Hibernate

Hello!

Rafał Misiak

Java Developer dla Stibo Systems (DK) w Ciklum

slack: @rafalmisiak

rafalmisiak@gmail.com

Materiały

https://github.com/infoshareacademy/jjdd2-materialy-hibernate

1.
ORM

Object Relational Mapping

Czym jest ORM?

Technika polegająca na konwertowaniu danych między relacyjną bazą danych, a językami zorientowanymi na programowanie obiektowe.

Chcąc ręcznie wykonać odwzorowanie relacyjnej bazy danych w kodzie stracilibyśmy za dużo czasu, szczególnie zwracając uwagę na jakość kodu, stabilność rozwiązania jak i jego wydajność.

Czym jest ORM?

ORM to metoda programistyczna która mapuje obiekty w Javie za pośrednictwem relacyjnych encji na strukturę bazy danych.

Wykorzystanie ORM pozwala na uniknięcie standardowego/podstawowego podejścia z użyciem JDBC.

Odpowiedniki w ORM

encja	tabela w bazie
obiekt	wiersz w tabeli
pole	kolumna w bazie

Zalety
ORM

- Usprawnienia wytwarzania
  - zaawansowane zorientowane obiektowo API
  - mniej kodu do wytworzenia
  - możliwość uniknięcia tworzenia SQL

- Usprawnienia wydajności
  - zaawansowane cache'owanie
  - lazy loading
  - eager loading

Zalety
ORM

- Ułatwiony proces utrzymania
  - mniej kodu do wytworzenia

- Przenaszalność/uniwersalizm
  - framework ORM generując SQL'a za nas dostosowuje go odpowiednio do wybranego silnika baz danych

Czym jest CRUD?

- CREATE – INSERT – tworzenie nowego rekordu
- READ – SELECT – odczyt istniejących rekordów
- UPDATE – UPDATE – zmiana istniejących rekordów
- DELETE – DELETE – kasowanie istniejących rekordów

2.
JPA

Java Persistence API

Czym jest JPA?

Jest to kolekcja interfejsów, klas, metod do obsługi przetwarzania danych względem bazy danych.

Implementacja API pozwala deweloperom na implementację wszystkich operacji CRUD w sposób niezależny od wybranej bazy danych.

Komponenty
JPA

- Entities – odwzorowanie tabeli w bazie danych w postaci klasy.
- Object-relational metadata – odwzorowanie kolumn tabeli w bazie danych w postaci pól klasy encji. Realizowane za pomocą adnotacji.
- Java Persistence Query Language (JPQL) – abstrakcyjny język zapytań będący zamienną formą dla SQL'a. Translacja z JPQL do SQL pozwala na swobodne stosowanie różnych silników baz danych.

JPA
EntityManager

Interfejs zwierający metody odpowiedzialne za wykonywanie operacji w pamięci trwałej wykorzystując do tego obiekty.

W Javie EE inicjalizacja EntityManager'a odbywa się za pośrednictwem adnotacji:
@PersistenceContext

Konfiguracja
persistence.xml

Operacje związane z JPA realizowane są za pośrednictwem **EntityManager'a**. Aby otrzymać jego instancję tworzymy obiekt **EntityManagerFactory**.

EntityManagerFactory jest bezpośrednio powiązany z persistence-unit opisanym w pliku **persistence.xml**

Operacje te realizuje za nas kontener aplikacji.

Lokalizacja pliku persistence.xml

Dla projektów mavenowych lokalizacją pliku persistence.xml będzie:

src/main/resources/META-INF

Przykład
persistence.xml

```
<persistence-unit name="puForJTA" transaction-type="JTA">
    <description>
        This is persistence unit for JTA workshops example.
    </description>
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>java:/MySqlDS</jta-data-source>

    <properties>
        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
    </properties>
</persistence-unit>
```

Opis pliku
persistence.xml

- **provider** – wybieramy dostawcę implementacji dla JPA. W naszym przypadku będzie to Hibernate
- **jdbc.driver** – wskazujemy ścieżkę do sterownika, który będzie obsługiwał wybrany przez nas typ bazy danych. W naszym przypadku będzie to mysql
- **jdbc.url** –ścieżka do schamatu bazy danych, zawiera również typ, host i port serwera bazy

3.
Hibernate

Framework dostępu do warstwy baz danych

Użycie (RESOURCE_LOCAL)
EntityManager

```
EntityManagerFactory  factory =
Persistence.createEntityManagerFactory("puForResourceLocal");

EntityManager entityManager = factory.createEntityManager();

entityManager.persist(entity);
```

Adnotacje
Persistence

@Entity – określa klasę jako encję
@Table – określa nazwę tabeli w bazie, będącej odwzorowaną klasą
@Basic – opisuje pole które ma zostać utrwalone w bazie, pozwala na zdefiniowanie FetchType
@Embedded – możliwość użycia klasy wewnątrz encji z zamianą nazw kolumn na nazwy pola tej klasy
@Id – opisuje pole będące kluczem głównym tabeli
@GeneratedValue – opisuje sposób generowania danych dla pola
@Transient – wartość nieprzechowywana w bazie danych
@Column – opisuje odwzorowanie pola na kolumnę w bazie
@SequenceGenerator – generuje sekwencyjnie dane opisane przez @GeneratedValue
@TableGenerator – umożliwia skonfigurowanie inkrementacji ID dla rekordów

Adnotacje
Persistence

@Access – określa w jaki sposób JPA ma się odwoływać do właściwości encji (FIELD – bezpośredni get/set, PROPERTY – użycie getterów i setterów
@JoinColumn – opisuje kolumnę, która stanowi klucz obcy do innej tabeli, wskazuje encję, która jest właścicielem relacji
@UniqueConstraint – definiuje unikalne wartości na poziomie @Table
@ManyToMany – relacja wiele-do-wielu
@ManyToOne – relacja wiele-do-jednego
@OneToMany – relacja jeden-do-wielu
@OneToOne – relacja jeden-do-jednego
@NamedQueries - oznaczenie listy nazwanych zapytań
@NamedQuery – zdefiniowane, nazwane zapytanie
@AttributeOverride(s) – stosowane przy @Embedded jako mapowanie pól klasy na kolumny z bazy

Adnotacje
@Embedded / @Embeddable

```
@Embeddable
public class MovieShowing {
    java.util.Date startDate;
    java.util.Date endDate;
    ...
}

============================================================

@Embedded
@AttributeOverrides({
   @AttributeOverride(name="startDate", column=@Column(name=„MOVIE_START")),
   @AttributeOverride(name="endDate", column=@Column(name=„MOVIE_END"))
})
public MovieShowing getMovieShowing() { ... }
```

Adnotacje
@UniqueConstraint

```
@Entity
@Table(
   name="EMP",
   uniqueConstraints={@UniqueConstraint(columnNames={„C_NAME", „C_VALUE"})}
)
public class Employee implements Serializable {
   ...
}
```

Ćwiczenie
Database Schema

Utwórz nową bazę danych wraz z nowym, dedykowanym użytkownikiem mającym uprawnienia operować na tej bazie.

Ćwiczenie
Entity: Address

Przygotuj pełną obsługę CRUD dla nowej encji Address.
Utwórz odpowiednią tabelę w bazie.

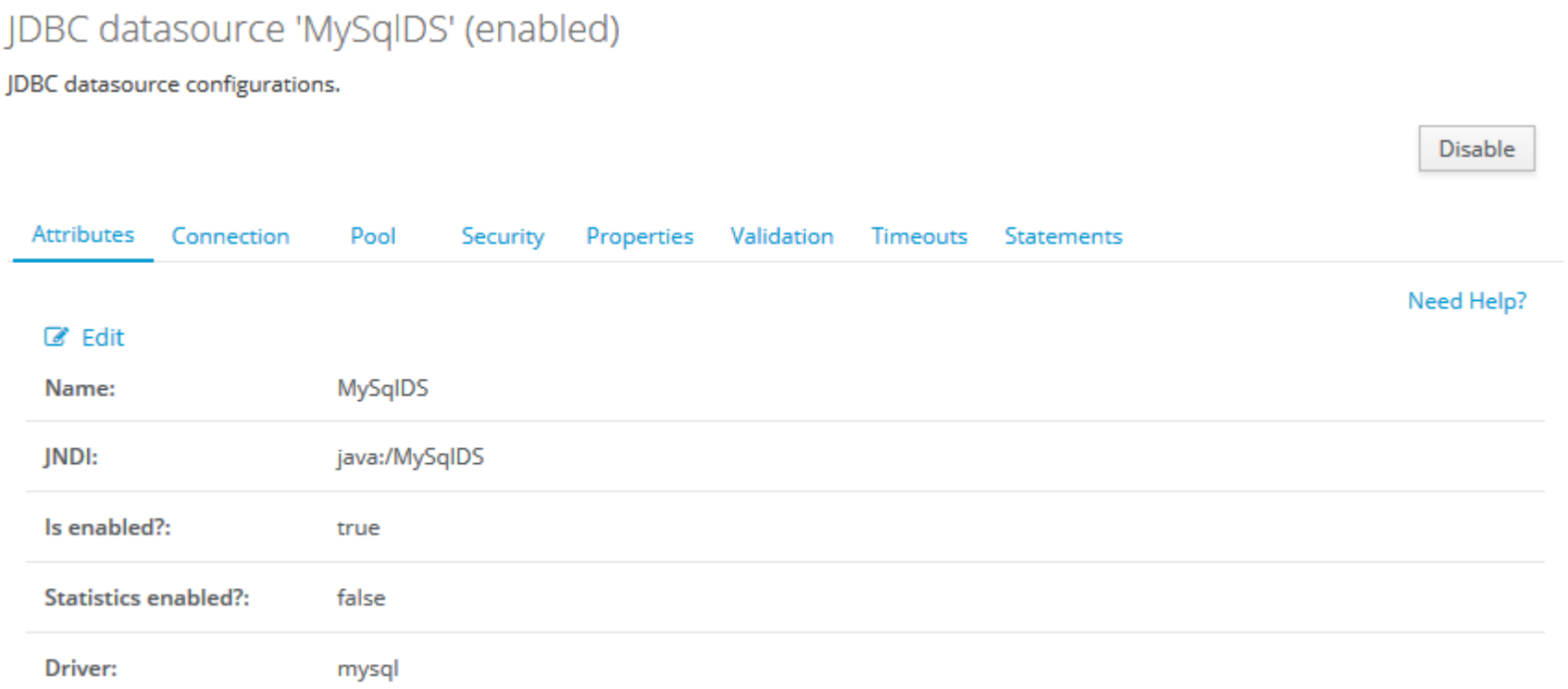
Pola encji:
- Ulica
- Numer budynku
- Numer lokalu
- Kod pocztowy
- Miasto
- Id użytkownika

Czym jest
Data Source?

Przy obsłudze baz danych dostęp do nich może być realizowany przez obiekty zwane Data Source'ami.

DS zawiera zestaw informacji wymaganych do połączenia się z bazą danych: lokalizacja bazy danych, nazwa bazy danych, protokół połącznia, itp.

Przykład
Data Source (WildFly)



Ćwiczenie
Data Source

Skonfiguruj DataSource na własnej instancji serwera WildFly.

Wildfly
MySQL

1. Pobierz Connector/J: https://dev.mysql.com/downloads/connector/j/
2. Przejdź do katalogu wildfly: modules\system\layers\base\com
3. Stwórz katalog mysql\main i przejdź do niego
4. Umieść w nim Connector/J oraz utwórz plik module.xml o treści:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="com.mysql">
   <resources>
      <resource-root path="mysql-connector-java-5.1.39-bin.jar"/>
   </resources>
   <dependencies>
      <module name="javax.api"/>
      <module name="javax.transaction.api"/>
   </dependencies>
</module>
```

Wildfly
MySQL

5. Przejdź do standalone\configuration, edytuj standalone.xml
6. Znajdź sekcję datasources->drivers i umieść w niej:

```
<driver name="mysql" module="com.mysql">
   <driver-class>com.mysql.jdbc.Driver</driver-class>
 </driver>
```

7. Przejdź do konfiguracji DS

4.
Transactions

Transakcyjna obsługa baz danych

Wprowadzenie do Transactions

Transakcje to wykonanie zbioru operacji jako jednostki atomowej: albo zostaną wykonane wszystkie z sukcesem lub wycofując dotychczasowe zmiany, żadna operacja nie zostanie wykonana.

Transakcje
JTA

Domyślny rodzaj transakcji w Java EE. Jeśli inny nie zostanie wprost zdefiniowany, JTA zostanie użyty domyślnie.

JTA to transakcje zarządzane przez kontener aplikacji.

Pozwala na odwołanie się po nazwie JNDI.

Jest to rodzaj transakcji rozproszonej definiowanej za pomocą DataSource'ów.

JTA
persistence.xml

Typ transakcji JTA definiujemy w pliku persistence.xml

```
...
<persistence-unit name="puForJTA" transaction-type="JTA">
<jta-data-source>java:/DefaultJTADS</jta-data-source>
<non-jta-data-source>java:/DefaultNonJTADS</non-jta-data-source>
...
```

jta – transakcje rozproszone
nonjta – transakcje z pojedynczą bazą danych

Transakcje
RESOURCE_LOCAL

Domyślny rodzaj transakcji w Java SE. Jeśli inny nie zostanie wprost zdefiniowany, RESOURCE_LOCAL zostanie użyty domyślnie.

RESOURCE_LOCAL to transakcje zarządzane przez programistę.

Nie powinno się go używać w kontenerach aplikacji!

„With RESOURCE_LOCAL you are re-implementing the already existing application server functionality - what is criminal :-)" Adam Bien, 24 czerwiec 2011

Transakcje
RESOURCE_LOCAL

Nie jest dozwolone użycie @PersistenceContext.

Do obsługi transakcji używamy interfejsu **EntityTransaction**

RESOURCE_LOCAL
persistence.xml

Typ transakcji RESOURCE_LOCAL definiujemy w pliku persistence.xml

```
...
<persistence-unit name="puForResourceLocal" transaction-type="RESOURCE_LOCAL">
...
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://vps148549.vps.ovh.ca:3306/isa-hibernate-workshops" />
<property name="javax.persistence.jdbc.user" value="isa" />
<property name="javax.persistence.jdbc.password" value="isa" />
...
```

Transakcje
RESOURCE_LOCAL

```
EntityTransaction transaction =
entityManager.getTransaction();


transaction.begin();
…
entityManager.persist(person);
…
transaction.rollback();
…
transaction.commit();
```

5.
JPQL

Java Persistence Query Language

JPQL
W oparciu o SQL syntax

Składnia języka JPQL jest bardzo zbliżona do SQL.

JPQL służy do tworzenia zapytań w oparciu o istniejące encje.

JPQL
Lista wyników

```
Query query = entitymanager.
    createQuery("Select UPPER(e.name) from
Employee e");

List<String> list = query.getResultList();
```

JPQL
Wynik zagregowany

```
Query query1 = entitymanager.createQuery("Select
MAX(e.salary) from Employee e");

Double result = (Double) query1.getSingleResult();
```

JPQL
NamedQuery

Jest to rodzaj wcześniej zdefiniowanego, niezmienialnego zapytania.

Osadzanie wartości parametrów odbywa się poprzez przekazywanie ich do zapytania, a nie polega na łączeniu łańcuchów.

NamedQuery
przykład

```
@NamedQuery(query = "Select e from Employee e
where e.eid = :id", name = "find employee by id")

Query query = entitymanager.createNamedQuery("find
employee by id");

query.setParameter("id", 1204);
List<Employee> list = query.getResultList( );
```

JPQL
Eager & Lazy

Koncepcją JPA jest stworzyć wierną kopię bazy danych w pamięci. Stąd, bardzo istotne jest by zwrócić uwagę na wydajność wykonywanych operacji.

Eager – pobiera cały rekord w czasie wyszukiwania z użyciem PK
Lazy – sprawdza czy rekord o zadanym PK istnieje ale nie pobiera go do pamięci. Dopiero po wywołaniu dowolnego gettera, rekord w całości zostanie załadowany do pamięci. Działa tylko w czasie transakcji.

DTO
Data Transfer Object

Obiekt, który przenosi dane między warstwami/komponentami/aplikacjami.

Zawiera tylko i wyłącznie pola oraz gettery/settery.

Obiekty DTO nie zawierają żadnej dodatkowej logiki.

6.
Relacje

One-to-One, One-to-Many, Many-to-One, Many-to-Many

Relacje
One-to-One

Relacja, która w bazie danych posiada klucz obcy w jednej z dwóch encji.

Realizowane za pomocą adnotacji @OneToOne

Jeśli nie chcemy zdegenerować relacji do jeden-do-wielu należy założyć parametr: unique=true

Wiązanie dwóch pojedynczych obiektów.

Relacje
One-to-One

```
@Entity
@Table(name="user")
public class User {

@OneToOne(cascade=CascadeType.ALL, optional=false)
 @JoinColumn(name="addr_id", unique=true)
 private Address address;
 ...
}
```

```
@Entity
@Table(name="address")
public class Address {
 @OneToOne(mappedBy="address")
 private User user;
 ...
}
```

Relacje
One-to-Many / Many-to-One

Realizowane za pomocą adnotacji
@ManyToOne oraz @OneToMany

@JoinColumn oznacza wskazanie klucza obcego

mappedBy – oznacza relację odwrotną, która uzupełnia jednokierunkowość do dwukierunkowości relacji

Relacje
Many-to-Many

Realizacja za pomocą adnotacji @ManyToMany
Aby relacja była kompletna wymagane jest istnienie tabeli intersekcji.

Zawsze musimy również wybrać encję właściciela.
Niech w naszym wypadku będzie to encja Car.

Relacje
Many-to-Many

```
@Entity
@Table(name = "T_CARS")
public class Car {

...

  @ManyToMany
  private Set<Meeting> meetings;

}
```

```
@Entity
@Table(name = "T_MEETINGS")
public class Meeting {

...

  @ManyToMany
  private Set<Car> members;

}
```

Relacje
Many-to-Many

```
@ManyToMany
 @JoinTable(
    name = "T_CAR_MEETINGS",
    joinColumns = @JoinColumn(name = "T_CARS_ID", referencedColumnName = "ID"),
    inverseJoinColumns = @JoinColumn(name = "T_MEETINGS_ID",
referencedColumnName = "ID")
 )
 private Set<Meeting> meetings;
```

Ćwiczenie
Entity: Address

Przygotuj pełną obsługę CRUD dla nowej encji Address.

Powiąż za pomocą klucza obcego User.addressID z Address.id
Ustanów relację między encjami.

Użytkownik może posiadać tylko jeden adres.
Jeden adres może mieć więcej niż jednego przynależącego do niego użytkownika.

Thanks!!

Any questions?

You can find me at @username &
user@mail.me