QuepasApp (Lucas and Manuel)

How to run

1. MySQL and PHP running
2. Ideally allow `.htaccess` usage in the folder from Apache
3. Appropriate access data to the database in `configuration.json`
4. PHPMailer (through *composer*) installed in the folder
5. `quepasapp.sql` run to have the appropriate structure on the database
6. Appropiate email data in the `email.json`
7. Ensure PHP can read and write on `attachments/` and `avatars/`, e.g. `sudo chown -R www-data ./attachments` or `sudo chmod -R 777 ./avatars`.

What we include as sample data in the database

We include five users, all of them with the password `server`:

- User *admin*: an user with administrator rights
- User *isolated-user*: an user that does not have any contact with anyone
- User *user-friend-1*: user with profile data (avatar, name, and about) which is friends with *user-friend-2* and belongs to the group *Group Chat* shared between *user-friend-1*, *user-friend-2*, and *user-group*
- User *user-friend-2*: similar to the *user-friend-1* but without an avatar, has a private conversation with the *user-friend-1*
- User *user-group*: creator of the *Group Chat* group

We also include a few messages by all of them, and one attached file, and one attached image. Attachments and avatars are also included in the corresponding folders.

Requirements specification

These are the current specifications for certain things in the program:

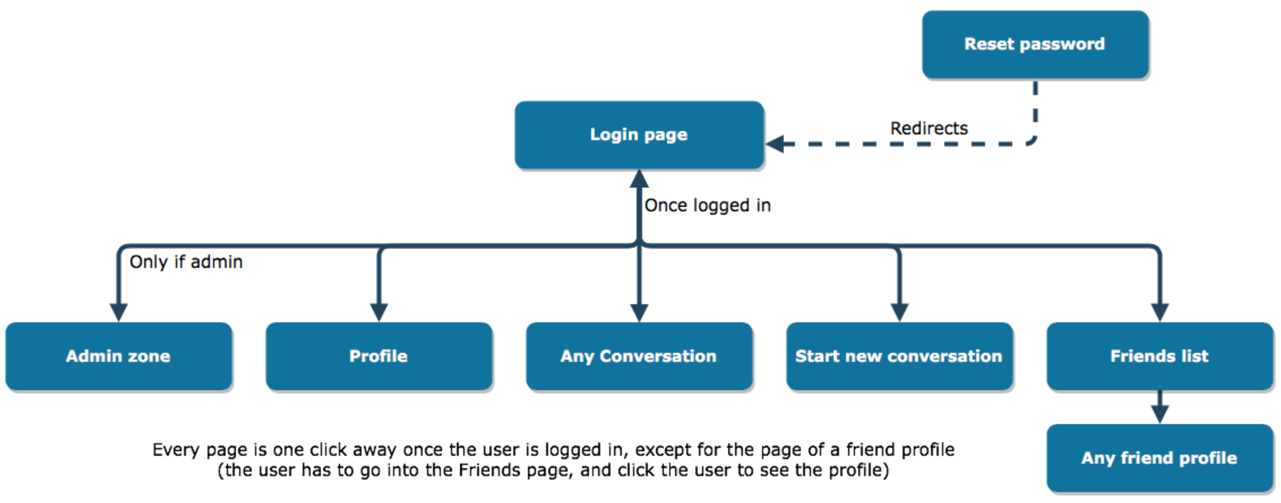
- *Username*: non-blank, 100 characters max.
- *Password*: non-empty
- *Max attachment file size*: 2MB
- *Max avatar file size*: 2MB
- *Image format* (otherwise not recognized as images): gif, png, jpg
- *Message character limit*: 1000 characters
- *Account activation token expire date*: never (can be used only once)
- *Password reset token expire date*: 10 minutes after it’s been requested (can be used only once)

Extensions made

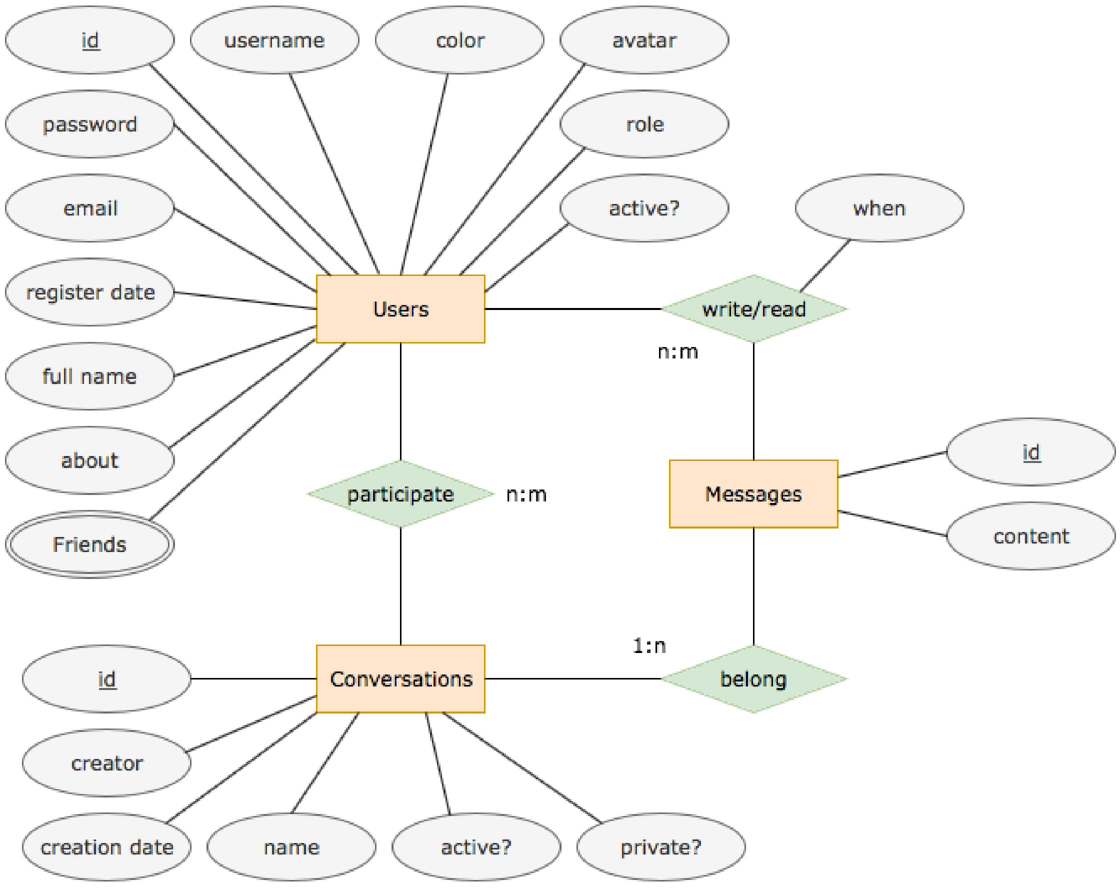
We made something similar to WhatsApp/Telegram, these are the extensions implemented. We explain how we implemented each of them in detail below.

Extension	Implemented
A1	*Yes*
A2	*Yes*
A3	No
A4	*Yes*
A5	*Yes*
A6	*Yes*
A7	*Yes*
A8	*Yes*
A9	*Yes*
A10	*Yes*
A11	*Yes*
A12	*Yes*
A13	*Yes*

Screen map


Reset password
Redirects
Login page
Once logged in
Only if admin
Admin zone
Profile
Any Conversation
Start new conversation
Friends list
Any friend profile
Every page is one click away once the user is logged in, except for the page of a friend profile
(the user has to go into the Friends page, and click the user to see the profile)

Entity relationship diagram

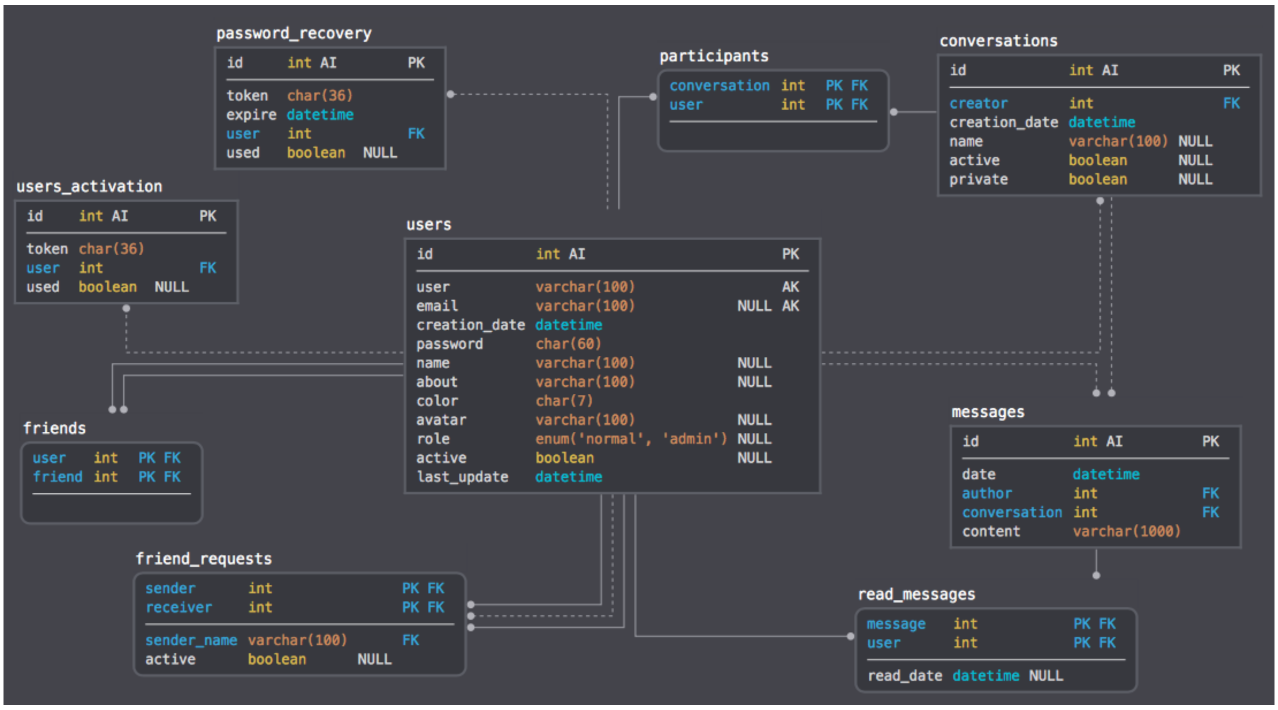


Logical schema

In **bold**, primary keys; in *italics*, foreign keys.

> users(**id**, user, email, creation_date, password, name, about, color, avatar, role, active, last_update)
> users_activation(**id**, token, *user*, used)
> password_recovery(**id**, token, expire, *user*, used)
> conversations(**id**, *creator*, creation_date, name, active, private)
> participants(***conversation***, ***user***)
> friends(***user***, ***friend***)
> friend_requests(***sender***, ***receiver***, active)
> messages(**id**, *author*, *conversation*, date, content)
> read_messages(***message***, ***user***, read_date)

Database diagram



File summary

Path	Description
`attachments/`	Folder that holds all the attachments of the app (subfolders are the sha1 of the file)
`avatars/`	Folder where the avatars are stored (subfolders are the ids of the users)
`php/`	Folder where all the PHP files are
`vendor/`	*Composer* application folder
`.htaccess`	.htaccess for Apache that disables folder file indexes and makes `quepasapp.html` the default file to be opened
`composer.json`	*Composer* application file
`composer.lock`	*Composer* application file
`configuration.json`	Database access configuration file
`email.json`	Email access data (used by PHPMailer)
`favicon.ico`	Favicon of the site
`quepasapp.html`	Main document o the app, loads the JS and CSS so that the app is usable
`quepasapp.js`	Holds all the client side code (JavaScript)
`quepasapp.sql`	Script to create the database, holds some example data
`style.css`	The CSS style file for the app

And now a more detailed file description for all the PHP files under the `php/` folder:

Path	Description	Parameters
`activate.php`	Tries to activate account given an activation token	`$_GET['token']`
`add-friend.php`	Adds a friend to the current users friends list and viceversa	`$_POST['friend']`
`add-participants.php`	Adds participant/s to a conversation	`$_POST['participant']` `$_POST['conversation']`
`check-user.php`	Checks that a user exists	`$_POST['user']`
`create-conversation.php`	Creates a new conversation	`$_POST['name']` `$_POST['private']`
`data.php`	Returns all the data relevant to the user (new messages, friend requests, user data, etc.)	`$_GET['message']`
`database.php`	Holds all the database related functions (except for the `SELECT` s that are in `data.php`)	
`de-activate-user.php`	So that an admin activates or deactivates accounts	`$_POST['user']` `$_POST['action']`
`deny-friend.php`	Denies or blocks a friend request	`$_POST['sender']` `$_POST['block']`
`email.php`	Holds all the functions related to emails	
`forgot.php`	For creating a password reset token and sending an email to the user	`$_POST['email']`
`leave-conversation.php`	Allows a user to leave a conversation	`$_POST['conversation']`
`log-out.php`	Closes session	
`logged-in.php`	Checks if the session is active and the user is active	`$_GET['initial']`
`login.php`	Logs a user in	`$_POST['user']` `$_POST['password']`
`read-messages.php`	Marks a conversation as read	`$_POST['conversation']`
`register.php`	Register a new user	`$_POST['user']` `$_POST['password']` `$_POST['email']`

Path	Description	Parameters
`remove-friend.php`	Removes a user from the current users friends list	`$_POST['friend']`
`remove-participant.php`	Removes a participant from a conversation	`$_POST['participant']` `$_POST['conversation']`
`request-friend.php`	Requests friendship with a user	`$_POST['receiver']`
`reset.php`	Resets an user password with a token	`$_POST['token']` `$_POST['password']`
`send-message.php`	Sends a message, including the attachment files (which saves in the `attachments/` folder)	`$_POST['attachment']` `$_POST['message']` `$_POST['conversation']` `$_FILES['attachment']`
`update-user-info.php`	Updates the profile information of an user (including a possible update of the avatar, which is moved to `avatars/`	`$_POST['email']` `$_POST['about']` `$_POST['color']` `$_POST['name']` `$_POST['avatar']` `$_FILES['avatar']`
`valid-reset-token.php`	Checks wether a token is valid to reset a password	`$_POST['token']`

App explanation (User manual)

Something to mention that's not related to something in particular is that we use JSON for every message returned by the server side to the client (which asks for that info through AJAX), so that the client can parse that (`JSON.parse()`) and then treat the answers as JavaScript elements.

Before explaininig how we implemented each part of the application, we have to mention that there are a few things that we do *inside* the database with triggers (rather than on the app code). We have five triggers:

- `unread_messages`: inserts the message in the `read_messages` table for every user in the conversation, marking the message as read for the creator of the message;
- `creator_participant`: once a conversation is created we insert the creator automatically into the `participants` table for that conversation;
- `user_random_color`: auto generate a random (dark) color for each new user;
- `user_activation_token`: auto generate on the database the activation token and insert it into `users_activation`; and
- `password_token_expire`: once a user requests a new password, we automatically generate the token *and* set te expire date to ten minutes later.

Now we will explain all the points that we made.

Basic functionality

> ✓ *Access*: Users access the application with username and password that are stored in a database.

In QuepasApp the username and password are stored in the `users` table in the columns `user VARCHAR(100) NOT NULL UNIQUE` and `password CHAR(60) NOT NULL`. The related functions in JavaScript are `login()` and `logout()`, and the PHP files related to this are `login.php`, `log-out.php`.

We do not impose any limitation regarding username or password (well, apart from the fact that the username column is `VARCHAR(100)` so it cannot be longer than 100 characters), they just cannot be empty; the password could be blank, but not the user, since we trim the spaces at the beginning and at the end of it before storing in the database.

Users log in through the login page [image].

> ✓ *Sending*: Users of the application can send messages to other users (the recipient is chosen by name).

To send a message, one has to create a conversation first. In the left menu, the button *Start Conversation* lets you create a conversation. After you write a username correctly and click on the button *Add*, you can create a conversation (after optionally giving it a name) by clicking on the *Create* button.

After the conversation is created, you can click on the left menu on the conversation to open it and at the bottom of the page write the message and hit Enter to send (or, alternatively, click on the rightmost button).

Conversations are stored in `conversations` table in the database, and messages are stored in the `messages` table. The content of the message is on the column `content VARCHAR(1000) NOT NULL`, so we do not allow messages of more than 1000 characters; actually, we truncate them on the server before inserting. Other than that, we also process the whole message to remove any HTML tag and convert them to printable characters (using the PHP function `htmlspecialchars()`).

The JavaScript functions that are most related to this task are `createConversation()`, `startConversation()`, `createPrivateconversation()`, `printConversation()`, and `sendMessage()`.

And the corresponding PHP file is `send-message.php`.

> ✓ *Inbox*: Users can see a list or table with the messages they have received, differentiating between read and not. The entire messages are not shown in the inbox, only the subject or the beginning of the message.

On the left, the user is shown all the current conversations he is part of, and can see from the outside, without actually opening the conversation, the beginning of the last message, and how many messages are unread in that conversation (a bubble with red background to call his attention). Once the user opens a conversation to see its messages, the conversation is marked as read, and consequently all the messages of that conversation are marked as read for the user.

The differentiation between read and unread is done in the database through the `read_messages` table. The closes JavaScript functions to this subject are `printConversations()`, and `markConversationAsRead()`. And on the server side, the file `read-messages.php`.

> - [x] *Messages*: Users can also check each message individually. In this case, they read the full content and they also have the opportunity to respond.

Following from the last two points, once the user opens the conversation (by clicking on the conversation from the left), all the messages of the conversation are shown. So the user can access all the messages and read the full content. And in the same vein as a message is sent, an *answer* to the conversation is sent: the message written at the bottom and then sending it.

The user receives messages and sends messages to conversation only if they are in the table `participants` that links users to conversations.

Related to printing the whole conversation, we could mention `msgDiv()`, `conversationName()`, `conversationImg()` that take care of printing the messages and the name and image of the conversation. But mostly it's the same as the last two points.

Extensions

> - [x] A1. Self-registration:
>   - Realistic process for users to register on the website.

The registration is done also at the login page. The users insert a username, a password, and an email where the activation link will be sent.

For the username, we just trim it server side to avoid it being blank, and also convert any possible HTML character to its printed form to avoid people "injecting" code in there. After all this, the username should be no longer than 100 character because of the length of the column in the database, but apart from this, we pose no other limit or restriction to the usernames.

Regarding password security, we decided for this assignment to not require anything, but if we were going to require something we do not agree with the current convention of asking the user to insert , *e.g.*, a number, an uppercase letter, a symbol, etc. We believe that if something has to be asked is length, say at minimum 16 characters, nothing else. In any case this could be changed in a line on the `register.php` file.

And for the mail, we need the user to give us a valid email because the account needs to be activated, and we send an activation link to the user. If the email is of a valid form, we try our best to send an email but cannot ensure that it's received.

Once the link in the email is visited, the account is activated and the user can now log in.

The activation tokens are stored in the database on the `users_activation` table. The most related JavaScript functions to this feature are `activateAccount()`, and `register()`. And, regarding PHP code, the files `register.php` and `activate.php`. These files make use of functions defined in `database.php` and `email.php`. We won't be mentioning those two utility files every time they are called from another of the PHP files, but they are used almost in any point that needs to insert or alter something in the database.

> - [x] A2. Password recovery:

> - Realistic process for users to recover their password or generate a new one.

Again, this can be achieved from the login page. Once the user clicks on *Forgot your password?*, an input is shown to insert the email where the mail will be sent. We do not give any hint wether the mail is associated with any of our accounts, the user is responsible for inserting the email correctly, and wether the email has been successfully sent.

If the email is correct, an email will be sent with a link. The user has 10 minutes to click on this link and change the password. After 10 minutes the link becomes invalid. The user could ask again for another reset link.

Again, we do not limit passwords in any way (other than they can't be blank).

In the database the reset tokens and their expire date are stored in the `password_recovery` table. The PHP files in charge of this feature are `forgot.php`, and `reset.php`; and they are called from JavaScript from `recoverPassword()`, `checkResetPassword()`, and `resetPassword()`.

> ☐ A3. Messages to several recipients:
> - Allow to enter more than one recipient in the message sending form.

We did not do this. What we had in mind was similar to the *Broadcast List* of WhatsApp, but due to time constraints we didn't end up developing it.

> ☑ A4. Encrypted user password:
> - Store the access codes of the users encrypted in the database (function `password_hash()`).

Rather than having the password stored as plain text in the database we store them encrypted. Thanks to the PHP functions `password_hash()` we store the hash (using the `PASSWORD_DEFAULT` recommended method, which as of today represents the *bcrypt* algorithm). And to check if an user is logged in rather than checking in database for the user and password, we grab the whole row based on the username, and use `password_verify()` to check if the given password would generate the same hash that's stored in the database.

This feature is taken into account in the `database.php` file in a few functions:

- `login()` function uses `password_verify()`;
- `insert_user_send_email()` uses `password_hash()`; and
- `reset_password()` uses `pasword_hash()` again when resetting a password.

> ☑ A5. User avatar:
> - Users can upload an image to use as an avatar.
> - It has to have some consequence in the application. For example, that the avatar appears in the messages that a user sends, or that can it be seen when accessing their profile.

Users have the option of uploading an image and using it as an avatar. This option is available in the *My Profile* tab. To upload an avatar you must upload an image (of type `.jpg`, `.png` or `.gif`) and click *Save Changes*. If the user attempts to upload a file of any other type, they won't be able to save their changes. This avatar is displayed on the users profile and only their friends can view it.

The avatar cannot exceed 2MB of file size, this is checked by the `quepasapp.js` function `saveProfileInfo()`.

> - [x] A6. User profile:
>   - Users can enter a series of information about themselves, such as age, city of residence, hobbies or their avatar.
>   - Other users must be able to access this information.

Users can view their own profiles by selecting the *My Profile* tab. From here, you can view and edit your profile information. This information includes: Email, Name, Avatar, Color and an About section. The Color chosen here determines the color of your name on the messages you send. Your profile can only be viewed by users on your friends list. You can view your friends profiles by clicking on their names in your friends list. From here you also have the option of removing them from your friends lit.

This feature is handled by `update-user-info.php`, `database.php` function `update_user_info()` and the `quepasapp.js` function `saveProfileInfo()`.

> - [x] A7. Friendship:
>   - Two users can establish a friendship relationship with each other.
>   - One of them starts the process by sending a request, which the other has to accept or reject.
>   - There has to be some consequence in the application. For example, that users can only write to their friends or that a user's profile is only visible to their friends.

To establish friendship with another user, you must go to the *Friends List* tab. Here you can write the name of the user you wish to add and click *Request*. This will add a new entry in the `friend_requests` table. Once the request is sent, the receiver will see the name of the sender of the request along with 3 options: Add, Deny or Block.

If the receiver decides to add the sender as a friend, two new entries are inserted into the `friends` table and they will both have each other on their respective friends list. On top of this, a new private conversation will be created between them. Neither of these users can add or remove participants from this conversation. Once somebody is on your friends list, you may now see their user profile. You can also remove somebody from your friends lsit by entering their profile and clicking *Remove Friend*.

If the request is denied, the request is deleted and nothing else happens. If the request is blocked, this entry in the `friend_requests` table is marked inactive. This means the request is no longer visible and the sender can't send any more requests.

The PHP files in charge of this feature are: `add-friend.php`, `deny-friend.php`, `remove-friend.php` and `request-friend.php`.

> - [x] A8. Groups:
>   - Option 1: Develop a group system similar to Telegram or WhatsApp.
>   - Option 2: Allow the creation of groups that are collections of user names. When sending a message to one of these groups, one message is sent to each of the users.

In Quepasapp users can create as many conversations as they want with as many people as they want. These conversations allow then to send/receive messages from multiple users in one location. All conversations (private/public) appear on the left-hand side.

To create a group conversation users can simply click *Start Conversation* and add as many participants as they want. Once the conversation is created they can click on it and view all participants in it on the top of the screen. The creator of the conversation can also add and remove participants if they wish.

Because of they we have made the database, this feature does not require any extra tables. We use the same `conversations` and `participants` tables that private conversations use.

> - [x] A9. Administration zone:
>   - Administration zone only visible to some users (depending on their role).
>   - In this area you have to implement tasks typical of an administrator. For example, block or unblock users.

Depending on the column `role ENUM('normal', 'admin') DEFAULT 'normal'` from the table `users` a user is an administrator or not. Administrators can only be created from within the database (altering it by hand), since every user registered through the main register fields will be created as `'normal'` users.

Being an administrator has two consequences.

- They get all the useful information from all the users; in contrast with normal users that only get all the useful information from friends, and just the id, the username, and the color, from everyone else.
- They have access to a menu that allows them to activate or deactivate accounts at will. The menu is accessible from the left panel by clicking on the *Admin zone* button, and shows a list of registered users: by clicking on each user they can toggle the accounts between active and inactive.

This has a few limitations and effects itself. First, admins can only activate or deactivate *normal* user accounts, they cannot do anything to another admins. In case there's a problem they need to escalate this to someone with access to the database to withdraw the admin privileges from other admin. Second, once they activate or deactivate an account the validity of the activation link sent to the email is not altered in any way (so if an user is activated by an admin —so the user does not *use* the activation link—, and then deactivated, the user can still re-activate the account through the email link).

And third, the moment an user is deactivated if that user is already logged in it will be kicked out.

> - [x] A10. Attached files:
>   - Allow the user to attach files to a message.
> - [x] A11. Images:
>   - Allow the user to insert images as part of a message.

To attach files or images one has to click on the “clip button” (the penultimate rightmost button, next to the message input text), and after attaching something, the body of the message is optional.

Both of these features are made together. If one attaches an image (of type `.jpg`, `.png`, `.gif`), the server recognises the filetype and shows it as an image. If the attached file is of another type, the file is attached, *per se*, so that the users can download it.

We decided to solve the attached files in a simple way. Since we do not allow users to insert HTML tags into the messages, we can actually insert that HTML tags from the server itself. If it's an image the message will contain these before the content:

```
<a href="path/to/image-2.extension" download>
  <img width=".." height=".." src="path/to/image-2.extension" alt="img-image-2.extension">
</a>
```

And if the attached file is not an image, a more general button will be in the message:

```
<a href="path/to/file.extension" download>
  <button type="button">file.extension</button>
</a>
```

This way, once an user has access to this message, it will be shown as an image or a button in the browser, and will be easily downloaded with one click.

As of how the path is calculated, to avoid that files could override each other, we decided to find the `sha1` hash of the file and put it in a folder with that name. That way there's no collision. For example, let's say that you upload `attachment-3.zip`, on the server we hash it and the resulting hash is `c98f4c638881ade85ad7e33d0844bac2a4b32d59`, then, the file will be stored in `./attachments/c98f4c638881ade85ad7e33d0844bac2a4b32d59/attachment-3.zip` and the code that will eventually make it to the message body will be similar to this:

```
<a href="./attachments/c98f4c638881ade85ad7e33d0844bac2a4b32d59/attachment-3.zip" download>
  <button type="button">attachment-3.zip</button>
</a>
... [optional contents of the message]
```

The code for the attachments is inserted into the message in the `send-message.php` file, depending on the existence of attached files. We do not have any dedicated table for attachments in the database, since the only reference (for now) to attached files is just on the body of the messages. The attachments are saved in the `attachments/` folder (which has to have the correct permissions for Apache to write there, in our case we made the Apache user — `_www` on Mac or `www-data` on Linux— the owner of the folder). And related to this we have in JavaScript `somethingAttached()` to alter the color of the button in case something is attached, and then `sendMessage()` itself takes care of sending everything wether there are attached files or not.

> - [x] A12. AJAX:
>   - Perform the web application as a single page application.

The page is done as a single page application. At no point has the user to reload the page or go to another URL to use the application. To make the app receive data on "real time" the client asks the server for some data every second. Every second we get:

- a few data from the users (in case some data has changed, or new users are in the app),
- all the data from the friends (which includes the *About* field, avatar, email, etc.),
- the new friend requests that someone might've send us,
- the conversations we are currently in, and their participants, and
- the new messages that we've not already received.

This way we can use the application as a (quasi-) real-time application. Apart from this, there are other moments in which we also trigger the "update", for example after sending a message, rather than waiting the interval to update itself.

All the info is stored on the client side on a global `data` object, which has the following structure (just an example, not thorough):

- `data.user`: all the info about the user itself;
- `data.users`: all the info about all the users (if you are friends with someone or an admin you might have more info abot certain users);
- `data.conversations`: all the conversations that you take part of;
- `data.conversations[1].messages`: for example this would hold all the messages that belong to conversation `1`;
- `data.conversations[2].participants`: for instance this would hold all the partcipants that belong to conversation `2`;
- `data.requests`: all the new friend requests;
- `data.interval`: the `setInterval()` that asks for data to the server every second;
- `data.lastMessage`: the date of the last message sent from the server, so that the user can ask only for newer messages;

To remove burden from the server, we just dump "raw", unordered, data from the server, and we make the clients organize this information. On the JavaScript side the function that calls for data is `fetchAll()` which is the one that's executed every second. This function, in turn, once the data is received, calls `organize()` which is a function that puts all the received data in the explained structure inside the `data` object, and then `update()` which is a function that updates things accordingly: it updates the conversation info on the left side, and depending on the page that's open on the right, updates the values accordingly (*e.g.*, if a conversation is open and new messages arrive updates the messages so you can see new messages, or if you are seeing the friends and a new request arrives you can see it).

Server side, the file that's in charge of sending all the data to the user is `data.php`. Which also updates the info of `$_SESSION['user']` to have the updated info every time.