Spring Data JPA + Spring Security

Praktyczne zastosowanie w aplikacji wypożyczalni gier

Autor: Marek Daniszewski
21.07.2025

Aplikacja wypożyczalni gier

Aplikacja umożliwia użytkownikom przeglądanie katalogu gier oraz ich wypożyczanie. System obsługuje różne role użytkowników i zapewnia bezpieczeństwo poprzez uwierzytelnianie JWT. Główne funkcjonalności to zarządzanie użytkownikami, katalogiem gier oraz procesem wypożyczeń.

Funkcjonalności:

- Rejestracja i logowanie użytkowników
- Przeglądanie katalogu gier
- Wypożyczanie i zwracanie gier
- Różne role użytkowników (USER, ADMIN)
- Zabezpieczenia oparte na JWT Token

Technologie:

- Spring Boot 3.x
- Spring Data JPA
- Spring Security
- JWT (JSON Web Token)
- H2 Database

Spring Data JPA - co to?

Spring Data JPA to warstwa abstrakcji nad standardowym JPA/Hibernate, która znacząco upraszcza pracę z bazą danych. Automatycznie generuje implementacje repozytoriów na podstawie interfejsów, eliminując potrzebę pisania powtarzalnego kodu.

Spring Data JPA to:

- Warstwa abstrakcji nad JPA/Hibernate
- Automatyczne tworzenie implementacji repozytoriów
- Metody generowane z nazwy
- Metody zaimplementowane z góry: save(), findById(), findAll(), delete(), count(), existsById()

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Spring automatycznie tworzy implementację tych metod!
    Optional<User> findByUsername(String username);  // SELECT * FROM users WHERE username = ?
    Optional<User> findByEmail(String email);        // SELECT * FROM users WHERE email = ?
    boolean existsByUsername(String username);       // SELECT COUNT(*) > 0 FROM users WHERE username = ?
    boolean existsByEmail(String email);             // SELECT COUNT(*) > 0 FROM users WHERE email = ?
}
```

CRUD

Spring Data JPA automatycznie generuje implementacje repozytoriów dla operacji CRUD poprzez interfejsy i konwencje nazewnicze. Ogromną zaletą implementacji takiego interfejsu jest brak potrzeby pisania kodu SQL, automatyczne zarządzanie transakcjami, wbudowana paginacja i sortowanie.

Podstawowe interfejsy:
Repository<T, ID> - pusty marker interface
CrudRepository<T, ID> - podstawowe operacje CRUD
JpaRepository<T, ID> - najpełniejszy z flush(), batch operations

```
// Podstawowe CRUD
gameRepository.save(game);          // Create/Update
gameRepository.findById(1L);        // Read
gameRepository.deleteById(1L);      // Delete
gameRepository.findAll();           // Read All
```

```
// Spring generuje implementację na podstawie nazwy metody
List<Game> findByTitle(String title);                    // WHERE title = ?
List<Game> findByPriceGreaterThan(BigDecimal price);     // WHERE price > ?
List<Game> findByTitleAndGenre(String title, String genre); // WHERE title = ? AND genre = ?
Page<Game> findByGenre(String genre, Pageable pageable);   // Z paginacją
```

CRUD – przykładowe zapytania REST

Spring Data JPA automatycznie generuje implementacje repozytoriów dla operacji CRUD poprzez interfejsy i konwencje nazewnicze. Ogromną zaletą implementacji takiego interfejsu jest brak potrzeby pisania kodu SQL, automatyczne zarządzanie transakcjami, wbudowana paginacja i sortowanie.

Podstawowe interfejsy:
Repository<T, ID> - pusty marker interface
CrudRepository<T, ID> - podstawowe operacje CRUD
JpaRepository<T, ID> - najpełniejszy z flush(), batch operations

```
// Podstawowe CRUD
gameRepository.save(game);          // Create/Update
gameRepository.findById(1L);        // Read
gameRepository.deleteById(1L);      // Delete
gameRepository.findAll();           // Read All
```

```
// Spring generuje implementację na podstawie nazwy metody
List<Game> findByTitle(String title);                    // WHERE title = ?
List<Game> findByPriceGreaterThan(BigDecimal price);     // WHERE price > ?
List<Game> findByTitleAndGenre(String title, String genre); // WHERE title = ? AND genre = ?
Page<Game> findByGenre(String genre, Pageable pageable);   // Z paginacją
```

Encje JPA

Encje JPA to klasy reprezentujące tabele w bazie danych. Adnotacje JPA definiują sposób mapowania obiektów na strukturę bazy. W naszej aplikacji encja User reprezentuje użytkowników systemu.

```java
@Entity  // Oznacza że to encja JPA
@Table(name = "USERS")  // Mapuje na tabelę USERS
public class User {
    @Id  // Klucz główny
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // Auto-increment
    private Long id;

    @Column(nullable = false, unique = true)  // Kolumna wymagana i unikalna
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)  // Hasło zawsze wymagane
    private String password;

    @Enumerated(EnumType.STRING)  // Enum przechowywany jako tekst
    private Role role;

    // konstruktory, gettery, settery...
}
```

Relacje

Relacje między encjami pozwalają na modelowanie powiązań w bazie danych. W naszym przypadku jeden użytkownik może mieć wiele wypożyczeń, a jedna gra może być wypożyczona przez wielu użytkowników w różnym czasie.

```java
// W klasie User - jeden użytkownik ma wiele wypożyczeń
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JsonIgnore  // Zapobiega cyklom przy serializacji JSON
private Set<GameRental> gameRentals = new HashSet<>();

// W klasie GameRental - wiele wypożyczeń należy do jednego użytkownika
@ManyToOne(fetch = FetchType.LAZY)  // Lazy loading - ładuje na żądanie
@JoinColumn(name = "user_id", nullable = false)  // Kolumna FK
private User user;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "game_id", nullable = false)
private Game game;
```

Spring Security

Spring Security dostarcza kompleksowy system zabezpieczeń dla aplikacji. Konfiguracja definiuje które endpointy są publiczne, które wymagają uwierzytelniania, a które konkretnych ról. W tej aplikacji użyłem JWT zamiast tradycyjnych sesji.

```java
@Configuration
@EnableWebSecurity  // Włącza Spring Security
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .csrf(csrf -> csrf.disable())  // Wyłączamy CSRF dla API
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()  // Logowanie dostępne dla wszystkich
                .requestMatchers(HttpMethod.GET, "/api/games/**").hasAnyRole("USER", "ADMIN")  // Gry dla zalogowanych
                .requestMatchers("/api/admin/**").hasRole("ADMIN")  // Admin panel tylko dla adminów
                .anyRequest().authenticated()  // Reszta wymaga logowania
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))  // Brak sesji, używamy JWT
            .addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class)  // Nasz filtr JWT
            .build();
    }
}
```

JWT (JSON Web Token)

JWT to standard przesyłania informacji między stronami w sposób bezpieczny. Token składa się z trzech części oddzielonych kropkami. Zawiera informacje o użytkowniku i jest podpisany cyfrowo, co gwarantuje jego autentyczność.

Proces uwierzytelniania:

1.Użytkownik loguje się i otrzymuje token JWT
2.Przy każdym żądaniu wysyła token w nagłówku Authorization: Bearer xyz
3.Serwer sprawdza podpis i ważność tokenu
4.Jeśli token jest OK, użytkownik otrzymuje dostęp

Zalety JWT:

•Bezstanowość - serwer nie musi pamiętać sesji
•Bezpieczeństwo - token jest podpisany cyfrowo
•Automatyczne wygasanie - token ma określony czas życia

```
Authorization: Basic dXNlcjpwYXNz     # Username + password
Authorization: Bearer eyJhbGci...     # Token JWT
Authorization: ApiKey abc123          # Klucz API
```

Alternatywy dla JWT w Spring Security

Tradycyjne sesje HTTP:
•Sesje przechowywane na serwerze z identyfikatorem w cookie
•Automatyczne zarządzanie przez kontener servlet
•Idealne dla aplikacji monolitycznych
•Problemy ze skalowalnością w środowiskach rozproszonych
OAuth 2.0 / OpenID Connect:
•Standard dla aplikacji korzystających z zewnętrznych dostawców tożsamości
•Integracja z Google, Facebook, GitHub, Azure AD
•Delegowanie uwierzytelniania do zewnętrznych serwisów
•Doskonałe dla aplikacji B2C i SSO
Basic Authentication:
•Przesyłanie username/password w każdym żądaniu
•Zakodowane w Base64 w nagłówku Authorization
•Proste w implementacji ale wymaga HTTPS
•Odpowiednie dla API wewnętrznych i prostych integracji

API Keys:
•Statyczne klucze przypisane do użytkowników lub aplikacji
•Przesyłane w nagłówkach lub parametrach URL
•Brak wygasania - wymagają ręcznego zarządzania
•Popularne w API publicznych i B2B
SAML (Security Assertion Markup Language):
•Standard enterprise dla Single Sign-On
•XML-owy format wymiany informacji o tożsamości
•Popularne w korporacjach i instytucjach rządowych
•Kompleksowe ale ciężkie w implementacji
Certificate-based Authentication:
•Uwierzytelnianie oparte na certyfikatach X.509
•Najwyższy poziom bezpieczeństwa
•Używane w aplikacjach high-security i IoT
•Skomplikowane zarządzanie infrastrukturą PKI

JwtRequestFilter - filtr zabezpieczeń

JwtRequestFilter to filtr który wykonuje się przy każdym żądaniu HTTP. Sprawdza czy użytkownik przesłał poprawny token JWT i jeśli tak, loguje go automatycznie do Spring Security. Działa jako bramkarz aplikacji.

```java
@Override
protected void doFilterInternal(HttpServletRequest request,
                                HttpServletResponse response,
                                FilterChain chain) {

    // 1. Wyciągamy nagłówek Authorization z żądania
    String tokenHeader = request.getHeader("Authorization");

    // 2. Sprawdzamy czy zawiera token Bearer
    if (tokenHeader != null && tokenHeader.startsWith("Bearer ")) {
        String jwt = tokenHeader.substring( beginIndex: 7);  // Usuwamy "Bearer "

        // 3. Wyciągamy username z tokenu
        String username = jwtUtil.extractUsername(jwt);

        // 4. Sprawdzamy czy token jest ważny
        if (jwtUtil.validateToken(jwt, userDetails)) {
            // 5. Logujemy użytkownika do Spring Security
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }

    // 6. Puszczamy żądanie dalej do kontrolera
    chain.doFilter(request, response);
}
```

Role użytkowników

System ról pozwala na kontrolowanie dostępu do różnych funkcji aplikacji. W mojej aplikacji mam dwie role - zwykłych użytkowników i administratorów. Role są sprawdzane zarówno na poziomie konfiguracji Security (SpringSecurity) jak i w kontrolerach (adnotacje określające jakie role mają dostęp do danego endpointa)

Poziomy dostępu:
•Publiczne - endpointy logowania (/api/auth/**)
•USER - przeglądanie i wypożyczanie gier
•ADMIN - zarządzanie całym systemem

```java
// Definicja ról w enumie
public enum Role {
    USER,   // Zwykły użytkownik - może przeglądać i wypożyczać gry
    ADMIN   // Administrator - pełny dostęp do systemu
}
```

```java
// Dostęp dla zalogowanych użytkowników (USER lub ADMIN)
@GetMapping("/games")
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
public List<Game> getAllGames() { ... }

// Dostęp tylko dla administratorów
@DeleteMapping("/admin/games/{id}")
@PreAuthorize("hasRole('ADMIN')")
public void deleteGame(@PathVariable Long id) { ... }
```

```java
// Bean definiujący reguły bezpieczeństwa i uprawnienia- Definuje "kto może wejść gdzie"
@Bean  new *
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
        .authorizeHttpRequests( AuthorizationManagerRequestMat... authz -> authz
                // Publiczne endpointy
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/h2-console/**").permitAll()

                // Admin endpointy
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/users/**").hasRole("ADMIN")

                // User endpointy
                .requestMatchers("/api/games/**").hasAnyRole( ...roles: "USER", "ADMIN")
                .requestMatchers("/api/rentals/**").hasAnyRole( ...roles: "USER", "ADMIN")

                // Wszystkie inne żądania wymagają uwierzytelnienia
                .anyRequest().authenticated()
        )
```

Logowanie użytkownika

Proces logowania składa się z weryfikacji danych użytkownika i wygenerowania tokenu JWT. AuthController obsługuje żądania logowania, sprawdza dane uwierzytelniające i zwraca token do dalszego użycia.

```java
@PostMapping("/login")
public ResponseEntity<?> login(@RequestBody AuthRequest request) {
    try {
        // 1. Sprawdzamy username i password przez Spring Security
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                request.getUsername(),
                request.getPassword())
        );

        // 2. Ładujemy szczegóły użytkownika z bazy
        UserDetails userDetails = userDetailsService
            .loadUserByUsername(request.getUsername());

        // 3. Generujemy token JWT dla użytkownika
        String token = jwtUtil.generateToken(userDetails);

        // 4. Zwracamy token wraz z danymi użytkownika
        return ResponseEntity.ok(new AuthResponse(token, username, role));

    } catch (BadCredentialsException e) {
        // 5. Jeśli dane są błędne, zwracamy błąd 401
        return ResponseEntity.status(401).body("Błędne dane logowania");
    }
}
```

Wypożyczanie gry

Proces wypożyczania obejmuje sprawdzenie dostępności gry, utworzenie rekordu wypożyczenia i aktualizację liczby dostępnych kopii. System automatycznie identyfikuje użytkownika na podstawie tokenu JWT.

```java
@PostMapping("/rent/{gameId}")
@PreAuthorize("hasRole('USER')")  // Tylko dla zalogowanych użytkowników
public ResponseEntity<?> rentGame(@PathVariable Long gameId) {

    // 1. Pobieramy aktualnego użytkownika z kontekstu Security
    String username = SecurityContextHolder.getContext()
        .getAuthentication().getName();
    User user = userRepository.findByUsername(username).orElseThrow();

    // 2. Sprawdzamy czy gra istnieje i czy jest dostępna
    Game game = gameRepository.findById(gameId).orElseThrow();
    if (game.getAvailableCopies() <= 0) {
        return ResponseEntity.badRequest().body("Brak dostępnych kopii");
    }

    // 3. Tworzymy nowe wypożyczenie
    GameRental rental = new GameRental(user, game);

    // 4. Zmniejszamy liczbę dostępnych kopii
    game.setAvailableCopies(game.getAvailableCopies() - 1);

    // 5. Zapisujemy zmiany w bazie danych
    gameRentalRepository.save(rental);
    gameRepository.save(game);

    return ResponseEntity.ok(rental);
}
```

Zalety Spring Data JPA i Spring Security

Spring Data JPA:

- Szybkość rozwoju dzięki automatycznie generowanym implementacjom Repository
- Eliminacja powtarzalnego kodu CRUD
- Query Methods pozwalają tworzyć zapytania z nazw metod

Spring Security + JWT:

- Sprawdzone i bezpieczne rozwiązania uwierzytelniania
- Wysoka wydajność dzięki bezstanowym tokenom JWT
- Świetna skalowalność - brak przechowywania sesji na serwerze

Ogólne rozwiązanie Spring Data JPA z Spring Securtity pozwala na tworzenie aplikacji umożliwiające zarządzanie danymi wraz z obsługą bezpieczeństwa dostępu do nich.
Spring Boot również oferuje automatyczną konfigurację aplikacji, która redukuje do minimum ilość pisanego kodu przy zachowaniu swojej funkcjonalności.