Laboratorium 6 - SOA.
Tematyka: Hibernate - zastosowania zaawansowane

Celem laboratorium jest zaznajomienie z technologią pracy z relacyjnymi bazami danych przy użyciu biblioteki Hibernate. Laboratorium nr 6 jest uzupełnieniem zagadnień poruszanych w laboratorium nr 5. W laboratorium tym nauczyliście się Państwo współpracować z pojedynczą tabela z bazy danych – wykorzystując wbudowane w używane przez państwa środowisko zintegrowane kreatory wygenerowaliście Państwa odpowiednie tabele na podstawie istniejącego obiektu lub w druga stronę na podstawie tabeli stworzyli odpowiadający jej obiekt.

Dzisiejsze laboratorium obejmuje zagadnienia pracy z bardziej złożonymi strukturami – praca z kilkoma tabelami połączonymi różnymi relacjami - one-to –many lub many-to many. Przećwiczymy również różne techniki operowania na danych dostarczane przez bibliotekę Hibernate.

NA początku przypomnienie najważniejszych informacji na temat Hibernate?

Hibernate jest najpopularniejszą biblioteką służącą do mapowania obiektowo-relacyjnego w Javie (***ORM / Object Relational Mapping***). Powstała w 2001 z inicjatywy Gavina Kinga, który w późniejszych latach w dużym stopniu przyczynił się do wprowadzenia istotnych zmian do specyfikacji EJB oraz JPA, doprowadzając je do stanu, w którym znamy je obecnie.

Hibernate jest rozwiązaniem wszystkich problemów związanych z operowaniem na bazie danych z perspektywy uzytkowanika obiektowego. Pozwala on automatycznie mapować obiekty Javy na wiersze w bazie danych oraz odczytywać rekordy z bazy danych i automatycznie tworzyć z nich obiekty. Na dobrą sprawę wykorzystując Hibernate teoretycznie nie musimy mieć większego pojęcia o poprawnym konstruowaniu zapytań w języku SQL, ponieważ będą one budowane za nas.

Obecnie Hibernate to coś dużo bardziej rozbudowanego niż tylko pośrednik pomiędzy Javą a bazą danych. Oprócz podstawowych możliwości ORM znajdziemy tutaj także osobny moduł odpowiedzialny za wyszukiwanie pełnotekstowe w oparciu o silnik Apache Lucene (**Hibernate Search**) oraz rozszerzoną implementację specyfikacji Bean Validation (**Hibernate Validator**).

HQL / JPQL

Hibernate udostępnia 4 podstawowe metody pozwalające wykonywać proste zapytania CRUD na obiektach encji. Oczywiście w realnym świecie takie podstawowe operacje niemal nigdy nie są wystarczające. Z pomocą przychodzi specjalny język zapytań **HQL (Hibernate Query Language)** lub jego ustandaryzowana wersja JPQL (Java Persistence Query Language). Są to języki mocno zbliżone do SQL, jednak nie operujemy w nich na tabelach, a zamiast tego posługujemy się notacją obiektową.

Szczegółowy opis jezyka można znaleźć pod tym linkiem:
https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html

Criteria API

Criteria API jest kolejnym sposobem budowania bardziej zaawansowanych zapytań do bazy. Przypadnie on do gustu wszystkim osobom, które nie przepadają za językiem SQL i jemu podobnymi. Criteria API pozwala wykonać niemal dowolną operację na bazie danych wykorzystując jedynie notację obiektową, tzn. tworząc w programie odpowiednie obiekty i wywołując na nich odpowiednie metody.

Szczegółowy opis jezyka można znaleźć pod tym linkiem:
https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/Criteria.html

Modelowanie relacji złożonych w Hibernate

Relacje one-to-one

Zasadniczo są one najłatwiejsze do zrozumienia zatem od nich zaczniemy. Relacja tego typu zakłada sytuację, w której **dokładnie jednej** encji odpowiada **dokładnie jedna inna** encja. Przykładem z życia jest choćby człowiek i jego numer PESEL – każdy człowiek ma jeden numer pesel, każdy pesel reprezentuje jednego człowieka Inny przykład bohater (hero) w grze oraz smok (dragon). Każdy bohater posiada tylko jednego smoka.

```
@OneToOne

@JoinColumn(name="dragon_id")

private Dragon dragon;
```

Smoka dodajemy do naszego bohatera w ten sposób.

```
Dragon d = new Dragon();

d.setName("Smok Wawelski");

entityManager.persist( d );

hero.setDragon(d);
```

@OneToOne należy do grupy adnotacji, które pojawiają się zawsze przy mapowaniu relacji. Zalecam zapoznanie się z dokumentacją dotyczącą atrybutów tej adnotacji – na przykładzie powyżej pozostawiłem wszystkie wartości domyślne i zasadniczo działa to dobrze.

Drugą istotną adnotacją jest **@JoinColumn**. Zgodnie z nazwą określa ona jaka kolumna (no i z jakimi atrybutami) będzie odwzorowywać relację. W naszym przypadku nasz bohater jest stroną posiadającą (*owning side*) relację. W związku z tym będziemy trzymać w wygenerowanej tabeli z bohaterami referencję do tabeli ze smokami (klucz obcy). W powyższym przykładzie wskazałem

nazwę kolumny – domyślnie jest ona tworzona z nazwy własności, podreślnika + kilka zasad – jak zawsze polecam zapoznanie się z dokumentacją by poznać szczegóły.

Relację, którą pokazałem na powyższym przykładzie możemy nazwać jednostronną (*unidirectional*). Nasz heros wie wszystko o swoim smoku, ale sam smok nie bardzo ma pojęcie o istnieniu naszego bohatera. Dobrze by było, aby coś o nim wiedział. Taką relację (gdzie obie strony wiedzą o sobie) nazywamy *bidirectional*.

```
@OneToOne(mappedBy="dragon")

private Hero rider;
```

Teraz nasz smok również ma wiedzę o drugiej stronie relacji. W przypadku bazy danych nie zmieniło się nic – jednakże nie ma problemu by wykonać następujący kod:

```
Dragon dragon = entityManager.find(Dragon.class, 1L);

Hero rider = dragon.getRider();

System.out.println("Jezdziec nazywa sie " + rider.getName() );
```

Relacje many-to-one i one-to-many

Każdy bohater może mieć kilkanaście sztuk broni (jakiś mieczyk, sztylet, może i łuk dla odmiany). Mamy zatem kilkanaście sztuk konkretnego przedmiotu, które przynależą do jednej encji (naszego bohatera). Podręcznikowy zatem przykład relacji **many-to-one**. W jej przypadku to strona *many* jest 'posiadaczem' relacji, gdyż to w niej będzie zapisany klucz obcy do encji bohatera. Ma to zasadniczo sens – każda sztuka broni trzyma informację o swoim właścicielu. Sam zaś bohater (na poziomie tabeli w bazie danych) nie ma o broni pojęcia. W klasie broni dodajemy zatem taki kod:

```
@ManyToOne

private Hero owner;

// Getterki i setterki pominięte
```

Tym samym każda sztuka broni posiada referencję do swojego posiadacza. Jak już wspomniałem w przypadku takiej relacji to strona obdarzona adnotacją **@ManyToOne** jest stroną posiadającą. Dzięki temu w tabeli **Weapon** będzie składowana kolumna z kluczem obcym do encji bohaterów. Przy domyślnym zachowaniu adnotacji – zostanie ona wygenerowana z nazwy własności encji oraz nazwy kolumny z kluczem głównym w docelowej encji (czyli u nas będzie to kolumna *owner_id* w tabeli **Weapon**). Podobnie jak w przypadku poprzednich relacji możemy sterować tą relacją za pomocą adnotacji **@JoinColumn**. Moglibyśmy zatem nasz kod zmodyfikować by wyglądał w ten sposób:

```
@ManyToOne

@JoinColumn(name="hero_id")

private Hero owner;
```

Co sprawia, że przynajmniej na poziomie bazy danych nasza tabela jest trochę bardziej jednoznaczna i czytelna. W powyższym kodzie powtarzamy jednak sytuację z bohaterem i jego smokiem. Z całą pewnością dobrze by było, aby bohater wiedział jaką broń ma do dyspozycji. Wtedy będzie to niejako odwrócenie relacji **many-to-one**, czyli będziemy mieć do czynienia z relacją typu **one-to-many**. Zmodyfikujemy zatem klasę **Hero**.

```
@OneToMany(mappedBy="owner")

private List weapons;
```

Jak już wspomniałem bohater jest stroną podrzędną w relacji (*inverse side*) –jedynym zatem elementem poza adnotacją **@OneToMany** jest wskazanie na własność, która jest posiadającą relację w klasie **Weapon** (czyli na *owner*). Istnieje jednakże możliwość, aby nie specyfikować atrybutu **mappedBy**. W tym przypadku zajdą zmiany na poziomie bazy danych (przy użyciu tego atrybutu tak naprawdę wszystko pozostaje po staremu) – powstanie tabela łącząca encję bohatera z bronią. Kod encji broni w tym przypadku nie posiadałby w ogóle informacji o właścicielu (należy usunąć własność **owner**), zaś encja bohatera powinna wyglądać tak:

```
@OneToMany

@JoinTable(name="HERO_WEAPON",

    joinColumns=@JoinColumn(name="HERO_ID"),

    inverseJoinColumns=@JoinColumn(name="WEAPON_ID"))

private List weapons;
```

W tym momencie powstaje tabela łącząca o nazwie **HERO_WEAPON** z nazwami kolumn jak podaliśmy w adnotacji **@JoinTable**. Na poziomie bazy danych encje biorące udział w relacji w ogóle nie wiedzą o swym istnieniu (podobnie jak w przypadku relacji **many-to-many** nie posiadają kolumn z kluczami obcymi w tabelach).

Relacje many-to-many

Ten typ relacji jest równie łatwy do zrozumienia co jeden do jednego. Relacja wiele-do-wielu zakłada istnienie **tabeli pośredniczącej**. W tabeli tej występują dwie kolumny – z parami kluczy obcych wskazującymi na encje znajdujące się w innych tabelach. Przykładem takiej relacji są np. bóstwa w grze. Jedno bóstwo może mieć miliony wyznawców, ale też i każdy bohater może jednocześnie wyznawać kilka bóstw. Zaczniemy zatem od modyfikacji naszego bohatera:

```
@ManyToMany

List deities;
```

Jak i również trzeba w klasie bóstwa dorzucić wyznawców:

```
@ManyToMany

List believers;
```

Jeśli uruchomimy kod aplikacji (z *create-drop* w *persistence.xml*), wówczas pojawi się w bazie danych ciekawa sytuacja – dwie tabele łączące! Dlaczego tak? W przypadku relacji wiele-do-wielu nie można (w sensie logicznym) wskazać właściciela tej relacji. Jednakże programista musi dokonać arbitralnego wyboru tej encji, która zostanie potraktowana jako właściciel relacji, zaś tym samym druga strona relacji staje się podrzędną. Załóżmy, że w powyższym przykładzie za posiadającą uznamy encję bóstwa (tak dla odmiany).

```
@ManyToMany

@JoinTable(name="DEITY_HERO",

    joinColumns=@JoinColumn(name="DEITY_ID"),

    inverseJoinColumns=@JoinColumn(name="HERO_ID"))

List believers;
```

Bohater zaś jest elementem ‘podrzędnym’:

```
@ManyToMany(mappedBy="believers")

List deities;
```

Przy powyższym kodzie w bazie danych powstanie tylko jedna tabela łącząca zawierająca klucze obce. Więcej ciekawostek można znaleźć w dokumentacji adnotacji @ManyToMany

Zadanie. W oparciu o Hibernate samodzielnie zbudować aplikacje typu Biblioteka. Model danych powinien składać się z obiektów: Czytelnik, Ksiazka, Autor oraz Wypozyczenia. Czytelnik składa się z imienia, nazwiska. Książka to tytuł, id autora. Wypożyczenia przechowuje informacje o wypożyczonych książkach przez czytelnika. Składa się z id książki, id czytelnika oraz daty wypożyczenia i daty zwrotu. Aplikacja umożliwia podgląd, dodawanie, usuwanie i modyfikacje poszczególnych pozycji katalogu bibliotecznego.

Dodatkowo należy dorobić opcje pozwalające na wyszukiwanie bardziej zaawansowanych kryteriów wyszukiwania np.

1. podaj wszystkich czytelników, którzy pożyczyli książki danego autora (np.Sienkiewicza) w okresie od 1.01.2018 do 1.05.2018

2. Kto przeczytał książkę „kapitan nemo” w podanym okresie czasu

3. Wszystkich autorów których książki pożyczył pan Jan Kowalski (ewentualnie w jakim okresie czasu)

4. Jakiego autora czyta się najwięcej? itp. .

Wykorzystaj wszytki znane Ci metody operowania na danych (między innymi wymienione wyżej JPQL oraz Criteria API)

Jako interfejs użytkownika można użyc tryby tekstowego lub aplikacji Webowej (zalecane).

Jako pomoc możesz skorzystać z tutoriala znajdującego się pod tym adresem:
http://www.mkyong.com/tutorials/hibernate-tutorials

Java Persistence Query Language – Podstawy

Java Persistence Query Language - jest to technologia, pozwalająca na pobieranie danych z bazy danych za pomocą składni przypominającej SQLa, ale niezależnej od dostawcy bazy danych oraz operującej na notacji obiektowej.

Przykład:

```
SELECT h FROM Hero h
```

Co właśnie uczyniliśmy? Pobraliśmy listę wszystkich dostępnych bohaterów z tabeli.

Potrzebujemy tylko imion? Ależ proszę bardzo.

```
SELECT h.name FROM Hero h
```

Wykonując takie zapytanie otrzymamy listę tylko i wyłącznie imion herosów. Potrzebujemy imienia smoka, którego posiada nasz bohater? Problemu nie ma, za pomocą powyższej notacji możemy dostać się do dowolnej własności encji.

```
SELECT h.dragon.name FROM Hero h
```

Do czego może posłużyć nam **JPQL?**. Jeżeli musimy pobrać dane posługując się trochę bardziej wyszukanymi kryteriami niż klucz główny mamy gotowe do zastosowania narzędzie.

Filtrowanie wyników i złączenia

Powyżej używaliśmy prostych zapytań, które pobierały wszystkie rekordy danego typu. Oczywiście możemy filtrować pobierane wyniki.

```
SELECT h FROM Hero h WHERE h.name = 'Jarek'
```

Wspomniałem, że używanie **JPQL** ma tę przewagę nad 'wyciąganiem' encji w tym, iż potrafi zredukować dość istotnie ilość pobieranych danych. Jeżeli chcemy wygenerować prostą tabelkę z danymi dotyczącymi encji nie jest koniecznym pobieranie jej oraz wszystkich jej zależności. Możemy osiągnąć cel w prostszy sposób

```
SELECT h.name, h.level FROM Hero h
```

Dzięki tmu zwrócimy dane tekstowe bez konieczności obciążania bazy oraz łącza przesyłanie niepotrzebnych nam danych. To jedna z największych zalet **JPQL**. Oczywiście jeśli potrzebujemy danych z innych encji będących w relacji z naszą (czyli po prostu na poziomie **SQLa** wykonujemy złączenie) również nie będzie problemu. Załóżmy, że poszukujemy bohatera, jednakże nie znamy jego imienia. Znamy za to imię jego smoka..

```
SELECT h FROM Hero h, Dragon d WHERE d.name = 'Smok Wawelski'
```

Hibernate bez naszego udziału wygeneruje odpowiednie zapytania do bazy danych dzięki czemu będziemy mogli otrzymać poprawny rekord. Możemy skorzystać również z bardziej bezpośredniej składni.

```
SELECT h FROM Hero h JOIN h.dragon d WHERE d.name = 'Smok Wawelski'
```

Nie ma również problemu by skorzystać z oferowanych w **SQL** funkcji agregujących. Dla przykładu wyciągniemy liczbę wszystkich rekordów w tabeli.

```
SELECT COUNT(h) FROM Hero h
```

Dostępne są też standardowe funkcje jak *MAX, MIN* czy *AVG*. Istnieje również możliwość filtrowania wyników grupowania za pomocą instrukcji *HAVING*.
Powyższe przykłady pokazują tylko bardzo podstawową funkcjonalność **JPQL**. Póki co pokazałem podstawy, dzięki którym będzie można zaprezentować w jaki sposób używać **JPQL** z poziomu kodu. Dla niecierpliwych zaś polecam dokumentację JPQL na stronie Oracle, gdyż jej przepisywanie w formie krótkich kawałków kodu niespecjalnie ma sens. Zresztą sam **JPQL** pod kątem używanych słów kluczowych niespecjalnie różni się od czystego **SQL**.

Sposoby implmentacji?

Wrzucanie kolejnych metod do *Entity Managera* by reprezentować obiekt zapytania nie byłoby sensowne (ileż to musiałoby być metod) – dlatego też stworzono dwa interfejsy – Query oraz TypedQuery (wprowadzony w **JPA2**). Wyróżniamy dwa typy zapytań – dynamiczne i predefiniowane (statyczne)

- **dynamiczne** – jest tworzone w czasie działania programu
- **statyczne** – jest definiowane w metadanych encji (za pomocą adnotacji lub XML), przewagą tego typu zapytań jest ich prekompilacja, a tym samym przyspieszenie wykonywania

By fizycznie wykonać zapytanie musimy wrócić do *Entity Managera* – posiada on metodę *createQuery*, której parametrami jest łańcuch tekstowy z zapytaniem (jeśli chcemy stworzyć instancję typu *Query*) oraz obiekt *Class* zwracanego wyniku (jeśli jesteśmy zainteresowani stworzeniem obiektu typu *TypedQuery*). Oto przykład:

```
String jpql = "SELECT h.name FROM Hero h WHERE h.id = 1";

TypedQuery query = entityManager.createQuery(jpql, String.class);

System.out.println("Imie bohatera: " + query.getSingleResult());
```

Wynikiem działania powyższego kodu jest:

Hibernate: select hero0_.name as col_0_0_ from HEROES hero0_ where hero0_.id=1
Imie bohatera: Jarek

Użyta metoda *getSingleResult* jest jedną z wielu zdefiniowanych przez interfejs *Query*. Polecam zapoznanie się z opisem tego interfejsu, gdyż przeklejanie dokumentacji niespecjalnie ma sens. Dla pełności wywodu zajmiemy się teraz zapytaniami statycznymi. Zaleca się ich stosowanie w sytuacjach, w których z góry wiadomo jaka będzie treść zapytania. Jeśli w naszej aplikacji będziemy dość często potrzebować listy bohaterów wraz z imionami ich smoków, wówczas jest to świetna okazja by wprowadzić zapytanie statyczne. We wszystkich wpisach operuję na adnotacjach zatem i tym razem nie będzie inaczej.

```
@NamedQuery(name="getAllHeroesNamesWithDragonNames",

            query="SELECT h.name, d.name FROM Hero h, Dragon d")

@Entity

public class Hero
```

Jak widać podstawą jest adnotacja @NamedQuery. Atrybut *query* jest dość oczywisty, podobnie *name*. Jednakże w związku z tym ostatnim dobrze jest wspomnieć, iż nazwa jest rejestrowana w ramach danego *persistence-unit*. Dlatego też dobrze jest zapewniać ich unikalność by uniknąć ewentualnych konfliktów (np. poprzez stosowanie prefixów). Nie ma problemu by zdefiniować dla encji kilka zapytań, ale wtedy musimy posiłkować się adnotacją @NamedQueries, która przyjmuje tablicę z wartościami.

```
@NamedQueries({

      @NamedQuery(name="getAllHeroesNamesWithDragonNames",

                  query="SELECT h.name, d.name FROM Hero h, Dragon d"),

      @NamedQuery(name="getAllHeroesNamesWithLevelAndDragonNames",
```

```
                query="SELECT h.name, h.level, d.name FROM Hero h, Dragon d")
})
@Entity
public class Hero
```

W jaki sposób używać tego typu zapytań? Dość prosto:

```
TypedQuery<Object[]> query = (TypedQuery<Object[]>)
entityManager.createNamedQuery("getAllHeroesNamesWithLevelAndDragonNames");

List<Object[]> res = query.getResultList();



for( Object[] o : res ) {

     System.out.println("Imie bohatera: " + o[0]  );

     System.out.println("Level bohatera: " + o[1]  );

     System.out.println("Imie smoka: " + o[2]  );

}
```

Sam kod wyciągający wygląda dość brzydko (zauważ, że pobieramy wartości proste, a nie obiekty encji), ale reguła działania *named-queries* powinna być dość czytelna. Po raz kolejny zachęcam do zapoznania się z dokumentacją interfejsów zapytań – jest to kopalnia informacji.

Wykonywanie zapytań – wyniki?

Jak widać było na powyższych listingach rezultatem zwracanym przez zapytanie **JPQL** mogą być:

- typy proste – Stringi, prymitywy czy typy JDBC
- encje
- tablica obiektów
- typy zdefiniowane przez użytkownika (nie encje)!

Z typami prostymi i tablicą obiektów mieliśmy już styczność, z encjami zasadniczo też. W przypadku encji istotnym jednakże jest udzielenie odpowiedzi na pytanie – w jakim stanie otrzymujemy ową encję? W przypadku pobieraniu danych za pomocą metody *find Entity Manager* dostajemy zarządzaną encję. Dokładnie to samo zachowanie będziemy mogli zaobserwować w przypadku rezultatu zapytania **JPQL**. Zwrócona encja jest zarządzalna i ewentualne zmiany tej encji w ramach wciąż istniejącej transakcji zostaną odwzorowane w bazie danych. Jedynym wyjątkiem jest *transaction-scoped Entity Manager* – jeżeli jest użyty poza transakcją wszystkie pobrane encje od razu są pobierane w stanie *detached*. Co prawda możemy je potem podpiąć do kontekstu metodą *merge*, ale należy o tym pamiętać! Z drugiej strony, zwracanie encji w stanie *detached* ma niezaprzeczalną zaletę – o wiele mniej obciąża pamięć. Jeśli zatem jedynym celem działania naszej metody jest pobranie wyników do ich zaprezentowania (tylko i wyłącznie), dobrze byłoby z góry zaplanować metodę pobierającą *beana* jako nie korzystającą z transakcji (za pomocą adnotacji @TransactionAtribute z wartością *NOT_SUPPORTED*).

Criteria API

Możliwość tworzenia zapytań podobnych składniowo do **SQL**, ale z użyciem notacji obiektowej z całą pewnością upraszcza programowanie. Dzisiaj omówię rozwiązanie programistyczne – czyli tworzenie zapytań za pomocą notacji stricte obiektowej z użyciem odpowiednich metod – *criteria queries*.

Podstawy

Standardowo rozpoczniemy najprostszym możliwym przykładem. Oto wyjściowe zapytanie **JPQL**:

```
String jpql = "SELECT h.name FROM Hero h WHERE h.id = :id";

Query query = entityManager.createQuery(jpql, String.class)

                           .setParameter("id",1L);



System.out.println("Imie bohatera: " + query.getSingleResult()  );
```

Oczywiście jest to dość proste zapytanie, które jednakże posiada dość istotną wadę. Otóż o ile możemy użyć w nim parametrów, o tyle (np. w przypadku *NamedQuery*) musimy je określić przed uruchomieniem programu. Co w sytuacji kiedy wykonywane zapytanie zależy od danych dostarczonych przez użytkownika aplikacji? Oczywiście możemy sklejać zapytania **JPQL** bazując na przekazanych danych, ale ani to czytelne, a i o błąd dość łatwo.

Właśnie dla takich sytuacji jak najbardziej przydają się *Criteria Queries*.

By rozpocząć od kodu – oto powyższe zapytanie przepisane z użyciem kryteriów:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Hero> query = cb.createQuery(Hero.class);



Root<Hero> hh = query.from(Hero.class);

query.select(hh)

     .where(cb.equal(hh.get("id"), 1L));



TypedQuery<Hero> tq = entityManager.createQuery(query);

System.out.println("Imie bohatera: " + tq.getSingleResult().getName()  );
```

Kodu jakby więcej, ale jest on o wiele bardziej elastyczny. Podstawowym elementem wykorzystywania kryteriów jest interfejs CriteriaBuilder pobierany z instancji *Entity Managera*. Jest to punkt wejścia dla używania kryteriów. Dzięki metodzie tego interfejsu o nazwie *createQuery* tworzymy obiekt typu CriteriaQuery. To ten obiekt będzie reprezentował wszystkie warunki, które zamierzamy użyć w zapytaniu. Troszeczkę nie do końca oczywiste jest istnienie obiektu typu *Root*. Interfejs Root jest odpowiednikiem zmiennej wskaźnikowej w zapytaniach **JPQL**. Czyli jeśli w zapytaniu **JPQL** zastosujemy taką konstrukcję:

```
SELECT h FROM Hero h
```

To dokładnym odpowiednikiem tej konstrukcji w kryteriach jest właśnie:

```
Root<Hero> hh = query.from(Hero.class);
```

Na chwilę obecną wystarczy takie opisanie tego interfejsu. Dalej w kodzie odwołujemy się do obiektu reprezentującego nasze *criteria query* używając metod tego interfejsu jak i obiektu *CriteriaBuilder*. Nazwy metod są dość czytelne i póki co nie ma potrzeby ich wyjaśniać. Następnie wykorzystując nasz obiekt kryteriów tworzymy znaną już nam instancję *TypedQuery* i wywołujemy metody pobierające wyniki.
Dlaczego w ten sposób?
Przypomnijmy sobie, że w przypadku **JPQL** schemat działania jest taki sam, z tym tylko, że jako parametr przy tworzeniu obiektu zapytania służy łańcuch tekstowy z zapytaniem, a nie obiekt. Zmniejsza to powiązania między obiektami – obiekt kryteriów niesie informację co ma być pobrane z bazy, zaś obiekt *TypedQuery* wie w jaki sposób wykonać owo zapytanie.

Podstawy tworzenia zapytań

Jak widzieliśmy w powyższym przykładzie cała zabawa z kryteriami rozpoczyna się od interfejsu *CriteriaBuilder*. Posiada on trzy metody, których możemy użyć do stworzenia instancji reprezentującej interfejs *CriteriaQuery*:

- **createQuery(Class)** – tworzy instancję generyczną, najczęściej używana, co zresztą zademonstrowałem na poprzednim listingu
- **createQuery()** – to samo co powyżej, ale w wersji zwracającej *Object*
- **createTupleQuery()** – metoda będąca tak naprawdę wywołaniem metody *createQuery(Tuple.class)*. O tym czym jest *Tuple* (krotka) napiszę później

Kiedy posiadamy już obiekt *CriteriaQuery* możemy przystąpić do definiowania naszego zapytania. Podstawowe elementy języka **SQL** oraz tym samym **JPQL** jak słowa kluczowe SELECT, WHERE czy FROM mają swoje odpowiedniki w postaci metod obiektu *CriteriaQuery*. Jednakże zanim do nich przejdziemy zajmiemy się dość istotnym elementem – rdzeniem (*root*) zapytania. W przypadku **JPQL** mieliśmy do czynienia ze zmienną aliasu. Dzięki niej mogliśmy odwoływać się do kolejnych wartości encji, na której pracowaliśmy. W przypadku kryteriów również musimy stworzyć uchwyt, za pomocą którego będziemy mogli ‘dostać się’ do danych. W poprzednim przykładzie służyła do tego następująca konstrukcja:

```
Root<Hero> hh = query.from(Hero.class);
```

Posługujemy się tutaj interfejsem Root – zachęcam do zapoznania się z rodzicami tego interfejsu. Dzięki temu dość łatwo zrozumieć, co jest wykonalne za pomocą tegoż interfejsu). Na

razie trzeba zaznaczyć tylko, iż w zapytaniu możemy użyć kilkunastu tego typu obiektów (co dotyczy zwłaszcza złączeń). Drugim elementem, który można wyróżnić na początku poznawania kryteriów są *path expressions*. W skrócie – jest to 'ścieżka', za pomocą której dochodzimy do interesujących nas własności obiektu. W poniższym zapytaniu:

```
SELECT h FROM Hero WHERE h.id = :id
```

Tego typu wyrażeniem jest fragment **h.id**. Przekładając powyższe zapytanie na kryteria otrzymamy kod, który przedstawiłem na samym początku:

```
Root<Hero> hh = query.from(Hero.class);

query.select(hh)

     .where(cb.equal(hh.get("id"), 1L));
```

Fragment **hh.get("id")** to odpowiednik kropki w zapytaniach **JPQL**.

Klauzula SELECT

Standardowo do pobierania wyniku zapytania służy metoda *select*. Przyjmuje ona jako argument obiekt implementujący interfejs Selection. Do tej pory przekazywaliśmy do tej metody instancję o typie *Root* – w ten sposób informowaliśmy **JPA**, że interesuje nas encja jako wynik zapytania. Jednakże jeśli interesuje nas np. samo imię maga (czyli wartość łańcuchowa), wówczas musimy zastosować następującą konstrukcję:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<String> query = cb.createQuery(String.class);

Root<Hero> hh = query.from(Hero.class);

query.select(hh.<String>get("name"))

     .where(cb.equal(hh.get("id"), 1L));

TypedQuery<String> tq = entityManager.createQuery(query);



System.out.println("Imie bohatera: " + tq.getSingleResult()  );
```

Argument przekazywany do metody *select* musi być kompatybilny z typem zwracanym przez definicję zapytania (*CriteriaQuery*). Dostawca **JPA** nie jest w stanie po samej nazwie wyciąganego parametru rozpoznać jego typu, dlatego też musimy użyć parametryzacji.