Chat Project

MongoDB and Redis project

Student: Maria-Sânziana PRICHICI

IDU-4

# Goal

This project consists in an application client-server that will serve as a chat (a tool for sending messages between two users). For the moment is a local application, without any Web Sockets.

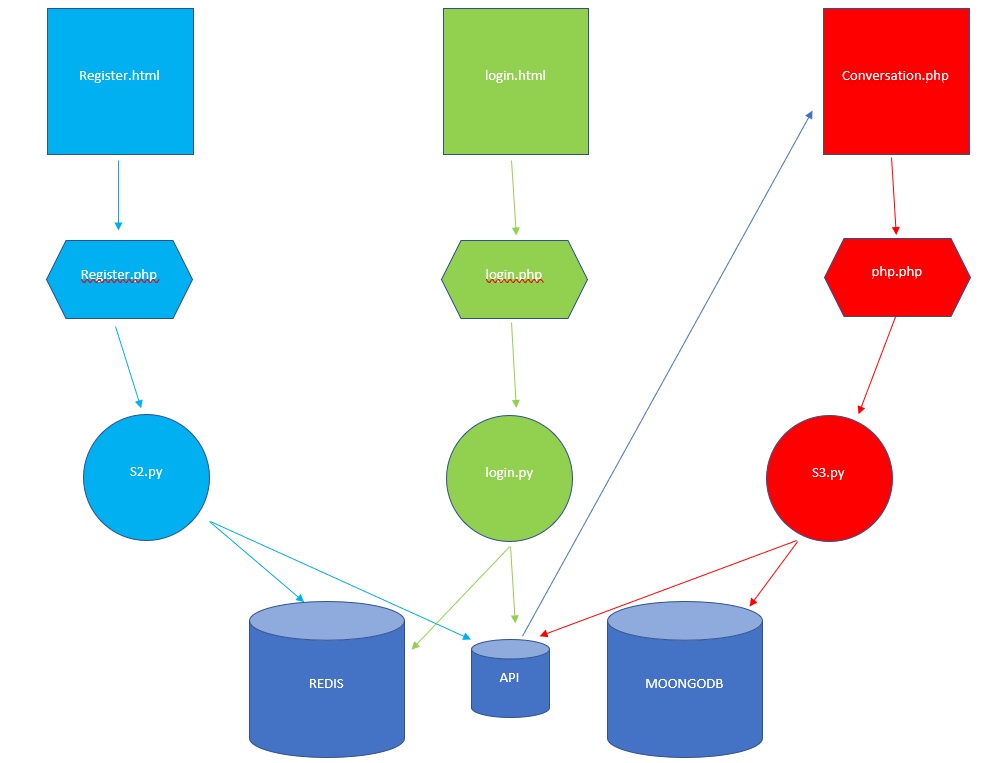
# Technologies used

When creating this project, I used these following tools:

* *HTML, CSS* and *JavaScript*, for the client part;
* *Php*, for sending and receiving data from the server to the client and the other way around;
* *Python* scripts, for receiving data from *php* and updating the servers;
* *MongoDB* (server side) for storing the messages between users;
* *Redis* (server side) for storing user data;
* *AJAX* in order to extract output from the client side and send it *php*;
* *mockAPI,* an API provider that allowed me to set an API in order to pass data from servers back to the client.

# How the project works

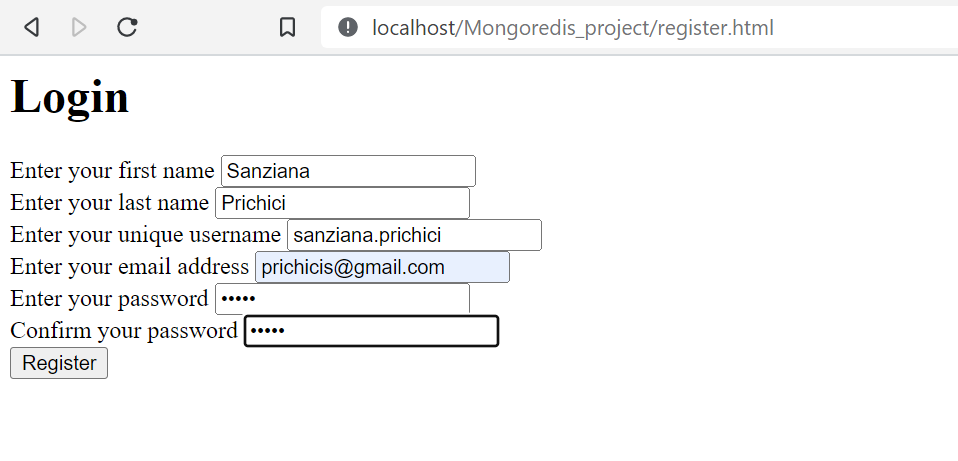
The project is quite simple consisting on only 3 main pages displayed for the client: register.html, login.html and Conversation.php.

****

In the following paragraphs, I will present the actions between these 3 pages and also the functionalities that they provide to the project.

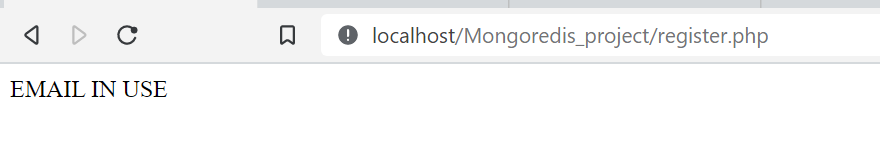
## Register page

This page it is meant to create new users in the database stored on the Redis server that runs on the 6379 port on my local machine. The new user will be stored in the API as well.



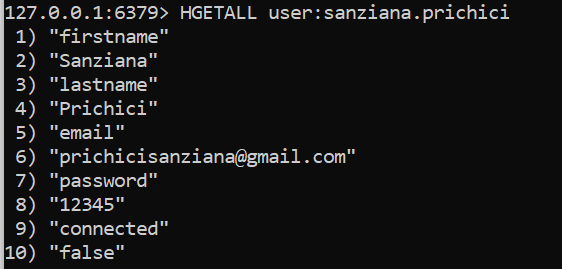
In order to create a new user, I demanded the person to enter their first and last name, unique username and email address and also a password. For a stronger security, I enabled the confirmation of the password. After pressing the “Register” button, the form sends the data using a *post* request to the php page named *register.php* that is just a bridge between html and python (*S2.py*). The php page only prepares the infos to be send to the python script that will start a new redis client in order to verify the constraints: unique username and unique email address[[1]](#footnote-1).

There were multiple error messages that could prevent the user to be created, such as: “Already existing user with this username”, “Email already in use” or “Password don’t match”. These messages are thrown in the register.php page:

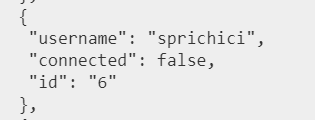


For further implementation details, I added comments in the register.php and S2.py files.

We can see the user stored in the redis database by using the HGETALL + key function in a redis client:



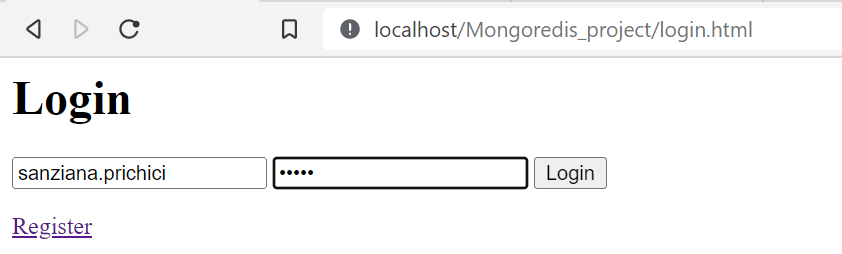
And in the API we will have only the necessary data such as: username, (is)connected and the id:



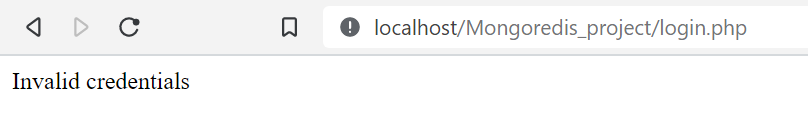
Here, the id is useful because it facilitates the process when searching for a conversation between 2 users. But this will be discussed in the subchapter dedicated to the conversations.

## Login

Now that the user is created, they can authenticate via the login page by entering the username and the password associated:

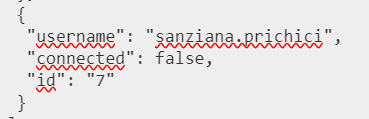
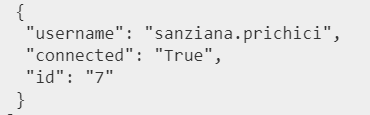


Again, this form will send these data by sending a post request to the login.php page. Here, after being encoded in a json format, the username and password go to a client redis run by a python script where we will verify the consistency in the database. If the password doesn’t match the username or the username doesn’t exist, we’ll throw an error message in the php page:

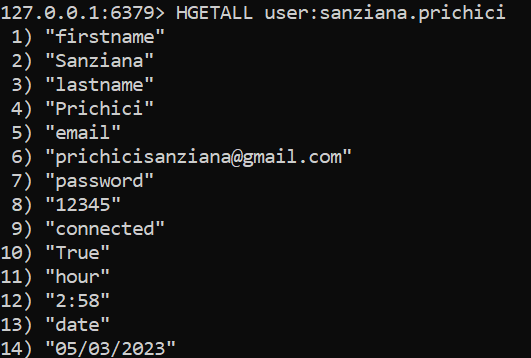


The login page is the one that changes the status of the connection from false to true in the database. Example on the sanziana.prichici user:

before login: after login:

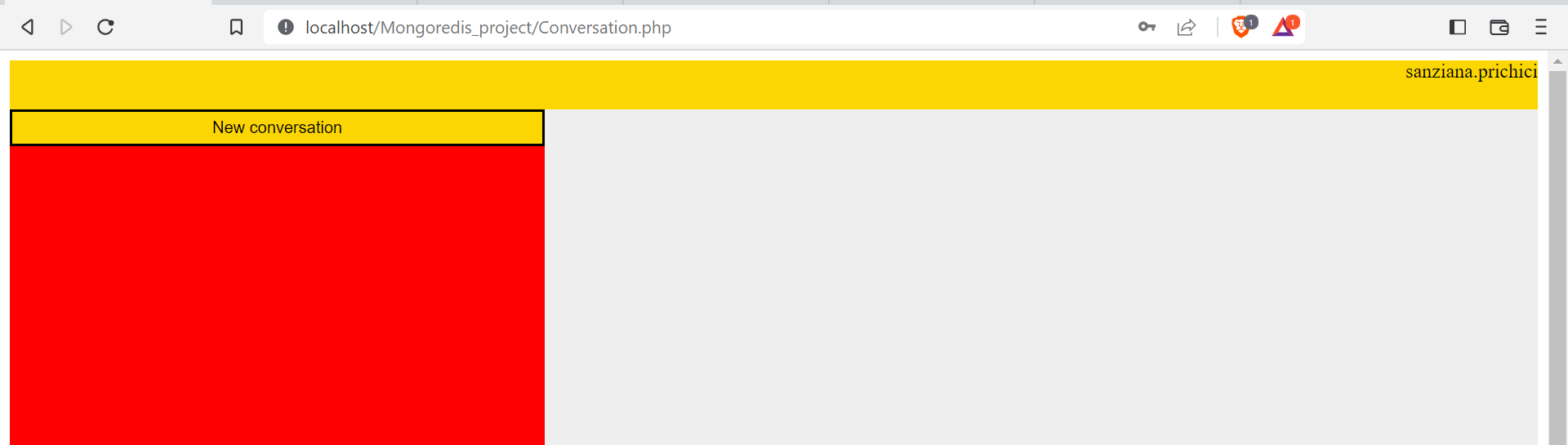
Also, in the redis database we will modify the “connected” field and we will also add the hour and date of the connection:



## Conversations

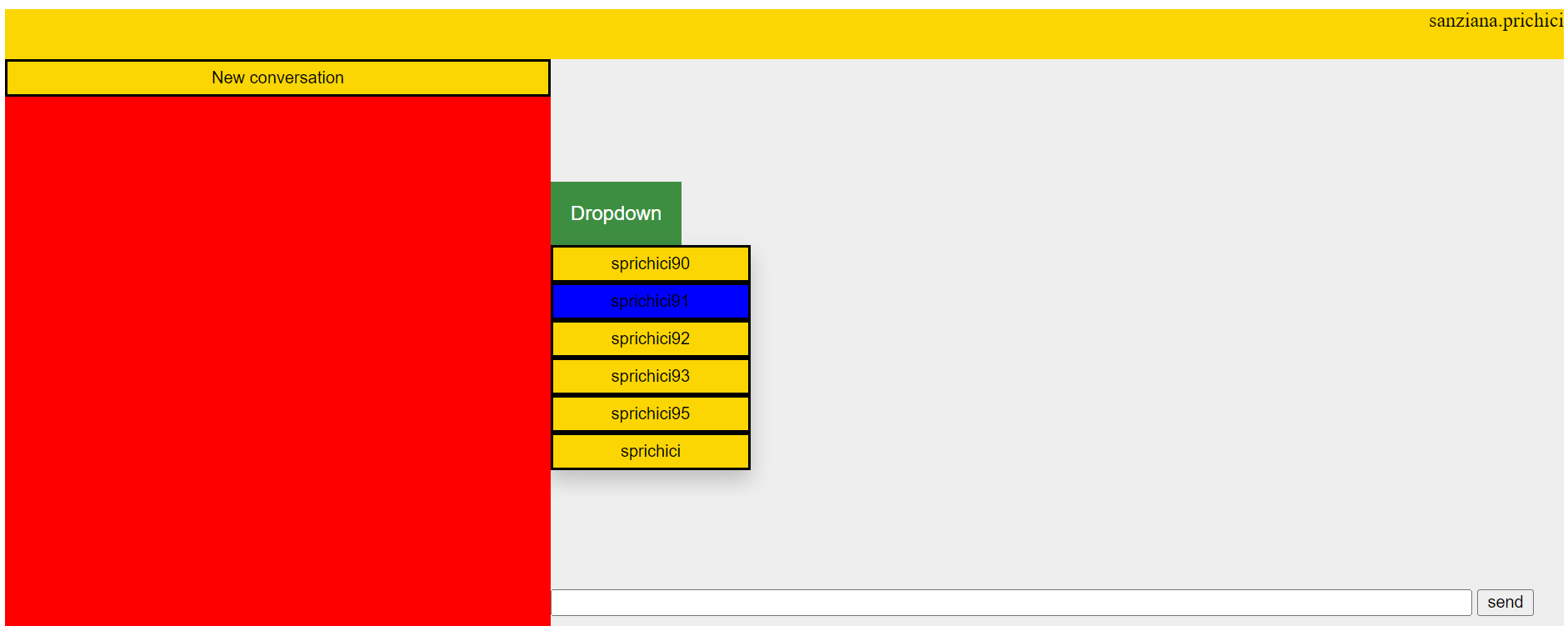
This is the actual page that sets the goal of the application. I will demonstrate the functionalities that I managed to add in the application by using the user created above: sanziana.prichici.

Firstly, after connecting in the app with this user, we can see that it doesn’t have any existing conversation with other users in the database:

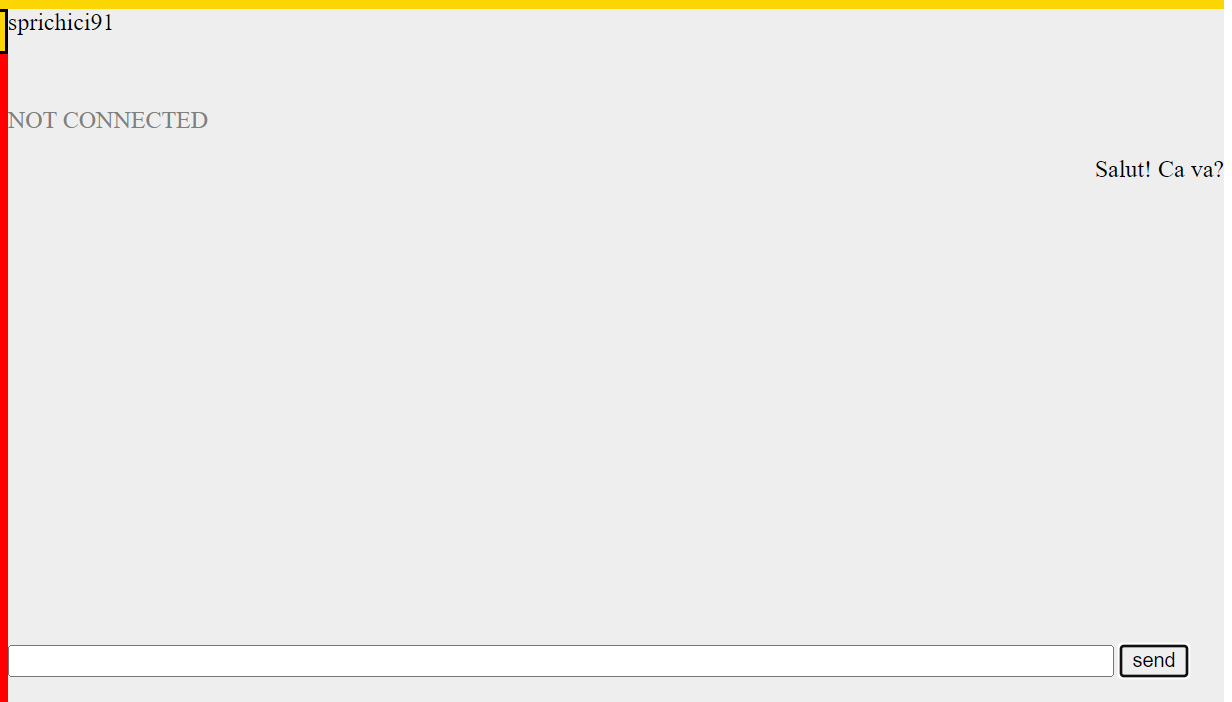


HERE IT WILL BE THE LIST OF CONVERSATIONS

So, in order to send a new message to a user from the database, we will action the “New conversation” button. Now, we will be displayed a dropdown list where we can see a list of possible users that we can choose. Note: this list will never contain the user in use or the users with whom he has already a conversation.

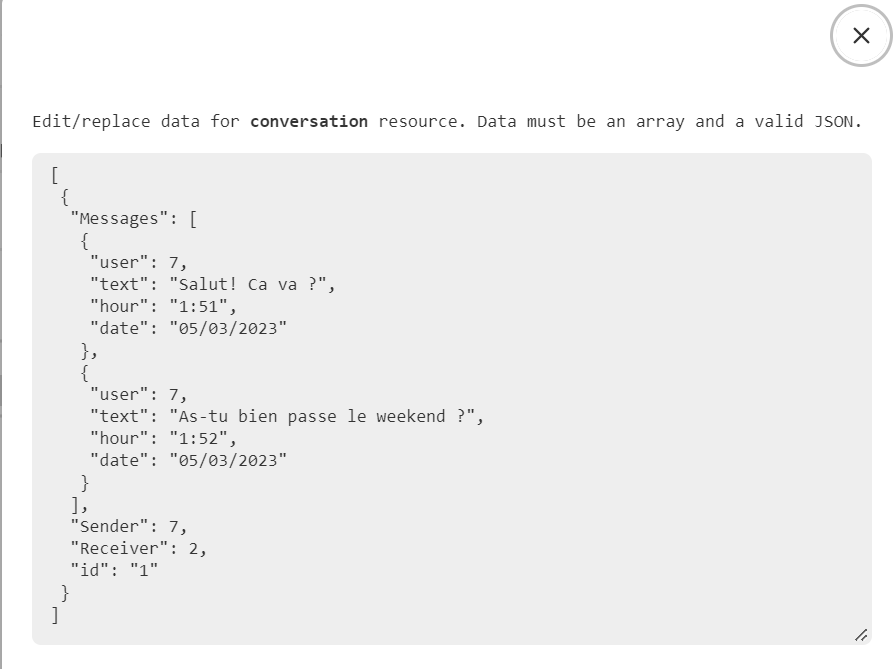


After choosing the user, we will be displayed the name and the status (connected or not) and then we can exchange messages by entering the text in the input box located on the bottom of the screen and by pressing the “Send” button.



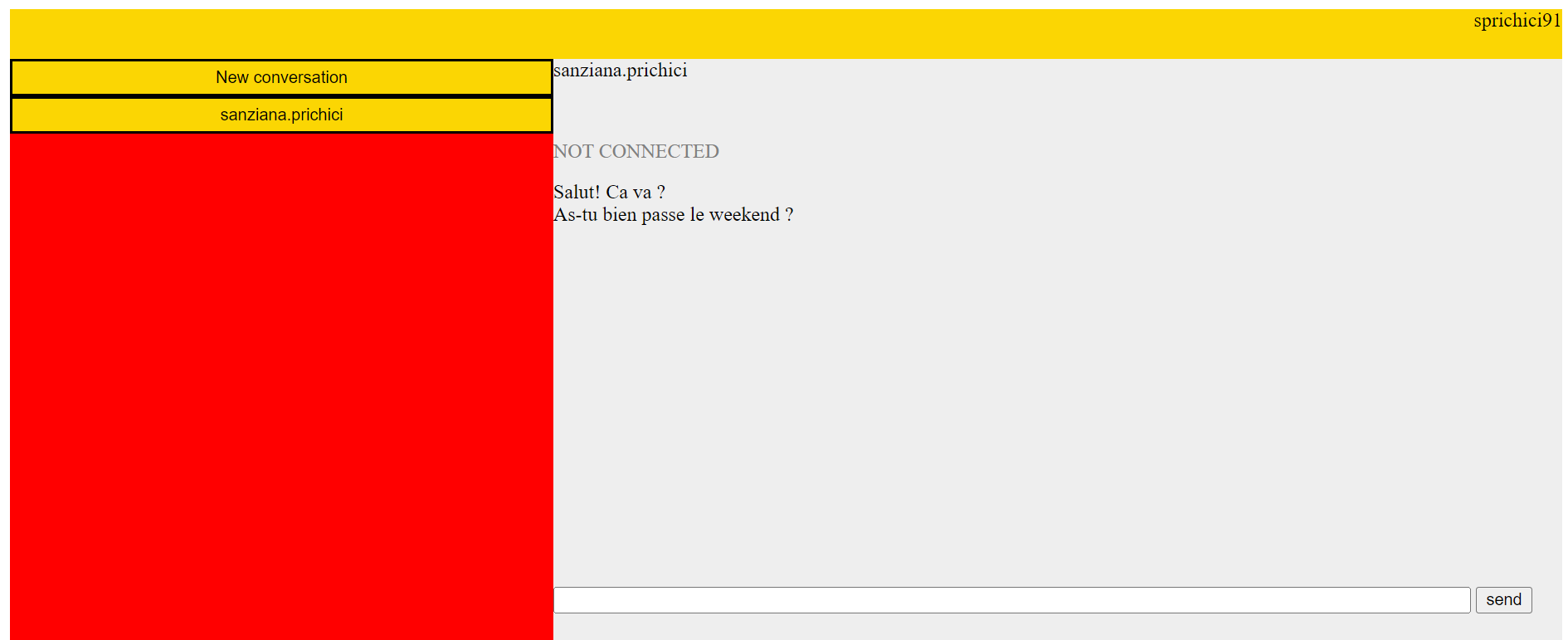
Now, we will check the mongodb database and the API in order to verify if the messages were stored correctly. The structure of a mongodb document must be: an array of messages where each message contains the id of the user that sent it, the text of the message, the hour and day of the message. In the document, the id of the *sender* and of the *receiver* will be required as well.

Mongodb: 

API: 

Note: the status of receiver/sender will not be changed in time, between users in the database, but the system knows how to correctly append a new message between the 2 users to the correct conversation.

For example, now if I connect as the sprichici91 user, I will see the conversation with sanziana.prichici in the list and also the messages sent:



Now, I will send a new message to sanziana.prichici in order to see that this conversation will be appended the correct message. Looking in the database, we can see that the message was successfully sent:

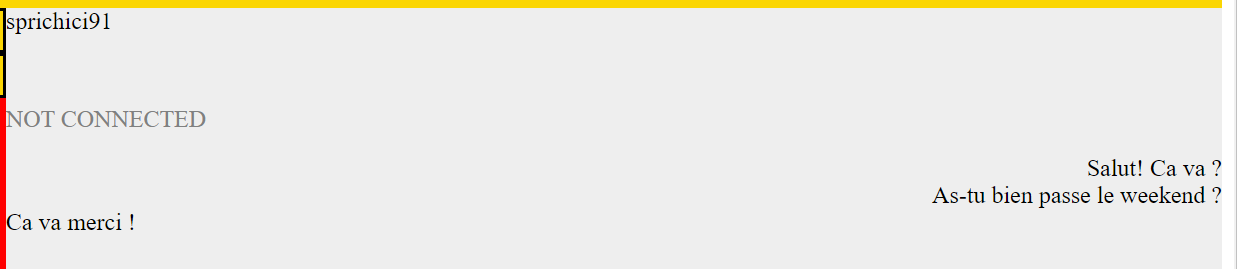


sanziana.prichici

sanziana.prichici

sprichici91

Final conversation:



We can see that the indentation is made to indicate the receiver/sender position. This is changing considering who is the authenticated user.

# Challenges and possible future developments

1. One of the first challenges was to verify the unicity of the email address for a new client. Because the username was, in fact, the key of the hash table where I stored the users in redis, this was easy to check. But in order to see of the email is not used, I had to declare a new redis **strict** client that allowed me to use the **scan**() function in order to extract all the hash tables having keys that respect the format „user:*x*” (where *x* is the username). After that, I just verified the list using a flag that indicated if the email was indeed unique or not.
2. Another challenge that I encounter was parsing the data sent by php in the python script. Here, the array that I previously sent is recognised as string and the json.dumps() function didn’t correctly decode my data. So, for that I had to use the split() function in python in order to distinguish the message from the user’s id. A big issue with this is that, for the moment, the application can’t support messages containing a comma.
3. Unfortunately, I couldn’t set the ReplicaSet for this project on my machine and I didn’t have the necessary time to spend on this matter.
4. The changing of the data types that were necessary in the php and python scripts were the most challenging parts of this developments.

This application is definitely not a viable final product. The reason why they are not present now in this project is the lack of time and experience with developing an app from scratch. In my opinion, there would be some future developments that could be done, such as:

* A logout page that would change the user’s status from connected to not connected;
* Integrating web sockets in order to allow users to message in real time;
* Refreshing of the list and conversation as soon as a new message is received;
* Adding the ReplicaSet.

1. For more information on establishing the unicity of the email address, check the [Challenges](#_Challenges_and_possible) chapter [↑](#footnote-ref-1)