
# Connect4

Report progetto Tecnologie e Applicazioni WEB

Facci Matteo 875377

## Introduction

In this report it is analyzed how ‘Connect4’ web application is made, taking into consideration all the important points.

## System component

Our ‘Connect4’ app is made by two important big parts: Angular application and Rest API server and a non-relational database using MongoDB.

Starting from backend to frontend, we have our database where we store all the information about client and matches, the two fundamental entities of our application. Every operation is made on matches and users and personally I didn’t have the necessity to create other collections. Maybe in a future with the implementation of other features I’ll need to add collection to satisfy a particular user’s request.

Next we have the Rest API server, it is the server where we can make http requests to retrieve informations from our models.

In the Rest API server I’ve used javascript modules to do the job easier.
In particular here are the modules that I’ve used are:

-dotenv: with this module we can get environmental variables from a file called .env. In particular, we use this module to store the jwt secret key, an attribute which we will use later. This module is useful even if you want to store DB configuration when using our web application in production mode.

-fs: when we need to open https cert.pem and key.pem files to make our self-signed certificate, we need to navigate in ‘key’ folder, so we use this module to search file in our file system.

-http and https: these two modules allow us to create a http and https server, to listen for our request to the endpoints on a certain port.

-colors: this module can be useful to categorize different type of message and associate them with a color in the CLI.

-mongoose: as we need to retrieve data from a MongoDB database, we need a module to get all the documents from MongoDBcollection. We have define

two typescript classes that we will use in the routes, to represent our collections in our REST API server.

-express: our REST API server's job is to serve us a list of a possible endpoints that we can call and express is made exactly for what we need. It exposes a list of routes where we can send GET, POST or DELETE requests.

-passport: authentication is mandatory when we are talking about reaching the endpoints. Passport can manage the login of a user, and allow us to authenticate user with basic and digest schemes in our application. We use passport "passport-http" module because it is easily integrable with express and let us use Basic and Digest authentication, used for login endpoint. More details about application authentication logic will follow later.

-jsonwebtoken: this module allow us to generate a jsonwebtoken to mantain a session about the user with useful information, that are kept even if the user quit from the browser. We can sign the token using the jwt secret key that we have set before in the .env file.

-express-jwt: is an important middleware that act as a validator of jsonwebtoken created with the previous "jsonwebtoken" module. This is an extremely important module, because with that we can receive from request params the information about the authenticated user that sent the request.

-cors: is a middleware for accessing to resources in requests in a different domain from the one in which we are serving the resources.

-pusher: last but not least, pusher is a module that we used to create . socket-connection between players when they want to send themself actions. We need to create an instance of pusher inserting the information that we got from the web interface and then we are ready to trigger events from the node server and bind them on our angular app.

As a frontend of our application I've used Angular framework, to create a SPA with MVC (Model View Controller) pattern where we can divide buisiness logic from presentation logic. A SPA can efficiently perform all the operations that the user need to use our web application. The Angular application use services to call the REST API endpoint and get all the data and make all the changes that are required.

## Data model description

As a was mentioning before, the MongoDB database store information about two fundamentals entities: users and matches.
So, I've created two collections "matches" and "users". Since we have a non-relational database I felt like I needed just these two collections because if i needed to add some information about a user or a match i could simply insert them inside the document and I do not need to create another table and relate them with a primary key - foreign key tuple (like we need to do in a relational database logic).

The User interface extends MongoDB Document and contains:

readonly _id, of type mongoose.Schema.Types.ObjectId: autogenerated id useful to uniquely identify the user;
username: string, it represents the username of the user;
name: string, it represents the name of the user;
surname: string, it represents the surname of the user;
moderator: boolean, boolean field that tells us if the user has a role moderator or not;
firstAccess: boolean, boolean field that tells us if the user has never logged in in. We use this to reset the credentials for added moderator with temporary credentials;
profilePic: string, a string that represents the base64 encode of the user's profile pic. We give it to the src attribute of img, so it can be read as an image;
salt: string, salt is a random string that will be mixed with the actual password before hashing to improve the level of randomicity of our password;
digest: string, digest of the clear password + salt, create a one-way hashing formula that can only be compare and not be decrypted (at the moment) as we are using the sha 512 hash function;
win: number, number of wins of the user;
loss: number, number of loss of the user;
draw: number, number of draw of the user;
friends: string[], all the friends of the user;
pendingRequests: string[], all of user's friends request (can be visible only to the user himself/herself);
isLookingForAMatch: boolean, tells us if the user is looking for a match with a random player;
setPassword: (pwd:string)=>void, method that create a digest by hashing together clear password and a random salt

validatePassword: (pwd:string)=>boolean, method that validate the password created at the registration by comparing the digest previously created with another created using the same salt;
setDefault: ()=>void, set win, loss and draw of the user to 0 at the moment of the creation.

The Match interface extends MongoDB Document and contains:
player1: string, the first player playing the match;
player2: string, the second player playing the match;
turn: string, the player that it's allowed to play at the moment;
winner: string, the winner of the match, set null at the beginning of the match and set null if the match is drawn;
loser: string, loser of the match;
ended: boolean, indicates if the match is ended or not;
private: boolean, if the match is private or not. Match executed between friends are private and not spectatable from other players.

## REST API description

description: return all the users registered
route: /users
method: GET
token required: yes;
success:

```
users
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: add a user to the db
route: /users
method: POST
token required: no;
success:

```
{ error: false, errormessage: "",message: "User successfully added with the id below", id: data._id, token: token_signed }
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: add a moderator to the db
route: /users/addModerator
method: POST
token required: yes;
success:

```
{ error: false, errormessage: "",message: "Moderator successfully added with the id below", id: data._id }
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: search user to send friendship request
route: /users/searchForUsers
method: GET
token required: yes;
success:

```
{users: users}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: reset credentials of new moderator
route: /users/setModerator/
method: POST
token required: yes;
success:

```
{error: false, errormessage:"", message: "User " + req.user.username +
" correctly updated"}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return user profilepic in base64
route: /users/:username/profilepic
method: GET
token required: yes;
success:

```
{"profilepic": response.profilePic}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: set user firstAccess to false
route: /users/setFirstAccess
method: GET
token required: yes;
success:

```
{"message": "Correctly setted user first access.", token: token_signed}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: pair a user with a random one searching in a match
route: /users/pairUserForAMatch
method: GET
token required: yes;
success:

```
{user:response[index]}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return friends of a certain user with stats associated
route: /users/friendsWithStats
method: GET
token required: yes;
success:

```
{result:friends_with_stats}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return all users with stats
route: /users/allUserWithStats
method: GET
token required: yes;
success:

```
{result:users_with_stats}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return a user with a certain username
route: /users/:username
method: GET
token required: no;
success:

```
{user: response}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: delete a user with a certain username
route: /users/:username
method: DELETE
token required: yes;
success:

```
{"message": 'User ' + req.params.username + ' successfully deleted from the DB'}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return the stats of a certain user
route: /users/:username/stats
method: GET
token required: yes;
success:

```
{stats:stats}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: set if user is looking for a match
route: /users/setLookingForAMatch/:value
method: GET
token required: yes;
success:

```
{message: "User state of looking for a match setted: " +
req.params.value}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return friends of a certain user
route: /users/:username/friends
method: GET
token required: yes;
success:

```
user.friends
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return friends' requests of a certain user
route: /users/:username/friendsRequests
method: GET
token required: yes;
success:

```
{friendsRequests:userList.pendingRequests}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: send friendship request to user
route: /users/sendFriendship/:username
method: GET
token required: yes;
success:

```
{message: "Friend request sent to " + req.params.username}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: accept friend request of a certain user
route: /users/acceptFriendship/:friend
method: GET
token required: yes;
success:

```
{message: "User " + req.user.username + " added friend " +
friendUser.username + " correctly" }
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: reject friend request of a certain user
route: /users/rejectFriendship/:friend
method: GET
token required: yes;
success:

```
{ statusCode:500, error: true, errormessage: "Couldn't delete "+
req.params.friend + " from pending requests. " + error}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return all matches
route: /matches
method: GET
token required: yes;
success:

```
matches
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: inserting a new match
route: /matches
method: POST
token required: yes;
success:

```
{message:'New match correctly added', id: matchCreated._id, match:
matchCreated}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description:  return all the active matches
route: /activeMatches
method: GET
token required: yes;
success:

```
matchesArray
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return a certain match given a defined ID
route: /matches/:id
method: GET
token required: yes;
success:

```
result
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return which players are playing the match
route: /matches/:id/players
method: GET
token required: yes;
success:

```
{players:[players.player1, players.player2]}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: return who is allowed to do the next move
route: /matches/:id/turn
method: GET
token required: yes;
success:

```
{turn:result.turn}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: set the match drawn
route: /matches/:id/setDraw
method: GET
token required: yes;
success:

```
{message: 'Added draw to ' + result.player1 + ' and ' + result.player2
+ ' then setted the match ' + myId + ' drawn.'}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: set the loser of the match
route: /matches/:id/setLoser
method: GET
token required: yes;
success:

```
{message:'Winner ' + opponent + ' of match ' + req.params.id + ' setted
correcty.\n Loser ' + req.user.username + ' of match ' + req.params.id
+ ' setted correcty.'}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher chat API
route: /messages
method: POST
token required: yes;
success:

```
{username:req.user.username, message:req.body.message}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher finding match API
route: /matchFound
method: POST
token required: yes;
success:

```
{message:"match found", username: req.user.username,
against:req.body.challenged}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher Connect4 API to make a move in the match
route: /doMove
method: POST
token required: yes;
success:

```
{message:"move executed on match " + req.body.matchId + " and updated
turn, now is " + user_turn + " turn.", columnIndex:
req.body.columnIndex, playerIndex:req.user.username}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher Connect4 API to communicate the loss
route: /communicateLoss
method: POST
token required: yes;
success:

```
{
message:"User " + req.user.username + " has declared his loss",
winner:allowed_players[allowed_players.indexOf(req.user.username) == 1
? 0:1],
loser:req.user.username}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher Connect4 API useful to a spectator to request the match state
route: /requestState
method: POST
token required: yes;
success:

```
{message:"User " + req.user.username + " has requested to spectate the match"}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher Connect4 API to send the match state
route: /sendState
method: POST
token required: yes;
success:

```
{message:"User " + req.user.username + " has requested to spectate the match"}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: Pusher chat API to manage friends requests
route: /friendRequests
method: POST
token required: yes;
success:

```
{username:req.user.username, message:req.body.message}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: log the user inside a webapp
route: /login
method: GET
token required: yes;
success:

```
{ error: false, errormessage: "", token: token_signed }
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

description: handle invalid endpoint
route: Invalid endpoint
method: GET
token required: yes;
success:

```
{statusCode:404, error:true, errormessage: "Invalid endpoint"}
```

failed:

```
{statusCode:code, error:true, errormessage: "message"}
```

## Authentication management and workflow

On the REST API Server:
Whenever a user request the login endpoint the passport module request a basic authentication and so we are required to send an Authorization header, which doesn't request session support (so we can leave the option session:false). Then the “Passport” module uses passportHTTP.BasicStrategy where we can define a function with the logic to make the user authenticated on some conditions. We pass username, password and done, a function that we call whether the authentication is successful or not, returning a different response as well. Inside the BasicStrategy we verify that validatePassword method of user returns true, so we can be sure that the digest generated corresponds with the one stored in the db, for that user.

On the Angular web application:
In order to provide REST API calls to the server we have an angular service called client-http that acts as controller, where we can manage the data and project that on the templates.

When the user lands on “login” route he has to enter his credentials.
At that point we’ll call a method in client-http service called “login”, that call the login endpoint in the REST API server and make the basic authentication. If the credentials match, the user is logged in into the system, if not an error message is displayed, and we ask the user to retry. If the user is a moderator and it’s his first access, when he successfully login he is redirected to a route where he can change his credentials (name. surname and password, because the previous one was temporary), and then he’s redirected to the homepage too. When the user is logged in a token is associated to “connect4_token” and he is remembered in the site until the token is expired(after he will need to login again). The token has 24h of duration. The user has the possibility to remember his identity in the login page, if he does not do that, after the browser close or on browser reload, he will have to login again. A special service, “authguardservice” is responsible for checking on every route if the user is authorized to access the endpoint. If the user is not authorized, he will be redirected to login page.

## Angular frontend description

Our Angular application is made by components and services, that arrange the data and serve them to us from templates.

Components:

bubble-menu: redirect the user in basics homepage actions (user stats, friends list, observe a game, search for a friend);
friends-list: shows to the user the friend list associated with the stats and if the user is a moderator, it shows all the possible users and their stats in the entire system;
friendship-requests: allow the user to accept or reject the incoming friendship requests;
home: the component associated to the main template of our application, containing the bubble-menu, chat and the possibility to challenge the users, privately or not;
login: perform login calling passport.BasicStrategy in REST API server;
manage-users: available to a moderator only, we can delete every user in the application in the associated template;
banner-info: show on the banner info (on the top of the screen) the match evolution. Initially it only display the turn and when the match is over it displays the winner or draw if no one of the players won;
board:manage the logic of what to show from the board during the match. The board will be updated on every move the users do;
chat: chat component for chatting among users. Can be used in the homepage, sending messages on global or to a friend or can be used in a single match. The observers of a match can chat only among themselves and the players of a match can chat by themselves too.
disk:
match: contain the components (banner info and board) necessary to play a match;
matches-list: can be used to see every non private match list in the system;
page-not-found: show a not-found-page messages when a user try to reach a non existing endpoint;
register-moderator: available to a moderator only, can register new moderators with temporary passwords;
reset-credentials: make new moderator reset credentials (name, surname, password and profile pic) at the first login;
search-friends: we can search users in the system with this component, and send them a friendship request;

search-matches: we can search and observe all the non private not ended matches in the system;
signup: allow to user to register himself in the system;
stats: show user's stats (win, losses, draws, played matches).

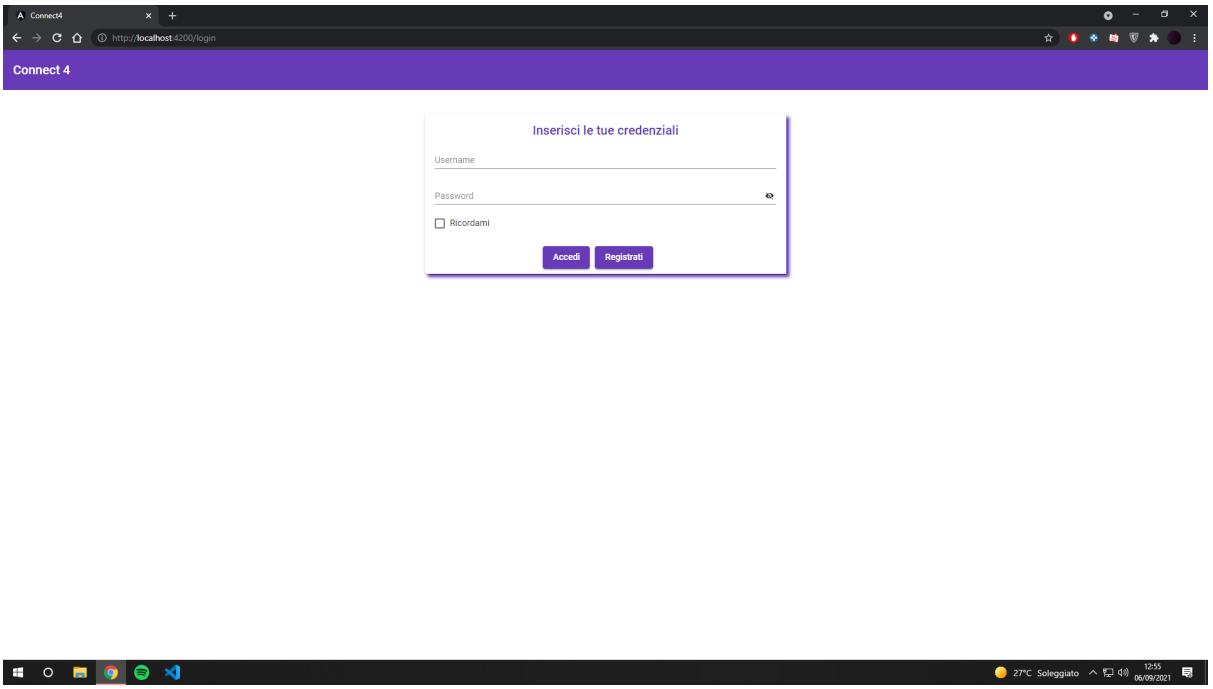
Services:
connect4: this service manages all the logic about the match and the state of a toolbar. If the user is in a match he cannot see toolbar icons and cannot perform actions.
app-service: it allow us to mute the system while playing or to change to dark mode (not used yet);
audio: it bind an audio and execute it every time the user do a move in a match or the user wins the match;
breakpoint: manage screen dimensions on different devices while in a match,
theming: useful to switch between dark and normal mode (not used yet)
authguard: manage the routes' authorization system by checking jwt;
client-http: contain all the methods to make user perform actions in the system. It calls http request to the REST API server;
matches: contains all the methods to manage match informations. It calls http request to the REST API server;
messages: contains methods to make the users chat themself and notify during a accepted friend request;
moderatorguard: manage the routes' authorization system by checking if a user is a moderator;

Routes:

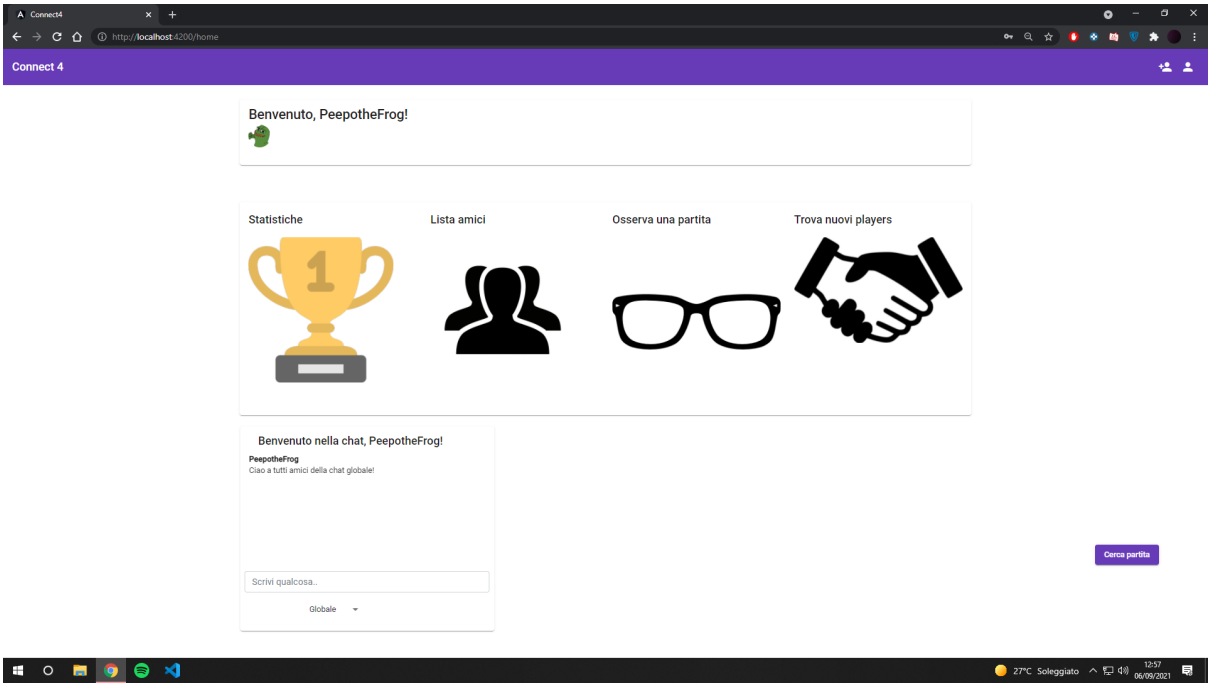
```
{ path: '',   redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent,
canActivate:[AuthguardService]},
  { path: 'home/stats', component: StatsComponent,
canActivate:[AuthguardService]},
  { path: 'home/friends', component: FriendsListComponent,
canActivate:[AuthguardService]},
  { path: 'home/searchFriends', component: SearchFriendsComponent,
canActivate:[AuthguardService]},
  { path: 'home/matchList', component: MatchesListComponent,
canActivate:[AuthguardService]},
  { path: 'friendshipRequests', component: FriendshipRequestsComponent,
canActivate:[AuthguardService]},
```

```
  { path: 'manageUsers', component: ManageUsersComponent,
canActivate:[AuthguardService, ModeratorguardService]},
  { path: 'match/:id', component: MatchComponent,
canActivate:[AuthguardService]},
  { path: 'signup', component: SignupComponent },
  { path: 'login', component: LoginComponent },
  { path: 'reset', component: ResetCredentialsComponent,
canActivate:[AuthguardService, ModeratorguardService]},
  { path: 'registerModerator', component: RegisterModeratorComponent,
canActivate:[AuthguardService, ModeratorguardService]},
  { path: '**', component: PageNotFoundComponent} // Has to be the last
routes path
```

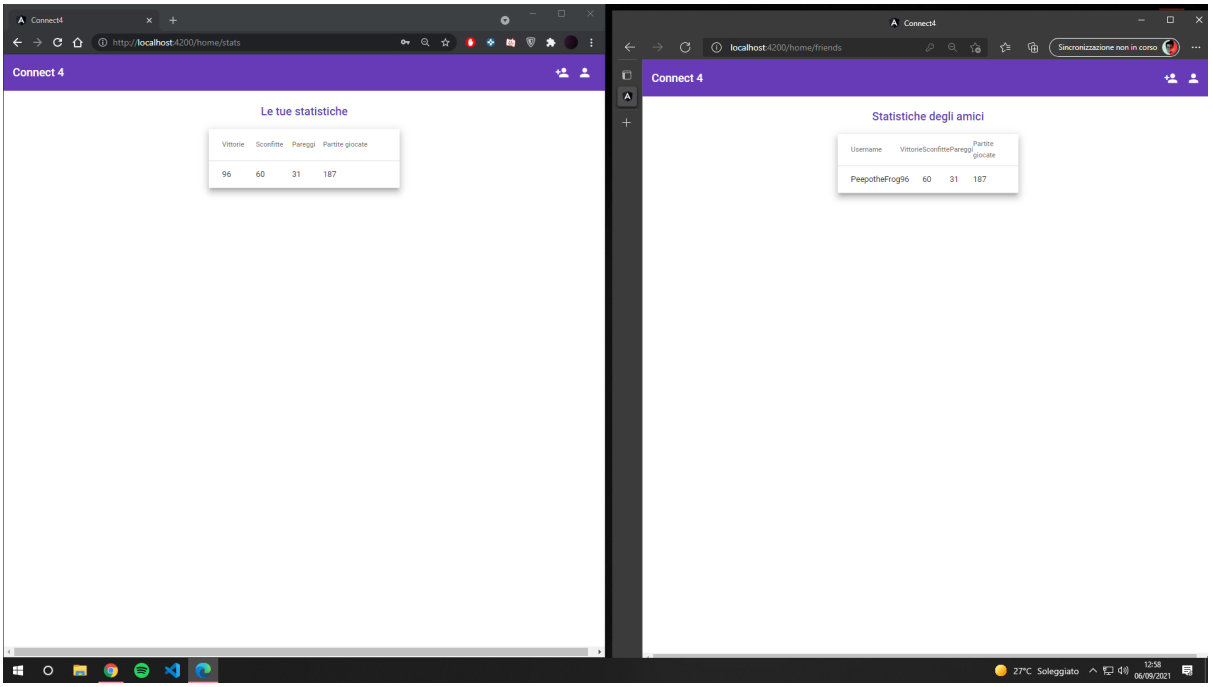
# Typically user workflow



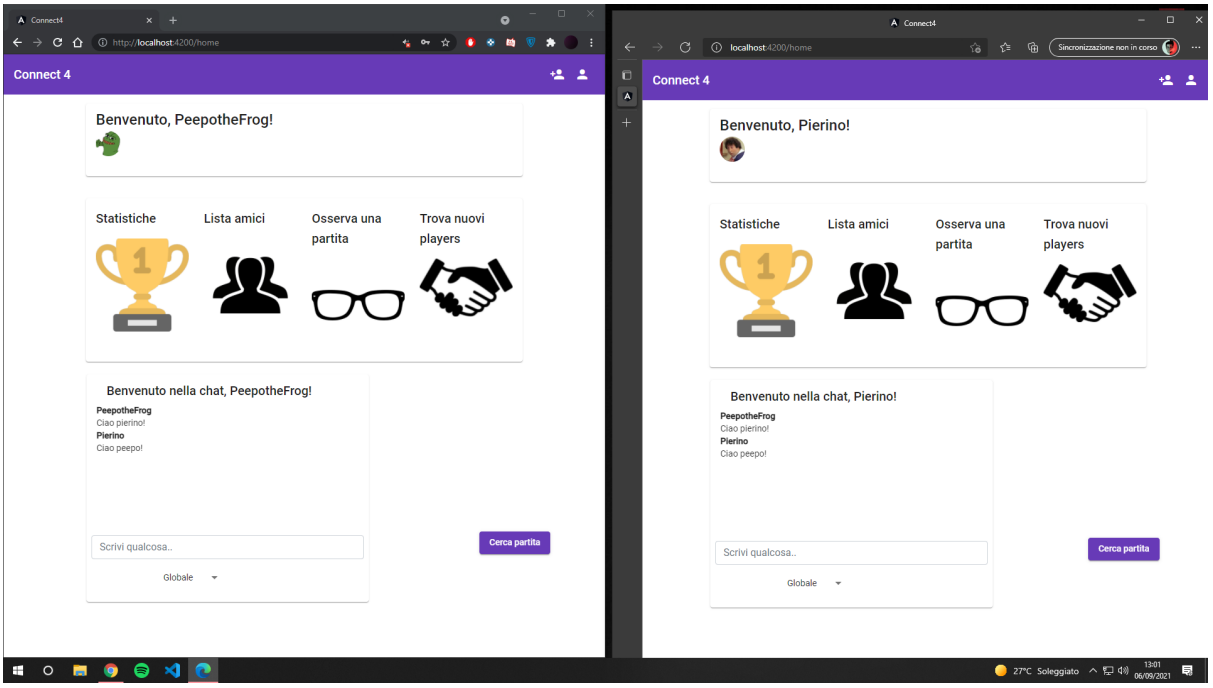

User can log in with his credentials.



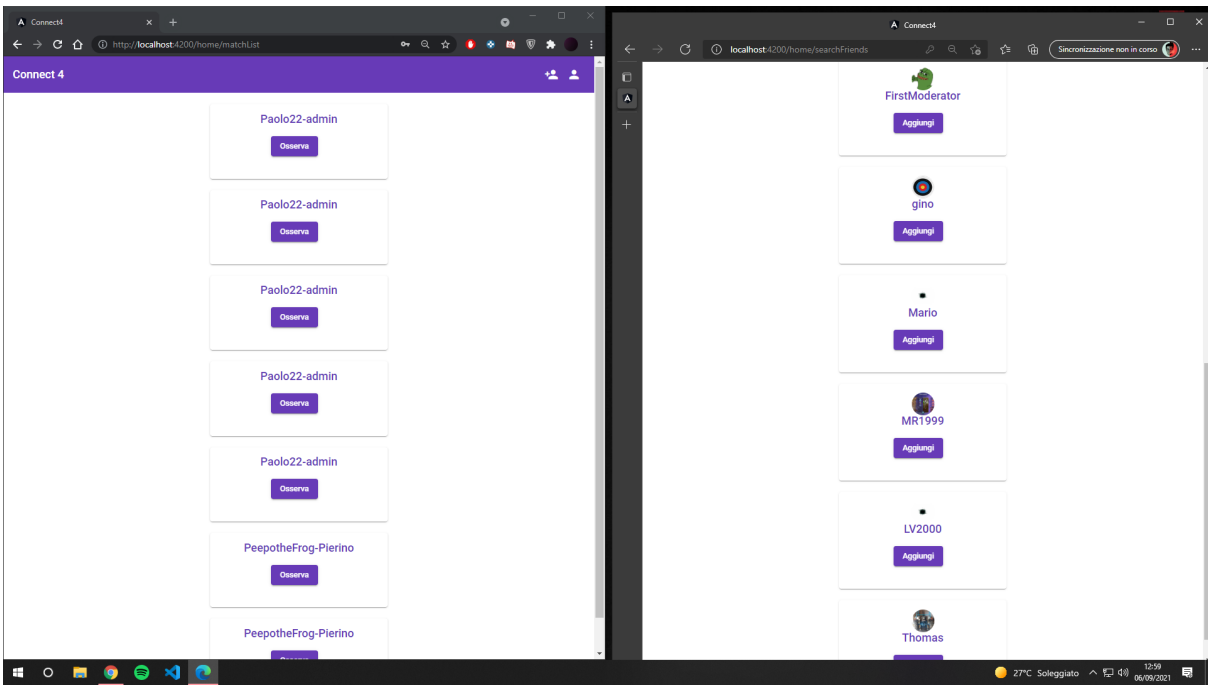

The main homepage, where the user can perform multiple actions.



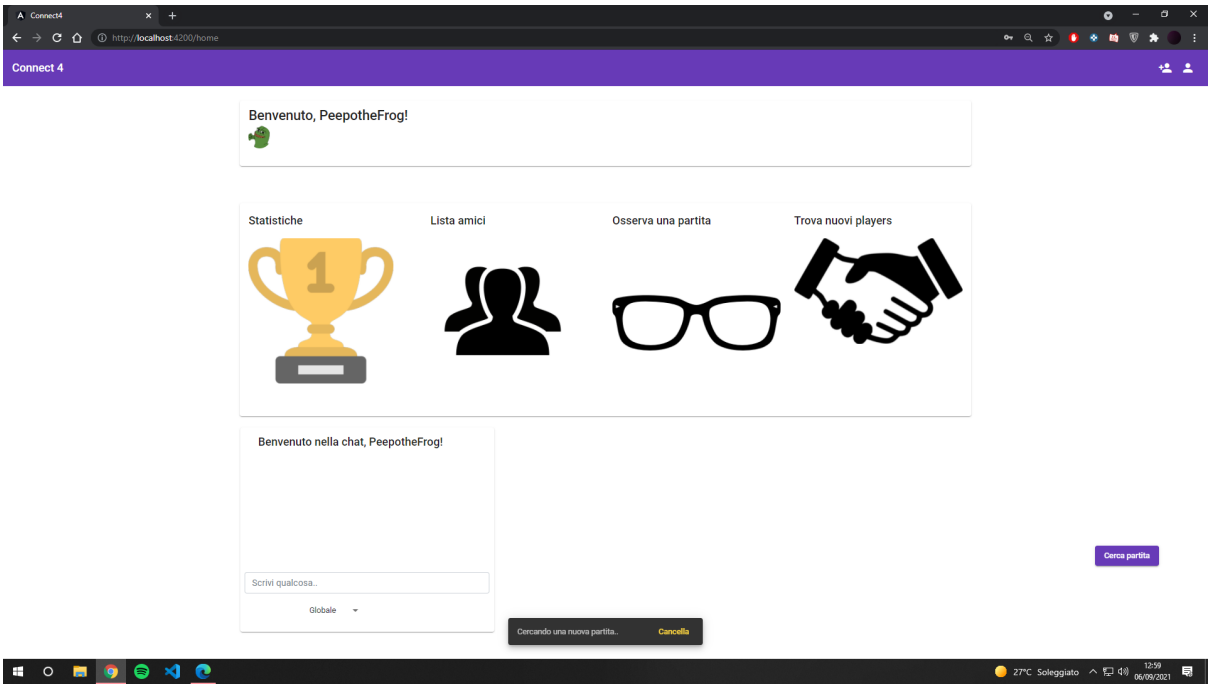

User stats on the left and all friends stats on the right.



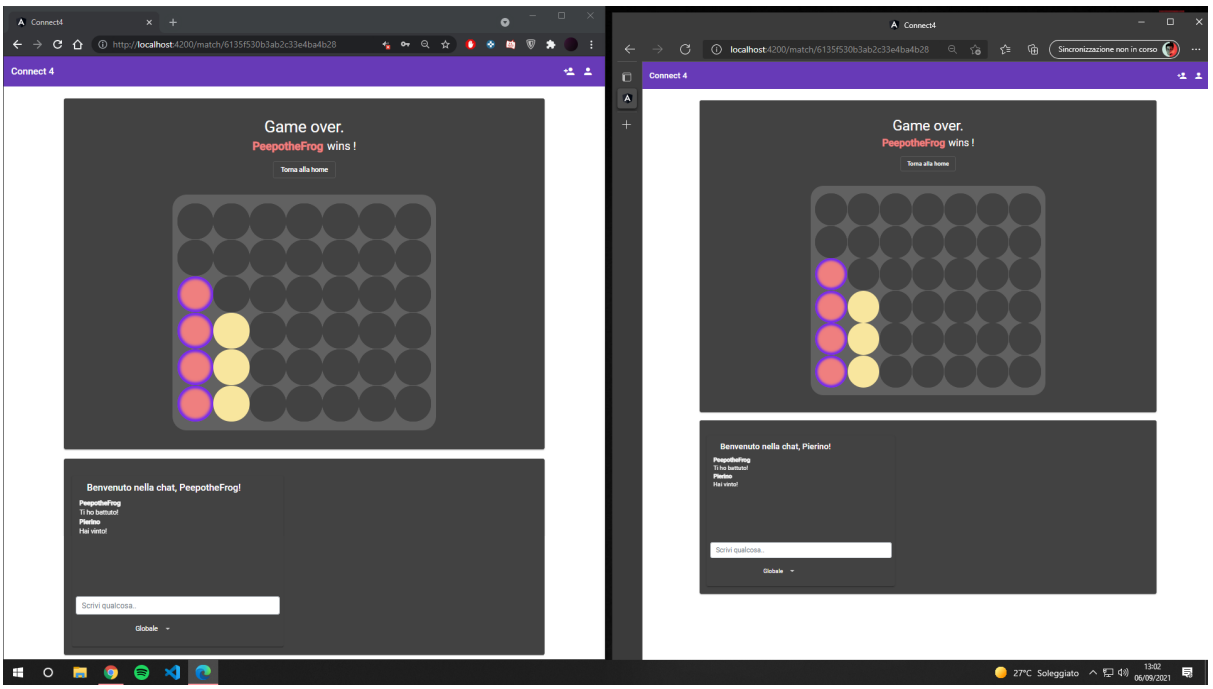

User chatting in global chat.




Active matches list on the left and search friends on the right.




User is currently looking for a match with random




Match execution with chat among users. The user PeepotheFrog has just won the game.