Final project

University of Aveiro
Masters in cybersecurity
Identificação, Autenticação e Autorização

Authors:

- Guilherme Amaral Ribeiro Pereira: 93134
- José Luís Costa: 92996

Index

1. Introduction

This report was developed under the course “Identificação, Autenticação e Autorização” with the objective of explaining the implementation and strategies used to achieve the objectives for both the first and second part of the course's final project.
As the project is divided into two parts, we will explain each part in detail in separate chapters.
Since the code for both the first and the second part are integrated together, we will begin by explaining the structure of the code, the requirements to run the project and lastly how to run it.
Before we begin we also made a small video demonstrating the project, the link to the video is available at: https://www.youtube.com/watch?v=8PMqwE6ks-A.
This project was developed by José Luís Costa (92996) and Guilherme Pereira (93134) with each member contributing 50% to the project.

1.1. Code structure

The structure of the project can be seen in figure 1. As we can see the entities are in a flask server that has a comun ip address and port, however the entities do not share any code or databases between themselves. So effectively it's as if they were in separated servers with different ips.

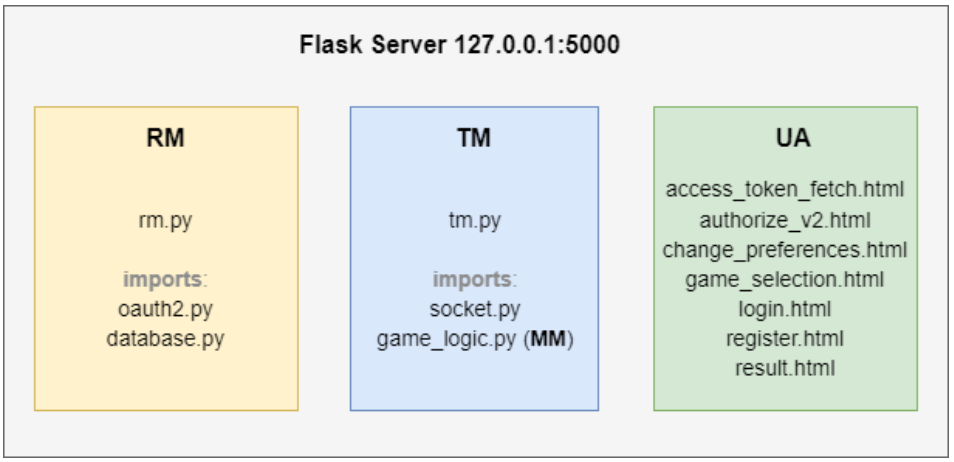


Figure 1: Code structure

Other files not present in Figure 1, include the bot script (bot.py), configurations of the flask server (app.py/app_start.py) and the database description file (database.sql).

1.2. Requirements to execute the project

This project was developed using python, more specifically python's version 3.8.8 .
In addition to having python installed, to run this project it is also necessary to have some libraries that can be installed using a requirements file that is provided alongside this project (requirments.txt). The requirements file contains all the libraries and the respective versions that were used when developing this project.

```
$ pip3 install -r requirements.txt
```

The libraries that are necessary include:

- **Flask:** Which is a web framework written in python.
  Documentation link: https://flask.palletsprojects.com/en/2.1.x/

- **AuthLib:** This is a python library that helps in the implementation of Oauth framework. This library has support for various frameworks including Flask.
  Documentation link: https://docs.authlib.org/en/latest/

- **Flask-SocketIO:** This library gives Flask applications access to low latency bi-directional communication between clients and servers.
  Documentation link: https://flask-socketio.readthedocs.io/en/latest/

- **Requests:** Requests provides a simple way to send HTTP/1.1 requests.
  Documentation link: https://requests.readthedocs.io/en/latest/

- **Cryptography:** Python package that provides interfaces to cryptographic algorithms such as symmetric ciphers, message digests and key derivation functions.
  Documentation link: https://cryptography.io/en/latest/

- **Tabulate:** Tabulate is a utility library used to print tabular data in python.
  Documentation link: https://pypi.org/project/tabulate/

It is also necessary to install sqlite3 as we use it to create and manage the databases that are used in the project and python3-flask.
Sqlite3 can be installed through apt with:

```
$ sudo apt install sqlite3
```

Python3-flask can be installed through apt with:

```
$ sudo apt-get install python3-flask
```

Lastly you might also need to disable https by changing the ambient variable that authlib uses:

```
$ export AUTHLIB_INSECURE_TRANSPORT=1
```

1.3. How to run the project

As mentioned in the previous section the first thing to do is to install the dependencies.
To launch the flask server one can use the below command on the root folder of the project.

```
$ flask run
```

This will launch the server on the localhost ip (127.0.0.1) port 5000.
Although the entities use the same server (ip address) they are completely independent. They were implemented using blueprints in flask and these entities do not share code or a database with each other, only information via the normal oauth procedure.

Note: There are already several users in the database, so one does not have to create other users to check out the project. For instance one can login with username “t1”, “t2”, “t3” or “t4” with the same password for all of them “pwd”.

2. First part of the project

The first part of the project consists of developing a system for managing and using the reputation of players in a series of online table games using the OAuth2.0 authorization framework. This system will consist of four major entities:

- The **User Agent (UA)** which is used by the players to participate in the online games.
- The **Reputation Manager (RM)** which maintains the reputation of the users relatively to the online games.
- The **Tables Matchmaker (TM)** that is responsible for organizing the online games. This entity does not know the identity of the players, it only knows their reputation indicators in a coarse-grained format.
- The **Match Manager (MM)** is responsible for managing the outcome of the matches and indicating the result to the Table Matchmaker.

Knowing this, in this chapter we will discuss the implementation that was done to achieve the objectives proposed for this part.
It is also worth mentioning that the code that was developed is also commented as to have better documentation of the whole project.

2.1. Workflow Diagram

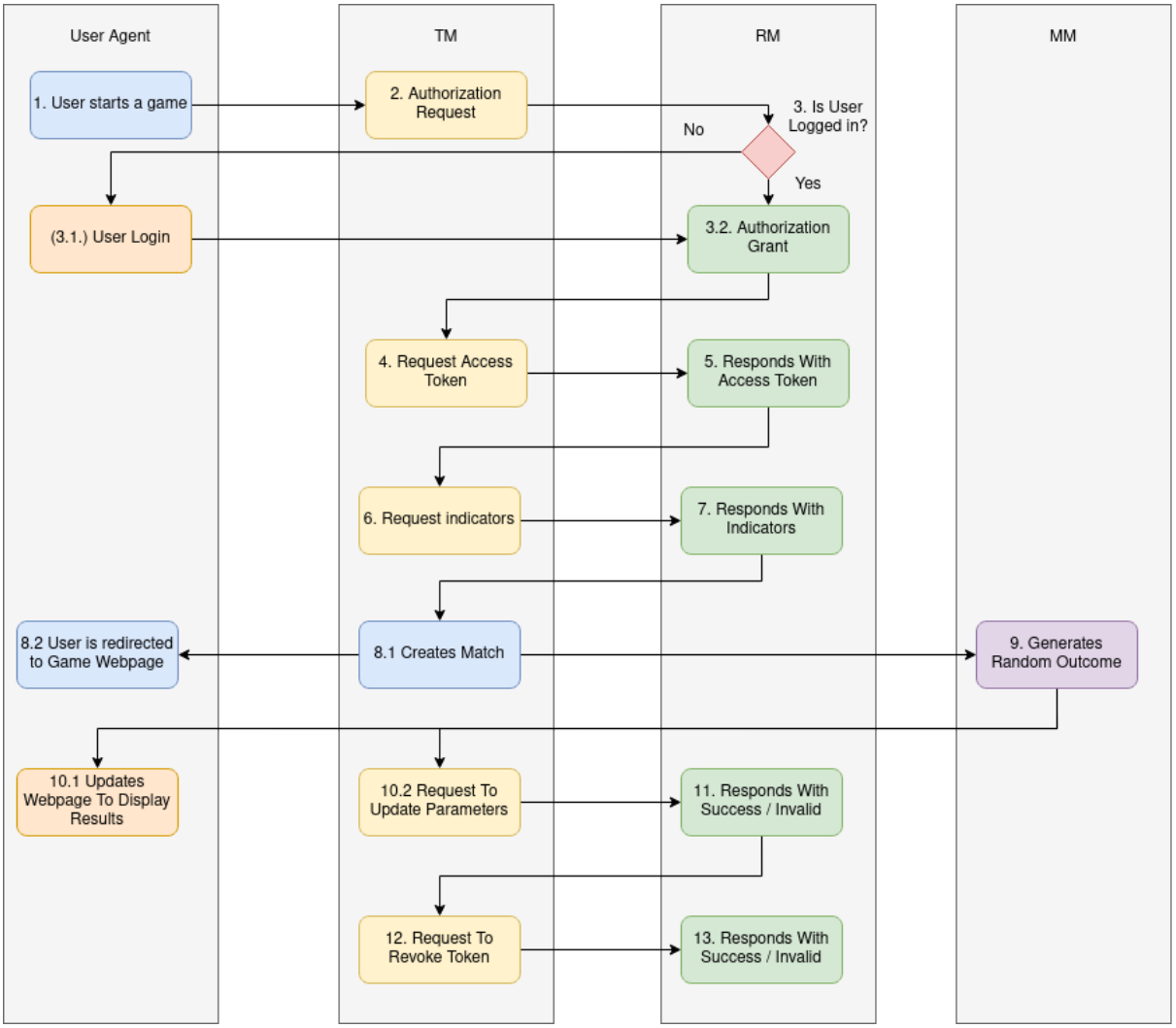


Figure 2: First part of the project workflow

The workflow diagram figure 2, represents the flow of the entire process of the application, since a user requests to start a game until the game is finished. To better understand the workflow diagram the following step-by-step brief explanation is presented with references to the corresponding implementation details:

1. The user requests the TM to start a game via the "/tm/start_game" endpoint.
2. The TM processes the request and redirects the user to perform an authorization request. (Start Game Endpoint)
3. The RM verifies if the user is logged in or not, if he is not logged in then it is requested to login with credentials (Login Endpoint). After logging in or already being logged in, an authorization grant is generated with the user consent (Authorization Endpoint).

4. The TM requests the access token using the authorization grant information (Callback Endpoint)
5. The RM responds with a valid access token (Token Endpoint).
6. The TM requests the coarse-grained indicators for the specific user (Game Endpoint).
7. The RM calculates and responds with these indicators (Users Reputation Endpoint).
8. The TM matches the user by taking into account all user information such as the coarse-grained skill/behavior indicator, number of bins, preferences and the user gets redirected to the webpage where the game "will be played" (Game Endpoint).
9. The MM generates a random outcome that can either be win, loss, draw, cheating or quitting for each user (2.4. MM)
10. The users webpage is updated with the outcome and the TM requests the RM to update the indicators according to the match outcome for each player (Game Endpoint)
11. The RM responds with either a successful or failed operation depending if the access token used to perform the request is valid and has the correct scope (Update Indicators Endpoint).
12. The TM requests the RM to revoke the token, as it's no longer going to be used (Game Endpoint).
13. The RM responds with either a successful or failed operation (Revoke Endpoint).

2.2. RM

2.2.1. Objective

The Reputation Manager objective is to keep record of the users reputation for each online game, incorporating both an (oauth2) authorization server and a (oauth2) resource server.

2.2.2. Authorization Server

The Authorization server is one of the main components of the RM that's part of the Oauth 2.0 standard.
The Authorization Server is not only responsible for authenticating the Resource Owner but also issuing and managing access tokens after getting proper authorization.
To create the Authorization Server, the Authlib library was used in conjunction with Flask to provide API server access.

Data structure

In order to implement the Authorization Server properly, a few data structures (tables) were created in a database in order to keep the User, Oauth2 Client, Authorization Code and Access Token information. This database is also shared with the Resource Server which will be explained in later sections.
The authorization server portion of the database is structured as follows:

TABLE: user

Types	Name	Description
INTEGER	Id	user database id
VARCHAR	username	user username
BLOB	password	user password
BLOB	cmd_id	hashed NIC (used and explained in the second part)

Figure 3: User table in the database

TABLE: oauth2_client

Types	Name	Description
INTEGER	Id	Client database id
INTEGER	user_id	User identifier (Foreign key to user table)
VARCHAR	client_id	OAuth2.0 client id
VARCHAR	client_secret	OAuth2.0 client secret
INTEGER	client_id_issued_at	Client id issue date
INTEGER	client_secret_expires_at	Not used
TEXT	client_metadata	JSON object that contains the client name, redirect uri, response type, scope and authentication method

Figure 4: Oauth2_client table in the database

TABLE: oauth2_token

Types	Name	Description
INTEGER	Id	Database id
INTEGER	user_id	User identifier (Foreign key to user table)
VARCHAR	client_id	OAuth2.0 client id
VARCHAR	token_type	Always "Bearer"
VARCHAR	access_token	OAuth2.0 access token value
TEXT	scope	Access token allowed scopes
BOOLEAN	revoked	Access token revocation status
INTEGER	issued_at	Access token request timestamp
INTEGER	expires_in	Access token validity

Figure 5: Oauth2_token table in the database

TABLE: oauth2_code

Types	Name	Description
INTEGER	Id	Database id
INTEGER	user_id	User identifier (Foreign key to user table)
VARCHAR	client_id	OAuth2.0 client id
TEXT	scope	Authorization requests scopes
INTEGER	auth_time	Authorization request timestamp
VARCHAR	code	OAuth2.0 authorization code

Figure 6: Oauth2_code table in the database

Authlib Configuration

To create the authorization server the Authlib library was used.
This library contains an authorization server element that can be instantiated by providing a function that queries and returns an existing user in the database and a function that saves authorization tokens in the oauth2_token table (oauth2.py).
To use the authorization server properly, we had to create multiple custom authlib wrapper classes that are used for the database interaction with the oauth2_client, oauth2_code and oauth2_token tables explained in the last chapter. (database.py)
Other important wrappers that had to be created were the AuthorizationCodeGrant, MyBearerTokenValidator and MyRevocationEndpoint which are also responsible for the custom integration with the database (oauth2.py). An example of this custom integration can be observed in figure 7, where the token is revoked and updated in the database.

```
def revoke_token(self, token, request):
    query("UPDATE oauth2_token SET revoked = True WHERE id =?",(token.id,))
    commit()
```

Figure 7: Code snippet from the revoke_token endpoint

Authorization Endpoint

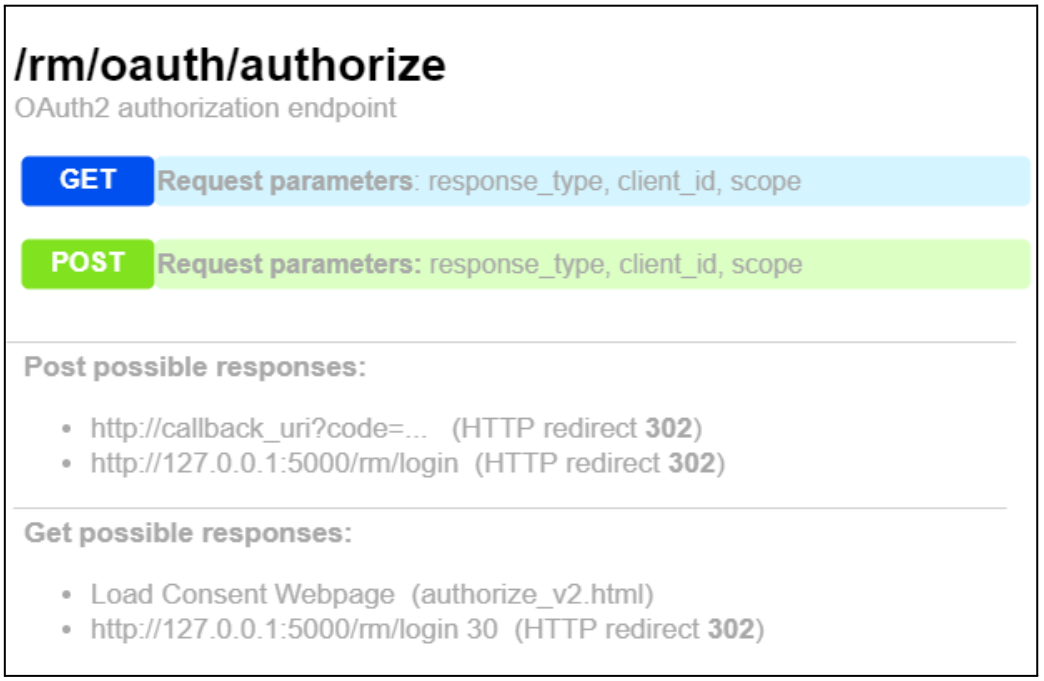


Figure 8: OAuth2 authorize endpoint

Implementation

The implementation of the authorization endpoint first verifies if the user is currently logged in or not via the session. If the user is not logged in, he will be redirected to the login page, else the consent webpage will be loaded. When the user presses the consent button, a post request is sent to this same endpoint, but this time it responds with the callback uri and the corresponding authorization code using the Authlib authorization server.

Login Endpoint

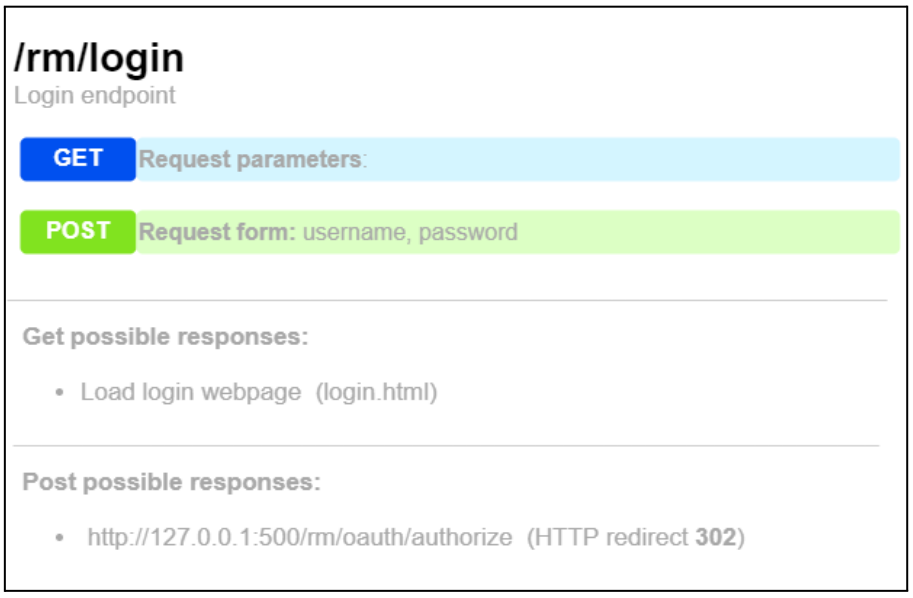


Figure 9: Rm login endpoint

Implementation

The implementation of the login endpoint first loads the login webpage on a GET request, where the user will fill the username and password.
After a POST request is sent to this same endpoint with the username and password. Due to requirements of the second part of this project (as we will explain), the password will be hashed using the PBKDF2 key derivation function as illustrated in figure 10 and both the username and hashed password will be queried in the database for a match.
If this match exists, the user will be redirected back to the authorize endpoint, else it will be redirected back to the login page.

```
kdf = PBKDF2HMAC(algorithm=hashes.SHA1(),length=16,salt=str.encode("IAA"),iterations=50000)
password = kdf.derive(str.encode(password))
```

Figure 10: Login endpoint code snippet

Token Endpoint

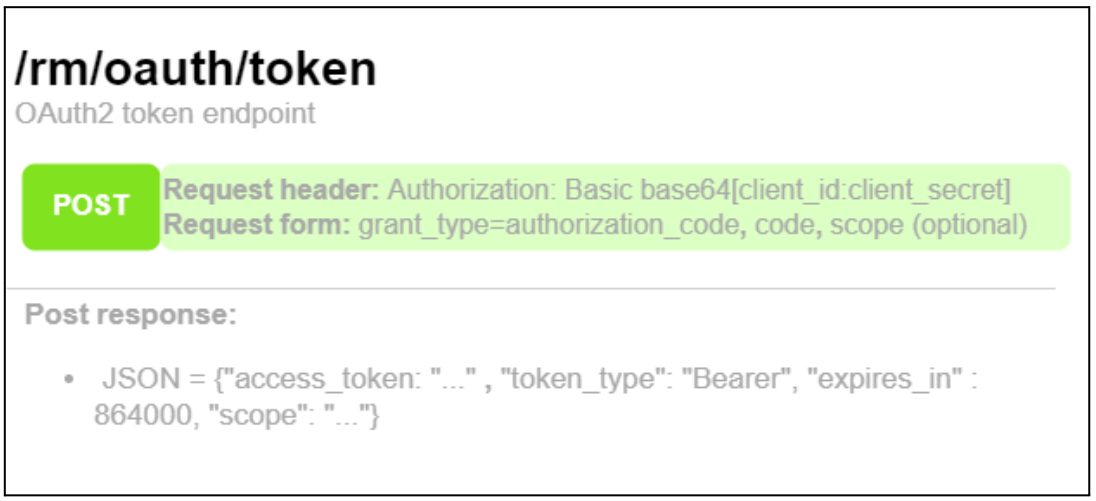


Figure 11: OAuth2 token endpoint

Implementation

The implementation of the token endpoint uses the Authlib authentication server not only to analyze and validate the request but also update the oauth2_token database table and respond with the json containing the access token and other information.

```
@rm_bp.route('/rm/oauth/token', methods=['POST'])
def issue_token():
    #Generate token response
    return authorization.create_token_response()
```

Figure 11: OAuth2 token endpoint code snippet

Revoke Endpoint

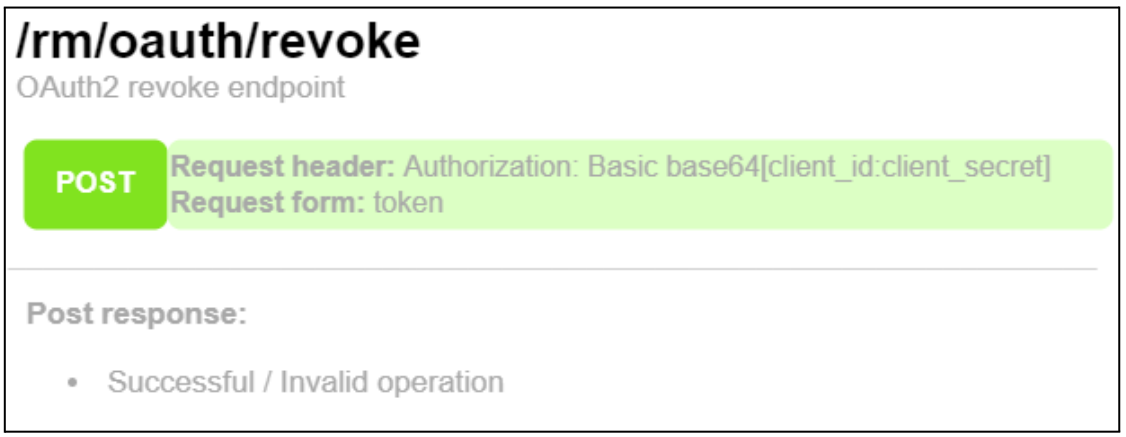


Figure 12: OAuth2 revoke endpoint

Implementation

The implementation of the revoke token endpoint uses the Authlib authentication server not only to analyze and validate the request but update the oauth2_token database table entry to revoke the access token.

```
@rm_bp.route('/rm/oauth/revoke', methods=['POST'])
def revoke_token():
    #Revoke access token
    return authorization.create_endpoint_response('revocation')
```

Figure 13: OAuth2 revoke endpoint code snippet

2.4.3 Resource server

The Resource Server is the other main component of the RM that's part of the Oauth 2.0 standard.
The Resource Server is responsible for protecting the resources via Oauth2 access tokens.
To create the Resource Server, the Authlib library was used in conjunction with Flask to provide API server access to the protected resources.

Data structure

The Resource Server portion of the database uses the previously mentioned Users table and keeps the game indicators for a user structured as follows:

TABLE: Games

Types	Name	Description
INTEGER	Id	Database id
VARCHAR	game	Game name
INTEGER	skill	Skill rating
INTEGER	behaviour	Behaviour rating
INTEGER	user_id	User identifier (Foreign key)

Figure 14: Games table in the database

Update Indicators Endpoint

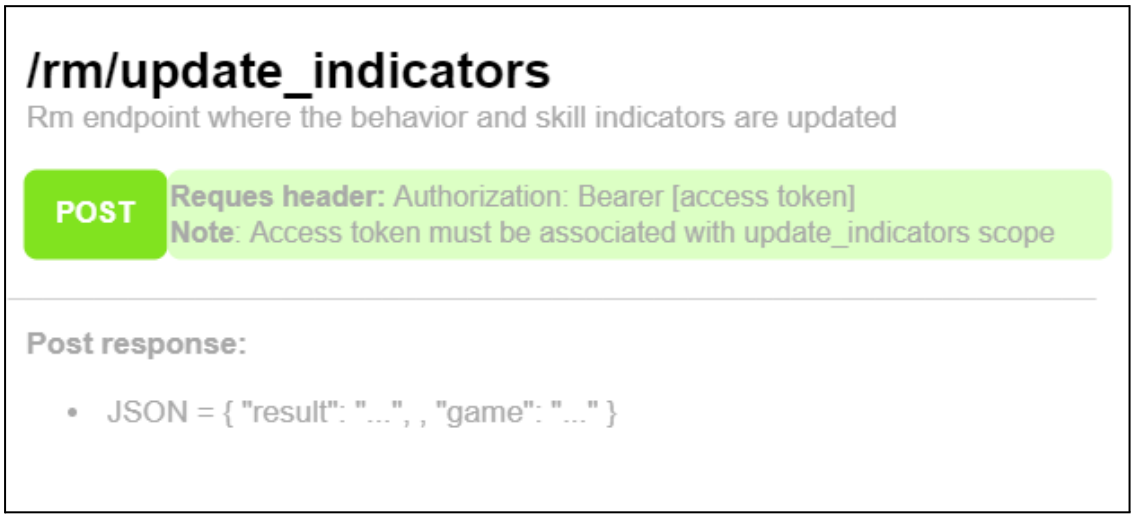


Figure 15: Rm update_indicators endpoint

Implementation

This endpoint uses the Authlib resource server to verify and validate the request. This includes the verification of the access token provided and the scope needed to access the endpoint (Figure 16).
If the access token is correct and has the scope “update_indicators”, the user will be identified using this token.
Given the result and game provided in the request json data, the database will be updated in the following manner:

- Win: +1 skill and behavior
- Lost: -1 skill and +1 behavior
- Draw: -
- Quit or Cheating: -1 skill

```
@rm_bp.route('/rm/update_indicators', methods=['POST'])
@require_oauth('update_indicators')
def update_indicators():
```

Figure 16: Rm update_indicators endpoint code snippet

Users Reputation Endpoint

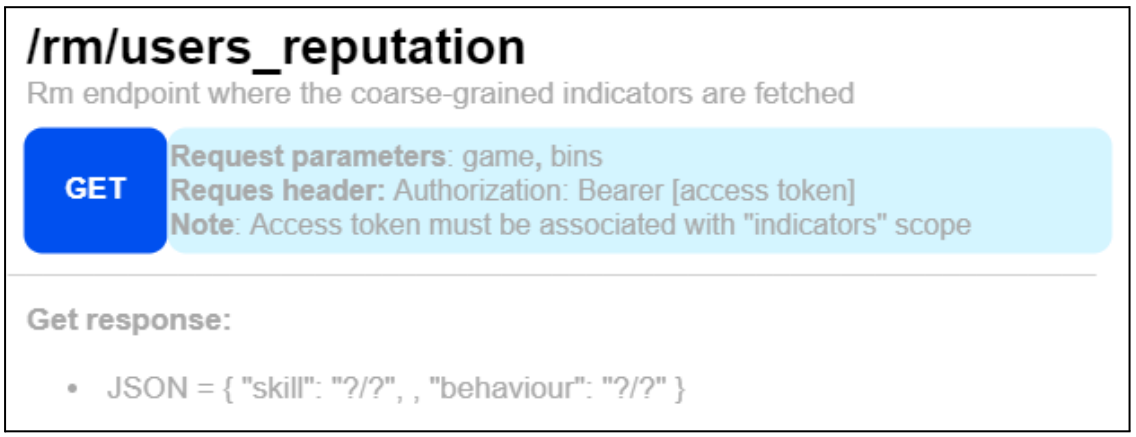


Figure 17: Rm users_reputation endpoint

Implementation

This endpoint uses the Authlib resource server to verify and validate the request. This includes the verification of the access token provided and the scope needed to access the endpoint (Figure 18).
If the access token is correct and has the scope “indicators”, the user will be identified using this token. After the user is identified, both the skill and behavior reputation are requested from the database and the coarse-grained form is calculated for each indicator.
This coarse-grained form was calculated in the following manner:

1. First all indicators in the database for the game in question are arranged in a descending order array.
2. Then the desired indicator position is found on the previous array and the indicator interval is calculated by dividing the total number of indicators on the database by the number of bins.
3. Finally the coarse-grained value is calculated by dividing (rounding up) the indicator position and the indicator interval.

All these steps can be observed in the code snipped provided in figure 19.

```
@rm_bp.route('/rm/users_reputation')
@require_oauth('indicators')
```

Figure 18: Rm users_reputation endpoint code snippet

```
#Calculate skill and behaviour bin
interval_skill = len(all_skill_sorted) / bins
interval_behaviour = len(all_behaviour_sorted) / bins

position_skill = all_skill_sorted.index(user_skill) + 1
position_behaviour = all_behaviour_sorted.index(user_behavior) + 1

bin_skill = math.ceil(position_skill / interval_skill)
bin_behaviour = math.ceil(position_behaviour / interval_behaviour)

value = {
    'skill': '{}/{}'.format(bin_skill,bins),
    'behaviour': '{}/{}'.format(bin_behaviour,bins)
}
```

Figure 19: Rm users_reputation endpoint code snippet

2.3. TM

2.3.1. Objective

The Table Manager objective is to organize online games with the candidate players. To organize these games, the TM must take into account the players preferences and reputation indicators.
The TM never truly knows the players identity, only their reputation in a coarse-grained form.

2.3.2. Oauth Client

Start Game Endpoint

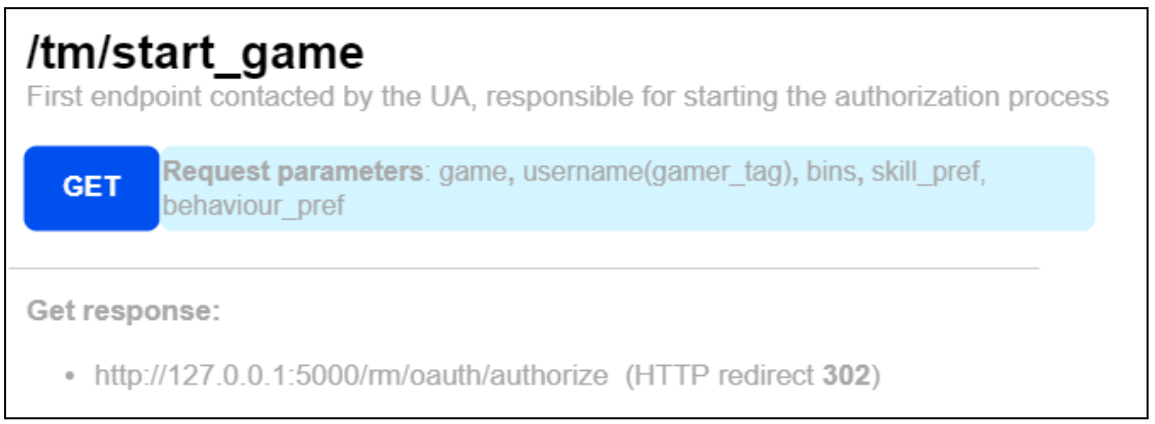


Figure 20: Tm start_game endpoint

Implementation

This endpoint is responsible for starting the authentication process so that the game can start. First it checks if the username (gamer tag) already exists and then saves the username, game, bins, skill preference and behavior preference from the player in the session.
It associates the username (gamer tag) with an oauth2_session custom class that wraps the OAuth2Session (Oauth2 client) from Authlib.
Lastly, using the oauth2_session, it redirects the user to the correct authorization url to start the authentication process.
The code snippet on figure 21 illustrates the process.

```
session['username'] = username
  ...
session['bins'] = request.args.get("bins")
session_list[username] = oauth2_session(client_id,client_secret,scope,authorize_url
return redirect(session_list[username].create_authorization_url(), code=302)
```

Figure 21: Tm start_game endpoint code snippet

Callback Endpoint

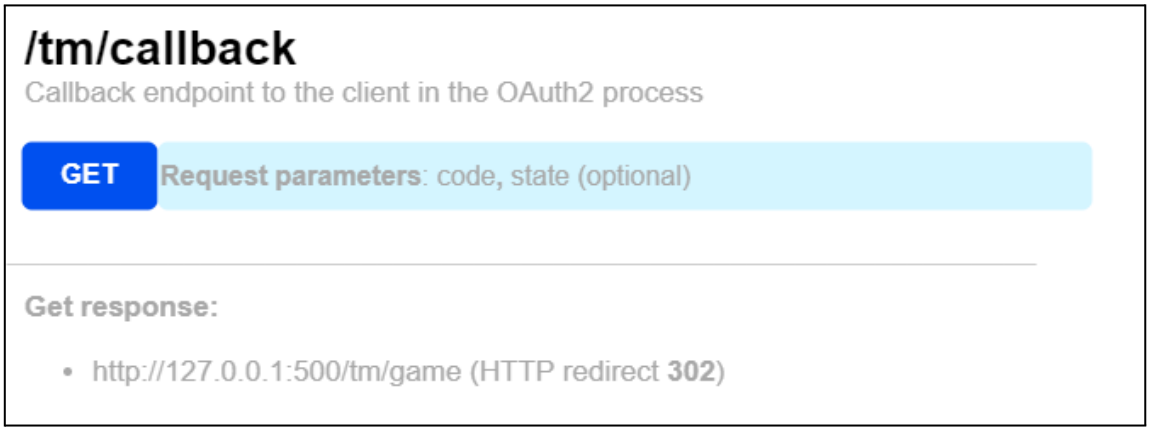


Figure 22: Tm callback endpoint

Implementation

The callback endpoint receives the authorization grant code and using the correspondent oauth2_session instance in relation to the username (gamer tag) kept in session, the access token will be fetched. This access token is then stored and the user is redirected to the game.
The code snippet on figure 23 illustrates the process.

```
token = session_list[session['username']].fetch_token(request.url)
session_list[session['username']].save_token(token)
return redirect('/tm/game')
```

Figure 23: Tm callback endpoint code snippet

Game Endpoint

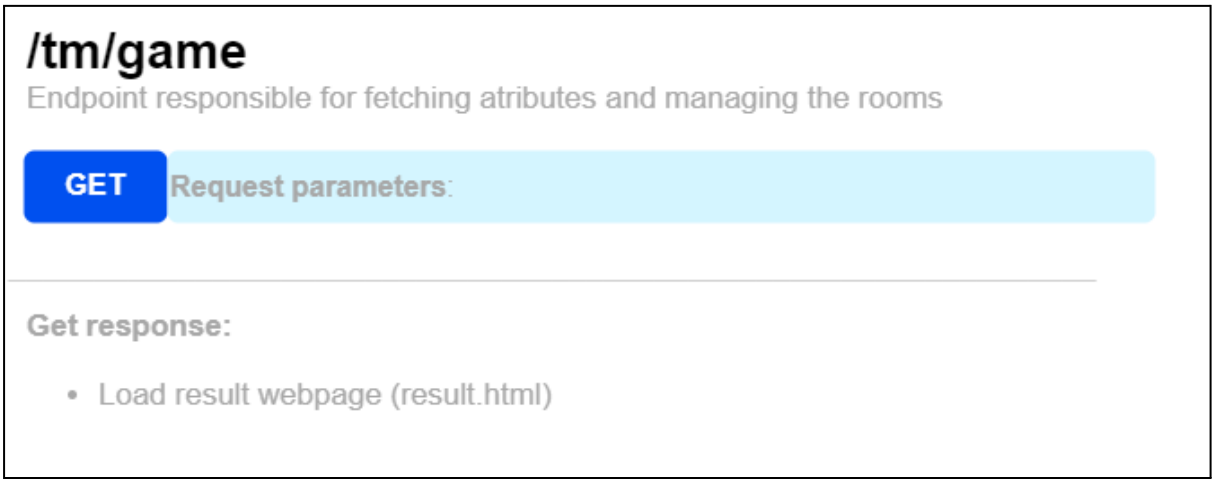


Figure 24: Tm game endpoint

Implementation

When this endpoint has been reached, the OAuth2 authentication process finishes and an access token to access protected endpoints already exists.
The first step of this endpoint's logic is to fetch the user coarse-grained reputation indicators using the corresponding oauth2_session that already contains the valid access token.
After the TM matches the current user with an available room, taking into account the game, the indicators and the preferences (explained in subsection 2.3.3. Matchmaking Logic).
In case the room is not full, a waiting page is rendered. When the room is full, the game can start and the MM generates the random result (subsection 2.4. MM).
With this random result, a socketio event is sent to the room with the corresponding results, the users webpages are updated, the result information is sent to the protected update indicators endpoint and finally the access token is revoked through the revoke endpoint.

2.3.3. Matchmaking Logic

The matchmaking logic of TM is responsible for matching players with each other taking into account the user skill, behavior and preferences by creating game rooms.
These game rooms contain an randomly generated ID, the game that is played in the room and a list of players that are currently waiting in the room for a match, as illustrated by figure 25.

```
room = {
    'id' : ''.join(random.choices(string.ascii_uppercase + string.digits, k = 10)),
    'game' : user.game,
    'players' : {
        user.user : user
    }
}
```

Figure 25: Tm (game_logic.py) code snippet

For each new player that starts a game, all pending rooms for the same game will be verified to try and fit the new player. To perform this verification the following validations must occur:

1. The room can not be full.

2. The preferences for both the new player and all players waiting in the room must be compatible, this means that if the user wants to match a player with a higher/lower skill/behavior rating, all players waiting in the room must have the opposite preference and the according levels. If no preference is chosen, the player can match with any room where the pending users have compatible preferences.

For example, for a room that is not empty, if the players already in a room want to play against an opponent that has a higher skill level, then the player that is trying to join the room must comply with that preference.

2.3.4. Matchmaking Status Output

When the flask server is running and players start to play games, the previously mentioned rooms start to be created for the players. These rooms contain an ID, game and the users that are currently waiting for a match.
All of this information is printed and updated on the terminal as illustrated in picture x:

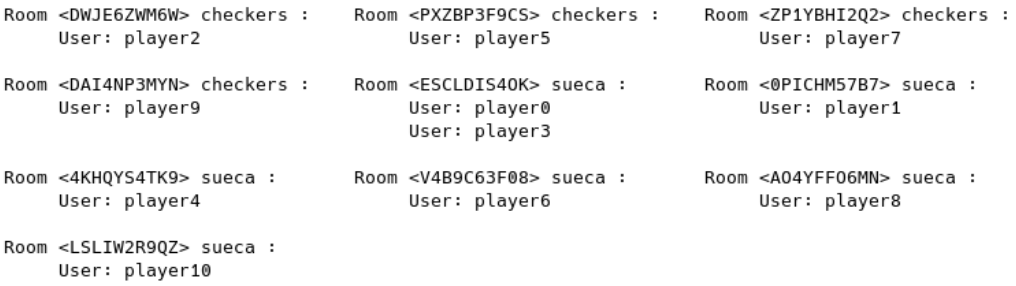


Figure 26: Display of the rooms in the console

2.4. MM

2.4.1. Objective

As mentioned in the beginning of this chapter the Match Manager (MM) is responsible for controlling the execution of a game and reporting the outcome to the Tables Matchmaker (TM). The outcome is selected at random and it includes winning, losing, cheating, quitting and drawing.

2.4.2. Implementation

As mentioned in the project description, since the games are not effectively run, we decided to implement this entity as a function that is available to the TM.
The function (called mm) is available in the file game_logic.py which is imported by the TM and contains several functions used by that entity.
The function takes as argument a dictionary that contains the id of the room, the name of the game and a dictionary that contains the players in that room (figure 25).
What the function does is get an array with the name of the players from the argument, generate a value between 0 and 10 (both included) and check whether the number generated is above 3. If so then we are in a normal win/lose scenario (occurs about 64% of the time assuming that the python random generator has the same probability for every number). If the number is below or equal to 3, then we will have a cheating/quitting scenario.
For our normal win/lose scenario the random generator is once again used to select the winner (if we are playing a 2 person game or 2 winners in case the game has 4 players) with the same probability for each player.

For our quitting cheating scenario we use the generator to both, with a 50 % chance, pick between quitting or cheating and then pick the affected player, the restant players will have a “draw” outcome.

The return value of this function is a dictionary with the name of the players as keys and the respective outcome as value.

```
def mm(room):

    number_of_players = len(room[0]['players'].keys())
    results = [a for a in room[0]['players'].keys()]
    results_dic = {}

    winlose_cheatingquiting = random.randint(0,10)

    if winlose_cheatingquiting <= 3:
        quitting_cheating = random.randint(0,1)

        affected_player = random.randint(0,number_of_players-1)
        for i in range(0,number_of_players):
            ...
    else:
        if (number_of_players == 2):
            affected_player = random.randint(0,number_of_players-1)
            for i in range(0,number_of_players):
                ...

        else:
            affected_player = random.sample(range(0, number_of_players-1), 2)
            for i in range(0,number_of_players):
                ...

    return results_dic
```

Figure 27: Code snippet from the mm function

2.5. User-Agent

2.5.1. Objective

The objective of the user agent is to represent the browser instance used by the user that participates in online games. Players use the User Agent to interact with the TM in order to be selected for a match.
To implement the user interfaces, we took as inspiration the code available in https://github.com/mariofornaroli/y-game-characters and https://github.com /mariofornaroli/y-responsive-login .

2.5.2. Game Selection Page

The game selection page (Figure 28) is used to select the game that the user wants to play. This webpage allows the user to insert a temporary username (gamer tag for that game) and change the number of bins, skill preference and behavior preference.
This webpage uses javascript to collect all of the previously mentioned attributes and perform the get request to "/tm/start_game".
The games available are chess and checkers, which are 2 person games and sueca which is a 4 person game in teams of 2.

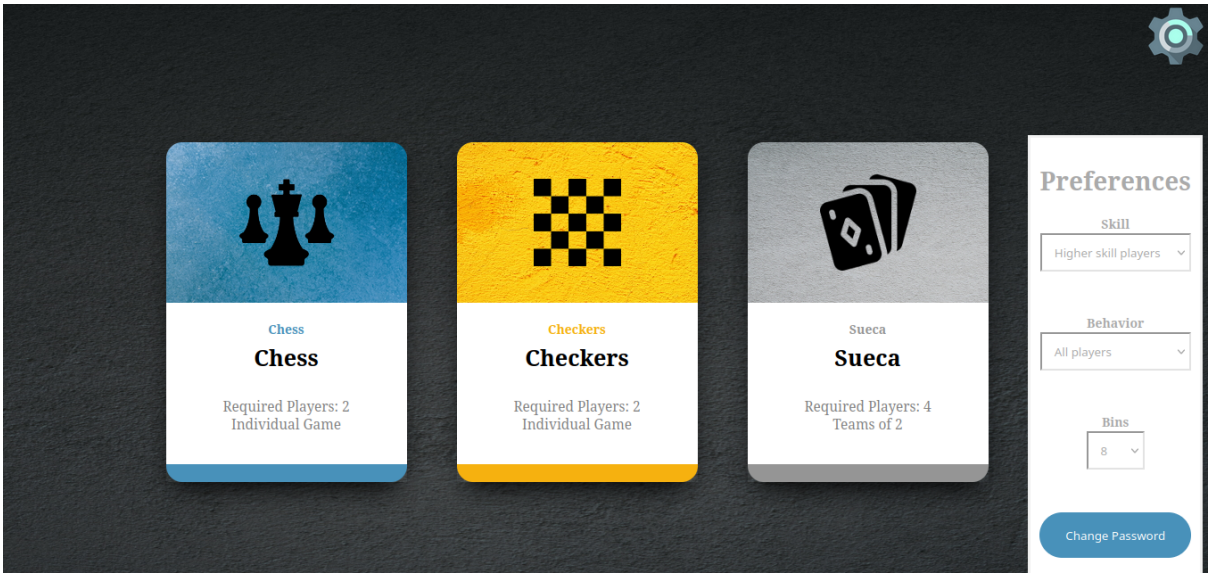


Figure 28: Game selection page

2.5.3. Authorize Page

The authorization page (Figure 29) is used by the user to authorize the TM to have access to the granulated skill/behavior and update the skill/behavior of the user according to the match outcome. If the user consents a POST request is performed to the "rm/oauth/authorize" endpoint.

2.5.5. Login Page

The login page (Figure 29) is used by the user to authenticate in the RM. The user can either authenticate with username and password or CMD (Chave Móvel Digital). The second authentication method (CMD) was implemented in the second part of this project.

The regular username and password login sends a POST request to the "/rm/login" endpoint with this information.

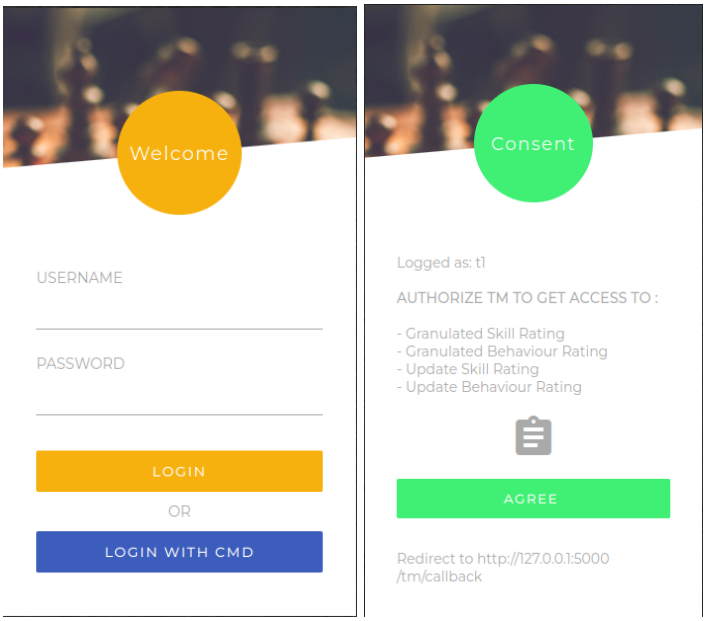


Figure 29: Login page on the left and authorization page on the right

2.5.4. Result Page

The result page is responsible for showing the user either the outcome of the match or the waiting for the opponent's message (Figure 30)
This webpage is automatically updated via javascript and websockets.
The websocket listens on the room ID that was attributed by the TM as mentioned in 2.3.3. Matchmaking Logic and when a room is full, the TM sends the outcome message to this socket and the javascript websocket updates the current message displayed to the user from waiting to either win, lost, cheat, draw and quit.

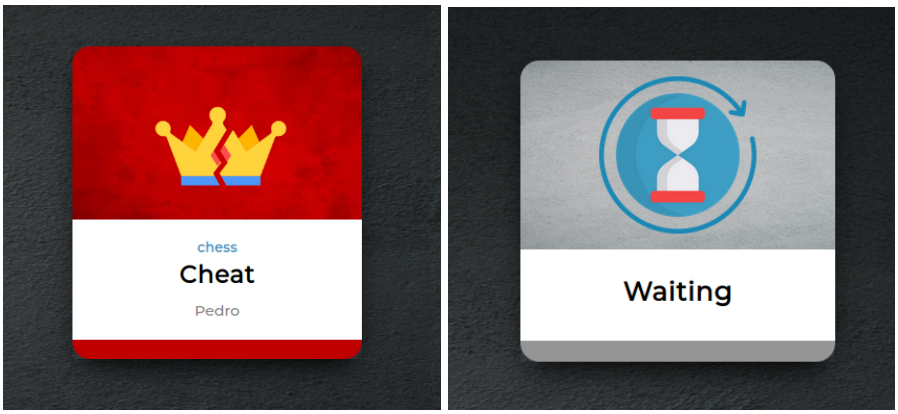


Figure 30: Examples of the results webpage

2.6. Bots

In this subsection we will explain how we developed a script that launches a desired number of threats that will act as bots and get in rooms to play a specific number of rounds. This was done mostly to help us test the system.

2.6.1. How to run

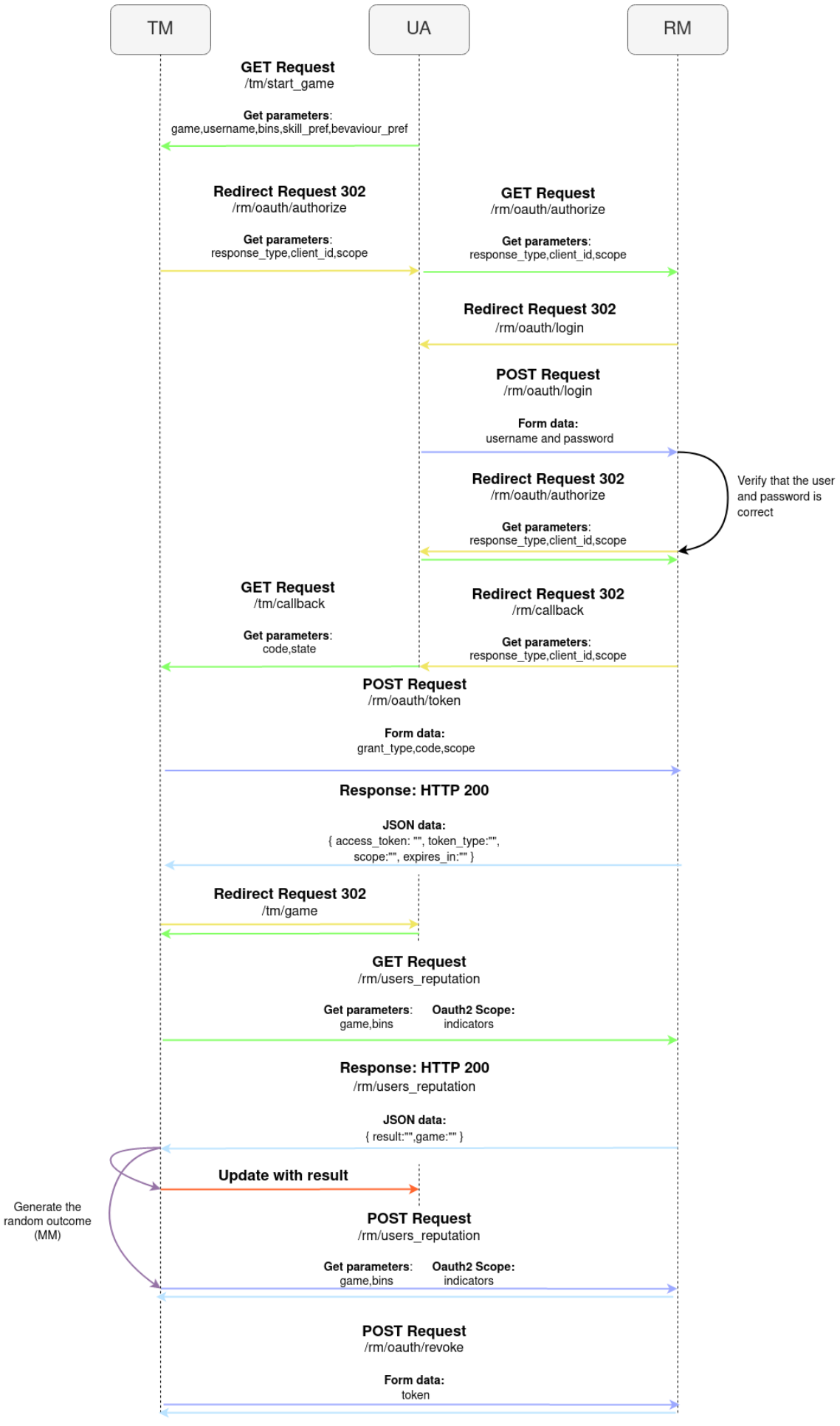
To execute the bots the flask server must be running on port 5000.
In the command used to execute the bots script, the user must specify the number of players (-p) and rounds (-r) that will be performed. Optionally the number of bins (-b) can be specified and a flag exists (-np) to make players have no skill/behavior preferences:

```
$ python3 bots.py -r [Rounds] -p [Players] -b [Bins] [-np]
```

2.6.2. Implementation

This script uses threads to create temporary bots that play random games.
The number of rounds indicates how many games the bots will try to play, in the case that they have already finished a match.
The first step of the implementation process is to generate random player names, bins and preferences according to the arguments provided by the user.
The bot's username (gamer tag) is created sequentially using the word “player” concatenated with the sequential number, for example “player1”, “player2” etc.
If the bot’s user and password do not exist in the database, an entry will be automatically added with the default password “pwd”.
Each bot will then replicate the full workflow to play a game using the requests library, exactly like a regular user playing in a browser would on seperate threads.
The result of the player's game will be displayed on the console output and saved on the bot_log.txt file.

2.7 Request-Response Diagram



3. Second part of the project

The second part of the project consists in making an improvement to the first part of the project, mainly on adapting the RM to use the portuguese government *chave movel digital* (CMD) as an external IdP.
The adaptation consists of adding the following features to the RM:

- Allowing the creation of personal reputation profiles using attributes from the external IdP that uniquely identify the player.
  Upon the creation of personal profiles, create an alternative set of access credentials formed by a user name and password.
- Authenticate in the RM using the external IdP or the username and password previously defined.
- Change the profile password upon a CMD-base authentication

To get the attributes from the external IdP we were presented with two options, either use SAML or OAuth2.0 with an implicit grant flow. We chose OAuth2.0 as we were already more familiar with OAuth from the first part of the project.

3.1. Activation of CMD in pre-production

Before implementing any changes/features in the code we needed to register our *chave movel digital* in the pre-production system so that we could make tests during our development process.
To activate the CMD the first step is to head over to the registration website available at (Autenticação Gov (autenticacao.gov.pt)). Once there, the next step is to click on the activate CMD where some options are presented, however the only option that works through online methods is to use the citizen card and its respective authentication pin.
To perform this method there are 2 additional requirements, which are, having a card reader available and software that makes the communication between the card and the website javascript possible. A link and instructions to download that software (plugin autenticação-gov) are made available during the process.
Having all of this is all that is left is to authenticate with the card and the pin, associate your CMD with your phone number and define a new pin for the CMD.

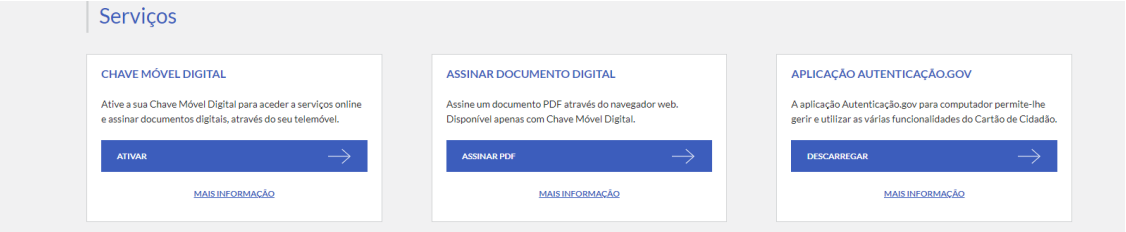


Figure 31: Website for the activation of CMD

3.2. Creation of personal profiles using the external IdP and alternative credentials

To complete the first objective of allowing the creation of personal profiles using the external IdP, we first needed to understand how we could fetch the Oauth access token to then get the attributes from the CMD IdP.
In order to do that we had to read the documentation available in the quick use guide available in the propreduction github .

3.2.1. Integration with CMD (obtaining the access token)

Figure 32: Obtaining the access token flow

As per the documentation the first step is to make a request to the attribute provider with the following fields:

- **response_type:** With the value token

- **client_id**: The identifier of the supplicant system (this should be accorded previously for instance using the form available in the CMD github). Also there is no need for client_secret as we are dealing with implicit grant flow.

- **redirect_uri**: The url that will be used to go back to the supplicant system.

- **scope**: A list with the attributes that we are requesting separated by a whitespace.

Note: it also contains some other optional fields that were not used.

We already know the response type as it is a static value and we also know the client_id as it was previously accorded and was made available for the students of the class to use in the project (that value being "9113170755799990166").
For the redirect uri we also knew that we had to use the domain 127.0.0.1 and it would be an endpoint to the RM. So we decided that the redirect_uri would be "http://127.0.0.1:5000/rm/callback".
In terms of the attributes that were gonna be used, as mentioned in the project description we had to request attributes that would uniquely identify the players in our system. We concluded that the best attribute to use was the NIC (*Número de identificação civil*) which is already a mandatory attribute to include in the scope and uniquely identifies a person without being such an intrusive attribute in terms of privacy.
With that being said we developed an endpoint in our RM that redirects the user to make the request to the attribute provider, presented in the variable auth_req in figure 33.

```
client_id="9113170755799990166"
scopes = "http://interop.gov.pt/MDC/Cidadao/NIC"
redirect_uri = "http://127.0.0.1:5000/rm/callback"

...

@rm_bp.route('/rm/authCMD', methods=['GET'])
def authorize_CMD():
    auth_req = "https://preprod.autenticacao.gov.pt/oauth/askauthorization\
        ?redirect_uri={}&client_id={}&scope={}&response_type=token"
    return redirect(auth_req.format(redirect_uri,client_id,scopes), code=302)
```

Figure 33: Rm authCMD endpoint code snippet

Once that redirect is made, a webpage from the government will be loaded where the user must first authorize the use of the attributes, in our case NIC (*Número de identificação civil*) and then authenticate itself using its CMD.

Figure 34: Authorizing RM to use the NIC attribute

Once that process is done, the webpage will redirect us back to redirect_uri that was passed in the previous GET with some parameters including the access_token.

http://127.0.0.1:5000/rm/callback#access_token=dc3aadb7-3dbb-4256-9b89-d454c5f9addc&token_type=bearer&expires_in=5000&state=&refresh_token=eyJhbGc…

Figure 35: Example of the returned value.

As we mentioned previously, the endpoint that we decided to be the callback to our RM was the "/rm/callback"endpoint.
Essentially the objective of the callback endpoint in our application is to fetch the access token that is returned by the IdP and fetch the attributes, that second part will be explained in the next subsection.
While trying to fetch the access token from the returned url we depared ourselves with a small inconvenience. That inconvenience being the fact that the access token came after a hashtag in the URL, meaning that values after the hashtag are never sent to the server. To work around that we made the callback endpoint redirect the browser to a temporary webpage (access_token_fetch.html) that uses javascript to replace the "#" before the "access_token" to a "?" and switch the location to that url, meaning that it will go back to the callback endpoint. This is only done if the URL does not contain the "?access_token" so it only happens once.
After that we use regex to extract the token from the URL.

```
<script>
    window.onload = function() {
        url = window.location.href
        url = url.replace('#','?')
        window.location.replace(url)
    }
</script>
```

Figura 36: Javascript used to replace the hashtag

```
@rm_bp.route('/rm/callback', methods=('GET', 'POST'))
def callback():
    if '?access_token' not in request.url :
            return render_template('access_token_fetch.html')

    access_token = re.search('\?access_token=(.+?)(?=&)',request.url).groups()[0]
    ...
```

Figura 37: Fetching access token in callback endpoint code snippet.

3.2.2. Integration with CMD (obtaining the the attributes using the token)

Having the access token in the callback endpoint the next step is to fetch the NIC attribute from the CMD IdP (this will still be done inside the callback endpoint).
As per the documentation available, to fetch the attributes we first need to make a HTTP POST to the attribute provider (FA) containing a JSON with the following values.

- **token:** The value of the access token previously obtained.
- **attributesName:** parameter that specifies the attributes to get.

This is implemented in our callback endpoint with the code in figure 38.

```
body = {'token':access_token,'attributesName':[scopes]}
token_context = requests.post("https://preprod.autenticacao.gov.pt/oauthresourceserver/api/AttributeManager",data=body)
```

Figura 38: First step to fetch the attributes callback endpoint

The return value for this POST is a JSON object that contains once again the token that was used and also the authenticationContextId.
These values are then used to make a GET request to the Attribute provider, where the answer will return the attributes requested.

```
api = 'https://preprod.autenticacao.gov.pt/oauthresourceserver/api/AttributeManager?token={}&authenticationContextId={}'

token_context_json = token_context.json()

api_request_url = api.format(token_context_json['token'],token_context_json['authenticationContextId'])
attributes_req = requests.get(api_request_url)

attr_val = [a['value'] for a in attributes_req.json() if a['name'] == "http://interop.gov.pt/MDC/Cidadao/NIC"]
```

Figura 39: Fetching attributes in callback endpoint.

The callback endpoint then before registering or authenticating the player (as we will explain in the previous subsection), uses PBKDF2 key derivation function from the cryptography library as to not save/handle the sensitive attribute in cleartext and mitigate damages from attacks such as dictionary attacks.
The PBKDF2 implemented uses SHA1 as its hash algorithm, uses the string “IAA” as its salt and 50000 iterations.

```
kdf = PBKDF2HMAC(algorithm=hashes.SHA1(),length=16,salt=str.encode("IAA"),iterations=50000)
NIC = kdf.derive(str.encode(attr_val[0]))
```

Figura 40: Hashing the NIC attribute with PBKDF2.

3.2.3. Creation of the profile and alternative set of credentials

The creation of the profile begins when a user (not logged in) in the main page picks a random gamer tag and decides to play a game by login in with CMD for the first time.

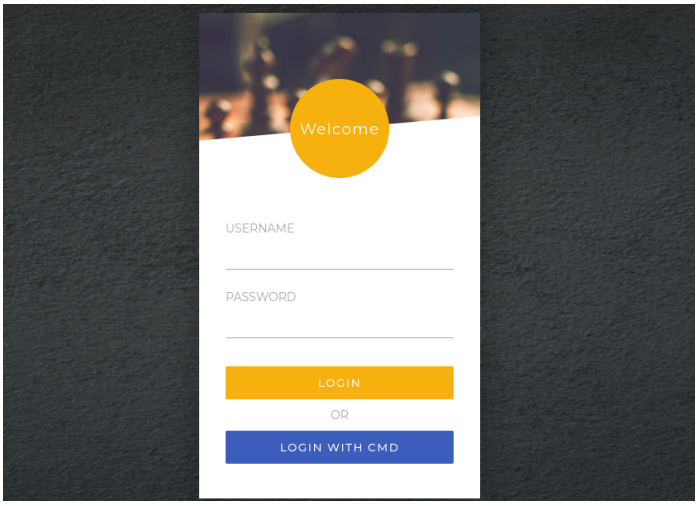


Figura 41: Login web page with login with CMD option.

This “Login with CMD” button will redirect the user to the authCMD endpoint, which as we mentioned previously starts the process of authentication and authorization to use the NIC attribute.
Once that authorization and authentication is done, we will go back to the callback endpoint where as we explained, the attribute(NIC) is fetched and hashed using the PBKDF2.
After that, the callback endpoint checks whether there is a user in the rm table with that hashed NIC (in the user table it corresponds to the cmd_id column and this attribute is optional in the database so that the bots can also play).

User table			
id	**username**	**password** (password hash with PBKDF2)	**cmd_id** (NIC hashed with PBKDF2)

Figura 42: User table blueprint in the database

If there is not a user in the database that contains that hashed NIC in the cmd_id field, then the endpoint will redirect to our /rm/register_cmd endpoint. This new endpoint starts by rendering a html page (register.html) where the user will enter a username and password that will be used as an alternative set of credentials.

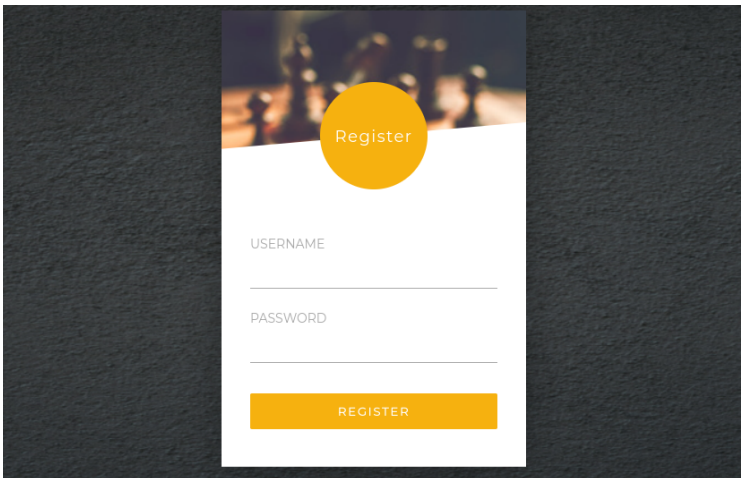


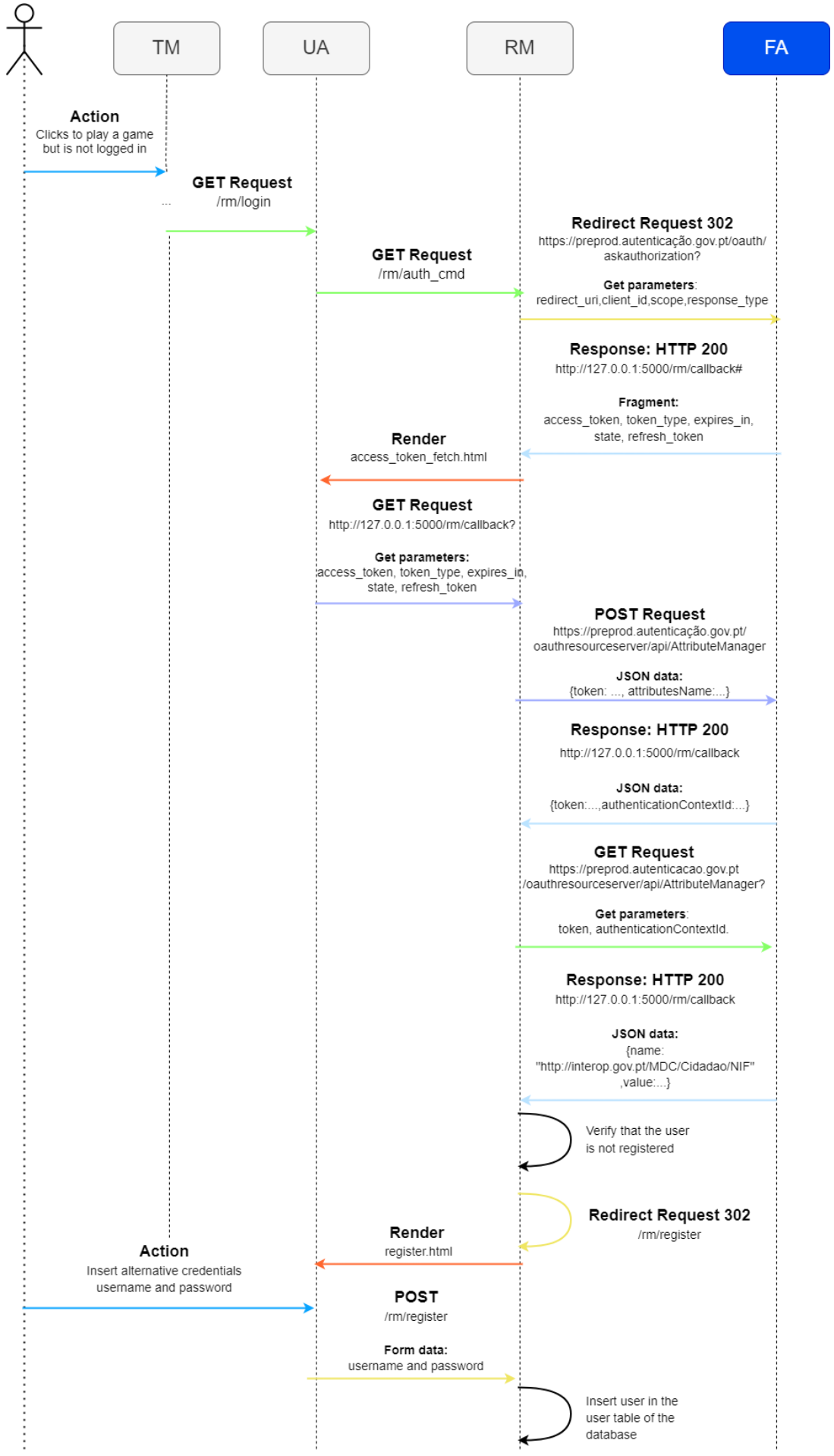
Figura 43: Register webpage.

When the “Register” button is pressed a POST with the username and password will be forwarded to the register_cmd endpoint, where the new user will be inserted in the database associated with its username, transformed password and transformed NIC.

```
kdf = PBKDF2HMAC(algorithm=hashes.SHA1(),length=16,salt=str.encode("IAA"),iterations=50000)
password = kdf.derive(str.encode(password))
queryString = "INSERT INTO user(username,password,cmd_id) VALUES (?,?,?) "
values = (username,password,session['NIC'])
query(queryString, values)
commit()
```

Figura 44: Code Snippet from the register_cmd endpoint.

3.2.4. Request-Response Diagram



3.3. Authentication on RM using the external IdP or username and password.

To achieve this objective there was not much that we had to actually change or do because of the way we were already developing the project. For one we already had an authentication system based on a username and password for the first part of the project, for the authentication as we explained previously, the creation of the profile is just the authentication flow with extra steps.

3.3.1. Authentication on RM using the external IdP

The process is fairly similar to the one explained in the creation of profiles, except the user is already in the database thus no registration process is needed.
So this process begins when a user that is not already logged in declares that he wants to play a game.
This will prompt the login page, where the user will select the option to login with the “Login with CMD” option. As explained previously this will start the process of obtaining the attributes from the CMD IdP.
Once those attributes are fetched and we are on the rm callback endpoint, we will query the user table for the user that contains the transformed NIC, if one is found instead of registering the user as we mentioned previously we save the id of the user in a session variable.
After getting the user based on the attribute that came from the CMD IdP we are logged in and the rest of the process (authorizing the TM and playing the game) will begin.

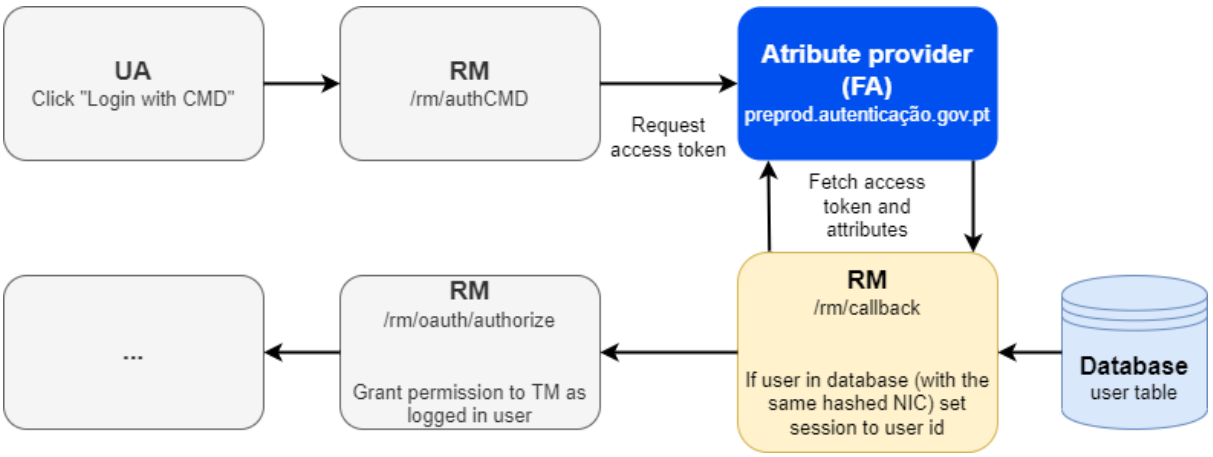


Figura 45: Authenticating in the RM with CMD flow.

3.3.2. Authentication on RM using the alternative set of credentials

As we mentioned in the first part of the project, we had already developed an authentication based on username and password. Also as we mentioned in the previous subsection 3.2.3, the alternative set of credentials is stored upon the creation of the personal profile in the user table.
With that being said to login with the alternative set of credentials the user just needs to use that alternative set in the login.html webpage and the system will fetch the user from the database that has the corresponding transformed password (remember that the password stored in the database is a transformation of the password using PBKDF2).

```
@rm_bp.route('/rm/login', methods=('GET', 'POST'))
def login():
    if request.method == 'GET':
        return render_template('login.html')

    #Get credentials from login page
    username = request.form.get('username')
    password = request.form.get('password')

    #Transform the password to comapare with database
    kdf = PBKDF2HMAC(algorithm=hashes.SHA1(),length=16,salt=str.encode("IAA"),iterations=50000)
    password = kdf.derive(str.encode(password))

    #Get the user with corresponding username and password
    user = query("SELECT id from user where username = ? and password = ?;",(username,password,))

    #If the user exists redirect back to the authorization URL
    if user != []:
        session['id'] = user[0][0]
        return redirect(session['authUrl'], code=302)

    #If the user does not exist reditect back to the login page
    return redirect('/rm/login')
```

Figura 46: Code Snippet from the rm login endpoint.

3.3. Change profile password upon CMD-based authentication

In this section we will explain how we implement a feature that allows changing the password of a user after they authenticate with the CMD.
To use that feature we added a button in the landing page settings called “Change password”. When the user clicks on that button a GET request will be made to the endpoint /rm/preferences.

The first thing that the endpoint does is verify if the user is already logged in. If not then the user will be redirected to the /rm/authCMD endpoint where he will authenticate himself and come back to this endpoint (session['authUrl'] will have the url to the /rm/preferences).

```
user = current_user()
# if user log status is not true (Auth server), then to log it in
if not user:
    session['authUrl'] = request.url
    return redirect('/rm/authCMD',code=302)
```

Figura 47: Code Snippet from the rm/preferences endpoint.

Then, the endpoint, if the user exists, will verify if it has the hashed attribute in the table user, this as to be done since the bots also authenticate with username and password but do not have the NIC attribute in the database.

```
if user.get_cmd_id() == None:
    session['authUrl'] = request.url
    return redirect('/rm/authCMD',code=302)
```

Figura 48: Code Snippet from the rm/preferences endpoint.

After knowing that we have a user and he is authenticated with CMD we will render the webpage where the user will insert the new password (change_preferences.html)

```
if request.method == 'GET':
    session['callback'] = request.args.get("callback")
    return render_template('change_preferences.html')
```

Figura 49: Code Snippet from the rm/preferences endpoint.

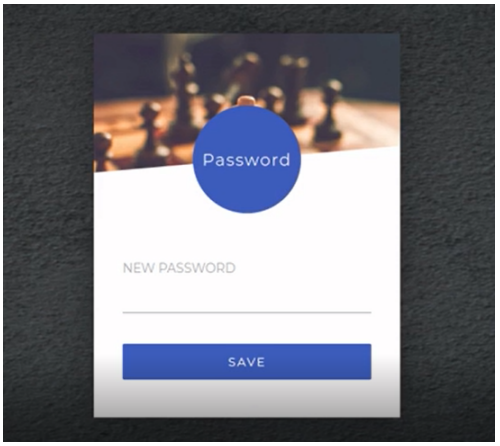


Figura 50: change password web page (change_preferences.html).

Once the user has introduced the new password on the webpage and clicked to save then a POST request will be made to the /rm/preferences endpoint where the new password will be hashed using PBKDF2 and inserted in the database.

```
password = request.form.get('password')
kdf = PBKDF2HMAC(algorithm=hashes.SHA1(),length=16,salt=str.encode("IAA"),iterations=50000)
password = kdf.derive(str.encode(password))
query("UPDATE user SET password = ? WHERE id = ?",(password,user.id))
commit()
```

Figura 51: Code Snippet from the rm/preferences endpoint.