Davide Pizzolato
!giveUp(you)
881347@stud.unive.it

Just 4 Fun

Application Design Report

Contents

1 Architecture and components

1.1 General

The purpose of this document is to illustrate and explain the design choices made during the creation of **just4fun**.

Just4fun is a web application that allows you to play *connect 4* in multi-player.

It was developed in **typescript** with the **mean** stack and is being served in the form of a **spa** and a **rest api**.

1.2 Components

The first and biggest separation is between **frontend** and **backend** which appear as two separate projects. Then we have the data storage handled by a **mongoDB** non-relational database and other minor components that contribute to the realization of the project.

1.2.1 Backend

In backend **Node.js** provides the runtime environment and permits javascript code execution in a desktop environment.

To facilitate the creation of the api we used **Express.js** which is a middleware based framework that provides useful tools to build rest endpoints.

In the backend there is also the **mongoose** library that allowed us an easy definition of the database schemas and a comfortable access to it.

1.2.2 Frontend

The frontend consists in a **SinglePageApplication** created with **Angular**.

For the graphic design we used **Bootstrap** and **Fontawesome**.

1.2.3 Authentication

The authentication mechanism is based on jwt tokens and managed by the **passport** library.

1.2.4 Websocket

There is also a real-time bilateral communication system between client and server based on websocket and provided by **Socket.io**

2 Database schemas

2.1 User

```
User {
    username: string,
    email: string,
    points: number,
    following: string[],
    friends: string[],
    friendRequests: string[],
    matchInvites: string[],
    roles: string[],
    salt: string,
    digest: string,
    isPasswordTemporary: boolean,
    avatar: string,
    isDeleted: boolean,
    notifications: {
        type: string,
        content: Object,
    }[],
    hasNewNotifications: boolean
}
```

Listing 1: User data model

Each document contains the representation of an application user.

When, both in this and in other documents, we refer to a user we use the email which is unique and immutable.

2.1.1 Attributes

Below a description of the non-trivial attributes:

points A score that indicates the experience/skill of a player.
(Points are won/lost through matches)

friendRequests List of received friend requests.

matchInvites List of received game invitations.

roles List of roles (the only one implemented is "MODERATOR", but being an array makes it open to future developments)

isPasswordTemporary If true the registration was not completed and before you can login, you need to complete the account with the missing information and change the password.

avatar Base64 encoded user profile image.

isDeleted If true this user is no longer authorized to perform any action and will no longer be returned by the API.

notifications User notifications list
type Can be: 'follow', 'request', 'invite', 'acceptedInvite', 'message', 'system'.

hasNewNotifications Indicates if the user has unread notifications.

2.2 Chat

```
Chat {
    matchID: string,
    members: string[],
    messages: {
        sender: string,
        text: string,
        timestamp: number
    }[]
}
```

Listing 2: Chat data model

Each document contains a representation of a chat and stores its messages.

2.2.1 Attributes

Below a description of the attributes:

matchID If the chat is relative to a game this attribute is valorized otherwise null.

members Contains the list of users authorized to read and send messages, in case of public chat (like match chat) members is null.

messages List of messages.

sender E-mail of sender.

text Text of the messagge.

timestamp Send datetime.

2.3 Match

```
Match {
    player0: string,
    player1: string,
    winner: {
        player: number?,
        positions: number[][]
    },
    turn: number,
    board: number[][],
    moves: number[],
    matchStart: Date,
    lastMove: Date
}
```

Listing 3: Match data model

Each document contains a representation of a match.

2.3.1 Attributes

Below a description of the attributes:

player0 The first player, he makes the first move.

player1 The second player.

winner Information about the result of the game.

player null=Game in progress, -1=Tie, 0/1=Game win by player0/player1

positions A list of four positions [row, column] indicating the four winning cells.

turn 0/1=The next move should be by player0/player1
The first move is always by player0.

board It represents the current playing field, or the final playing field if the game is over.

moves A list of moves. In each cell there is the number of the selected column, remember that it is always player 0 that starts.

matchStart Match start datetime.

lastMove Last move datetime.

2.4 Matchmaking

```
Matchmaking {
  playerID: string,
  min: number,
  max: number,
  timestamp: Date
}
```

Listing 4: Matchmaking data model

Each document contains a representation of a match search.

2.4.1 Attributes

Below a description of the attributes:

playerID User looking for a match.

min - max Player is looking for an opponent whose score is between min and max. Periodically during the search min and max are expanded.

timestamp Search start datetime.

3 API Description

3.1 General

3.1.1 Order by

Where you can use the `order_by` get parameter you can insert every field name to get the list sort **ASC** by that field or by prepending the **-** (minus sign) to the field name you can have **DESC** sort.

3.1.2 Limit

The `limit` get parameter accept a positive integer.

The limit value is used to specify the maximum number of results to be returned.

3.1.3 Skip

The `skip` get parameter accept a positive integer.

The skip value specifies the maximum number of documents to skip. Together with limit it can be used for pagination.

3.2 Index

3.2.1 `GET /`

Authentication: Anonymous

This endpoint always respond with a **200** status code, the api version and the list of object present. The api version follow the semver specification.

```
{
    "statusCode":200,
    "api_version":"1.0.0",
    "endpoints":["/chat", "/match", "/user"]
}
```

Listing 5: Response

3.2.2 `GET /login`

Authentication: Basic

Here you can obtain a token.

If you insert invalid credentials you will receive a **401** status code.

If you try to login with a temporary password you will receive a **422** status code and a message that ask you to change your password, you can do it at `PUT /user/:email`.

In case of success you will obtain a Jwt Token.

```
{
    "statusCode": 200,
    "error": false,
    "errormessage": "",
    "token": "eyJhb..."
}
```

Listing 6: Success response

3.3 User

3.3.1 `GET /user`

Authentication: Jwt, Anonymous
Order by: ✓ **Limit:** ✓ **Skip:** ✓

It return all the non-deleted user. You could use the `username` get parameter to search for user. You will get an array of these objects:

```
{
    "_id": string,
    "email": string,
    "username": string
    "points": int,
    "matchInvites": string[],

    // Information available only if you are a moderator //
    "following": string[],
    "friends": string[],
    "friendRequests": string[],
    "roles": string[]
    // ------------------------------------------------ //
}
```

Listing 7: User model

3.3.2 `GET /user/leaderboard`

Authentication: Jwt, Anonymous

It is a static redirect to **/user?limit=10&order_by=-points**.

3.3.3 `POST /user`

Authentication: Jwt, Anonymous

Here you can register yourself or a new moderator if you are a moderator. In response you would have the *_id* of the created user.

You should put all the information on the body like this:

```
{
    "email": string, //Should be unique and not empty
    "password": string,  //Can't be empty
    "moderator": boolean?, //False is default

    // Information not required if moderator is true //
    "username": string, //Unique, not empty and no whitespace
    "avatar": string  //Can't be empty
    // ------------------------------------------------ //
}
```

Listing 8: Input information

3.3.4 GET /user/:email

Authentication: Jwt, Anonymous
Order by: ✓ **Limit:** ✓ **Skip:** ✓

It return if present the user with the specified email.
Model:

```
{
    "_id": string,
    "email": string,
    "username": string
    "points": int,
    "matchInvites": string[],
    "avatar": string,

    // Information available only if you are a moderator //
    // or if you are the user itself                     //
    "following": string[],
    "friends": string[],
    "friendRequests": string[],
    "friendRequestsSent": string[],
    "roles": string[],
    "notifications": string,
    "hasNewNotifications": boolean
    // ------------------------------------------------- //
}
```

Listing 9: User model

3.3.5 PUT /user/:email

Authentication: Basic, Jwt

Here you can change your information. You can only change your account. If you want to change your password you must use basic authentication, see Basic HTTP.
Model:

```
{
    "username": string,
    "avatar": string,
    "password": string,
}
```

Listing 10: Input information

3.3.6 DELETE /user/:email

Authentication: Jwt

If you are a moderator you can delete non-moderator user. If you call this endpoint against a moderator or if you are not a moderator you will receive a **403** status code.

3.3.7 POST /user/:yourEmail/follow

Authentication: Jwt

Here you can follow some other user. You must call this endpoint on your user and insert the user you want to follow on body.

Model:

```
{ "user": string }
```

Listing 11: Input information

3.3.8 DELETE /user/:yourEmail/follow/:followedEmail

Authentication: Jwt

Here you can unfollow users.

3.3.9 POST /user/:yourEmail/friend

Authentication: Jwt

Here you can send and accept friend invite. If the other user already sent you a friend request you will accept it otherwise you are sending him a friend request. You must call this endpoint on your user and insert the user you want to become fiend with on body.

Model:

```
{ "user": string }
```

Listing 12: Input information

3.3.10 DELETE /user/:yourEmail/friend/:friendEmail

Authentication: Jwt

Here you can refuse friend request or remove a user from your friends.

3.3.11 POST /user/:yourEmail/invite

Authentication: Jwt

Here you can send match invite. You must call this endpoint on your user and insert the user you want to invite on body. You can only invite friends.

Model:

```
{ "user": string }
```

Listing 13: Input information

3.3.12 DELETE /user/:yourEmail/invite/:sender

Authentication: Jwt

Here you can refuse match invites.

3.3.13 `DELETE /user/:yourEmail/notification/:idNot`

Authentication: Jwt

Here you can delete a notification.

3.3.14 `DELETE /user/:yourEmail/newNotifications`

Authentication: Jwt

Here you can mark all your new notifications as readed.

3.4 Chat

3.4.1 `GET /chat`

Authentication: Jwt, Anonymous
Order by: ✓ **Limit:** ✓ **Skip:** ✓

It return a list of chat. By default it will show public chat, you could search private chat by members but only if you are inside too. You could also specify in get parameters a specific `matchID` or set `matchID=null`.

`?user=my-email`	Get all the private chat with us inside
`?user=my-email&user=friend-email`	Get all the chat with us and friend-email
`?matchID=id`	Get the chat relative to match *id*

You will get an array of these objects:

```
{
    "_id": string,
    "matchID": string?,
    "members": string[]?,
}
```

Listing 14: Chat model

3.4.2 `GET /chat/:id`

Authentication: Jwt, Anonymous

Here you can get a chat and it's messagges. You can access to public chat or to chat where you are a member otherwise you will get a **403** status code.

Response Model:

```
{
    "_id": string,
    "matchID": string?,
    "members": string[]?
    "messages": [
        {
            "_id": string,
            "timestamp": Date,
            "sender": string,
            "text": string
        },
    ]
}
```

Listing 15: Chat model

3.4.3 `POST /chat/`

Authentication: Jwt

Here you can create a new chat. You must be in members and all other members must be your friend otherwise you will get a **403** status code.
Body content:

```
{
    "members": string []
}
```

Listing 16: Input information

3.4.4 `GET /chat/:id/message`

Authentication: Jwt, Anonymous
Order by: ✗ **Limit:** ✗ **Skip:** ✗

It return a list of messages. You can access public chat or chat where you are a member otherwise you will get a **403** status code. By default it return all the messages but you can specify `afterTimestamp` get parameter and obtain only the messages newer than that timestamp.
You will get an array of these objects:

```
{
    "_id": string ,
    "timestamp": Date ,
    "sender": string ,
    "text": string
}
```

Listing 17: Message model

3.4.5 `POST /chat/:id/message`

Authentication: Jwt

Here you can send messages. You can access public chat or chat where you are a member otherwise you will get a **403** status code. Sender must be your mail or you will get a **403** status code.
Body content:

```
{
    "sender": string ,
    "text": string
}
```

Listing 18: Input information

3.5 Match

3.5.1 `GET /match`

Authentication: Anonymous
Order by: ✓ **Limit:** ✓ **Skip:** ✓

It return a list of all the matches. You could use `player` get parameter to filter all the match with that player. There is also a `ended` get parameter that can be set to true or false. `order_by` default value is *-lastMove*.
You will get an array of these objects:

```
{
    "_id": string,
    "player0": string,
    "player1": string,
    "winner": {
        "player": number?,
        "positions": number[][]
    },
    "turn": number,
    "board": number[][],
    "moves": number[],
    "matchStart": Date,
    "lastMove": Date,
},
```

Listing 19: Match model

3.5.2 `POST /match/`

Authentication: Jwt

Here you can create a match with one of your friend. You should have received an invite from `opponent` otherwise you will get a **403** status code. In response you will receive the id of the newly created match.
Body:

```
{
    "user": string, //Your email
    "opponent": string //Your friend email
}
```

Listing 20: Input information

Response:

```
{
    "matchID": string
}
```

Listing 21: Response

3.5.3 `GET /match/:id`

Authentication: Jwt, Anonymous

Here you can get a single match.
Response Model: See Match model in `GET /match`

3.5.4 `POST /match/:id/moves`

Authentication: Jwt

Here you can make a move. If you call this endpoint and you are not a player of this match you will get a **403** status code.
Body:

```
{
    "user": string , //Your email
    "column": number
}
```

Listing 22: Input information

3.5.5 `POST /match/random`

Authentication: Jwt

With this endpoint you can search for a random opponent. If you call this endpoint again before the end of matchmaking you will get a **400** status code. In case of match found you will be notified through socket.io (See 1.2.4)
Body:

```
{
    "user": string //Your email
}
```

Listing 23: Input information

3.5.6 `DELETE /match/random`

Authentication: Jwt

Here you can stop match search. You should set `user` get parameter with your email.

4 Authentication

4.1 General

In our backend we have implemented two types of authentication: **Basic HTTP** and **Jwt Token**.

4.2 Typical workflow

A user should register himself with `POST /user` endpoint;

After this he can login with `GET /login` endpoint (using Basic authentication) and he will receive a Jwt Token;

Then up to the expiration of the token he can make calls to endpoints using the token.

4.3 Jwt Token

We use this type of authentication in most cases where we need to verify the identity of someone making a request (See exceptions in Basic HTTP).

The tokens generated expires in one day.

```
{
    id: string,
    email: string,
    username: string,
    roles: string[]
}
```

Listing 24: Data inserted in token

4.4 Basic HTTP

This type of authentication is based on transmission of email and password. It is not a secure protocol by itself and should be secured with https. If a communication of this type were to be intercepted in clear text, the attacker would have access to the email and password and could impersonate the user steal the user's account.

The password is salted and hashed in **sha512**, in the database we save the hash and the salt to be safe against data leak.

We use basic authentication in two case:

Login See 3.2.2 `GET /login`

In order to get a jwt token we need to accredit ourselves in a different ways (With email and password).

Password Change See 3.3.5 `PUT /user/:email`

We do not allow password changes with jwt tokens to limit the risk in case of stolen jwt. In case of stolen jwt we would then only have access to resources until token expiration.

Complete registration of moderator See 3.3.5 `PUT /user/:email`

When a user with a temporary password try to login receive a "**422** *Please change your temporary password*". In that case we don't release a token to force the user to complete his registration, so this action must be done with basic authentication.

4.5 Extra: Anonymous

We also have some endpoint that doesn't require authentication.

5 Angular

5.1 Services

All the service are located in `just4fun-fe/src/app/services` folder.

5.1.1 User

Filename: `user.service.ts`

This service manage the user information and hold login status.
Is used by other service to obtain token, and by many components to get information about the user.

5.1.2 Chat

Filename: `chat.service.ts`

The **ChatService** construct **Chat** class. **Chat** class can be used to read and send messagges.
It is used by **Messages** component.

5.1.3 Match

Filename: `match.service.ts`

Expose useful methods to view and play matches.
It is used mostly by **Match** component.

5.1.4 Matchmaking

Filename: `matchmaking.service.ts`

Permit to start and stop match search.
It is used by **Logged Home** component.

5.1.5 Socket.io

Filename: `socketio.service.ts`

This service hold a singleton instance of websocket connection.
It also manage the handshake with the server and insert the user in the correct room.

5.2 Components

All the components are located in `just4fun-fe/src/app` folder.

5.2.1 Home

Folder: `home`

This is the landing page, it contains the logo, the leaderboard and a list of on-going matches.

5.2.2 Header

Folder: `header`

This component manage the top navigation bar and the menu.

5.2.3 Notifications

Folder: `notifications`

This component show a list of notifications, match invites and friend requests received.

5.2.4 Logged Home

Folder: `logged-home`

This component is used in home and showed only if we are logged-in. It show the matchmaking button and a welcome message.

5.2.5 Messages

Folder: `messages`

This component show one chat instance. It show all the messages of that chat and the input box where we can write new messages.

5.2.6 Chat

Folder: `chats`

This component use Messages component and show a list of chat. It permits to chat with our friends.

5.2.7 Match

Folder: `match`

This component permit to play or spectate a match.

5.2.8 Profile

Folder: `profile`

This component show information about ourself.

5.2.9 User View

Folder: `user-view`

This component show another player, here we can send friend request and match invite.

5.2.10 User Change Avatar

Folder: `user-change-avatar`

This component is used in User Register and User Settings components and permit to load and cut a profile picture.

5.2.11 User Change Password

Folder: `user-change-password`

This component is used alone and in User Settings component and permit password change.

5.2.12 User Login

Folder: `user-login`

This component show the login prompt.

5.2.13 User Register

Folder: `user-register`

This component permit registration.

5.2.14 User Logout

Folder: `user-logout`

This component log-out the user and show a goodbye message.

5.2.15 User Settings

Folder: `user-settings`

This component permit to change our information.

5.3 Routes

All the routes definition can be found in `just4fun-fe/src/app/app-routing.module.ts`

/ ↦ `HomeComponent`

/login ↦ `UserLoginComponent`

/logout ↦ `UserLogoutComponent`

/register ↦ `UserRegisterComponent`

/changePassword ↦ `UserChangePasswordComponent`

/match/:id ↦ `MatchComponent`

/profile ↦ `ProfileComponent`

/user/:mail ↦ `UserViewComponent`

/messages ↦ `ChatsComponent`

/notifications ↦ `NotificationsComponent`

/settings ↦ `UserSettingsComponent`

6 Workflows

We start from the home page:

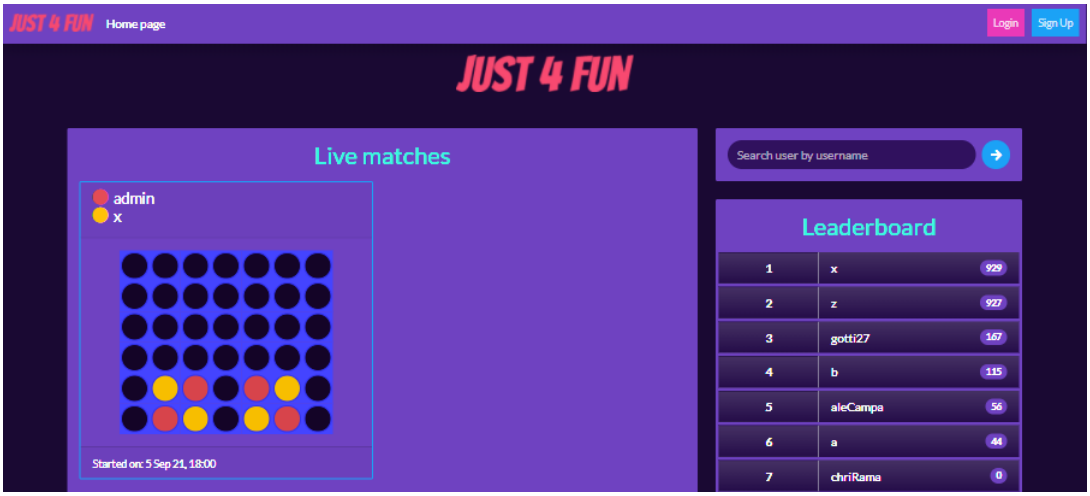

Figure 1: Home page

We click *Sign Up* and get registered:

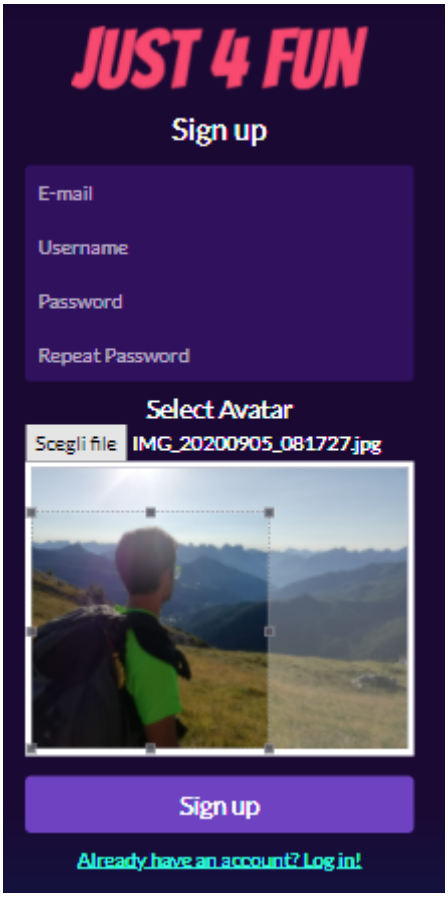

Figure 2: Sign up

We are redirected to login page:

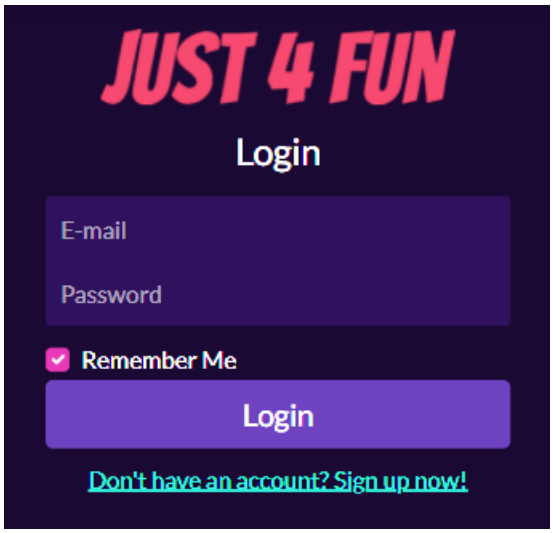

Figure 3: Login

And after login back to home page, here we can search for an opponent:

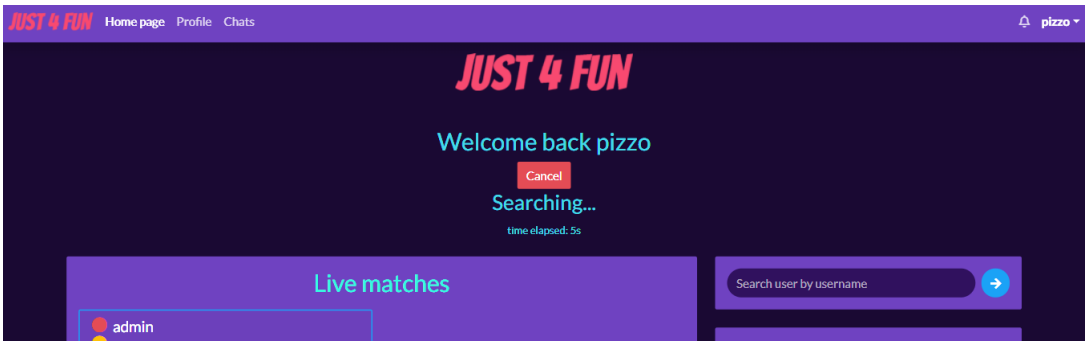

Figure 4: Matchmaking

Finally we can play a game:

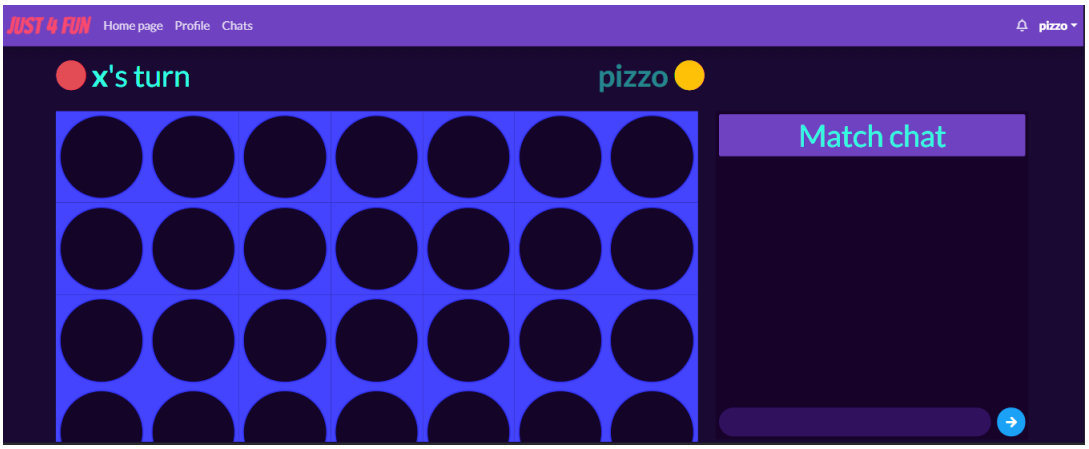

Figure 5: Match