Development of Backend REST API for Auto Chess Multiplayer Game with Multi-Historical Setting

Ayman Aljarbouh*, Dmytro Zubov, Aktan Tursunbaev
Department of Computer Science, University of Central Asia, 722918 Naryn, Kyrgyzstan.

* Corresponding author. Tel.: +996 770 822 972; email: ayman.aljarbouh@ucentralasia.org (A.A.)
Manuscript submitted February 18, 2023; revised May 8, 2023; accepted June 10, 2023; published January 12, 2024.
doi: 10.17706/jsw.19.1.9-18

Abstract: In this paper, a new technique for better management of winning conditions in online auto chess multi-player games is proposed. This technique is based on the development of a dynamic API that communicates with the server's database in real-time with minimum user intervention. Many existing mechanisms for auto-gaming rely on HTTP, which needs constant update requests from the client/player. To handle this problem, web sockets are employed that allow real-time communication between the client/player and the server, so the updates are sent immediately over the network whenever they are available. The proposed methods were validated by the practical implementation and a set of experimental tests. The web application uses Python Django REST Framework, allowing the extension of the web application to API. The database is implemented employing the PostgreSQL database management system.

Keywords: API, auto chess, backend REST, multiplayer game, web application.

1. Introduction

The auto chess genre is a tactical survival strategy, where the hero units (chess-like pieces) are placed on a grid board against the opponent's set of hero units. The match consists of several rounds, players are randomly divided into pairs. The main concept of the game is that players have to arrange their units efficiently using strategic and tactical thinking skills. The player may get a random set of hero units from the predefined pool. The winner of the round is the player whose units defeat the opponent's whole squad. The match ends when there is only one player left with a non-empty health bar [1–6].

Typically, a multiplayer game implies that the players need in-game interaction with each other, as well as a server and a database are required both. Web Application Programming Interface (API) is an interface that provides server-client communication [7–9]. API grants permission to register and authorize players, find a match, view the leaderboard, etc. All these actions are executed as the Hyper Text Transfer Protocol (HTTP) requests to the API that handles them accordingly. The development of API is crucial since it allows the clients to communicate with the database in a simple way. However, one of the main disadvantages of using the HTTP protocol in auto-gaming is that does not provide real-time server-client communication [10]. One possible solution to deal with this issue is to use web sockets. The web socket is the event-driven protocol that holds a persistent connection, allowing real-time communication [10, 11]. For instance, in matchmaking, whenever a player sends a request to find a match, the web socket connection is built to provide information to the waiting player about the actual status of the search for a match in real-time. To the extent of making the web application of the game accessible for the clients/players, it is necessary to deploy it. Moreover, the server needs to know how to redirect incoming requests to the web application. One of the methods to achieve this is to use the Asynchronous Server Gateway Interface (ASGI) and Daphne platforms [12]. ASGI is a descendant of Web Server Gateway Interface, which is a standard for asynchronous server communication with Python web applications employing web sockets. Daphne is a Python ASGI server that allows running multiple instances of web applications using incoming requests [13]. This paper introduces a new technique and application for better management of winning conditions in online auto chess multi-player games.

The contribution of this paper could be divided into three parts: (1) the development of Web API that provides better communication between the clients and database, (2) the implementation of an online matchmaking system and playing process handling, and (3) the deployment of the application to the server. Python and Django programming platforms are used in this work as they both have powerful and flexible web API features with built-in authentication options. The Django Channels library is also considered to handle web sockets and can be easily integrated into the existing Django application. An efficient search-and-find matching algorithm is implemented taking into consideration many factors for auto-matching such as the player's rating. The web application is deployed using the Heroku server, because it satisfies the minimal conditions to run the application for testing purposes. Also, the Heroku server has a built-in pack for Python applications that allows an automatic rebuild if the application is modified.

The paper is organized as follows: Section II presents the methodology and implementation; Section III discusses the conclusion and perspectives.

2. Methodology and Implementation

2.1. Development of the Web API

To develop the Web API, the Django framework is employed because it is a high-level Python web framework for reliable development. PostgreSQL is used as a database management system (DBMS) because it is an open-source powerful relational DBMS with high performance. Django uses the Model–View–Template (MVT) pattern to design applications, which is actually a modification of a classic Model–View–Controller (MVC) pattern. MVT pattern implies that the application is represented by three components: model, view, and template. First, the model component represents a simplified and more abstract view of the database. Each model corresponds to a particular database table, where models are defined as Python classes containing attributes and serve as the database table fields. Second, the view component stands for wrapping the logic of dealing with incoming requests from the client and sending responses. Views can be implemented in Python as functions accepting a request as a parameter and returning the HTTP response, or classes containing methods that perform the same way as function-based views. Third, HTML files combined with specific Django code files represent the template component. This code is run by the Django framework to render a classical HTML file and sends it back as the HTTP response. Fig. 1 shows the MVT pattern visualization.

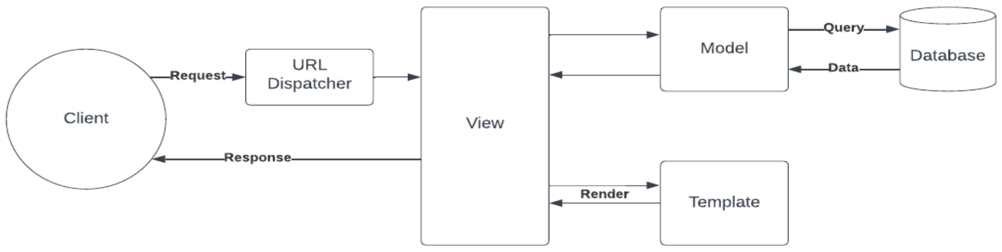

Fig. 1. MVT pattern visualization.

Instead of returning HTML files, we reconfigure the API so that the data is returned in the JavaScript Object Notation (JSON) format with the help of the Django Representational State Transfer (REST) Framework (DRF). The API resulting architecture is presented in Fig. 2.

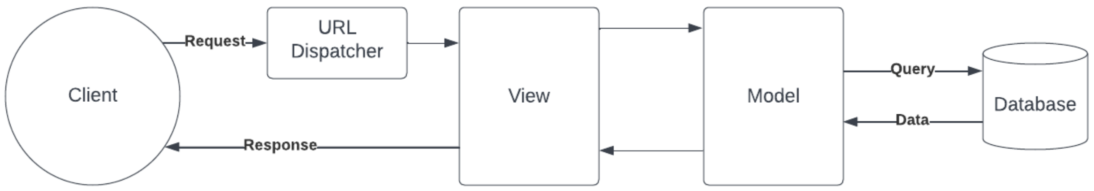

Fig. 2. Modified MVT pattern visualization with the JSON data formatting.

2.1.1. Designing the database

Designing a well-structured database schema makes the development simple throughout the whole process. As was mentioned above, the Django models help to work with the database. For instance, since this is a multiplayer game, the first model is the *Player* model. This model was created using the default *User* model of the DRF, i.e., there is a one-to-one relationship between the *Player* model and the *User* model. The default *User* model includes vital fields necessary for any user and already has the authentication features ready for that. The *Player* model is extending the default *User* model by adding the *rating*, *status*, *opponent*, *health*, and *channel_name* fields. The *rating* field serves as an indicator of the player's skill level. The *status*, *opponent*, *health*, and *channel_name* fields are helper fields that are employed in the matching processing. Fig. 3 shows the code snippet from the implementation of the database. The model is a Python class inherited from the calss *django.db.models.Model* representing the database table. This class contains the attributes that use the table field representations provided by the class *django.db.models*.

```
class Player(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    rating = models.IntegerField()
    status = models.CharField(max_length=255, default="unknown")
    opponent = models.ForeignKey('Player', on_delete=models.DO_NOTHING,
blank=True, null=True)
    health = models.IntegerField(default=100)
    channel_name = models.CharField(max_length=255, default="")
```

Fig. 3. Code snippet from the implementation of the database.

2.1.2. Implementing the API

Serializers are important in the implementation of the API since they help in the conversion of an instance of a model to JSON format and vice versa. Fig. 4 shows the code snippet for the implementation of serializers for the *User* and *Player* models.

```python
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['username', 'password', 'email']
        read_only_fields = ['last_login', 'date_joined']
        extra_kwargs = {'password': {'write_only': True}}
```

Fig. 4. Code snippet for the implementation of serializers for the *User* and *Player* models.

Serializers are Python classes inheriting *rest_framework.serializers.ModelSerializer* which allows the creation of a serializer for a specific model. *UserSerializer* is a serializer for the default DRF *User* model that allows interaction with username, email, password, *last_login* and *date_joined* fields, where *last_login,* and *date_joined* are read-only fields meaning that they cannot be changed, and password is a write-only field meaning that it cannot be read but can be changed. It is used in *PlayerSerializer* because of the one-to-one relationship in the database (see Fig. 5).

```python
class PlayerSerializer(serializers.ModelSerializer):
    user = UserSerializer()
    class Meta:
        model = Player
        fields = ['user', 'rating']
        read_only_fields = ['rating']
    def create(self, validated_data):
        user = User(
            username=validated_data['user']['username'],
            email=validated_data['user']['email'],
        )
        user.set_password(validated_data['user']['password'])
        user.save()
        player = Player(
            user=user,
            rating=validated_data['rating']
        )
        player.save()
        return player
```

Fig. 5. Code snippet for the implementation of *Playerserializer*.

PlayerSerializer allows to interact the client with related *user* and *rating* fields, where the *rating* is a read-only field because it needs to be changed only by the system. Additionally, the serializer does not show other *Player* fields since they are used in match processing. Also, the default *create* method of *ModelSerializer* was overwritten because it cannot automatically handle nested structures. In this case, the related user is a nested structure that needs to be manually unpacked and properly saved to the database.

Before implementing the API endpoints, the project was configured to use Token Authentication. The token is granted when the user logs into the system by providing a username and password so that the user does not need to send the credentials again. For further requests, the token is stored on the client side, e.g., browser cookies, mobile device, etc. DRF provides built-in token-based authentication that is easily configured. API endpoints are created using the DRF because it simplifies the whole process by providing generic classes that can work with existing models and automatically generates the four basic operations of persistent storage (Create, Read, Update, and Delete). Classes provided by DRF can be modified to alter their behavior. For instance, the handling of specific HTTP methods can be easily defined. For further explanation, the class *PlayerList* (see Fig. 6) was created for two endpoints that allow the creation of a player and retrieve the list of all players.

```
class PlayerList(generics.ListCreateAPIView):
    queryset = Player.objects.all()
    serializer_class = PlayerSerializer
    def post(self, request, *args, **kwargs):
        try:
            if request.POST:
                user = User.objects.create_user(
                    username=request.POST.get('user.username'),
                    email=request.POST.get('user.email'),
                    password=request.POST.get('user.password')
                )
            else:
                user = User.objects.create_user(
                    username=request.data.get('user').get('username'),
                    email=request.data.get('user').get('email'),
                    password=request.data.get('user').get('password')
                )
        except IntegrityError as e:
            return Response({'message': str(e.__cause__)}, status=400)
        player = Player.objects.create(user=user, rating=1000)
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'player': PlayerSerializer(player).data
        })
```

Fig. 6. Code snippet for the implementation of *PlayerList*.

Class *PlayerList* is inheriting *rest_framework.generics.ListCreateAPIView*, which is a class that already implements the basic logic for creating and listing some objects; in our case these objects are players. The class was extended by defining two class attributes: *queryset*, which defines what objects to list, and *serializer_class*, which defines the serializer to use when converting an object to or from JSON. As a *queryset*, all players stored in the database were passed taking into consideration the class *PlayerSerializer*. Also, the method that handles the creation of the player was overwritten in a way that creates *User* object and then creates *Player* object using, but, most importantly, it creates a token for the user that is returned as a response together with the *Player* object serialized by the implemented serializer.

2.2. Matchmaking System and Playing Process Handling

To implement the matchmaking system and playing process handling, the Django Channels library is employed because it allows embedding the web sockets into the Django application. Django Channels use two components to describe incoming connections employing web sockets Scope and Events. The Scope component represents the details of one incoming connection. These details include the information about the request and the client that sends it. The scope stays during the whole connection until the socket is closed. In this application, the scope is instantiated when a player decides to find a match. The Events component represents the desired actions on the client side which can be received throughout the whole connection. In the case of matchmaking, an event that might be received is to stop looking for a match. So, when there is an incoming connection, the scope is created, and until the connection is closed, there can be multiple events coming to the application whenever the client wants to interact.

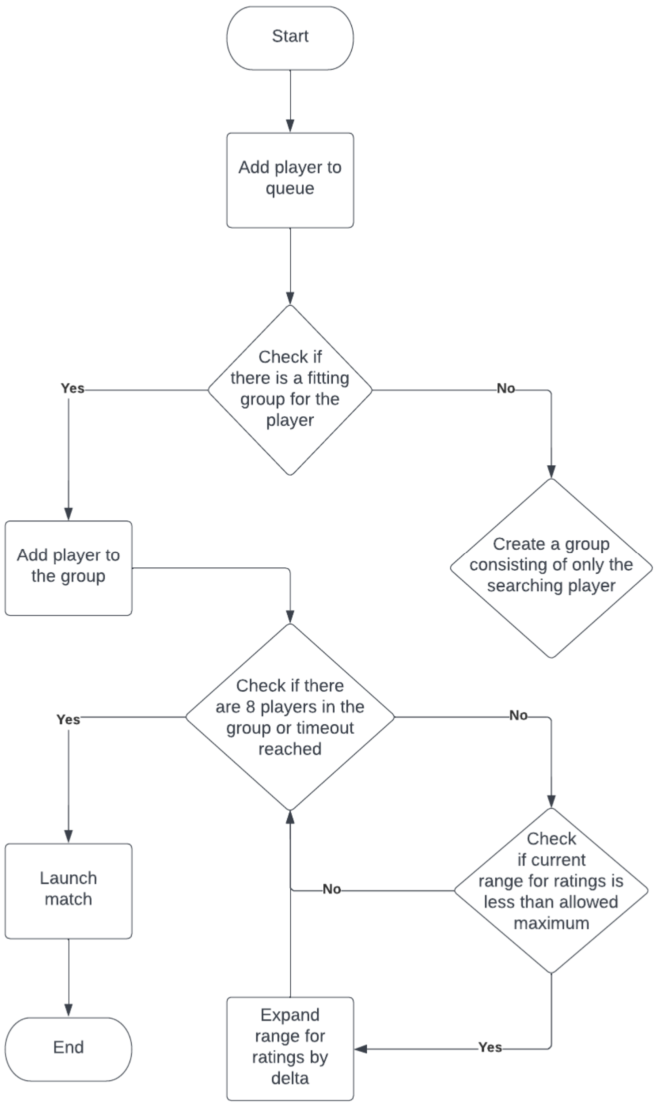
Start
Add player to queue
Check if there is a fitting group for the player
Yes
No
Add player to the group
Create a group consisting of only the searching player
Check if there are 8 players in the group or timeout reached
Yes
No
Launch match
Check if current range for ratings is less than allowed maximum
No
Yes
End
Expand range for ratings by delta

Fig. 7. The flowchart of the matchmaking algorithm.

2.2.1. Designing the algorithm for matchmaking system

To design an algorithm for the matchmaking system, two significant factors are considered: (1) the waiting time that is necessary to find a match, and (2) the difference between players' ratings. These factors affect the level of satisfaction of players because the player does not want to wait too much time to start playing, and every player wants to play against those who have approximately the same level of playing skill since this makes the match intriguing and fair. Fig. 7 shows the flowchart of the algorithm used in this project. The algorithm considers adding players looking for a match to a queue until there is a group of players with the same ratings, so the difference between the players is as small as possible. The main configuration variable is the difference between ratings. Other configuration variables include, the rating difference expansion, time between expansions, the maximum difference between ratings, and a timeout for any group of two matching players looking for a match.

2.2.2. Implementing the algorithm for matchmaking system

The matchmaking system was implemented using the Django Channels. Since web sockets work asynchronously, default Django views cannot be applied here. So, the first stage was to implement Consumers - an abstraction provided by the Django Channels that allows the creation of ASGI applications. Django Channels allow the implementation of regular series of functions. Moreover, they handle threads and handoffs. For any web socket, there are three methods to be implemented: Connect, Receive, and Disconnect. The Connect method is called whenever there is a request to create a connection between the client and server. When there are incoming requests during the connection, the Receive method is called. These requests are represented in Django Channels as events. The Disconnect method is called when there is a request to close the connection.

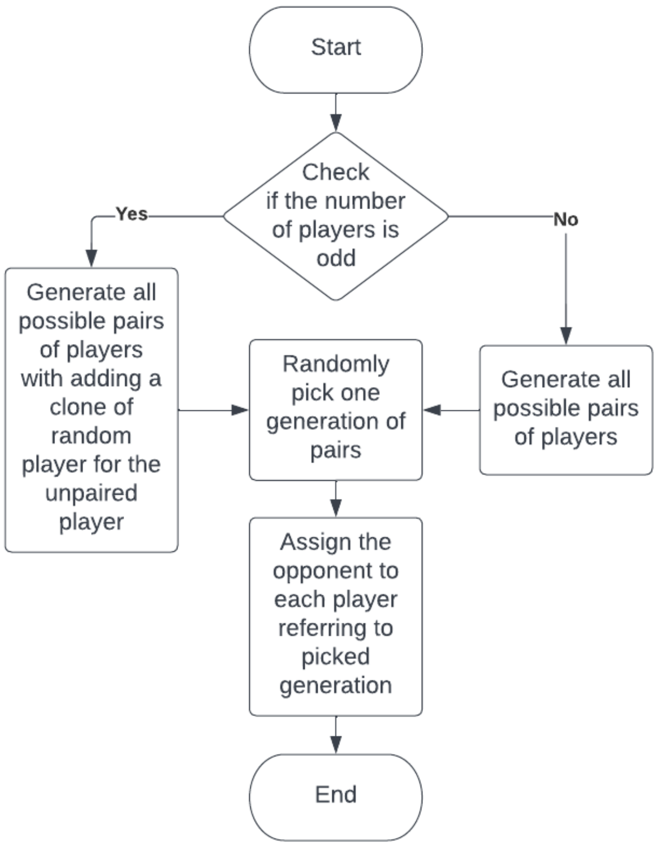
Start
Check if the number of players is odd
Yes
No
Generate all possible pairs of players with adding a clone of random player for the unpaired player
Randomly pick one generation of pairs
Generate all possible pairs of players
Assign the opponent to each player referring to picked generation
End

Fig. 8. The flowchart of the match process handling algorithm.

The process of finding a match is implemented as a task which is called every two seconds, that implements the matchmaking algorithm after adding a player to a queue. To accomplish this task, Django Celery was embedded into the application, which is a simple and flexible system that allows processing tasks in the background [14]. Django Celery provides an extension named Django Celery Beat that allows the creation of scheduled tasks that are run after some time. So, the task for matchmaking was set to run every two seconds.

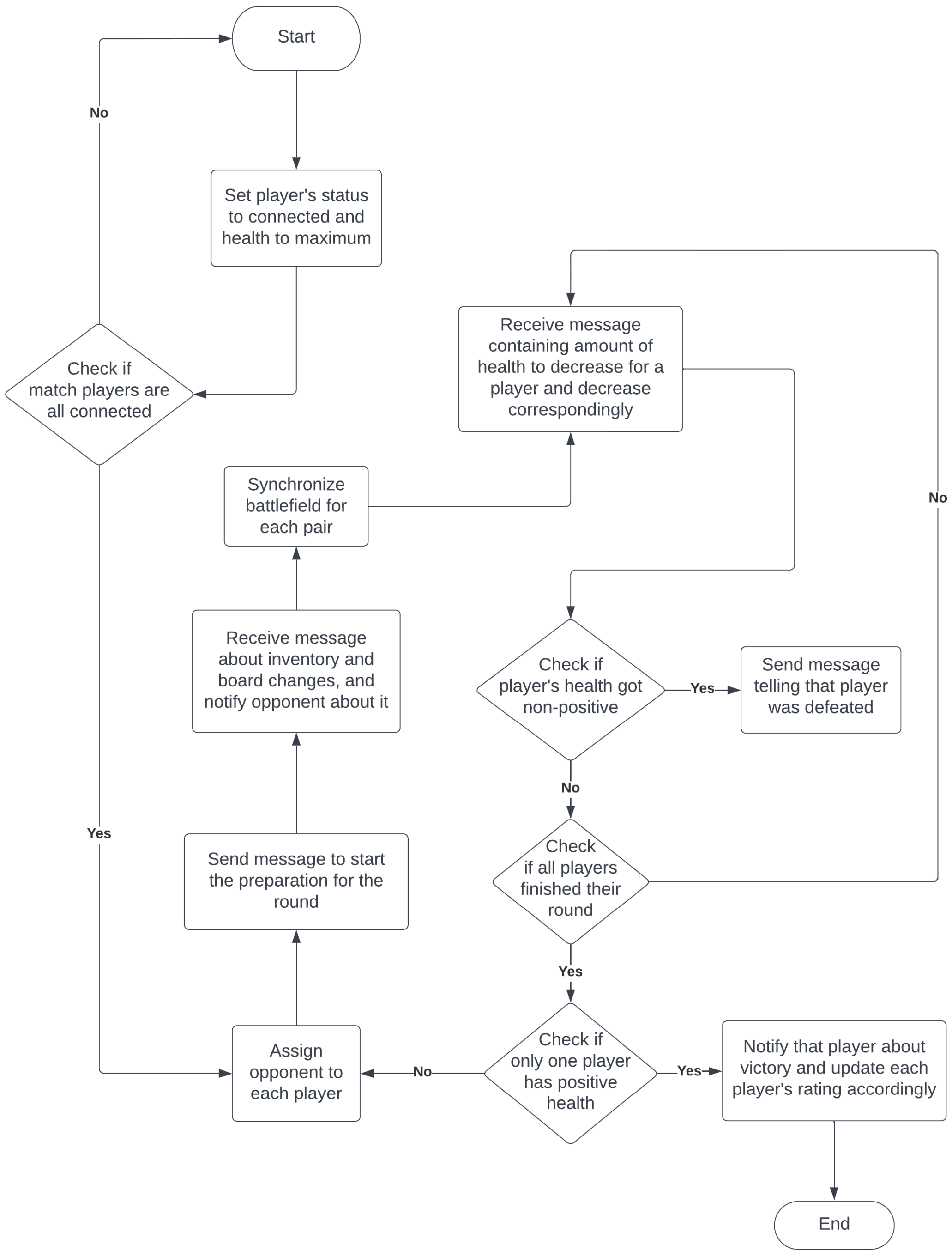
Start
Set player's status to connected and health to maximum
Check if match players are all connected
No
Yes
Assign opponent to each player
Send message to start the preparation for the round
Receive message about inventory and board changes, and notify opponent about it
Synchronize battlefield for each pair
Receive message containing amount of health to decrease for a player and decrease correspondingly
Check if player's health got non-positive
Yes
Send message telling that player was defeated
No
Check if all players finished their round
No
Yes
Check if only one player has positive health
No
Yes
Notify that player about victory and update each player's rating accordingly
End

Fig. 9. The flowchart of the entire process of the match process handling.

2.2.3. Implementing the match process handling

When a player connects to the web socket, the status is set to “connected” and the health bar is set to maximum. Opponents are assigned using an algorithm that generates all possible distribution of players into pairs. If the number of players is odd, then the player without an opponent is assigned to a clone of a randomly picked player except itself. Fig 8 shows the flowchart of the match process handling algorithm.

When all players are connected and have assigned opponents, the round phase starts. It consists of two parts: preparation and battle. During the preparation, players arrange the hero units on the board. Whenever there is an action done by a player, the opponent should know about it. So, for every such action, there is a message sent through the web socket. After the preparation, the second part of the round phase starts. This part is processed only on the client side. Fig. 9 shows the flowchart of the entire process of the match process handling.

2.3. Deployment of the Application on the Server

The application was deployed on the Heroku cloud platform. DRF includes a comfortable ready web server that can be run; however, it is not used for production purposes because it does not handle requests in parallel. Therefore, the Daphne platform was used, which is a Python HTTP and Web Socket protocol server used for ASGI applications. Unlike Django's built-in web server, Daphne provides the concurrent handling of incoming requests by running several Python processes. Heroku server needs Procfile - a set of all commands that run on the startup of the application. The following code specifies Procfile as follows:

```
release: python manage.py migrate

web: daphne -b 0.0.0.0 -p $PORT auto_chess.asgi:application

worker: celery -A auto_chess worker --beat --scheduler django --loglevel=info
```

This allows to apply new database changes before running the server. The application was also configured by adding both PostgreSQL and Redis to the Heroku server as add-ons.

3. Conclusion and Perspectives

In this paper, a new technique for better management of conditions in online auto chess multi-player games is proposed. To implement the matchmaking system and match process handling, the Django Channels library is used together with web sockets. In addition, Django Celery Beat and Celery are used for background processing of the matchmaking system. The application uses PostgreSQL DBMS. The application is deployed in the Heroku cloud platform. Since this application is a web API, DRF is considered the most effective toolkit that was embedded in the main Django application. To implement the matchmaking system and match processing in the game, the Django Channels library is employed because it allows using web sockets technology together with Django. The successful testing of all HTTP and Web Socket endpoints shows the final application is working correctly. A good database schema is a consequence of the easy implementation of the whole application. The designed algorithm for the matchmaking system is working fast enough to find a match with players of the approximately same skill level.

Further possible improvements would be the exploration of more non-crucial additional endpoints and the extension of the application with a website that provides the functionality of registration and a wiki for the game.

Abbreviations

The table lists the abbreviations and acronyms used in the paper:

API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface

DBMS	Database Management System
DRF	Django REST Framework
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
MVC	Model–View–Controller
MVT	Model–View–Template
REST	Representational State Transfer

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

AA and AT conducted the research; AA, AT, and DZ analyzed and validated the design; AA wrote the paper; AA, AT, and DZ revised the paper; all authors had approved the final version.

References

[1] Viana, B. (2020, June 6). What is an autobattler? CNN. Retrieved from: https://dotesports.com/news/what-is-an-autobattler

[2] Goslin, A. (2019). Blizzard announces hearthstone battlegrounds, a new autobattler set in the warcraft universe. *Polygon*.

[3] Grayson, N. (2019). A guide to auto chess, 2019's most popular new game genre. Kotaku.

[4] Goslin, A. (2019). Which auto battler should you play: Teamfight Tactics, Underlords, or dota 2 auto chess. *Polygon*.

[5] Gilliam, R. (2019). Riot games is making its own league of legends auto chess game. *Polygon*.

[6] Gilliam, R. (2019). Auto chess creators bringing stand-alone game to PC later this year. *Polygon*.

[7] Meng, M., Steinhardt, S., & Schubert, A. (2018). Application programming interface documentation: what do software developers want? *Journal of Technical Writing and Communication*, *48*(*3*), 295–330.

[8] IBM Cloud Education. (2020, August 19). *Application Programming Interface (API).* Retrieved from: https://www.ibm.com/cloud/learn/api

[9] Hou, D., & Li, L. (2011). Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroup discussions. *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*.

[10] Liu, Q., & Sun, X. (2012). Research of web real-time communication based on web socket. *International Journal of Communications, Network and System Sciences, 5*(*12*).

[11] Fietkiewicz, M. (2021, July 6). *WebSockets vs. HTTP*. Retrieved from: https://ably.com/topic/websockets-vs-http

[12] ASGI. (n.d.). *ASGI Documentation.* Retrieved from: https://asgi.readthedocs.io/en/latest/

[13] Daphne. (n.d.). *Daphne.* Retrieved from: Retrieved from: https://docs.celeryq.dev/en/stable/.