
# Programowanie aplikacji w Java

## Maciej Gowin

### Zjazd 7 - dzień 1

# Linki

## Opis

https://maciejgowin.github.io/wsb-java/

## Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

# Hibernate: zapytania HQL

HQL (Hibernate QL) jest specyficznym językiem łączącym świat obiektowy ze światem relacyjnych baz danych. Pozwala na odwoływanie się do klas i ich pól w kontekście zapytań znanych z klasycznego języka SQL.

# Hibernate: Query/HQL

Do tej pory w głównej mierze korzystaliśmy z interfejsu `Session` do operacji na obiektach i komunikacji z bazą danych. Interfejs ten umożliwia też wykonywanie operacji za pomocą zapytać HQL wykorzystując interfejs `Query`.

```
Query<Customer> query = session.createQuery("FROM Customer", Customer.class);
```

Co istotne interfejs ten może posłużyć do tworzenia zapytań CRUD i nie ogranicza się tylko do wyszukiwań. Większość operacji ma podobną logikę do tych znanych z klasycznego użycia `Statement` z pakietu `java.sql`.

# Hibernate w praktyce: Query/HQL oraz FROM

Hibernate dostarcza opcji wyszukiwania poprzez interfejs `Query`. Wyszukiwanie to może odbywać się przy pomocy zorientowanego obiektowo języka zapytań HQL oraz klauzuli `FROM`.

```
Query<Customer> query = session.createQuery("FROM Customer", Customer.class);
List<Customer> customers = query.list();
```

# Hibernate w praktyce: Query/HQL oraz FROM

Pobieranie wartości odbywa się poprzez zapytanie `FROM`. Automatycznie pobrane zostaną wszystkie wartości z tabeli oraz rezultat zdeserializowany do obiektu. Istnieje szereg metod do pobrania wartości zwróconych przez zapytanie.

## list()

Pozwala na pobranie elementów do listy. Metoda `getResultList()` odwołuje się do tego samego wywołania.

```
Query<Customer> query = session.createQuery("FROM Customer", Customer.class);
List<Customer> customers = query.list();
```

# Hibernate w praktyce: Query/HQL oraz FROM

## stream()

Pozwala na pobranie elementów do listy w postaci strumienia.

```
Query<Customer> query = session.createQuery("FROM Customer", Customer.class);
List<Customer> customers = query.stream().collect(Collectors.toList());
```

## iterate()

Pozwala na pobranie elementów do listy w postaci iteratora. Powinna być używana jedynie w przypadkach, gdy dane elementy są już załadowane do sesji.

```
Query<Customer> query = session.createQuery("FROM Customer", Customer.class);
Iterator<Customer> customers = query.iterate();
```

# Hibernate w praktyce: Query/HQL oraz FROM

## uniqueResult()

Pozwala na pobranie jednego elementu lub `null` jeżeli element nie został odnaleziony. Metoda `getSingleResult()` odwołuje się do tego samego wywołania.

```
Query<Customer> query = session.createQuery("FROM Customer WHERE lastName = 'Gowin'", Customer.class);
Customer customer = query.uniqueResult();
```

W przypadku, gdy zwrócony zostanie więcej niż jeden rezultat, zgłoszony zostanie błąd.

Dodatkowo istnieje opcja pobrania wartości opcjonalnej poprzez wywołanie `uniqueResultOptional()`.

# Hibernate w praktyce: Query/HQL oraz WHERE

W połączeniu z zapytaniami możemy użyć klauzuli `WHERE`, która znana jest ze zwykłych zapytań SQL. Wartości mogą zostać sparametryzowane.

### setParameter(int, Object);

Parametryzowanie zapytania przy pomocy kolejnych indeksów parametrów.

```
Query<Customer> query = session
    .createQuery("FROM Customer WHERE firstName = ?0", Customer.class)
    .setParameter(0, firstName);
List<Customer> customers = query.list();
```

# Hibernate w praktyce: Query/HQL oraz WHERE

### setParameter(String, Object);

Parametryzowanie zapytania przy pomocy nazwy parametrów.

```
Query<Customer> query = session
    .createQuery("FROM Customer WHERE firstName = :fn", Customer.class)
    .setParameter("fn", firstName);
List<Customer> customers = query.list();
```

> Dla typów String nie dodajemy pojedynczych cudzysłowów podczas przypisywania wartości.

# Programowanie: przykład 59

Wyszukiwanie przy użyciu `FROM` oraz `WHERE` na interfejsie `Query` w praktyce.

# Hibernate w praktyce: Query/HQL oraz SELECT

Podobnie, jak w przypadku standardowego zapytania SQL możemy pobierać konkretne wartości za pomocą klauzuli `SELECT`.

```
List list = session.createQuery("SELECT firstName, lastName FROM Customer").list();
```

Należy zauważyć, że typ zwracany nie jest już równoznaczny typowi z powiązanemu z klauzuli `FROM`. Zwrócona zostaje lista, która przechowuje wartości typu `Object`.

Jeżeli wywołaliśmy zapytanie sparametryzowane, pojawiłby się błąd konwersji.

```
List list = session.createQuery("SELECT firstName, lastName FROM Customer", Customer.class).list();
```

> Zwrócona zostaje lista tablic z elementami typu Object.

# Hibernate w praktyce: Query/HQL oraz SELECT

Rozwiązaniem tego problemu może być wywołanie konstruktora i konwersja explicite do danego typu.

```
List<Customer> list = session.createQuery("SELECT new Customer(lastName) FROM Customer", Customer.class).list();
```

Konwersa może nastąpić do dowolnego typu. Konieczne jest podanie pełnej ścieżki klas, dla typów, które nie zostały wcześniej zarejestrowane.

```
List<LastName> list = session.createQuery("SELECT new LastName(lastName) FROM Customer", LastName.class).list();
```

# Hibernate w praktyce: Query/HQL oraz AS

W zapytaniach HQL możliwe jest używanie aliasów przy użyciu słowa `AS`. Jest to szczególnie przydatne w łączniach, o których będzie mowa w dalszej części.

```
List list = session
        .createQuery("SELECT new Customer(C.lastName) FROM Customer AS C", Customer.class)
        .list();
```

# Hibernate w praktyce: Query/HQL oraz ORDER BY

Sortowanie rezultatów odbywa się przy pomocy klauzuli `ORDER BY`.

```
List<Customer> list = session
        .createQuery("FROM Customer ORDER BY lastName, firstName", Customer.class)
        .list();
```

# Hibernate w praktyce: Query (stronicowanie)

W przypadku konieczności stronicowania elementów wymagającej określenia elementu początkowego oraz ilości elementów koniecznych do pobrania nie musimy klauzuli `LIMIT` znanej z języka `SQL`. Interfejs `Query` dostarcza takiej obsługi poprzez ustawienie tych wartości za pomocą metod pomocniczych: `setFirstResult` oraz `setMaxResults`.

```
Query<Customer> query = session
        .createQuery("FROM Customer", Customer.class)
        .setFirstResult(100)
        .setMaxResults(50);
List<Customer> list = query.list();
```

> Liczenie dla pierwszego elementu rozpoczynamy od 0.

# Hibernate w praktyce: metody agregacyjne w HQL

Podobnie jak SQL, HQL wspiera szereg metod agregacyjnych. Są one analogiczne.

```
Query<Double> query = session.createQuery("SELECT sum(o.value) FROM Order o", Double.class);
Double sum = query.list().get(0);
```

- `sum()` - suma wartości.
- `avg()` - wartość średnia.
- `count()` - ilość wystąpień.
- `max()` - wartość maksymalna.
- `min()` - wartość minimalna.

# Hibernate w praktyce: Query/HQL oraz GROUP BY

Wraz z metodami agregacyjnymi możemy użyć grupowania znanego z SQL.

```
Query query = session.createQuery("SELECT o.type, sum(o.value) FROM Order o GROUP BY o.type");
List list = query.list();
```

# Hibernate w praktyce: Query/HQL oraz CRUD

Do tej pory dodawaliśmy, aktualizowaliśmy oraz usuwaliśmy wartości przy użyciu interfejsu `Session`. Istnieje możliwość wykonania tych operacji na poziomie interfejsu `Query` i metody `executeUpdate()`.

```
Query query = session
        .createQuery("UPDATE Order SET value = :value WHERE id = :id")
        .setParameter("value", 12.6)
        .setParameter("id", 10);
int affected = query.executeUpdate();
```

```
Query query = session
        .createQuery("DELETE FROM Order WHERE id = :id")
        .setParameter("id", 10);
int affected = query.executeUpdate();
```

> Działanie INSERT jest ograniczone i zostało celowo pominięte.

# Hibernate w praktyce: NativeQuery oraz natywny SQL

Hibernate nie ogranicza nas do użycia HQL. Możemy też używać natywnego SQL (ang. native SQL). Do operacji na natywnym SQL używamy metody `createNativeQuery()`, która pozwala stworzyć interfejs `NativeQuery`.

```
NativeQuery query = session.createNativeQuery("select * from customers where first_name = :fn");
```

# Hibernate w praktyce: NativeQuery oraz natywny SQL

Interfejs `NativeQuery` wspiera szereg opcji pozwalających na automatyczną konwersję typów i wartości, jak i pełną parametryzację zapytań.

```
NativeQuery query = session.createNativeQuery("select * from customers where first_name = :fn");
query.addEntity(Customer.class);
query.setParameter("fn", "Maciej");
```

# Programowanie: przykład 60

Opcje wyszukiwanie przy użyciu interfejsu `Query` w praktyce.

# Hibernate w praktyce: Criteria

Hibernate wprowadza programistyczny interfejs pozwalający na definiowanie zapytań.

Wraz z Hibernate 5.2 wsparcie to obejmuje JPA Criteria API.

Obsługa odbywa się przy użyciu interfejsów `CriteriaBuilder` oraz `CriteriaQuery`.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery.select(root);
List<Customer> list = session.createQuery(criteriaQuery).list();
```

# Hibernate w praktyce: Criteria oraz Expressions

Interfejs pozwala na ograniczanie rezultatów poprzez `CriteriaQuery.where()` oraz interfejsy: `CriteriaBuilder` i `Expressions`.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery
        .select(root)
        .where(criteriaBuilder.equal(root.get("firstName"), "Maciej"));
List<Customer> list = session.createQuery(criteriaQuery).list();
```

# Hibernate w praktyce: Criteria oraz Expressions

Możemy tworzyć wiele predykatów opartych na `Expression` oraz danych wartości.

- `lt(), gt(), ...` - mniejsze niż, większe niż, itp

```
criteriaQuery.select(root)
    .where(cb.lt(root.get("value"), 1000));
```

- `like()` - takie jak

```
criteriaQuery.select(root)
    .where(cb.like(root.get("firstName"), "Mac%"));
```

# Hibernate w praktyce: Criteria oraz Expressions

- `between()` - pomiędzy

```
criteriaQuery.select(root)
    .where(cb.between(root.get("itemPrice"), 100, 200));
```

- `in()` - w zbiorze

```
criteriaQuery.select(root)
    .where(root.get("firstName").in("Maciej", "Andrzej", "Adam"));
```

# Hibernate w praktyce: Criteria oraz Expressions

- `isNull()` - czy jest równe `null`

```
criteriaQuery.select(root)
    .where(cb.isNull(root.get("lastName")));
```

- `isNotNull()` - czu nie jest równe `null`

```
criteriaQuery.select(root)
    .where(cb.isNotNull(root.get("itemDescription")));
```

# Hibernate w praktyce: Criteria oraz Expressions

Istnieje możliwość łączenia wyrażeń na podstawie łączenia predykatów przy pomocy: `CriteriaBuilder.and()` oraz `CriteriaBuilder.or()`.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery
        .select(root)
        .where(criteriaBuilder
                .and(
                        criteriaBuilder.equal(root.get("firstName"), "Maciej"),
                        criteriaBuilder.equal(root.get("lastName"), "Gowin")));
List<Customer> list = session.createQuery(criteriaQuery).list();
```

# Hibernate w praktyce: Criteria oraz sortowanie

Sortowanie odbywa się na poziomie `CriteriaQuery.orderBy` za pomocą `CriteriaBuilder.asc()` oraz `CriteriaBuilder.asc()`.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery
        .select(root)
        .orderBy(criteriaBuilder.asc(root.get("lastName")));
List<Customer> list = session.createQuery(criteriaQuery).list();
```

# Hibernate w praktyce: Criteria oraz grupowanie

Grupowanie odbywa się na poziomie `CriteriaQuery.select()` za pomocą metod grupujących np.: `CriteriaBuilder.sum()`.

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Double> criteriaQuery = criteriaBuilder.createQuery(Double.class);
Root<Order> root = criteriaQuery.from(Order.class);
criteriaQuery
        .select(criteriaBuilder.sum(root.get("value")));
List<Double> list = session.createQuery(criteriaQuery).list();
```

# Programowanie: zadanie 29

Za pomocą `Criteria` zapisz przykłady wyszukiwań przedstawione w przykładzie 60.

# Hibernate: relacje

W świecie obiektowym relacje pomiędzy obiektami są definiowane poprzez kompozycje lub też dziedziczenie. W przypadku relacyjnych baz danych definiują je klucze obce. Hibernate pozwala na połączenie tych podejść poprzez konfigurację na poziomie obiektów.

Poznaliśmy już typy relacji, takich jak: one-to-one, many-to-one, one-to-many, many-to-many. Do tej pory realizowaliśmy je przy użyciu relacyjnych baz danych. Hibernate pozwala na ukrycie szczegółów i podejście stricte obiektowe.

> W przykładach będziemy korzystali z konfiguracji przy użyciu adnotacji.

# Hibernate w praktyce: one-to-one

Najprostszym typem relacji jest relacja jeden-do-jednego (ang. one-to-one). W przypadku encji - jednej encji odpowiada dokładnie jedna encja.

Dla przykładu: jeden kraj ma dokładnie jedną stolic oraz jedna stolica może być przypisana tylko do jednego kraju.

# Hibernate w praktyce: one-to-one

```
@Entity
@Table(name = "cities")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @Column(name = "code")
    private String code;

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

    @OneToOne
    private City capital;
}
```

# Hibernate w praktyce: many-to-one

Relacja wiele-do-jednego (ang. many-to-one) to relacja, w której wiele encji może mieć przypisaną tę samą wartość innej encji.

Dla przykładu: wiele miast należy do tego samego kraju.

# Hibernate w praktyce: many-to-one

```
@Entity
@Table(name = "cities")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

    @ManyToOne
    private Country country;
}

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @Column(name = "code")
    private String code;

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

# Hibernate w praktyce: one-to-many

Relacja jeden-do-wielu (ang. many-to-one) to relacja, w której dana encja może mieć przypisanych wiele encji (jest to odwrotność relacji wiele-do-jednego).

Dla przykładu: wiele miast należy do tego samego kraju.

# Hibernate w praktyce: one-to-many

```
@Entity
@Table(name = "countries")
public class Country {
    @Id
    @Column(name = "code")
    private String code;

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

    @OneToMany
    private List<City> cities;
}

@Entity
@Table(name = "cities")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

# Hibernate w praktyce: many-to-many

Relacja wiele-do-wielu (ang. many-to-many) to relacja, w której wiele encji może mieć przypisanych wiele encji.

Dla przykładu: wiele miast należy do wielu krajów.

# Hibernate w praktyce: many-to-many

```
@Entity
@Table(name = "countries")
public class Country {
    @Id
    @Column(name = "code")
    private String code;

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

    @ManyToMany
    private List<City> cities;
}

@Entity
@Table(name = "cities")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

# Programowanie: przykład 61

Przykłady łączeń.

# Hibernate w praktyce: łączenia

Warto zauważyć, że podczas wyszukiwania danych automatycznie dociągane za wartości dla relacji. Hibernate niejawnie dokonuje łączeń znanych z relacyjnych baz danych (`JOIN`). Oczywiście sam HQL również wspiera jawne łączenia, które w niektórych przypadkach mogą być przydatne.

# Hibernate w praktyce: Query/HQL oraz łączenia

HQL wspiera łączenia zapożyczone ze standardy SQL:

- inner join
- left outer join
- right outer join
- full join (rzadko używane)

W zapytaniach możemy używać skróconych nazw łączeń: `join`, `left join` oraz `right join`.

# Hibernate w praktyce: Query/HQL oraz łączenia

Istnieje mała rożnica pomiędzy automatycznym łączeniem danych (implicite) a użyciem łączenia explicite. Podczas użycia łączeń zwrócony dane są zgodne z danymi zwróconymi przez zapytania znane z SQL.

```
session.createQuery("FROM City", City.class);
session.createQuery("SELECT c FROM City c JOIN c.country", City.class);
```

# Programowanie: przykład 62

Porównanie zapytań z oraz bez łączenia `JOIN`.

# Hibernate: mapowanie struktury z dziedziczenia

Wyobraźmy sobie strukturę dziedziczenia.

```
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {};
```

Problemem może być przeniesienie tej struktury na relacyjną bazę danych. Istnieją 3 podejścia rozwiązujące ten problem.

# Hibernate: mapowanie struktury z dziedziczenia

### Table-Per-Type (TPT)

Definiuje tabeli dla każdej z klas. Tabele dla klas pochodnych zawierają tylko elementy dla nich specyficzne.

### Table-Per-Hierarchy (TPH)

Definiuje jedną tabelę dla wszystkich klas. Typ rozpoznawany jest poprzez kolumnę wyróżniającą (dyskryminator).

### Table-Per-Concrete (TPC)

Definiuje tabelę dla każdej z klas. Każda z table posiada wszystkie elementy bez odniesień do innych tabel.

# Hibernate: mapowanie struktury z dziedziczenia

Hibernate wspiera mapowanie struktur dziedziczenia oraz ich opisywanie poprzez adnotacje.

# Programowanie: zadanie 30

Zdefiniuj strukturę bazy dla lotnisk, lotów, klientów oraz rezerwacji przy pomocy Hibernate.