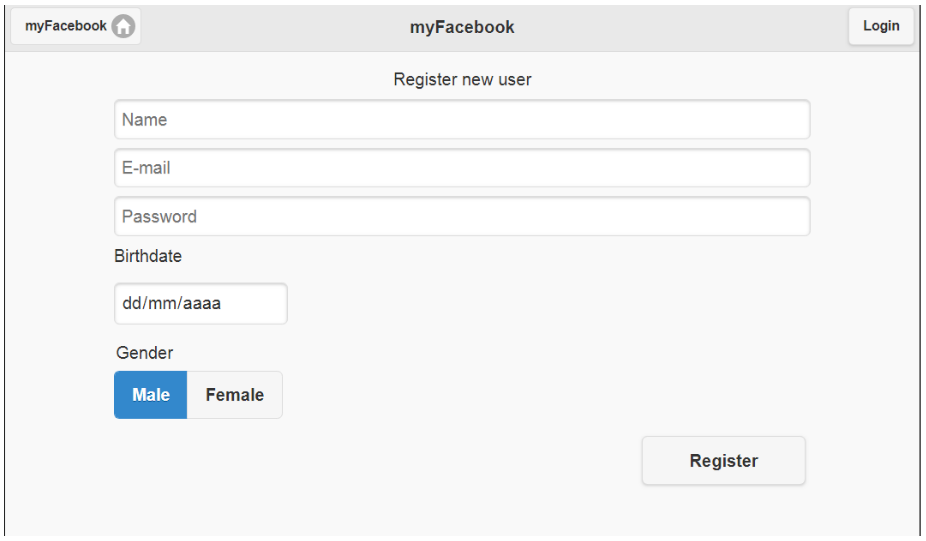

Figure 2: Registration form designed for this project.

However, *jQuery Mobile* has its own set of problems. Asides from not supporting the most recent versions of *jQuery*, it's not very customizable. The automatic handling of button presses and page changes also created some problems regarding URLs.

While *jQuery Mobile* was not the best choice in regards to ease of use, it allowed an interface to be implemented rather quickly, which, due to time constraints, was the most desired outcome.

Since *jQuery Mobile* already handles most of the page elements, only a small amount of client-side Javascript had to be written, mostly to handle the search and commenting functions of the application.

3.3 Business Logic

As previously hinted, the application was developed with *Express*, due to how easy it is to create routing inside the application.

Two sets of routes were implemented: a route for data calls and a route for interface calls. This allows other applications to access the data without the need for an interface.

As to protect the data, most routes use authentication, not only to determine who is the user making the requests, but also to determine if that user is allowed to access the content it is requesting. Still on this topic of privacy, even content flagged as being public is 'locked' behind authentication. This means that a non-registered user does not have access to such public content.

As per the initial requirements for the application, a set of scripts for importing and exporting data were also developed.

Due to time constraints, a large number of rather small quality of life features could not be implemented in the interface. These will be addressed in a later section of this document.

3.3.1 Authentication

The authentication process is done using the packages *passport* and *jsonwebtoken*. This will allow the application to know who is the user making requests, and if he should have access to the data he is requesting.

An authentication module was created to hold all the strategies created for *passport*. This module was heavily based on the authentication module created in class.

When a request is made to a route, first it is checked whether or not the requester has a token. This token is generated when the user logs in and it's stored in the session cookies. If the requester does have the token, the request is allowed to continue to the next handler. Otherwise, a response with an unauthorized status (401) is sent.

3.3.2 Data API routing

In order to allow other applications to access the data contained in this application, appropriate routing was implemented. After the user is authenticated by *passport*, the route will process the request, most of the times starting with a verification of the token to ascertain the *id* of the user.

Most of the times, this *id* will allow the controller to determine whether or not the user has access to the requested content. Finally, a response is built and sent with either the content from the controller function or an error

All of the required functionalities are available in the data API routes.

3.3.3 Interface routing

Developed separately from the data API routes, the routes for the interface are slightly different from what would be expected.

Only routes pertaining to specific user-created content have authentication. Routes that relate to 'public' pages, such as the registration or login page.

Requests that require access to the database, would be made through the data API routes, using a package capable of sending HTTP requests, such as *axios*. However, due to hardware and time constraints, routes that handle such requests were not implemented in such a way, and instead send queries directly to the database, through the appropriate controller.

Due to time constraints and problems imposed by *jQuery Mobile*, it was not possible to implement quite a number of features, which, however, are available through the data API routes.

3.3.4 Importing and Exporting data

As imposed by the initial requirements of this project, a set of scripts were developed which allow importing and exporting the data the application uses.

As a result of how the models were designed, allowing a single user to import data of his own would required a very powerful script that would check for each field before submitting. Once again, due to time constraints, the scripts were not designed to be used by the common user.

Instead, the scripts were designed with their main focus on moving the data as whole, in instances such as making a complete backup copy of the database.

4 Concluding remarks

4.1 Challenges

The main challenge was certainly the time constraints. Due to multiple academic reasons, the available time for developing this project was reduced to about 10 days.

The creation of a set of *mongoose* models capable of holding all the information needed, and allow for quick and flexible retrieval and update of such information was the first challenge. When creating the models, I also had to be mindful of how it would be possible to check for access permissions for any content. For this reason, the models went through many iterations until reaching the current stage.

As a result of the iterations the models suffered, their controllers also suffered a good number of iterations. This ended consuming most of the time available for this project.

jQuery Mobile imposed a serious challenge. The automatic handling of button functionality was a double-edge sword: it was good when handling error responses from requests, but at times it also generated weird requests. It was also hard to customize default behaviours, being almost impossible in some cases.

As stated before, the machine in which this project was developed also created some problems. Very high latency, both when querying the database and when when making requests through *axios*. To dispel doubts on whether the high latency was a problem created by malfunctioning hardware or badly constructed queries, some latency tests were conducted on better performing hardware. According to those tests, it was possible to determine that the problem was indeed the hardware.

4.2 Future work

Due to the time constraints previously mentioned, some shortcuts were taken, and not all features could be implemented. In the future, this will be corrected. As a result of the way the application is split into, how data API routes and interface routes are separate, fixing those shortcuts will be relatively easy, as the changes will be mostly contained. Features present in the data API routes but absent from the interface will also be implemented.

As for *jQuery Mobile*, since it created a fair amount of problems, it will be replaced by a more reliable and customizable interface system, which should also help on fixing some of the aforementioned shortcuts.