



Sprawozdanie z projektu

Bazy Danych 2

Autorzy:

Krystian Włodek

Wojciech Przybytek

Igor Sitek

Spis treści

1	Opis projektu	3
1.1	Serwer	3
1.2	Baza danych	6
1.3	Klient	11
2	Przypadki użycia	14
2.1	Rejestracja i logowanie	14
2.2	Pokoje do gry	17
2.3	Gra	18

1 Opis projektu

Stworzonym przez nas projektem jest aplikacja pozwalająca grać użytkownikom w warcaby przez internet. Użytkownik po stworzeniu konta w aplikacji i zalogowaniu może rozgrywać partie z innymi graczami w tworzonych przez nich pokojach. Wśród wszystkich graczy prowadzony jest ranking, który ustalany jest za pomocą liczby punktów zdobywanych w rozgrywkach.

Aplikacja składa się z trzech części: serwera REST API, relacyjnej bazy danych oraz klienta z interfejsem graficznym.

1.1 Serwer

Serwer REST API został napisany w języku Python przy użyciu frameworka Flask. Odpowiada on za wystawianie endpointów, dzięki którym możliwa jest komunikacja z klientami oraz modeluje, tworzy i wykonuje operacje na bazie danych.

Do przeprowadzania operacji bazodanowych wykorzystano bibliotekę SQLAlchemy. Jest to object-relational mapper (ORM) stworzony dla języka Python, który na podstawie klas i ich pól w aplikacji tworzy tabele i ich kolumny w bazie danych i umożliwia do nich dostęp za pomocą poleceń w składni języka.

Każda klasa modelująca tabelę dziedziczy po bibliotecznej klasie Model, a każde jej pole modelujące kolumnę w tabeli jest obiektem klasy Column. Przy tworzeniu takiego obiektu można jako parametry konstruktora wskazać klucz główny, klucz obcy oraz warunki integralnościowe.

```
class User(db.Model):
    __tablename__ = 'users'

    user_id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(40), unique=True, nullable=False)
    password = db.Column(db.String(80), nullable=False)
    password_salt = db.Column(db.String(10), nullable=False)
    rankings = db.relationship('Ranking', backref='users', lazy=True)

    create_time = db.Column(db.DateTime, nullable=False, server_default=func.now())
    update_time = db.Column(db.DateTime, nullable=False, server_default=func.now(), onupdate=func.now())
```

Rysunek 1: Przykład klasy modelującej w kodzie serwer - User

```
create table warsaby.users
(
    user_id      int auto_increment
        primary key,
    username      varchar(40)                not null,
    password      varchar(80)                not null,
    password_salt varchar(10)                not null,
    create_time    datetime default CURRENT_TIMESTAMP not null,
    update_time    datetime default CURRENT_TIMESTAMP not null,
    constraint username
        unique (username)
);
```

Rysunek 2: Kod SQL wygenerowany na podstawie klasy modelowej User

W celu komunikacji z klientami serwer wystawia endpointy za pomocą REST API. Każdy z klientów requesty na adres serwera z danymi, na przykład loginu i hasła lub aktualnie wykonywanego ruchu. Jako odpowiedź serwer wysyła odpowiedź na zapytanie wraz z kodem.

Endpointy wystawione przez serwer to:

- **/sign-up**

Wysyłany przez klienta w trakcie rejestracji. Serwer sprawdza, czy wszystkie potrzebne dane są wprowadzone oraz czy nie istnieje już konto o takich danych. Zwraca informację czy konto udało się stworzyć.

- **/account-data**

Zwraca informacje na temat konta aktualnie zalogowanego na kliencie, takie jak nazwa użytkownika, miejsce w rankingu i ilość punktów.

- **/rankings**

Przesyła klientowi pełną listę rankingową z nazwami użytkowników i ilością punktów.

- **/sign-in**

Wysyłany przez klienta podczas logowania, sprawdza poprawność danych oraz czy użytkownik o takich danych istnieje w bazie. W przypadku poprawnej autentykacji dodaje sesję użytkownika do bazy.

- **/is-authenticated**

Sprawdza czy sesja użytkownika jest aktywna.

- **/sign-out**

Wylogowuje użytkownika poprzez zakończenie jego aktualnej sesji w bazie danych.

- **/rooms**

Zwraca klientowi listę aktualnych pokoi gry wraz z informacją, czy znajdują się w niej inni gracze.

- **/create-room**

Tworzy nowy pokój gry w bazie danych.

- **/show-room**

Zwraca szczegółowe informacje o wybranym pokoju i grze, która się w nim rozgrywa, takie jak na przykład pozycja pionków.

- **/manage-places**

Pozwala klientowi zająć miejsce w pokoju, jeżeli jest wolne lub zwolnić miejsce które wcześniej zajął.

- **/start-game**

Umożliwia klientowi zgłosić chęć do gry przy zajętym miejscu w pokoju. Gdy dwóch różnych klientów zgłosi chęć do gry tworzy nową rozgrywkę w bazie danych pomiędzy graczami w pokoju.

- **/move**

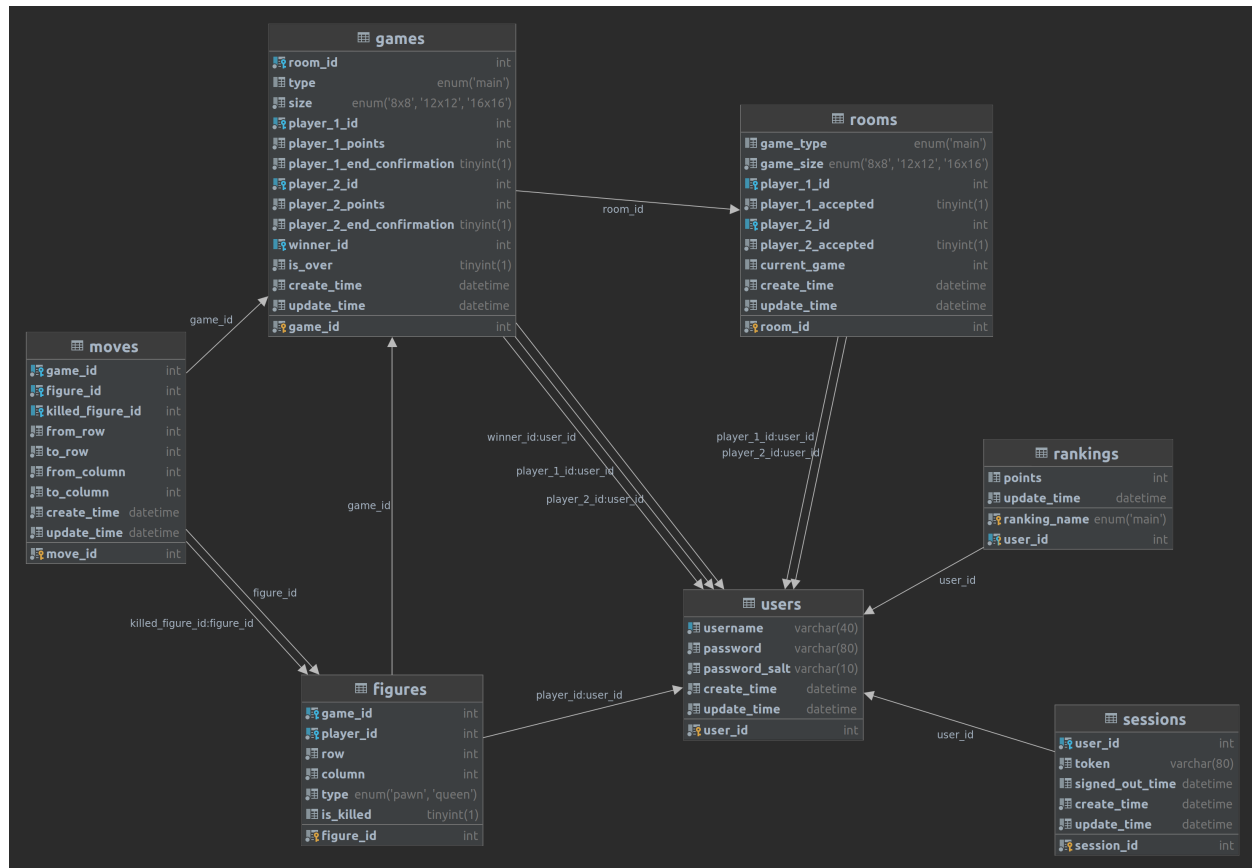
Klient wysyła informacje o swoim ruchu do serwera, a ten sprawdza czy jest to ruch poprawny i może zostać wykonany. Jeżeli wszystkie warunki są spełnione, to serwer dokumentuje ruch w bazie danych i zmienia aktywnego gracza, a w przypadku zakończenia rozgrywki kończy grę i odnotowuje zwycięzcę.

- **/end-game**

Po zakończeniu rozgrywki serwer zapisuje do bazy informacje o zwycięzcy, zmienia liczbę punktów graczom oraz zwalnia zajęty przez grę pokój.

1.2 Baza danych

Relacyjna baza danych została stworzona w technologii MySQL. Każda jej tabela i relacja jest modelowana na podstawie klas w aplikacji serwera.



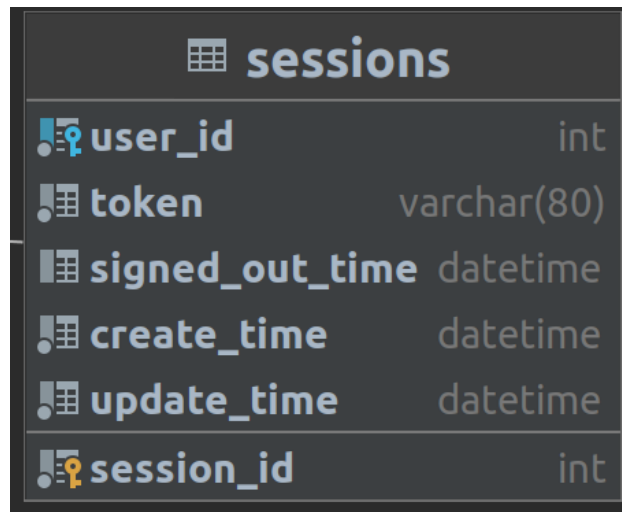
Rysunek 3: Schemat bazy danych







users	
username	varchar(40)
password	varchar(80)
password_salt	varchar(10)
create_time	datetime
update_time	datetime
user_id	int

Rysunek 4: Tabela users - zawiera dane użytkowników wraz z czasem utworzenia i edycji konta.

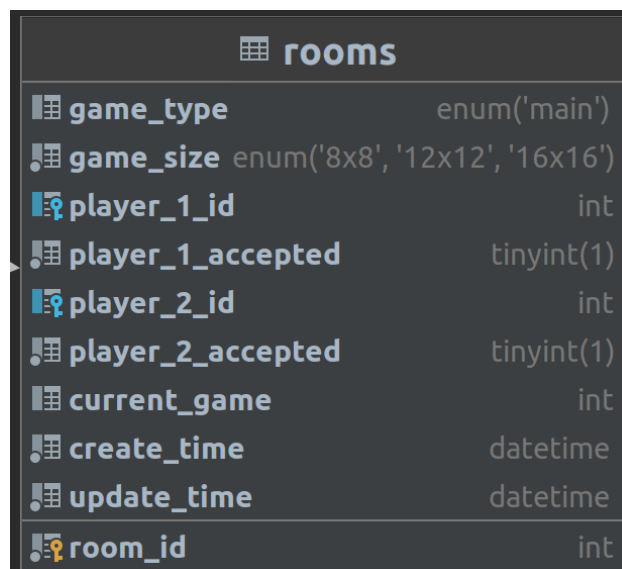
rankings	
points	int
update_time	datetime
ranking_name	enum('main')
user_id	int











Rysunek 5: Tabela rankings - zawiera informacje o liczbie punktów każdego gracza.



 user_id	int
 token	varchar(80)
 signed_out_time	datetime
 create_time	datetime
 update_time	datetime
 session_id	int

Rysunek 6: Tabela sessions - zawiera historię wszystkich sesji aktualnych i archiwalnych.

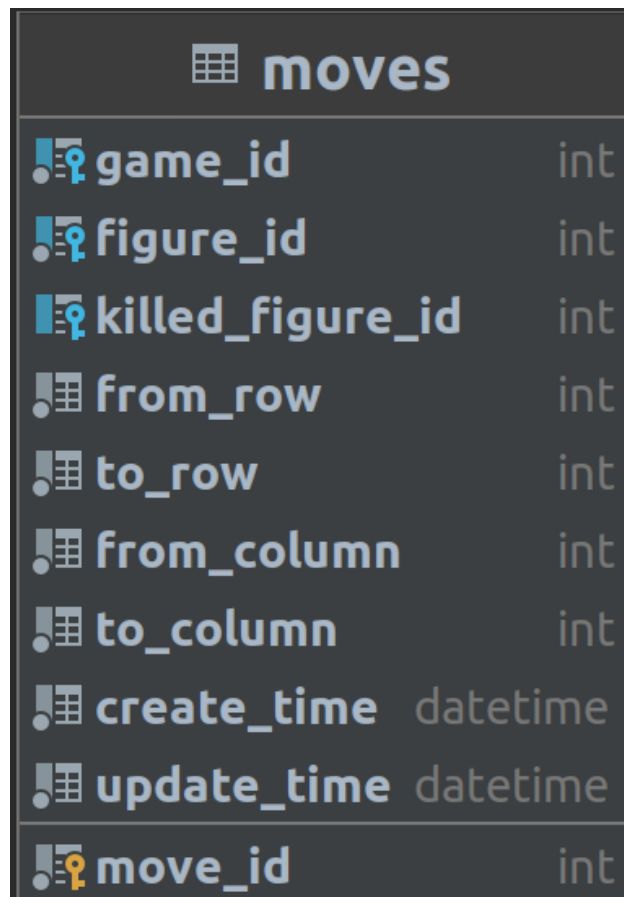


 game_type	enum('main')
 game_size	enum('8x8', '12x12', '16x16')
 player_1_id	int
 player_1_accepted	tinyint(1)
 player_2_id	int
 player_2_accepted	tinyint(1)
 current_game	int
 create_time	datetime
 update_time	datetime
 room_id	int

Rysunek 7: Tabela rooms - zawiera informacje o aktualnym stanie danego pokoju i rozgrywanej w nim grze. Po zakończeniu gry wszystkie dane w wierszu są resetowane.

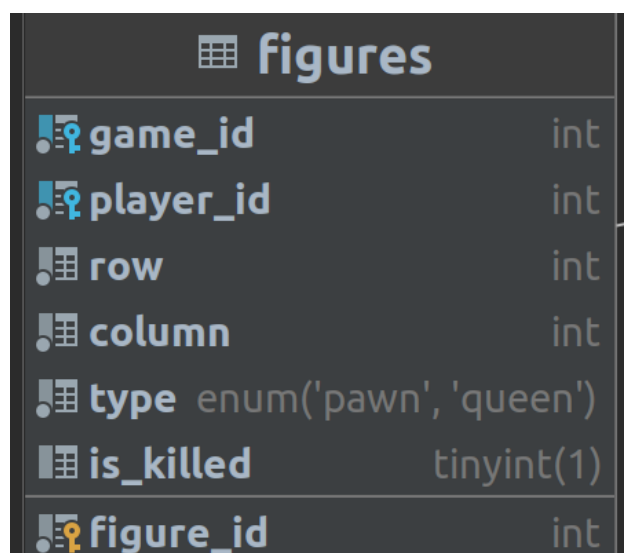
games	
room_id	int
type	enum('main')
size	enum('8x8', '12x12', '16x16')
player_1_id	int
player_1_points	int
player_1_end_confirmation	tinyint(1)
player_2_id	int
player_2_points	int
player_2_end_confirmation	tinyint(1)
winner_id	int
is_over	tinyint(1)
create_time	datetime
update_time	datetime
game_id	int

Rysunek 8: Tabela games - zawiera informacje o wszystkich rozgrywkach aktualnych oraz archiwalnych



game_id	int
figure_id	int
killed_figure_id	int
from_row	int
to_row	int
from_column	int
to_column	int
create_time	datetime
update_time	datetime
move_id	int

Rysunek 9: Tabela moves - zawiera historię wszystkich wykonanych ruchów w grach.



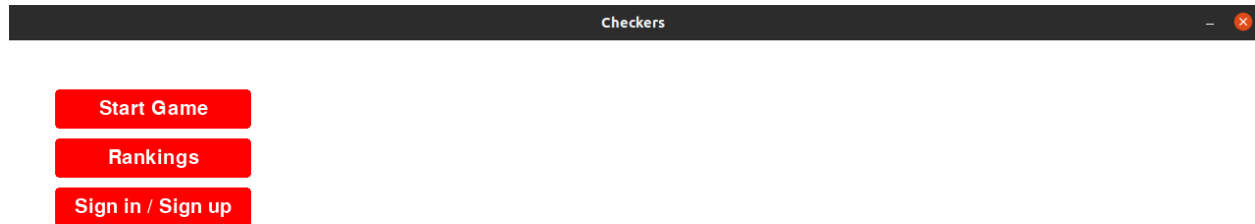
game_id	int
player_id	int
row	int
column	int
type	enum('pawn', 'queen')
is_killed	tinyint(1)
figure_id	int

Rysunek 10: Tabela figures - zawiera informacje o wszystkich pionkach w grach aktualnych oraz archiwalnych.

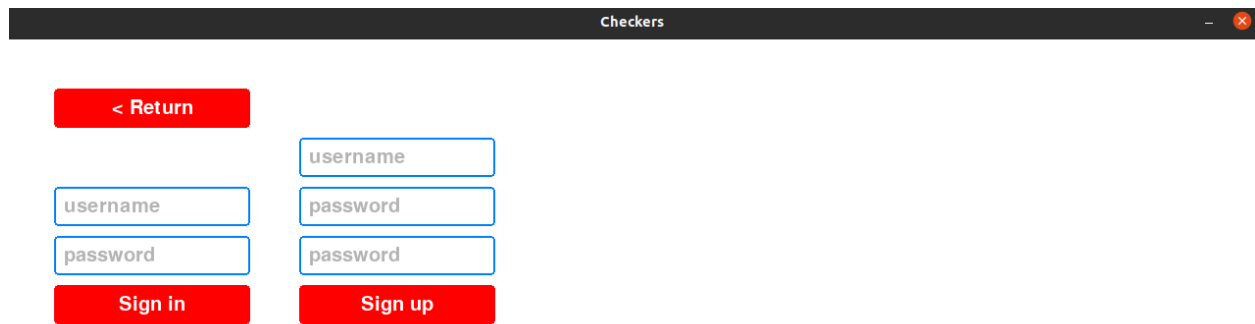
1.3 Klient

Aplikacja kliencka została napisana w języku Python, a do stworzenia GUI wykorzystano bibliotekę pygame. Interfejs użytkownika składa się z różnych widoków i poprzez interakcje z nimi użytkownik decyduje o działaniu aplikacji. Klient komunikuje się z serwerem za pomocą requestów HTTP i w ten sposób otrzymuje informacje o pozycji w rankingu czy aktualnym stanie gry.

Widoki w aplikacji



Rysunek 11: Menu główne



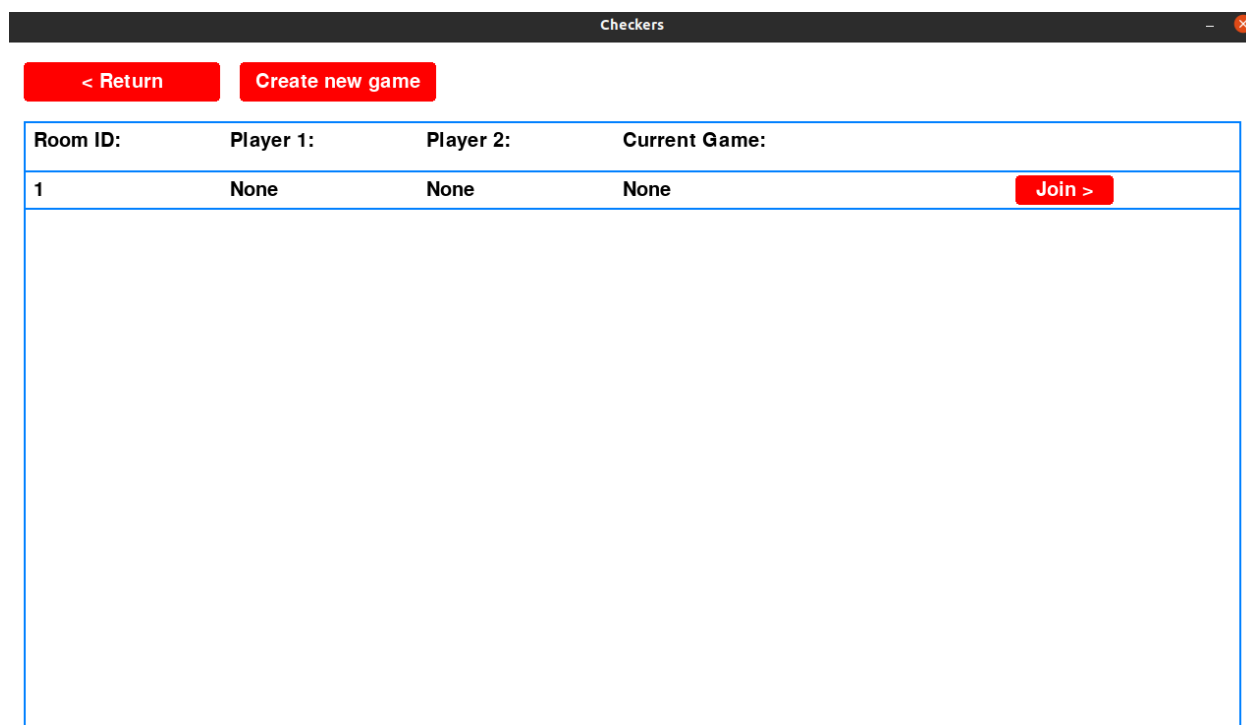
A screenshot of a web application window titled "Checkers". The window has a dark header bar with the title and standard window controls. The main content area is white and contains a login form. On the left, there is a red button labeled "< Return". To its right, there are two columns of input fields. The first column has a "username" field and a "password" field. The second column has a "username" field and a "password" field. Below the first column is a red "Sign in" button, and below the second column is a red "Sign up" button.

Rysunek 12: Ekran logowania

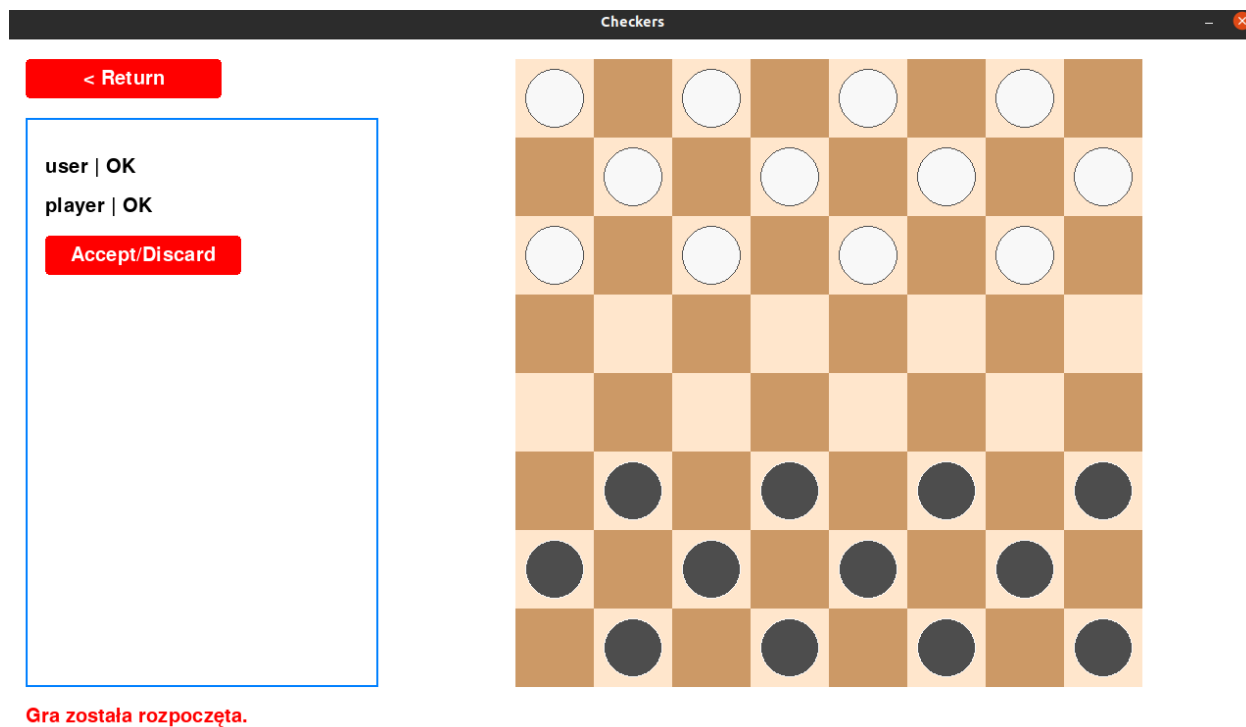


A screenshot of a web application window titled "Checkers". The window has a dark header bar with the title and standard window controls. The main content area is white. On the left, there is a red button labeled "< Return". To its right, the word "RANKING" is displayed in bold. Below the "RANKING" text, the text "Player: user | Points: 1000" is displayed.

Rysunek 13: Ranking użytkowników



Rysunek 14: Lista pokojów gier



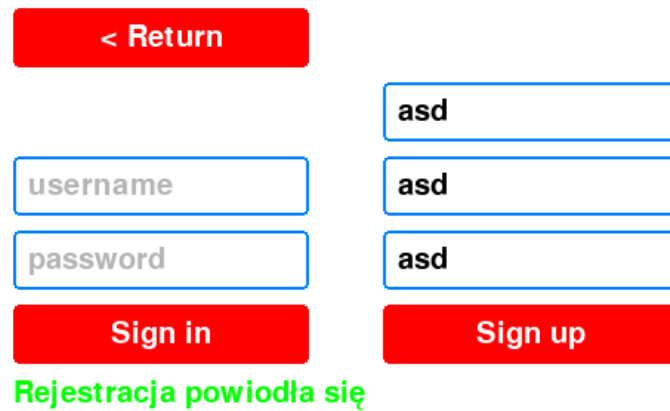
Rysunek 15: Ekran gry

2 Przypadki użycia

2.1 Rejestracja i logowanie

Klient

Klient otwiera okno aplikacji i wybiera opcję logowania i rejestracji, następnie wprowadzając swoje dane (do rejestracji):



The image shows a user interface for registration and login. It features a red button labeled "< Return" at the top left. Below it, there are two columns of input fields. The left column has two fields labeled "username" and "password". The right column has three fields, each containing the text "asd". Below the input fields, there are two red buttons: "Sign in" on the left and "Sign up" on the right. At the bottom, there is a green text message that reads "Rejestracja powiodła się".

Rysunek 16: Widok rejestracji i logowania w aplikacji.

Przy kliknięciu wykonywany jest kod end-pointu *account controller*:

```
def sign_up():
    if request.method == 'POST':
        json_data = request.get_json()
        flag, data = account_validator.sign_up(json_data)

        if not flag:
            return {'is_authenticated': False, 'message': data}, 400

        data['password_salt'] = ''.join(random.choice(utils.characters) for i in range(10))
        data['password'] = utils.hash_password(data['password'], data['password_salt'])

        user = User(**data)
        db.session.add(user)
        db.session.commit()

        main_ranking = {'user_id': user.user_id}

        main_ranking = Ranking(**main_ranking)
        db.session.add(main_ranking)
        db.session.commit()

        return {'is_authenticated': False, 'message': 'Konto zostało utworzone pomyślnie. '}, 200
    else:
        return {'message': 'Bad request. '}, 400
```

Rysunek 17: **End point** rejestracji użytkownika - plik *account-controller.py*

Następnie klikając opcję *sign in*, wykonuje się poniższy kod modułu *session-controller.py*:

```
def sign_in():
    if request.method == 'POST':
        json_data = request.get_json()
        user = db.session.query(User).filter_by(username=json_data['username']).first()

        if user is None or user.password != utils.hash_password(json_data['password'], user.password_salt):
            return {'is_authenticated': False, 'message': 'Nazwa użytkownika lub hasło jest nieprawidłowe. '}, 400

        session = {'user_id': user.user_id,
                    'token': ''.join(random.choice(utils.characters) for i in range(80))}
        session = Session(**session)
        db.session.add(session)
        db.session.commit()

        return {'is_authenticated': True, 'message': 'Zostałeś zalogowany pomyślnie. ',
                'data': {'session_id': session.session_id,
                          'token': session.token,
                          'user_id': session.user_id}}, 201
    else:
        return {'message': 'Bad request. '}, 400
```

Rysunek 18: **End point** logowania - plik *session-controller.py*

Poniżej przedstawiono także szczegóły zapytań wysyłanych do bazy danych oraz zwrotne informacje po ich wysłaniu.

```
## SIGN UP
POST http://{{host}}/sign-up HTTP/1.1
Content-Type: application/json

{
  "username": {{username}},
  "password": {{password}},
  "confirm_password": {{password}}
}

### SIGN IN
# @name sign_in
POST http://{{host}}/sign-in HTTP/1.1
Content-Type: application/json

{
  "username": {{username}},
  "password": {{password}}
}

###

@session_id = {{sign_in.response.body.$.data.session_id}}
@token = {{sign_in.response.body.$.data.$.token}}
@user_id = {{sign_in.response.body.$.data.$.user_id}}
```

Rysunek 19: Zapytania autentykujące do bazy danych.

Baza danych

Jak widzimy, w bazie danych dodano nowego użytkownika:

```
> select * from users
```

	user_id	username	password	password_salt	create_time	update_time
1	1	q	78e83bd4f1ab39aa87d7c2a2167b79cfea22c...	sUjby46063	2022-06-03 13:14:06	2022-06-03 13:14:06
2	2	g	7e30038ffdc030414913d7dfe0db8540992ea...	pL0Rwwh6Vx	2022-06-03 13:14:41	2022-06-03 13:14:41
3	3	asd	2f34f8a84f74417b68cf74a63e022dd97df38...	67ph102Hx0	2022-06-03 14:49:54	2022-06-03 14:49:54

Rysunek 20: Stan bazy danych - tabela Users.

2.2 Pokoje do gry

Klient

Gracz, chcąc zagrać partię warcab, ma dwie opcje: stworzyć nowy pokój do gry lub dołączyć do istniejącego.

< Return Create new game All warnings will be displayed here.

Room ID:	Player 1:	Player 2:	Current Game:
1	q	g	1
Join >			

Rysunek 21: Widok tworzenia i dołączania do pokoi w aplikacji.

Tworzenie nowego pokoju

Poniższy kod jest wywoływany przy tworzeniu nowego pokoju:

```
def create_room():
    if request.method == 'POST':
        json_data = request.get_json()
        response, status_code = utils.is_authenticated(json_data, False)
        if status_code != 200:
            return response, status_code
        rooms = db.session.query(Room).all()

        if len(rooms) >= max_rooms_amount:
            response['message'] = 'Nie można utworzyć nowego pokoju, znajdź jakiś wolny. '
            status_code = 400
            return response, status_code

        room = Room(**{})
        db.session.add(room)
        db.session.commit()

        response['message'] = 'Pokój został utworzony. '
        response['data'] = {}
        response['data']['room_id'] = room.room_id
        status_code = 201
        return response, status_code
    else:
        return {'message': 'Bad request. '}, 400
```

Rysunek 22: Kod tworzenia nowego pokoju - plik *room-controller.py*

Pokażmy również zapytanie do bazy danych, tworzące nowy pokój:

```
### CREATE ROOM
POST http://{{host}}/create-room HTTP/1.1
Content-Type: application/json

{
  "session_id": "{{session_id}}",
  "token": "{{token}}",
  "user_id": {{user_id}}
}
```

Rysunek 23: Zapytanie tworzące w bazie nowy pokój.

Miała miejsce teraz poniższa sytuacja:

1. Gracz q (id: 1) stworzył dwa nowe pokoje.
2. Gracz q wszedł do drugiego pokoju, zatwierdził pokój oraz chęć do gry.
3. Gracz g (id: 2) wszedł do pokoju o id 2 (do którego wszedł także gracz q), zatwierdził pokój oraz chęć do gry.

Baza danych

Po poniższej sytuacji stan bazy danych wygląda następująco:

```
> select * from rooms
```

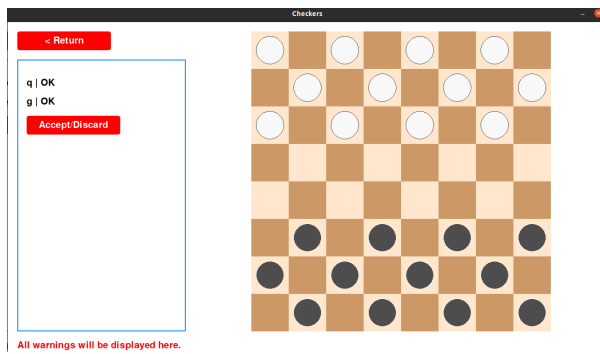
room_id	game_type	game_size	player_1_id	player_1_accepted	player_2_id	player_2_accepted	current_game	create_time	update_time
1	Main	8x8	NULL	0	NULL	0	NULL	2022-06-03 15:23:52	2022-06-03 15:23:52
2	Main	8x8	1	1	2	1	1	2022-06-03 15:23:58	2022-06-03 15:24:17

Rysunek 24: Stan bazy danych po wykonanej serii operacji - tabela **Rooms**.

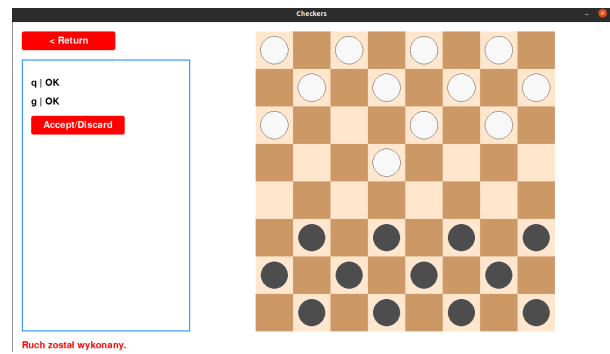
2.3 Gra

Gdy dwójka graczy w pokoju potwierdzi grę, ta się rozpoczyna. Rozpatrzmy teraz, jak zachowuje się aplikacja kliencka, klient podpięty do bazy oraz sama baza w czasie, gdy gracze wykonują kolejne ruchy pionkami.

Aplikacja kliencka



Rysunek 25: Aplikacja po rozpoczęciu gry.



Rysunek 26: Aplikacja po jednym ruchu.

Klient

Gracz, który ma swój ruch i wykona go poprawnie (dobrym pionkiem na odpowiednie miejsce), powoduje wywołanie poniższego kodu:

```
def move():
    if request.method == 'POST':
        json_data = request.get_json()
        response, status_code = utils.is_authenticated(json_data, False)

        if status_code != 200:
            return response, status_code

        figure = db.session.query(Figure).filter_by(figure_id=json_data['figure_id']).first()
        game = db.session.query(Game).filter_by(game_id=figure.game_id).first()

        def validate(row, column):...

        def make_move(message, row, column):...

        can_move, message = validate(json_data['row'], json_data['column'])
        if can_move:
            make_move(message, json_data['row'], json_data['column'])
            response['message'] = 'Ruch został wykonany. '
            status_code = 201
            return response, status_code
        else:
            response['message'] = message
            status_code = 400
            return response, status_code
    else:
        return {'message': 'Bad request. '}, 400
```

Rysunek 27: Ruch pionkiem - plik *game-controller.py*

Baza danych

Wykonano jeden ruch, ruch należał do gracza q ($id = 1$), zatem tylko tę część tabeli **Figures** ma sens pokazywać w kontekście ruchu pionka.

```
> select * from figures where player_id = 1
```

figure_id	game_id	player_id	row	column	type	is_killed
1	1	1	0	0	Pawn	0
2	1	1	0	2	Pawn	0
3	1	1	0	4	Pawn	0
4	1	1	0	6	Pawn	0
5	1	1	1	1	Pawn	0
6	1	1	1	3	Pawn	0
7	1	1	1	5	Pawn	0
8	1	1	1	7	Pawn	0
9	1	1	2	0	Pawn	0
10	1	1	2	2	Pawn	0
11	1	1	2	4	Pawn	0
12	1	1	2	6	Pawn	0

Rysunek 28: Baza po rozpoczęciu gry.

figure_id	game_id	player_id	row	column	type	is_killed
1	1	1	0	0	Pawn	0
2	1	1	0	2	Pawn	0
3	1	1	0	4	Pawn	0
4	1	1	0	6	Pawn	0
5	1	1	1	1	Pawn	0
6	1	1	1	3	Pawn	0
7	1	1	1	5	Pawn	0
8	1	1	1	7	Pawn	0
9	1	1	2	0	Pawn	0
10	1	1	3	3	Pawn	0
11	1	1	2	4	Pawn	0
12	1	1	2	6	Pawn	0

Rysunek 29: Baza po jednym ruchu.

Jak widzimy, ruch został poprawnie zapisany w bazie danych.