POLITECHNIKA
LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI

Zaawansowane programowanie w Javie

Dokumentacja Projektowa

Projekt: „Gra wieloosobowa ”

Autor:
Wojciech Wnuk

Prowadzący zajęcia:
mgr inż. Albert Rachwał

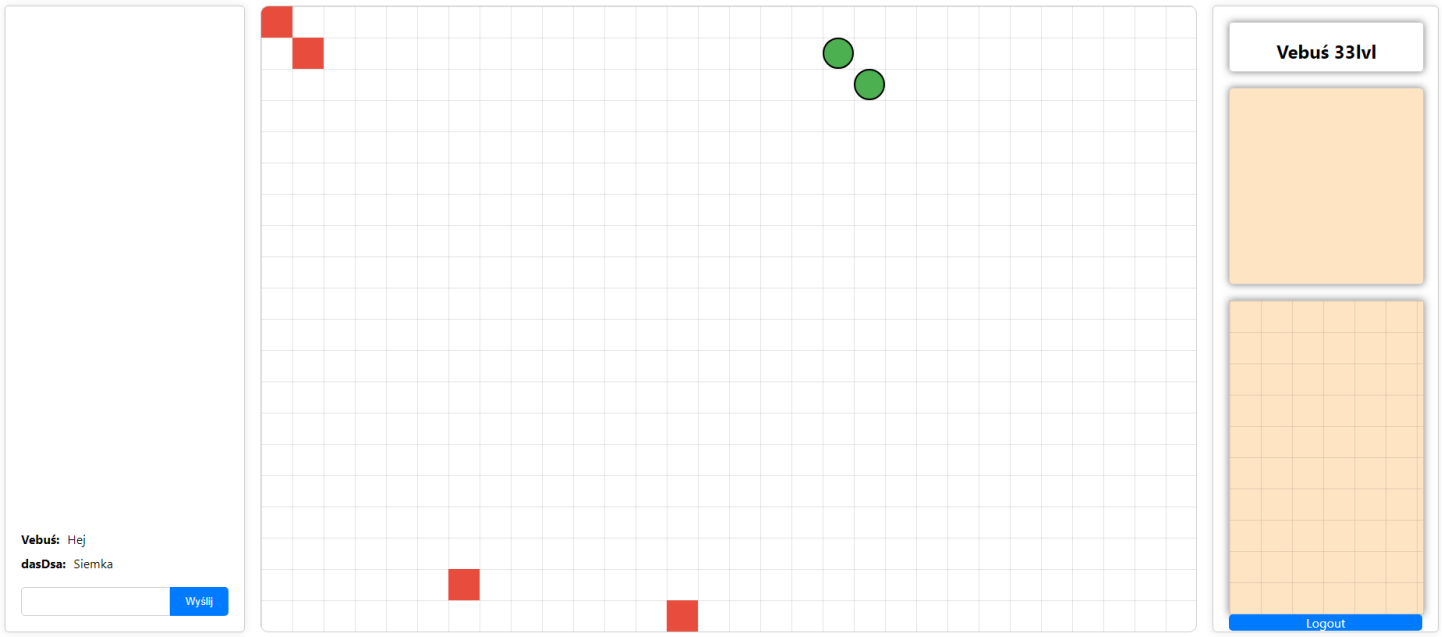
Lublin, 2023

Spis treści

1. Opis projektu

Projekt zaliczeniowy jaki zaimplementowano na potrzeby laboratoriów jest prostą grą wieloosobową stworzoną przy użyciu Frameworka Spring na serwerze oraz Frameworka React na kliencie, z wykorzystaniem technologii umożliwiających komunikację w czasie rzeczywistym. Gra została wyposażona w takie widoki jak:

- Widok strony rejestracji
- Widok strony logowania
- Widok dodawania przeciwnika
- Widok gry

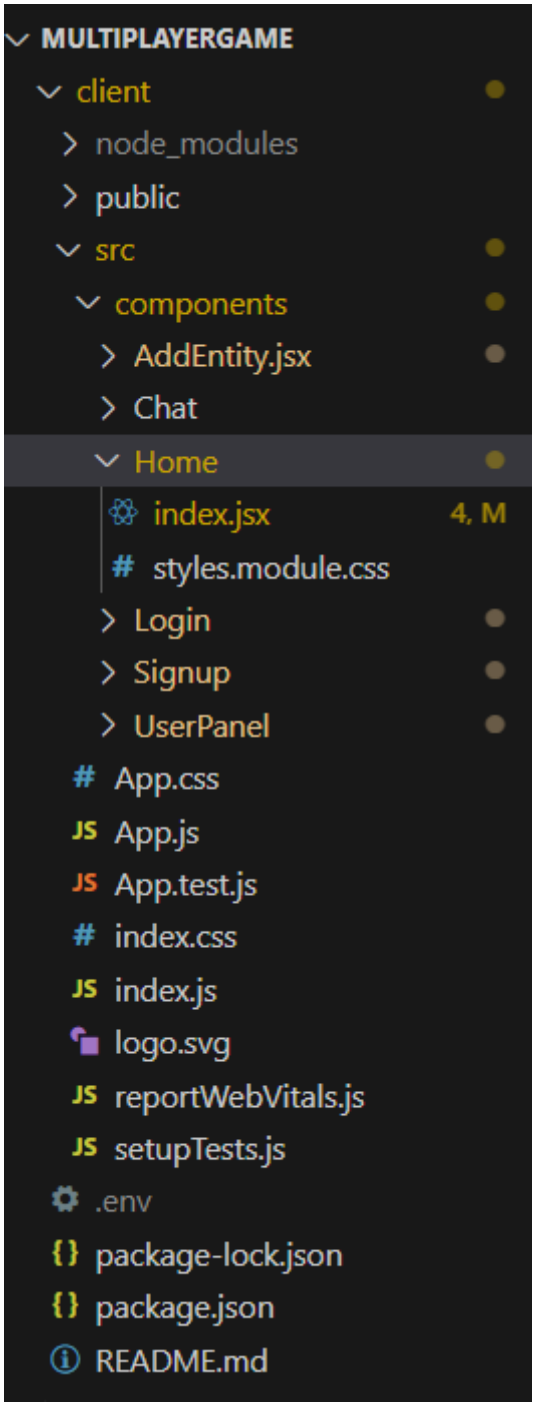


Rys 1. Widok strony gry

Gra polega na prostym zbieraniu punktów przez różnych użytkowników. Każdy gracz posiada swoją postać (zielone kropki), którymi porusza się za pośrednictwem klawiatury. Celem gracza jest zebrać jak najwięcej czerwonych kwadratów, które po zebraniu pojawiają się od nowa na losowo wygenerowanych koordynatach. Całość oczywiście jest widoczna dla wszystkich użytkowników jednocześnie – każdy ruch jakiegokolwiek użytkownika, zebranie czerwonego kwadratu oraz chat są natychmiastowo synchronizowane dla każdego użytkownika korzystając z zaawansowanych technologii w celu zminimalizowania opóźnienia. Dodatkowo użytkownik z rolą „Admin” dostaje opcję takie jak dodawanie nowego „entity” (kwadraty) oraz edycje aktualnego. Dane takie jak pozycje graczy oraz entity są przechowywane na bieżąco w bazie danych lecz w przypadku chatu zachowana jest ulotność danych.

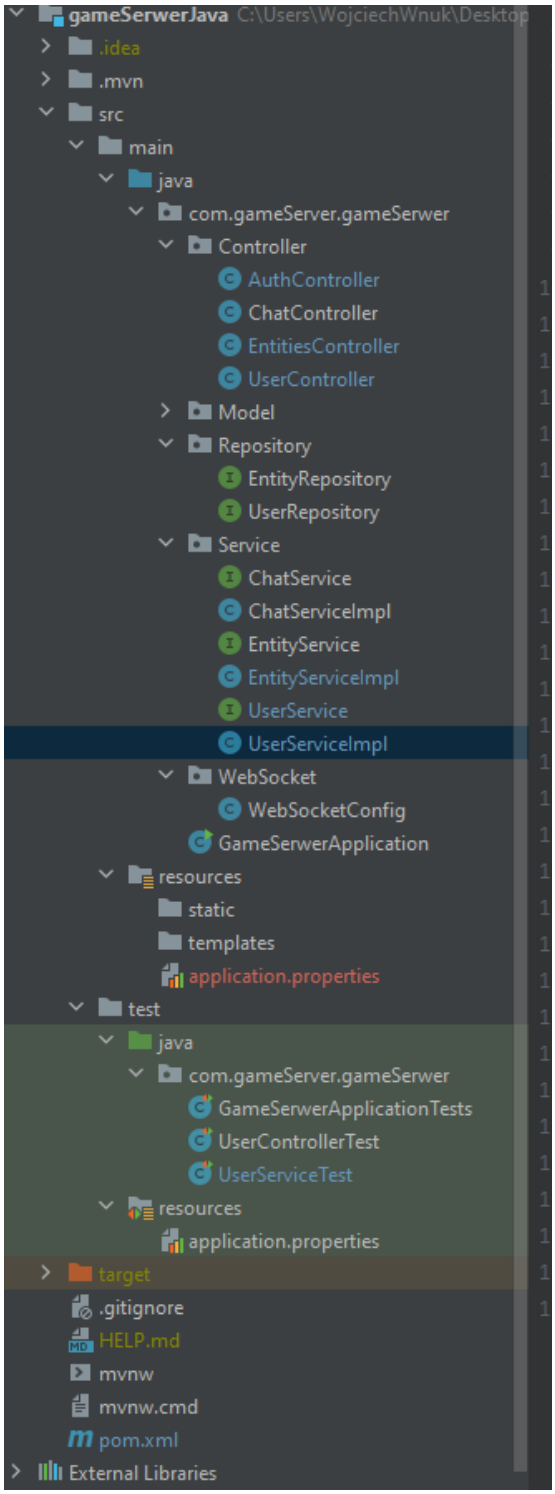
2. Struktura aplikacji serwerowej oraz klienckiej

Zaimplementowana aplikacja kliencka podzielona jest na komponenty, cztery główne będące jednocześnie podstronami oraz dwa – Chat oraz UserPanel będące elementami Home.



Rys. 2 Struktura aplikacji klienta

Natomiast aplikacja serwerowa została odpowiednio podzielona na kontrolery, modele, repozytoria, serwisy oraz konfigurację WebSocketa. Dodatkowo w wydzielonej części projektu znajduje się katalog zawierający testy – zarówno jednostkowe jak i integracyjne.



Rys. 3 Struktura aplikacji serwerowej

3. Wykorzystane technologię

Do zaimplementowania aplikacji wykorzystano szereg zaawansowanych technologii, skorzystano między innymi z:

- WebSocket
- NodeJS
- JavaScript
- Java
- Spring
- React
- Lombok
- MongoDB
- JUnit

Kluczowymi z wymienionych są Websocket dzięki, któremu można było zaimplementować aktualizacje danych w czasie rzeczywistym oraz MongoDB – szybka i elastyczna baza danych, która pomogła zminimalizować opóźnienie do nieznacznych wartości.

4. Implementacja

W tej części skupiono się głównie na pokazaniu części serwerowej – w końcu to ona jest tą ważniejszą częścią tego projektu. Jednakże aby dopełnić projekt zaprezentowane zostaną także kluczowe elementy aplikacji klienckiej.

Spośród kontrolerów zostanie omówiony najważniejszy pod względem systemu – UserController w którym wykonywane są wszelkie endpoiny dotyczące użytkownika. Oczywiście omawiany kontroler zawiera wszelkie działania CRUD lecz zostaną omówione jedynie najbardziej rozbudowane punkty końcowe.

```
@PutMapping("/{playerId}")
public ResponseEntity<?> movePlayer(@PathVariable("playerId") String playerId, @RequestBody User user) {
    try {
        User existingUser = userService.getAllUsers().stream()
                .filter(u -> u.getId().equals(playerId))
                .findFirst()
                .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "User with this id not found"));

            User updatedUser = userService.updateUserPosition(user);
            messagingTemplate.convertAndSend( destination: "/topic/playerPosition", getAllUsers());

            System.out.println("Aktualizacja pozycji gracza" + updatedUser.getX() + updatedUser.getY() + updatedUser.getId() + updatedUser.toString());
            return ResponseEntity.status(HttpStatus.OK).body(updatedUser.getId());
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(e.getMessage());
    }
}
```

Rys. 4 Endpoint odpowiadający za aktualizacje pozycji graczy

Fragment kodu z rysunku 4 otrzymuje dane użytkownika, a następnie aktualizuje jego pozycje dodatkowo rozsyłając te dane do wszystkich użytkowników pozwalając na dostarczanie aktualnych pozycji graczy.

```
@Override
public User updateUserPosition(User user) {
    try {
        User userToUpdate = userrepository.findById(user.getId()).get();
        userToUpdate.setX(user.getX());
        userToUpdate.setY(user.getY());
        return userrepository.save(userToUpdate);
    } catch (NoSuchElementException e) {
        throw new NoSuchElementException("User with this id does not exist");
    }
}
```

Rys. 5 Metoda updateUserPosition w UserServiceImpl

```
const movePlayer = async (playerId, x, y) => {
  try {
    const checkedX = x < 0 ? 0 : x > 1160 ? 1160 : x;
    const checkedY = y < 0 ? 0 : y > 760 ? 760 : y;
    console.log("Move player", playerId, checkedX, checkedY);
    const url = `${process.env.REACT_APP_DEV_SERVER}/api/users/${playerId}`;
    const response = await axios.put(url, {
      id: playerId,
      x: checkedX,
      y: checkedY,
    });
    const player = response.data.data;
    console.log(player);

    setSocketData((prevSocketData) => ({
      ...prevSocketData,
      playerId: playerId,
      x: checkedX,
      y: checkedY,
      lvl: actualLevel,
      online: true,
    }));
    console.log("Data" + socketData);
  } catch (error) {
    console.log("Error fetching players", error);
  }
};
```

Rys. 6 Obsługa poruszania graczy po stronie serwerowej

```java
public interface UserService {

    List<User> getAllUsers();

    User addUser(User user);

    String hashPassword(String password);

    Boolean loginValidation(String email, String password);

    Boolean registerValidation(String email, String password, String nick);

    User updateUserPosition(User user);

    Boolean deleteUser(String id);

    Optional<User> getUserById(String id);

    ResponseEntity<?> updateUserLvl(User user);

    ResponseEntity<?> updateUserOnline(User user, Boolean online);


}
```

Rys. 6 Interfejs UserService

Aplikacja oczywiście została wyposażona w walidacje dla wszelkich danych, które mogą powodować błędy. Do przedstawienia przykładowej walidacji posłużymy się przejściem przez cały proces zalogowania użytkownika.

Po wypełnieniu formularza logowania użytkownik klika przycisk zaloguj powodując wykonanie poniżej przedstawionej akcji:

```
const handleSubmit = async (e) => {
    e.preventDefault()
    try {
        const url = `${process.env.REACT_APP_DEV_SERVER}/api/auth`
        const { data: res } = await axios.post(url, data)
        localStorage.setItem("token", res.data)
        localStorage.setItem("email", data.email)
        localStorage.setItem("playerId", res)
        window.location = "/"
    } catch (error) {
        if (
            error.response &&
            error.response.status >= 400 &&
            error.response.status <= 500
        ) {
            setError(error.response.data.message)
        }
    }
}
```

Rys. 7 Obsługa przycisku zaloguj

Następnie dane przekazane z formularza lądują na serwerze gdzie odbywa się sprawdzanie danych oraz walidacja.

```
@RestController
@RequestMapping("/api/auth")
@CrossOrigin
public class AuthController {
    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity<?> auth(@RequestBody User user) {

        Optional<User> existingUser = userService.getAllUsers().stream()
                .filter(u -> u.getEmail().equals(user.getEmail()))
                .findFirst();

        if (!existingUser.isPresent() || !BCrypt.checkpw(user.getPassword(), existingUser.get().getPassword()) || userService.loginValidation(user.getEmail(), user.getPassword()) == false) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid Email or Password");
        }

        return ResponseEntity.status(HttpStatus.OK).body(existingUser.get().getId());

    }


}
```

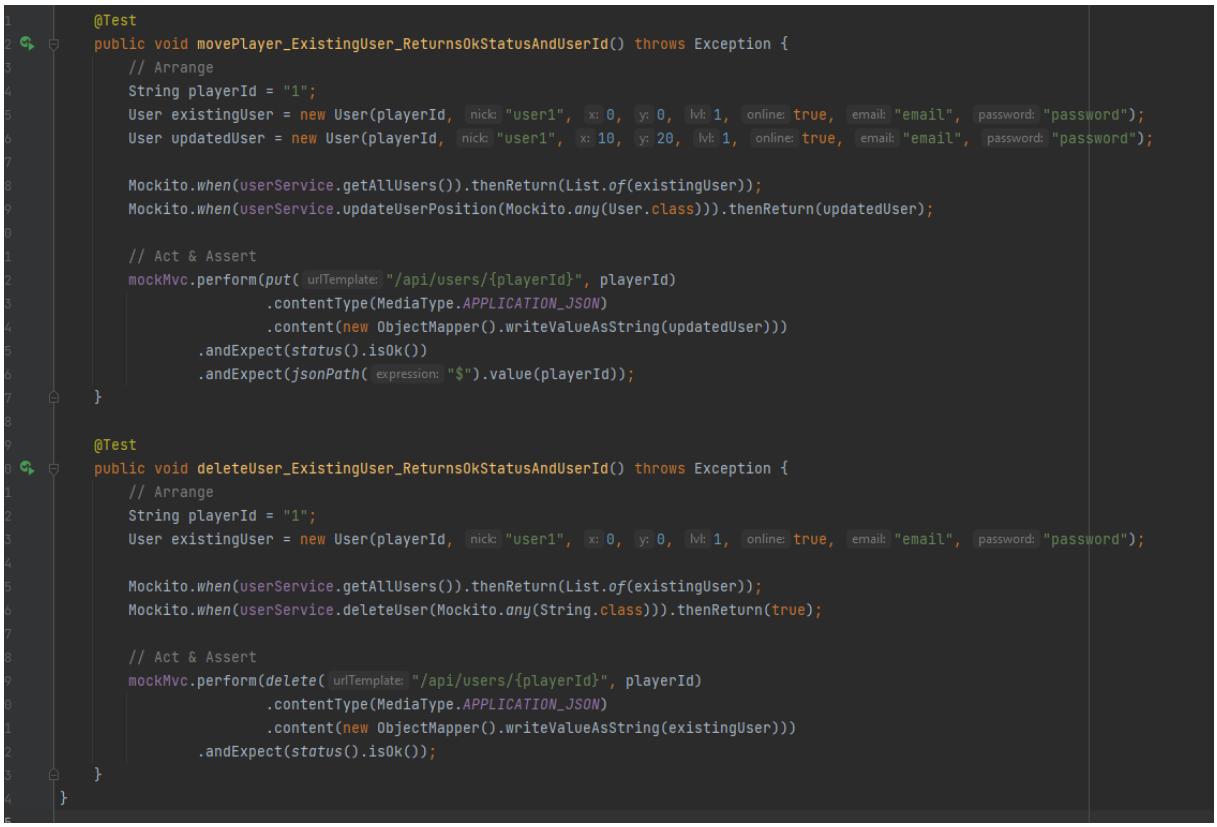
Rys. 8 AuthController

5. Testy

Aplikacja serwerowa została wyposażona w testy jednostkowe jak i integracyjne, które są szczególnie ważne przy rozwoju gry, w takim typie aplikacji nie może być miejsca na błędy. Oto przykłady kilku z testów:

```
@Test
public void movePlayer_shouldUpdateUserPosition() {
    // Arrange
    User user = new User( id: "1",  nick: "user1",  x: 0,  y: 0,  lvl: 1,  online: true,  email: "email",  password: "password");

    userService.addUser(user);

    User userUpdated = new User( id: "1",  nick: "user1",  x: 40,  y: 40,  lvl: 1,  online: true,  email: "email",  password: "password");


    assertEquals( expected: 0, user.getX());
    assertEquals( expected: 0, user.getY());
    // Act
    userService.updateUserPosition(userUpdated);

    // Assert
    Optional<User> actualUser = userService.getUserById("1");
    assertTrue(actualUser.isPresent());

    User retrievedUser = actualUser.get();
    assertEquals( expected: "1", retrievedUser.getId());
    assertEquals( expected: 40, retrievedUser.getX());
    assertEquals( expected: 40, retrievedUser.getY());
    assertEquals( expected: "user1", retrievedUser.getNick());
}

@Test
public void loginValidation() {

    // Act & Assert
    assertTrue(userService.loginValidation( email: "nonEmptyEmail@wp.pl",  password: "nonEmptyPassword"));

    assertFalse(userService.loginValidation( email: "",  password: "password"));
    assertFalse(userService.loginValidation( email: "email",  password: ""));
    assertFalse(userService.loginValidation( email: "",  password: ""));
}
```

Rys. 9 Testy jednostkowe



```
    @Test
    public void movePlayer_ExistingUser_ReturnsOkStatusAndUserId() throws Exception {
        // Arrange
        String playerId = "1";
        User existingUser = new User(playerId, nick: "user1", x: 0, y: 0, lvl: 1, online: true, email: "email", password: "password");
        User updatedUser = new User(playerId, nick: "user1", x: 10, y: 20, lvl: 1, online: true, email: "email", password: "password");

        Mockito.when(userService.getAllUsers()).thenReturn(List.of(existingUser));
        Mockito.when(userService.updateUserPosition(Mockito.any(User.class))).thenReturn(updatedUser);

        // Act & Assert
        mockMvc.perform(put( urlTemplate: "/api/users/{playerId}", playerId)
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(new ObjectMapper().writeValueAsString(updatedUser)))
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$").value(playerId));
    }

    @Test
    public void deleteUser_ExistingUser_ReturnsOkStatusAndUserId() throws Exception {
        // Arrange
        String playerId = "1";
        User existingUser = new User(playerId, nick: "user1", x: 0, y: 0, lvl: 1, online: true, email: "email", password: "password");

        Mockito.when(userService.getAllUsers()).thenReturn(List.of(existingUser));
        Mockito.when(userService.deleteUser(Mockito.any(String.class))).thenReturn(true);

        // Act & Assert
        mockMvc.perform(delete( urlTemplate: "/api/users/{playerId}", playerId)
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(new ObjectMapper().writeValueAsString(existingUser)))
                .andExpect(status().isOk());
    }
}
```

Rys. 10 Testy integracyjne

6. Podsumowanie

Finalnie udało się zrealizować projekt realizujące poniższe wymagania:

- Aplikacja wykonana z wykorzystaniem Frameworka Spring
- Wykorzystanie bazy danych
- Logowanie i rejestracja
- Hashowanie haseł z wykorzystaniem soli i pieprzu
- Operacje CRUD
- Rozbudowane formularze do zapisu oraz edycji danych
- Walidacja danych
- Testy jednostkowe
- Obsługa ról użytkowników*
- Budowa REST API z oddzielnym front-endem**

A także kilka ponadprogramowych, jedyne co nie zostało uwzględnione – obsługa sesji użytkownika, została zastąpione poprzez użycie ograniczeń nałożonych poprzez aplikacje kliencką.