

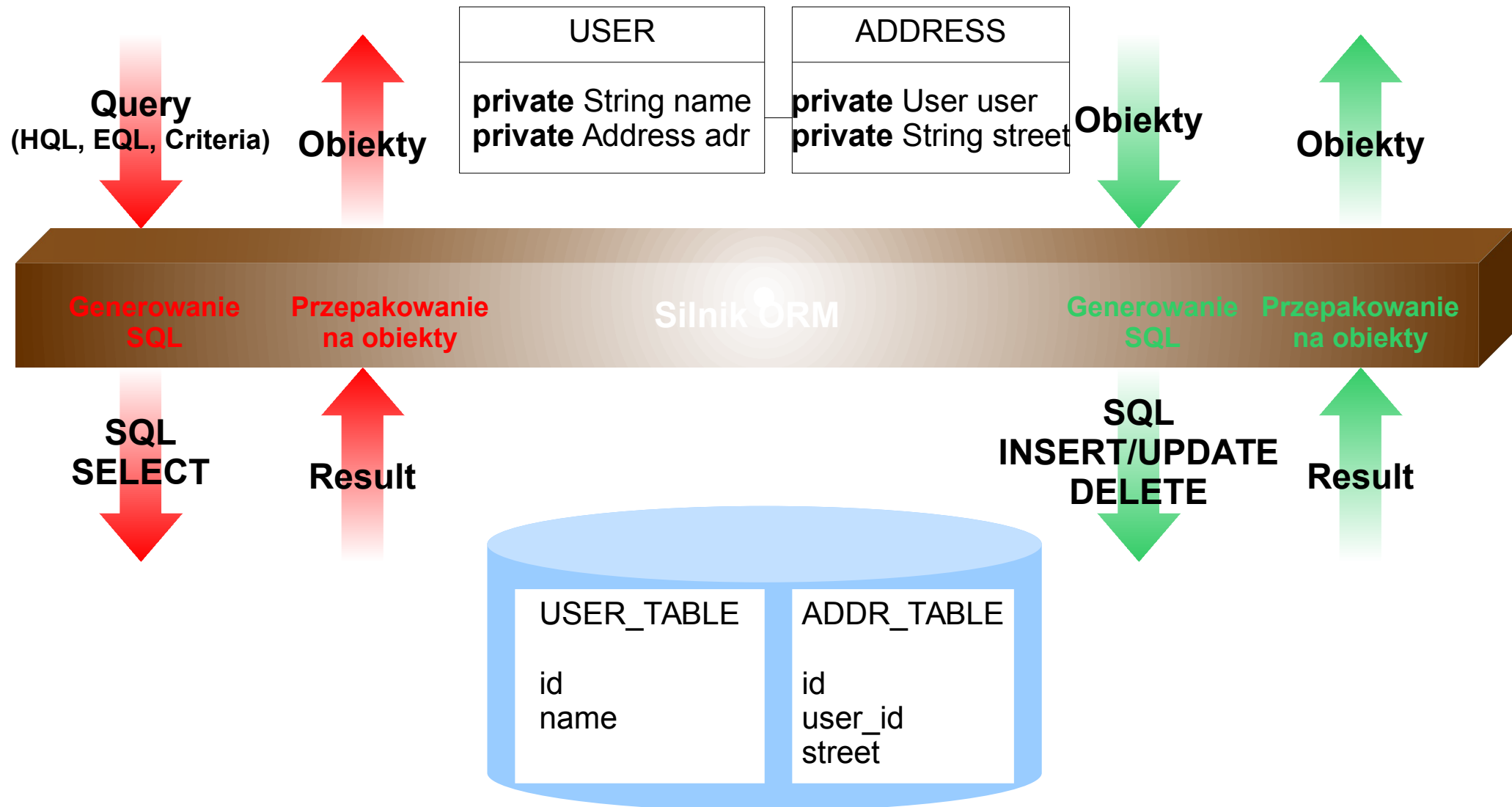


Praktycy dla Praktyków  
Szkolenia i doradztwo

## Java Persistence API 2.0

- Podstawy Object-Relational Mapping
- Konfiguracja Hibernate
- Zasada działania silnika ORM
- Mapowanie encji
  - Lazy Loading
  - Operacje kaskadowe
  - Dziedziczenie
- EntityManager
- Zapytania JPQL, Criteria API
- Optymalizacja zapytań
- Praktyczne wykorzystanie Callbacks
- Hibernate
  - Cache

# Ogólna idea Object-relational Mapping



- **Przeszłość**
  - **EJB 2.0**
    - Beany encyjne – namiastka ORM
    - Zależne od technologii
  - **Hibernate – prawdziwy ORM (wersje 1,2,3)**
- **Standaryzacja**
  - **Specyfikacja JPA inspirowana Hibernate 3**
  - **POJO – niezależne od technologii**
- **Teraźniejszość**
  - **JPA – specyfikacja, zestaw interfejsów i adnotacji**
    - **Standard JEE**
    - **Ogólny standard (również JSE)**
  - **Implementacje**
    - **Hibernate - „stare” API + implementacja JPA**
      - **Wiele dodatkowych features (CriteriaAPI, Cache, Cascade+)**
    - **OpenJPA, TopLink, Kodo, ...**

- Mapowanie Klasa-Tabele
  - Klasa zawiera pola odpowiadające kolumnom
    - Mapowanie przez XML lub Adnotacje
  - Kod logiki?
- Generowanie SQL na podstawie mapowania – zgodnie z wybranym dialektem SQL
- Śledzenie zmian encji (oraz encji zagregowanych)
  - Mechanizm „brudzenia”
  - Automatyczne utrwalanie zmian
- Wygodne mechanizmy wbudowane
  - Lazy loading – pobieranie zagregowanych encji w momencie pierwszego dostępu do nich
  - Operacje kaskadowe – operacje na całych grafach obiektów zagregowanych
  - **Uwaga na wydajność!!!**

## **Dowiesz się:**

- Na czym polega mapowanie
- Poznasz składnię anotacji
- Poznasz techniki mapowania
  - agregacji
  - generatorów kluczy
  - typów wyliczeniowych
  - dziedziczenia
- Poznasz ustawienia strategii
  - ładowania danych
  - wykonywania operacji kaskadowych

```
@Entity
@Table(name="Users")
public class User{
    @Id
    @GeneratedValue
    String id;

    String firstName;

    @Basic(fetch=FetchType.LAZY)
    Address lastName;

    @Column(name="couter", nullable=false)
    int points;

    @Temporal(TemporalType.TIMESTAMP)
    Date birthDate

    @Transient
    Date loginTime;
}
```





- Powiązania pomiędzy tabelami są mapowane jako agregacja obiektów

Rodzaje agregacji:

- **One-to-one**
  - Obiekt ma referencję do innego obiektu
- **One-to-many**
  - Obiekt ma referencję do kolekcji obiektów
- **Many-to-one**
  - Obiekt ma referencję do innego obiektu (Wiele innych obiektów ma referencję do niego)
- **Many-to-many**
  - Niedozwolony w paradygmacie relacyjnym – wymaga tabeli linkującej
  - W modelu klas pomijamy tabelę linkującą
  - Obiekty mają referencję do kolekcji obiektów

# @OneToOne

## Jednokierunkowa: Klient ma jeden Adres



```
@Entity
public class Customer{

    @OneToOne (
        cascade=CascadeType.ALL, //default: empty
        fetch=FetchType.LAZY, //default EAGER!!!
        optional=false //default: true)
    @JoinColumn(name=ADDRESS_ID)
    private Address address;
}
```

# @OneToOne

## Dwukierunkowa: Klient ma jeden Adres



```
@Entity
public class Customer{

    @OneToOne()
    private Address address;
}
```

```
@Entity
public class Address{

    @OneToOne(mappedBy="address")
    private Customer customer;
}
```

### mappedBy

- Customer może mieć wiele pól klasy Address – stąd dwuznaczność
- Właścicielem w tym powiązaniu jest Customer
  - Ustawienie tego adresu innej osobie wymaga `customer.setAddress(null)`

## @OneToMany

### Jednokierunkowa: Klient ma wiele adresów



```
@Entity
public class Customer{

    @OneToMany (
        cascade=CascadeType.ALL, //default: empty
        fetch=FetchType.LAZY, //default LAZY
    )
    @JoinColumn(name="customer_id")
    @OrderBy („city ASC”)
    private Collection<Address> addresses;
}
```

**Domyślnie (bez @JoinColumn) tworzy tabelę linkującą!!!**

## @OneToMany

### Dwukierunkowa: Klient ma wiele adresów



```
@Entity
public class Customer{

    @OneToMany(mappedBy="customer")
    private Collection<Address> addresses;
}
```

```
@Entity
public class Address{

    @ManyToOne
    private Customer customer;
}
```

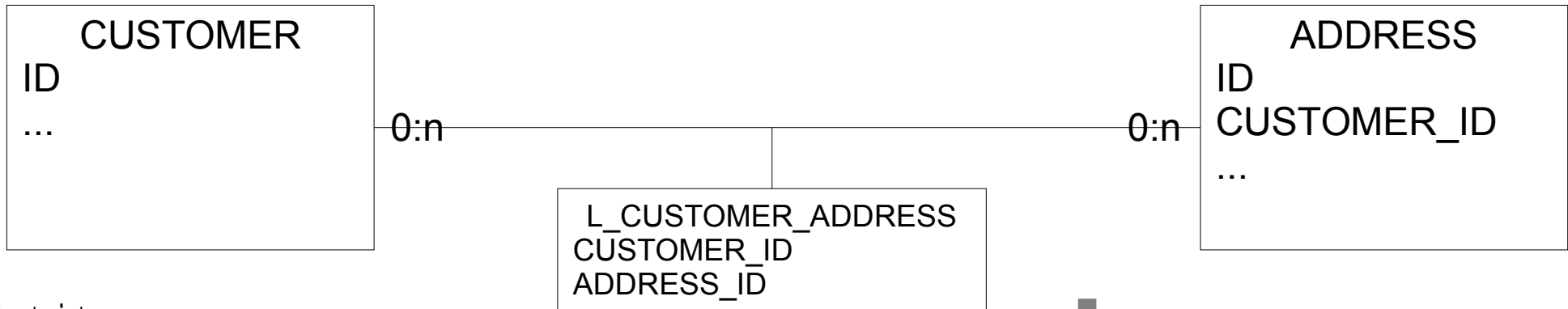
## Klient „nie wie” jakie ma zamówienia

- Przydatne w przypadku gdy klient ma bardzo wiele zamówień



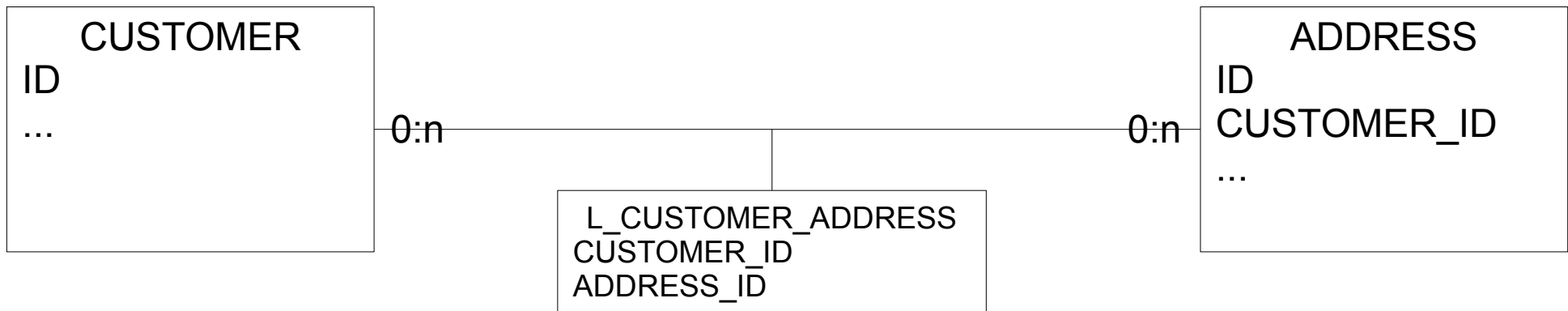
```
@Entity
public class Order{

    @ManyToOne (
        cascade=CascadeType.ALL, //default: empty
        fetch=FetchType.LAZY, //default EAGER!!!
        optional=false //default: true)
    @JoinColumn(name=CUSTOMER_ID)
    private Customer customer;
}
```



```
@Entity
public class Customer{

    @ManyToMany (
        cascade=CascadeType.ALL, //default: empty
        fetch=FetchType.EAGER, //default LAZY
    )
    private Collection<Address> addresses;
}
```



```
@Entity
public class Customer{

    @ManyToMany()
    private Collection<Address> addresses;
}

@Entity
public class Address{

    @ManyToMany(mappedBy="addresses")
    private Collection<Customer> Customers;
}
```



- Agregacje dwukierunkowe
  - Stosować tylko gdy są uzasadnione biznesowo
    - Enkapsulacja
  - Są tworzone przez generatory
  - Mogą ułatwiać pisanie zapytań
- Kolekcje
  - Uwaga na ilość obiektów
  - Uwaga na Lazy Loading i "n+1 select problem"
- FetchType - Uwaga na EAGER
  - OneToOne
  - ManyToOne

- Mapowanie automatyczne
  - Z bazy na encje – generatory
    - Naiwne mapowanie wszystkich agregacji
  - Z encji na bazę – feature implementacji JPA
    - Rapid development
      - Wygodne w fazie prototypowania
    - Zweryfikować
      - Może wymagać tuningu bazy
- Ręczne
  - Całościowo - żmudne
  - Tuning po automatycznym



# Mapowanie zaawansowane

## Klasy zagnieżdżone

```
@Entity
public class Customer{
    @Id
    String id;
    String name;

    @Embedded
    Address address ;
}
```

**@Embeddable**

```
public class Address{
    String street;
    String coutry;
}
```

CUSTOMER	
ID	
NAME	
STREET	
COUNTRY	

# Mapowanie zaawansowane

## Klasy zagnieżdżone

@Entity

```
public class Customer{
```

```
    @Id
```

```
    String id;
```

```
    String name;
```

```
    @Embedded
```

```
    @AttributeOverrides({
```

```
        @AttributeOverride(name="street", column="sh_street"),
```

```
        @AttributeOverride(name="country", column="sh_country"),
```

```
    })
```

```
    Address shippingAddress;
```

```
    @Embedded
```

```
    @AttributeOverrides({
```

```
        @AttributeOverride(name="street", column="bl_street"),
```

```
        @AttributeOverride(name="country", column="bl_country"),,
```

```
    })
```

```
    Address billingAddress;
```

```
}
```

CUSTOMER

ID

NAME

SH\_STREET

SH\_COUNTRY

BL\_STREET

BL\_COUNTRY

```
public enum Gender{  
    MALE („gender.m”),  
    FEMALE („gender.f”),  
    UNKNOWN („gender.uk”),  
    UNDISCOVERED („gender.ud”);  
  
    private String msgKey;  
  
    public String getName(){  
        return MessageUtils.get(msgKey);  
    }  
}
```

```
@Entity  
public class User{  
    @Enumerated(EnumType.STRING)  
    private Gender gender  
  
    ...  
}
```



# Mapowanie zaawansowane

## Generator kluczy

```
@Id
@GeneratedValue(strategy=GeneratorType...)
private Long id;
```

AUTO, IDENTITY, SEQUENCE, TABLE

```
@Entity
public class Customer {

    @TableGenerator(
        name=„custGen“,
        table=" KEY_TABLE ",
        pkColumnName=„GEN_ID“
        valueColumnName="GEN_VALUE",
        pkColumnValue=„CUST_GEN")

    @Id
    @GeneratedValue(
        strategy=TABLE,
        generator=„custGen")
    public int id;
}
```

```
@Entity
@SequenceGenerator(
    name="custSeq",
    sequenceName="CUST_SEQ")

public class Customer {

    @Id
    @GeneratedValue(
        strategy=SEQUENCE,
        generator=„custSeq")
    public int id;

}
```



- JPA może wykonać daną operację zarówno na encji jak i na jej zagregowanych składowych
- Strategie operacji kaskadowych
  - PERSIST
  - MERGE – również dodaje składowe
  - REMOVE
  - REFRESH - kosztowne!
  - ALL
- Uwaga na sensowność tych operacji z biznesowego punktu widzenia
  - Modyfikacja składowych ma sens gdy związek ma naturę **kompozycji**
    - np: zamówienie i jego pozycje
    - pozycja nie ma sensu bez zamówienia
- **Uwaga na aspekt bezpieczeństwa!!!**
  - **Przesyłanie grafów obiektów na server (PERSIST/MERGE)**

```
@Entity
public class Customer{
    @ManyToOne(cascade={
        CascadeType.PERSIST,
        CascadeType.REMOVE})
    private Address addr;
}
```

#### DELETE\_ORPHAN

- Aplikuje się dla @OneToMany
- Kaskadowe usunięcie tych encji składowych, które usunięto z kolekcji encji głównej

JAP2: orphanRemoval = true

```
@Cascade({  
    org.hibernate.annotations.CascadeType.SAVE_UPDATE,  
    org.hibernate.annotations.CascadeType.DELETE_ORPHAN})
```



# Mapowanie zaawansowane

## FetchType - Polityka ładowania danych

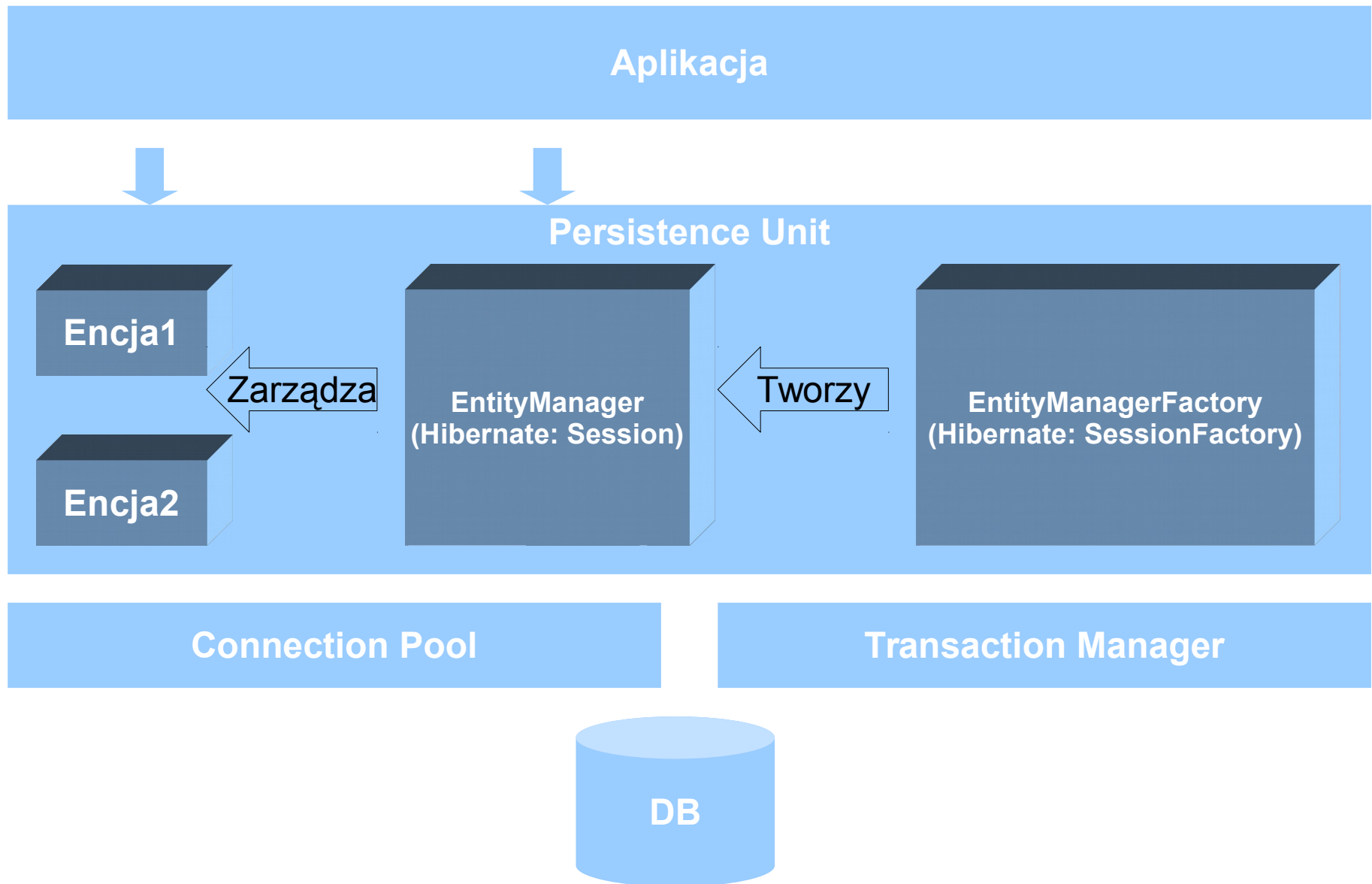
- ORM wykorzystuje
  - wzorzec Proxy – pośrednik do encji
  - modyfikacja ByteCode
- Adnotacje powiązań posiadają atrybut *fetch* o wartościach
  - FetchType.EAGER
    - Chciwie/łapczywie pobiera zagregowane encje
      - JOIN w SQL
      - Dodatkowe zapytanie! - **n+1 Select Problem**
    - Domyślny dla zagregowanych encji
      - @OneToOne
      - @ManyToOne
  - FetchType.LAZY
    - Leniwe ładowanie danych gdy są potrzebne – wywołanie get()
      - Dodatkowe zapytanie! - **n+1 Select Problem**
    - Domyślne dla kolekcji i obiektów LOB
      - @OneToMany
      - @ManyToMany
      - @Lob

```
@Entity
public class User{
    @OneToOne(fetch=FetchType.LAZY)
    Address address;
}
```



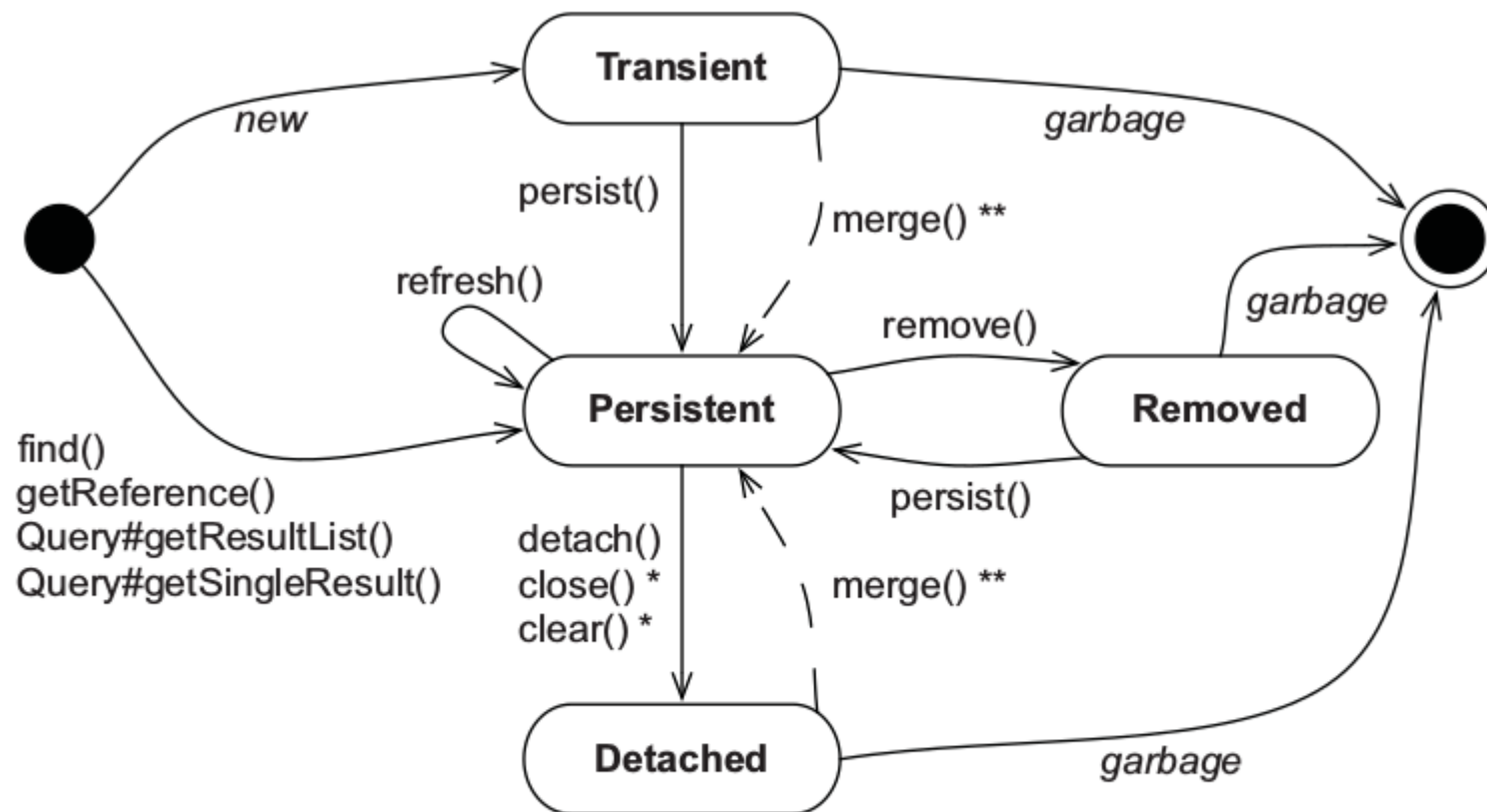
## **Dowiesz się:**

- Jakie są główne składowe architektury JPA
- Jaki jest cykl życia Encji
- Jak korzystać z Entity Managera
- Jak zarządzać encjami



- Obiekt zarządzający encjami
  - Jednostka pracy
  - Śledzi encje
    - Zarządza ich cyklem życia
    - Synchronizuje je z bazą danych
  - Jego utworzenie jest „tanie”
- Stanowi cache
- Współpracuje z Managerem transakcji (JTA)
- Stanowi bramę, przez którą encje komunikują się z bazą danych
  - Lazy loading
  - Po jego zamknięciu encje nie mogą korzystać już z LL





\* Affects all instances in the persistence context

\*\* Merging returns a persistent instance, original doesn't change state

**@PrePersist** —wywołanie `em.persist()`  
**@PostPersist** —po SQL INSERT  
**@PreRemove** —wywołanie `em.remove()`  
**@PostRemove** —przed SQL DELETE  
**@PreUpdate** —EM wykrył zmianę encji  
**@PostUpdate** —po SQL UPDATE  
**@PostLoad** — po załadowaniu encji

```
@Entity
@EntityListener(OrderCallbackListeners.class)
public class User {...}
```

```
public class OrderCallbackListener {
    @PrePersist
    public void auditOrderCreation(Order order) {...}
}
```

# Entity Manager

## API zarządzania encjami

**void persist(Object o)**

utrwała encję

**void remove(Object o)**

usuwa encję

**Object merge(Object o)**

scalenie nowej/odłączonej encji

**wynikowa** encja jest zarządzana

**void refresh(Object o)**

odświeża stan encji na podstawie bazy

**boolean contains(Object o)**

true jeżeli encja jest zarządzana przez EM

**void flush()**

synchronizacja EM z bazą

```
@PersistenceContext
```

```
EntityManager entityManager;
```

```
public void deleteUser(Integer id) {
```

```
    User u = entityManager
```

```
        .find(User.class, id);
```

```
    entityManager.remove(u);
```

```
}
```

### **Object find(Class clazz, Object primaryKey)**

wyszukuje encję o zadanym kluczu głównym

### **Object getReference(Class clazz, Object primaryKey)**

tworzy proxy ale jeszcze nie pobiera danych z bazy,  
dane zostaną leniwie pobrane przy pierwszym dostępie

### **Query createQuery(String query)**

przygotowuje do wykonania zapytanie w EJB QL

### **createNamedQuery(String name)**

przygotowuje do wykonania istniejące zapytanie

```
@PersistenceContext
```

```
EntityManager entityManager;
```

```
public User loadUser(Integer id) {
```

```
    return entityManager.find(User.class, id);
```

```
}
```

## **PersistenceException**

klasa bazowa

## **OptimisticLockException**

zmodyfikowano encję chronioną przez mechanizm wersjonowania

## **EntityExistsException**

zapisujemy encję już zarządzaną

## **EntityNotFoundException**

modyfikacja nieistniejącej encji

## **NoResultException**

Query.singleResult() zwraca 0 danych

## **NonUniqueResultException**

Query.singleResult() zwraca >1 danych

## **org.hibernate.LazyInitializationException**

Encja korzysta z Lazy Loading gdy EM jest już zamknięty

**Specyfikacja nie określa sposobu reagowania!**



## Dowiesz się:

- Jak tworzyć zapytania
- Jak je hermetyzować
- W jaki sposób sterować optymalnym pobieraniem danych
  - Chciwe pobieranie zagregowanych Encji
  - Unikanie „n+1 Select Problem”
  - Unikanie pobierania nadmiernych danych
- Poznasz Criteria API
  - Alternatywny sposób zapytań
  - Praktyczne aspekty wykorzystania

**Object find(Class clazz, Object primaryKey)**

wyszukuje encję o zadanym kluczu głównym

**Query createQuery(String query)**

przygotowuje do wykonania zapytanie w EJB QL

**Query createNamedQuery(String name)**

przygotowuje do wykonania istniejące zapytanie

**Query createNativeQuery(String sql)**

przygotowuje do wykonania natywny SQL



**List getResultList();**

wykonanie zapytania o listę encji

**Object getSingleResult();**

wykonanie zapytania o dokładnie jedną encję (wyjątki!)

**int executeUpdate();**

wykonanie modyfikacji (DELETE, INSERT, UPDATE)  
zwraca ilość zmodyfikowanych rekordów

```
Query query = entityManager
    .createQuery("SELECT u FROM User u");
List users = query.getResultList();
```

**Query setParameter(String name, Object value);**  
Nie stosować konkatencji Stringów!!! (Atak SQL injection)

**Query setMaxResults(int maxResult);**  
**Query setFirstResult(int startPosition);**  
Jeżeli baza nie wspiera to obcina wynik w pamięci (Hibernate)

Interfejs Query to tak zwany "fluent interface"

```
List users= entityManager
    .createQuery("SELECT u FROM User u WHERE u.name LIKE :name")
    .setParameter("name", searchName)
    .setMaxResults(100)
    .getResultList();
```

```
@Entity
@NamedQueries ({
    @NamedQuery (
        name="usersByName",
        query="SELECT u FROM User u WHERE u.name LIKE :searchName")
})

public class User {

    //...

}
```

```
List users = em.createNamedQuery("usersByName")
                .setParameter("searchName", "Poszukiwany")
                .getResultList();
```

## *EXISTS, ALL, ANY/SOME*

```
SELECT article
FROM Article article
WHERE article.price > ALL (
    SELECT c.salary
    FROM Customer c)
```

**between** : user.points between 10 and 20  
**in**: user.role in ('admin', 'god')      user.role in (subselect)  
**member of**: user.role member of stats.topRoles  
**is null**: user.address is not null  
**is empty**: user games is not empty

- Dwa typy złączeń

- inner join (join)
- left outer join (left join)

- Złączenie aby nałożyć kryteria na elementy kolekcji

```
SELECT u FROM user u  
      JOIN u.address a JOIN a.type t WHERE t = ...
```

- Złączenie **fetch** aby chciecie pobrać zagregowane dane

```
SELECT u FROM user u  
      JOIN FETCH u.address
```

- Zwijanie iloczynu kartezjańskiego do drzewa encji

```
SELECT DISTINCT u FROM User u  
      JOIN FETCH u.addresses  
SELECT DISTINCT p.department  
      FROM Professor p
```



Dowiesz się:

- Na czym polega ten problem wydajnościowy
- Z czego wynika
- Jak go wykrywać
- W jaki sposób można go unikać oraz niwelować dotkliwość



```
List<User>
•user1
  • List<Address>
    • address1
    • address2
•user2
  • List<Address>
    • address3
    • address4
    • address5
•user3
  • List<Address>
    • address6
```

- Przy pomocy naiwnego zapytania pobieramy listę użytkowników
- Iterujemy po liście i pobieramy zagregowane obiekty

```
@Entity
public class User{
    @OneToMany
    private List<Address> addresses;
}
```

```
List<User> users = entityManager.
    createQuery("SELECT u FROM User u").getResultList();

for (User u : users){
    for (Address a : u.getAddresses()){
        //...
    }
}
```

```
List<User>
•user1
  • List<Address>
    • address1
    • address2
•user2
  • List<Address>
    • address3
    • address4
    • address5
•user3
  • List<Address>
    • address6
```

- Jeżeli sesja persystencji jest **aktywna** wówczas działa mechanizm Lazy Loading
  - Dla każdego z N użytkowników wykonywane jest zapytanie o jego adresy – (N razy adresy + 1 raz użytkownicy, w sumie N+1)
- Jeżeli sesja jest zamknięta wówczas **w Hibernate** dostajemy LazyInitializationException

## n+1 Select Problem

### Podejście (nie rozwiązanie) gorliwe

```
List<User>
•user1
  • List<Address>
    • address1
    • address2
•user2
  • List<Address>
    • address3
    • address4
    • address5
•user3
  • List<Address>
    • address6
```

- Mapowanie kolekcji z fetch=FetchType.EAGER
  - Działa dla find()
  - **Dla zapytań o listę jest to jedynie sugestia – wciąż możliwe n+1 zapytań**
- Mapowanie Encji z FetchType.JOIN (tylko Hibernate)
- @LazyCollection (tylko Hibernate)
  - FALSE
  - EXTRA – lazy, ale próba uniknięcia pobieranie
- WADY
  - Usztywnienie mapowania
  - Chciwe/lapczywe pobieranie **nie** jest zawsze pożądane

```
@Entity
public class User{
    @OneToMany(fetch=FetchType.EAGER)
    private List<Address> addresses;
}
```

## n+1 Select Problem

### Rozwiązanie „na szybko”

```
List<User>
•user1
  • List<Address>
    • address1
    • address2
•user2
  • List<Address>
    • address3
    • address4
    • address5
•user3
  • List<Address>
    • address6
```

- @org.hibernate.annotations.BatchSize
- Pobierając element kolekcji pobiera x kolejnych „na zapas”

- WADY
  - Specyfika Hibernate
  - Działa na ślepo

```
@Entity
public class User{
    @OneToMany
    @BatchSize(size=10)
    private List<Address> addresses;
}
```

## n+1 Select Problem Rozwiązanie właściwe

```
List<User>
•user1
  • List<Address>
    • address1
    • address2
•user2
  • List<Address>
    • address3
    • address4
    • address5
•user3
  • List<Address>
    • address6
```

- Stworzenie rzetelnego zapytania z JOIN FETCH
  - Zapytanie „szyte na miarę” - per Use Case
  - Warto hermetyzować w DAO

```
SELECT DISTINCT u FROM User u
      JOIN FETCH u.addresses
```

- Można próbować parametryzować ogólną metodę DAO
  - doklejenie JOIN FETCH
  - lepiej użyć Criteria API

```
Criteria criteria = hibernateSession
    .createCriteria(User.class);
```

```
if (...) {
    criteria.setFetchMode("addresses", FetchModel.JOIN);
}
```

- Kolekcja (LAZY) jest iterowana przez komponent graficzny
  - Listy
  - h:dataTable
- Przy klasycznym podejściu kontekst persystencji jest zamknięty gdy wykonuje się kod GUI
  - LazyInitializationException
  - Jest to sygnał, że model został źle zainicjowany
- „Wygodne ulepszenia” pozwalają jednak korzystać z Lazy Loading w warstwie widoku
  - Open Session in View
  - Transakcje w Seam

- Manualne

- kontrola konsoli

```
<property name="show_sql">true</property>  
<property name="format_sql">true</property>
```

```
log4j.logger.org.hibernate.type=DEBUG
```

- Automatyczne

- Testy integracyjne mierzące ilość zapytań (API Statistics)

```
Statistics stats = sessionFactory.getStatistics()
```





### **Dowiesz się:**

- Jak są sposoby na pobranie tylko tych danych, które są potrzebne
- Jak tworzyć własne paginatory tabel

Zapytania „przekrojowe” pobierają dane z wielu Encji (tabel), ale w konkretnym Use Case potrzeba zaledwie kilku kolumn z każdej z tabel

1. Obciążenie komunikacji z bazą danych
2. W razie zdalnego serwisu narzut na serializację
3. Chwilowe obciążenie pamięci (niektóre pola mogą być „ciężkie”)

Przepakowanie z encji do DTO rozwiązuje jedynie problem #2

## Pobieranie zbyt dużej ilości danych Rozwiązanie – Lazy loading

```
@Basic(fetch=FetchType.LAZY)  
private String documentContent;
```

- Hibernate: wymaga instrumentalizacji ByteCode
- Dodatkowe zapytanie gdy pole jednak jest potrzebne

## Pobieranie zbyt dużej ilości danych

### Rozwiązanie – Specyficzne klasy mapujące

Specyficzne klasy mapujące zawierających potrzebne atrybuty

- Mnożenie bytów w domenie
- Brak wsparcia dla cache

```
@MappedSuperClass
public class DocumentBase{
    @Id
    private Long id;
}
```

```
@Entity
public class DocumentLite extends DocumentBase{
    private String title;
}
```

```
@Entity
public class DocumentBig extends DocumentBase{
    private String content;
}
```

## Pobieranie zbyt dużej ilości danych Rozwiązanie – Pobieranie DTO

```
SELECT NEW pakiet.UserDTO(u.id, u.name, u.address)  
FROM User u JOIN FETCH u.address
```

- Pobieranie danych wprost do Data Transfer Object
  - Ograniczenia dla konstruktora

- Native SQL

```
sess.createQuery("SELECT id, title FROM Documents").list();
```

```
sess.createQuery("SELECT id, title FROM Documents")  
    .addEntity("d", Document.class)  
    .addJoin("d.author");
```

- Hermetyzacja JDBC w DAO...

- Warto stworzyć wygodne klasy w stylu Spring
  - JdbcTemplate – hermetyzuje operacje na JDBC z uwzględnieniem transakcji
  - JdbcDaoSupport – klasa bazowa dla DAO (zawiera template)
- Command-query Separation...

## **EntityManager (pobieranie wybranych pól)**

## **Dowiesz się:**

- Czym jest Criteria API
- Poznasz zasadę wzorca projektowego Builder
- Poznasz konwencję Fluent Interface
- Poznasz podstawowe API
- Poznasz wygodne techniki korzystania z niego



- Obiektowa technika budowania zapytań
  - Składania opiera się na obiektach zamiast na konkatenacji napisów
  - Pozwala wygodnie tworzyć dynamiczne zapytania
  - Bez potrzeby mozolnej konkatenacji
- API oparte o wzorzec budowniczego
  - Abstrakcyjny budowniczy oferuje zestaw metod przyjmujących abstrakcyjne składniki
  - Budowniczy produkuje jako wynik SQL (poza zakresem programistów – użytkowników)
- [https://en.wikibooks.org/wiki/Java\\_Persistence/Criteria](https://en.wikibooks.org/wiki/Java_Persistence/Criteria)

- **CriteriaBuilder**
  - tworzy CriteriaQuery
  - Tworzy Predicates – warunki logiczne zapytań
- **CriteriaQuery**
  - Składa poszczególne fragmenty zapytania
- type-restricted mode, non-typed mode

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

CriteriaQuery criteriaQuery = criteriaBuilder.createQuery();
Root employee = criteriaQuery.from(Employee.class);
criteriaQuery.where(criteriaBuilder.greaterThan(employee.get("salary"), 100000));
Query query = entityManager.createQuery(criteriaQuery);
List<Employee> result = query.getResultList();
```

- `createQuery()`
- `createQuery(Class)`
- `createTupleQuery()`
- `createCriteriaDelete(Class)`
- `createCriteriaUpdate(Class)`
- Metody do tworzenia predykatów
- Metody do tworzenia wyrażeń

- `distinct(boolean)`
- `from(Class)`
- `from(EntityType)`
- `select(Selection)`
- `multiselect(Selection...), multiselect(List<Selection>)`
- `where(Expression), where(Predicate...)`
- `orderBy(Order...), orderBy(List<Order>)`
- `groupBy(Expression...), groupBy(List<Expression>)`
- `having(Expression), having(Predicate...)`
- `subQuery(Class)`

```
CriteriaQuery criteriaQuery = criteriaBuilder.createQuery();
Root employee = criteriaQuery.from(Employee.class);
criteriaQuery.where(
    criteriaBuilder.equal(employee.get("id"),
        criteriaBuilder.parameter(Long.class, "id"))
);
Query query = entityManager.createQuery(criteriaQuery);
query.setParameter("id", id);
Employee result2 = (Employee) query.getSingleResult();
```

## Wygodne tworzenie dynamicznych kwerend

```
CriteriaQuery criteriaQuery = criteriaBuilder.createQuery();
Root employee = criteriaQuery.from(Employee.class);
Set<Predicate> conditions = new HashSet<>();
if(params.searchByEmployeeId())
    conditions.add(criteriaBuilder.equal(employee.get("id"),
        params.employeeId()));
if(params.searchBySalary())
    conditions.add(criteriaBuilder.greaterThan(employee.get("salary"),
        params.salary()));
criteriaQuery.where(conditions.toArray(new Predicate[0]));
Query query = entityManager.createQuery(criteriaQuery);
Employee result2 = (Employee)query.getSingleResult();
```



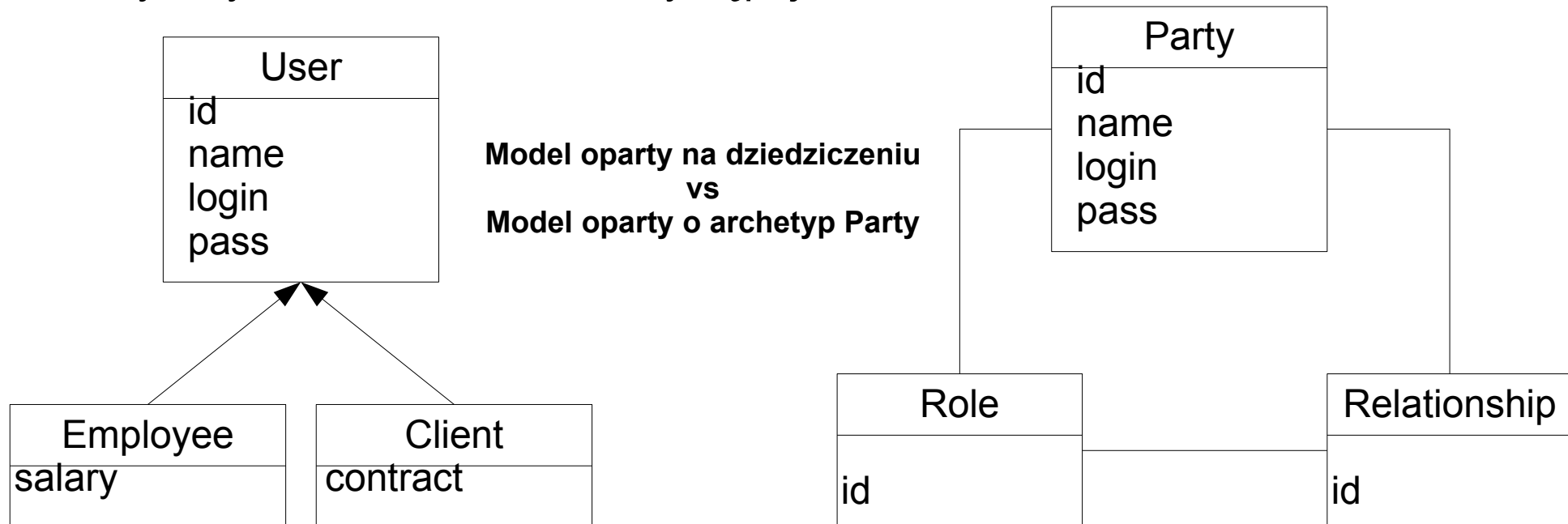
### **Dowiesz się:**

- Kiedy dziedziczenie w obiektach domenowych ma sens
- Jakie są techniki odwzorowania dziedziczenia w modelu relacyjnym
- Jakie są ich słabe strony
- Czym kierować się przy wyborze techniki
- Poznasz zapytania polimorficzne



# Dziedziczenie w obiektach domenowych

- Dziedziczenie usztywnia model
  - Nie ma możliwości aby Encja zmieniła typ
  - Lepszym rozwiązaniem są atrybuty
- Dziedziczenie doskonale modeluje byty z odpowiedzialnością wykonania czynności
  - Umożliwia ich polimorficzne **zachowanie**
  - np. Strategie (Polityki)
  - W klasycznym modelowaniu nie występuje



- SINGLE
  - Oparte o kolumnę dyskryminatora
  - + Wysoka wydajność (brak JOIN)
  - - Problem pustych kolumn (brak normalizacji)
  - - Problem z ograniczeniami (np. NOT NULL)
- TABLE PER CLASS
  - + Nie występuje problem ograniczeń
  - - Redundancja (powielenie wspólnych kolumn)
  - - Niższa wydajność - konieczność stosowania Uni (o ile baza wspiera) lub kilku zapytań przy **zapytaniach polimorficznych**
- JOINED
  - + Zredukowany problem ograniczeń (wspólne kol. wciąż problematyczne)
  - + Znormalizowany model
  - - Niższa wydajność – konieczność stosowania JOIN

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="user_type",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("User")
public class User { ... }

@Entity
@DiscriminatorValue("Client")
public class Client extends User { ... }
```

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class User { ... }

@Entity
public class Client extends User { ... }
```

## Dziedziczenie

### Przykład TABLE\_PER\_CLASS

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class User { ... }

@Entity
public class Client extends User { ... }
```

Zapytania polimorficzne wyszukują encji, które są danego typu w sensie operatora **instanceof**

- Dana klasa i jej podklasy

```
SELECT u FROM User u
```

- Czasem encje posiadają wspólne atrybuty
  - ale nie chcemy modelować domeny na zasadzie dziedziczenia encji
- Wówczas należy posłużyć się klasami bazowymi z zamapowymi atrybutami

```
@MappedSuperclass
public abstract class BaseAbstractEntity {

    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }

}
```

- Klasa bazowa **nie** jest encją
- Jedynie „wyciąga przed nawias” wspólne atrybuty
  - Uwaga na naruszenie Liskov Substitution Principle (LSP)
  - Ale: jest to wzorzec arch: Layer Super Class

```
@Entity class Document extends BaseAbstractEntity {  
    @Id public Integer getId() { ... }  
    ...  
}
```



## Zadanie 5

### Wspólne atrybuty

## **Dowiesz się:**

- Na czym polegają problemy z dostępem do wspólnych danych
- Poznasz standardowe sposoby obsługi
  - Blokowanie optymistyczne
  - Blokowanie zapisu/odczytu

- Blokowanie optymistyczne zezwala na posługiwanie się encją w wielu wątkach
  - W momencie zapisy właściciel „przestarzałej” wersji otrzymuje wyjątek *OptimisticLockException*
- Pole wersji
  - Adnotacja `@Version` – pole zwiększane przy każdym UPDATE
  - Może być tylko jedno w Encji
  - Aplikacja nie może go modyfikować
  - Może być Timestamp – jednocześnie „last update”

```
@Entity
public class User{
    //...
    @Version
    private int version;
}
```

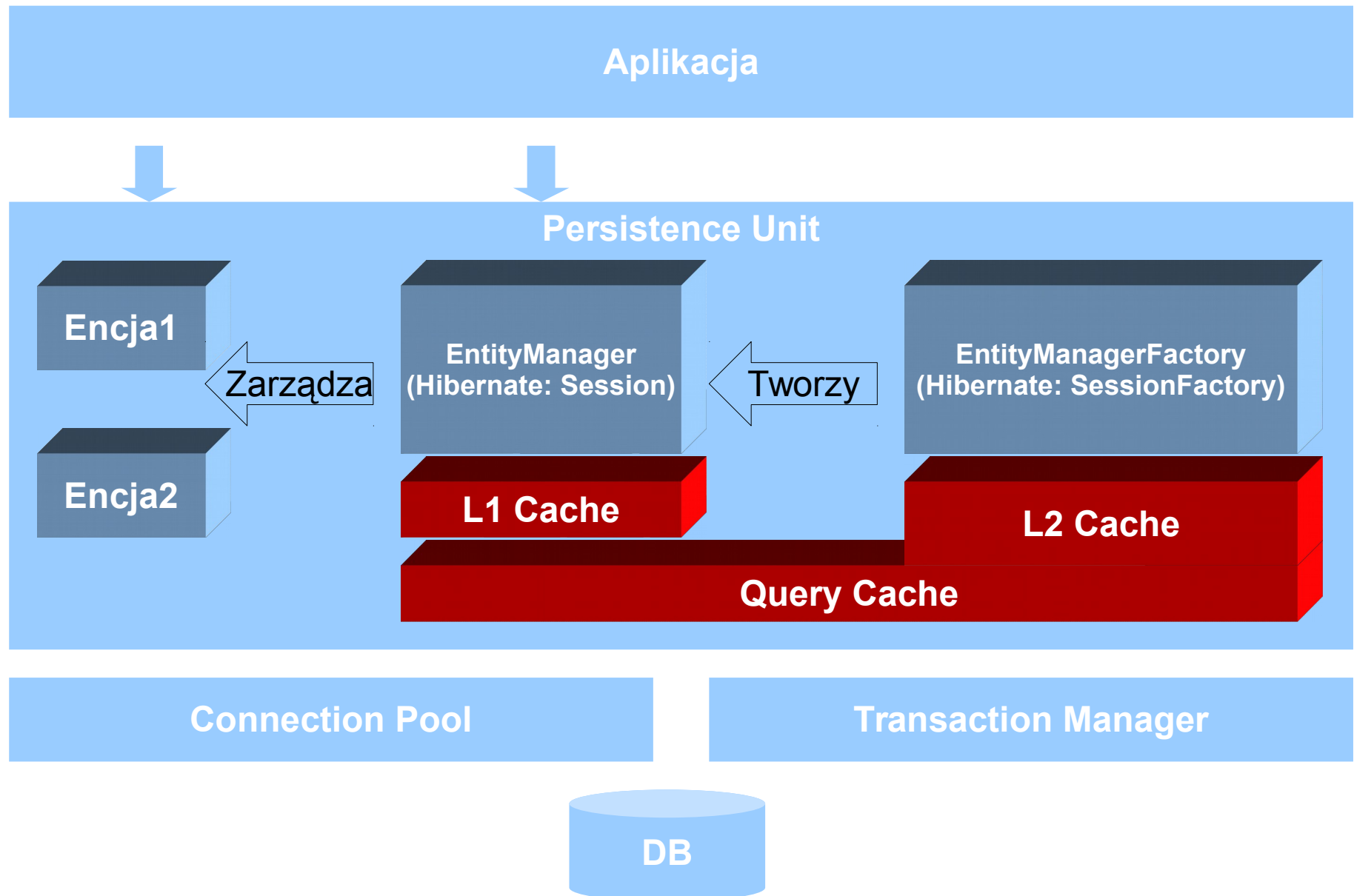
- Blokowanie encji na wyłączność kontekstu persystencji
- Działa na encjach **wersjonowanych** (i **może** działać na niewersjonowanych)
- `EntityManager.lock(Object o, LockModeType);`
  - READ
    - eliminuje niepowtarzalne odczyty
    - eliminuje brudne odczyty
  - WRITE
    - dodatkowo zwiększa pole `@Version`



## **Dowiesz się:**

- Jak są poziomy Cache
- Jak jest skonfigurować
- Jak z nich korzystać
- Jak współpracować w środowisku, w którym inne systemy również modyfikują dane

# Architektura JPA Cache



- L1 – Cache Encji pierwszego poziomu
  - **Domyślnie włączony w Hibernate**
  - Skojarzony z EntityManager (Session w Hibernate)
  - Optymalizuje operacje EntityManager - w obrębie „jednostki pracy”
    - Wielokrotne find() → jeden SELECT
    - wielokrotne merge() → jeden UPDATE w SQL
  - Silnik sprawdza istnienie encji w pierwszej kolejności w tym cache



- L2 - Cache Encji lub kolekcji drugiego poziomu
  - Skojarzony z EntityManagerFactory (SessionFactory w Hibernate)
  - Optymalizuje dostęp do encji lub kolekcji na poziomie całej aplikacji
    - find() odwołuje się do bazy tylko raz
  - Silnik sprawdza istnienie encji w drugiej kolejności w tym cache
    - W pierwszej kolejności jest sprawdzany L1

- Query Cache - Cache zapytań HQL
  - Ma sens dla zapytań
    - Wykonywanych często
    - Z tymi samymi parametrami
  - L2 cache **musi** być włączony

```
<property  
    name="hibernate.cache.provider_class">  
        org.hibernate.cache.EHCacheProvider  
</property>
```

```
<property  
    name="hibernate.cache.use_second_level_cache"  
    value="true"/>
```

- **Hashtable** – prosta implementacja w RAM
- **EHCache** (Easy Hibernate Cache) (org.hibernate.cache.EhCacheProvider)
  - Szybka, lekka, łatwa w użyciu
  - Wspiera cache read-only i read/write
  - Działa w pamięci lub na dysku
  - Nie obsługuje clusteringu
- **OSCache** (Open Symphony Cache) (org.hibernate.cache.OSCacheProvider)
  - Wydajna
  - Wspiera cache read-only i read/write
  - Działa w pamięci lub na dysku
  - Podstawowa obsługa clusteringu
- **SwarmCache** (org.hibernate.cache.SwarmCacheProvider)
  - Oparta o klastry
  - Wspiera cache read-only i nonstrict read/write
  - Odpowiednia dla systemów z przewagą odczytów nad zapisami
- **JBoss TreeCache** (org.hibernate.cache.TreeCacheProvider)
  - Wydajna
  - Wspiera replikacje i transakcyjność cache

- **Read-only**
  - Najbardziej wydajna
  - Encje są często czytane ale nigdy modyfikowane
- **Nonstrict read-write**
  - Encje są rzadko modyfikowane
- **Read-write**
  - Większy narzut
  - Encje są modyfikowane

```
@Entity
@Cache(
    usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class User {
    @OneToMany()
    @Cache(
        usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    public List<Address> addresses;
}
```

Należy pamiętać o wygaszaniu gdy inny system modyfikuje dane

```
sessionFactory.evict (User.class,  userId) ;  
sessionFactory.evict (User.class) ;  
sessionFactory.evictCollection ("User.addresses",  
    userId) ;  
sessionFactory.evictCollection ("User.adresses") ;
```

Należy zawsze stosować z L2 cache ponieważ

- Query cache nie przechowuje wartości
- Query cache przechowuje jedynie ID

```
<property  
    name="hibernate.cache.use_query_cache"  
    value="true"/>
```



```
@NamedQuery(  
    name="allusers",  
    query="FROM User",  
    hints={  
        @QueryHint(  
            name="org.hibernate.cacheable",  
            value="true") })  
@Entity  
public class User{..}
```

```
hibernateSession.createQuery("FROM User")  
    .setCacheable(true).list();
```

```
Criteria criteria = hibernateSession.createCriteria(Document.class);  
criteria.setCacheable(true);
```

```
public List<User> findUsersByAddress (Address a) {  
    return hibernateSession  
        .createQuery („FROM User u WHERE u.address = ?")  
        .setParameter (0, a)  
        .setCacheable (true)  
        .list ();  
}
```

- Parametry kwerendy/kryteriów będą przechowywane wraz z zależnościami w cache kwerend
  - Do czasu usunięcia danego wyniku z cache
- Parametry zapytania – obiekty czy Id?
  - Bardziej OO
  - Zdalne wywołanie == narzut

## Pobieranie encji gdy są rzeczywiście potrzebne

```
User user = (User)entityManager.getReference(User.class, 1L);
```

- Pobranie „uchwyty” - proxy do niezainicjowanego obiektu
- `getReference` **może** opóźnić pobranie encji do momentu, gdy będzie rzeczywiście użyta
- W razie braku encji o danym ID nie jest zwracany null
  - lecz `EntityNotFoundException` przy pierwszym dostępie
- Zastosowanie
  - Ciężkie wartości mogą ale muszą być potrzebne w algorytmie/aplikacji
  - Encja jest potrzebna jedynie w celu ustawienia jako wartość innej encji
    - Z poziomu technicznego: gdy potrzeba jedynie ID w poleceniu INSERT innej encji
- Niezainicjowana encja nie będzie miała sensu w stanie detached

