Hibernate

Czym jest Hibernate

- Hibernate to narzędzi dzięki, któremu możemy zapisywac obiekty Javy w relacyjnej bazie danych (MySQL/Oracle/ PostgreSQL/inne) oraz odczytywać dane z bazy danych
- Hibernate jest czołowym rozwiązaniem do mapowania obiektowo-relacyjnego (ORM) dla środowisk Java
- Hibernate odnosi się bezpośrednio do złożoności ORM, zapewniając możliwość zmapowania danych z modelu obiektowego na relacyjny model danych i odpowiadający mu schemat bazodanowy.
- Mapowanie:
  - Klasa Java odpowiada jednej tabeli
  - Obiekt Java odpowiada jednemu rekordowi w tabeli
  - Typy Java na typy SQL

Dlaczego

- Wsparcie dla stylu programowania odpowiedniego dla Javy
- Obsługa asocjacji, kompozycji, dziedziczenia, polimorfizmu, kolekcji
- Wysoka wydajność i skalowalność
- Dual-layer cache, możliwość wykorzystania w klastrze
- UPDATE tylko dla zmodyfikowanych obiektów i kolumn
- Wiele sposobów wydawania zapytań:
  - HQL – własne przenaszalne rozszerzenie obiektowe SQL
  - JPAQL – j.w. ale dot. JPA
  - Natywny SQL

Podstawowe Cechy

- Wykorzystuje siłę technologii relacyjnych baz danych, SQL, JDBC
- Professional Open Source
  - Zalety rozwiązań Open Source: otwarty kod
  - Wsparcie uznanej firmy Red Hat (m. in. JBoss)
  - Komponent serwera aplikacyjnego Java EE JBoss
  - Elastyczna licencja LGPL
- Hibernate implementuje język zapytań i persistence API z EJB 3.0/3.1
  - Hibernate EntityManager i Annotations ponad Hibernate Core

Czego potrzebujemy?

- Java, min. 1.5 (zalecane 1.6 lub 1.7)
- Biblioteki Hibernate
- Driver JDBC dla wybranej bazy danych (u nas MySQL)
- Pomoc: Eclipse (najlepiej Java EE) oraz wtyczka JBoss Tools, która zawiera Hibernate Tools

Mapowanie relacyjno-obiektowe

- Przechowujemy dane w bazie relacyjnej, wykorzystujemy w kodzie Java (poprzez obiekty POJO) w sposób jednolity i wspólny dla różnych projektów przy minimum operacji konfiguracyjnych
- Problemy przy mapowaniu:
  - tabele łączące (relacje wiele-do-wiele) a obiektowość?
  - Dziedziczenie w bazach danych relacyjnych?
  - Różnice w typach:
    - Integer/int, String, Double/double
    - vs Integer(10), varchar(20), char(20), real(10)
- Przechodząc od jednego modelu do drugiego trzeba sobie radzić z problemem

Adnotacje

- Niestety ORM musi wiedzieć o POJO więcej niż zwykły kod Java
- Jaka jest nazwa tabeli?
- Jaka jest nazwa atrybutu?
- Jakie są szczegóły relacji (jeden-do-wiele, pole klucza obcego etc.)?
- Informacje o połączeniu?

Przykład metadanych, adnotacje

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

    protected String login;
    protected String password;

    public User() {
        super();
    }

    @Id
    @Column(name = "login", nullable=false)
    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    @Column(name = "password", nullable=false)
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
```

Relacje - adnotacje

```
@Entity
@Table( name = "request_parameter" )
public class RequestParameter {
    protected long requestParameterId;

    protected String parameterName;
    protected String parameterValue;

    protected Request request;


   ...


    @ManyToOne(optional=false, cascade=javax.persistence.CascadeType.ALL, fetch=FetchType.LAZY)
    @JoinColumn(name="request_id", referencedColumnName="request_id", nullable=false)
    public Request getRequest() {
        return request;
    }

    public void setRequest(Request request) {
        this.request = request;
    }
```

problemy

- W związku z wprowadzeniem adnotacji POJO już nie są czystymi POJO
- Dawniej informacja o mapowaniu była przechowywana w osobnym pliku XML: wtedy POJO zawierały tylko atrybuty oraz gettery/settery ale wymagało to rozwijania dwóch plików niezależnie.

Tworzenie bazy danych

- Mamy 2 możliwości w zależności od tego co jest dla nas wygodniejsze: czy tworzenie kodu Java czy tworzenie kodu SQL
- Możliwość 1: piszemy klasy POJO, dodajemy adnotacje
- Możlwiość 2: piszemy kod SQL, robimy reverse engineering żeby otrzymać klasy POJO (uwaga: Hibernate Tools działa niederministycznie :( )

Konfiguracja źródła danych

- Plik persistence.xml (wymagany przez JPA) zawiera konfigurację źródła danych:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
persistence_1_0.xsd">
    <persistence-unit name="Hibernate1">

   <!-- tutaj będą dodatkowe rzeczy -->

    </persistence-unit>
</persistence>
```

Konfiguracja źródła danych

```
<properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/
paw_hibernate1" />
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
    <property name="javax.persistence.jdbc.user" value="root" />
    <property name="javax.persistence.jdbc.password" value="root" />

    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
    <property name="hibernate.use_sql_comments" value="true" />
    <!--  z tym można eksperymentować -->
   <property name="hibernate.hbm2ddl.auto" value="create"/>


    <!-- albo: MySQLInnoDBDialect -->
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
</properties>
```

Przykład w skrócie

1. Dodanie biblioteki Hibernate do Java Build Path w Eclipse:
– Tools -> Preferences -> Java -> Build Path -> User libraries
    • New -> podać nazwę “Hibernate” -> Add External JARs -> dodać rozpakowane pliki z hibernate.zip
    • New -> podać nazwę “MySQL-driver” -> Add External JARs -> dodać rozpakowany pliki z driverem

2. Utworzenie nowego projektu:
– File -> New -> JPA Project (w 3 kroku kreatora: Platform: Hibernate JPA (2.x) oraz JPA Implementation -> User Library -> wybieramy Hibernate praz MySQL-driver)
– opcjonalnie File -> New -> Java Project (trzeba dodatkowo dodać biblioteki do Hibernate do classpath)

3. Dodanie driver’a mysql do classpath:
– Project -> Properties -> Java Build Path -> Libraries -> Add External Jars
4 Utworzenie nowego pakietu w projekcie:
– Add new package

5 Dodanie nowej klasy Student (następny slajd)

6 Dodanie nowej klasy Main (kolejny slajd)
7 Utworzenie bazy danych: test_hibernate
– opcjonalnie: dodanie tabeli student: id (PK), imie, nazwisko, create_at

POJO, klasa Student

```
package paw.jpa;

import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table( name = "student" )
public class Student {
    private int id;
    private String imie;
    private String nazwisko;
    private Date dodanieData;

    public Student() {
        super();
    }

    @Id
    @GeneratedValue
    @Column(name = "id", nullable=false)
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getImie() {
        return imie;
    }

    @Column(name = "imie", nullable=false)
    public void setImie(String imie) {
        this.imie = imie;
    }

    @Column(name = "nazwisko", nullable=false)
    public String getNazwisko() {
        return nazwisko;
    }

    public void setNazwisko(String nazwisko) {
        this.nazwisko = nazwisko;
    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_at", nullable=true)
    public Date getDodanieData() {
        return dodanieData;
    }

    public void setDodanieData(Date dodanieData) {
        this.dodanieData = dodanieData;
```

Main

```
public class Main {

public static void main(String[] args) {
    EntityManagerFactory entityManagerFactory;
    EntityManager entityManager;
    entityManagerFactory = Persistence.createEntityManagerFactory("Hibernate1");
    entityManager = entityManagerFactory.createEntityManager();
    Student student = new Student();
    student.setImie("jan");
    student.setNazwisko("nowak");
    student.setDodanieData(new Date());
    try {
        entityManager.getTransaction().begin();
        entityManager.persist(student);
        entityManager.getTransaction().commit();
        System.out.println("Student został dodany. Id: " + student.getId());
    }
    catch (Exception e) {
        System.err.println("Nie można dodać rekordu: " + e);
        e.printStackTrace();
    }
}
}
```