

1. Tematyka projektu

W ramach projektu zrealizowany został prosty serwis szachowy w formie aplikacji webowej. Wybrane funkcje: utworzenie konta, dodawanie znajomych, gra ze znajomymi, przeglądanie historii gier i historii poszczególnych partii, dostęp do podstawowych statystyk konta, dostęp do rankingu ogólnego.

Realizacja projektu:

- FRONTEND: REACT (:3000)
- BACKEND: NODE.js + Express (:3002)
- DB: MongoDB (ATLAS)

2. Prezentacja wybranych funkcjonalności

2.1 Konfiguracja połączenia z bazą danych

```
const MongoClient = require('mongodb').MongoClient;

class DBConnection {
  static mongoClient;
  static dbName = "Chess";
  static isInitialized() {
    return this.mongoClient !== undefined;
  }
  static getClient(){
    if (this.isInitialized()) return this.mongoClient;
    const mongoUrl = "mongodb+srv://admin:admin@cluster.an1bw.mongodb.net/?retryWrites=true&w=majority";
    this.mongoClient = new MongoClient(mongoUrl);
    return this.mongoClient
  }
  static connect(collectionName,client){
    return new Promise ((async resolve => {
      await client.connect();
      console.log("Connected correctly to DB");
      let db = client.db(DBConnection.dbName);
      let col = db.collection(collectionName);
      resolve( value: {database:db,collection:col})
    })))
  }
}

module.exports = DBConnection
```

Grafika 1: Konfiguracja połączenia z bazą danych

Singleton, który umożliwia wykonywanie operacji na zdefiniowanej bazie danych.

2.2 Konfiguracja serwera

W pliku user.js zdefiniowane zostały ścieżki do poszczególnych requestów:

```
router.use(cors({
  origin: 'http://localhost:3000',
}))

router.get( path: '/user/:username/friends', userController.getFriends);
router.get( path: '/user/:id', userController.getUser);
router.get( path: '/users', userController.getAllUsers);
router.get( path: '/userStats', userController.getWins);
router.get( path: '/user/:username/games', userController.getGames);
router.post( path: '/user/add', userController.addFriend);
router.post( path: '/user/addGame', userController.addGame);
router.post( path: '/user/register', userController.addUser);
router.post( path: '/user/login', userController.login);
router.post( path: '/user/remove', userController.removeFriend);
```

Grafika 2: Definicja routingu

Po wysłaniu zapytania uruchamiana jest odpowiednia funkcja userControllera (funkcje zdefiniowane w pliku userController.js). Następnie wybrana funkcja uruchamia wybrane operacje na bazie danych zdefiniowane w plikach GameService.js oraz UserService.js.

2.2.1 Przykładowy request

```
// 20220912132723
// http://localhost:3002/users

{
  "status": "ok",
  "data": [
    {
      "_id": "62c6e9d367bd672bbb04f92c",
      "username": "user1",
      "password": "ed7c54a144574beaae02a1feb7d877c93e5c43598002fcf1d4132a883c170f1",
      "friends": [
        "user2",
        "user3",
        "user4"
      ],
      "games": [
        "user1-user2-1657203806725",
        "user2-user1-1657204074041",
        "user3-user1-1657205187299",
        "user2-user1-1657540761896",
        "user1-user2-1657541956057",
        "user1-user2-1657545016260",
        "user2-user1-1657990238805",
        "user1-user3-1658315863384",
        "user3-user1-1658399878620",
        "user1-user3-1658400627047",
        "user3-user1-1658401313759",
        "user1-user3-1658403088146",
        "user1-user2-1662621169332",
        "user1-user2-1662621172669",
        "user1-user2-1662621182453",
        "user1-user2-1662621195123"
      ]
    },
    {
      "_id": "62c6e9e167bd672bbb04f92d",
      "username": "user2",
      "password": "edc003bd97e95fc03f065a84501a93acc7391c5d03bdd2701bf055e358d0c0bf",
      "friends": [
        "user1",
        "user3",

```

Grafika 3: Wynik przykładowego zapytania

Po wysłaniu zapytania: `http://localhost:3002/users` uruchamiana jest funkcja `getAllUsers` (patrz Grafika 2).

```
const getAllUsers = async ( req, res ) => {  
  const users = await UserService.getAllUsers();  
  res.send(ResponseHelper( status: 'ok', users))  
}
```

Grafika 4: Funkcja `userController.getAllUsers`

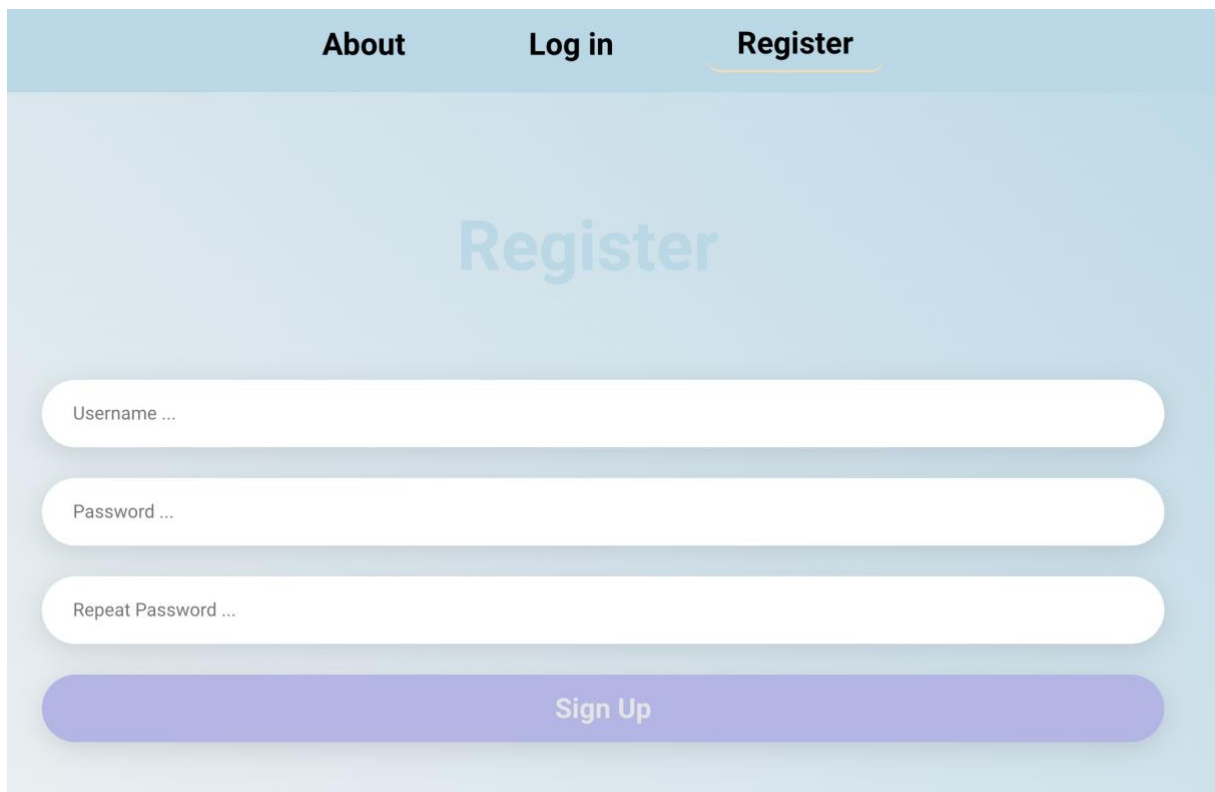
```
class UserService {  
  
  static getAllUsers = async () => {  
    const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient());  
    return connection.collection.find().toArray()  
  }  
}
```

Grafika 5: `UserService.getAllUsers`

Po połączeniu z bazą danych zwracani są wszyscy użytkownicy z kolekcji 'Users' (Grafika 3).

2.3 Utworzenie konta

2.3.1 Client



Grafika 6: Ekran rejestracji

Po wprowadzeniu danych i kliknięciu 'Sing up' na serwer wysyłany jest request:

```
onSubmit={ async (values, {setSubmitting})=> {  
  console.log(values);  
  const res = await fetch( input: 'http://localhost:3002/user/register', init: {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json'},  
    body: JSON.stringify( value: { username: values.username, password: values.password } )  
  });  
  const data = await res.json();  
  notify(data.message);  
  setTimeout( handler: ()=>{  
    if(data.message === 'Registered')  
      navigate('/login')  
  }, timeout: 3000);  
} }  
}
```

Grafika 7: Obsługa rejestracji po stronie klienta

W parametrach przekazywane są wartości czytane z formularza, w przypadku poprawnej rejestracji użytkownik zostanie przekierowany do strony z logowaniem.

2.3.2 Server

W przypadku rejestracji uruchamiana jest funkcja `UserController.addUser` (Grafika 2).

```
const addUser = async ( req,res ) => {  
  const user = new User(req.body.username, req.body.password);  
  const exists = await UserService.exists( user.username );  
  if ( exists ) res.json( {"message": "User Exists"} );  
  else{  
    await UserService.createUser(user)  
    //res.send(ResponseHelper('ok', {"message":"Registered"}));  
    res.json({"message":"Registered"});  
  }  
}
```

Grafika 8: Obsługa rejestracji po stronie serwera

```
static exists = async (nick) => {  
  const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient());  
  const user = await connection.collection.findOne(  
    { username: nick }  
  )  
  return user !== null && user !== undefined;  
}
```

Grafika 9: `UserService.exists`

```

static createUser = async (user) => {
  console.log(user);
  const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient());
  connection.collection.insertOne({username:user.username, password: user.password,
    friends:user.friends, games: user.games}, (err, result) => {
    console.log(result)
    return result
  });
}

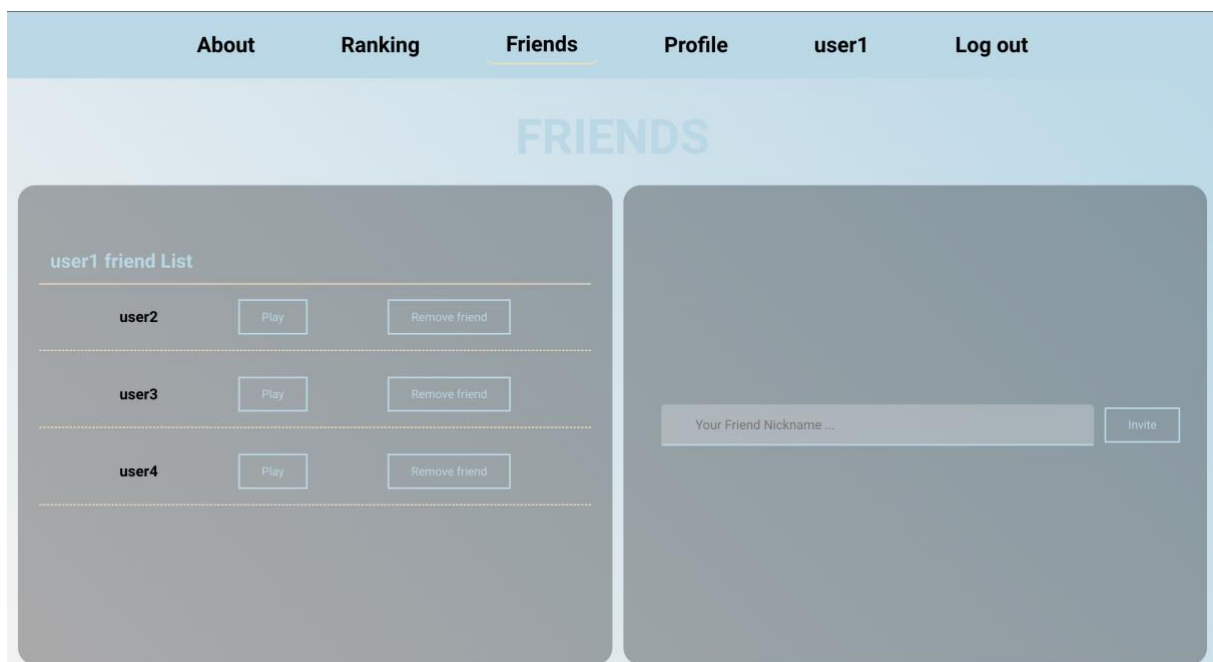
```

Grafika 10: UserService.createUser

Po sprawdzeniu, czy w bazie nie ma użytkownika o podanych danych (UserService.exists), w UserService.createUser do kolekcji 'Users' wstawiany jest nowy użytkownik oraz zwracana jest informacja o poprawnej rejestracji.

2.4 Dodawanie znajomego

2.4.1 Client



Grafika 11: Ekran Friends

Po wprowadzeniu nazwy przyjaciela i kliknięciu 'Invite' na serwer wysyłany jest request:

```
const addFriend = async ()=>{
  let u = user !== null ? user.username : '0';
  const res = await fetch( input: 'http://localhost:3002/user/add', init: {
    method: 'POST',
    headers: { 'Content-Type': 'application/json'},
    body: JSON.stringify( value: {user:u, friend: formField.current?.value})
  })
  const data = await res.json();
  console.log(data.message);
  await updateFriends(u);
}
```

Grafika 12: Obsługa zapytania po stronie klienta

W parametrach przekazywany jest użytkownik oraz znajomy, którego dotyczy zapytanie.

2.4.2 Server

Uruchamiana jest funkcja `userController.addFriend` (Grafika 2):

```
const addFriend = async (req,res) => {
  const data = req.body;
  console.log(data);
  const exists = await UserService.doesExist( data.friend );
  if(data.user === data.friend){
    res.send({message:'Cannot invite yourself.'});
  }
  else if ( exists ){
    await UserService.addFriend(data.user, data.friend);
    await UserService.addFriend(data.friend, data.user);
    res.send({message:'Added.'});
  }
  else{
    res.send({message:'User not found.'});
  }
}
```

Grafika 13: Obsługa zapytania po stronie serwera

Następnie uruchamiane są funkcje `UserService.exists` (Grafika 9) oraz `UserService.addFriend`:

```
static addFriend = async (user, friend) =>{
  const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient());
  connection.collection.updateOne(
    { username: user},
    { $push: { friends: friend } },
  )
}
```

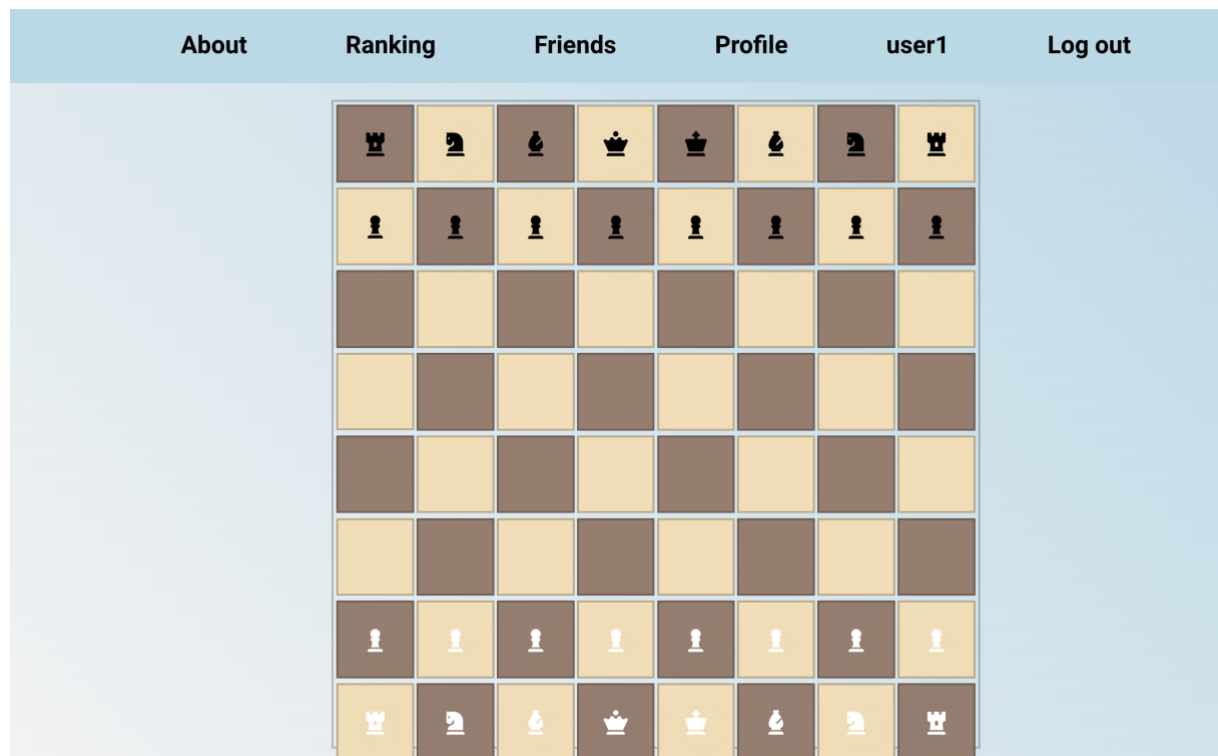
Grafika 14: `UserService.addFriend`

Po wcześniejszym sprawdzeniu warunków, do pola `friends` użytkownika dodawany jest `friend` przekazany w zapytaniu.

2.5 Tworzenie i obsługa partii

2.5.1 Client – tworzenie

Po kliknięciu 'Play' przy wybranym znajomym oraz akceptacji zaproszenia obaj użytkownicy zostaną przekierowani do ekranu rozgrywki (Grafika 18).



Grafika 15: Ekran rozgrywki


```

<button onClick={ ()=> {
  let gameId = `${ user?.username ?? 'user' }-${ friend.username }-${ Date.now() }`;
  // @ts-ignore

  socket?.emit('create_game', {white: user?.username ?? 'user',
    black: friend.username,
    id: gameId} );

  socket?.emit('request_game', {user:friend.username,
    game:gameId});

```

Grafika 16: Obsługa po stronie klienta z wykorzystaniem web socketu - strona wysyłająca

```

socket?.off('request_game').on('request_game', (data)=>{
  console.log(data.user);
  const invitingPerson = data.game.split('-')[0];
  if(user?.username === data.user){
    if(window.confirm(`Accept invitation from ${invitingPerson}.`)){
      socket?.emit('start_game', {user1:invitingPerson, user2:user?.username, game:dat
      socket?.emit('add_to_game', {user:user?.username, game:data.game});
      setCurrentGame(data.game);
      setTimeout( handler: ()=>{
        navigate('/game');
      }, timeout: 2000);
      console.log(currentGame);
    }
    else{
      socket?.emit('cancel_game', data.game);
      console.log('Rejected. ');
    }
  }
}

```

Grafika 17: Obsługa po stronie klienta z wykorzystaniem web socketu - strona odbierająca

```

socket?.on('start_game', (data)=>{
  let split = data.game.split('-');
  if(user?.username === split[0] || user?.username === split[1]){
    setCurrentGame(data.game);
    setTimeout( handler: ()=>{
      navigate('/game');
    }, timeout: 2000);
    console.log(currentGame);
  }
})

```

Grafika 18: Obsługa zdarzenia 'start_game' po stronie klienta

2.5.2 Server – tworzenie

Po otrzymaniu zdarzenia 'request_game' od użytkownika, serwer przekazuje je dalej do wszystkich użytkowników (zdarzenie wykona użytkownik, którego nazwa jest członem gamelid przekazanego w zdarzeniu – Grafika 17).

```
socket.on("request_game", ( data )=>{  
  console.log(data)  
  socket.broadcast.emit("request_game", {user:data.user, game:data.game})  
});
```

Grafika 19: Obsługa po stronie serwera z wykorzystaniem web socketu – przesłanie zdarzenia dalej

Po otrzymaniu zdarzenia 'create_game' wykonywane są GameService.createGame oraz UserService.addGame:

```
socket.on("create_game", async (game)=>{  
  await GameService.createGame(game);  
  await UserService.addGame(game.white, game.id);  
  socket.join(game);  
});
```

Grafika 20: Obsługa zdarzenia 'create_game' po stronie serwera

```
static createGame = async (game) => {  
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());  
  const result = await connection.collection.insertOne({white:game.white, black:game.black, id:game.id,  
    draw:false, winner:'',started:false, moves:[]});  
  return result;  
}
```

Grafika 21: GameService.createGame

```
static addGame = async (user, gameID) => {  
  const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient());  
  connection.collection.updateOne(  
    { username: user},  
    { $push: { games: gameID } },  
  )  
}
```

Grafika 22: UserService.addGame

Do kolekcji Games wstawiana jest nowa gra z wartościami początkowymi. Pole 'started' ustawione jest początkowo na wartość 'false'.

Do pola 'games' wybranego użytkownika dodawana jest gra przekazana w parametrze.

Następnie, w zależności czy użytkownik zaakceptował zaproszenie, serwer odbiera zdarzenia: 'start_game' i 'add_to_game' lub zdarzenie 'cancel_game' (Grafika 17):

```
socket.on("start_game", async (data) => {
  await GameService.startGame(data.game);
  socket.broadcast.emit("start_game", {user1: data.user1, user2: data.user2, game: data.game});
})
```

Grafika 23: Obsługa zdarzenia 'start_game' po stronie serwera

```
socket.on("add_to_game", async (data) => {
  await UserService.addGame(data.user, data.game);
})
```

Grafika 24: Obsługa zdarzenia 'add_to_game' po stronie serwera

```
socket.on("cancel_game", async (game) => {
  await GameService.deleteGame(game);
})
```

Grafika 25: Obsługa zdarzenia 'cancel_game' po stronie serwera

Następnie mogą być wykonane akcje GameService.startGame, UserService.addGame, GameService.deleteGame.

```
static startGame = async (gameId) => {
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());
  return await connection.collection.updateMany({id: gameId}, {$set: {started: true}});
}
```

Grafika 26: GameService.startGame

Pole 'started' wybranej gry z kolekcji 'Games' ustawiane jest na wartość 'true'.

```
static deleteGame = async (gameId) => {
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());
  const result = await connection.collection.deleteOne({id: gameId});
  return result;
}
```

Grafika 27: GameService.deleteGame

Wcześniej utworzona gra zostaje usunięta z kolekcji Games.

2.5.3 Client - Inicjalizacja ekranu i dalsza rozgrywka

W komponencie Board.tsx tworzony jest BoardState:

```
const [ boardState, setBoardState ] = useState<FieldElement[][]>([
  [ ...order.map( f=> ({ piece: f, color: oppositeColor(playerColor), state:'initial' }) ) ],
  [ ...order2.map( f=> ({ piece: f, color: oppositeColor(playerColor), state:'initial' }) ) ],
  [...order1.map( f=> ({ piece: f, color: '', state:'initial' }) )],
  [...order1.map( f=> ({ piece: f, color: '', state:'initial' }) )],
  [...order1.map( f=> ({ piece: f, color: '', state:'initial' }) )],
  [...order1.map( f=> ({ piece: f, color: '', state:'initial' }) )],
  [ ...order2.map( f=> ({ piece: f, color: playerColor, state:'initial' }) ) ],
  [ ...order.map( f=> ({ piece: f, color: playerColor, state:'initial' }) ) ],
])
```

Grafika 28: boardState w komponencie Board.tsx

Jest on następnie mapowany na tablicę komponentów Field, które po nadaniu parametrów reprezentują odpowiednie figury na szachownicy.

```
boardState.map(
  (row, rowID) => (
    <div key={rowID} className="row">
      {
        row.map( (field, colID) => ( <Field key={`-${rowID}-${colID}`}
          stateUpdate={{board:FieldElement[][]=>setBoardState({...board}}
            blackField={{ (rowID+colID)%2===0 }
            board={boardState}
            kings={kingsPositions}
            turn={turn}
            turnUpdate={{(turn:boolean)>=>setTurn(turn)}
            kingsUpdate={{(kings:{white_king:{x:number,y:number},
              black_king:{x:number, y:number}}=>setKingsPositions(kings))
            intact={intact}
            intactUpdate={{(intact:{king:boolean,long:boolean,short:boolean})=>setIntact(intact)}
            x={rowID}
            y={colID} /> ))
        }
      }
    </div>
  )
)
```

Grafika 29: Ustawianie Field wartościami z boardState

Poszczególne kliknięcia na pola są obsługiwane w funkcji 'clickHandle':

```

if(board[x][y].state === 'initial' && board[x][y].color === playerColor){

    for (let i=0;i<8;i++){
        for(let j=0;j<8;j++){
            board[i][j].state = 'initial';
        }
    }

    let toUpdate = possibleMoves(board,x,y, check: true,kings);
    toUpdate.forEach((e : {x: number, y: number} )=>{
        board[e.x][e.y].state = 'possible';
    });
    board[x][y].state = 'selected';
    console.log(board);
    stateUpdate(board);
}

```

Grafika 30: Obsługa pola w stanie 'initial'

Po kliknięciu na pole w stanie 'initial' w swojej turze, za pomocą funkcji possibleMoves, aktualizowany jest stan poszczególnych Field na 'possible'. Stan klikniętego pola ustawiany jest na 'selected'.

```

const possibleMoves = (board:FieldElement[][], x:number, y:number, check:boolean, kings:any) :{x:number,y:number}[] =>{

    if(board[x][y].piece===PieceEnum.Knight){
        return Knight(board,x,y,check,kings);
    }
    if(board[x][y].piece===PieceEnum.King){
        return King(board,x,y,check,kings);
    }
    if(board[x][y].piece===PieceEnum.Rook){
        return Rook(board,x,y,check,kings);
    }
    if(board[x][y].piece===PieceEnum.Bishop){
        return Bishop(board,x,y,check,kings);
    }
    if(board[x][y].piece===PieceEnum.Queen){
        return Queen(board,x,y,check,kings);
    }
    if(board[x][y].piece===PieceEnum.Pawn){
        return Pawn(board,x,y,check,kings);
    }
    else
        return []
}

```

Grafika 31: Funkcja possibleMoves

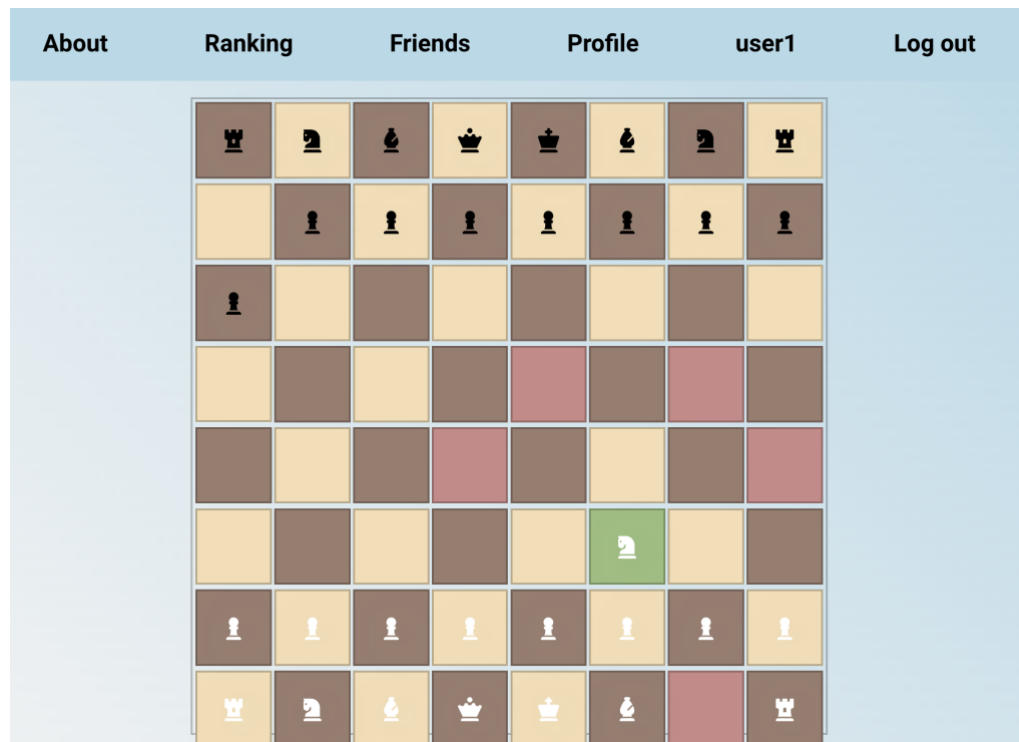
Schemat działania przedstawiony zostanie na przykładzie skoczka szachowego.

```
const Knight=(board:FieldElement[][], x:number, y:number, check:boolean, kings:any)=>{
  let res = Array( arrayLength: 0);
  let toCheck = [{x:x+1,y:y+2}, {x:x+1,y:y-2}, {x:x-1,y:y+2}, {x:x-1,y:y-2}, {x:x-2,y:y+1},
    {x:x-2,y:y-1}, {x:x+2,y:y-1}, {x:x+2,y:y+1}];

  toCheck.forEach((e :{x: number, y: number} )=>{
    if(validField(e.x, e.y) && board[e.x][e.y].color !== board[x][y].color) {
      res.push(e);
    }
  })
  let empty = Array( arrayLength: 0);
  if(check){
    for(let i=0;i<res.length;i++){
      if(!testMove(x,y,res[i].x,res[i].y,board,kings)){
        empty.push(res[i]);
      }
    }
    console.log("e: ",empty);
  }
  return check ? empty : res;
}
```

Grafika 32: Sposób wyznaczania możliwych ruchów dla skoczka

- 1) W liście toCheck umieszczone zostały wszystkie potencjalne ruchy dla skoczka znajdującego się na pozycji (x,y).
- 2) Dla każdego ruchu sprawdzane jest czy mieści się w zakresie szachownicy (validField) oraz czy na wskazanym polu nie ustawiona jest już inna figura tego koloru.
- 3) Ruchy które przeszły wstępne sprawdzenie są następnie testowane pod kątem potencjalnej odsłony własnego króla (testMove)
- 4) Funkcja zwraca listę ruchów, które spełniły wszystkie warunki.



Grafika 33: Rezultat funkcji possibleMoves

Po kliknięciu na pole w stanie 'selected' wartości 'state' wszystkich Field ustawiane są na 'initial'.

```
else if(board[x][y].state === 'selected'){
  board.forEach((row : FieldElement[] )=>{
    row.forEach((e : FieldElement )=>{
      e.state = 'initial';
    })
  })
  console.log(board);
  stateUpdate(board);
}
```

Grafika 34: Obsługa pola w stanie 'selected'

Schemat obsługi pola w stanie 'possible' (wykonanie ruchu):

- 1) Zamiana wartości w boardState
- 2) Sprawdzenie ruchów specjalnych (ruch królem, ruch wieżą, roszada)
- 3) Wygenerowanie etykiety ruchu
- 4) Dodanie ruchu do listy ruchów danej partii
- 5) Emitowanie zdarzenia 'player_move' oraz zmiana tury

```
socket?.emit('player_move', {moves:currentMoves, game:currentGame});
turnUpdate(false);
```

Grafika 35: Fragment obsługi pola 'possible'

6) Sprawdzenie warunków końca rozgrywki po ruchu

```
if(!can_move(board,oppositeColor(playerColor),kings)){
  const in_check : boolean = playerColor === 'white' ? inCheck(board,kings.black_king.x, kings.black_king.y, kings) :
    inCheck(board,kings.white_king.x, kings.white_king.y, kings);
  if(!in_check){
    console.log('Stalemate.');
```

```
    socket?.emit('game_over', {game:currentGame, winner:'', draw:true})
    refreshMoves();
    notify( text: 'Game drawn.')
```

```
    setTimeout( handler: ()=>{
      navigate('/friends')
    }, timeout: 10000);
  }
  else{
    console.log(`${oppositeColor(playerColor)} king mated.`);
    socket?.emit('game_over', {game:currentGame, winner:user!.username, draw:false});
    refreshMoves();
    notify( text: 'You won.')
```

```
    setTimeout( handler: ()=>{
      navigate('/friends')
    }, timeout: 10000);
  }
}
```

Grafika 36: Sprawdzenie warunków końca partii

Na podstawie otrzymanej etykiety ruchu klient odbierający zdarzenie wykonuje ruch (funkcja move) na swojej szachownicy (aktualizacja boardState).

```
socket?.off('player_move').on('player_move', ({moves, game})=>{
  if(currentGame === game){
    addMove(moves[moves.length - 1]);
    let split = moves[moves.length-1].split('-');
    if(split[0] === '0'){
      if(split.length === 2){
        performShortCastle();
      }
      else
        performLongCastle();
    }
    else{
      let field1 = getField(split[0]);
      let field2 = getField(split[1]);
      console.log(moves);
      console.log('field1:', field1);
      console.log('field2:', field2);
      console.log('move');
      if(split.length === 4){
        const board = [...boardState];
        board[field1.x][field1.y].piece = getPiece(split[3]);
        setBoardState(board);
      }
      move({x:field1.x,y:field1.y}, {x:field2.x, y:field2.y});
    }
    setTurn(true);
  }
});
```

Grafika 37: Obsługa zdarzenia 'player_move' po stronie klienta - strona odbierająca

```
socket?.off('game_over').on('game_over', ({game, winner, draw})=>{
  console.log('payload:', game, winner, draw);
  if(currentGame === game){
    refreshMoves();
    if(draw){
      notify( text: 'Game drawn.')
      setTimeout( handler: ()=>{
        navigate('/friends')
      }, timeout: 10000);
    }
    else{
      notify( text: 'You lost.')
      setTimeout( handler: ()=>{
        navigate('/friends')
      }, timeout: 10000);
    }
  }
});
```

Grafika 38: Obsługa zdarzenia 'game_over' po stronie klienta - strona odbierająca

2.5.4 Server – obsługa rozgrywki

```
socket.on("player_move", async({moves, game})=>{
  console.log(moves);
  console.log(game);
  await GameService.updateBoardState(game,moves);
  socket.broadcast.emit("player_move", {moves:moves, game:game});
});
```



```
socket.on("game_over", async ({ game, winner, draw})=> {
  await GameService.endGame(game, winner, draw);
  socket.broadcast.emit("game_over", {game:game, winner:winner, draw:draw});
});
```

Grafika 39: Obsługa zdarzeń 'player_move' oraz 'game_over' po stronie serwera

W obu przypadkach zdarzenia przekazywane są dalej do użytkowników (broadcast.emit), które obsługiwane są u odpowiedniego użytkownika (Grafika 37, 38).

Po zarejestrowaniu zdarzenia 'player_move' wykonywana jest funkcja GameService.updateBoardState, która jako parametry otrzymuje gameId do edycji oraz zaktualizowaną listę ruchów.

```
static updateBoardState = async (gameId, moves) => {
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());
  return await connection.collection.updateMany({id: gameId}, {$set: {moves: moves}});
}
```

Grafika 40: GameService.updateBoardState

Dla wybranej gry z kolekcji Games ustawiane jest pole 'moves' na przekazaną wartość.

```
static endGame = async (gameId, winner, draw) => {
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());
  return await connection.collection.updateMany({id: gameId}, {$set: {winner: winner, draw:draw}});
}
```

Grafika 41: GameService.endGame

W zależności od wyniku partii (parametry winner:boolean, draw:boolean) ustawiane są odpowiednie wartości dla wybranej partii z kolekcji Games.

2.6 Profil użytkownika i historia partii

2.6.1 Client – lista gier

The screenshot displays the 'user1 Game History' page. At the top, there is a navigation bar with links: 'About', 'Ranking', 'Friends', 'Profile' (highlighted), 'user1', and 'Log out'. Below the navigation bar, the page is divided into two main sections: 'Game list' and 'Statistics'.

Game list: This section contains a list of five games, each with a unique 'gameId' and a 'See details' button. The games are color-coded: green for 'user1-user2-1657203806725' and 'user2-user1-1657540761896', and red for 'user2-user1-1657204074041', 'user3-user1-1657205187299', and 'user1-user2-1657541956057'.

Statistics: This section provides a summary of the user's performance. It includes the following data:

user1 Game History		
Total games played: 18	Total games won: 7	Win ratio: 38.89%
White games: 11	White wins: 6	White win ratio: 54.55%
Black games: 7	Black wins: 1	Black win ratio: 14.29%

Grafika 42: Ekran Profil

Klient zgłasza zapytanie o swoje gry, na podstawie którego, tworzona jest lista gier oraz wyliczane są podstawowe statystyki.

```
const data = await fetch( input: `http://localhost:3002/user/${username}/games`, init: {
  method: 'GET',
  headers: { 'Content-Type': 'application/json'},
});
const json = await data.json();
```

Grafika 43: Zapytanie o gry użytkownika

```
// 20220912201102
// http://localhost:3002/user/user1/games

{
  "status": "ok",
  "data": [
    {
      "_id": "62c6ec5e67bd672bbb04f939",
      "white": "user1",
      "black": "user2",
      "id": "user1-user2-1657203806725",
      "draw": false,
      "winner": "user1",
      "started": true,
      "moves": [
        "e2-e4",
        "e7-e5",
        "f1-c4",
        "f8-c5",
        "d1-f3",
        "b8-c6",
        "f3-f7"
      ]
    },
    {
      "_id": "62c6ed6a67bd672bbb04f93a",
      "white": "user2",
      "black": "user1",
      "id": "user2-user1-1657204074041",
      "draw": false,
      "winner": "user2",
      "started": true,
      "moves": [
        "g1-f3",
        "g8-f6",
        "e2-e4",
        "f6-e4",
        "f1-c4",
        "e4-d6",
        "d1-e2",
        "d6-c4"
      ]
    }
  ]
}
```

Grafika 44: Wynik przykładowego zapytania o gry użytkownika

2.6.2 Server – lista gier

W odpowiedzi na zapytanie o listę gier uruchamiana jest funkcja `userController.getGames` (Grafika 2).

```
const getGames = async (req, res) => {
  if (req.params.username === '' || req.params.username === '0') {
    res.send(ResponseHelper( status: 'No user found', data: []));
  }
  else {
    const data = await GameService.getAllUserGames(req.params.username);
    console.log(data);
    res.send(data ? ResponseHelper( status: 'ok', data) : ResponseHelper( status: 'bad', data: []))
  }
};
```

Grafika 45: `userController.getGames`

Następnie wykonywana jest funkcja `GameService.getAllUserGames` dla konkretnego użytkownika przekazanego w parametrze.

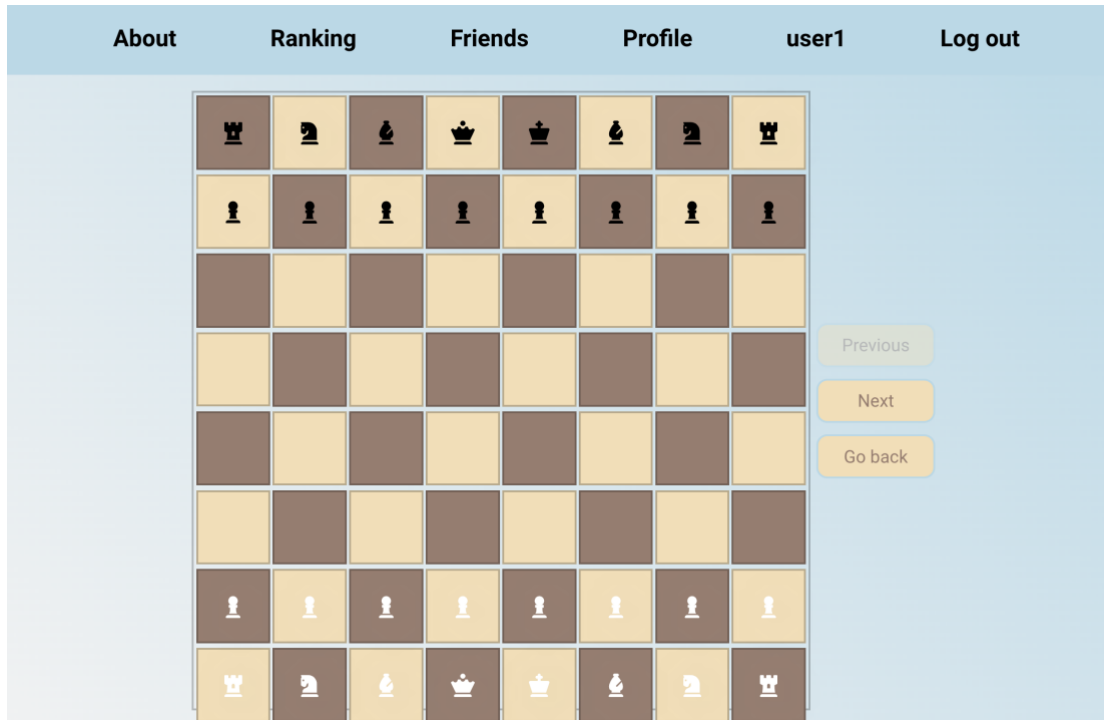
```
static getAllUserGames = async (userId) => {
  const connection = await DBConnection.connect( collectionName: "Games", DBConnection.getClient());
  const res = connection.collection.find({$or: [ {black: userId}, {white:userId}]}).toArray();
  return res;
}
```

Grafika 46: `GameService.getAllUserGames`

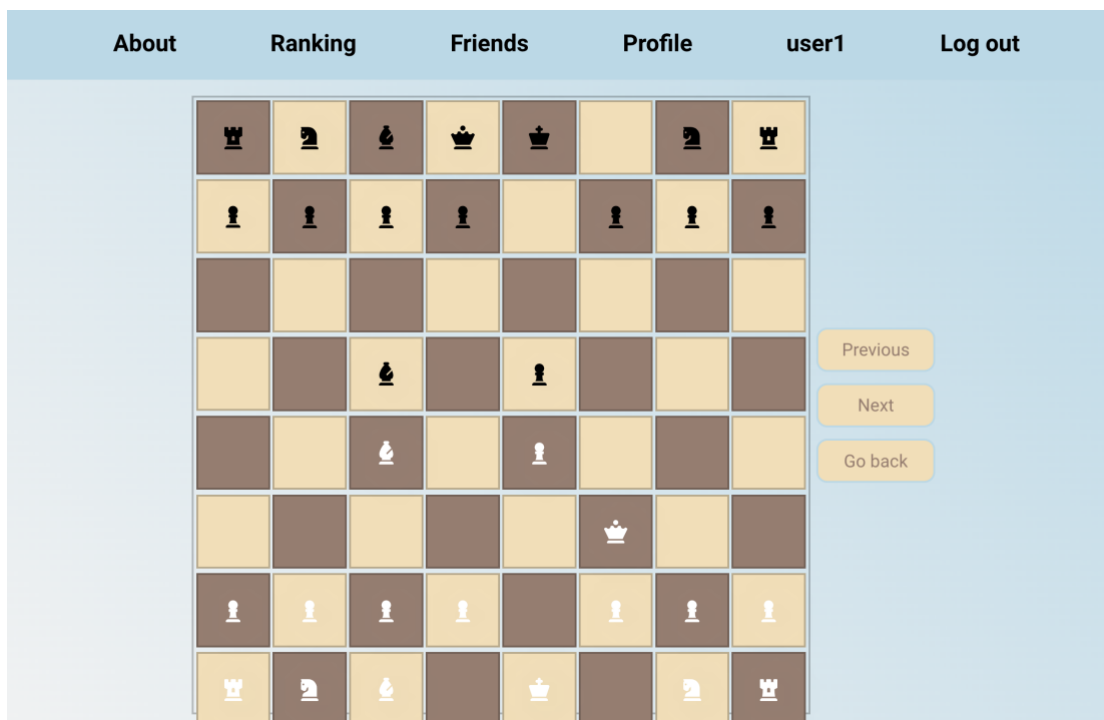
Zwracana jest lista gier, w których uczestniczył użytkownik podany w parametrze.

2.6.3 Client – historia partii

Po kliknięciu 'See details' przy wybranej partii z listy rozegranych gier, klient przekazuje listę ruchów danej partii do komponentu ReviewBoard.tsx, który na tej podstawie generuje tablicę stanów szachownicy zadanej partii, umożliwia przesłedzenie jej przebiegu za pomocą przycisków 'Next' i 'Previous' oraz przycisku 'Go back' przenoszącego do ekranu Profil.



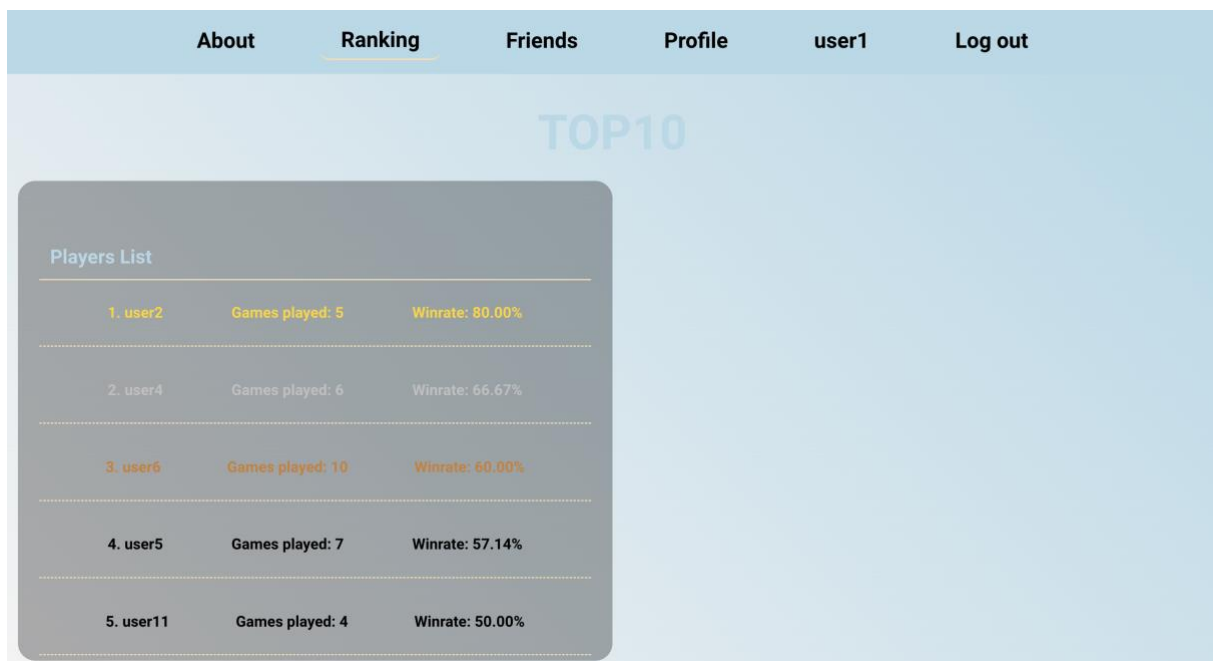
Grafika 47: Ekran BoardReview



Grafika 48: Ekran BoardReview - przewijanie ruchów

2.7 Ranking użytkowników

2.7.1 Client



Grafika 49: Ekran Ranking

Klient wysyła zapytanie do serwera o statystyki użytkowników.

```
export const Ranking: React.FC<RankingProps> = ({}) => {
  const [top10, setTop10] = useState<{username:string, games:number, winrate:string}[]>([]);
  const place = ['first', 'second', 'third', ''];
  const updateTop10 = async ()=>{
    const data = await fetch( input: `http://localhost:3002/userStats`, init: {
      method: 'GET',
      headers: { 'Content-Type': 'application/json'},
    });

    const json = await data.json()
    console.log(json);
    let array = Array( arrayLength: 0);
    json.data.forEach((user:{username:string, games:number, winrate:string})=>{
      array.push({username:user.username, games:user.games, winrate:user.winrate})
    })
    setTop10(array);
  }
}
```

Grafika 50: Obsługa rankingu po stronie klienta

2.7.2 Server

W odpowiedzi serwer wykonuje funkcję `UserController.getWins` (Grafika 2), która po otrzymaniu danych z `UserService.getStats`, przygotowuje z nich posortowane zestawienie top10 użytkowników.

```
const getWins = async (req, res) => {
  const data = await UserService.getStats();
  console.log(data);
  let leaderboard = Array( arrayLength: 0 );
  data.forEach((user) => {
    leaderboard.push({ username: user.username, games: user.games.length,
      winrate: user.games.length > 0 ? ((user.wins.count * 100) / user.games.length).toFixed( fractionDigits: 2 ) : 0 });
  });
  leaderboard.sort( compareFn: (user1, user2) => (user2.winrate - user1.winrate || user2.games - user1.games ) )
  const top10 = leaderboard.slice(0, 10);
  res.send(ResponseHelper( status: 'ok', top10 ));
};
```

Grafika 51: `UserController.getWins`

```
static getStats = async () => {
  const connection = await DBConnection.connect( collectionName: "Users", DBConnection.getClient() );
  return connection.collection.aggregate([
    {
      $lookup: {
        from: "Games",
        let: { user: "$username" },
        pipeline: [
          { $match: { $expr: { $eq: [ "$$user", "$winner" ] } } },
          { $count: "count" }
        ],
        as: "wins"
      }
    },
    {
      $unwind: "$wins"
    }
  ]).toArray()
}
```

Grafika 52: `UserService.getStats`

Wynik zapytania

```
// 20220912210743
// http://localhost:3002/userStats

{
  "status": "ok",
  "data": [
    {
      "username": "user2",
      "games": 5,
      "winrate": "80.00"
    },
    {
      "username": "user4",
      "games": 6,
      "winrate": "66.67"
    },
    {
      "username": "user6",
      "games": 10,
      "winrate": "60.00"
    },
    {
      "username": "user5",
      "games": 7,
      "winrate": "57.14"
    },
    {
      "username": "user11",
      "games": 4,
      "winrate": "50.00"
    },
    {
      "username": "user9",
      "games": 2,
      "winrate": "50.00"
    },
    {
      "username": "user1",
      "games": 17,
      "winrate": "41.18"
    }
  ]
}
```

Grafika 53: Wynik zapytania o statystyki użytkowników

3. Podsumowanie

W dokumencie w sposób skrótowy przedstawiono najważniejsze funkcjonalności aplikacji oraz działanie poszczególnych składowych projektu.