

Miniprojekt bazy danych

Wiktor Smaga, Maksymilian Katolik

4 czerwca 2024

Spis treści

1	Opis funkcjonalności aplikacji	2
2	Model danych	2
2.1	Schemat bazy danych	2
2.2	Modelowanie przy pomocy adnotacji JPA	3
3	Komunikacja Backend - Database	5
3.1	Tworzenie zapytań do bazy	5
3.2	Realizacja zapytań CRUD	6
3.3	Metody optymalizacji zapytań: Paginacja	10
3.4	Metody optymalizacji zapytań: Problem n+1 zapytań	11
4	Transakcje	12
5	Tworzenie raportów	13
6	Testowanie	16

1 Opis funkcjonalności aplikacji

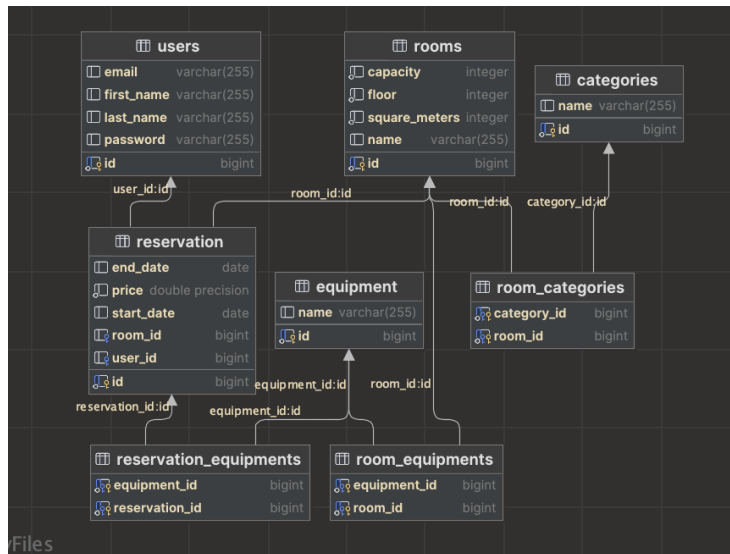
Tematem naszego projektu jest system rezerwacji sal konferencyjnych, zbudowany przy użyciu technologii: Hibernate, Spring Data oraz PostgreSQL. Jest to aplikacja backendowa, która udostępnia endpointy HTTP do wykonywania operacji na bazie danych.

Zostały zrealizowane w nim między innymi:

- zamodelowanie bazy danych przy pomocy JPA
- operacje CRUD na obiektach bazodanowych
- operacje o charakterze transakcyjnym - dokonywanie rezerwacji
- tworzenie złożonych zapytań tworzących raporty
- testowanie oprogramowania poprzez testy jednostkowe, integracyjne i przy pomocy programu Postman

2 Model danych

2.1 Schemat bazy danych



- Główne tabele:
 - **Users**: użytkownicy
 - **Rooms**: informacje o salach

- **Categories:** kategorie sal, każda sala może mieć wiele kategorii, np. sala bankietowa, konferencyjna
- **Reservations:** rezerwacje danych pomieszczeń w danym przedziale czasowym
- **Equipments:** dostępne dodatkowe sprzęty do wypożyczenia wraz z salą
- **Tabele pośrednie:**
 - **reservation equipments:** jakie sprzęty zostały wypożyczone w ramach danej rezerwacji
 - **room equipment:** sprzęty dostępne w ramach danego pomieszczenia
 - **room categories:** kategorie przypisane do sal

2.2 Modelowanie przy pomocy adnotacji JPA

Tworzenie tabel pośrednich odbywa się poprzez stworzenie tabeli ManyToMany wraz z JoinTable, która w joinColumns - określa klucz główny pierwszej strony relacji inverseJoinColumns - określa klucz główny drugiej strony relacji

```

1 Wenszel
@Entity
@Getter
@Setter
@Table(name="Rooms")
public class Room {
    @Id
    private long id;
    private String name;
    @ManyToMany()
    @JoinTable(
        name = "RoomCategories",
        joinColumns = @JoinColumn(name = "room_id"),
        inverseJoinColumns = @JoinColumn(name = "category_id")
    )
    private Set<Category> categories = new HashSet<>();

    4 usages 1 Wenszel
    public void addCategory(Category category) {
        this.categories.add(category);
        category.getRooms().add(this);
    }

    @ManyToMany()
    @JoinTable(
        name = "RoomEquipments",
        joinColumns = @JoinColumn(name = "room_id"),
        inverseJoinColumns = @JoinColumn(name = "equipment_id")
    )
}

```

Rysunek 1: Klasa encji Room

Druga strona relacji przy mapowaniu musi dodać stronę właściciela relacji przy pomocy mappedBy:

```

31 usages  ± Wenszel *
@Entity
@Data
@Table(name="Categories")
public class Category {
    @Id
    private long id;
    private String name;
    @ManyToMany(mappedBy = Room_.CATEGORIES)
    private Set<Room> rooms = new HashSet<>();
}

```

Rysunek 2: Klasa encji Category

```

± Wenszel *
@Entity
@Data
public class Equipment {
    @Id
    private long id;
    private String name;
    @ManyToMany(mappedBy = Reservation_.EQUIPMENTS)
    private Set<Reservation> reservations = new HashSet<>();
    @ManyToMany(mappedBy = Room_.EQUIPMENTS)
    private Set<Room> rooms = new HashSet<>();
    2 usages  ± Wenszel
    public void addRoom(Room room) {
        this.rooms.add(room);
        room.getEquipments().add(this);
    }
}

```

Rysunek 3: Klasa encji Equipment

Mapowanie wiele rezerwacji należy do jednego użytkownika:

```

± Wenszel +1 *
@Entity
@Builder
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Reservation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @ManyToOne(fetch = FetchType.LAZY)
    private Room room;
    private Date startDate;
    private Date endDate;
    @ManyToMany()
    @JoinTable(
        name = "ReservationEquipments",
        joinColumns = @JoinColumn(name = "reservation_id"),
        inverseJoinColumns = @JoinColumn(name = "equipment_id")
    )
    private Set<Equipment> equipments = new HashSet<>();
    7 usages  ± Wenszel
    public void addEquipment(Equipment equipment) {
        this.equipments.add(equipment);
        equipment.getReservations().add(this);
    }
    @ManyToOne(fetch = FetchType.LAZY)
    private User user;
}

```

Rysunek 4: Klasa encji Reservation

Mapowanie jeden użytkownik może mieć wiele rezerwacji:

Rysunek 5: Klasa encji User

```
29 usages  ↕ Wenzel
@Entity
@Data
@Setter
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String email;
    private String password;
    private String firstName;
    private String lastName;

    @OneToMany(mappedBy = "Reservation.USER")
    private Set<Reservation> reservations;
}
```

3 Komunikacja Backend - Database

Operacje CRUD zostały stworzone przy pomocy **interface'u JpaRepository** dostarczanego przez **Spring Data**. Jest on warstwą abstrakcji pomiędzy logiką biznesową, a dostępem do zasobów bazy danych. Wykorzystaliśmy następujące możliwości tej technologii:

3.1 Tworzenie zapytań do bazy

Tworzenie zapytań przy pomocy Spring Data polega na dodaniu do repozytorium metody, którą następnie Spring na podstawie samej jej nazwy interpretuje i generuje jej implementację.

```
9 usages  ↕ Wenzel
@Repository
public interface ReservationRepository
    extends JpaRepository<Reservation, Long> { // JpaRepository extends PagingAndSortingRepository
    no usages  ↕ Wenzel
    List<Reservation> findAllByUserId(Long userId);
    1 usage  ↕ Wenzel
    Page<Reservation> findAllByUserId(Long userId, Pageable pageable);
    1 usage  ↕ Wenzel
    Set<Reservation> findByStartDateBetween(Date startDate, Date endDate);
}
```

Alternatywnym podejściem jest zdefiniowanie przy pomocy adnotacji `@Query` zapytania w języku JPQL

```

3 usages  Wenszel
@Query("SELECT r FROM Room r LEFT JOIN FETCH r.reservations WHERE r.id = :id")
Room findRoomWithReservationsById(Long id);

```

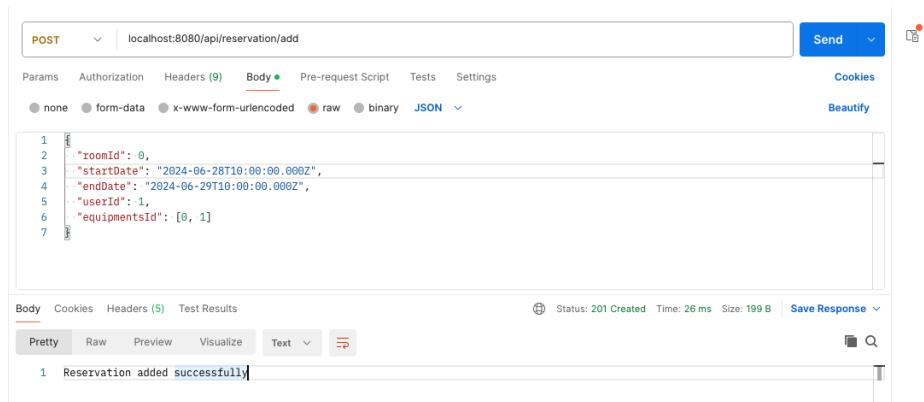
3.2 Realizacja zapytań CRUD

```

6 usages  Wenszel
@Transactional
public void addReservation(Reservation reservation) {
    if (!roomRepository.isRoomAvailable(
        reservation.getRoom().getId(),
        reservation.getStartDate(),
        reservation.getEndDate())) {
        throw new IllegalArgumentException("Room is not available");
    }
    reservationRepository.save(reservation);
}

```

Rysunek 6: Realizacja transakcji dodawania rezerwacji



Rysunek 7: Wykonanie zapytania dodającego rezerwację

```

1 usage  Wenszel
@Transactional(readOnly = true)
public Reservation getReservation(Long id) {
    return reservationRepository.findById(id).orElse(other: null);
}

```

Rysunek 8: Transakcja pobierającą rezerwację po podanym ID

The screenshot shows a REST client interface with the following details:

- URL:** localhost:8080/api/reservation/0
- Method:** GET
- Path:** localhost:8080/api/reservation/4
- Status:** 200 OK, Time: 47 ms, Size: 335 B
- Response Body (JSON):**

```

1 {
2   "id": 4,
3   "startDate": "2024-06-28",
4   "endDate": "2024-06-29",
5   "roomName": "Room 1",
6   "roomId": 0,
7   "price": 120.0,
8   "equipments": [
9     "Whiteboard",
10    "Projector"
11  ],
12   "user": "john.smith@gmail.com"
13 }

```

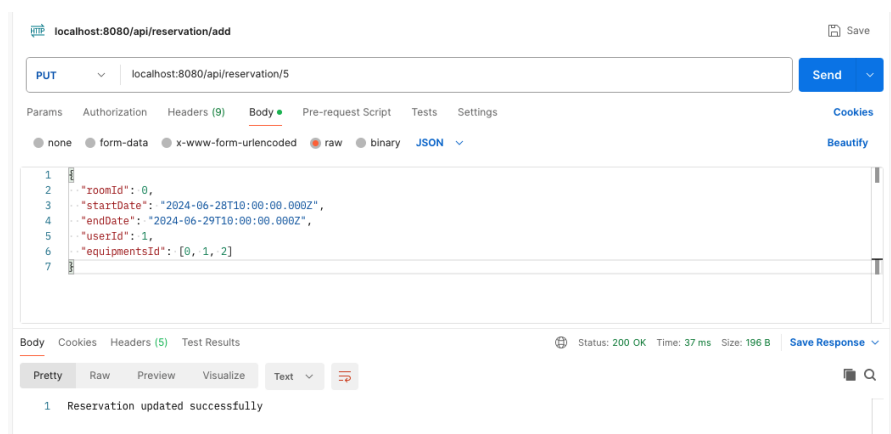
Rysunek 9: Wykonanie zapytania pobierającego wcześniej dodaną rezerwację

Transakcja aktualizacji rezerwacji. Korzystamy tutaj z faktu, że Hibernate w transakcji śledzi pobrane encje (tzw. mechanizm Dirty Checking), dzięki czemu nie musimy sami zapisywać zmian metodą save().

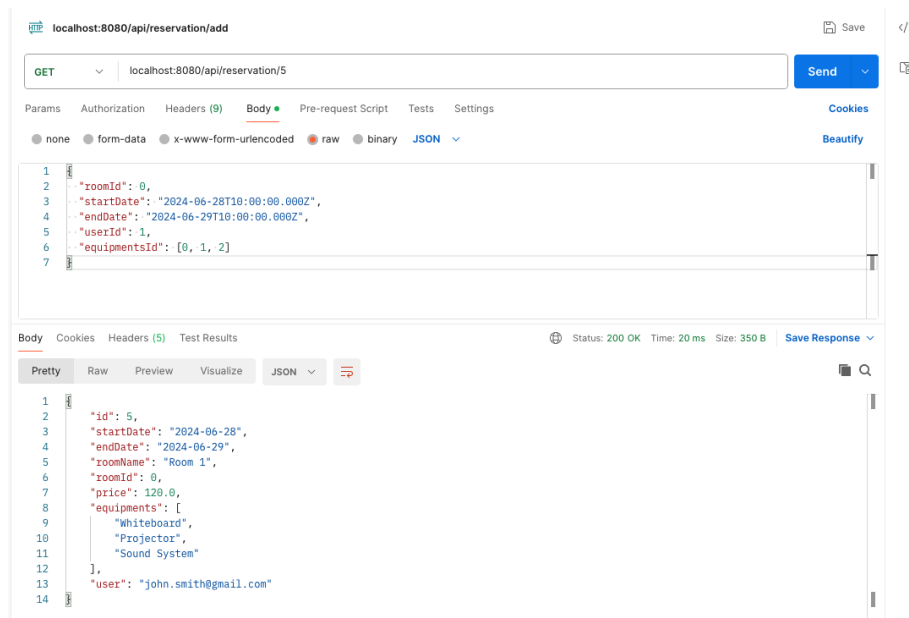
```

1 useage new *
2 @Transactional
3 public void updateReservation(Long reservationId, ReservationRequest reservationRequest) {
4     Reservation reservation = reservationRepository.findById(reservationId)
5         .orElseThrow(() -> new IllegalArgumentException("Reservation not found"));
6
7     Room room = roomRepository.findById(reservationRequest.getRoomId())
8         .orElseThrow(() -> new IllegalArgumentException("Room not found"));
9
10    User user = userRepository.findById(reservationRequest.getUserId())
11        .orElseThrow(() -> new IllegalArgumentException("User not found"));
12
13    Set<Equipment> equipments = new HashSet<>();
14    for (Long equipmentId : reservationRequest.getEquipmentsId()) {
15        Equipment equipment = equipmentRepository.findById(equipmentId)
16            .orElseThrow(() -> new IllegalArgumentException("Equipment not found"));
17        equipments.add(equipment);
18    }
19
20    reservation.setRoom(room);
21    reservation.setUser(user);
22    reservation.setStartDate(reservationRequest.getStartDate());
23    reservation.setEndDate(reservationRequest.getEndDate());
24    reservation.setEquipments(equipments);
25
26    double price = room.getPriceByDay() * reservation.getDays(reservationRequest.getStartDate(), reservationRequest.getEndDate());
27    reservation.setPrice(price);
28 }

```



Rysunek 10: Wykonanie zapytania PUT



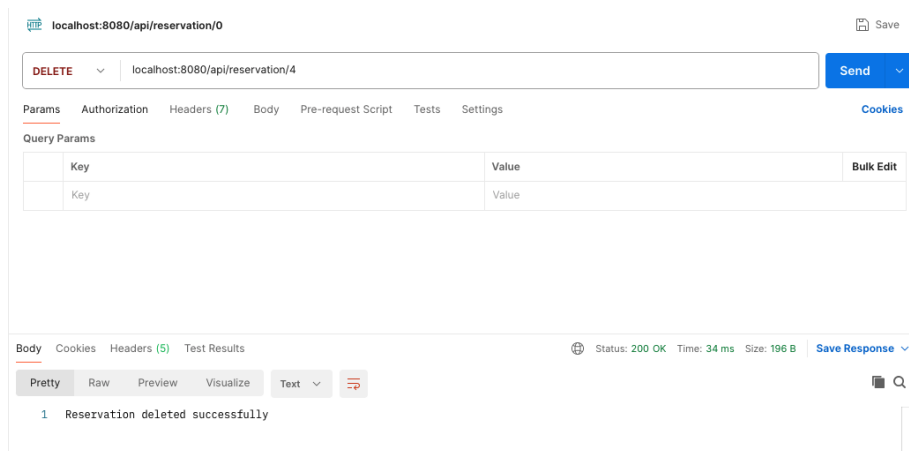
Rysunek 11: Wynik powyższego zapytania - został dodany Equipment o id 2

```

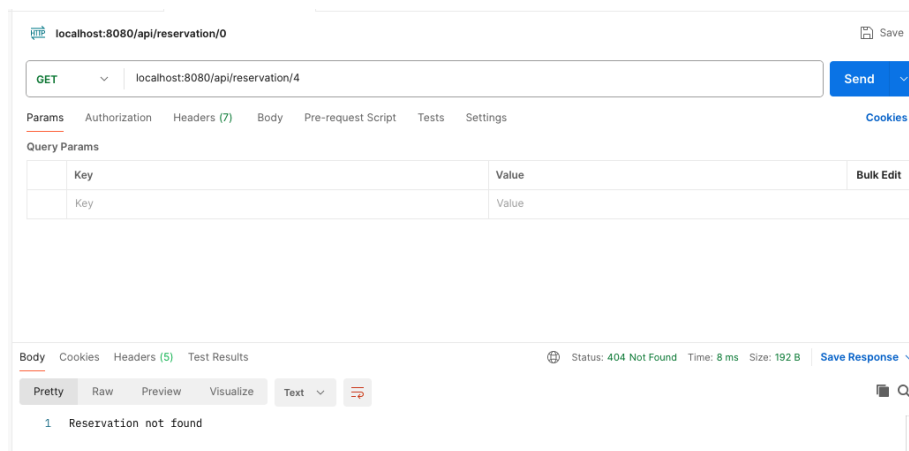
1 usage  Maksymilian-Katolik
@Transactional
public void deleteReservation(long reservationId) {
    if (!reservationRepository.existsById(reservationId)) {
        throw new IllegalArgumentException("Reservation not found");
    }
    reservationRepository.deleteById(reservationId);
}

```

Rysunek 12: Transakcja usuwająca rezerwację z bazy danych



Rysunek 13: Wykonanie zapytania DELETE



Rysunek 14: Wynik powyższego zapytania

3.3 Metody optymalizacji zapytań: Paginacja

Przy pobieraniu rezerwacji dla danego użytkownika zastosowaliśmy mechanizm paginacji dostarczony przez interface `PagingAndSortingRepository`. Pozwala on na pobieranie tylko wybranej ilości rekordów zamiast całej dostępnej zawartości. Jest to użyteczne między innymi do tego, żeby nie doprowadzić do `OutOfMemoryException` przy pobieraniu dużej ilości obiektów do pamięci serwera.

```

9 usages  Wenszel
@Repository
public interface ReservationRepository
    extends JpaRepository<Reservation, Long> { // JpaRepository extends PagingAndSortingRepository
    no usages  Wenszel
    List<Reservation> findAllByUserId(Long userId);
    1 usage  Wenszel
    Page<Reservation> findAllByUserId(Long userId, Pageable pageable);
    1 usage  Wenszel
    Set<Reservation> findByStartDateBetween(Date startDate, Date endDate);
}

```

Rysunek 15: Funkcja realizująca mechanizm paginacji w repository

```

2 usages  Wenszel
@Transactional(readOnly = true)
public Page<Reservation> getUserReservations(long userID, int page, int size) {
    Pageable pageable = PageRequest.of(page, size);
    return reservationRepository.findAllByUserId(userID, pageable);
}

```

Rysunek 16: Transakcja pobierająca część rezerwacji

The screenshot shows a REST client interface with the following details:

- URL:** localhost:8080/api/reservation/add
- Method:** GET
- Query Params:**

Key	Value
userid	2
size	10
page	0
- Status:** 200 OK, Time: 209 ms, Size: 359 B
- Response Body (JSON):**

```

{
  "id": 1,
  "startDate": "2024-05-12",
  "endDate": "2024-05-13",
  "roomName": "Room 5",
  "roomId": 4,
  "price": 200.0,
  "equipment": [
    "Whiteboard",
    "Projector",
    "Laptop",
    "Sound System"
  ],
  "user": "jane.doe@gmail.com"
}

```

Rysunek 17: Wynik zapytania

3.4 Metody optymalizacji zapytań: Problem n+1 zapytań

Lazy Loading

Podstawowym narzędziem konfiguracji pozwalającym radzić sobie z problemem n+1 zapytań jest zdefiniowanie przy mapowaniu właściwości fetch na FetchType-

pe.LAZY. Sprawia to, że dane z bazy danych są pobierane w sposób leniwy, tylko wtedy gdy tych danych potrzebujemy. Domyślnie jest to ustawione dla wszystkich mapowań poza ManyToOne, gdzie trzeba to ustawić manualnie

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```

JOIN FETCH

Przy pobieraniu danych z wielu tabel istotne jest zastosowanie mechanizmu JOIN FETCH, aby wszystkie te dane zostały pobrane w ramach jednego zapytania, zamiast pobierania osobno każdego z powiązanych rekordów.

```
3 usages 1 Wenszel
@Repository
public interface EquipmentRepository extends JpaRepository<Equipment, Long> {
    3 usages 1 Wenszel
    @Query("SELECT e FROM Equipment e LEFT JOIN FETCH e.reservations as r LEFT JOIN FETCH r.room as room LEFT JOIN FETCH room.categories")
    List<Equipment> findEquipmentsWithReservationsAndCategoriesAndRooms();
}
```

Rysunek 18: Przykład użycia mechanizmu join fetch do pobrania danych z wielu tabel jednocześnie

4 Transakcje

```
1 Wenszel +1
@PostMapping("/add")
@Transactional
public ResponseEntity<String> addReservation(@RequestBody ReservationRequest reservationRequest) {
    try {
        Reservation reservation = reservationService.createReservation(reservationRequest);
        reservationService.addReservation(reservation);
        return new ResponseEntity<>("Reservation added successfully", HttpStatus.CREATED);
    } catch (Exception e) {
        logger.error("Failed to add reservation", e);
        return new ResponseEntity<>("Failed to add reservation: " + e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Rysunek 19: Transakcja dodania rezerwacji - Controller

```

1 usage  Maksymilian-Katolik *
@Transactional
public Reservation createReservation(ReservationRequest reservationRequest) {
    Room room = roomRepository.findByIdLocked(reservationRequest.getRoomId())
        .orElseThrow(() -> new IllegalArgumentException("Room not found"));

    User user = userRepository.findById(reservationRequest.getUserId())
        .orElseThrow(() -> new IllegalArgumentException("User not found"));

    Set<Equipment> equipments = new HashSet<>();
    for (Long equipmentId : reservationRequest.getEquipmentsId()) {
        Equipment equipment = equipmentRepository.findById(equipmentId)
            .orElseThrow(() -> new IllegalArgumentException("Equipment not found"));
        equipments.add(equipment);
    }

    Reservation reservation = Reservation.builder()
        .room(room)
        .user(user)
        .startDate(reservationRequest.getStartDate())
        .endDate(reservationRequest.getEndDate())
        .equipments(equipments)
        .build();

    double price = room.getPriceByDay() * reservation.getDays(reservationRequest.getStartDate(), reservationRequest.getEndDate());
    reservation.setPrice(price);

    return reservation;
}

```

Rysunek 20: Metoda tworzenie nowej rezerwacji

W powyższej metodzie używamy do pobrania pokoju funkcji z zastosowanym mechanizmem pessimistic blocking, który blokuje dostęp do tego rekordu aż zakończy się ta transakcja.

```

1 usage  new *
@Query("SELECT r FROM Room r WHERE r.id = :id")
@Lock(LockModeType.PESSIMISTIC_WRITE)
Optional<Room> findByIdLocked(Long id);
}

```

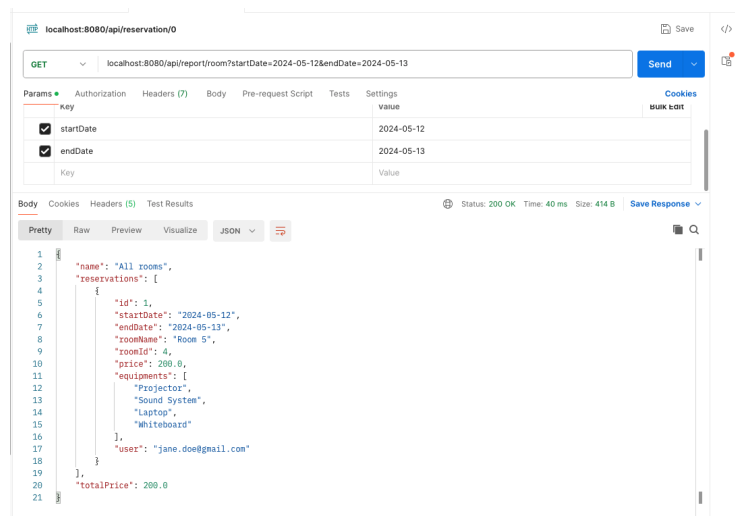
5 Tworzenie raportów

```

1 usage  Wenszel *
@Transactional(readOnly = true)
public ReportResponse getReportBetweenDates(Date startDate, Date endDate) {
    Set<Reservation> reservations = reservationRepository.findByStartDateBetween(startDate, endDate);
    double total = reservations.stream().map(Reservation::getPrice)
        .reduce(Identity::0, Double::sum);
    return new ReportResponse("All rooms",
        reservations.stream().map(ReservationResponse::getReservationResponse).collect(Collectors.toSet()),
        total);
}

```

Rysunek 21: Raport rezerwacji pomiędzy danymi datami, wyświetla wszystkie rezerwacje i sumuje ceny rezerwacji



Rysunek 22: Wywołanie raportu

```

3 usages  A Wenzel
@Transactional(readOnly = true)
public EquipmentReportResponse getEquipmentReport() {
    EquipmentReportResponse response = new EquipmentReportResponse();
    List<Room> rooms = roomRepository.findAll();
    List<Equipment> equipments = equipmentRepository.findEquipmentsWithReservationsAndCategoriesAndRooms();

    equipments.forEach(equipment -> {
        EquipmentReportResponse.EquipmentReport equipmentReport = new EquipmentReportResponse.EquipmentReport();
        equipmentReport.setEquipmentName(equipment.getName());

        Map<Room, Long> numberOfEquipmentReservationsPerRoom = getNumberOfEquipmentReservationsPerRoom(equipment);

        rooms.stream().map(room -> getRoomStats(room, numberOfEquipmentReservationsPerRoom.getOrDefault(room, defaultValues.getRoomUsageStats())))
            .forEach(equipmentReport::addRoomUsageStats);

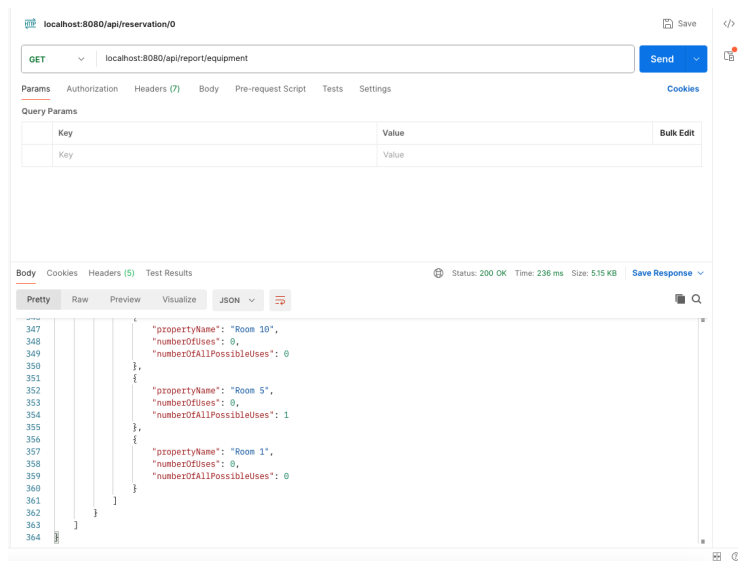
        Map<Category, Long> categoriesUsages = getNumberOfEquipmentReservationsPerCategory(equipment);
        List<Category> categories = equipment.getRooms().stream().map(Room::getCategories).flatMap(Set::stream).toList();
        categories.forEach(category -> {
            if (!categoriesUsages.containsKey(category)) {
                categoriesUsages.put(category, 0L);
            }
        });

        categoriesUsages.entrySet().stream().map(entry -> getCategoryStats(entry.getKey(), entry.getValue(), equipment))
            .forEach(equipmentReport::addCategoryUsageStats);

        response.addEquipmentReport(equipmentReport);
    });
    return response;
}

```

Rysunek 23: Raport dla danego urządzenia. Generuje statystyki ile razy zostało użyte dane urządzenie w danej sali, a ile razy mogło być użyte oraz analogicznie dla danych kategorii sal



Rysunek 24: Wynik raportu urządzenia

6 Testowanie

Testowanie operacji na bazie danych umożliwia nam obiekt `TestEntityManager` oraz adnotacja `@DataJpaTest`, dzięki nim tworzona jest osobna baza danych w pamięci, na której wykonywane są testy.

```
± Wenszel
@DataJpaTest
public class RoomRepositoryTests {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private RoomRepository roomRepository;

    11 usages
    private Room room;

    ± Wenszel
    @BeforeEach
    void setUp() {
        room = new Room();
        entityManager.persist(room);

        Reservation reservation = new Reservation();
        reservation.setRoom(room);
        reservation.setStartDate(Date.valueOf(LocalDate.of( year: 2024, month: 6, dayOfMonth: 1)));
        reservation.setEndDate(Date.valueOf(LocalDate.of( year: 2024, month: 6, dayOfMonth: 10)));
        entityManager.persist(reservation);
    }

    ± Wenszel
    @Test
    void givenNoOverlappingReservation_whenCheckingAvailability_thenRoomIsAvailable() {
        boolean isAvailable = roomRepository.isRoomAvailable(
            room.getId(),
            Date.valueOf(LocalDate.of( year: 2024, month: 6, dayOfMonth: 11)),
            Date.valueOf(LocalDate.of( year: 2024, month: 6, dayOfMonth: 15))
        );
        Assertions.assertTrue(isAvailable);
    }
}
```