



universidade
de aveiro

University of Aveiro
Master in Cybersecurity
Identification, Authentication and Authorization

Project:
Management of players reputation
with OAuth2.0
and
Authentication of players' with CMD
(Chave Móvel Digital)

Author:

- Daniel Baptista Andrade: 93313
- Diogo Miguel Rocha Amaral: 93228

Index

Introduction	2
Instructions	3
1. Requirements	3
2. How to run	4
1º Part	4
1. Technologies/Concepts used	4
2. OAuth2.0	4
2.1 Client	6
2.2 Authorization/Oauth server	8
2.3 Resource server	9
3. Requests and endpoints	10
3.1 Requests structure	10
3.1.1 Authorization grant	10
3.1.2 Access token	11
3.2 Endpoints	11
4. Gameplay	12
4.1 Database	13
4.2 Match between players	14
4.3 Websockets	15
4.4 Web interface	16
4.4.1 Username page	16
4.4.2 Games page	17
4.4.3 Authentication page	17
4.4.4 Authorization page	18
4.4.5 Waiting page	18
4.4.6 Result page	19
2º Part	19
1. Chave móvel digital (CMD)	19
1.1 Activation of CMD	20
1.2 Implementation	20
1.2.1 Token fetch flow	20
1.2.2 Attributes fetch flow	21
1.3 Authentication	22
1.4 Creation of a personal account	23
1.5 Changing password	23

1.6 Endpoints and requests	24
References	26
Feedback about github CMD	26
Contribution of each author	26

Introduction

1. Many service providers can have interest in providing services to end-users without having to know exactly who they are, instead, they just need to know a few trustworthy attributes that are fundamental for providing the service, without knowing anything else besides that.

So, with that in mind, we developed a system that manages and uses the reputation of players in an online table game and this is the first part of the project.

For that, we included 4 entities:

- The User Agent (UA): It is used by the players, it allows players to participate in several types of games.
- The Reputation Manager (RM): Is the keeper of the users' reputation relatively to online games (past actions)
- The Tables Matchmaker (TM): Is the entity responsible for organizing online games with candidate players. The TM does not know the identity of the players, only their reputation. It decides which players it should join for a game based only on that.
- The Match Manager (MM): Is responsible for controlling the execution of a game match, avoiding and detecting cheating and determining the winner and the loser. The outcome of the game is generated randomly.

It was also developed an OAuth entity that manages all the information related to the OAuth2.0, like the authentication of the user, the creation of the authorization grant and the creation of the access token.

2. For the second part of the project, we adapted the RM entity to use CMD (Chave móvel Digital) as an external IdP, including the following features:

- Creation of a personal reputation profile, one for each person and therefore the IdP (CMD) must be used to query attributes. The profile must also have a username + password authentication as an alternative.
- Authenticate in the RM using the external IdP or the username + password.
- Change the profile password upon a CMD-based authentication.

Instructions

1. Requirements

To correctly execute the project, it is necessary to have Node.js installed as well as Python3. For the Node.js in specific, a Framework NestJS was used to make all the process easier. From the python perspective, the framework used was FastAPI.

2. How to run

To run the entire project, it is necessary to start all the three modules: client, oauth server and resource server. For that, do this commands on the respective folders:

- **Client:** python3 -m uvicorn main:app
- **OAuth server:** npm run start
- **Resource server:** npm run start

Now the project is accessible on **<http://localhost:8000/>**.

1º Part

1. Technologies/Concepts used

This project was implemented by using FastAPI and NestJS. The tokens generated were developed with the help of JSON web tokens.

FastAPI: Is a modern, high-performance, web framework for building APIs with Python 3.6+ based on standard Python type hints.

NestJS: It is a progressive Node.js framework for building efficient, reliable and scalable server-side applications.

JSON Web Tokens: Is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Websocket: WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol enables interaction between a web browser (or other client application) and a web server.

2. OAuth2.0

OAuth 2.0 is the industry-standard protocol for authorization. The goal of the OAuth 2.0 is to allow an application to access user resources maintained by a service/server.

There are four entities involved in this process. The resource owner, that is the entity capable of granting access to a protected resource, in our case it is a player (UA). The resource server is the server hosting protected resources, capable of accepting and responding to requests using access tokens, in our scenario it is the RM entity. The client (TM) is an application making requests for protected resources on the behalf of the resource owner and its authorization. Lastly there is the authorization server or OAuth server that consists of issuing access tokens to clients after successfully authenticating the resource owner and obtaining his authorization. Figure 1 describes all the interactions between those entities.

It begins with an UA requesting to play a game. It generates a request to the client, in this case the TM entity. To start a match, the TM needs to know the reputation (skill and behavior) of the player so it needs an access token to obtain this information (protected resource). So the exchange with the Authorization server starts with an authorization request from the TM. For the Authorization server to grant an authorization grant, it is required the authentication and the authorization of the owner. After acquiring the authorization grant, the TM sends it in conjunction with the resource access request to the Authorization server so it has everything that is necessary to generate an access token and give it to the client. In the end, after the TM gets the access token, it can finally request the protected resource stored on the resource server (RM). Now the TM has everything that it needs to initiate a game between the players.

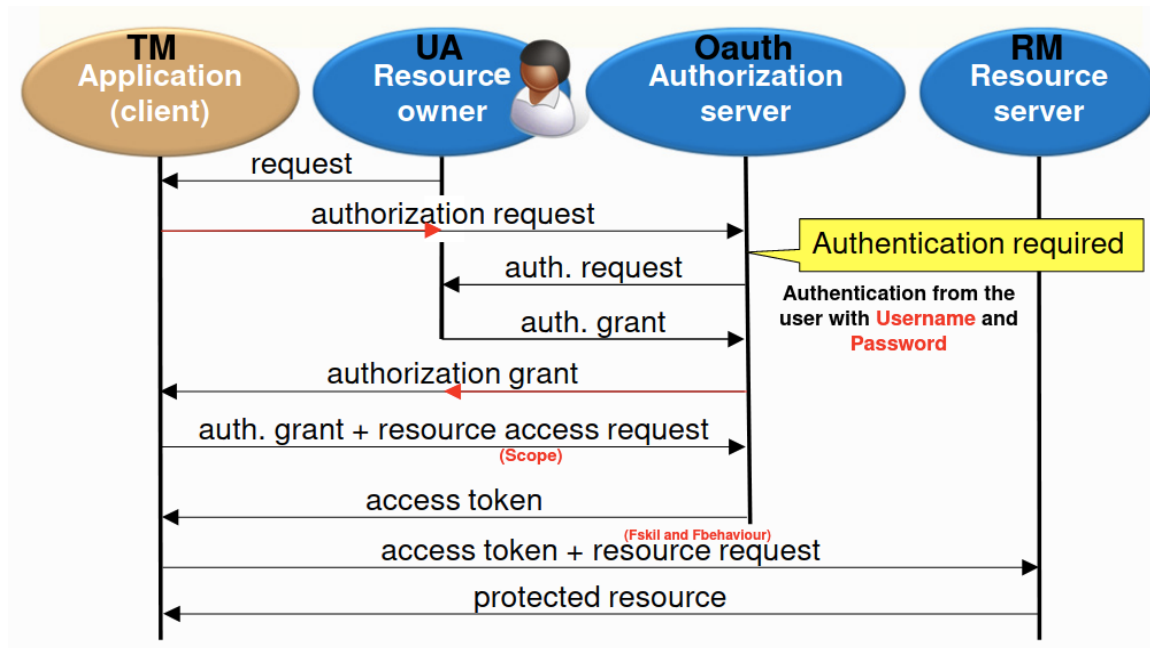


Figure 1: OAuth 2.0 flow

Some authentication for players must be implemented in order to authorize a client (TM) to access his personal reputation. In this first part, we implemented a basic username/password based protocol. The authentication of players will be changed in the second part of this project, adding the possibility to authenticate themselves using the CMD.

2.1 Client

On the Oauth overall flow, the client has a very important role, we would even say that it is the main module of the whole process. The client is initialized with a "client_id" and "client_secret" so it can be authenticated on the Oauth server.

The whole process is implemented with endpoints (using fastAPI) and starts with the UA accessing the site and choosing an username and some preferences. The TM obtains the parameters from the user, the preference of skill and behavior, the nBins and the username. Lastly creates a parameter for a token (for now, it stills None) and redirects the player to the game page, where he can choose which game to play.

After the user clicks on the button of the specific game that he wants to play, the TM obtains the scope (cards, chess or checkers) and now it has all the information needed to start the interaction with the Oauth to obtain the player reputation of that specific game. So it sends an "authorization request" to the Oauth server.

Following all the interaction between the UA (player) and the Oauth server (detailed further head), the client obtains the "authorization grant".

```
uri, state = clientcon.create_authorization_url(authorization_endpoint) #'http://localhost:3001/auth'
```

Figure 2.1: Authorization grant fetch

```
@Post('/auth')
@Redirect('/login', 303)
async authenticateUser(@Query() query: any, @Request() req: OtherReq){
    return this.oauthservice.authenticateUser(query, req);
}
```

Figure 2.2: Authorization grant fetch

```
authCodeGen(query: any, user: any) {
    var url_string = query.redirect_uri;
    const state = query.state;
    if (user) {
        var auth_code = codes.generateAuthCode(
            user,
            query.client_id,
            query.redirect_uri,
            query.scope,
        );
        url_string = url_string + '?code=' + auth_code + '&state=' + state;
        return { url: url_string, statusCode: 303 };
    }
}
```

Figure 2.3: Authorization grant fetch

With the “authorization grant” in the possession of the client, now the TM is capable of requesting an “access token” again to the server, so the client fetches the “access token” by giving his “authorization grant”.

```
agent["token"] = await clientcon.fetch_token("http://localhost:3001/token"
, authorization_response=authorization_response
, client_id="123abc"
, client_secret="projiaa2" )
```

Figure 3.1: Access token fetch

```
@Post('/token')
token(@Body() body: TokenDto){
    var token = this.oauthservice.tokenGen(body)
    tokens.saveToken(token)
    return token;
}
```

Figure 3.2: Access token fetch


```

tokenGen(body: {grant_type: string, code: string, redirect_uri: string, client_id: string, client_secret: string}){
  if(!codes.checkClient(body.client_id,body.client_secret)){
    return {
      "error":"invalid_request (client secret wrong)"
    };
  }
  return codes.generateAccessToken(body.code, body.client_id);
}

```

Figure 3.3: Access token fetch

Lastly, the client has everything that he needs to obtain the reputation of the players, so TM requests the RM the protected resource (FSkill and FBehaviour).

```

stats = requests.post('http://localhost:3002/'
, headers = headers
, json ={'nbins': agent["nbins"], 'scope': agent["game"]})

```

Figure 4: Protected resource fetch

2.2 Authorization/Oauth server

One important module for the Oauth2.0 is the server, the one who is responsible for generating the authorization grant in response to the authentication and authorization of the owner, and the access token if the client presents a valid authorization grant.

The interaction with the server begins with a request from a client to access a specific protected resource. First, the server verifies the client with a function called “checkClient” according to the client_id and client_secret .

Then, the server asks if the owner consents with that by requesting to authenticate himself and authorize the access (webpage where the player/owner needs to put his credentials and then click on a “consent” button). After this, an authorization grant will be generated. This authorization grant has information about the user in it and the scope (it means, the resource access request, in this case can be “chess”, “cards” and “checkers”). This token is sent in reply to the client.

```

static generateAuthCode(user: any, client_id:string, redirect_uri:string, scope:string){
  var authcode = AUTHORIZATION_CODE
  authcode.client = client_id
  authcode.redirectUri= redirect_uri
  authcode.scope=scope
  authcode.user = user
  authcode.authorizationCode = crypto.randomBytes(16).toString('hex')
  codes.authorization.push(authcode)

  return authcode.authorizationCode
}

```

Figure 5: Generation of the authorization grant

Another purpose of the server is to generate an access token based on a valid authorization grant. So, first the server validates the authorization grant and if it is valid, it generates the access token as follows:

```
return {"access_token": jwt.sign({user, client, scope}, "projiaa2", {expiresIn: "2h"}), "token_type": "Bearer"};
```

Figure 6: Generation of the access token

The “projiaa2” parameters correspond to the signature so, later on, the server can confirm if it is a valid token or not.

The server, in our case, also has the purpose of setting a cookie on the browser in order to associate an account with the UA. Therefore, for each browser it is only possible to enter an account at each time.

2.3 Resource server

The resource server (RM) is just responsible for verifying an access token and if it is valid, it is allowed to give the resource requested to the client. The verification is done using the jwt (json web token) “verify” method.

```
@Post('/')
information(@Request() req: OtherReq, @Body() body: {nbins: string}){
  if(!this.authService.VerifyJwt(req.headers.authorization.split(" ")[1])){ return {"error": "invalid token"} }
  return this.authService.RetrieveData(req.headers.authorization.split(" ")[1], Number(body.nbins));
}
```

Figure 7.1: Verification of the access token

```
async VerifyJwt(token: string){
  try {
    jwt.verify(token, "projiaa2")
    const result = await fetch('http://localhost:3001/findToken', {
      method: 'GET',
      body: token,
    });
    console.log(result)
    return true;
  }
  catch(err) {
    return false;
  }
}
```

Figure 7.2: Verification of the access token

Since the access token is valid, the resource server extracts information about the user and the scope from the token. Now, it has all the information needed to provide the

protected resource (skill and behavior) expect, so it sends a reply containing this information.

```
RetrieveData(token: string, nbins: number){
  const decoded = jwt.verify(token, "projiaa2");
  const jsonobj = JSON.stringify(decoded)
  var array = JSON.parse(jsonobj)
  const username = array.user.userName
  const scope = array.scope
  var fskill = this.Fskill(nbins, username,scope)
  var fbehav = this.Fbehaviour(nbins, username,scope)
  return "Fskill: "+fskill+", Fbehaviour: "+fbehav
}
```

Figure 8: Extract information from token and send reputation

This module is also responsible for generating the result of a game and updating the value of the reputation after the game finishes. Considering the following results:

- Result == 0 -> Cheat/Quit
- Result == 1 -> Win;
- Result == -1 -> Lose

```
UpdateData(token:string, result:number){
```

Figure 9: Function to update reputation

3. Requests and endpoints

3.1 Requests structure

3.1.1 Authorization grant

Authorization request:

- **response_type:** code - indicates that the server expects to receive a authorization code)
- **client_id:** ID of the client
- **redirect_uri:** indicates the URI to return the user to after authorization is complete
- **scope:** One or more scope values indicating which parts of the user's account you wish to access
- **state:** A random string generated by your application, which will be verified later

Example:

`http://localhost:3001/auth?response_type=code&client_id=123abc&redirect_uri=http://localhost:8000/tm&scope=cards&state=tPhPaaYWlxEPBz8q9SNsiPOEga6vqE`

Authorization reply:

- **code:** The server returns the authorization code
- **state:** The server returns the same state value that were sent

Example:

`https://example-app.com/cb?code=AUTH_CODE_HERE&state=tPhPaaYWlxEPBz8q9SNsiPOEga6vqE`

3.1.2 Access token

Access token request:

- **grant_type:** `authorization_code` - The grant type for this flow is `authorization_code`
- **code:** `AUTH_CODE_HERE` - The code that was received on the previous step (authorization grant)
- **redirect_uri:** `REDIRECT_URI` - Must be identical to the redirect URI provided in the original link
- **client_id:** `CLIENT_ID` - ID of the client
- **client_secret:** `CLIENT_SECRET` - the secret is included

Access token reply:

- **access_token:** token
- **expires_in:** Time before the token keeps invalid

3.2 Endpoints

To assure that only a certified user can access protected endpoints, two measures were taken. First, on the client side (TM), in every needed endpoint, there is an argument “permissions” that invokes `Depends()`. This function has the objective of checking if there is an Access Token associated with the UA that made those requests. Due to our TM implementation, only the endpoints “/gameresult” and “/g/{client_id}” are secure by this method, since they are the ones responsible for the match lobby and result. If the verification fails, the player is instantly redirected to the initial page of the website, forcing him/her to pass through all the correct steps to play again.

Note: Since this figure size does not fit very well on the report we will also place it in a folder separately for easier visualization and analysis.

4. Gameplay

The flow of a player playing a game is described as follows. A player asks the TM to play a given game, to find another player in the same rank as this player, TM needs to collect their reputation indicators (skill and behavior), with the OAuth2.0 explained in the last chapter. After an opponent is found, the player is placed in a table managed by an MM and the game begins. The match ends successfully or not, the MM reports the match outcome to the TM and TM updates the players reputation accordingly, also with OAuth2.0. All the flow is shown on figure 11.

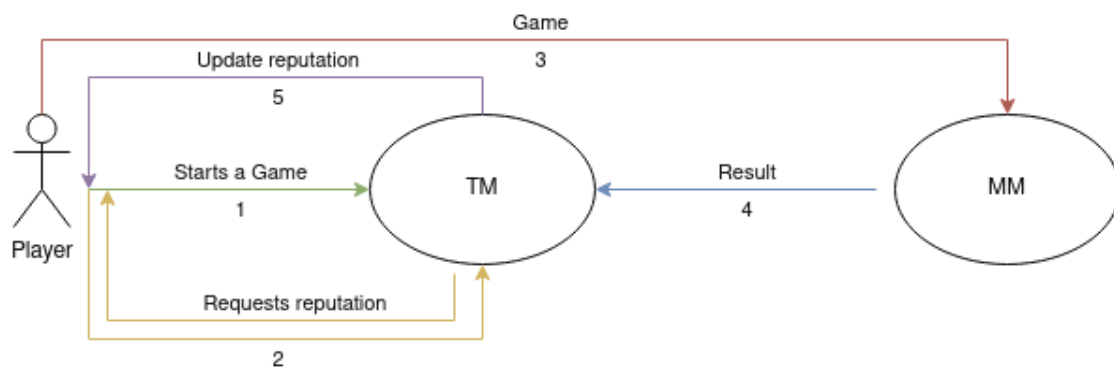


Figure 11: Gameplay flow

For simplicity, the results of a game are generated randomly, we only start the game and immediately, a random outcome is generated. The possible results are: win, lose and quit/cheat.

4.1 Database

For the platform to work properly, it is fundamental to have a database with some accounts already created. With that in mind, two databases were created, one with the name of "database.ts" containing the username of a player and his personal reputation for each game (being the first column the skill value and the second the behavior value) (figure 12) and another called "users.ts" holding the username and passwords of all the users (figure 13).

```
{
  "username": "Antoinette",
  "chess": ["45", "30"],
  "cards": ["22", "33"],
  "checkers": ["43", "23"],
},
```

Figure 12: database.ts example

```
{
  "id": 1,
  "username": "Antoinette",
  "password": "62a5e7a54490e361f7dc8824",
  "CC": ""
},
```

Figure 13: user.ts example

Note: The data stored on the files is not protected or ciphered, because we considered it was not the point of the project.

4.2 Match between players

With the aim of maintaining the reputation indicator exposed just to their owners and never directly to others, a coarse-grained form was used to expose these indicators to TMs (e.g Level L out of N , with $L = 1$ being the highest one). This provides some anonymity to players, in which they can choose the N from a maximum allowed determined by RM. N defines the number of bins that are used to rank players.

Players can also choose the reputation of the opponent, it means that a player can indicate to play a game against a player with higher, equal or lower skills than him and also the behavior reputation, prefer to play against players worse or better than him.

Since players have the possibility to choose the value N , a way of players with different N being capable to play with each other should be taken into consideration. For example, a player with skill $1/10$ and prefers to play with more skilled players, he can play against a player with $1/3$ but not against a player with $2/3$.

So, basically, if both players have opposite preferences, they match. It means, if player₁ wants to play against a less skilled player and player₂ wants to play against a more skilled player, they match (if the skills are really higher/lower compared to each other), if they both prefer to play against highly/less skilled players, they don't match.

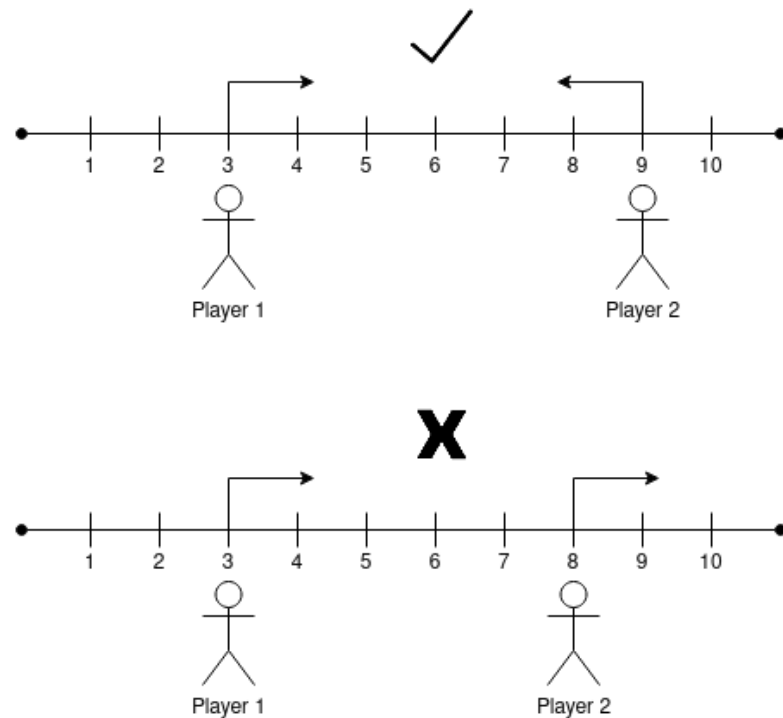


Figure 14: Match process. Opposite direction, they match. Same direction, don't match

Our implementation looks like this, considering that “-1” is “Lower”, “0” is “no preference” and “1” is “Higher”:

```
if p[5] != agent["game"]:
    break

if p[3] <= 0 and agent["skill"] >= 0:
    if Fraction(p[0],p[2]) >= Fraction(agent["Fskill"]):
        count += 1

if p[3] >= 0 and agent["skill"] <= 0:
    if Fraction(p[0],p[2]) <= Fraction(agent["Fskill"]):
        count += 1

if p[4] <= 0 and agent["behaviour"] >= 0:
    if Fraction(p[1],p[2]) >= Fraction(agent["Fbehaviour"]):
        count += 1

if p[4] >= 0 and agent["behaviour"] <= 0:
    if Fraction(p[1],p[2]) <= Fraction(agent["Fbehaviour"]):
        count += 1

if count == 2:
    return RedirectResponse("http://localhost:8000/g/"+item.path_params["client_id"], 303)
```

Figure 15: Match code

With that implementation, it turns the code a lot cleaner instead of doing all the cases possible (combination between all the possible options (low, equal and higher for both skill and behavior).

4.3 Websockets

Since the games are designed to play with many (real) players, it was necessary to do an implementation with websockets, so it would be possible to create a connection between both players.

It was implemented using fastapi and it can be found [here](#). It is based on a main class called “ConnectionManager” where there are methods to “connect”, “disconnect”, “send_personal_message” and “broadcast”.

The endpoint to create rooms for games is “/g{client_id}” and the client_id part is made like : “_FSKILL_FBEHAV_NBINS_SKILLPREFERENCE_BEHAVIOURPREFERENCE_GAME” and this connection is stored. The next player who tries to connect to the same game type is evaluated in order to see if it fits some of the already pending matches. If so, the UA enters the endpoint given to him. If not, a new endpoint is created, therefore a new room exists and there, the UA awaits for someone to match against.

In the case of similar preferences and game indicators, since there is only games of two players, the player awaiting is disconnected, the websocket part of the code (MM) generates a result to the match and both players are redirected to the endpoint “/gameresult/” with their respective scores.

For debug purposes , a flag is present in the code to make sure that, after a 3 second period, a player disconnects and a result is assigned, to make it easier to test in demonstrations. Setting this flag to false, the code works as intended, the UA awaits for an opponent to unroll all the process.

4.4 Web interface

For the project, a web interface (GUI) was implemented so it would be better, user friendly and simpler for a normal user (instead of a CLI). The pages were based on an already implemented web page, which can be found [here](#).

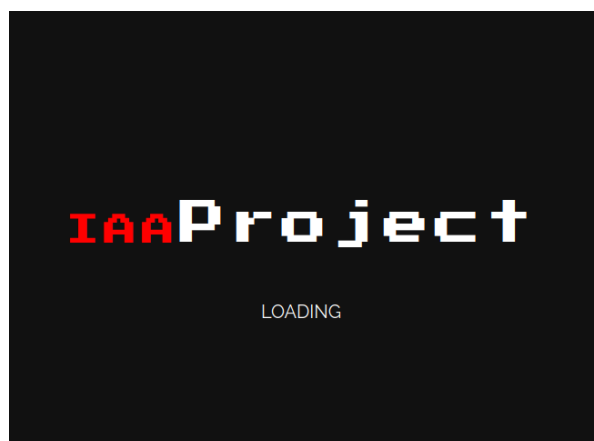


Figure 16: Loading

4.4.1 Username page

This page allows an user to put every username possible and continue with the same reputation values as the account, it means that the reputation is related/connected to the user account and not to his username at the moment. So, with that, there is the username chosen to appear on the games and the username from his account (for the authentication and authorization part)

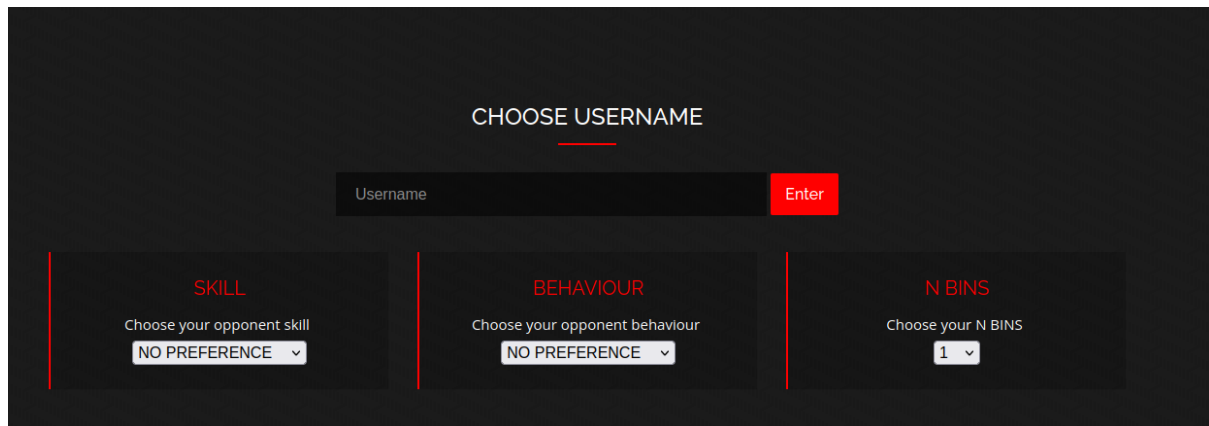


Figure 17: Username page

4.4.2 Games page

Here, the user can choose which game to play, there are three options: Cards, Chess and Checkers.

The users have a skill/behavior value for each game, so the program can find a fair opponent to play against.

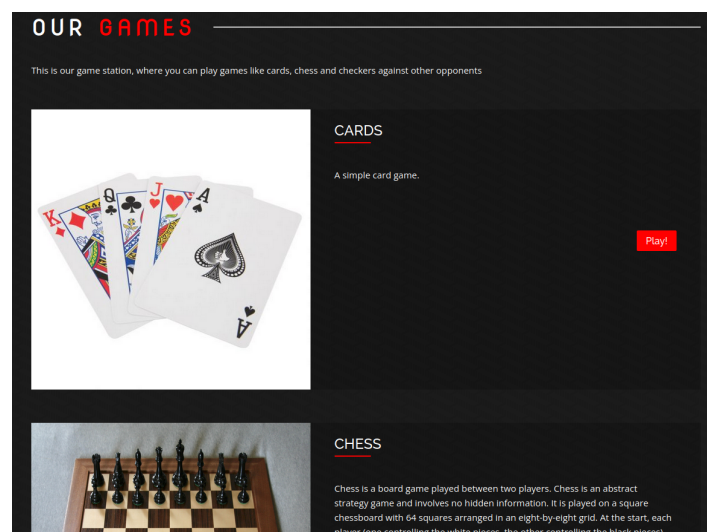


Figure 18: Games page

4.4.3 Authentication page

On the authentication page, the user needs to authenticate himself with the combination of username and password. If they are correct, the authentication is valid, If the username + password is incorrect, nothing happens on the system.

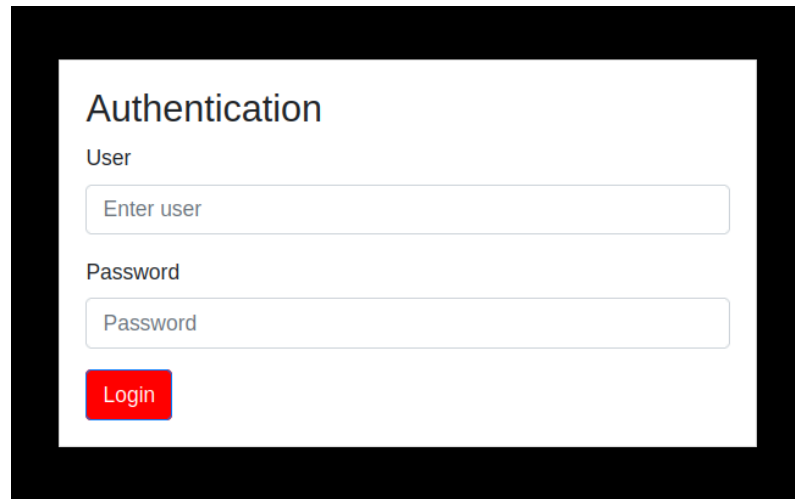
A screenshot of a web page titled "Authentication". It features two input fields: "User" with a placeholder "Enter user" and "Password" with a placeholder "Password". Below these fields is a red button labeled "Login". The entire form is enclosed in a white box with a black border.

Figure 19: Authentication page

4.4.4 Authorization page

On the OAuth flow, the player (the resource owner) needs to give consent for the TM to use his personal information to find an opponent with a similar reputation.

With that in mind, the player gives permission after authenticating himself (to prove that it is really him) and authorizes the access by clicking on the button.

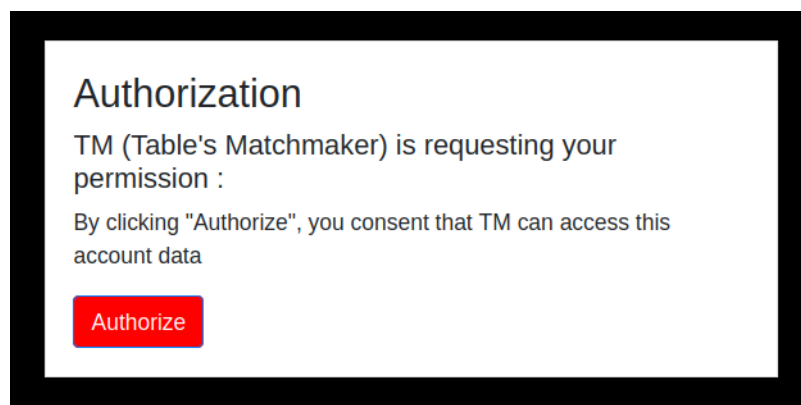
A screenshot of a web page titled "Authorization". It contains the text "TM (Table's Matchmaker) is requesting your permission :", followed by "By clicking 'Authorize', you consent that TM can access this account data". At the bottom is a red button labeled "Authorize". The entire form is enclosed in a white box with a black border.

Figure 20: Authorization page

4.4.5 Waiting page

After a player chooses a game, authenticates himself and gives permission for the client to use his reputation to find a fair opponent according to the player's personal preference, a waiting page is shown. On this page, while the player is waiting for an opponent/result, his personal reputation is shown (Fskill and Fbehavior for the game in question).

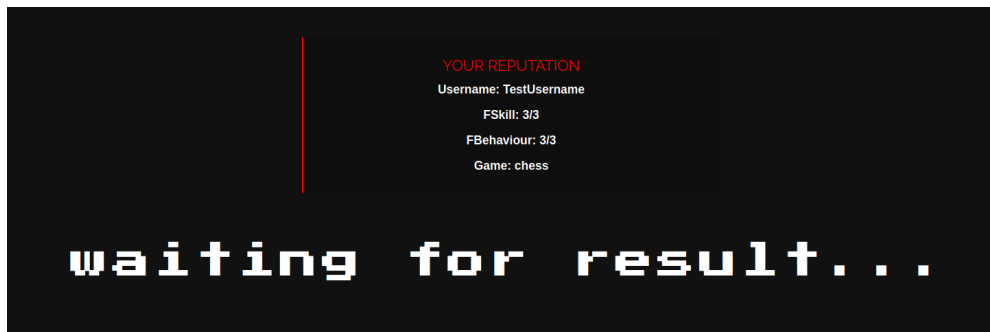


Figure 21: Waiting page

4.4.6 Result page

On this last page, it only shows the result of the game.

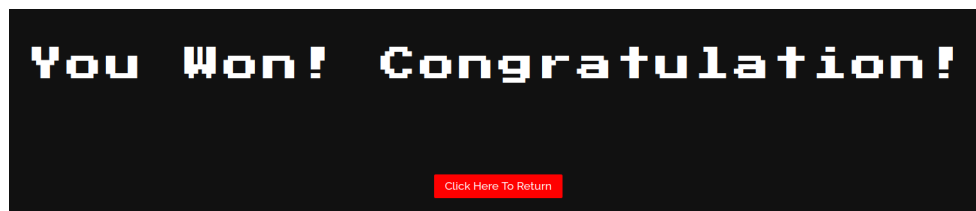


Figure 22: Result page

2º Part

1. Chave móvel digital (CMD)

Chave móvel digital or CMD is a Portuguese IdP and the main goal is to give a way of authentication and a certified digital signature for Portuguese people provided by the Portuguese government. We will be using this to identify players within the Reputation Manager (RM) but the implementation was done on the OAuth module.. The CMD IdP is accessed also using OAuth2.0.

We added three features:

- Creation of a personal account, using the CMD
- Authentication on the platform using the CMD
- Modification of the account password

1.1 Activation of CMD

The first thing we had to do for this second part of the project was the activation of the CMD, for that, we proceeded to the activation [site](#) and filled the information required.

It was required to have the Cartão de Cidadão (CC), to know the confidential PIN of the CC and finally to have in possession a Card reader.

1.2 Implementation

To develop this feature on our project, we followed the instructions presented on the [official documentation](#), produced by the Portuguese government. We act in accordance with the OAuth2 implementation, more specifically “Guia rápido de utilização do OAuth2” file.

1.2.1 Token fetch flow

The usage of the CMD starts by a request made by our platform to the Attribute Provider (FA) with the following parameters :

- response_type: token
- client_id: 9113170755799990166
- redirect_uri: http://127.0.0.1:3001/Authorized
- scope: http://interop.gov.pt/MDC/Cidadao/NIC

Completed request:

https://preprod.autenticacao.gov.pt/oauth/askauthorization?redirect_uri=http://127.0.0.1:3001/Authorized&client_id=9113170755799990166&scope=http://interop.gov.pt/MDC/Cidadao/NIC&response_type=token

After the request, the player is redirected to the CMD site where he is required to authenticate himself (phone number and CMD PIN). With the authentication granted, it is replied to with a message containing a token (access_token) and the expiration time (expires_in).



Figure 23: Token fetch flow

```
@Post("/login/cmd")
cmd(@Res() res){
    return res.redirect("https://preprod.autenticacao.gov.pt/oauth/askauthorization?redirect_uri="
}
```

Figure 24: Endpoint redirect

Note: Since the reply was a URI fragment and because of that it is impossible to take out the information presented on it, we made a javascript script to take that information and pass it to the next step.

1.2.2 Attributes fetch flow

Now, in the possession of the token, the next step to do is send another request again, to the Attribute Provider. On this request must be the following parameters:

- token: Obtained on the previous step
- attributesName: <http://interop.gov.pt/MDC/Cidadao/NIC>



Figure 25: Attributes fetch flow

The API of the Attribute Provider returns a JSON containing the token previously sent and an authenticationContextId (figure 26).

```
{
  "token": "a7ff26fb-392e-433b-94d6-c7182b5e7983",
  "authenticationContextId": "b3e98d70-dc69-4e5b-bafa-184da35c0a13"
}
```

Figure 26: JSON example

With that information, afterwards an GET request is made to the API with the recently information obtained, like:

<https://preprod.autenticacao.gov.pt/oauthresourceserver/api/AttributeManager?token=a7ff26fb-392e-134b-94d6-c7182b5e7983&authenticationContextId=b3e98d70-dc69-4e5b-bafa-1849a35c1233>

The API will finally return a JSON with the attribute requested presented in it, in our case, the CC of the player.

```
async cmdfetch(access_token: string) {
  var body = {
    token: access_token,
    attributesName: 'http://interop.gov.pt/MDC/Cidadao/NIC',
  };

  const res = await fetch(
    'https://preprod.autenticacao.gov.pt/oauthresourceserver/api/AttributeManager',
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(body),
    },
  ).then((response) => response.json());

  const values = await fetch(
    'https://preprod.autenticacao.gov.pt/oauthresourceserver/api/AttributeManager?token=' +
    res.token +
    '&authenticationContextId=' +
    res.authenticationContextId,
    { method: 'GET', headers: { 'Content-Type': 'application/json' } },
  ).then((response) => response.json());
}
```

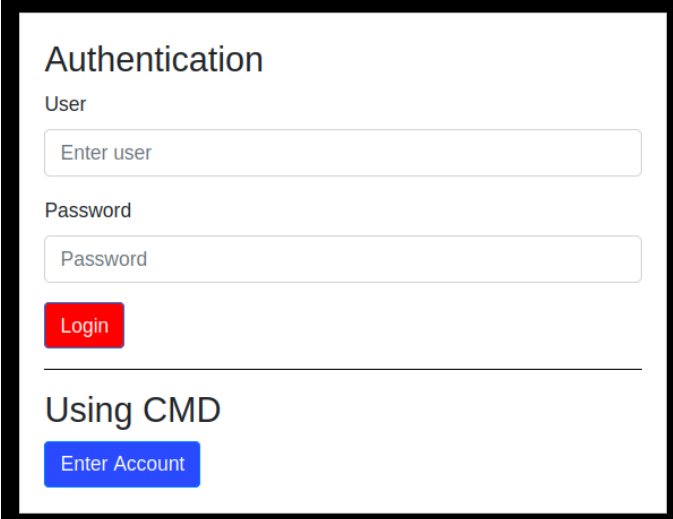
Figure 27: Code implementation

1.3 Authentication

With the CMD feature implemented, some webpages were altered to add those features, more specifically the Authentication page and the Authorization page.

The flow of the CMD will be the following:

When a player clicks on the “Enter account” button, the server verifies if an account with that CMD already exists, if not, an account will be created. If an account is already created it redirects automatically to the authorization page where there is the possibility of consenting to give his credentials for the TM or change the password of the account.

The image shows a web form titled "Authentication". It contains two input fields: "User" with a placeholder "Enter user" and "Password" with a placeholder "Password". Below these fields is a red button labeled "Login". A horizontal line separates this section from the section below, which is titled "Using CMD". Below this title is a blue button labeled "Enter Account".

Authentication

User

Enter user

Password

Password

Login

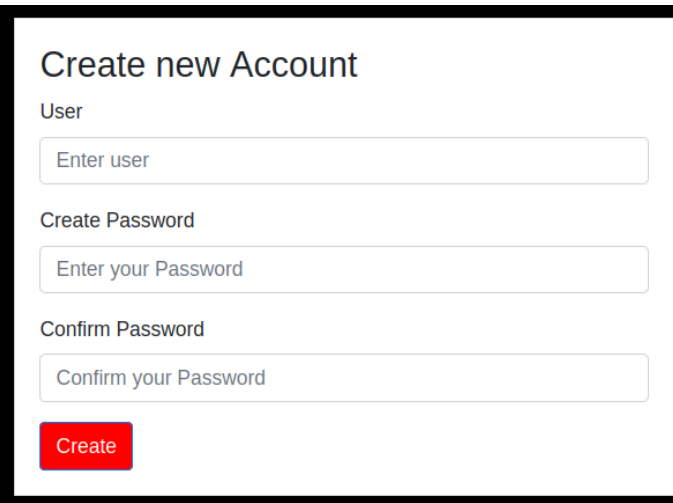
Using CMD

Enter Account

Figure 28: Authentication page with CMD

1.4 Creation of a personal account

Functionality to create an account, based on the CC (Cartão de Cidadão) number. But it is necessary to add an username and password as an alternative to access the account.

The image shows a web form titled "Create new Account". It contains three input fields: "User" with a placeholder "Enter user", "Create Password" with a placeholder "Enter your Password", and "Confirm Password" with a placeholder "Confirm your Password". Below these fields is a red button labeled "Create".

Create new Account

User

Enter user

Create Password

Enter your Password

Confirm Password

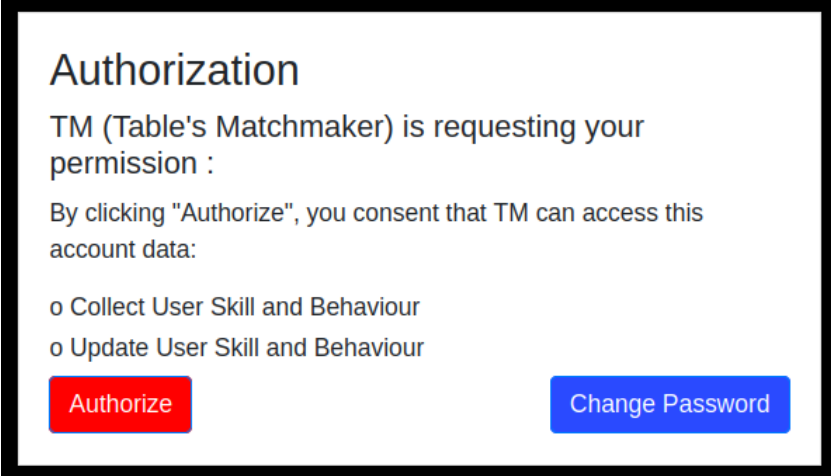
Confirm your Password

Create

Figure 29: Create new account page

1.5 Changing password

Feature to change the password of an account upon a CMD-based authentication. This is only available using CMD, an authentication based on username+password is not capable of changing that information.



The image shows a web page titled "Authorization". The text reads: "TM (Table's Matchmaker) is requesting your permission :". Below this, it says: "By clicking 'Authorize', you consent that TM can access this account data:". There are two bullet points: "o Collect User Skill and Behaviour" and "o Update User Skill and Behaviour". At the bottom, there are two buttons: a red "Authorize" button and a blue "Change Password" button.

Authorization

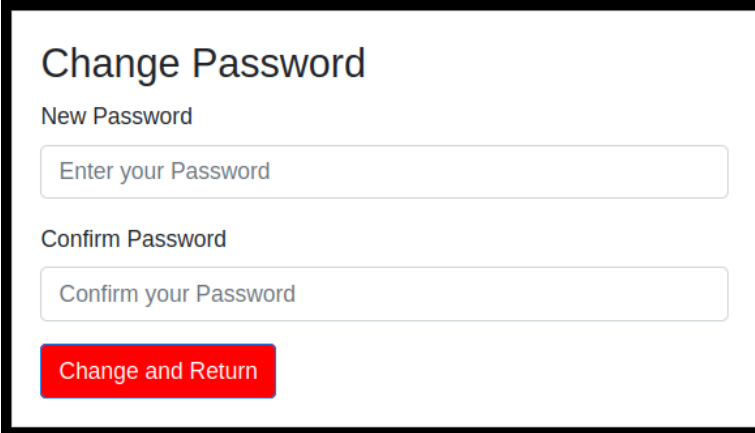
TM (Table's Matchmaker) is requesting your permission :

By clicking "Authorize", you consent that TM can access this account data:

- o Collect User Skill and Behaviour
- o Update User Skill and Behaviour

Authorize Change Password

Figure 30: New authorization page



The image shows a web page titled "Change Password". It has two input fields: "New Password" with the placeholder text "Enter your Password" and "Confirm Password" with the placeholder text "Confirm your Password". Below the input fields is a red button labeled "Change and Return".

Change Password

New Password

Enter your Password

Confirm Password

Confirm your Password

Change and Return

Figure 31: Change password page

1.6 Endpoints and requests

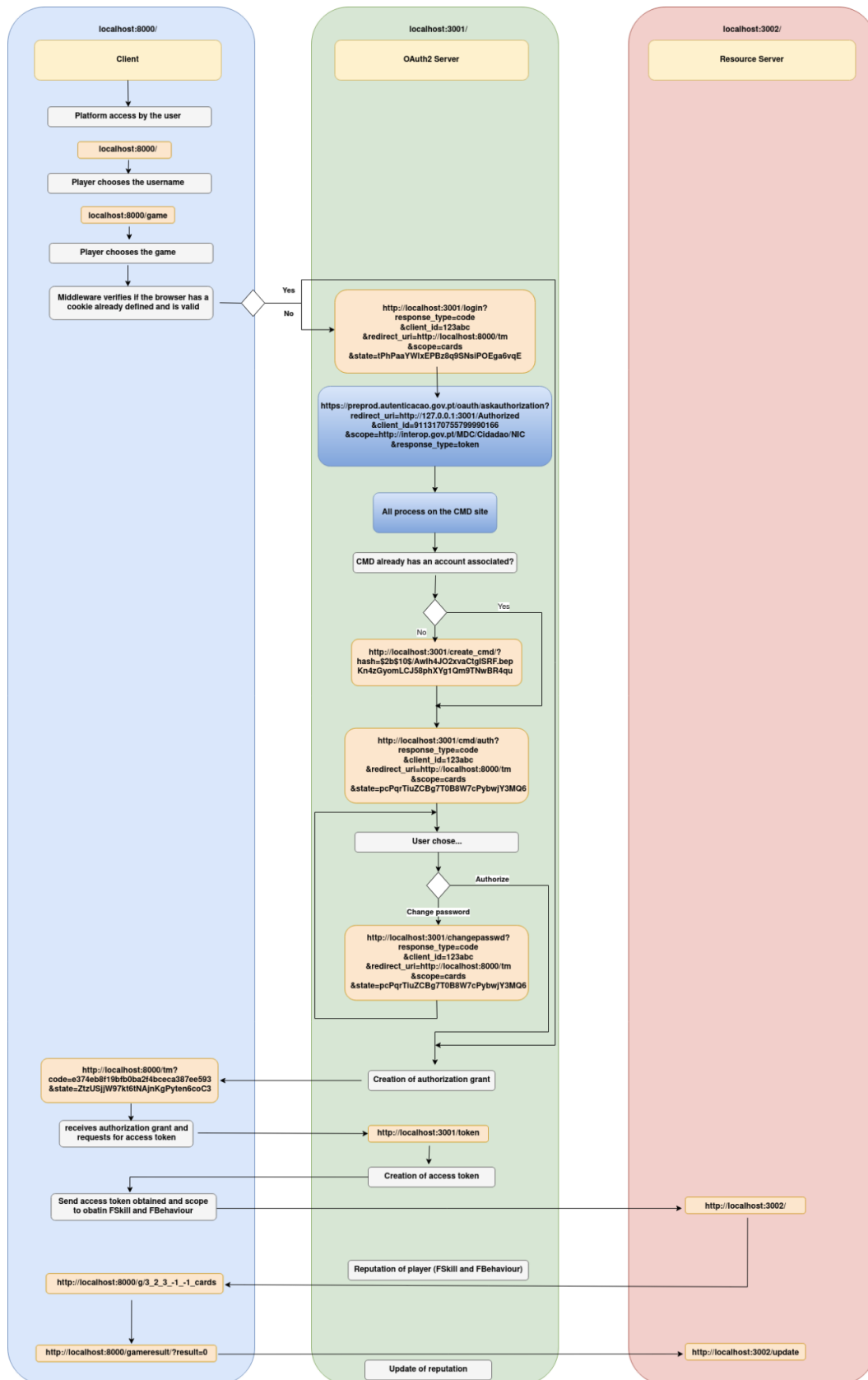


Figure 32: Flow and endpoints with CMD

Note: Since this figure size does not fit very well on the report we will also place it in a folder separately for easier visualization and analysis.

References

<https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/>

<https://aaronparecki.com/oauth-2-simplified/>

<https://fastapi.tiangolo.com/advanced/websockets/>

<https://pprwww.autenticacao.gov.pt/>

<https://github.com/amagovpt/doc-AUTENTICACAO>

<https://github.com/sanchi231/gamenact>

Feedback about github CMD

The main concern about this specific part is the following. Certain steps of the process seemed a bit clunky. The access tokens and other codes related to the process appear to be only needed by other server counterparts, however, there is a specific step of the CMD process that ignores this need. The access token is given using URL Fragments, an url parameter that is only available to the browser. To bypass this obstacle, a temporary webpage had to be made. This webpage only exists to support a javascript script that parses those same fragments into code and redirects them to the servers that have the function to process them and to continue the rest of the flow.

Contribution of each author

In the development of this project, both authors made similar contributions and showed the same effort so the final project would be the best version possible, so 50% to each member.