Hibernate i Operacje CRUD

Spis treści

Operacje CRUD

Przypomnijmy sobie, że **CRUD** rozwija się jako (***C**reate*, ***R**ead*, ***U**pdate*, ***D**elete*). Czyli jest to skrót oznaczający operacje tworzenia danych, odczytu danych, aktualizacji danych i usuwania danych. Możemy to podsumować na poniższej grafice:

Obraz 1. Operacje CRUD

Zaczniemy zatem pracę z Hibernate od bardzo prostych przykładów, które będą obrazowały, w jaki sposób można dane tworzyć, pobierać, aktualizować i usuwać.

Przechodzimy do konkretów. Przygotujmy sobie konfigurację Hibernate, możemy to zrobić na dwa sposoby (wybierz ten, który Ci bardziej pasuje).

Sposób 1: plik `hibernate.cfg.xml`;

Przykładowa konfiguracja w pliku `hibernate.cfg.xml`*:*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://localhost:5432/zajavka</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">postgres</property>

        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
```

```
        <property name="hibernate.hbm2ddl.auto">none</property>

        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <mapping class="pl.zajavka.Employee"/>
    </session-factory>
</hibernate-configuration>
```

Konfiguracja Hibernate:

```
package pl.zajavka;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static SessionFactory sessionFactory = null;

    static {
        try {
            loadSessionFactory();
        } catch (Exception ex) {
            System.err.println("Exception while initializing: " + ex);
        }
    }

    private static void loadSessionFactory() {
        Configuration configuration = new Configuration();
        configuration.configure("/META-INF/hibernate.cfg.xml");
        sessionFactory = configuration.buildSessionFactory();
    }

    static void closeSessionFactory() {
        try {
            sessionFactory.close();
        } catch (Throwable ex) {
            System.err.println("Exception while closing SessionFactory: " + ex);
        }
    }

    static Session getSession() {
        try {
            return sessionFactory.openSession();
        } catch (Throwable ex) {
            System.err.println("Exception while getting Session: " + ex);
        }
        return null;
    }
}
```

Sposób 2: tylko klasa `HibernateUtil`:

Klasa HibernateUtil

```
package pl.zajavka;
```

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Environment;
import org.hibernate.service.ServiceRegistry;

import java.util.Map;

public class HibernateUtil {

    private static final Map<String, Object> SETTINGS = Map.ofEntries(
        Map.entry(Environment.DRIVER, "org.postgresql.Driver"),
        Map.entry(Environment.URL, "jdbc:postgresql://localhost:5432/zajavka"),
        Map.entry(Environment.USER, "postgres"),
        Map.entry(Environment.PASS, "postgres"),
        Map.entry(Environment.DIALECT, "org.hibernate.dialect.PostgreSQLDialect"),
        Map.entry(Environment.HBM2DDL_AUTO, "none"),
        Map.entry(Environment.SHOW_SQL, true),
        Map.entry(Environment.FORMAT_SQL, false)
    );

    private static final SessionFactory sessionFactory = loadSessionFactory();

    private static SessionFactory loadSessionFactory() {
        try {
            ServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
                .applySettings(SETTINGS)
                .build();

            Metadata metadata = new MetadataSources(standardRegistry)
                .addAnnotatedClass(Employee.class) ①
                .getMetadataBuilder()
                .build();

            return metadata.getSessionFactoryBuilder().build();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    static void closeSessionFactory() {
        try {
            sessionFactory.close();
        } catch (Throwable ex) {
            System.err.println("Exception while closing SessionFactory: " + ex);
        }
    }

    static Session getSession() {
        try {
            return sessionFactory.openSession();
        } catch (Throwable ex) {
            System.err.println("Exception while getting Session: " + ex);
        }
        return null;
    }
}
```

① Encja `Employee` została dodana w tym miejscu.

DDL tworzący przykładową tabelę `employee`:

```
CREATE TABLE employee
(
    employee_id SERIAL                      NOT NULL,
    name        VARCHAR(20)                 NOT NULL,
    surname     VARCHAR(20)                 NOT NULL,
    salary      NUMERIC(7, 2)               NOT NULL,
    since       TIMESTAMP WITH TIME ZONE NOT NULL,
    PRIMARY KEY (employee_id)
);
```

Mapowanie encji w klasie Employee:

```
package pl.zajavka;

import jakarta.persistence.*;
import lombok.*;

import java.math.BigDecimal;
import java.time.OffsetDateTime;

@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "employee")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "employee_id")
    private Integer employeeId;

    @Column(name = "name")
    private String name;

    @Column(name = "surname")
    private String surname;

    @Column(name = "salary")
    private BigDecimal salary;

    @Column(name = "since")
    private OffsetDateTime since;

}
```

Klasa EmployeeData:

```
package pl.zajavka;

import java.math.BigDecimal;
import java.time.OffsetDateTime;
```

```
class EmployeeData {

    static Employee someEmployee1() {
        return Employee.builder()
            .name("Agnieszka")
            .surname("Pracownik")
            .salary(new BigDecimal("5910.73"))
            .since(OffsetDateTime.now())
            .build();
    }

    static Employee someEmployee2() {
        return Employee.builder()
            .name("Stefan")
            .surname("Nowacki")
            .salary(new BigDecimal("3455.12"))
            .since(OffsetDateTime.now())
            .build();
    }

    static Employee someEmployee3() {
        return Employee.builder()
            .name("Tomasz")
            .surname("Adamski")
            .salary(new BigDecimal("6112.42"))
            .since(OffsetDateTime.now())
            .build();
    }
}
```

Gdy mamy już dodany wspomniany kod, możemy przejść do napisania metod, które pokażą istotę pracy z JPA i ORM.

Przykład dodawania danych do bazy danych

```
package pl.zajavka;

// importy ...

public class EmployeeRepository {

    void insertEmployee(final Employee employee) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            session.persist(employee);
            session.getTransaction().commit();
        }
    }

    // ... pozostałe metody
}
```

Przykład wyświetlenia listy rekordów w bazie oraz pobrania jednego rekordu z bazy

```
package pl.zajavka;
```

```
// importy ...

public class EmployeeRepository {

    List<Employee> listEmployees() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "SELECT emp FROM Employee emp"; ①
            List<Employee> employees = session.createQuery(query, Employee.class).list();
            session.getTransaction().commit();
            return employees;
        }
    }

    Optional<Employee> getEmployee(Integer employeeId) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            return Optional.ofNullable(session.find(Employee.class, employeeId));
        }
    }

    // ... pozostałe metody
}
```

① To nie jest typowy **SQL**. Tutaj użyty został **JPQL**, w którym odwołujemy się do klasy `Employee`. Wrócimy jeszcze do tego.

Przykład aktualizacji danych

```
package pl.zajavka;

// ... importy

public class EmployeeRepository {

    void updateEmployee(Integer employeeId, BigDecimal newSalary) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            Employee employee = session.find(Employee.class, employeeId);
            employee.setSalary(newSalary);
            session.getTransaction().commit();
        }
    }

    // ... pozostałe metody
}
```

Przykład usunięcia pojedynczego rekordu

```
package pl.zajavka;

// ... importy

public class EmployeeRepository {

    void deleteEmployee(Integer employeeId) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            session.remove(session.find(Employee.class, employeeId));
            session.getTransaction().commit();
        }
    }

    // ... pozostałe metody
}
```

Przykład usunięcia wszystkich rekordów w bazie

```
package pl.zajavka;

// ... importy

public class EmployeeRepository {

    void deleteAll() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "select emp from Employee emp";
            session.createQuery(query, Employee.class).list().forEach(session::remove);
            session.getTransaction().commit();
        }
    }

    // ... pozostałe metody
}
```

I na koniec klasa `EmployeeExample`

Klasa EmployeeExample

```
package pl.zajavka;

import java.math.BigDecimal;

public class EmployeeExample {

    public static void main(String[] args) {
        EmployeeRepository employeeRepository = new EmployeeRepository();
```

```
        employeeRepository.deleteAll();

        Employee employee1 = employeeRepository.insertEmployee(EmployeeData.someEmployee1());
        Employee employee2 = employeeRepository.insertEmployee(EmployeeData.someEmployee2());
        Employee employee3 = employeeRepository.insertEmployee(EmployeeData.someEmployee3());

        System.out.println("###Employee 1: "
            + employeeRepository.getEmployee(employee1.getEmployeeId()));
        System.out.println("###Employee 2: "
            + employeeRepository.getEmployee(employee2.getEmployeeId()));

        employeeRepository.updateEmployee(employee3.getEmployeeId(), new BigDecimal("10348.91"));
        System.out.println("###Employee updated: "
                + employeeRepository.getEmployee(employee3.getEmployeeId()));

        employeeRepository.listEmployees()
            .forEach(employee -> System.out.println("###Employee: " + employee));

        employeeRepository.deleteEmployee(employee2.getEmployeeId());

        employeeRepository.listEmployees()
            .forEach(employee -> System.out.println("###Employee: " + employee));

        HibernateUtil.closeSessionFactory();
    }

}
```

To co zostało przedstawione to bardzo podstawowy sposób na pracę z Hibernate. W przykładach tworzyliśmy `SessionFactory`, `Session` oraz `Transaction` manualnie. W późniejszych etapach dołączymy jeszcze do tego Springa i wszystko uprości się w zapisie. Dlatego nie ucz się powyższych przykładów na pamięć, bo niedługo będziemy to pisać inaczej.

Istotą powyższych przykładów jest pokazanie przydatności ORM w pisaniu aplikacji. Zwróć uwagę, że nigdzie nie musieliśmy mapować ręcznie danych z tabeli na pola w klasie. Hibernate robi to za nas. Stosowanie narzędzi tego typu powoduje, że developer może się skupić na realizacji logiki biznesowej, bo nie musi tracić czasu np. na pisanie kodu, który mapuje kolumny na pola w klasie. Funkcjonalność taka jest nam dostarczana przez framework.

Transakcje

Czym są transakcje bazodanowe, jaki jest cel ich istnienia, jaka jest sekwencja ich działania - tego dowiedzieliśmy się już w poprzednim warsztacie. Poruszyliśmy również tematykę tego jak Spring wspiera transakcje.

Mam nadzieję, że udało Ci się oddzielić w głowie Spring od Hibernate. Zauważ, że praktycznie wszystkie przykłady w tym warsztacie są omawiane bez wykorzystania Springa - mamy tylko Hibernate. Warto jest mieć to poukładane w głowie, że oba te rozwiązania najczęściej pracują razem ze sobą, ale są to dwa oddzielne rozwiązania.

Dowiedzieliśmy się również jakie zasady powinna spełniać transakcja oraz jakich anomalii możemy się spodziewać. Generalnie sporo. Teraz skupimy się na moment nad tematem pracy z transakcjami przy wykorzystaniu tylko Hibernate.

Tutaj od razu chcę zaznaczyć, że w ramach ścieżki Zajavka, zaczynamy poznawać pewne zagadnienia od zera, a dopiero potem zaczynamy je ze sobą złączać. Nie inaczej będzie w przypadku Spring + Hibernate. Na razie omówiliśmy podstawy Springa bez Hibernate, przeszliśmy do omówienia Hibernate bez Springa. Przyjdzie też moment, że omówimy jedno i drugie współpracujące ze sobą jednocześnie.

Zaznaczam to dlatego, że poznajemy wiele sposobów na podejście do tych samych kwestii (np. wykonywanie zapytań na bazie danych), ale finalnie część z nich nie będzie wykorzystywana w praktyce. Dużo przykładów jest tłumaczonych, żeby dać Ci obraz tego *co się dzieje pod spodem*. Dlatego poświęcimy chwilę na transakcje przy wykorzystaniu samego Hibernate, ale w praktyce raczej będziemy stosowali poznaną już wcześniej adnotację `@Transactional`. Tylko, że ona jest częścią Springa, a teraz omawiamy Hibernate bez Springa.

Przypomnijmy sobie bardzo krótko, że transakcja bazodanowa jest jednostką pracy, która jest wykonywana na bazie danych. Schemat działania transakcji składa się z trzech podstawowych operacji: `begin`, `commit`, `rollback`.

- **begin** — komenda rozpoczynająca transakcje,
- **commit** — komenda kończąca transakcję i zatwierdzająca wprowadzone zmiany,
- **rollback** — komenda kończąca transakcję i wycofująca wprowadzone zmiany.

Transakcje muszą spełniać zasady **ACID**, czyli niepodzielność, spójność, izolacja i trwałość:

- Niepodzielność - każda transakcja powinna być przeprowadzona w całości. Jeśli jedna część transakcji się nie powiedzie, cała transakcja się nie powiedzie.
- Spójność - Spójność zapewnia, że każda transakcja musi zmienić dane, których dotyczy, tylko w dozwolony sposób.
- Izolacja - Każda transakcja powinna być wykonywana w całkowitej izolacji bez wiedzy o istnieniu innych transakcji.
- Trwałość - Po zakończeniu transakcji zmiany wprowadzone przez transakcję są trwałe (nawet w przypadku wystąpienia nietypowych zdarzeń, takich jak utrata zasilania).

Transakcje w Hibernate

W przypadku Hibernate, transakcje są blisko związane z obiektem `Session`. Dzięki wykorzystaniu obiektu `Session`, możemy rozpocząć i zakończyć działanie transakcji. Transakcja w Hibernate jest reprezentowana przez obiekt `Transaction`, który pozwala nam na wykonanie metod: `begin()`, `commit()` oraz `rollback()`.

W każdym z poprzednich przypadków, zanim wykonaliśmy jakieś operacje na bazie danych, było to poprzedzone rozpoczęciem transakcji:

```
package pl.zajavka;

// ... importy

public class EmployeeRepository {
```

```
    void deleteEmployee(Integer employeeId) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction(); ①
            session.remove(session.find(Employee.class, employeeId));
            session.getTransaction().commit(); ②
        }
    }

    // ... pozostałe metody
}
```

① Tutaj rozpoczynamy transakcję.

② Tutaj commitujemy transakcję.

`Session` jest `Closeable`, zatem po zakończeniu bloku `try-with-resources`, automatycznie zamykane jest otwarte połączenie do bazy danych. Na końcu działania całej aplikacji, czyli na końcu metody `main()`, powinniśmy również zamknąć `SessionFactory` przy wykorzystaniu metody `close()`.

W pokazanym wyżej przykładzie moglibyśmy również złapać wyjątek w klauzuli `catch()`, zalogować go i wykonać `rollback()` ręcznie, ale nie musimy tego robić. Jeżeli zostanie wyrzucony wyjątek, to w tym przypadku zostanie zamknięta cały obiekt `Session` i nie powinien być więcej przez nas używany. Czyli moglibyśmy napisać ten sam kod tak jak poniżej, ale nie musimy, bo Hibernate wycofa transakcję za nas.

```
package pl.zajavka;

// ... importy

public class EmployeeRepository {

    void deleteEmployee(Integer employeeId) {
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            transaction = session.beginTransaction();
            session.remove(session.find(Employee.class, employeeId));
            transaction.commit();
        } catch (Exception e) {
            System.err.println("Exception occurred: " + e.getMessage());
            if (transaction != null) {
                transaction.rollback();
            }
        }
    }

    // ... pozostałe metody
}
```

Kiedy tworzyć obiekt Session

Prawda jest taka, że jak przejdziemy do pracy w zestawie Spring + Hibernate, to nie będziesz tworzyć ani `SessionFactory`, ani `Session` ręcznie. Będzie się tym zajmował framework. Ale zgodnie ze sztuką, należy pamiętać o pewnej istotnej kwestii, która jest wspomniana pod tym źródłem:

> This is an anti-pattern of opening and closing a Session for each database call in a single thread. It is also an anti-pattern in terms of database transactions. Group your database calls into a planned sequence.

Czyli za każdym razem, gdy będziemy chcieli otworzyć `Session` ręcznie, powinniśmy zgrupować więcej wywołań w jedną sekwencję i dopiero wtedy opakować to sesją. Kolejne przykłady będą przykładami edukacyjnymi i niestety często będziemy otwierać `Session` żeby wykonać jedno zapytanie. Mówiąc wprost, gdy będziemy robić to w swojej piaskownicy to nic się nie stanie. Do tego tak jak wspomniałem wcześniej, w praktyce nie będziemy tworzyć `Session` ręcznie, tylko będzie nam w tym pomagał Spring.

Z drugiej strony jeden obiekt `Session` na całą aplikację również jest antywzorcem 😄.

Metody z interfejsu Session

Dotychczas korzystaliśmy z metod `Session#persist()` oraz `Session#remove()`. Przejdziemy teraz do omówienia kilku kolejnych możliwości.

refresh()

Wykonanie metody `refresh()` można rozumieć jak wyciągnięcie wszystkich zmian stanu encji z bazy danych, które zostały wykonane poza bieżącą sesją i po wczytaniu encji.

Metodę tą możemy zastosować gdy będziemy mieli do czynienia z sytuacją, gdzie nasza baza danych będzie modyfikowana przez jakąś inną aplikację i w konsekwencji, encja nad którą pracujemy może nie posiadać najnowszych informacji o swojej reprezentacji w bazie danych. W takim przypadku zastosujemy metodę `refresh()`, która odświeży nam encję najnowszymi danymi z bazy danych.

Metoda ta przeładowuje wartości pól w obiekcie nadpisując je. W praktyce, wykorzystanie tej metody jest potrzebne raczej rzadko.

merge()

Wywołanie metody `merge()` może być rozumiane odwrotnie do metody `refresh()`. Wywołanie metody `merge()` spowoduje zaktualizowanie bazy danych wartościami encji. Metoda ta będzie potrzebna jeżeli będziemy mieli encję w stanie **detached** i chcielibyśmy, żeby przeszła ponownie do stanu **persistent**.

Jaka jest różnica pomiędzy `persist()`, a `merge()`?

Bardzo mocno to upraszczając:

- `persist()` powinno być użyte do encji, które są kompletnie nowe i chcemy je dodać do bazy danych,
- `merge()` powinno być użyte gdy chcemy przywrócić encję do Persistence Context, która została wcześniej odłączona (*detached*).

detach()

Działanie tej metody jest dobrze opisane w dokumentacji:

> Remove this instance from the session cache. Changes to the instance will not be synchronized with the database.

W późniejszych etapach przejdziemy do omówienia tematu Hibernate cache.

Hibernate vs JPA

Wspomniane metody są tylko przykładami. `Session` zapewnia o wiele więcej metod i chciałbym zwrócić tutaj Twoją uwagę na dwie kwestie.

1. Jeżeli korzystasz z Hibernate w wersji większej niż 6, możesz zauważyć, że wiele z dostępnych metod zostało oznaczonych jako `@Deprecated`. Oznacza to, że twórcy chcą nas odwieść od używania tych metod, bo być może w przyszłości mogą one zostać usunięte.
2. Interfejs `Session` rozszerza interfejs `EntityManager`. `Session` jest z Hibernate, a `EntityManager` to "czyste" JPA. Dlaczego jest to istotne? `EntityManager` ma zdefiniowaną pewną grupę metod. Hibernate będący implementacją JPA implementuje te metody. Oprócz tego, Hibernate ma dostępne "swoje własne" metody, które znajdziesz w interfejsie `Session`, ale których nie ma w `EntityManager`. Stosowanie metod z `Session`, których nie ma w `EntityManager` będzie oznaczało, że stosujesz "czystego" Hibernate, a nie JPA. Jeżeli kiedykolwiek przyjdzie taka potrzeba, żeby wymienić dostawcę JPA, to metody z "czystego" Hibernate nie będą się kompilowały. Trzeba je będzie wtedy zastąpić metodami od nowego dostawcy, albo metodami z JPA. Przypadek taki raczej jest rzadki, ale warto rozumieć jaka jest różnica pomiędzy `Session` a `EntityManager`. To samo z resztą będzie miało miejsce w odniesieniu do adnotacji. Możemy stosować "czyste adnotacje Hibernate", albo opierać się o te z JPA, które Hibernate musi również obsługiwać. My w materiałach korzystamy z adnotacji i metod dostarczanych przez JPA.