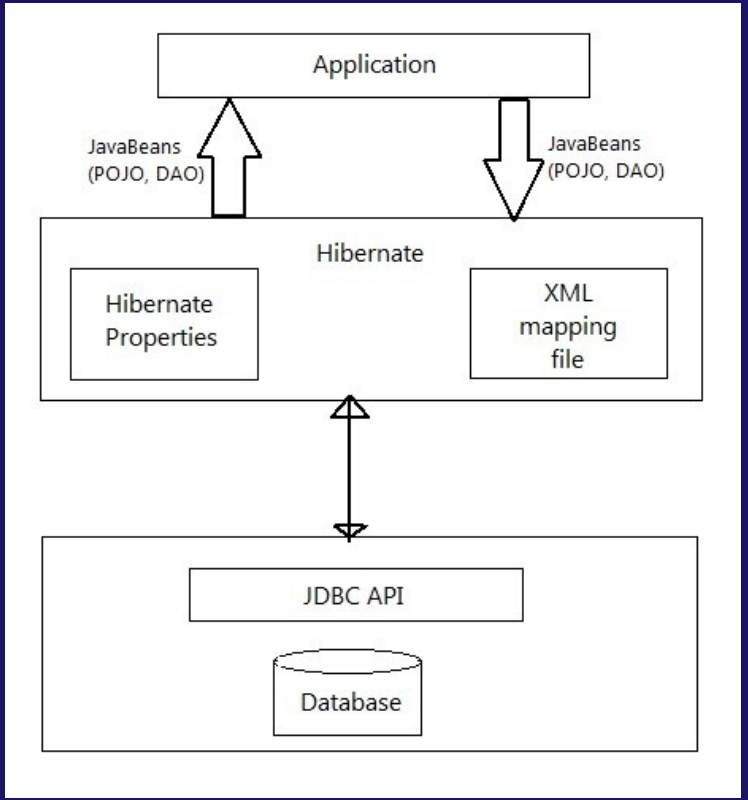
Hibernate

Hibernate jest frameworkiem do mapowania obiektowo-relacyjnego (ORM).

Hibernate jest jedną z implementacji specyfikacji JPA

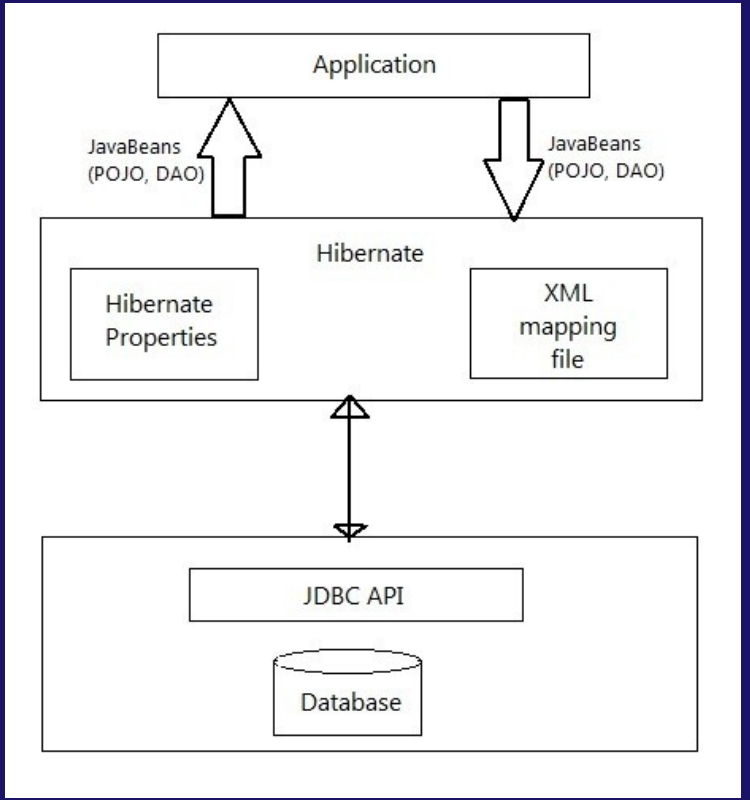
Posiada też własne natywne API specyficzne dla Hibernate’a.



ORM (ang. Object-Relational Mapping)

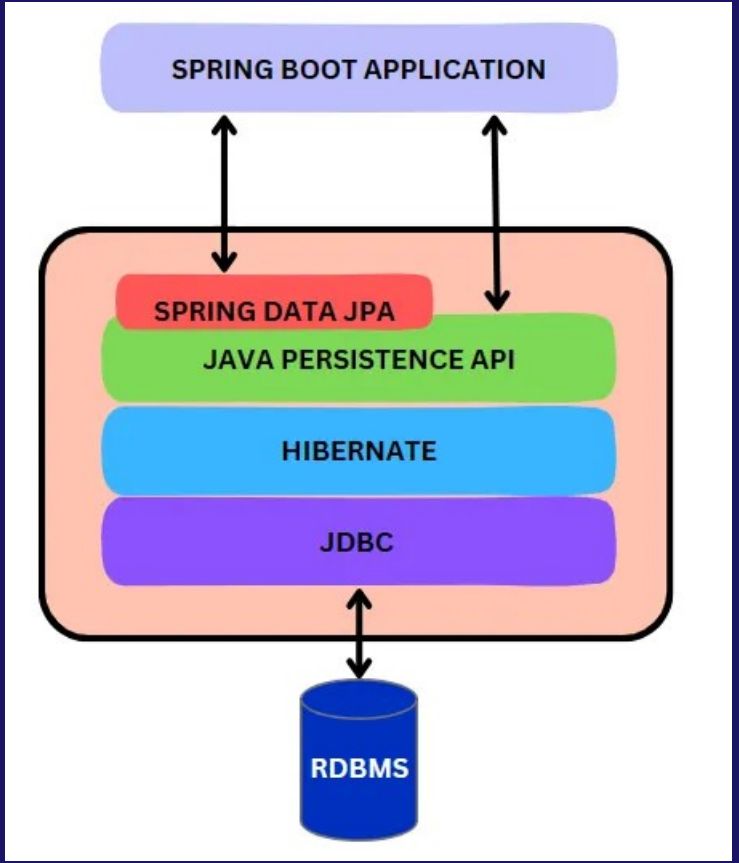
ORM tworzy warstwę między bazą danych a kodem. Pozwala mapować obiekty w kodzie na wiersze w tabelach w bazie danych oraz umożliwia automatyczne wykonywanie operacji, bez konieczności pisania zapytań SQL.

Podczas tworzenia systemu z wykorzystaniem ORM, najpierw należy zdefiniować klasy obiektów, które reprezentują modele danych przechowywanych w bazie danych. Przy użyciu ORM tworzy się mapowanie pomiędzy tymi klasami a tabelami w bazie danych.



JPA

Java Persistence API (JPA) to specyfikacja Java EE i Java SE, która dostarcza standardowy sposób mapowania obiektów na tabele w relacyjnych bazach danych. JPA definiuje sposób zarządzania relacyjnymi danymi w aplikacjach napisanych w języku Java. Obejmuje ona API służące do operacji CRUD (Create, Read, Update, Delete) na danych, zapytań, zarządzania encjami i konfiguracji mapowania obiektowo-relacyjnego (ORM).



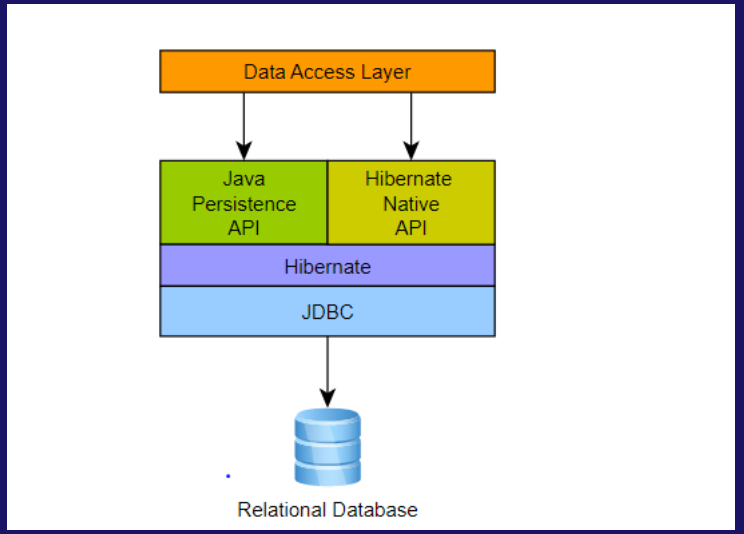
Cechy JPA:

Mapowanie obiektowo-relacyjne (ORM): Umożliwia deweloperom pracę z obiektami w aplikacji zamiast bezpośrednio z tabelami i zapytaniami SQL.

API zarządzania encjami: Zapewnia zestaw operacji do zarządzania cyklem życia encji (obiektów), takich jak ich tworzenie, odczytywanie, aktualizowanie i usuwanie.

Język zapytań JPQL (Java Persistence Query Language): Język zapytań podobny do SQL, ale operujący na obiektach zamiast tabel.

Criteria API: Programistyczny sposób budowania zapytań, który zapewnia bezpieczeństwo typów i łatwiejszą refaktoryzację kodu.



Cd. i również Cechy Hibernate:

Mapowanie Obiektowo-Relacyjne (ORM): Hibernate mapuje obiekty Java na rekordy w tabelach bazy danych, co pozwala na łatwe przekształcanie danych między reprezentacją obiektową (w aplikacji) a relacyjną (w bazie danych).

Abstrakcja i Automatyzacja: Automatyzuje wiele rutynowych zadań związanych z dostępem do danych, takich jak otwieranie połączeń, zarządzanie transakcjami, wykonywanie zapytań i mapowanie wyników zapytań na obiekty.

Nieinwazyjność: Nie wymaga zmian w kodzie klas domenowych (modeli).

Przenośność między bazami danych: Dzięki abstrakcji dostępu do danych, aplikacje napisane z użyciem Hibernate mogą być łatwo przenoszone między różnymi systemami zarządzania bazami danych.

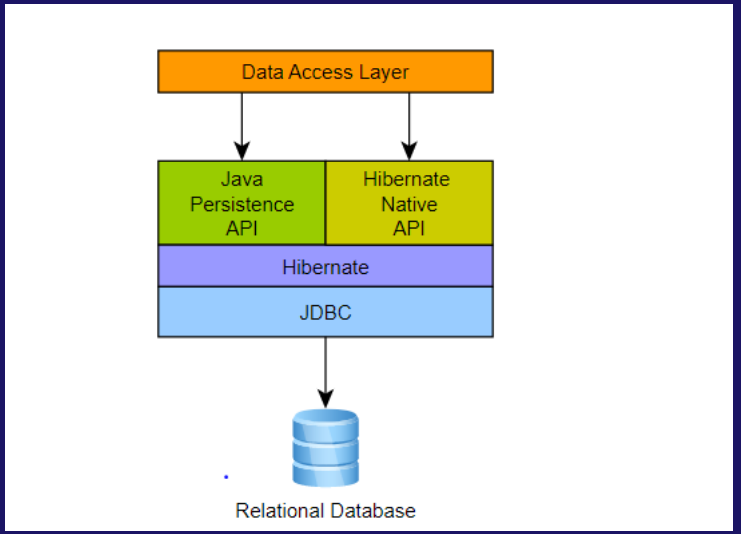
W hibernate Zapytania HQL i Criteria API: Oferuje Hibernate Query Language (HQL), język zapytań oparty na SQL, ale operujący na obiektach zamiast tabel, oraz Criteria API, które pozwala na tworzenie zapytań za pomocą konstrukcji programistycznych.

Hibernate – Tryb natywny

W trybie natywnym używane jest Session i Transaction z Hibernate, podczas gdy JPA używa EntityManager i EntityTransaction.

Kod używający JPA jest bardziej przenośny między różnymi dostawcami ORM, podczas gdy kod trybu natywnego jest ściśle związany z Hibernate.

W JPA tworzy się plik konfiguracyjny persistence.xml
W Hibernate native: hibernate.cfg.xml



Jak to działa?

Hibernate używa plików konfiguracyjnych XML lub adnotacji w klasach Javy do zdefiniowania mapowania między klasami Javy a tabelami bazy danych. Na podstawie tych mapowań Hibernate jest w stanie automatycznie generować odpowiedni SQL do wykonywania operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie) na danych, a także obsługiwać złożone zapytania i relacje między danymi.

Jak to działa w naszym projekcie?

Dodanie hibernate do projektu:

```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.4.Final</version>
</dependency>
```

Jak to działa w naszym projekcie?

Stara metoda!!
Tak nie robimy
W projekcie

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">org.postgresql.Driver</property>
        <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>

        <property name="connection.url">jdbc:postgresql://serwer/baza</property>
        <property name="connection.username">***</property>
        <property name="connection.password">***</property>

        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">validate</property>
        <property name="hibernate.connection.pool_size">10</property>

        <mapping class="org.example.model.Vehicle"/>
        <mapping class="org.example.model.Car"/>
        <mapping class="org.example.model.Motorcycle"/>
        <mapping class="org.example.model.User"/>

    </session-factory>
</hibernate-configuration>
```

NOWA METODA +jsonb

```java
import com.umcsuser.carrent.models.Rental;
import com.umcsuser.carrent.models.User;
import com.umcsuser.carrent.models.Vehicle;
import lombok.Getter;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import io.hypersistence.utils.hibernate.type.json.JsonBinaryType;

public class HibernateConfig {

    @Getter
    private static final SessionFactory sessionFactory;
    static {
        try {
            Configuration configuration = new Configuration();


            configuration.setProperty("hibernate.connection.driver_class", "org.postgresql.Driver");
            configuration.setProperty("hibernate.connection.url", System.getenv("DB_URL"));
            configuration.setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect");
            configuration.setProperty("hibernate.show_sql", "true");
            configuration.setProperty("hibernate.format_sql", "true");
            configuration.setProperty("hibernate.hbm2ddl.auto", "validate");
            configuration.registerTypeOverride(new JsonBinaryType(), new String[]{"jsonb"});


            configuration.addAnnotatedClass(Vehicle.class);
            configuration.addAnnotatedClass(User.class);
            configuration.addAnnotatedClass(Rental.class);

            sessionFactory = configuration.buildSessionFactory();

        } catch (Throwable ex) {
            throw new ExceptionInInitializerError("Inicialize Hibernate ERROR: " + ex);
        }
    }
}
```

SessionFactory

Na podstawie konfiguracji obiekt **SessionFactory** jest odpowiedzialny za inicjalizację konfiguracji Hibernate i za tworzenie obiektów **Session**, przez które odbywa się interakcja z bazą danych.

Inicjalizacja SessionFactory jest procesem relatywnie kosztownym, ponieważ wymaga wczytania całej konfiguracji i mapowań, a także ustanowienia połączenia z bazą danych. Z tego względu zaleca się, aby SessionFactory był tworzony raz na czas działania aplikacji i był wykorzystywany do tworzenia wielu sesji.

Session – reprezentuje pojedynczy wątek komunikacji z bazą danych i jest używana do wykonywania operacji na danych.

Transaction – operacje (zapis/aktualizacja/usunięcie danych), które mają wykonać się razem.

SessionFactory

Poza podstawowymi ustawieniami umożliwimy Hibernateowi konwersję Map<String, Object> do JSONB w PostgreSQL.
Potrzebujemy odpowiednich zależności, oraz adnotacji w kodzie.

```
<!-- https://mvnrepository.com/artifact/io.hypersistence/hypersistence-utils-hibernate-63 -->
<dependency>
    <groupId>io.hypersistence</groupId>
    <artifactId>hypersistence-utils-hibernate-63</artifactId>
    <version>3.9.9</version>
</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.18.3</version>
</dependency>
```

Integracja aplikacji z hibernatem

Przed implementacją metod(lub po) należy się zastanowić nad procesem integracji naszego projektu z Hibernatem. Biorąc pod uwagę posiadania już gotowej aplikacji i gotowej bazy danych nie jest to proces bezproblemowy.

Występujące problemy podczas operacji integracji:

1. Gdzie są klasy obiektów, które mają być zmapowane? (**encje**) i jak są oznaczone?

2. Jakiego typu pola posiadają te klasy, jakiego typu są one w bazie? Na jakiego typu kolumny będą domyślnie konwertowane przez Hibernate?

3. Co z mechanizmem dziedziczenia klas? Jak reprezentować dziedziczenie w tabelach?

4. Jakiego typu relacje mamy w bazie? Jak je określić w klasach javowych?

1. Gdzie są klasy, które mają być zmapowane? (encje) i jak są oznaczone?

org.hibernate.UnknownEntityTypeException: Unable to locate entity descriptor: org.example.model.User

Encja jest to klasa, która jest mapowana na tabelę w bazie danych. Oznaczamy ją adnotacją **@Entity** umieszczoną w danej klasie lub w pliku xml np. User.hbm.xml.
Adnotacja **@Table** została użyta do podania nazwy tabeli w bazie.
Adnotacja @Id jest niezbędna do określenia klucza głównego.
Hibernate potrzebuje dodatkowo **bezparametrowego konstruktora**.

```
@Entity
@Table(name = "users")
public class User {
    @Id
    private String id;

    public User() {}
```

1. Gdzie są klasy, które mają być zmapowane? (encje) i jak są oznaczone?

```java
import lombok.*;
import java.util.Map;
import jakarta.persistence.*;
import io.hypersistence.utils.hibernate.type.json.JsonBinaryType;
import org.hibernate.annotations.Type;
import java.util.HashMap;

@Entity
@Table(name = "vehicle")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Vehicle {

    @Id
    @Column(nullable = false, unique = true)
    private String id;
    @Column(columnDefinition = "NUMERIC")
    private double price;
    private String category;
    private String brand;
    private String model;
    private int year;
    private String plate;
    @Type(JsonBinaryType.class)
    @Column(columnDefinition = "jsonb")
    @Builder.Default
    private Map<String, Object> attributes = new HashMap<>();

    public Object getAttribute(String key) {
        return attributes.get(key);
    }

    public void addAttribute(String key, Object value) {
        attributes.put(key, value);
    }

    public void removeAttribute(String key) {
        attributes.remove(key);
    }
}
```

2. Jakiego typu pola posiadają te klasy, jakiego typu są one w bazie? Na jakiego typu kolumny będą domyślnie konwertowane przez Hibernate?

Schema-validation: wrong column type encountered in column [price] in table [vehicle]; found [numeric (Types#NUMERIC)], but expecting [float(53) (Types#FLOAT)]

Problem z Enumem rozwiązujemy prosto - wpisując jakiego ma być typu, czyli NUMERIC:

```
@Column(columnDefinition = "NUMERIC")
private double price;
```

```
@Type(JsonBinaryType.class)
    @Column(columnDefinition = "jsonb")
    @Builder.Default
    private Map<String, Object> attributes = new HashMap<>();
```

3. Co z mechanizmem dziedziczenia klas? Jak reprezentować dziedziczenie w tabelach? W naszym przypadku JSONB!!!!!

Hibernate potrafi rozwiązać problem dziedziczenia na 3 sposoby:

1. Jedna Tabela na Całą Hierarchię Klas (Single Table Strategy)
2. Tabela na Klasę (Table Per Class Strategy) (w tabeli atrybuty dziedziczone i swoje)
3. Pojedyncza Tabela na Każdą Klasę Konkretną (Joined Strategy) (tylko swoje atrybuty relacja jeden do jednego z tabelą klasy głównej)

Zamiast dziedziczenia użyliśmy kompozycji a dodatkowe parametry przechowujemy w jsonb

4. Jakiego typu relacje mamy w bazie? Jak je określić w klasach javowych?

W przykładowej bazie mamy relację wiele do jednego.
Używając ORM posługujemy się obiektami –

chcąc zmapować relację, należy zrezygnować z pól String dla user_id i vehicle_id w klasie Rental, na rzecz pola Vehicle i pola User !

Każde wypożyczenie (Rental) dotyczy tylko jednego pojazdu (Vehicle).

Ale jeden pojazd (Vehicle) może mieć wiele wypożyczeń.

4. Jakiego typu relacje mamy w bazie? Jak je określić w klasach javowych?

```java
import lombok.*;
import jakarta.persistence.*;

@Entity
@Table(name = "rental")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Rental {

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

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "vehicle_id", nullable = false)
    private Vehicle vehicle;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Column(name = "rent_date", nullable = false)
    private String rentDate;

    @Column(name = "return_date")
    private String returnDate;
}
```

Przykład encji User

```java
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {

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

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

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String role;
}
```

Zarządzanie sesją

W przykładowej aplikacji Serwisy, które używają kilku repo naraz, będą zarządzały sesją – należy wydzielić funkcje do interfejsów i zaimplementować szczególny przypadek Serwisu dla Hibernate'a. JDBC i JSON serwisy mogą korzystać z tych samych serwisów co do tej pory – przeniesionych pod inną nazwą – np. SimpleAuthService itp.

Zarządzanie sesją

```java
public interface RentalService {
    boolean isVehicleRented(String vehicleId);
    Rental rent(String vehicleId, String userId);
    boolean returnRental(String vehicleId, String userId);
    List<Rental> findAll();
}
```

```java
public interface AuthService {
    boolean register(String login, String rawPassword, String role);
    Optional<User> login(String login, String rawPassword);
}
```

```java
public interface VehicleService {
    List<Vehicle> findAll();
    Optional<Vehicle> findById(String id);
    Vehicle save(Vehicle vehicle);
    List<Vehicle> findAvailableVehicles();
    boolean isAvailable(String vehicleId);
}
```

Zarządzanie sesją

```
public class RentalHibernateService implements RentalService {

    private final RentalHibernateRepository rentalRepo;
    private final VehicleHibernateRepository vehicleRepo;
    private final UserHibernateRepository userRepo;

    public RentalHibernateService(RentalHibernateRepository rentalRepo, VehicleHibernateRepository vehicleRepo,
UserHibernateRepository userRepo) {
        this.rentalRepo = rentalRepo;
        this.vehicleRepo = vehicleRepo;
        this.userRepo = userRepo;
    }
```

```
@Override
public boolean isVehicleRented(String vehicleId) {
    try (Session session = HibernateConfig.getSessionFactory().openSession()) {
        rentalRepo.setSession(session);
        return rentalRepo.findByVehicleIdAndReturnDateIsNull(vehicleId).isPresent();
    }
}
```

Zarządzanie sesją

```
@Override
public Rental rent(String vehicleId, String userId) {
    Transaction tx = null;

    try (Session session = HibernateConfig.getSessionFactory().openSession()) {
        tx = session.beginTransaction();

        rentalRepo.setSession(session);
        vehicleRepo.setSession(session);
        userRepo.setSession(session);

        if (rentalRepo.findByVehicleIdAndReturnDateIsNull(vehicleId).isPresent()) {
            throw new IllegalStateException("Vehicle is rented");
        }

        Vehicle vehicle = vehicleRepo.findById(vehicleId)
                .orElseThrow(() -> new RuntimeException("Vehicle not found"));

        User user = userRepo.findById(userId)
                .orElseThrow(() -> new RuntimeException("User not found"));

                Rental rental = Rental.builder()
                        .id(UUID.randomUUID().toString())
                        .vehicle(vehicle)
                        .user(user)
                        .rentDate(LocalDateTime.now().toString()
                        .build();

                 rentalRepo.save(rental);
                 tx.commit();

                 return rental;
            } catch (Exception e) {
                if (tx != null && tx.isActive()) {
                    tx.rollback();
                }
                throw e;
            }
        }
```

Implementacja metod w klasach Repository

```java
public class RentalHibernateRepository implements RentalRepository {

    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }
```

```java
@Override
public List<Rental> findAll() {
    return session.createQuery("FROM Rental", Rental.class).list();
}
```

```java
@Override
public Optional<Rental> findById(String id) {
    return Optional.ofNullable(session.get(Rental.class, id));
}
```

```java
@Override
public Rental save(Rental rental) {
    return session.merge(rental);
}
```

```java
@Override
public void deleteById(String id) {
    Rental rental = session.get(Rental.class, id);
    if (rental != null) {
        session.remove(rental);
    }
}
```

```java
@Override
public Optional<Rental> findByVehicleIdAndReturnDateIsNull
(String vehicleId) {
    Query<Rental> query = session.createQuery("""
            FROM Rental r
            WHERE r.vehicle.id = :vehicleId
            AND r.returnDate IS NULL
            """, Rental.class);
    query.setParameter("vehicleId", vehicleId);
    return query.uniqueResultOptional();
}
```

Podsumowanie funkcji:

createQuery() – tworzy zapytanie.

list() – pobiera wiele wyników.

get() – pobiera pojedynczy obiekt po ID.

merge() – zapisuje lub aktualizuje.

remove() – usuwa obiekt.

setParameter() – wstawia wartość parametru do zapytania.

uniqueResultOptional() – zwraca pojedynczy wynik bezpiecznie (Optional).

Implementacja metod w klasach DAO

Przykład HQL:

```
@Override
public Optional<Rental> findByVehicleIdAndReturnDateIsNull
(String vehicleId) {
    Query<Rental> query = session.createQuery("""
            FROM Rental r
            WHERE r.vehicle.id = :vehicleId
            AND r.returnDate IS NULL
            """, Rental.class);
    query.setParameter("vehicleId", vehicleId);
    return query.uniqueResultOptional();
}
```

Więcej przykładów:

https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm

Dziękuję za uwagę!