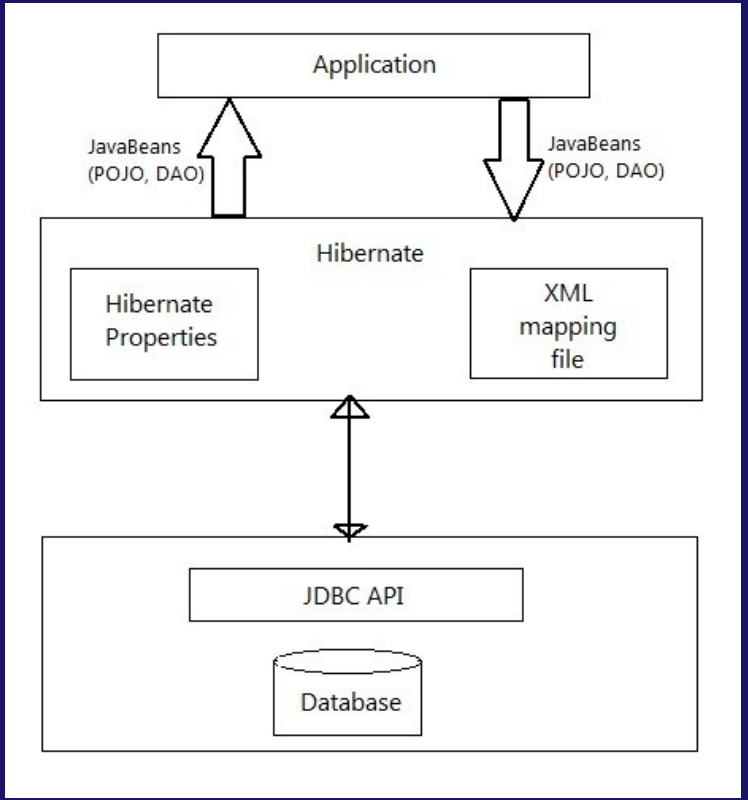
Hibernate

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

Hibernate jest jedną z implementacji specyfikacji JPA

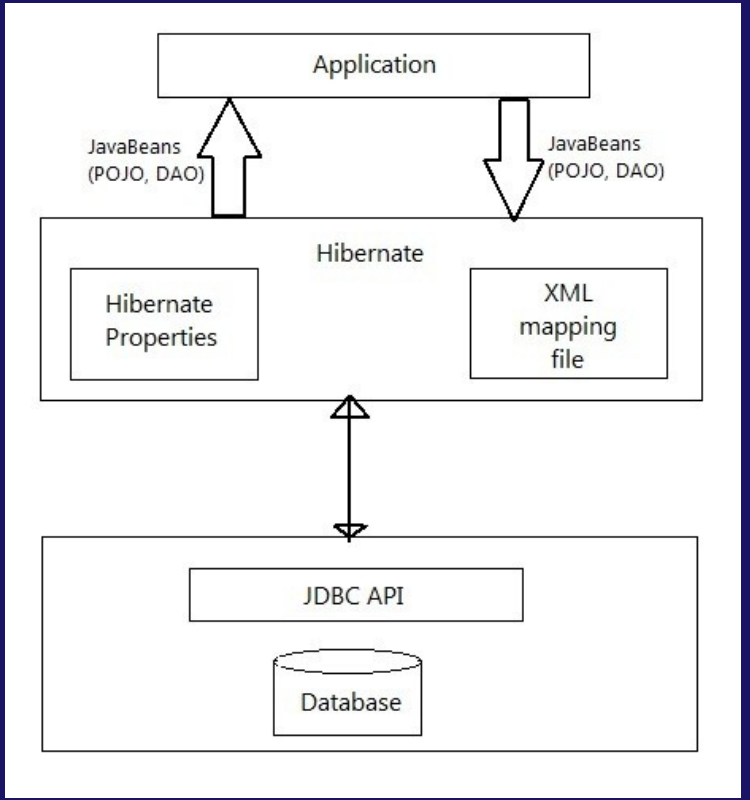
Posiada też własne natywne API specyficzne dla Hibernate’a.



ORM (ang. Object-Relational Mapping)

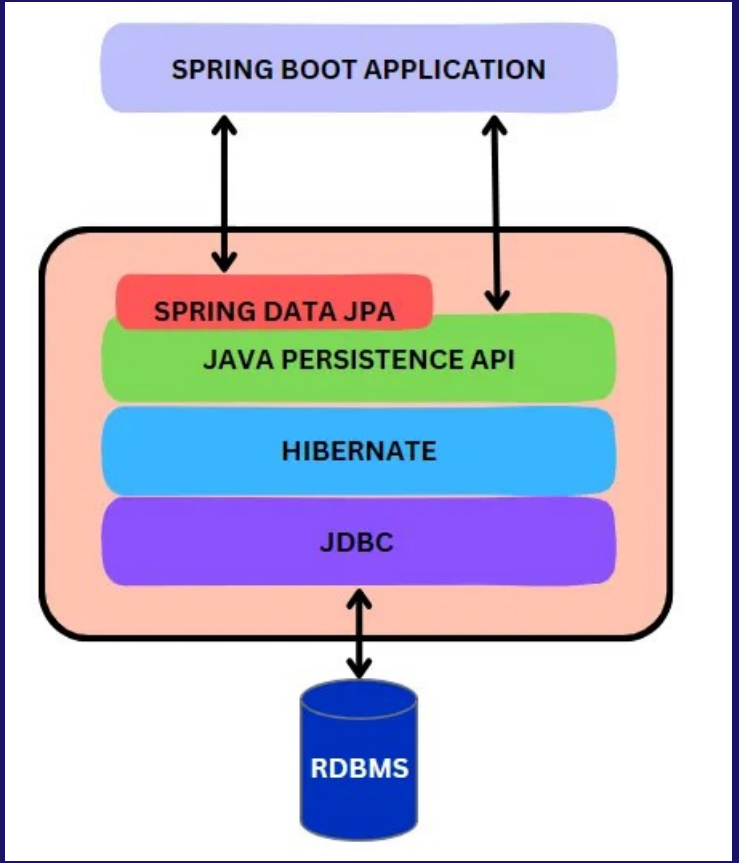
ORM tworzy warstwę między bazą danych a kodem. Pozwala mapować obiekty w kodzie na wiersze w tabelach w bazie danych oraz umożliwia automatyczne wykonywanie operacji, bez konieczności pisania zapytań SQL.

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).



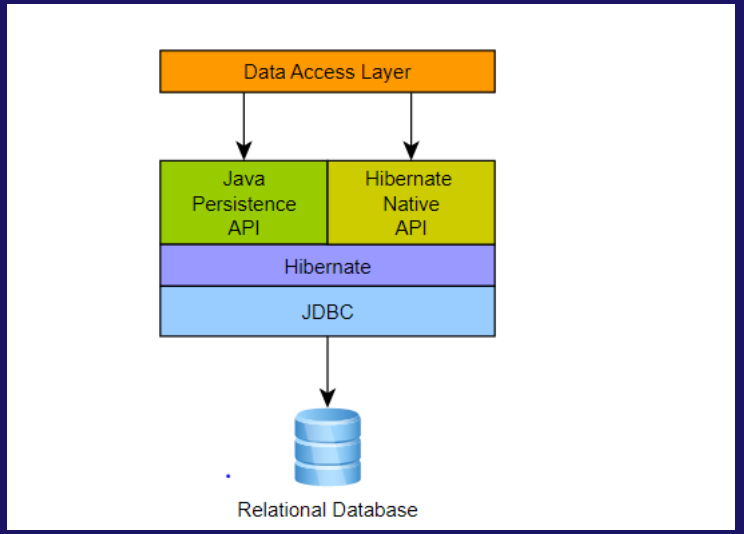
Cechy JPA:

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.

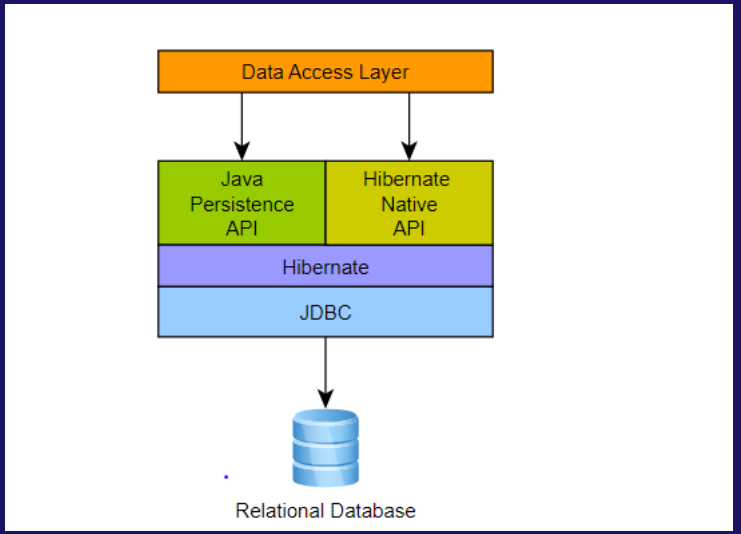
W hibernate Zapytania HQL i Criteria API: Oferuje Hibernate Query Language (HQL), język zapytań oparty na SQL, ale operujący na obiektach zamiast tabel, oraz Criteria API, które pozwala na tworzenie zapytań za pomocą konstrukcji programistycznych.

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?


```xml
<?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>
```

Jak to działa w naszym projekcie?

```xml
<?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:***</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>
```

Plik **hibernate.cfg.xml** to główny plik konfiguracyjny. Zawiera ustawienia niezbędne do inicjalizacji sesji Hibernate, w tym szczegóły dotyczące połączenia z bazą danych, dialektu bazy danych, strategii tworzenia schematu oraz mapowania encji.

SessionFactory

Na podstawie dostarczonego pliku konfiguracyjnego (**hibernate.cfg.xml**) 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.

SessionFactory

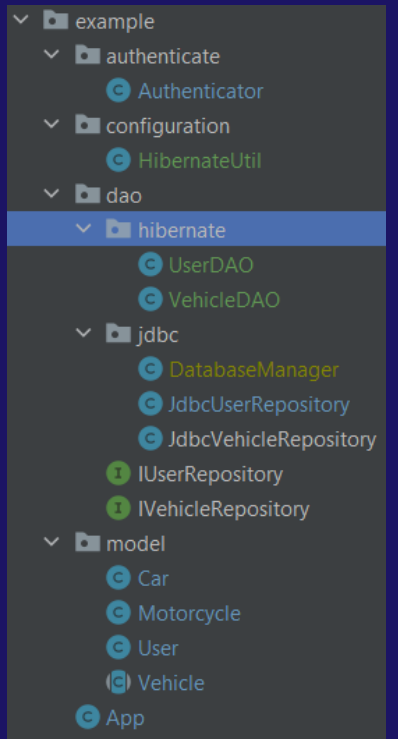
```java
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory;
    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            //konfiguracjia z hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
        }
        return sessionFactory;
    }
}
```

SessionFactory

Można zaimplementować klasy DAO analogicznie do tych z JdbcRepository używając jednej instancji SessionFactory:

```
public class UserDAO implements IUserRepository {
private static UserDAO instance;
SessionFactory sessionFactory;
//TODO: metody do zaimplementowania...
private UserDAO(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
public static UserDAO getInstance(SessionFactory
sessionFactory) {
    if (instance == null) {
        instance = new UserDAO(sessionFactory);
    }
    return instance;
}
}
```



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 = "tuser")
public class User {
    @Id
    private String login;

    public User() {}
```

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 [role] in table [tuser]; found [varchar (Types#VARCHAR)], but expecting [smallint (Types#TINYINT)]

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

```
@Enumerated(EnumType.STRING)
private Role role;
```

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

wrong column type encountered in column [price] in table [tvehicle]; found [numeric (Types#NUMERIC)], but expecting [float(53) (Types#FLOAT)]

Problem price w Vehicle można łatwo rozwiązać podając typ kolumny:

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

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

wrong column type encountered in column [rent] in table [tvehicle]; found [int2 (Types#SMALLINT)], but expecting [boolean (Types#BOOLEAN)]

W tym przypadku jest trudniej – Hiberante oczekuje BOOLEAN a w tabelce mamy SMALLINT.
Jeżeli nie chcemy zmieniać tabeli należy napisać własny konwenter boolean na short ! ! !

Można zmienić w tabeli, np.. w postgresql:

```
ALTER TABLE tvehicle ALTER COLUMN rent DROP DEFAULT;
ALTER TABLE tvehicle ALTER COLUMN rent TYPE BOOLEAN USING CASE WHEN rent = 0 THEN FALSE ELSE TRUE END;
ALTER TABLE tvehicle ALTER COLUMN rent SET DEFAULT FALSE;
```

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

```java
import jakarta.persistence.AttributeConverter;
import jakarta.persistence.Converter;

@Converter
public class BooleanToShortConventer implements
AttributeConverter<Boolean, Short> {
    @Override
    public Short convertToDatabaseColumn(Boolean attribute) {
        if (attribute == null) {
            return null;
        }
        return attribute ? (short) 1 : (short) 0;
    }
    @Override
    public Boolean convertToEntityAttribute(Short value) {
        return value != null && value == 1;
    }
}
```

W trybie natywnym należy dodać adnotacje przy polu, które ma być konwertowane:

```java
@Convert(converter = BooleanToShortConventer.class)
private boolean rent;
```

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

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)

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

1. Jedna Tabela na Całą Hierarchię Klas (Single Table Strategy)

```sql
ALTER TABLE tvehicle
    ADD COLUMN vehicle_type VARCHAR(255)

UPDATE tvehicle
    SET vehicle_type = 'MOTORCYCLE'
    WHERE category IN ('A', 'A1','A2', 'AM');
UPDATE tvehicle
    SET vehicle_type = 'CAR'
    WHERE category IS NULL OR category NOT IN ('A', 'A1','A2', 'AM');
```

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

1. Jedna Tabela na Całą Hierarchię Klas (Single Table Strategy)

```java
@Entity
@Table(name = "tvehicle")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "vehicle_type", discriminatorType = DiscriminatorType.STRING)
public abstract class Vehicle {//..}
```

```java
@Entity
@DiscriminatorValue("CAR")
public class Car extends Vehicle {//..}
```

```java
@Entity
@DiscriminatorValue("MOTORCYCLE")
public class Motorcycle extends Vehicle {//...}
```

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

W przykładowej bazie mamy relację jeden do jednego.
Używając ORM posługujemy się obiektami – **chcąc zmapować relację, należy zrezygnować z pola String plate w klasie User, na rzecz pola Vehicle, oraz dodać pole User w klasie Vehicle !**

W klasie User:

```
@OneToOne(fetch = FetchType.EAGER) //default
@JoinColumn(name = "rentedPlate", referencedColumnName = "plate")
private Vehicle vehicle;
```

W klasie Vehicle:

```
@OneToOne(mappedBy = "vehicle", fetch = FetchType.EAGER)
private User user;
```

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

Przykładowe dane:

```
SELECT v.*, u.login
FROM tvehicle v
LEFT JOIN tuser u ON v.plate = u.rentedplate
ORDER BY login;
```

plate	brand	model	year	price	category	rent	vehicle_type	login
Lu1234	Audi	A4	2021	0.4e3		1	CAR	lukasz
Lu1000	Honda	CBR1000RR-R Fireblade SP	2023	0.5e3	A	0	MOTORCYCLE	
LU3000	BMW	s3	2019	0.3e3		0	CAR	
"PL1234"	audi	a3	2018	0.1e3		0	CAR	
LU80085	BMW	3	2018	0.222e3		0	CAR	

Implementacja metod w klasach DAO

Wczytanie pojazdu z bazy:

```
@Override
public Vehicle getVehicle(String plate) {
    Session session = sessionFactory.openSession();
    try {
        Vehicle vehicle = session.get(Vehicle.class, plate);
        return vehicle;
    } finally {
        session.close();
    }
}
```

Usunięcie usera:

```
@Override
public void removeUser(String login) {
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = session.beginTransaction();
        User user = session.get(User.class, login);
        if (user != null && user.getVehicle()==null) {
            session.remove(user);
        } else {
            return;
        }
        transaction.commit();
    } catch (RuntimeException e) {
        if (transaction != null) transaction.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}
```

Implementacja metod w klasach DAO

Zmiany w obu encjach przy użyciu transakcji – wypożyczenie auta.

```java
@Override
public boolean rentVehicle(String plate, String login) {
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = session.beginTransaction();

        User user = session.get(User.class, login);
        Vehicle vehicle = session.get(Vehicle.class, plate);

        if (user != null && vehicle != null && user.getVehicle() == null) {
            vehicle.setUser(user);
            vehicle.setRent(true);
            user.setVehicle(vehicle);

            session.saveOrUpdate(user);
            session.saveOrUpdate(vehicle);

            transaction.commit();
            return true;
        } else {
            if (transaction != null) {
                transaction.rollback();
            }
            return false;
        }
    } catch (RuntimeException e) {
        if (transaction != null) transaction.rollback();
        throw e;
    } finally {
        session.close();
    }
}
```

Implementacja metod w klasach DAO

Dodawanie nowego usera:

```
@Override
public void addUser(User user) {
        Session session = sessionFactory.openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            session.persist(user);
            transaction.commit();
        } catch (RuntimeException e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        } finally {
            session.close();
        }
}
```

Implementacja metod w klasach DAO

Prosty przykład HQL:

```java
@Override
public Collection<User> getUsers() {
    Collection<User> users;
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = session.beginTransaction();
        users = session.createQuery("FROM User", User.class).getResultList();
        transaction.commit();
    } catch (RuntimeException e) {
        if (transaction != null) transaction.rollback();
        throw e;
    } finally {
        session.close();
    }
    return users;
}
```

Więcej przykładów:

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

Dziękuję za uwagę!