



# Hibernate



- Technologia javy SE
- Wymaga znajomości struktury bazy danych
- Kłopotliwe wyłuskiwanie danych
- Potencjalna podatność na sql-injection



# Relacyjne bazy danych, a świat obiektowy

- Relacyjne bazy danych są przetestowane pod względem wydajności
- RDB mogą przetrzymywać ogromne ilości danych
- Sortowanie, wyszukiwanie, grupowanie
- Integralność, constrainty
- Brak polimorfizmu
- Brak dziedziczenia
- Paradygmat programowania obiektowego odzwierciedla rzeczywistość



# ORM

- Object-Relational Mapping - odwzorowanie obiektów i zależności między nimi na tabele bazy danych i na odwrót.
- Hibernate jest jednym z wielu frameworków ORMowych. Pozostałe javowe frameworki to m.in. Ebean (Playframework), Sugar (Android) itd...



# Hibernate

- Open source
- Wewnętrzna pamięć podręczna użyta w Hibernate zwiększa wydajność
- Automatyczne tworzenie lub modyfikacja tabel
- Łatwe tworzenie zapytań z wielu tabel ( poprzez relacje)



# Jak korzystać z Hibernate?

- Konieczne jest stworzenie klas reprezentujących tabele w bazie danych, wskazując które pola odpowiadają którym atrybutom
- Hibernate sam tworzy odpowiednie zapytania

# Co się może zdarzyć, gdy schemat nie jest spójny z modelem?



- Create – stworzy schemat uprzednio usuwając poprzedni
- Create-drop – stworzy schemat, usunie zbędne na końcu transakcji
- None – nic
- Update – uaktualni tabele w razie potrzeby
- Validate – zwaliduje spójność, ale nie wprowadzi zmian



# Adnotacje Encji

- @Entity - oznacza klasę, odzwierciedla tabelę BD
- @Entity(name="...") – nigdy nie używać
- @Table – zmienia domyślną nazwę tabeli BD
- @Id – oznacza atrybut jako klucz główny
- @GeneratedValue – auto generacja wartości PK
- @Column – pozwala ustawić wartości kolumny
- @Lob – typ dla dużych danych tekstowych
- @Enumerated(EnumType.STRING) – enumy
- @Transient – wyłączenie pola z persystencji
- @MappedSuperclas – zwykłe dziedziczenie
- @Version – inkrementowany z każdym updatem



# Relacje

@OneToOne – encja A może mieć relację do dokładnie jednej encji B

@OneToMany – encja A może mieć relację do kilku encji B

@ManyToOne – wiele encji A może mieć relację do dokładnie jednej encji B

@ManyToMany – wiele encji A może mieć relację do wielu encji B



# Relacja @OneToOne

## Jednokierunkowa

```
@Entity  
public class User {  
    @OneToOne(  
        cascade = CascadeType.ALL, // default: empty  
        fetch = FetchType.LAZY, // default: EAGER  
        optional = false) // default: true  
    private Address address;  
  
}  
  
@Entity  
public class Address { }
```

## Dwukierunkowa

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



# Relacja @OneToMany/@ManyToOne

## Jednokierunkowa

```
@Entity  
public class User {  
    @OneToMany(  
        cascade = CascadeType.ALL, // default: empty  
        fetch = FetchType.EAGER, // default: LAZY)  
    @JoinColumn(name= "user_id")  
    private Collection<Address> address;  
}
```

```
@Entity  
public class Address { }
```

## Dwukierunkowa

```
@Entity  
public class User {  
    @OneToMany(mappedBy="user")  
    private Collection<Address> address;  
}  
  
@Entity  
public class Address {  
    @ManyToOne  
    @JoinColumn(name= "user_fk", nullable=false)  
    private User user;  
}
```



# Relacja @ManyToMany (asocjacyjna)

## Jednokierunkowa

```
@Entity  
public class User {  
    @ManyToMany( cascade = CascadeType.ALL, // default: empty  
                fetch = FetchType.LAZY) // default: EAGER  
    @JoinTable(name = "USER_ADDRESS",  
               joinColumns = {@JoinColumn(name = „USER_ID”, nullable = false, updatable = false)},  
               inverseJoinColumns = {@JoinColumn(name = „ADDRESS_ID”, nullable = false, updatable = false)})  
    private Collection<Address> address;  
}
```



# Relacja @ManyToMany (asocjacyjna)

## Dwukierunkowa

```
@Entity  
public class Address{  
    @ManyToMany(mappedBy="address")  
    private Collection<User> users;  
}
```



# Spring Data Repository

- Pozwala na tworzenie zapytań bazując na nazwie metody
- Konieczne jest stworzenie interfejsu dziedziczącego po JpaRepository
- Brak sql,
- Wsparcie dla stronicowania
- Łatwa konfiguracja
- Zestawienie słów kluczowych:
  - <http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>



# Transakcje

ACID w relacyjnych bazach danych:

- Atomicity – “wszystko albo nic”
- Consistency – zapewnia przejście z jednego spójnego stanu w inny, też spójny(constrainty, triggers itd)
- Isolation – zabezpieczenie przed wielowiątkowością
- Durability – trwałość danych zaraz po scomitowaniu transakcji



# Dziedziczenie



# @MappedSuperclass

Dzięki oznaczeniu klasy rodzica tą adnotacją, klasy dziedziczące będą posiadały wszystkie jej pola i metody (włącznie z odwzorowaniem). Oznaczona w ten sposób klasa powinna być abstrakcyjna (nie jest to wymóg ale dobra praktyka, nie powinna istnieć instancja tej klasy).



# SINGLE\_TABLE

```
@Entity  
@Inheritance(strategy = javax.persistence.InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name = "type", discriminatorType =DiscriminatorType.STRING)  
public class ParentEntity { (...)  
}
```

# JOINED



```
@Entity  
@Inheritance(strategy = javax.persistence.InheritanceType.JOINED)  
public class ParentEntity { (...)  
}
```



# TABLE\_PER\_CLASS

```
@Entity  
@Inheritance(strategy = javax.persistence.InheritanceType.TABLE_PER_CLASS)  
public class ParentEntity { (...)  
}
```