bottega
IT SOLUTIONS
www.bottega.com.pl

JPA i Hibernate

Dlaczego u mnie działa tak wolno?

Sławomir Sobótka

bottega
Akademia Java

bottega
IT SOLUTIONS
www.bottega.com.pl

Krótko o sobie...

Twórca oprogramowania – całościowe podejście

- Technologie
- Architektury
- Metodyki, podejścia, najlepsze praktyki
- Usability

Trener i konsultant

- Java EE
- Inżynieria oprogramowania

Agenda

- Architektura Hibernate
  - EntityManager i EntityManagerFactory – zasada działania
- Pułapki wydajności JPA
  - „n+1 Select problem” - wykrywanie i zapobieganie
  - Pułapki Lazy loadingu oraz zbyt chciwego pobierania danych
    - Nadmierne pobieranie danych
  - Optymalne mapowanie encji
    - Nadmiarowość pobierania danych
  - Problemy z leniwym ładowaniem pól
- Hibernate cache – niezastąpione rozwiązanie.
  - Idea działania i konfiguracja
  - Cache pierwszego poziomu
  - Cache drugiego poziomu
  - Mapowanie encji zorientowanie na cacheowanie
- Cache aplikacji
  - EntityManager w trybie rozszerzonym
  - Manualne opróżnianie sesji

Specyfika Java Persistence API

- 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!!!**

Mapowanie
Aspekty praktyczne

- Agregacjie 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
Aspekty praktyczne

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

Mapowanie zaawansowane
Operacje kaskadowe

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

- 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)**

DELETE_ORPHAN

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

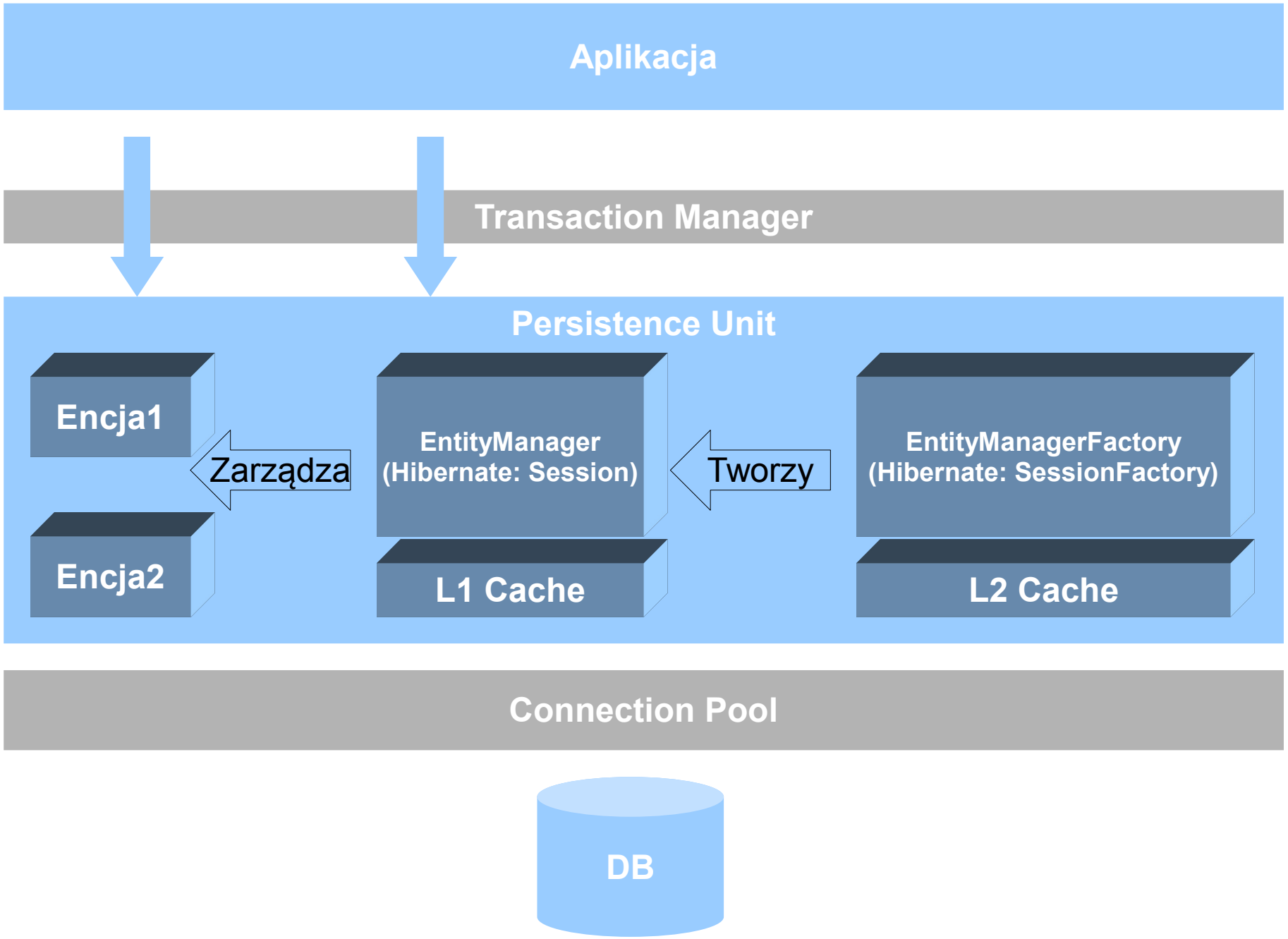
```
@Cascade({
org.hibernate.annotations.CascadeType.SAVE_UPDATE,
org.hibernate.annotations.CascadeType.DELETE_ORPHAN})
```

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/gorliwie 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;
}
```

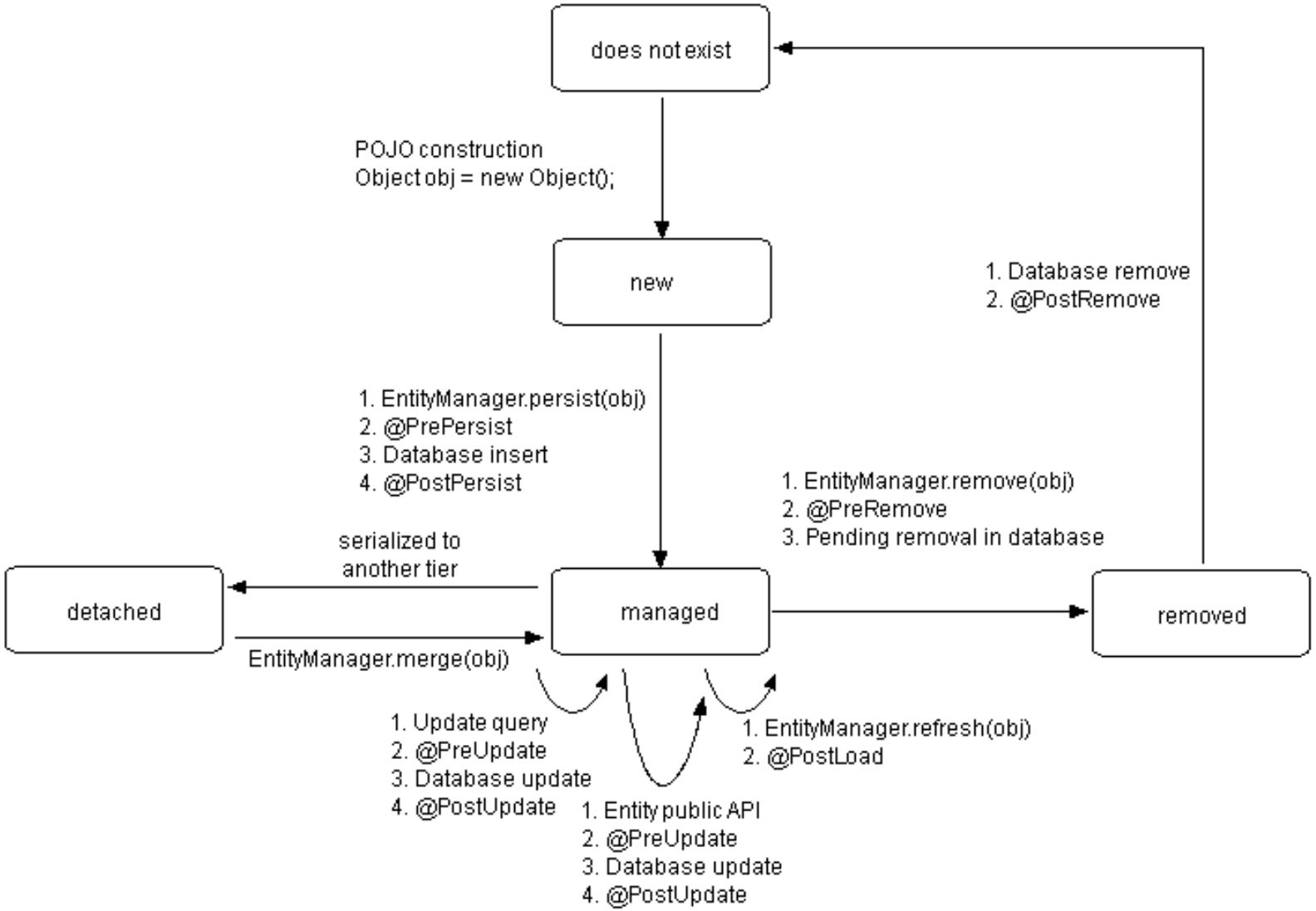
Architektura JPA



EntityManager

- 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

Cykl życia encji



Źródło: Oracle http://download.oracle.com/docs/cd/B32110_01/web.1013/b28221/undejbs003.htm#CIHCJGGJ

Entity Manager
Wybrane, typowe wyjątki API

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!

n+1 Select Problem

Dowiesz się:

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

n+1 Select Problem
Typowy przypadek

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()){
        //...
    }
}
```

n+1 Select Problem
Opis problemu

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 LazyInitalizationException

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 dal find()
  - **Dla zapytań o listę jet 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/łapczywe 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
```

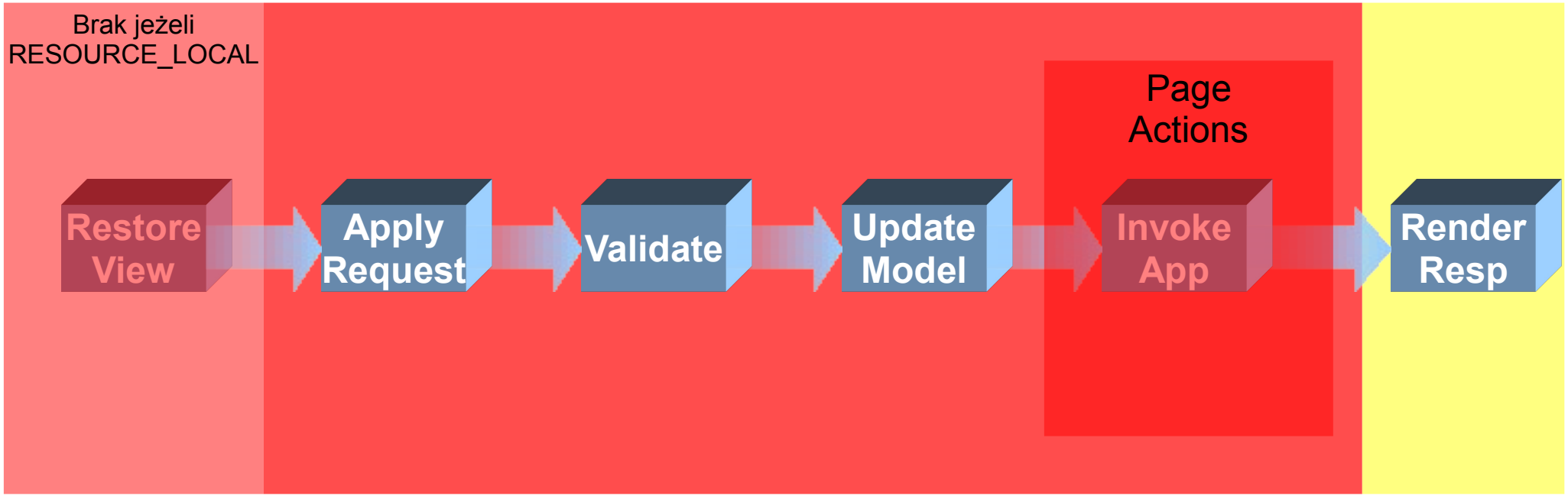
- Można próbować parametryzować ogólną metodę DAO
  - doklejenie JOIN FETCH
  - lepiej użyć Criteria API

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

if (…){
    criteria.setFetchMode(„addresses”, FetchModel.JOIN);
}
```

n+1 Select Problem
Wariant „utajony”

- 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

Wariant utajony – jak to działa
Transakcje globalne w Seam



Pierwsza transakcja

- Izolacja logiki biznesowej od renderingu
  - Jeżeli rendering spowoduje błąd
  - wówczas mimo tego zmiany biznesowe są utrwalone

Render Response

- Lazy Loading w osobnej transakcji
  - Jako całość
  - Bez otwierania nowej transakcji per operacja LL
- Blokada FLUSH
  - Rendering nie zmieni stanu bazy

n+1 Select Problem
Wykrywanie

- 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()
```

Pobieranie zbyt dużej ilości danych

Dowiesz się:

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

Pobieranie zbyt dużej ilości danych
Opis problemu

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 servisu 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

Pobieranie zbyt dużej ilości danych
Rozwiązanie – SQL

- Native SQL

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

```
sess.createSQLQuery("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...

Naiwna paginacja

Problem:
- Działanie standardowych paginatorów w JSF (np. RichFaces)
  - Przechowywanie całego wyniku zapytania w Sesji
  - Wyświetlenie na GUI jedynie wycinka
- Zaletą jest cache danych (jednorazowy dostęp do bazy)
- Wadą jest zajęcie pamięci i transfer dużej ilości danych z bazy
  - dla dużych list oraz gdy użytkownik nie przegląda kolejnych stron

Rozwiązanie:
- Własna paginacja
  - Implementacja własnego modelu
    - rozszerzenie SerializableDataModel dziedziczący po ExtendedDataModel
  - Kluczowa metoda
    - walk(FacesContext context, DataVisitor visitor, Range range, Object argument)
      - Range określa wiersz początkowy i ilość potrzebnych wierszy
      - Na tej podstawie można pobrać z bazy odpowiednią ilość wierszy

Count

```
int count = sess.createQuery("FROM Document")
                .list().size();
```

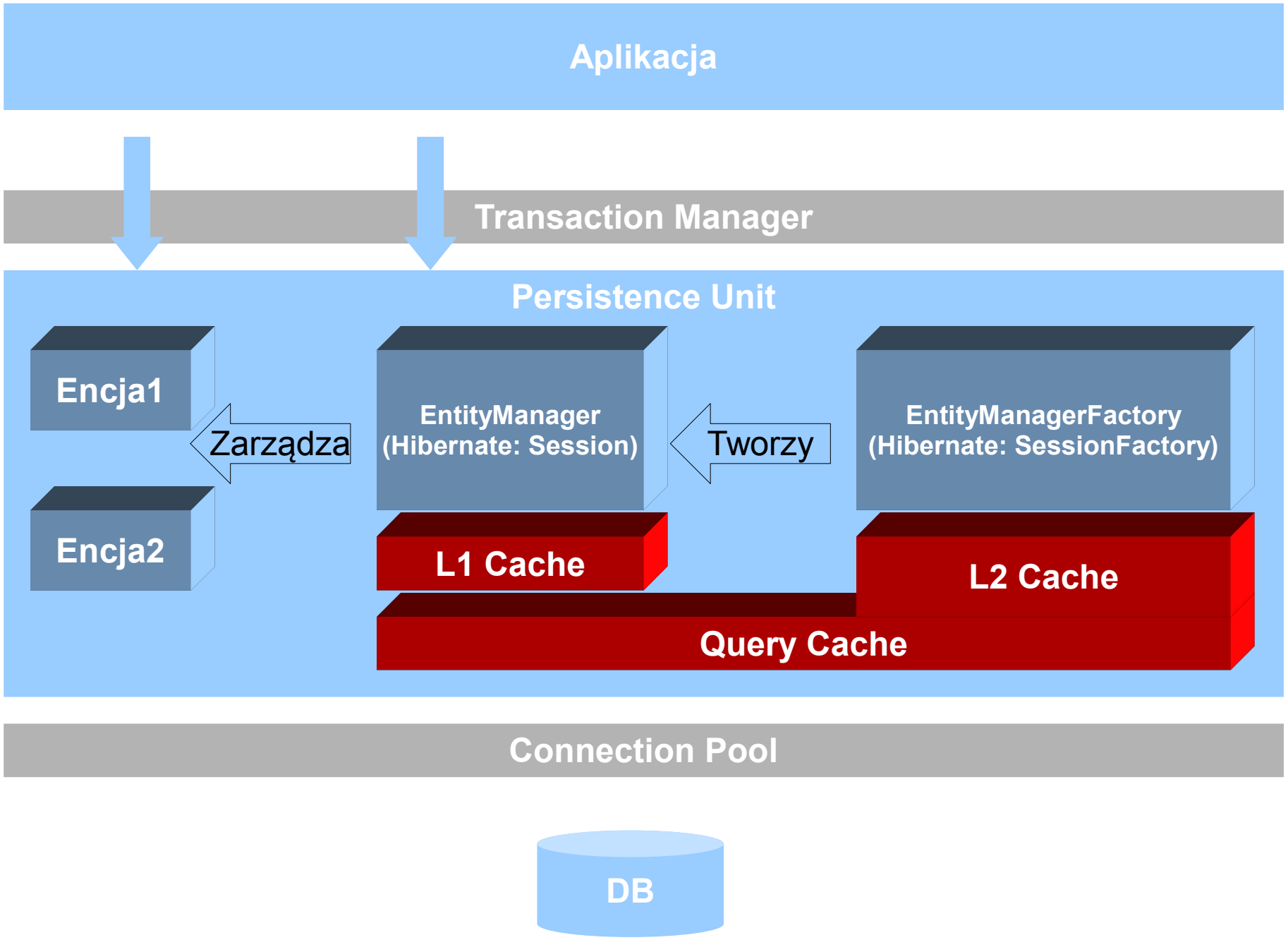
```
public int countDocuments(...){
    Criteria crit = createCriteria(...);
    return (Integer) crit.setProjection(Projections.rowCount())
                         .list().get(0);
}

public List<Document> searchDocuments(...){
    Criteria crit = createCriteria(...);
    return crit.list();
}

private Criteria createCriteria(...){
    Criteria criteria = hibernateSession.createCriteria(Document.class);
    …
    return criteria;
}
```

Hibernate Cache

Dowiesz się:

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

Architektura JPA Cache



Rodzaje Cache
Level 1

- 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

Rodzaje Cache
Level 2

- 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

Rodzaje Cache
Query

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

L2 Cache
Konfiguracja

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

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

L2 Cache Implementacje

- **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

L2 Cache
Strategie cache

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

L2 Cache przykład
Encje i kolekcje

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

Encja z listy również musi posiadać adnotację Cache

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");
```

Cache kwerend
Konfiguracja

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"/>
```

Cache kwerend
Przykłady

```
@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);
```

Cache kwerend
Pułapka – wyciek pamięci

```
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 „uchwytu” - 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

Pobieranie danych dla widoku w getterach

```
<h:dataTable value="#{bean.users}" var="_user" >
```

```
public List<Users> getUsers(){
  this.users = entityManager.... //lub DAO/Serwis
  return this.users;
}
```

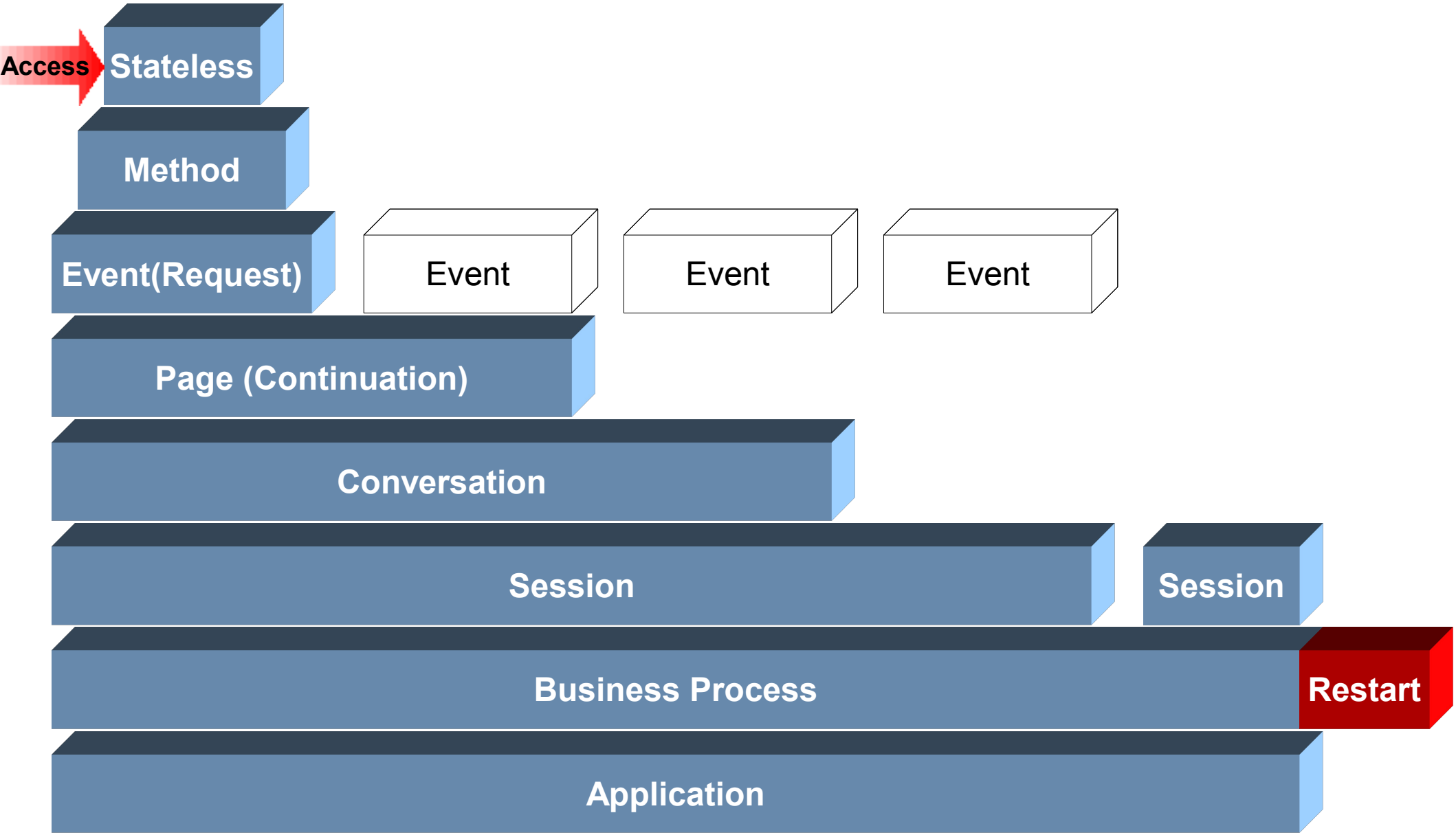
- JSF woła gettery wielokrotnie (w kilku fazach obsługi żądania)
  - **Operacje na bazie danych zostaną wykonane wiele razy**
- Model dla prezentacji należy inicjować wcześniej
  - Styl PULL – metoda obsługująca akcję
  - Styl PUSH – inicjalizacja obiektu
    - @PostConstruct - JSR-250 (Common Annotation for Java Platform)
    - @Create – Seam
  - @Factory – gdy obiekt nie istnieje
  - @Unwrapp – za każdym razem

Konwersacje
Wstęp do aplikacyjnego cache

Dowiesz się:

- Na czym polega mechanizm konwersacji
- W jakiś sposób Seam zarządza EntityManagerem w konwersacji
- Poznasz tryb rozszerzony EntityManagera
- Poznasz mechanizm ręcznego opróżnainia EntityManagera
- Poznasz zależność EntityManagera od cyklu życia transakcji
- Dowiesz się na jakiej zasadzie działa dołączania odłączonych Encji do EntityManagera
- Jak konwersacje współpracują z Kontekstem Persystencji działającym w trybie rozszerzonym
- Dowiesz się jak wykorzystać wymienione mechanizmy
- Zrozumiesz na jakiej zasadzie działa cache aplikacyjny

Zasięg Komponentów (przykład Seam)



Konwersacje

- Unit of work
  - Może przekładać się na Use Case
  - Przechowuje stan pomiędzy żądaniami
  - Posiada własny ID, dzięki któremu jest rozróżnialny w zakładkach
  - Stanowi cache procesu – optymalizacja dostępu do danych
- Mogą przechowywać
  - Dane nieutrwalane (transient)
  - Encja odłączone (detached)
  - Encja zarządzane (managed)
  - Zasoby
    - EntityManager – również w stanie rozszerzonym

Persistence Context
Założenia

EntityManager (Session w Hibenrate)
- Został zaprojektowany tak aby istnieć przez dłuższy czas
- Dłuższy niż transakcja
- Nie należy utożsamiać go z chwilowym połączeniem z bazą

EM nie jest threadsafe
- Nie należy go współdzielić pomiędzy równoległymi wątkami

Persistence Context Zarządzanie

- Zarządzanie przez kontener JEE
  - EntityManager jest wstrzykiwany do SFSB
  - Zasięg życia EM jest taki jak „posiadającego go” komponentu
  - Skomplikowane reguły propagacji
- Zarządzanie przez Seam
  - Wstrzykiwanie do POJO/EJB (@In)
  - EM jest takim samym komponentem jak pozostałe
  - Znajduje się w kontekście konwersacji
    - Również w Konwersacji tymczasowej
      - Obejmującej request
      - Lazy Loading działa w warstwie widoku:)
      - Z tego powodu należy śledzić logi!!!
  - Zasięg życia konwersacji
    - Bez związku z zasięgiem innego komponentu
  - Możliwość wstrzykiwania do wielu komponentów – bez ograniczeń

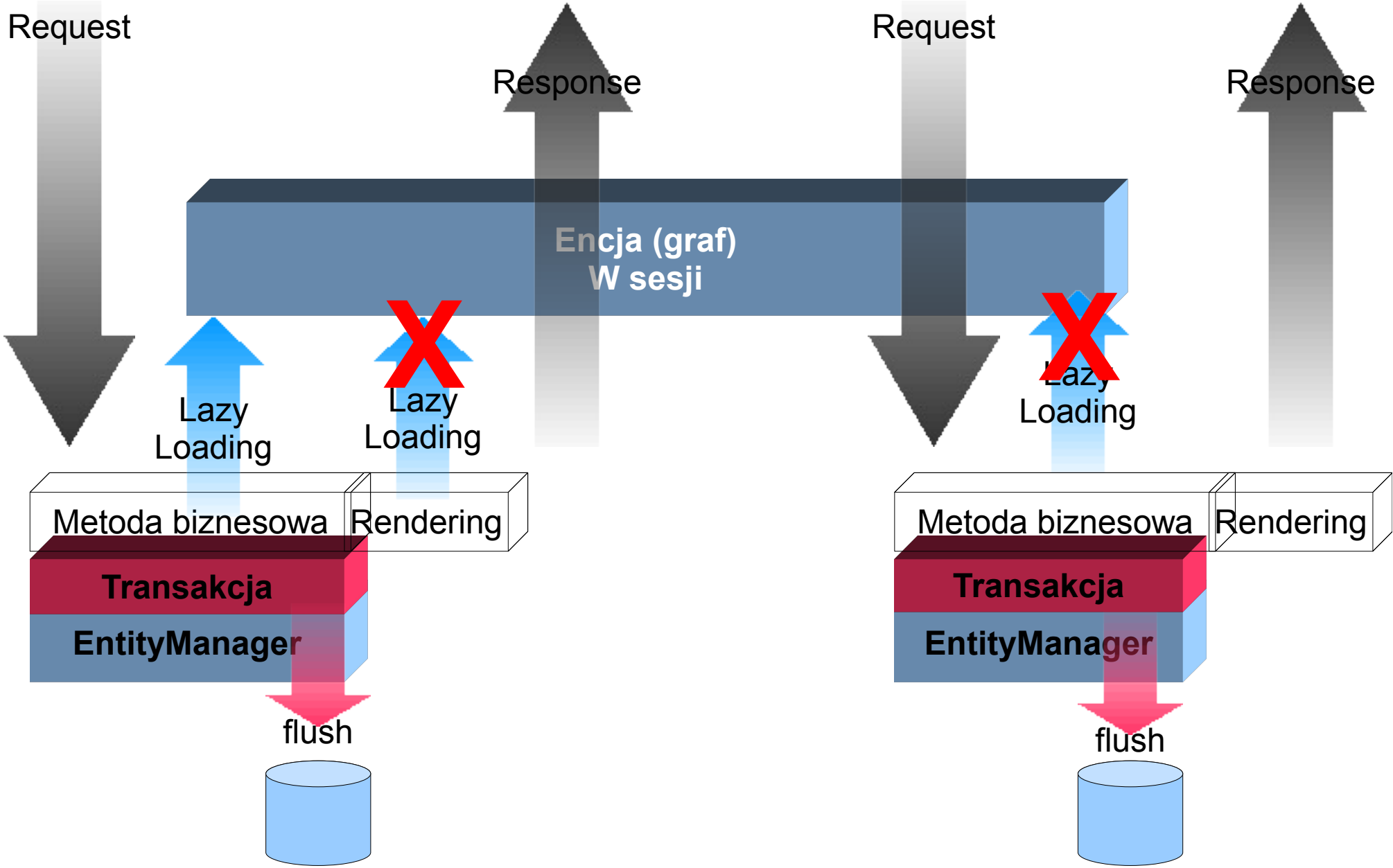
Persistence Context
Tryby pracy

- Tryb transakcyjny (domyślny dla JEE)
  - Zasięg sesji EntityManagera jest taki sam jak transakcji JTA
  - Zwykle jest to wywołanie jednej metody EJB
  - Czyli jeden request (ale tylko w warstwie biznesowej)
  - Po zamknięciu Encją są w stanie Detached
- Tryb EXTENDED (domyślny w Seam)
  - Zasięg sesji EntityManagera jest szerszy...
  - Encje są cały czas w stanie Managed
    - Redukcja komunikacji z bazą - ponowne odczyty (merge)
    - Działający Lazy Loading
    - Należy pamiętać o Optimistic Locking (@Version)!
  - Działa jako „naturalny” Cache (L1)

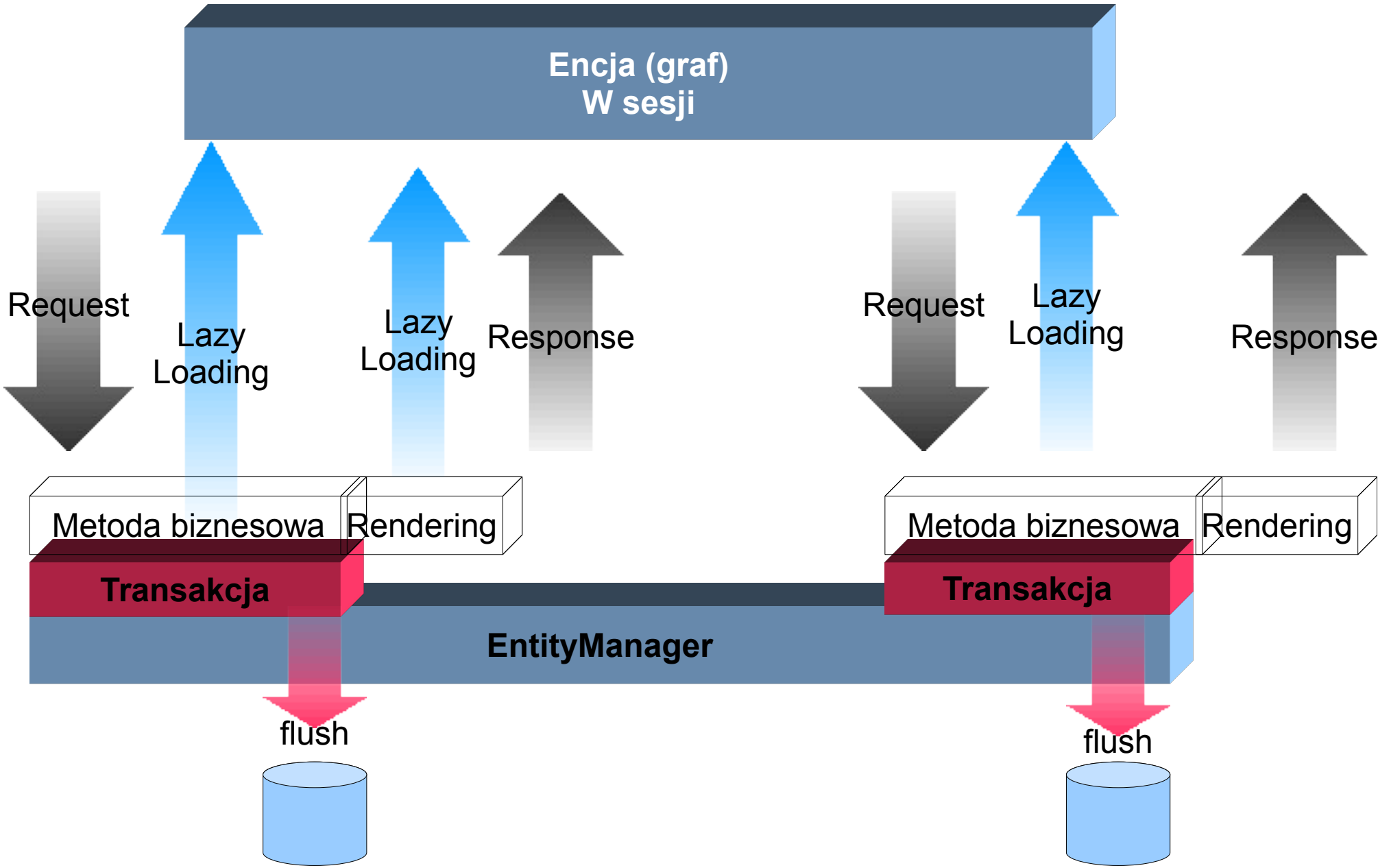
Tryb rozszerzony w JEE
Przykład

```
@Stateful
public class UserAgentBean implements UserAgent {

   @PersistenceContext(
      type = PersistenceContextType.EXTENDED)
   private EntityManager em;

}
```

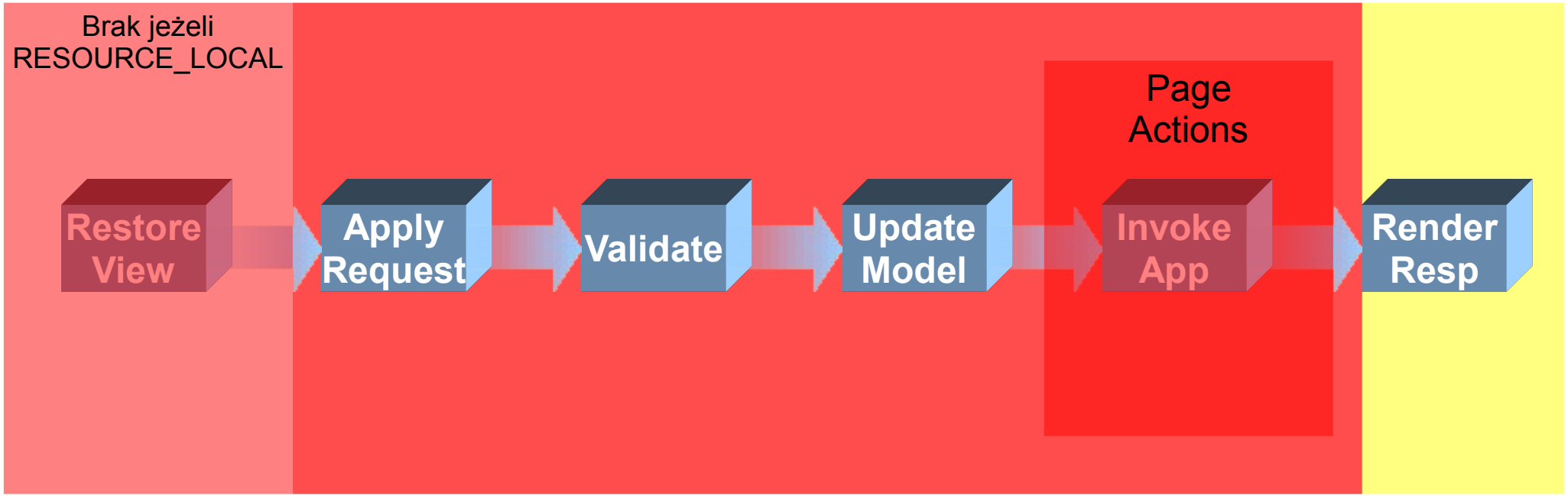
EntityManager
Tryb transakcyjny



EntityManager
Tryb rozszerzony



Transakcje globalne



Pierwsza transakcja

- Izolacja logiki biznesowej od renderingu
  - Jeżeli rendering spowoduje błąd
  - wówczas mimo tego zmiany biznesowe są utrwalone

Render Response

- Lazy Loading w osobnej transakcji
  - Jako całość
  - Bez otwierania nowej transakcji per operacja LL
- Blokada FLUSH
  - Rendering nie zmieni stanu bazy

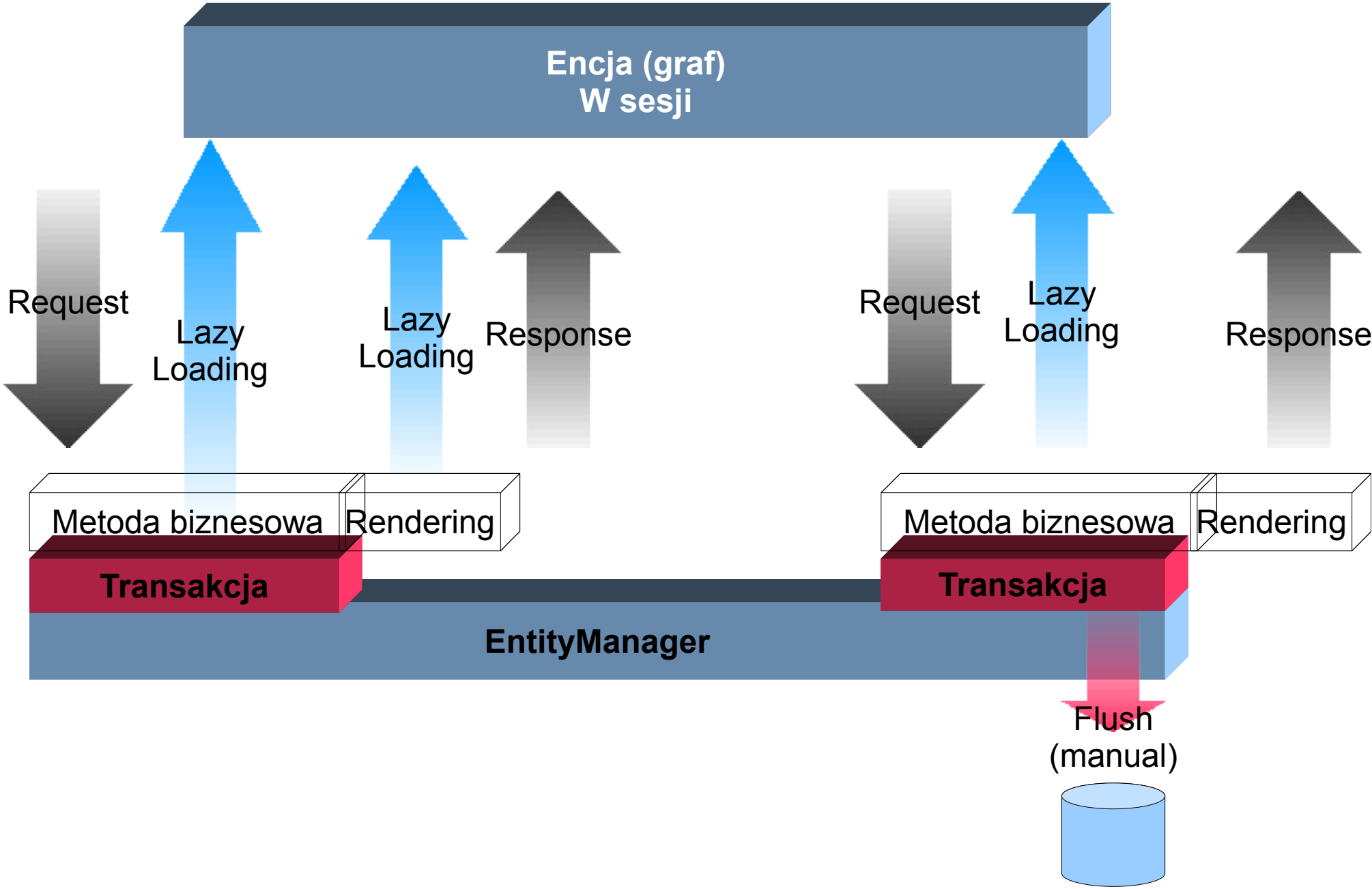
Transakcje aplikacyjne
Wstęp

- Kontekst perystencji w trybie rozszerzonym rozciąga się na kilka requestów
- Podczas każdego z requestów tworzona jest transakcja
- Kontekst perystencji jest do niej „podczepiany”
- Po każdorazowym zamknięciu transakcji następuje flush kontekstu perystencji – zapis zmienionych (dirty) Encji
  - Ponieważ w JPA wyspecyfikowano dwie polityki
    - AUTO – kontekst opróżnia się w momentach, które uzna za najbardziej stosowne
    - COMMIT – kontekst opróżnia się przy komitowaniu transakcji
- Skutek
  - Mimo że konwersacja trwa wiele requestów
  - Po każdym z nim zmienia się stan bazy
  - W dalszej części konwersacji może wystąpić błąd
  - Ale dane są już zapisane

Transakcje aplikacyjne
Zastosowanie Hibernate

- Istnieje możliwość odroczenia opróżniania kontekstu persysyencji
  - Do momentu wywołania metody flush()
  - Nawet mimo tego, że transakcja jest zamykana
- Specyfika Hibernate
  - Session jako kontekst persystencji
  - Lub nawet jako implementację JPA (używając EntityManager)
- Rozpoczynając konwersację należy włączyć ręczny tryb opróżniania
- Wówczas w każdym request kontekst jest podłączany do nowej transakcji
  - Działa LL
  - Ale na zakończenie transakcji kontekst nie zapisuje zmian
  - Zrobi to dopiero po wywołaniu flush()
- Taki kontekst działa jako cache poleceń zapisu danych
  - Czyli rodzaj „transakcji” - lecz nie bazodanowej a aplikacyjnej

EntityManager
Tryb rozszerzony + Manual Flush



Transakcje aplikacyjne
Przykład

```
@Name("userWizardAction")@Scope(ScopeType.CONVERSATION)
@Transactional
public class UserWizardAction implements Serializable {
    @In private EntityManager entityManager;
    @Out private User user;

    @Begin(flushMode = FlushModeType.MANUAL)
    public void addUser() {
        user = ...
        entityManager.persist(user);
    }
    public String addBasicData() {
        ...
        return "next";
    }
    @End public String save() {
        entityManager.flush();
        return "saved";
    }
}
```

bottega
IT SOLUTIONS
www.bottega.com.pl

Dziękuję za uwagę

więcej...

http://art-of-software.blogspot.com

slawomir.sobotka@bottega.com.pl

Sławomir Sobótka

bottega
Akademia Java

Photo Credits

- http://merlin.pl/Spiewajacy-zolw_Fisher-Price,images_big,6,FPM4925.jpg