

WIEDZA KTÓRA ZMIENIA

Enterprise JavaBeans 3.0

Inżynieria Oprogramowania
Altkom Akademia

Cele szkolenia

- Wprowadzenie do koncepcji:
 - programowania komponentowego
 - obiektów rozproszonych
 - asynchronicznego przesyłania komunikatów
 - usług sieciowych
- Zapoznanie z podstawami architektury:
 - Enterprise JavaBeans 3.0
 - Java Persistence API 1.0
- Omówienie praktycznych technik wytwarzania skalowalnych i przenośnych rozwiązań biznesowych

Zakres materiału szkoleniowego

- Moduł 1 – Wprowadzenie
- Moduł 2 – Podstawy architektury
- Moduł 3 – Utrwalanie i usługa *EntityManager*
- Moduł 4 – Odwzorowania obiektowo-relacyjne
- Moduł 5 – Zależności zachodzące między encjami
- Moduł 6 – Podstawy języka JPQL
- Moduł 7 – Wywołania zwrotne i klasy nasłuchujące
- Moduł 8 – Komponenty sesyjne
- Moduł 9 – Komponenty sterowane komunikatami
- Moduł 10 – Usługa *TimerService*
- Moduł 11 – Obsługa transakcji
- Moduł 12 – Bezpieczeństwo aplikacji

Przedstawienie uczestników szkolenia

- Imię i nazwisko, firma
- Rodzaj wykonywanej pracy
- Doświadczenie w programowaniu
- Oczekiwania dotyczące szkolenia

Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

1

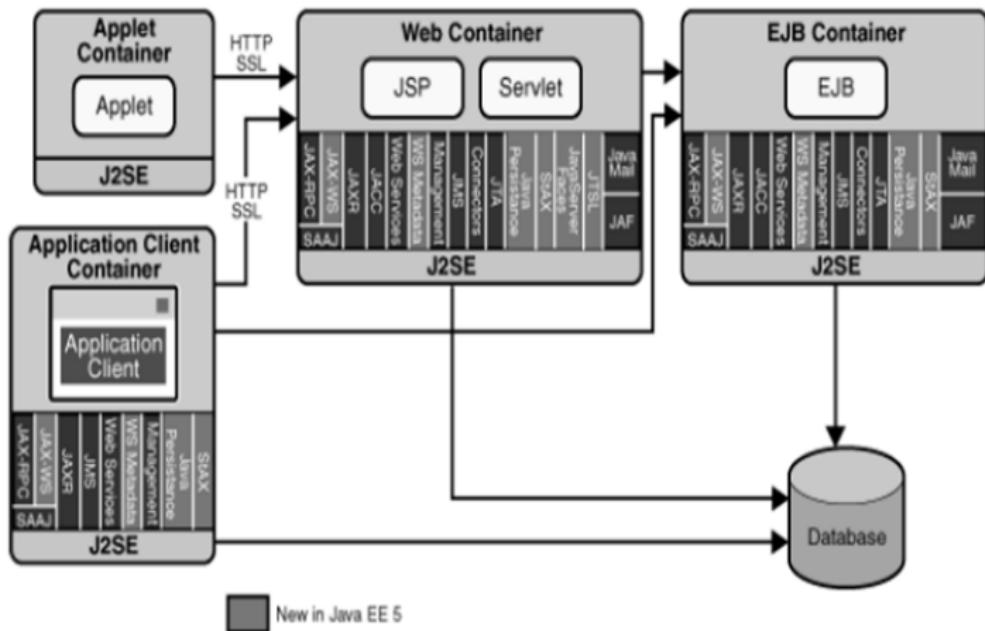
Wprowadzenie

- platforma Java Enterprise Edition
- architektura komponentowa
- specyfikacja Enterprise JavaBeans
- ekosystem EJB

Java Enterprise Edition

- Jest platformą przeznaczoną do tworzenia i uruchamiania złożonych, wielowarstwowych, rozproszonych i skalowalnych aplikacji korporacyjnych
- Zbiór specyfikacji dotyczących powiązanych wzajemnie technologii, których celem jest ułatwianie procesu realizacji takich zadań jak:
 - wysoka wydajność i dostępność
 - bezpieczeństwo
 - niezawodność
 - skalowalność
 - transakcyjność
- Przyczynia się do większej otwartości systemów korporacyjnych i pozwala wybierać pomiędzy różnymi dostawcami oprogramowania zgodnego z platformą Java EE

Platforma Java Enterprise Edition



Co to jest komponent?

- Przez pojęcie **komponentu** rozumie się wyodrębniony (hermetycznie zamknięty) fragment kodu, implementujący pewien zbiór interfejsów
- Komponenty nie są samodzielnymi aplikacjami i mogą być używane wyłącznie w specjalnym środowisku uruchomieniowym, które stanowi serwer aplikacji

Zalety architektury komponentowej

- Pozwala na szybsze tworzenie aplikacji – na zasadzie składania układanki z gotowych klocków
- Komponenty stanowią swoistą “czarną skrzynkę” z dobrze zdefiniowanym interfejsem (używamy ich, abstrahując od sposobu rozwiązywania problemu)
- Budowanie systemów wymaga mniejszej wiedzy specjalistycznej
- Istnieje rozwinięty rynek komponentów gotowych
- Zapewnia dużą reużywalność i przenośność kodu

Rola serwera aplikacji

- Serwer aplikacji stanowi **środowisko uruchomieniowe** dla komponentów takich jak: serwlety, strony JSP, komponenty EJB
- W stosunku do aplikacji opartych o Java EE pełni analogiczną rolę, jak system operacyjny Windows dla aplikacji typu *.exe
- Udostępnia szereg usług systemowych takich, jak np.: obsługa transakcji, autoryzacja dostępu, utrwalanie, itd.
- Przykłady serwerów aplikacji:
 - JBoss
 - WebLogic
 - WebSphere

Specyfikacja Enterprise JavaBeans

- Wyodrębniony fragment platformy Java EE, definiujący **standardowy model wytwarzania i wdrażania komponentów reprezentujących procesy biznesowe**
- Specyfikacja łącząca komponenty serwerowe z technologią obiektów rozproszonych, asynchronicznym przesyłaniem komunikatów, usługami sieciowymi i utrwalaniem danych
- Zapewnia skalowalność, transakcyjność, a także bezpieczeństwo w środowiskach wielużytkownikowych
- Pozwala na wdrażanie i uruchamianie aplikacji na dowolnych platformach serwerowych obsługujących specyfikację EJB

Zalety i wady technologii

• Zalety:

- specyfikacja EJB jest opublikowana i dostępna nieodpłatnie
- stanowi standard przemysłowy
- umożliwia szybkie tworzenie aplikacji (serwer zapewnia środowisko uruchomieniowe i dostarcza odpowiednich usług)
- istnieje wielu dostawców serwerów aplikacji
- pozwala na stosowanie podejścia "*Write Once, Run Anywhere*"

• Wady:

- komponenty EJB mogą być tworzone wyłącznie w Javie

Utrwalanie danych

- Warstwa utrwalania danych odpowiada za odwzorowywanie obiektów w bazie danych, umożliwia ich przeszukiwanie, odczytywanie, modyfikację i usuwanie **bez konieczności korzystania z konkretnego interfejsu API** takiego, jak np. JDBC
- Począwszy od EJB 3.0 abstrakcja ta została przeniesiona do odrębnej specyfikacji *Java Persistence API*, definiującej kompletny zbiór reguł odwzorowań obiektowo-relacyjnych (ORM), dzięki czemu możliwe jest przenoszenie komponentów encyjnych (*entity beans*) pomiędzy rozwiązaniami różnych producentów

Asynchroniczne przesyłanie komunikatów

- Technologia EJB oferuje obsługę asynchronicznego przesyłania komunikatów, umożliwiającą dwóm lub większej liczbie aplikacji wymianę informacji
- Dane przesyłane w ten sposób mogą mieć dowolną formę i są przekazywane za pośrednictwem sieci z wykorzystaniem oprogramowania typu MOM (*Message-Oriented Middleware*)
- Produkty MOM gwarantują prawidłowe dostarczanie komunikatów do rozproszonych składników aplikacji, a także biorą na siebie rozwiązywanie problemów związanych z:
 - tolerancją błędów
 - równoważeniem obciążenia
 - skalowalnością
 - przetwarzaniem transakcyjnym

Asynchroniczne przesyłanie komunikatów

- Aplikacje wymieniają dane za pomocą kanałów wirtualnych (węzłów docelowych), dzięki czemu wyeliminowana została konieczność wiązania aplikacji generującej komunikat z aplikacją, która te komunikaty odbiera
- Dzięki temu odbiorcy i nadawcy w żaden sposób nie są od siebie uzależnieni – mogą wysyłać i odbierać wiadomości w dowolnym momencie
- Technologia EJB 3.0 pozwala na obsługę komunikatów Javy JMS oraz specjalnych komponentów sterowanych komunikatami

Architektura JCA

- Poczynając od EJB 2.1 model komponentów sterowanych komunikatami został rozszerzony, dzięki czemu mogą one współpracować z dowolnym systemem przesyłania komunikatów zgodnym z *Java Connector Architecture* (JCA 1.5)
- JCA definiuje przenośny model programowania interfejsów informacyjnych dla systemów korporacyjnych
- Każdy kontener EJB obsługujący konektor JCA może współpracować z dowolnymi zasobami zgodnymi z tym standardem (analogia do portu USB używanego w komputerach)

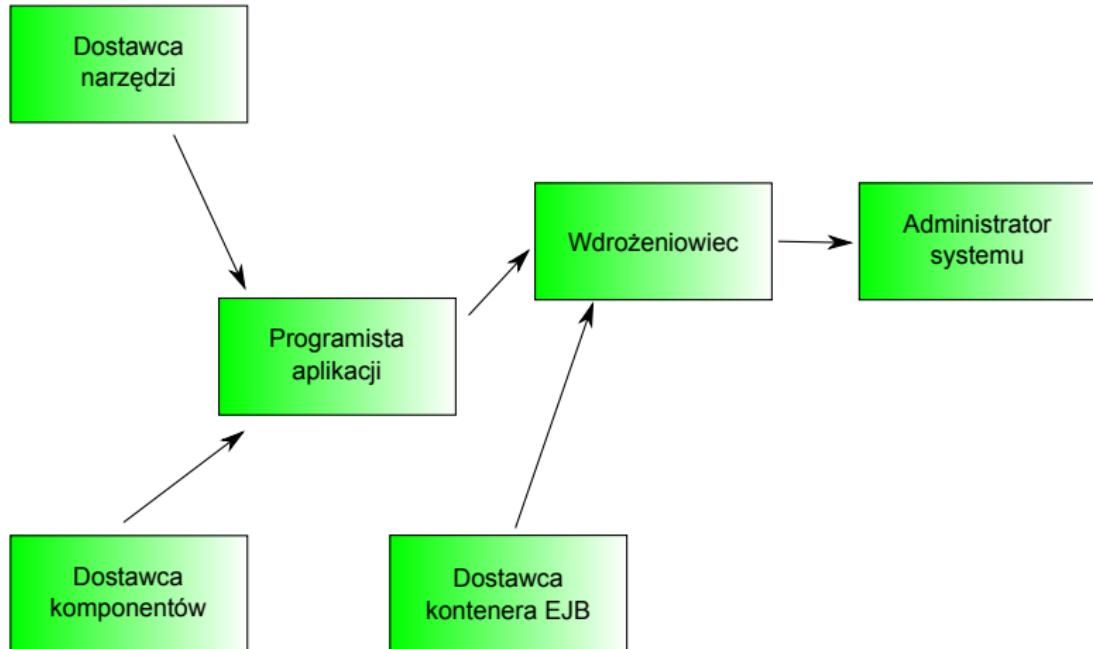
Usługi Web Services

- Usługi sieciowe są aplikacjami sieciowymi wykorzystującymi protokół SOAP i język WSDL do wymiany informacji w formie dokumentów XML
- Oferują faktyczną niezależność od platformy
- Specyfikacja EJB 3.0 umożliwia udostępnianie komponentów biznesowych w formie usług *WebServices*, co pozwala na ich wywoływanie zarówno przez pozostałe aplikacje typu Java EE, jak i inne, utworzone na bazie odmiennych technologii

Ekosystem EJB

- Stosowanie EJB wymaga współpracy 6 różnych ekspertów. Są to:
 - dostawca komponentu
 - programista aplikacji
 - wdrożeniowiec
 - administrator systemu
 - dostawca serwera aplikacji (kontenera)
 - sprzedawca narzędzi

Ekosystem EJB



Role w ekosystemie

- Dostawca komponentów
 - oferuje gotowe komponenty EJB, np. kalkulator podatkowy, analizator ryzyka ubezpieczeniowego
- Programista aplikacji
 - pełni rolę architekta w zakresie doboru zewnętrznych komponentów EJB potrzebnych do rozwiązania konkretnego problemu
 - implementuje własne komponenty
 - pisze kod integrujący komponenty różnych dostawców
 - buduje interfejs użytkownika w postaci aplikacji Swing, appletów lub serwletów, czy stron JSP

Role w ekosystemie

- Wdrożeniowiec
 - odpowiada za zabezpieczenie aplikacji, np. zaporą ogniową
 - integruje aplikację z systemami zewnętrznymi, np. LDAP
 - dokonuje wyboru odpowiedniego sprzętu
 - przeprowadza konfigurację
- Administrator systemu
 - odpowiada za utrzymanie i monitorowanie działania aplikacji

Role w ekosystemie

- Dostawca kontenera
 - tworzy kontener zgodny ze specyfikacją
 - odpowiada za spełnienie wymagań funkcjonalnych:
 - równoważenie obciążenia,
 - odporność na awarie,
 - grupowanie,
 - skalowalność i buforowanie
 - wspiera zadania administratora – oferuje narzędzia diagnostyczne i monitorujące
- Sprzedawca narzędzi
 - dostarcza narzędzi do tworzenia i modelowania komponentów EJB, np. *JBuilder*, *NetBeans*, *BEA WebLogic WorkShop*

Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury**
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

2

Podstawy architektury

- przedstawienie i charakterystyka komponentów EJB
- dostęp z poziomu klientów umieszczonych poza kontenerem
- mechanizmy służące do zarządzania zasobami
- podstawowe usługi oferowane przez serwer aplikacji

Klasyfikacja komponentów

- Specyfikacja EJB umożliwia budowanie trzech zasadniczo różnych typów komponentów
- Każdy przeznaczony jest do innego typu zadań i charakteryzuje się odmiennymi właściwościami
- Ich podział wygląda następująco:
 - komponenty encyjne
 - komponenty sesyjne
 - stanowe
 - bezstanowe
 - komponenty sterowane komunikatami

Komponenty encyjne (*entity beans*)

- Służą do modelowania elementów biznesowych, które można wyrazić w formie rzeczowników
 - np. klient, zamówienie, raport, karta płatnicza
- Pozwalają opisywać obiekty świata rzeczywistego, które będą utrwalane w formie rekordów relacyjnej bazy danych (po wcześniejszym odwzorowaniu)
- Poczynając od wersji EJB 3.0 stworzenie **komponentu encyjnego** sprowadza się do napisania **zwykłej klasy Javy oznaczonej odpowiednimi adnotacjami**
- W odróżnieniu od innych rodzajów komponentów, encje mogą być serializowane i przesyłane za pośrednictwem sieci

Komponenty encyjne (*entity beans*)

- W celu implementacji komponentu encyjnego należy zdefiniować klasę opisującą żądany element biznesowy i określić specjalne pole spełniające funkcję **identyfikatora** – klucza głównego klasy
- Klucz zapewnia powiązanie z odpowiednim rekordem w bazie danych, a także pełni rolę identyfikatora egzemplarza komponentu składowanego w pamięci
- Klasa komponentu encyjnego nie powinna zawierać złożonej logiki biznesowej – ma służyć wyłącznie jako obiektowa reprezentacja trwałych danych

Komponenty encyjne (*entity beans*)

- Obowiązkowymi adnotacjami każdej klasy komponentu encyjnego są:

`@javax.persistence.Entity`

informuje usługę utrwalania o tym, że dana klasa jest encją

`@javax.persistence.Id`

służy do określenia klucza głównego, który wiąże encję z odpowiednim rekordem w bazie danych

- Istnieje szereg innych adnotacji pozwalających na definiowanie skomplikowanych mechanizmów odwzorowań obiektowo-relacyjnych, które zostaną przedstawione w dalszej części kursu

Przykład klasy komponentu encyjnego

```
@Entity
public class Klient implements Serializable {
    @Id
    private long id;
    private String imie;
    private String nazwisko;

    public long getId() {
        return id;
    }

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

    // + pozostałe metody dostępowe
}
```

Jednostka utrwalania

- Klasy definiujące komponenty encyjne są grupowane w ramach skończonych zbiorów nazywanych **jednostkami utrwalania** (*persistence units*)
- Każda jednostka utrwalania jest zarządzana przez specjalną usługę *EntityManager* i musi być identyfikowana w sposób, który umożliwia stosowanie jednoznacznych odwołań w kodzie aplikacji, a także pozwala na powiązanie z konkretną bazą danych
- Potrzebne informacje określane są za pomocą adnotacji i pliku *persistence.xml* umieszczonego w katalogu *META-INF*
- Całość może zostać dodatkowo spakowana do archiwum *jar*

Przykład prostego pliku *persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
    <persistence-unit name="Clients" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>java:/MySqlDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto"
                      value="create-drop"/>
        </properties>
    </persistence-unit>
</persistence>
```

Komponenty sesyjne (*session beans*)

- Służą do modelowania elementów, które można wyrazić za pomocą czasowników, a więc wykorzystywanych do realizacji procesów biznesowych
 - np. system płatniczy, algorytm wyceny
- W ramach komponentów sesyjnych wyróżniamy dwa podtypy:
 - stanowe (*stateful*)
 - bezstanowe (*stateless*)

Komponenty sesyjne (*session beans*)

- Komponenty sesyjne stanowe (*stateful*)
 - posiadają wewnętrzny stan
 - podlegają pasywacji i aktywacji
 - jeden komponent obsługuje jednego użytkownika w ramach sesji
- Komponenty sesyjne bezstanowe (*stateless*)
 - nie posiadają wewnętrznego stanu
 - nie podlegają pasywacji i aktywacji
 - jeden komponent obsługuje jednego użytkownika w ramach jednego wywołania metody

Komponenty sterowane komunikatami (*message-driven beans*)

- Podobnie jak komponenty sesyjne, modelują procesy biznesowe, które mogą być wyrażone poprzez czasowniki
- Stanowią węzły integrujące różne aplikacje, mogące przesyłać wiadomości za pośrednictwem systemu JMS
- W ramach przetwarzania otrzymanych komunikatów, mogą wywoływać inne komponenty EJB
- Przykład:
 - komponent otrzymujący komunikaty w systemie *workflow*

Implementacja komponentów sesyjnych i sterowanych komunikatami

- Aby zaimplementować komponent sesyjny lub sterowany komunikatami należy w pierwszej kolejności zdefiniować odpowiednie interfejsy
- Do dyspozycji mamy:

interfejs zdalny	oznaczany adnotacją <code>@javax.ejb.Remote</code> , definiuje metody biznesowe, które mają być dostępne z poziomu aplikacji znajdującej się poza kontenerem EJB
interfejs lokalny	oznaczony adnotacją <code>@javax.ejb.Local</code> , definiuje metody biznesowe dostępne dla pozostałych komponentów należących do tego samego kontenera
interfejs punktu końcowego	oznaczany adnotacją <code>@javax.jws.WebService</code> , definiuje metody, które mogą być wywoływanie za pośrednictwem protokołu SOAP
interfejs komunikatów	definiuje metody pozwalające na przetwarzanie komunikatów

Implementacja komponentów sesyjnych i sterowanych komunikatami

- Klasa komponentu sesyjnego:
 - zawiera logikę biznesową niezbędną do realizacji określonych zadań i implementuje przynajmniej jeden z interfejsów (zdalny, lokalny lub punktu końcowego)
 - dodatkowo musi być oznaczona, w zależności od typu, adnotacją `@javax.ejb.Stateful` lub `@javax.ejb.Stateless`
- Klasy komponentów sterowanych komunikatami:
 - implementują odpowiedni dla siebie interfejs (w zależności od typu obsługiwanej usługi) i są oznaczane adnotacją `@javax.ejb.MessageDriven`

Przykład interfejsu zdalnego

```
@Remote  
public interface RecepcajRemote {  
  
    public void dodajKlienta(Klient klient);  
    public void usunKlienta(long id);  
    public Klient znajdzKlienta(long id);  
    public void aktualizujDaneKlienta(Klient klient);  
  
}
```

Przykład klasy komponentu sesyjnego

```
@Stateless  
public class RecepcaBean implements RecepcaRemote {  
    public void dodajKlienta(Klient klient) {  
        // logika biznesowa  
    }  
  
    public void usunKlienta(long id) {  
        // logika biznesowa  
    }  
  
    public Klient znajdzKlienta(long id) {  
        // logika biznesowa  
        return klient;  
    }  
  
    public void aktualizujDaneKlienta(Klient klient) {  
        // logika biznesowa  
    }  
}
```

Dostęp do komponentów z poziomu aplikacji umieszczonej poza kontenerem

- Klienci nigdy nie posiadają bezpośredniego dostępu do egzemplarza klasy komponentu
- Korzystają z metod zdefiniowanych w odpowiednich interfejsach, które są implementowane przez specjalne obiekty (pośredników lub namiastki)
- Kiedy dochodzi do wywołania metody (za pośrednictwem interfejsu zdalnego lub lokalnego) namiastka przekazuje żądanie do kontenera EJB, który deleguje je do właściwego egzemplarza komponentu

Dostęp do komponentów z poziomu aplikacji umieszczonej poza kontenerem

- W celu uzyskania dostępu do komponentu EJB, aplikacja klienta musi skorzystać ze specjalnego interfejsu JNDI
- Wspomniany interfejs udostępnia niezależne od implementacji API, oferujące dostęp do usług nazewniczych i katalogowych
- Za implementację JNDI odpowiada dostawca kontenera, który udostępnia klientom usługę katalogową
- Ponieważ kod służący do uzyskiwania kontekstu jest zależny od dostawcy kontenera, jego użycie wymaga określenia specjalnych parametrów – charakterystycznych dla producenta i dostarczanych wraz z dokumentacją kontenera

Dostęp do komponentów z poziomu aplikacji umieszczonej poza kontenerem

- Uzyskanie kontekstu i nawiązanie połączenia może zostać zrealizowane w następujący sposób:

```
Context ctx = new javax.naming.InitialContext(własciwosci);
```

- Kolejnym krokiem jest wykorzystanie kontekstu do odnalezienia zdalnego interfejsu komponentu

```
Object ref = ctx.lookup(nazwa_JNDI);
```

Dostęp do komponentów z poziomu aplikacji umieszczonej poza kontenerem

- Ostatecznie przeprowadzane jest rzutowanie otrzymanej referencji do odpowiedniego typu:

```
MyBeanRemote mb = (MyBeanRemote)  
PortableRemoteObject.narrow(ref, MyBeanRemote.class)
```

lub od wersji EJB 3.0:

```
MyBeanRemote mb = (MyBeanRemote) ref;
```

Dostęp do komponentów z poziomu aplikacji umieszczonej poza kontenerem

```
public class Client1 {  
    public static void main(String[] args) {  
        try {  
            Context jndiContext = new InitialContext();  
            Object ref = jndiContext.lookup("RecepcjaBean/remote");  
            RecepcjaRemote recepcja = (RecepcjaRemote)  
                PortableRemoteObject.narrow(ref, RecepcjaRemote.class);  
  
            recepcja.jakasMetoda();  
  
        } catch (NamingException ne) {  
            ne.printStackTrace();  
        }  
    }  
}
```

Parametry niezbędne do uzyskania kontekstu dla serwera JBoss

- Przykład pliku *jndi.properties*:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory  
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces  
java.naming.provider.url=localhost
```

- Przykład parametrów zdefiniowanych za pomocą obiektu typu *Properties*:

```
Properties p = new Properties();  
p.put(Context.INITIAL_CONTEXT_FACTORY,  
        "org.jnp.interfaces.NamingContextFactory");  
p.put(Context.URL_PKG_PREFIXES,  
        "org.jboss.naming:org.jnp.interfaces");  
p.put(Context.PROVIDER_URL, "localhost");  
Context jndiContext = new InitialContext(p);
```

Parametry niezbędne do uzyskania kontekstu dla serwera GlassFish

- Przykład pliku *jndi.properties*:

```
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
java.naming.factory.url.pkgs=com.sun.enterprise.naming
java.naming.factory.state=com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl
#optional. Defaults to localhost. Only needed if web server is running
#on a different host than appserver
org.omg.CORBA.ORBInitialHost=localhost
#optional. Defaults to 3700. Only needed if target orb port is not 3700.
org.omg.CORBA.ORBInitialPort=9715
```

Zarządzanie zasobami

- Systemy korporacyjne, obsługujące dużą liczbę użytkowników mogą wymagać jednoczesnego istnienia ogromnej liczby obiektów niezbędnych do realizacji określonych zadań biznesowych
- W celu zapewnienia odpowiedniej wydajności stosowane są dwa mechanizmy, które pozwalają zarządzać egzemplarzami komponentów w czasie ich działania:
 - mechanizm puli egzemplarzy
 - mechanizm aktywacji/pasywacji

Zarządzanie pulą egzemplarzy

- **Pula egzemplarzy** ogranicza liczbę komponentów wymaganych do obsługi żądań otrzymywanych od klientów
- Aplikacje klienckie współpracują z komponentami sesyjnymi za pośrednictwem interfejsów zdalnych lub lokalnych, implementowanych przez odpowiednie obiekty EJB – w związku z tym nie mają bezpośredniego dostępu do właściwych egzemplarzy klas komponentów sesyjnych
- Podobna sytuacja występuje w przypadku komponentów sterowanych komunikatami
- Dzięki temu, że aplikacje klienckie nie uzyskują bezpośredniego dostępu do komponentów, możliwe jest efektywne zarządzanie pulą egzemplarzy przez kontener

Pula egzemplarzy bezstanowego komponentu sesyjnego

- Bezstanowe komponenty sesyjne **nie utrzymują stanu konwersacji pomiędzy kolejnymi wywołaniami metod**. Dzięki temu możliwa jest obsługa nawet kilkuset aplikacji klienckich za pomocą kilku komponentów EJB tego typu
- Egzemplarz, który kończy obsługę żądania staje się natychmiast dostępny dla dowolnego, innego obiektu EJB, który go potrzebuje
- W efekcie możliwe jest zmniejszenie zużycia zasobów i zwiększenie wydajności

Pula egzemplarzy komponentów sterowanych komunikatami

- Dla każdego typu MDB tworzona jest osobna pula egzemplarzy odpowiedzialnych za obsługę komunikatów przychodzących
- Kiedy klient wysyła komunikat, ten trafia do kontenera EJB zawierającego komponenty, które zarejestrowały się jako odbiorcy
- Kontener wybiera z odpowiedniej puli jeden egzemplarz i zleca mu obsługę wiadomości
- Kiedy przetwarzanie zostanie zakończone komponent jest zwracany z powrotem do puli

Mechanizm aktywacji/pasywacji

- Ponieważ stanowe komponenty sesyjne **utrzymują swój stan pomiędzy kolejnymi wywołaniami metod**, nie ma możliwości stosowania mechanizmu puli egzemplarzy
- Zamiast tego wykorzystuje się **mechanizm aktywacji/pasywacji**
- Kiedy zachodzi potrzeba zwolnienia zasobów kontener usuwa komponenty stanowe z pamięci, a stan konwersacji z klientem serializowany jest w pamięci pomocniczej, np. na dysku (*pasywacja*)
- Podczas kolejnego żądania kontener tworzy egzemplarz stanowego komponentu sesyjnego i przywraca serializowany stan konwersacji (*aktywacja*)

Usługi podstawowe

- Istnieje osiem podstawowych usług implementowanych przez wszystkie kompletne platformy uruchomieniowe EJB.
Są to:
 - współbieżność
 - przetwarzanie transakcyjne
 - utrwalanie danych
 - obsługa obiektów rozproszonych
 - asynchroniczne przesyłanie komunikatów
 - bezpieczeństwo
 - licznik czasowy
 - nazewnictwo

Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager**
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

3

Utrwalanie i usługa Entity Manager

- wprowadzenie do Java Persistence API
- kontekst utrwalania
- usługa Entity Manager

Utrwalanie danych

- W momencie powstania EJB w wersji 3.0 warstwa utrwalania danych została przeniesiona do odrębnej specyfikacji nazwanej *Java Persistence API (JPA)*
- Wspomniana specyfikacja:
 - stanowi łatwą w użyciu abstrakcję, rozciągającą się ponad interfejsem JDBC, dzięki czemu programiści mają możliwość izolowania własnego kodu od stosowanej bazy danych i rozwiązań charakterystycznych dla określonych producentów
 - umożliwia automatyczne odwzorowywanie obiektów Java w relacyjnej bazie danych oraz udostępnia specjalny język zapytań JPQL przeznaczony do współpracy z obiektami

Entity Manager

- Centralna usługa pozwalająca na przeprowadzanie działań związanych z utrwalaniem danych i odwzorowaniami obiektowo-relacyjnymi
- Ponieważ encje mają postać obiektów typu POJO nie są utrwalane, aż do momentu jawnego użycia usługi *EntityManager*, która oferuje metody pozwalające między innymi na:
 - konstruowanie zapytań
 - odnajdywanie obiektów
 - synchronizację obiektów
 - dodawanie/usuwanie obiektów

Encje zarządzane i niezarządzane

- **Encje zarządzane (*managed*)** – czyli połączone z menedżerem encji – są kontrolowane pod względem wprowadzanych zmian w celu ich ewentualnej synchronizacji z bazą danych
- **Encje niezarządzane (*unmanaged*)** – odłączone od menedżera encji – nie są śledzone (modyfikacje nie będą odzwierciedlane w bazie)
 - w przypadku encji odłączonych istnieje możliwość ich serializacji i przesłania za pośrednictwem sieci do klienta, który po modyfikacji może je odesłać z powrotem i zażądać scalenia

Kontekst utrwalania

- Na **kontekst utrwalania** (*persistence context*) składa się zbiór zarządzanych egzemplarzy komponentów encyjnych
- Zmiany dotyczące komponentów encyjnych, w ramach danego kontekstu, są na bieżąco śledzone przez usługę *EntityManager* i odwzorowywane w bazie danych
- Po zamknięciu kontekstu utrwalania wszystkie należące do niego egzemplarze encji zostają odłączone i stają się nie zarządzane – kolejne zmiany nie będą miały wpływu na dane przechowywane w bazie
- Rozróżnia się dwa rodzaje kontekstów utrwalania:
 - o zasięgu transakcyjnym
 - o zasięgu rozszerzonym

Uzyskiwanie kontekstu utrwalania

- Egzemplarze usługi *EntityManager* mogą zostać wstrzyknięte bezpośrednio do komponentu EJB za pomocą adnotacji `@javax.persistence.PersistenceContext`
- Atrybuty adnotacji:

<code>unitName</code>	określa nazwę kontekstu utrwalania, obowiązkowy
<code>type</code>	pozwala ustawić zasięg kontekstu (domyślnie transakcyjny), opcjonalny

Kontekst utrwalania o zasięgu transakcyjnym

- Czas życia kontekstu obejmuje wyłącznie czas życia pojedynczej transakcji, po zakończeniu której kontekst jest niszczony, a wszystkie encje wchodzące w jego skład zostają odłączone
- Standardowo każdy kontekst utrwalania ma zasięg transakcyjny
- Może zostać ustawiony w następujący sposób:

```
@PersistenceContext (unitName="hotel",
                     type=PersistenceContextType.TRANSACTION)
```

Kontekst utrwalania o zasięgu rozszerzonym

- Czas życia kontekstu nie jest ograniczony do czasu trwania pojedynczej transakcji
- Po jej zakończeniu encje nadal zachowują status encji zarządzanych
- Zmiany, które zostaną na nich wykonane, zostaną odzwierciedlone w bazie danych
- Może zostać ustawiony w następujący sposób:

```
@PersistenceContext (unitName="hotel",
    type=PersistenceContextType.EXTENDED)
```

Deskryptor wdrożenia *persistence.xml*

- Specjalny plik o składni XML, konfigurujący właściwości definiowanych jednostek utrwalania (zbiorów klas odwzorowywanych w bazie danych)
- Przykładowy plik *persistence.xml*:

```
<persistence>
    <persistence-unit name="Clients" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>java:/ MySqlDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto"
                      value="create-drop" />
        </properties>
    </persistence-unit>
</persistence>
```

Deskryptor wdrożenia *persistence.xml*

- Pojedynczy plik *persistence.xml* może definiować dowolną liczbę jednostek utrwalania i być umieszczony w jednej z następujących lokalizacji:
 - w katalogu *META-INF* pliku *ejb-jar.jar*
 - w katalogu *WEB-INF/lib* w formie **.jar*
 - w katalogu głównym pliku **.ear* w formie **.jar*
 - w katalogu *lib* pliku **.ear* w formie **.jar*
- Klasy wchodzące w skład danej jednostki utrwalania mogą być:
 - odnalezione automatycznie przez usługę *EntityManager*
 - jawnie określone przez programistę za pomocą elementów XML:
 - <**jar-file**> względna_ścieżka_do_pliku_jar </**jar-file**>
 - <**class**> w_pełni_kwalifikowana_nazwa_klasy </**class**>
 - <**exclude-unlisted-classes**>/> – powoduje wyłączenie przeszukiwania jar'a pod kątem klas komponentów encyjnych

Ważniejsze metody Entity Manager

```
public void persist(Object obj)
public <T> T find(Class<T> entityClass, Object primaryKey)
public <T> T getReference(Class<T> entityClass, Object primaryKey)
```

- Metoda *persist*:
 - pozwala na utrwalenie wcześniej utworzonych, nie istniejących w bazie encji
- Metody *find* oraz *getReference*:
 - umożliwiają odnajdywanie encji według unikatowych kluczy głównych
 - jeśli nie uda się odnaleźć szukanej referencji:
 - *find* zwraca wartość *null*
 - *getReference* wyrzuca wyjątek typu *EntityNotFoundException*

Ważniejsze metody Entity Manager

```
public <T> T merge(T entity)
public void remove(Object entity)
```

- Metoda *merge*:

- umożliwia scalenie zmian stanu egzemplarza encji odłączonej z pamięcią trwałą
- jeżeli menedżer encji nie zarządza egzemplarzem klasy danego typu z taką samą wartością klucza, metoda skonstruuje i zwróci odrębną kopię obiektu, która będzie zarządzana przez menedżer encji
- w sytuacji, kiedy menedżer encji już zarządza egzemplarzem klasy o danym identyfikatorze jej zawartość zostanie zaktualizowana i zwrócona

- Metoda *remove*:

- umożliwia usunięcie pojedynczej encji z bazy danych

Ważniejsze metody Entity Manager

```
public void refresh(Object entity)
public boolean contains(Object entity)
```

- Metoda *refresh*:
 - pozwala na odświeżenie stanu aktualnie zarządzanego komponentu encyjnego na podstawie danych przechowywanych w pamięci trwałej
- Metoda *contains*:
 - zwraca wartość logiczną informującą, czy dany egzemplarz komponentu encyjnego jest już zarządzany przez usługę menedżera encji

Ważniejsze metody Entity Manager

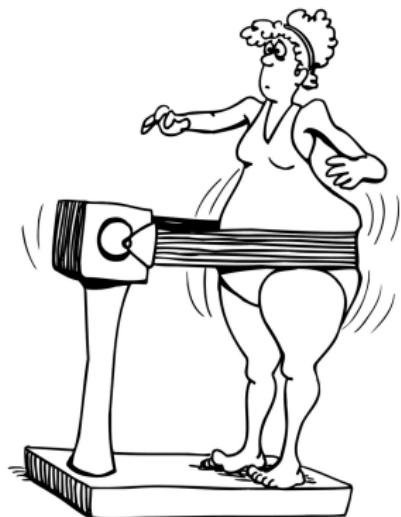
```
public void clear()  
public void flush()
```

- Metody *clear* oraz *flush*:

- clear* powoduje odłączenie od kontekstu utrwalania wszystkich zarządzanych egzemplarzy komponentów encyjnych
- nie utrwalone zmiany dokonane na encjach zostaną bezpowrotnie utracone
- aby temu zapobiec trzeba wcześniej wywołać metodę *flush*

ćwiczenia

- Ćwiczenie 3.1:
 - A. budowa pierwszych komponentów
 - B. wykorzystanie rozszerzonego kontekstu utrwalania



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne**
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

4

Odwzorowania obiektowo-relacyjne

- definiowanie reguł odwzorowań
- techniki generowania kluczy głównych
- odwzorowywanie właściwości encji
- obiekty osadzone

Odwzorowywanie obiektów

- Głównym zadaniem egzemplarzy komponentów encyjnych jest reprezentacja informacji przechowywanych w wierszach relacyjnej bazy danych
- Kiedy zostaje utworzona nowa encja, a następnie utrwalona za pomocą usługi menedżera, do bazy wstawiany jest nowy rekord (powiązany z encją za pomocą klucza)
- W sytuacji, kiedy stan encji się zmienia, zmiany są propagowane do pamięci stałej (tylko dla encji zarządzanych)
- Encje – to obiekty typu POJO, dlatego proces ich tworzenia i modyfikacji jest identyczny, jak w przypadku najwyklejszych obiektów Javy

Odwzorowywanie obiektów

- Java Persistence narzuca, aby każda klasa encji posiadała bezargumentowy konstruktor oraz przynajmniej dwie, poniższe adnotacje:
 - **@javax.persistence.Entity**
 - informuje o tym, że dana klasa jest encją i powinna być odwzorowywana w bazie danych
 - może posiadać jeden atrybut **name** określający nazwę wykorzystywaną w zapytaniach JPQL
 - jest stosowana na poziomie klasy
 - **@javax.persistence.Id**
 - wskazuje właściwość, która powinna spełniać funkcję klucza głównego
 - może być stosowana na poziomie pola lub metody

Odwzorowywanie obiektów

- Domyślnie usługa utrwalania przyjmuje, że wszystkie właściwości encji powinny być umieszczone w kolumnach tabeli (o nazwie zgodnej z nazwą klasy) o odpowiednio takiej samej nazwie i typie, jak same właściwości
- Powyższe ustawienia można nadpisać za pomocą adnotacji:
 - **@javax.persistence.Table**
 - z parametrem **name** wymusza docelową tabelę, w której mają być odwrzorowywane egzemplarze danej klasy
 - dodatkowo pozwala określić katalog i schemat relacyjny poprzez atrybuty **catalog** i **schema**
 - jest stosowana na poziomie klasy
 - **@javax.persistence.Column**
 - posiada parametr **name** określający nazwę kolumny docelowej dla danej własności, a także inne, jak np. opisujące typ (**columnDefinition**), długość (**length**), możliwość występowania wartości pustych (**nullable**), itp.
 - jest stosowana na poziomie pola lub metody

Przykład komponentu encyjnego z odpowiednimi adnotacjami

```
@Entity
@Table(name="KLIENCI")
public class Klient implements Serializable {
    @Id
    @Column(name="KLIENT_ID")
    private long id;

    @Column(name="IMIE", nullable=false, length=8)
    private String imie;

    @Column(name="NAZWISKO", nullable=false, length=12)
    private String nazwisko;

    ...
}
```

Odwzorowywanie obiektów

- W sytuacji, kiedy programista nie chce stosować mechanizmu adnotacji, może stworzyć specjalny plik XML definiujący wymagane reguły odwzorowywania
- Deskryptory mapowania mogą być umieszczone w katalogu *META-INF* i nazywać się *orm.xml* lub w dowolnym innym katalogu, jeśli w deskryptorze wdrożenia *persistence.xml* zadeklarowano element <**mapping-file**> wskazujący jego położenie

Przykład deskryptora mapowania

```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <table name="KLIENCI"/>
        <attributes>
            <id name="id">
                <column name="KLIENT_ID" nullable="false"/>
            </id>
            <basic name="imie">
                <column name="IMIE" nullable="false" length="8"/>
            </basic>
            <basic name="nazwisko">
                <column name="NAZWISKO" nullable="false" length="12"/>
            </basic>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Klucze główne

- Każdy komponent encyjny musi posiadać dokładnie jeden unikatowy klucz główny, który stanowi jego **identyfikator**
- Może on powstać na bazie jednej lub wielu właściwości i mieć postać jednego z typów prostych, *String* lub specjalnej klasy klucza głównego
- W przypadku kluczy bazujących na pojedynczej właściwości wystarczy użyć adnotacji `@javax.persistence.Id` przy odpowiednim polu lub metodzie

Klucze główne

- Jeśli klucz ma być generowany w sposób automatyczny należy dodatkowo użyć adnotacji
`@javax.persistence.GeneratedValue`
- Domyślnie decyzję o sposobie generowania kluczy pozostawiamy usłudze utrwalania, co odpowiada ustawieniu atrybutu
`strategy=GenerationType.AUTO`
- Z kolei `strategy=GenerationType.IDENTITY` wymusza stosowanie specjalnego typu kolumny stworzonego z myślą o kluczach głównych

Klucze główne – generatory tabel

- JPA pozwala na generowanie kluczy głównych przy użyciu tzw. **generatora tabel**
- W tym przypadku parametr
strategy=GenerationType.TABLE
- Algorytm wykorzystuje zdefiniowaną tabelę, na podstawie której zwracane będą kolejne wartości klucza
- Tabela powinna zawierać dwie kolumny, np.:

PRIMARY_KEY_COLUMN	VARCHAR	not null	zawiera klucze
VALUE_COLUMN	long	not null	zawiera wartość licznika

- W celu automatycznego wygenerowania tabeli można użyć adnotacji **@javax.persistence.TableGenerator**

Klucze główne – generatory kodu

```
@Entity
public class Klient implements Serializable {
    @Id
    @TableGenerator(name="ID_GEN", table="GEN_TABLE",
                    pkColumnName="pkColumn",
                    valueColumnName="valColumn",
                    pkColumnValue="id", allocationSize=30)
    @GeneratedValue(strategy=GenerationType.TABLE,
                   generator="ID_GEN")
    private long id;

    private String imie;
    ...
}
```

Klucze główne – generatory kodu

```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <table-generator name="ID_GEN" table="GEN_TABLE"
            pk-column-name="pkColumn"
            value-column-name="valColumn"
            pk-column-value="id" allocation-size="30"/>
        <table name="KLIENCI"/>
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE" generator="ID_GEN"/>
            </id>
            <basic name="imie">
                <column name="IMIE" nullable="false" length="8"/>
            </basic>
            <basic name="nazwisko">
                <column name="NAZWISKO" nullable="false" length="12"/>
            </basic>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Klucze główne – generatorы сеансов

- Specyfikacja daje możliwość wykorzystania generatorów sekwencji oferowanych przez niektóre relacyjne systemy zarządzania bazami danych
- W tym celu należy ustawić atrybut
strategy=GenerationType.SEQUENCE
i stworzyć odpowiedni generator za pomocą adnotacji
@javax.persistence.SequenceGenerator

```
@Entity
public class Klient implements Serializable {

    @Id
    @SequenceGenerator(name="ID_SEQ", sequenceName="KLIENT_SEQ")
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
                   generator="ID_SEQ")
    private long id;
    ...
}
```

Klucze główne – generatory sekwencji

```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <sequence-generator name="ID_SEQ" sequence-name="KLIENT_SEQ"
            initial-value="0" allocation-size="50"/>
        <table name="KLIENCI"/>
        <attributes>
            <id name="id">
                <generated-value strategy="SEQUENCE" generator="ID_SEQ"/>
            </id>
            <basic name="imie">
                <column name="IMIE" nullable="false" length="8"/>
            </basic>
            <basic name="nazwisko">
                <column name="NAZWISKO" nullable="false" length="12"/>
            </basic>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Klucze główne złożone

- Specyfikacja JPA dostarcza możliwość stosowania kluczy złożonych, a więc obejmujących więcej niż jedną właściwość trwałą
- Klucze złożone nie mogą być generowane w sposób automatyczny
- W celu określenia klucza złożonego należy stworzyć specjalną klasę zawierającą wszystkie potrzebne właściwości
- Wspomniana klasa musi spełniać następujące założenia:
 - musi implementować interfejs *Serializable*
 - musi posiadać bezargumentowy konstruktor
 - musi nadpisywać metody *equals* i *hashCode*
- Klasa komponentu encyjnego musi wykorzystywać adnotację **@javax.persistence.IdClass** w celu wskazania klasy klucza oraz zawierać dokładnie te same właściwości wyróżnione dodatkowo adnotacją **@javax.persistence.Id**

Klucze główne złożone – przykład klasy klucza

```
public class KluczKlient implements Serializable {  
    private String nazwisko;  
    private String telefon;  
  
    public KluczKlient() {}  
  
    public KluczKlient(String nazwisko, String telefon) {  
        this.nazwisko = nazwisko;  
        this.telefon = telefon;  
    }  
  
    public String getNazwisko() {  
        return nazwisko;  
    }  
  
    public String getTelefon() {  
        return telefon;  
    }  
    ...
```

Klucze główne złożone – przykład klasy klucza

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result  
        + ((nazwisko == null) ? 0 : nazwisko.hashCode());  
    result = prime * result + ((telefon == null) ? 0 :  
        telefon.hashCode());  
    return result;  
}  
  
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
    if (getClass() != obj.getClass()) return false;  
    final KluczKlient other = (KLuczKlient)obj;  
    if (nazwisko == null) {  
        if (other.nazwisko != null) return false;  
    } else if (!nazwisko.equals(other.nazwisko))  
        return false;  
    if (telefon == null) {  
        if (other.telefon != null) return false;  
    } else if (!telefon.equals(other.telefon))  
        return false;  
    return true;  
}
```

Klucze główne złożone – przykład użycia klasy klucza

```
@Entity  
@IdClass(KluczKlient.class)  
public class Klient implements Serializable {  
  
    private String imie;  
  
    @Id  
    private String nazwisko;  
  
    private String ulica;  
  
    private String miasto;  
  
    @Id  
    private String telefon;  
    ...  
}
```

Klucze główne złożone

```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <id-class>pl.altkom.KluczKlient</id-class>
        <table name="KLIENCI"/>
        <attributes>
            <id name="nazwisko"/>
            <id name="telefon"/>
            <basic name="imie">
                <column name="IMIE" nullable="false" length="8"/>
            </basic>
            <basic name="nazwisko">
                <column name="NAZWISKO" nullable="false" length="12"/>
            </basic>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Klucze główne złożone

- Innym sposobem na uzyskiwanie kluczy złożonych jest osadzenie klas definiujących wspomniane klucze wewnątrz komponentów encyjnych
- Realizacja tego zadania jest możliwa przy użyciu dwóch adnotacji:
 - `@javax.persistence.EmbeddedId`
 - `@javax.persistence.Embeddable`

Klucze główne złożone – przykład klasy klucza

```
@Embeddable
public class KluczKlient implements Serializable {
    private String nazwisko;
    private String telefon;

    public KluczKlient() {}

    public KluczKlient(String nazwisko, String telefon) {
        this.nazwisko = nazwisko;
        this.telefon = telefon;
    }

    public String getNazwisko() {
        return nazwisko;
    }

    public String getTelefon() {
        return telefon;
    }
}
```

Klucze główne złożone – przykład klasy klucza

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result  
        + ((nazwisko == null) ? 0 : nazwisko.hashCode());  
    result = prime * result + ((telefon == null) ? 0 :  
        telefon.hashCode());  
    return result;  
}  
  
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
    if (getClass() != obj.getClass()) return false;  
    final KluczKlient other = (KLuczKlient)obj;  
    if (nazwisko == null) {  
        if (other.nazwisko != null) return false;  
    } else if (!nazwisko.equals(other.nazwisko))  
        return false;  
    if (telefon == null) {  
        if (other.telefon != null) return false;  
    } else if (!telefon.equals(other.telefon))  
        return false;  
    return true;  
}
```

Klucze główne złożone – przykłady użycia klasy klucza

```
@Entity
public class Klient implements Serializable {
    private KluczKlient kk;
    private String imie;
    private String ulica;
    private String miasto;

    @EmbeddedId
    public KluczKlient getKk() {
        return kk;
    }
    public void setKk(KluczKlient kk) {
        this.kk = kk;
    }
    ...
}
```

Klucze główne złożone – przykłady użycia klasy klucza

```
@Entity
public class Klient implements Serializable {
    private KluczKlient kk;
    private long id;
    private String imie;
    private String ulica;
    private String miasto;

    @EmbeddedId
    @AttributeOverrides({
        @AttributeOverride(name="nazwisko",
                           column=@Column(name="KL_NAZWISKO")),
        @AttributeOverride(name="telefon",
                           column=@Column(name="KL_TELEFON"))
    })
    public KluczKlient getKk() {
        return kk;
    }
    public void setKk(KluczKlient kk) {
        this.kk = kk;
    }
    ...
}
```

Klucze główne złożone

```
<entity-mappings>
    <embeddable class="pl.altkom.KluczKlient"
        access-type="PROPERTY">
        <attributes>
            <basic name="nazwisko">
                <column name="NAZWISKO"/>
            </basic>
            <basic name="TELEFON">
                <column name="TELEFON"/>
            </basic>
        </attributes>
    </embeddable>
    <entity class="pl.altkom.Klient" access="PROPERTY">
        <table name="KLIENCI"/>
        <attributes>
            <embedded-id name="nazwisko">
                <attribute-override name="nazwisko">
                    <column name="KL_NAZWISKO"/>
                </attribute-override>
            </embedded-id>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Odwzorowywanie właściwości

• **@javax.persistence.Transient**

- służy do oznaczania właściwości encji, które nie powinny być utrwalane
- odpowiednik XML:
`<transient name="nazwa" />`

• **@javax.persistence.Basic**

- określa domyślny rodzaj odwzorowania właściwości reprezentujących dane typów prostych i klas opakowujących Javy
- z reguły nie musimy stosować tej adnotacji, ponieważ usługa utrwalania sama rozpozna typ danych
- zawiera dwa atrybuty:
 - **optional** – określający, czy dopuszczalna jest wartość pusta
 - **fetch** – definiujący sposób ładowania danych z bazy

- odpowiednik XML:

```
<basic name="nazwa" fetch="tryb ładowania"  
      optional="true/false" />
```

Odwzorowywanie właściwości

• @javax.persistence.Temporal

- pozwala na przekazanie usłudze utrwalania dodatkowych informacji dotyczących danych typu *java.util.Date* i *java.util.Calendar*, co wymusza stosowanie kolumn zawierających typ daty, godziny lub znaczników czasowych
- odpowiednik XML:
`<temporal>TIME/DATE/TIMESTAMP</temporal>`

• @javax.persistence.Lob

- wymusza odwrzorowywanie w formie danych *Blob* (jeśli typem jest *byte[]*, *Byte[]*, *Serializable*) lub *Clob* (jeśli typem jest *char[]*, *Character[]*, *String*)
- odpowiednik XML:
`<lob/>`

Odwzorowywanie właściwości

• **@javax.persistence.Enumerated**

- pozwala na odwrzorowywanie typów wyliczeniowych w formie wartości tekstowej lub liczbowej, określonej za pomocą parametru *EnumType.String* lub *EnumType.Ordinal*
- odpowiednik XML:
`<enumerated>ORDINAL / STRING</enumerated>`

• **@javax.persistence.SecondaryTable(s)**

- pozwala na reprezentowanie danych encji w wielu tabelach bazy

Odwzorowywanie właściwości w wielu tabelach

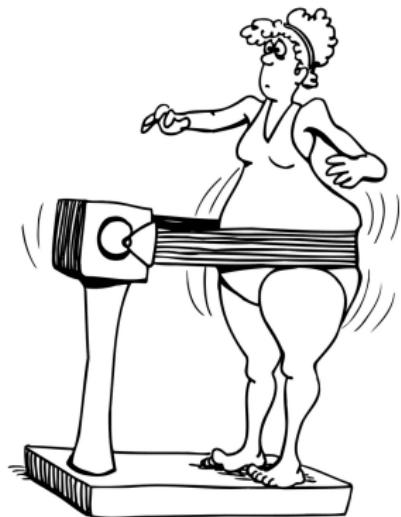
```
@Entity
@Table(name="KLIENCI")
@SecondaryTable(
    name="ADRESY",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="ADRES_ID"))
)
public class Klient implements Serializable {
    @Id
    private long id;
    @Column(name="IMIE")
    private String imie;
    @Column(name="NAZWISKO")
    private String nazwisko;
    @Column(name="ULICA", table="ADRESY")
    private String ulica;
    ...
}
```

Obiekty osadzone

- W ramach komponentów encyjnych istnieje możliwość osadzania innych obcych obiektów (nie encyjnych), które będą odwzorowywane w tabeli bazy danych
- Tego rodzaju obiekty traktowane są jako własność klasy
- Klasa, na podstawie której tworzony jest obiekt, powinna być oznaczona przez `@javax.persistence.Embeddable`
- Osadzenie odbywa się za pomocą adnotacji
`@javax.persistence.Embedded`
- Istnieje możliwość stosowania adnotacji
`@javax.persistence.AttributeOverrides`, która nadpisuje odwzorowania użyte w klasie osadzanej

Ćwiczenia

- Ćwiczenie 4.1:
*wykorzystanie podstawowych mechanizmów odwzorowań;
definiowanie klucza złożonego:
A. z użyciem @IdClass
B. z użyciem @EmbeddedId*



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami**
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

5

Zależności zachodzące między encjami

- typy relacji łączących komponenty encyjne
- sposób implementacji poszczególnych relacji
- propagacja kaskadowa
- realizacja dziedziczenia encji

Typy relacji

- Bardzo często podczas modelowania rzeczywistych pojęć biznesowych zachodzi potrzeba łączenia komponentów encyjnych w ramach najróżniejszych relacji
- Wyróżnia się następujące typy związków:
 - jeden-do-jednego (*one-to-one*)
 - jeden-do-wielu (*one-to-many*)
 - wiele-do-jednego (*many-to-one*)
 - wiele-do-wielu (*many-to-many*)
- Każda z powyższych relacji może mieć charakter jednokierunkowy lub dwukierunkowy (kierunek określany jest na podstawie wzajemnych odwołań istniejących w kodzie)

Jednokierunkowa relacja jeden-do-jednego

- Występuje, kiedy jeden z komponentów encyjnych posiada właściwość umożliwiającą zwracanie lub ustawianie drugiego komponentu tworzącego relację, np. *Klient* → *Adres*
- Obie encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę właściwą:
 - `@javax.persistence.OneToOne` oraz
 - `@javax.persistence.JoinColumn(s)`
- W przypadku, kiedy obie tabele wykorzystują takie same klucze główne, można zastosować adnotację
`@javax.persistence.PrimaryKeyJoinColumn(s)`
(zamiast `@javax.persistence.JoinColumn(s)`)
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania wystarczy użyć samej adnotacji `@OneToOne`

Jednokierunkowa relacja jeden-do-jednego

```
@Entity  
public class Klient implements Serializable {  
    @Id @GeneratedValue  
    private long id;  
    private String imie;  
    private String nazwisko;  
    @OneToOne  
    @JoinColumn(name="adres_id")  
    private Adres adres;  
    // + metody dostepowe dla powyzszych pol...  
}
```

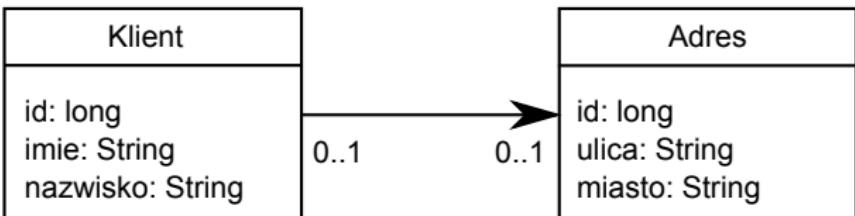
```
@Entity  
public class Adres implements Serializable {  
    @Id @GeneratedValue  
    private long id;  
    private String ulica;  
    private String miasto;  
    // + metody dostepowe dla powyzszych pol...  
}
```

Jednokierunkowa relacja jeden-do-jednego

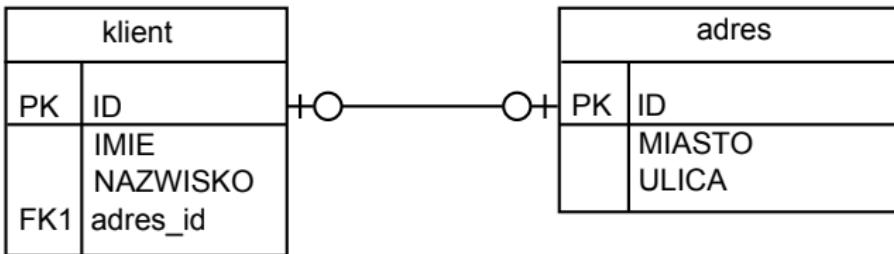
```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-one name="adres" target-entity="pl.altkom.Adres">
                <join-column name="adres_id"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

Jednokierunkowa relacja jeden-do-jednego

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Dwukierunkowa relacja jeden-do-jednego

- Występuje, kiedy jeden z komponentów encyjnych posiada właściwość umożliwiającą zwracanie lub ustawianie drugiego komponentu tworzącego relację i odwrotnie, np. *Klient* ↔ *KontoBankowe*
- Obie encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę:
 - właścielską:
 - `@javax.persistence.OneToOne` oraz
 - `@javax.persistence.JoinColumn(s)`
 - przeciwną:
 - `@OneToOne (mappedBy="konto")`

Dwukierunkowa relacja jeden-do-jednego

- W przypadku, kiedy obie tabele wykorzystują takie same klucze główne można zastosować
`@javax.persistence.PrimaryKeyJoinColumn(s)`
(zamiast `@javax.persistence.JoinColumn(s)`)
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Klient* do `@OneToOne`

Dwukierunkowa relacja jeden-do-jednego

```
@Entity
public class Klient implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;
    @OneToOne
    @JoinColumn(name="konto_id")
    private KontoBankowe konto;
    // + metody dostepowe dla powyzszych pol...
}
```

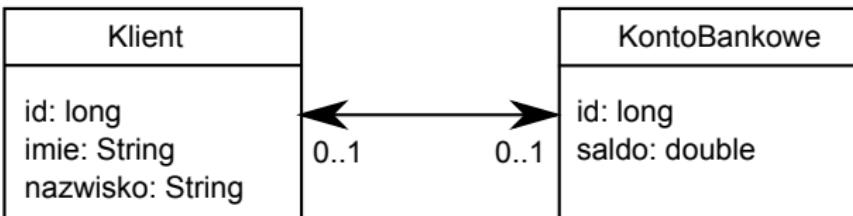
```
@Entity
public class KontoBankowe implements Serializable {
    @Id @GeneratedValue
    private long id;
    private double saldo;
    @OneToOne(mappedBy="konto")
    private Klient klient;
    // + metody dostepowe dla powyzszych pol...
}
```

Dwukierunkowa relacja jeden-do-jednego

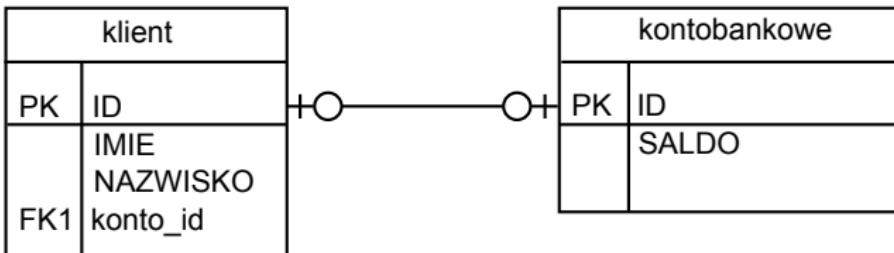
```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-one name="konto"
                        target-entity="pl.altkom.KontoBankowe">
                <join-column name="konto_id"/>
            </one-to-one>
        </attributes>
    </entity>
    <entity class="pl.altkom.KontoBankowe" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-one name="klient" target-entity="pl.altkom.Klient"
                        mapped-by="konto"/>
        </attributes>
    </entity>
</entity-mappings>
```

Dwukierunkowa relacja jeden-do-jednego

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Dwukierunkowa relacja jeden-do-jednego

- Podczas przeprowadzania modyfikacji encji znajdujących się w relacjach dwukierunkowych należy dokonać właściwego wiązania obu stron – aby aktualizacja była możliwa na poziomie bazy danych
- Dlatego **obowiązkowo** należy ustawić odpowiednie wartości po obu stronach relacji
 - np. jeśli klient będzie chciał zrezygnować z konta powinniśmy ustawić mu właściwość *konto* na *null* i usunąć z bazy odpowiednią encję typu *KontoBankowe*

Jednokierunkowa relacja jeden-do-wielu

- Występuje, kiedy jeden z komponentów encyjnych posiada właściwość w postaci struktury danych (pochodzącej z pakietu `java.util`) reprezentującą grupę referencji do obiektów innych encji, np. *Klient* → *Telefon*
- Encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę właścielską:
 - `@javax.persistence.OneToMany` oraz
 - `@javax.persistence.JoinColumn(s)`
- W przypadku, kiedy obie tabele wykorzystują takie same klucze główne można zastosować
`@javax.persistence.PrimaryKeyJoinColumn(s)`
(zamiast `@javax.persistence.JoinColumn(s)`)
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Klient* do `@OneToMany`

Jednokierunkowa relacja jeden-do-wielu

```
@Entity
public class Klient implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;
    @OneToMany
    @JoinColumn(name="klient_id")
    private List<Telefon> telefony = new ArrayList<Telefon>();
    // + metody dostepowe dla powyzszych pol...
}
```

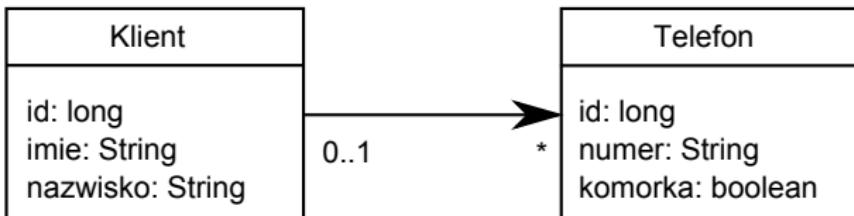
```
@Entity
public class Telefon implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String numer;
    private boolean komorka;
    // + metody dostepowe do powyzszych pol
}
```

Jednokierunkowa relacja jeden-do-wielu

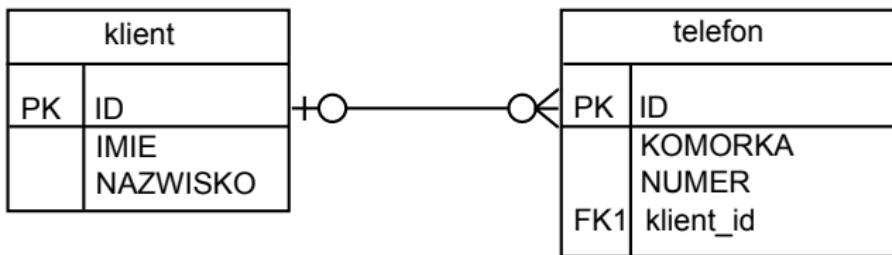
```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-many name="telefony"
                          targetEntity="pl.altkom.Telefon">
                <join-column name="klient_id"/>
            </one-to-many>
        </attributes>
    </entity>
</entity-mappings>
```

Jednokierunkowa relacja jeden-do-wielu

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Jednokierunkowa relacja jeden-do-wielu

- Rozwiązaniem alternatywnym jest zastąpienie adnotacji `@javax.persistence.JoinColumn` adnotacją `@javax.persistence.JoinTable` wraz z atrybutami:

<code>joinColumns</code>	definiuje odwzorowanie klucza obcego w klucz główny strony właścielskiej relacji
<code>inverseJoinColumns</code>	definiuje odwzorowanie klucza obcego na klucz główny strony przeciwej

Jednokierunkowa relacja jeden-do-wielu

```
@Entity
public class Klient implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;
    @OneToMany
    @JoinTable(name="KLIENT_TELEFON",
               joinColumns={@JoinColumn(name="Klient_id")}),
               inverseJoinColumns={@JoinColumn(name="Telefon_id")})
    private Collection<Telefon> telefony = new ArrayList<Telefon>();
    // + metody dostepowe dla powyzszych pol...
}
```

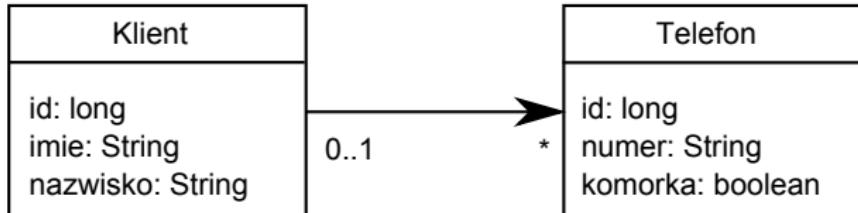
```
@Entity
public class Telefon implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String numer;
    private boolean komorka;
    // + metody dostepowe do powyzszych pol
}
```

Jednokierunkowa relacja jeden-do-wielu

```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-many name="telefony"
                          target-entity="pl.altkom.Telefon">
                <join-table name="KLIENT_TELEFON">
                    <join-column name="Klient_id"/>
                    <inverse-join-column name="Telefon_id"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>
</entity-mappings>
```

Jednokierunkowa relacja jeden-do-wielu

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Jednokierunkowa relacja wiele-do-jednego

- Występuje, kiedy wiele komponentów encyjnych posiada właściwość stanowiącą odwołanie do jednego komponentu encyjnego, np. *Klient* → *Doradca*
- Encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę właścielską:
 - `@javax.persistence.ManyToOne` oraz
 - `@javax.persistence.JoinColumn(s)`
- W przypadku, kiedy obie tabele wykorzystują takie same klucze główne można zastosować
`@javax.persistence.PrimaryKeyJoinColumn(s)`
(zamiast `@javax.persistence.JoinColumn(s)`)
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Klient* do `@ManyToOne`

Jednokierunkowa relacja wiele-do-jednego

```
@Entity  
public class Klient implements Serializable {  
    @Id @GeneratedValue  
    private long id;  
    private String imie;  
    private String nazwisko;  
    @ManyToOne  
    @JoinColumn(name="Doradca_id")  
    private Doradca doradca;  
    // + metody dostepowe dla powyzszych pol...  
}
```

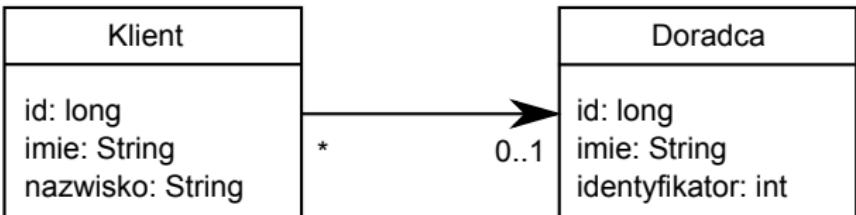
```
@Entity  
public class Doradca implements Serializable {  
    @Id @GeneratedValue  
    private long id;  
    private String imie;  
    private int identyfikator;  
    // + metody dostepowe do powyzszych pol  
}
```

Jednokierunkowa relacja wiele-do-jednego

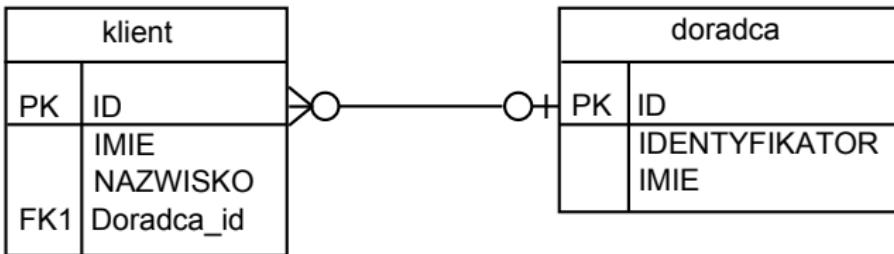
```
<entity-mappings>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <many-to-one name="doradca"
                target-entity="pl.altkom.Doradca">
                <join-column name="Doradca_id"/>
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

Jednokierunkowa relacja wiele-do-jednego

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Dwukierunkowa relacja jeden-do-wielu

- Z relacją tego typu mamy do czynienia, kiedy komponent encyjny posiada właściwość w postaci kolekcji referencji do innych komponentów, z których każdy utrzymuje referencję do obiektu agregującego, np. *Klient* ↔ *Bank*
- Jest **tożsama z dwukierunkową relacją wiele-do-jednego**
- Encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę
 - właścielską:
 - `@javax.persistence.ManyToOne` oraz
 - `@javax.persistence.JoinColumn(s)`
 - przeciwną:
 - `@javax.persistence.OneToMany (mappedBy="bank")`

Dwukierunkowa relacja jeden-do-wielu

- W przypadku, kiedy obie tabele wykorzystują takie same klucze główne można zastosować
`@javax.persistence.PrimaryKeyJoinColumn(s)`
(zamiast `@javax.persistence.JoinColumn(s)`)
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Klient* do `@ManyToOne`

Dwukierunkowa relacja jeden-do-wielu

```
@Entity
public class Klient implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;
    @ManyToOne
    @JoinColumn(name="Bank_id")
    private Bank bank;
    // + metody dostepowe dla powyzszych pol...
}
```

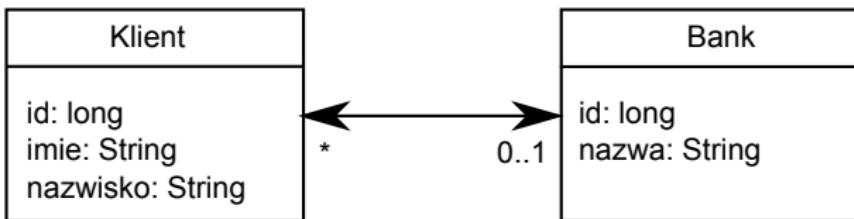
```
@Entity
public class Bank implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String nazwa;
    @OneToMany(mappedBy="bank")
    private Collection<Klient> klienci = new ArrayList<Klient>();
    // + metody dostepowe dla powyzszych pol...
}
```

Dwukierunkowa relacja jeden-do-wielu

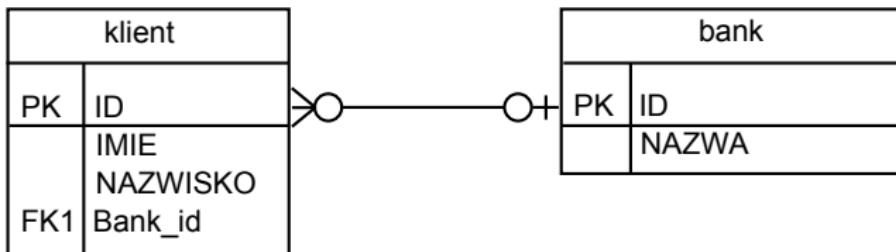
```
<entity-mappings>
    <entity class="pl.altkom.Bank" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <one-to-many name="klienci"
                target-entity="pl.altkom.Klient"
                mapped-by="bank"/>
        </attributes>
    </entity>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <many-to-one name="bank"
                target-entity="pl.altkom.Bank"
                optional="true">
                <join-column name="Bank_id"/>
            </many-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

Dwukierunkowa relacja jeden-do-wielu

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Dwukierunkowa relacja wiele-do-wielu

- Z relacją tego typu mamy do czynienia, kiedy wiele komponentów encyjnych posiada właściwość reprezentowaną przez kolekcję referencji do innych komponentów, z których każdy zawiera kolekcje referencji zwrotnych do komponentów agregujących, np. *Rezerwacja* ↔ *Klient*
- Encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę
 - właścielską:
 - `@javax.persistence.ManyToMany` oraz
 - `@javax.persistence.JoinTable`
 - przeciwną:
 - `@ManyToMany (mappedBy="klienci")`
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Rezerwacja* do `@ManyToMany`

Dwukierunkowa relacja wiele-do-wielu

```
@Entity
public class Rezerwacja implements Serializable {
    @Id @GeneratedValue
    private long id;
    @Temporal(TemporalType.DATE)
    private Date data;
    @ManyToMany
    @JoinTable(name="REZERWACJA_Klient",
               joinColumns={@JoinColumn(name="Rezerwacja_id")},
               inverseJoinColumns={@JoinColumn(name="Klient_id")})
    private Set<Klient> klienci = new HashSet<Klient>();
    // + metody dostepowe do powyzszych pol
}
```

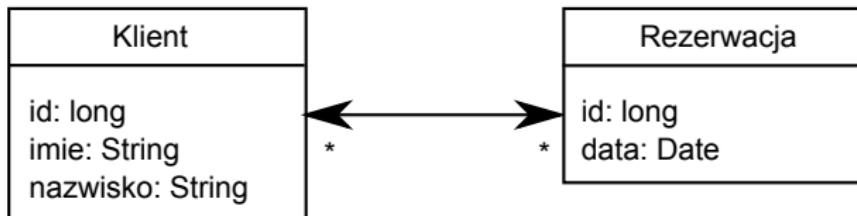
```
@Entity
public class Klient implements Serializable {
    @Id @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;
    @ManyToMany(mappedBy="klienci")
    private List<Rezerwacja> rezerwacje
        = new ArrayList<Rezerwacja>();
    // + metody dostepowe dla powyzszych pol...
}
```

Dwukierunkowa relacja wiele-do-wielu

```
<entity-mappings>
    <entity class="pl.altkom.Rezerwacja" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <many-to-many name="klienci"
                target-entity="pl.altkom.Klient">
                <join-table name="REZERWACJA_KLIENT">
                    <join-column name="Rezerwacja_id"/>
                    <inverse-join-column name="Klient_id"/>
                </join-table>
            </many-to-many>
        </attributes>
    </entity>
    <entity class="pl.altkom.Klient" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <many-to-many name="rezerwacje"
                target-entity="pl.altkom.Rezerwacja"
                mapped-by="klienci"/>
        </attributes>
    </entity>
</entity-mappings>
```

Dwukierunkowa relacja wiele-do-wielu

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Jednokierunkowa relacja wiele-do-wielu

- Z relacją tego typu mamy do czynienia, kiedy wiele komponentów encyjnych posiada właściwość reprezentowaną przez kolekcję referencji do innych komponentów, np. *Rezerwacja* → *Pokój*
- Encje są ze sobą powiązane za pomocą adnotacji umieszczonych po stronie pełniącej rolę właścielską:
 - `@javax.persistence.ManyToMany` oraz
 - `@javax.persistence.JoinTable`
- Jeżeli powierzamy proces generowania tabel usłudze utrwalania, to można zredukować adnotacje w *Rezerwacja* do `@ManyToMany`

Jednokierunkowa relacja wiele-do-wielu

```
@Entity
public class Rezerwacja implements Serializable {
    @Id @GeneratedValue
    private long id;
    @Temporal(TemporalType.DATE)
    private Date data;
    @ManyToMany
    @JoinTable(name="POKoj_REZERWACJA",
               joinColumns={@JoinColumn(name="Rezerwacja_id") },
               inverseJoinColumns={@JoinColumn(name="Pokoj_id") })
    private Set<Pokoj> pokoje = new HashSet<Pokoj>();
    // + metody dostepowe dla powyzszych pol...
}
```

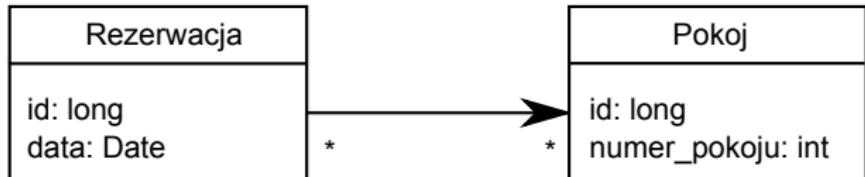
```
@Entity
public class Pokoj implements Serializable {
    @Id @GeneratedValue
    private long id;
    private int numer_pokoju;
    // + metody dostepowe dla powyzszych pol...
}
```

Jednokierunkowa relacja wiele-do-wielu

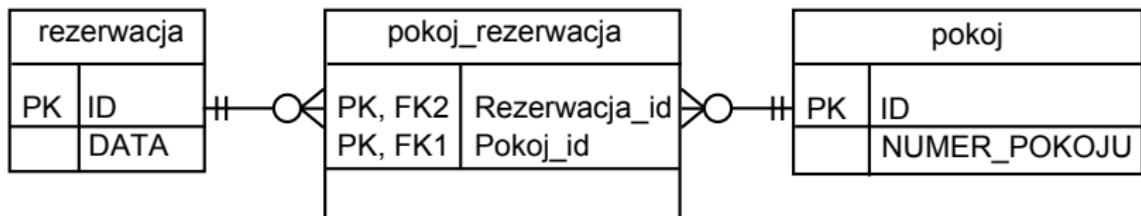
```
<entity-mappings>
    <entity class="pl.altkom.Rezerwacja" access="FIELD">
        <attributes>
            <id name="id">
                <generated-value/>
            </id>
            <many-to-many name="pokoje" target-entity="pl.altkom.Pokoj">
                <join-table name="Rezerwacje_Pokoje">
                    <join-column name="Rezerwacja_id"/>
                    <inverse-join-column name="Pokoj_id"/>
                </join-table>
            </many-to-many>
        </attributes>
    </entity>
</entity-mappings>
```

Jednokierunkowa relacja wiele-do-wielu

KLASY ENCYJNE:



TABELE W BAZIE DANYCH:



Propagacja kaskadowa

- Powoduje, że podczas wywoływania metod udostępnianych przez *EntityManager* na danym egzemplarzu encji, ich wykonanie zostanie automatycznie przeprowadzone na wszystkich jego elementach składowych
- Konfiguracja mechanizmu propagacji odbywa się za pomocą atrybutu **cascade** dla adnotacji **@OneToOne** , **@OneToMany** , **@ManyToOne** lub **@ManyToMany**
- Może on przybierać jedną lub kilka następujących wartości:
 - ALL
 - PERSIST
 - MERGE
 - REMOVE
 - REFRESH
- Standardowo propagacja jest wyłączona

Dziedziczenie encji

- JPA dostarcza trzy różne sposoby na **odwzorowywanie relacji dziedziczenia encji** w bazach danych:
 - prezentacja hierarchii klas w jednej tabeli (*single-table strategy*)
 - jedna tabela zawiera wszystkie właściwości, wszystkich klas
 - po każdej tabeli dla każdej klasy (*table-per-class strategy*)
 - każda klasa umieszczana jest w osobnej tabeli zawierającej jej wszystkie własności oraz własności nadklasy, jeśli taka istnieje
 - jedna tabela dla każdej podklasy (*joined strategy*)
 - każda klasa reprezentowana jest w osobnej tabeli, która zawiera właściwości tylko tej klasy

Reprezentacja hierarchii w pojedynczej tabeli

- Strategia wymaga stosowania dodatkowej kolumny (tzw. **dyskryminatora**), identyfikującego typ reprezentowanej w danym wierszu encji

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Dis",
                      discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Klient")
public class Klient implements Serializable {

    @Id
    @GeneratedValue
    private long id;

    private String imie;
    private String nazwisko;
    // + metody dostepowe do powyzszych pol
}
```

Reprezentacja hierarchii w pojedynczej tabeli

```
@Entity  
@DiscriminatorValue("STALY_KLIENT")  
public class StalyKlient extends Klient implements Serializable {  
  
    private double rabat;  
  
    // + metody dostepowe  
}
```

Reprezentacja hierarchii w pojedynczej tabeli

```
<entity-mappings>
    <entity class="pl.altkom.Klient">
        <inheritance strategy="SINGLE_TABLE"/>
        <discriminator-column name="DIS" discriminator-type="STRING"/>
        <discriminator-value>KLIENT</discriminator-value>
        <attributes>
            <id>
                <generated-value/>
            </id>
        </attributes>
    </entity>
    <entity class="pl.altkom.StalyKlient">
        <discriminator-value>STALY_KLIENT</discriminator-value>
    </entity>
</entity-mappings>
```

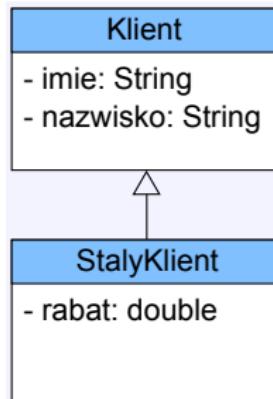
Reprezentacja hierarchii w pojedynczej tabeli

- Zalety:

- łatwa w implementacji
- zapewnia największą wydajność

- Wady:

- brak możliwości zastosowania ograniczenia *NOT NULL*
- wyklucza możliwość normalizacji



Single-table strategy

tabela *klient*

ID	Dis	IMIE	NAZWISKO	RABAT
1	Klient	Jan	Kowalski	null
2	STALY_Klient	Anna	Nowak	10.0

Jedna tabela dla konkretnej klasy

- Strategia wymaga zdefiniowania tylu tabel, ile klas znajduje się w danej hierarchii
- Wszystkie tabele zawierają kolumny opisujące właściwości danej klasy oraz wszystkich nadklas

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public class Klient implements Serializable {  
  
    @Id  
    @GeneratedValue  
    private long id;  
  
    private String imie;  
    private String nazwisko;  
    // + metody dostepowe do powyzszych pol  
}
```

Jedna tabela dla konkretnej klasy

```
@Entity  
public class StalyKlient extends Klient implements Serializable {  
  
    private double rabat;  
  
    // + metody dostepowe  
}
```

Jedna tabela dla konkretnej klasy

```
<entity-mappings>
    <entity class="pl.altkom.Klient">
        <inheritance strategy="TABLE_PER_CLASS"/>
        <attributes>
            <id>
                <generated-value/>
            </id>
        </attributes>
    </entity>
    <entity class="pl.altkom.StalyKlient"/>
</entity-mappings>
```

Jedna tabela dla konkretnej klasy

- Zalety:

- umożliwia stosowanie ograniczeń *NOT NULL*
- łatwiejsze odwzorowanie encji w schemacie bazy istniejącym przed tworzeniem komponentów

- Wady:

- najmniej wydajna

Klient	
- imie: String	
- nazwisko: String	



StalyKlient	
- rabat: double	

Table-per-concrete-class strategy

ID	IMIE	NAZWISKO
1	Jan	Kowalski

tabela
klient

ID	IMIE	NAZWISKO	RABAT
2	Anna	Nowak	10

tabela
stalyklient

Jedna tabela dla każdej podklasy

- Wszystkie podklasy odwzorowywane są w osobnych tabelach, z których każda zawiera właściwości wyłącznie z danej klasy

```
@Entity  
@Inheritance(strategy=InheritanceType.JOINED)  
public class Klient implements Serializable {  
  
    @Id  
    @GeneratedValue  
    private long id;  
  
    private String imie;  
    private String nazwisko;  
    // + metody dostepowe do powyzszych pol  
}
```

Jedna tabela dla każdej podklasy

```
@Entity  
@PrimaryKeyJoinColumn(name="SK_PK")  
public class StalyKlient extends Klient implements Serializable {  
  
    private double rabat;  
  
    // + metody dostepowe  
}
```

Jedna tabela dla każdej podklasy

```
<entity-mappings>
    <entity class="pl.altkom.Klient">
        <inheritance strategy="JOINED" />
        <attributes>
            <id>
                <generated-value/>
            </id>
        </attributes>
    </entity>
    <entity class="pl.altkom.StalyKlient">
        <primary-key-join-column name="SK_PK" />
    </entity>
</entity-mappings>
```

Jedna tabela dla każdej podklasy

- Zalety:

- umożliwia stosowanie ograniczeń *NOT NULL*
- oferuje większą wydajność niż *TABLE_PER_CLASS*

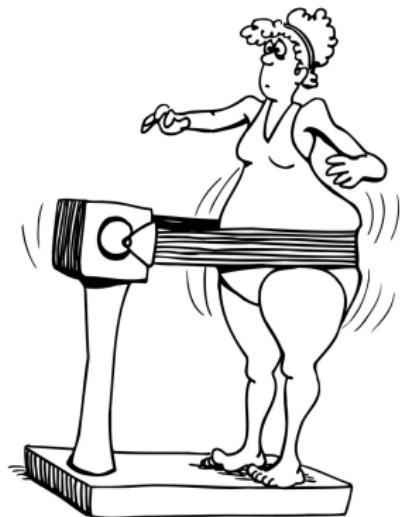
- Wady:

- efektywność niższa niż dla *SINGLE_TABLE*



Ćwiczenia

- Ćwiczenie 5.1:
*implementacja związków zachodzących
pomiędzy encjami*



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL**
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

6

Podstawy języka JPQL

- wprowadzenie do JPQL i interfejsu Query
- podstawowa składnia zapytań
- klauzule i wyrażenia funkcjonalne

Język JPQL

- Deklaratywny język zapytań służący do przetwarzania obiektów Javy
- Składnia w dużej mierze przypomina język SQL, stosowany z relacyjnymi bazami danych z tą różnicą, że zapytania odnoszą się do właściwości i wzajemnych relacji komponentów encyjnych, a nie rekordów
- Każde zapytanie tłumaczone jest automatycznie na tradycyjne zapytanie SQL'a i wykonywane za pomocą interfejsu JDBC
- Składnia JPQL jest w pełni przenośna (niezależna od użytej bazy danych)

Interfejs *Query*

- Do tworzenia i wywoływania zapytań służy interfejs *Query* definiujący poniższe metody:

```
public Query createQuery(String jpqlString)
public Query createNamedQuery(String name)
public Query createNativeQuery(String sqlString)
public Query createNativeQuery(String sqlString,
                               Class resultClass)
public Query createNativeQuery(String sqlString,
                               String resultSetMapping)
```

-
- Ich implementacja znajduje się w *EntityManager*

Metoda *createQuery*

- Pozwala na dynamiczne tworzenie zapytań, np.:

```
try {
    Query q = entityManager.createQuery(
        "SELECT k FROM Klient k WHERE k.imie='Marek'"
        + " AND k.nazwisko='Nowak'");
    Klient klient = (Klient)q.getSingleResult();
} catch (EntityNotFoundException e) {
    // obsługa wyjątku
} catch (NonUniqueResultException e) {
    // obsługa wyjątku
}
```

- Zapytanie wykonywane jest w momencie wywołania metody *getSingleResult*, która zwraca pojedynczy wynik

Metoda *createQuery*

- Alternatywna metoda *getResultSet* zwraca zbiór pasujących wyników
- Jej użycie nie wymaga obsługi wyjątku *EntityNotFoundException*, ponieważ w przypadku braku pasujących elementów zwracana jest pusta lista
- Przykład użycia:

```
Query q = entityManager.createQuery(  
    "SELECT k FROM Klient k WHERE k.imie='Marek'"  
        + " AND k.nazwisko='Nowak'");  
List klienci = q.getResultList();
```

Parametry zapytań

- Definiowane przez programistę zapytania mogą posiadać parametry (nazwane lub pozycyjne)
- Takie podejście pozwala na budowanie uniwersalnego kodu umożliwiającego wielokrotne użycie

```
Query q = entityManager.createQuery(  
    "SELECT k FROM Klient k WHERE k.imie=:i AND k.nazwisko=:n");  
q.setParameter("i", "Marek");  
q.setParameter("n", "Nowak");  
List klienci = q.getResultList();
```

```
Query q = entityManager.createQuery(  
    "SELECT k FROM Klient k WHERE k.imie=?1 AND k.nazwisko=?2");  
q.setParameter(1, "Marek");  
q.setParameter(2, "Nowak");  
List klienci = q.getResultList();
```

Ograniczanie liczby wyników

- Składnia JPQL pozwala ograniczać liczbę zwracanych wyników za pomocą następujących metod:
 - setMaxResults (maksymalna_liczba_wynikow)
 - setFirstResult (pozycja_elementu)
 - pozycja elementu w ramach całego zbioru, od którego zacznie się generowanie rezultatów
- Przykład użycia:

```
Query q = entityManager.createQuery(  
    "SELECT k FROM Klient k WHERE k.imie='Marek'");  
List klienci = q.setMaxResults(5).setFirstResult(2)  
                .getResultList();
```

Tworzenie zapytań

- Zapytanie JPQL w najprostszej postaci mogłoby wyglądać następująco:
`"SELECT k FROM Klient k"`
- **SELECT** służy do określenia typu wartości, które powinny być zwrócone – w tym przypadku będzie to komponent encyjny *Klient* opisywany na poziomie samego zapytania przez parametr *k*
- **FROM** definiuje typ komponentów uwzględnianych w generowanych wynikach
- Ewentualne użycie **DISTINCT** zaraz po klauzuli **SELECT** gwarantuje, że zwrócony wynik nie będzie posiadał powtarzających się wartości
- Jeżeli w klasie *Klient* ustalono atrybut **name** dla adnotacji **@Entity** to w zapytaniach można odwoływać się do encji *Klient* przez użytką tam nazwę

Tworzenie zapytań

- Klauzula **SELECT** może zwracać dowolną liczbę własności podstawowych encji, np.:

```
"SELECT k.imie, k.nazwisko FROM Klient AS k"
```

- Istnieje także możliwość tworzenia ścieżek nawigacyjnych, łączących egzemplarze komponentu encyjnego

```
"SELECT k.adres.ulica FROM Klient AS k"
```

Tworzenie zapytań

- Jeśli **SELECT** obejmuje więcej niż jedną kolumnę lub encję, wynik generowany będzie w postaci tablicy obiektów, np.:

```
Query q = entityManager.createQuery(  
    "SELECT k.imie, k.nazwisko FROM Klient AS k");  
List klienci = q.getResultList();  
Iterator i = klienci.iterator();  
while (i.hasNext()) {  
    Object[] wynik = (Object[])i.next();  
    System.out.println("Imię: " + (String)wynik[0]  
                      + " Nazwisko: " + (String)wynik[1]);  
}
```

Tworzenie zapytań

- W ramach polecenia **SELECT** istnieje możliwość zbudowania konstruktora służącego do tworzenia nieencyjnych obiektów Javy na podstawie zwracanych wyników
- Przykład użycia:

```
Query q = entityManager.createQuery(  
    "SELECT new pl.altkom.Info(k.imie, k.nazwisko) FROM Klient AS k");  
List<Info> klienci = q.getResultList();
```

Tworzenie zapytań

- Przykład użycia (cd.):

```
public class Info {  
    private String imie;  
    private String nazwisko;  
  
    public Info(String imie, String nazwisko) {  
        this.imie = imie;  
        this.nazwisko = nazwisko;  
    }  
  
    public String getImie() {  
        return imie;  
    }  
  
    public String getNazwisko() {  
        return nazwisko;  
    }  
}
```

Operatory **IN**, **INNER JOIN** oraz **LEFT JOIN**

- Operator **IN** oraz **INNER JOIN** umożliwiają stosowanie identyfikatora wskazującego poszczególne elementy kolekcji składowanej w polu reprezentującym relację, np.:

```
SELECT t FROM Klient AS k, IN(k.telefony) t
SELECT t.numer FROM Klient AS k, IN(k.telefony) t
SELECT t FROM Klient k INNER JOIN k.telefony t
SELECT t.numer FROM Klient k INNER JOIN k.telefony t
```

-
- Operator **LEFT JOIN** pozwala uzyskiwać wyniki, gdy po jednej stroniełączenia tego operatora występują wartości puste, np.:

```
SELECT k.imie, k.nazwisko, p.telefon
FROM Klient k LEFT JOIN k.telefony p
```

Klauzula WHERE

- Umożliwia zawężenie zakresu zwracanych wyników przy pomocy wyrażeń stałych i odpowiednich operatorów
- Do dyspozycji programisty są:
 - operator nawigacji: (.)
 - operatory arytmetyczne: *, /, +, -
 - operatory porównania: =, >, >=, <, <=, <>, **LIKE**, **BETWEEN**, **IN**, **IS NULL**, **IS EMPTY**, **MEMBER OF**
 - operatory logiczne: **NOT**, **AND**, **OR**
- Przykład użycia:

```
SELECT k FROM Klient AS k WHERE k.wiek >= 18 AND k.wiek <= 32
```

Wyrażenia funkcjonalne

- Pozwalają na przetwarzaniełańcuchów i wartości numerycznych:

```
LOWER(String)
UPPER(String)
TRIM([[LEADING | TRAILING | BOTH] [trim_char] FROM] String)
CONCAT(String1, String2)
LENGTH(String)
LOCATE(String1, String2 [, start])
SUBSTRING(String1, start, length)
ABS(number)
SQRT(double)
MOD(int, int)
```

Wyrażenia funkcjonalne

- Do dyspozycji mamy trzy funkcje zwracające aktualną datę, godzinę lub znacznik czasowy:

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP

- Istnieją również funkcje agregujące:

COUNT (identyfikator lub sciezka)

MAX (sciezka)

MIN (sciezka)

AVG (wartosc numeryczna)

SUM (wartosc numeryczna)

Klauzula *ORDER BY*

- Umożliwia sortowanie zwracanych wyników
- Może być stosowana w zapytaniach zawierających **WHERE** lub nie
- Domyślnie stosowany jest porządek rosnący
- Kolejność można określić za pomocą **ASC** i **DESC**
- Przykład zastosowania:

```
SELECT k FROM Klient AS k ORDER BY k.nazwisko ASC, k.imie DESC
```

Klauzula **GROUP BY** i **HAVING**

- Klauzula **GROUP BY** umożliwia dzielenie zwracanych wyników według określonej kategorii

```
SELECT cr.name, COUNT (res) FROM Cruise cr
    LEFT JOIN cr.reservations res
    GROUP BY cr.name
```

-
- **HAVING** stosowane z **GROUP BY** może ograniczyć wyniki generowane przez zapytanie (można używać tylko te wyrażenia agregujące, które zostały użyte w **SELECT**)

```
SELECT cr.name, COUNT (res) FROM Cruise cr
    JOIN cr.reservations res
    GROUP BY cr.name
    HAVING count(res) > 10
```

Podzapytania

- To zapytania mające postać wyrażeń **SELECT**, stosowane w ramach innych zapytań (**WHERE** i **HAVING**)
- Przykład użycia:

```
SELECT COUNT(res) FROM Reservation res
    WHERE res.amountPaid > (SELECT avg(r.amountPaid)
        FROM Reservation r)
```

-
- W momencie, kiedy podzapytanie zwraca dużą liczbę wierszy można dodatkowo ograniczyć ich liczbę za pomocą operatorów **ALL**, **ANY**, **SOME** lub **EXISTS**

Operacje **UPDATE** i **DELETE**

- Operacje **UPDATE** i **DELETE** służą odpowiednio do aktualizowania i usuwania rekordów bazy, przy czym mogą być wywoływane na całych zbiorach wierszy, np.:

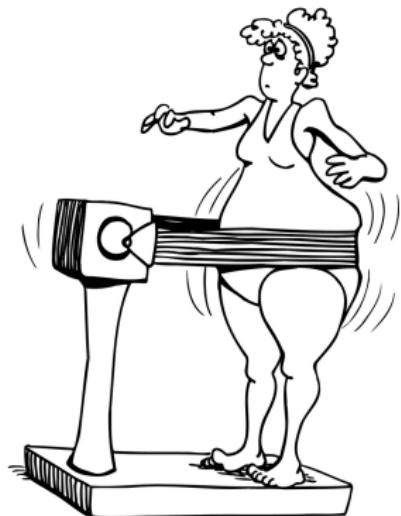
```
UPDATE Rezerwacja r SET r.cena = (r.cena+100)
    WHERE EXISTS (SELECT k FROM r.klienci k
                  WHERE k.nazwisko='Nowak')
```

```
DELETE FROM Rezerwacja r
    WHERE EXISTS (SELECT k FROM r.klienci k
                  WHERE k.nazwisko='Nowak')
```

- Stosując zbiorowe aktualizacje lub usuwanie można doprowadzić do niespójności zawartości bazy danych i encji

Ćwiczenia

- Ćwiczenie 6.1:
*wykorzystanie składni języka JPQL do
przetwarzania informacji
przechowywanych w systemie*



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące**
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

7

Wywołania zwrotne i klasy nasłuchujące

- wywołania zwrotne
- klasy nasłuchujące

Zdarzenia związane z cyklem życia encji

- Każde użycie takich metod jak: *persist*, *merge*, *remove*, *find*, czy nawet wykonanie zapytania, pociąga za sobą zbiór zdarzeń związanych z cyklem życia kontekstu utrwalania
- JPA pozwala na definiowanie w klasach komponentów specjalnych **metod zwrotnych**, informujących encje o zachodzących zdarzeniach
- Istnieje także możliwość tworzenia tzw. **klas nasłuchujących**, które po zarejestrowaniu będą reagowały identycznie

Wywołania zwrotne

- Wszystkie etapy związane z cyklem życia komponentu encyjnego reprezentują poniższe adnotacje:
 - `@javax.persistence.PrePersist`
 - `@javax.persistence.PostPersist`
 - `@javax.persistence.PostLoad`
 - `@javax.persistence.PreUpdate`
 - `@javax.persistence.PostUpdate`
 - `@javax.persistence.PreRemove`
 - `@javax.persistence.PostRemove`

Wywołania zwrotne

- W momencie zajścia zdarzenia wywołana zostaje metoda oznaczona odpowiednią adnotacją
- Alternatywą dla adnotacji jest użycie elementów XML, np.:

```
...
<entity class="pl.altkom.test.Klient">
    <post-persist method-name="poUtrwaleniu"/>
    <post-remove method-name="poUsunieciu"/>
</entity>
...
```

- Wspomniane metody muszą spełniać poniższe warunki:
 - nie mogą pobierać żadnych argumentów
 - typ zwracany musi być ustawiony na **void**
 - nie mogą generować nie weryfikowalnych wyjątków

Klasy nasłuchujące

- Obsługa zdarzeń może zostać przeniesiona z encji do osobnej klasy
- Związek encji i klasy nasłuchującej określa odpowiednia adnotacja lub składnia XML
- Metody przechwytyjące muszą zwracać typ **void** i przyjmować pojedynczy parametr typu *Object* reprezentujący egzemplarz encji, której dotyczy zdarzenie
- Każda z takich metod powinna być oznaczona odpowiednią adnotacją – określającą typ zdarzenia (alternatywnie opisana za pomocą elementów XML)
- Dodatkowo klasa musi posiadać bezargumentowy konstruktor publiczny

Przykład klasy nasłuchującej

```
public class GeneralListener {  
    @PostPersist  
    public void postPer(Object entity) {  
        System.out.println("Entity " +  
            entity.getClass().getSimpleName() + " persisted");  
    }  
  
    @PostRemove  
    public void postRem(Object entity) {  
        System.out.println("Entity " +  
            entity.getClass().getSimpleName() + " removed");  
    }  
  
    @PostLoad  
    public void postLoad(Object entity) {  
        System.out.println("Entity " +  
            entity.getClass().getSimpleName() + " loaded");  
    }  
}
```

Klasy nasłuchujące

- W celu zarejestrowania klasy słuchacza można użyć:
 - adnotacji `@javax.persistence.EntityListeners`
 - elementów XML
- Kolejność wywoływanego metod jest zgodna z kolejnością definiowania klas słuchaczy

Klasy nasłuchujące

- Rejestracja listenera za pomocą adnotacji:

```
@Entity
@EntityListeners({KlientListener.class})
public class Klient implements Serializable {
    @Id
    @GeneratedValue
    private long id;
    private String imie;
    private String nazwisko;

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    // + pozostałe metody dostepowe...
}
```

Klasy nasłuchujące

- Rejestracjalistenerza pomocą elementów XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
    xmlns="http://java.sun.com/xml/ns/persistence/orm" ...>
    <entity class="pl.altkom.Klient">
        <entity-listeners>
            <entity-listener class="pl.altkom.KlientListener">
                <post-persist method-name="poUtrwaleniu"/>
                <post-remove method-name="poUsunieciu"/>
            </entity-listener>
        </entity-listeners>
    </entity>
</entity-mappings>
```

Domyślne klasy nasłuchujące

- Istnieje możliwość definiowania domyślnych klas nasłuchujących stosowanych dla wszystkich encji z danej jednostki utrwalania

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
    xmlns="http://java.sun.com/xml/ns/persistence/orm" ...>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener class="pl.altkom.KlientListener">
                    <post-persist method-name="poUtrwaleniu"/>
                    <post-remove method-name="poUsunieciu"/>
                </entity-listener>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

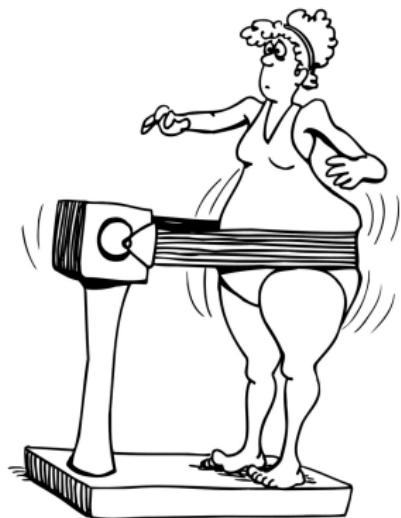
- W celu zignorowania domyślnych klas słuchaczy można użyć `@javax.persistence.ExcludeDefaultListeners` lub `<exclude-default-listeners/>`

Dziedziczenie i klasy nasłuchujące

- W przypadku, kiedy mamy do czynienia z hierarchią encji, wszystkie podklasy dziedziczą klasy nasłuchujące z klas bazowych
- Jeśli podklasy posiadają własne klasy nasłuchujące, zdarzenia będą przechwytywane przez jedne i drugie (z uwzględnieniem kolejności zgodnej z hierarchią dziedziczenia)
- Istnieje możliwość ignorowania dziedziczonych klas słuchaczy, jeśli skorzystamy z adnotacji
@javax.persistence.ExcludeSuperclassListeners

Ćwiczenia

- Ćwiczenie 7.1:
*realizacja wywołań zwrotnych
związanych z cyklem życia encji na
poziomie adnotacji i składni XML*



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne**
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

8

Komponenty sesyjne

- charakterystyka i zastosowanie komponentów sesyjnych
- komponenty bezstanowe
- komponenty stanowe

Komponenty sesyjne

- Służą do implementowania konkretnych zadań biznesowych
- Zarządzają interakcjami między komponentami encyjnymi
- Mimo, że nie reprezentują danych przechowywanych w bazie mają do nich dostęp – mogą odczytywać, aktualizować i dodawać informacje w ramach wywoływanych metod
- Dzielą się na komponenty stanowe i bezstanowe

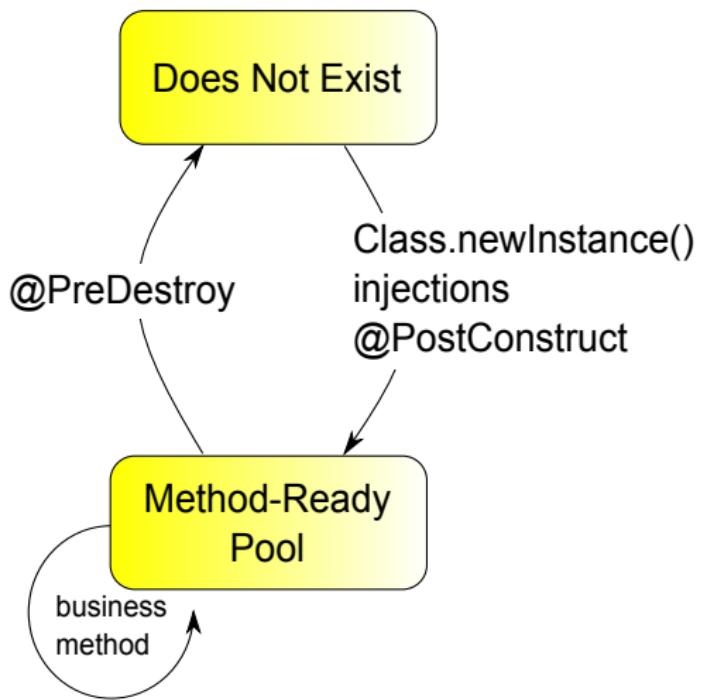
Komponenty sesyjne

- Mogą oferować dowolną liczbę interfejsów biznesowych – zdalnych lub lokalnych
- Zdalne interfejsy umożliwiają wywołanie metod za pośrednictwem sieci, natomiast lokalne są dostępne w ramach tej samej JVM
- Wywołanie metody za pośrednictwem interfejsów zdalnych powoduje kopiowanie wartości parametrów przekazywanych i zwracanych
- Wywołanie metody za pośrednictwem interfejsów lokalnych powoduje kopiowanie referencji do parametrów przekazywanych i zwracanych
- Specyfikacja pozwala na definiowanie wspólnej klasy bazowej dla interfejsów zdalnych i lokalnych, jeśli oba udostępniają te same metody

Bezstanowy komponent sesyjny

- Stanowi kolekcję wzajemnie powiązanych usług, z których każda realizowana jest przez odpowiednią metodę
- Między kolejnymi wywołaniami stan komponentu nie jest utrzymywany – wynik działania metody nie jest zależny od innych żądań realizowanych wcześniej przez komponent
- Bezpośrednio po obsłudze jednego żądania, bezstanowy komponent sesyjny może przystąpić do realizacji kolejnego, pochodzącego od innego klienta
- Odznaczają się dużą wydajnością i lekkością
- Nie obsługują mechanizmu aktywacji/pasywacji

Bezstanowy komponent sesyjny – cykl życia



Bezstanowy komponent sesyjny – cykl życia

- Stan “*nie istnieje*” – egzemplarz komponentu nie jest składowany w pamięci systemu (nie został jeszcze utworzony)
- Stan “*w puli gotowych komponentów*” – podczas przejścia do tego stanu tworzony jest nowy egzemplarz komponentu, wstrzykiwane są zasoby skonfigurowane przez metadane, a także generowane jest odpowiednie zdarzenie reprezentujące zakończenie procesu konstruowania komponentu (komponent trafia do puli, gdzie oczekuje na żądania od klientów)
- W sytuacji, kiedy komponent przestaje być potrzebny zostaje przeniesiony do stanu “*nie istnieje*”, następuje usunięcie referencji i zwolnienie pamięci, co również jest związane z odpowiednim zdarzeniem

Bezstanowy komponent sesyjny – zdarzenia związane z cyklem życia

- Klasa komponentu zainteresowana określonym zdarzeniem powinna posiadać metody do jego obsługi
- Wspomniane metody powinny być oznaczone przez odpowiednią adnotację:
@javax.annotation.PostConstruct
lub:
@javax.annotation.PreDestroy
 - alternatywnym sposobem jest wpis w deskryptorze wdrożenia
- Każda z metod obsługujących zdarzenie musi spełniać poniższe warunki:
 - zwraca typ **void**
 - nie może pobierać żadnych argumentów
 - nie może generować wyjątków nie weryfikowalnych

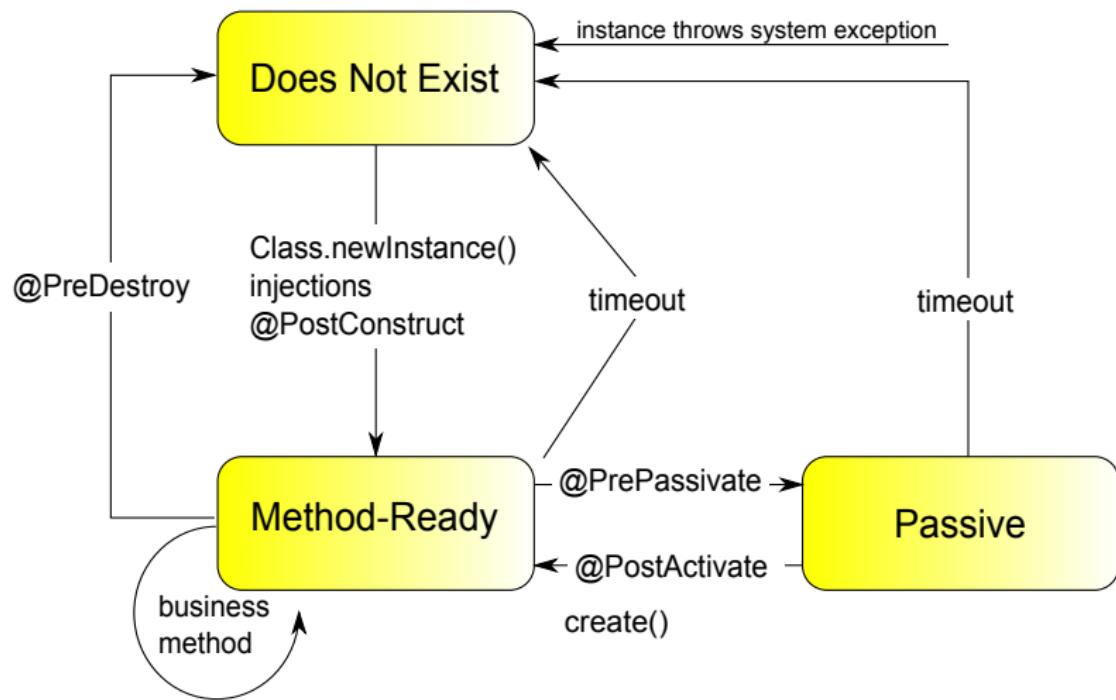
Stanowy komponent sesyjny

- Stanowi rozszerzenie aplikacji klienckiej – komponenty tego typu wykonują zadania w imieniu klienta, utrzymując przy tym stan konwersacji w ramach sesji
- Reprezentują logikę, która mogłyby być zaimplementowana w ramach aplikacji klienckiej systemu dwuwarstwowego
- W przeciwieństwie do komponentów bezstanowych, nie są składowane w puli – utworzony komponent po przypisaniu do konkretnego klienta pozostaje z nim związany przez cały cykl życia (ewentualnie następuje aktywacja/pasywacja)
- Każde żądanie pochodzące od jednego klienta jest obsługiwane przez ten sam egzemplarz komponentu

Stanowy komponent sesyjny

- Mimo utrzymywania stanu konwersacji, same komponenty nie mają charakteru trwałego – nie reprezentują danych składowanych w bazie, ale mają do nich dostęp
- Umożliwiają hermetyczne zamykanie i przenoszenie na serwer logiki biznesowej oraz stanu konwersacji z klientem, co w znacznym stopniu ułatwia zarządzanie systemem
- Można powiedzieć, że komponenty tego typu reprezentują klientów, ponieważ oprócz realizacji zleconych zadań pozwalają na komunikację z innymi komponentami
- Stanowe komponenty sesyjne mogą używać rozszerzonych kontekstów utrwalania

Stanowy komponent sesyjny – cykl życia



Stanowy komponent sesyjny – cykl życia

- Stan “*nie istnieje*” – egzemplarz nie został zainicjowany i nie znajduje się w pamięci systemowej
- Stan “*gotowy do obsługi wywołania*” – w momencie pierwszego żądania klienta tworzony jest nowy egzemplarz komponentu, wstrzykiwane są niezbędne zależności, a także generowane jest odpowiednie zdarzenie. Na tym etapie komponent zostaje związany z konkretnym klientem. Na koniec następuje wywołanie żądanej metody
- Funkcjonowanie w stanie “*gotowy do obsługi wywołania*” – egzemplarz oczekuje kolejnych żądań od klienta, podczas obsługi których utrzymywany jest stan konwersacji
- Wychodzenie ze stanu “*gotowy do obsługi wywołania*” – jest możliwe na dwa sposoby – przejście do stanu “*nie istnieje*” lub “*pasywowany*”. W obydwu przypadkach generowane są odpowiednie zdarzenia

Stanowy komponent sesyjny – cykl życia

- W czasie życia stanowego komponentu sesyjnego mogą występować okresy bezczynności (kiedy nie są realizowane żadne żądania od klienta). W tym przypadku kontener może zdecydować o jego pasywacji – stan komponentu jest zapisywany w pamięci dodatkowej, a sam komponent jest niszczony
- Komponent może zostać aktywowany w momencie, kiedy nadaje żądanie od klienta lub zniszczony po przekroczeniu limitu czasowego

Stanowy komponent sesyjny – zdarzenia związane z cyklem życia

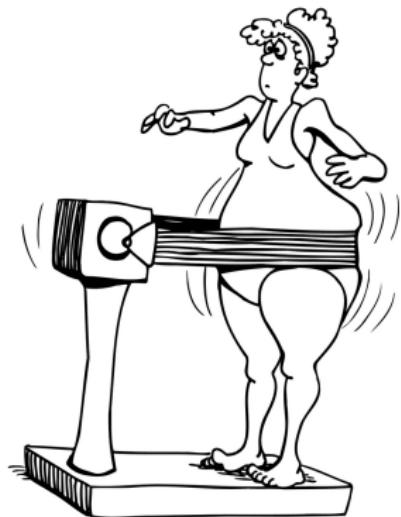
- Klasa komponentu zainteresowana określonym zdarzeniem powinna posiadać metody do jego obsługi
- Metody muszą być oznaczone przez odpowiednią adnotację:
 - `@javax.annotation.PostConstruct`
 - `@javax.annotation.PreDestroy`
 - `@javax.annotation.PrePassivate`
 - `@javax.annotation.PostActivate`
 - alternatywnym sposobem jest wpis w deskryptorze wdrożenia
- Każda z metod obsługujących zdarzenie musi spełniać poniższe warunki:
 - zwraca typ **void**
 - nie może pobierać żadnych argumentów
 - nie może generować wyjątków nieweryfikowalnych

Stanowy komponent sesyjny – wyjątki systemowe

- Każde wystąpienie wyjątku systemowego podczas wywołania metody komponentu powoduje unieważnienie obiektu EJB i zniszczenie komponentu sesyjnego
- Komponent automatycznie przechodzi do stanu “*nie istnieje*” bez generowania zdarzenia *PreDestroy*

Ćwiczenia

- Ćwiczenie 8.1:
implementacja stanowego komponentu sesyjnego



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami**
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

9

Komponenty sterowane komunikatami

- usługa JMS
- komunikaty i modele przesyłania
- komponenty sterowane komunikatami

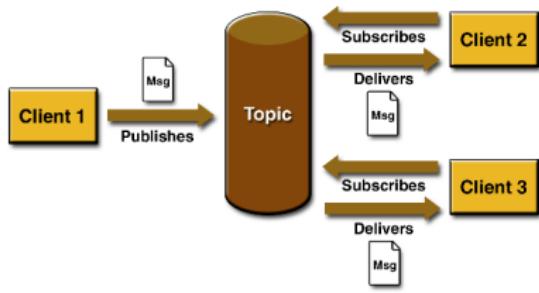
Java Message Service

- Specyfikacja EJB 3.0 narzuca dostawcom kontenerów obowiązek wsparcia dla usługi przesyłania komunikatów, ponieważ jest ona niezbędna do prawidłowego działania komponentów MDB
- JMS to niezależne od producentów API, za pomocą którego można korzystać z korporacyjnych systemów przesyłania komunikatów (*Message-Oriented Middleware*)
- Systemy typu MOM wspomagają procesy wymiany komunikatów między aplikacjami za pośrednictwem sieci
- Przykłady systemów obsługujących standard JMS:
 - JBossMQ
 - Sun ONE Message Queue
 - WebLogic JMS firmy BEA

Modele przesyłania komunikatów

- Specyfikacja JMS oferuje dwa modele przesyłania komunikatów:
 - **publikacja-subskrypcja** (*publish-and-subscribe*)
 - w skrócie *pub-sub*
 - opisuje rozsyłanie komunikatów *jeden-do-wielu*
 - **punkt-punkt** (*point-to-point*)
 - w skrócie *p2p*
 - działa w trybie przesyłania *jeden-do-jednego*

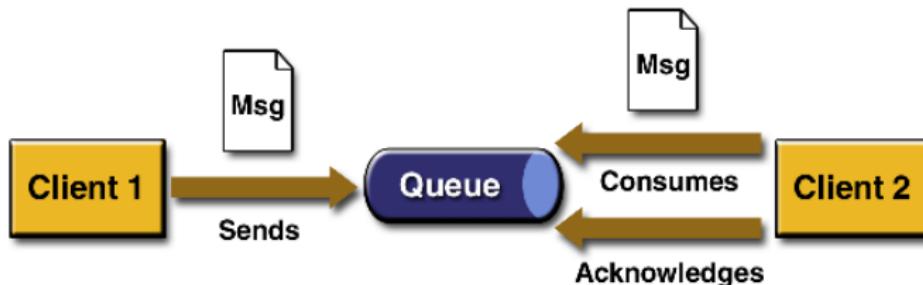
Model publikacja-subskrypcja



- Umożliwia producentowi wysłanie komunikatu do wielu konsumentów za pośrednictwem kanału wirtualnego określonego jako **temat** (*topic*)
- Aplikacje klienckie rejestrują zainteresowanie subskrypcją danego tematu i od tej pory otrzymują wszystkie komunikaty z nim związane

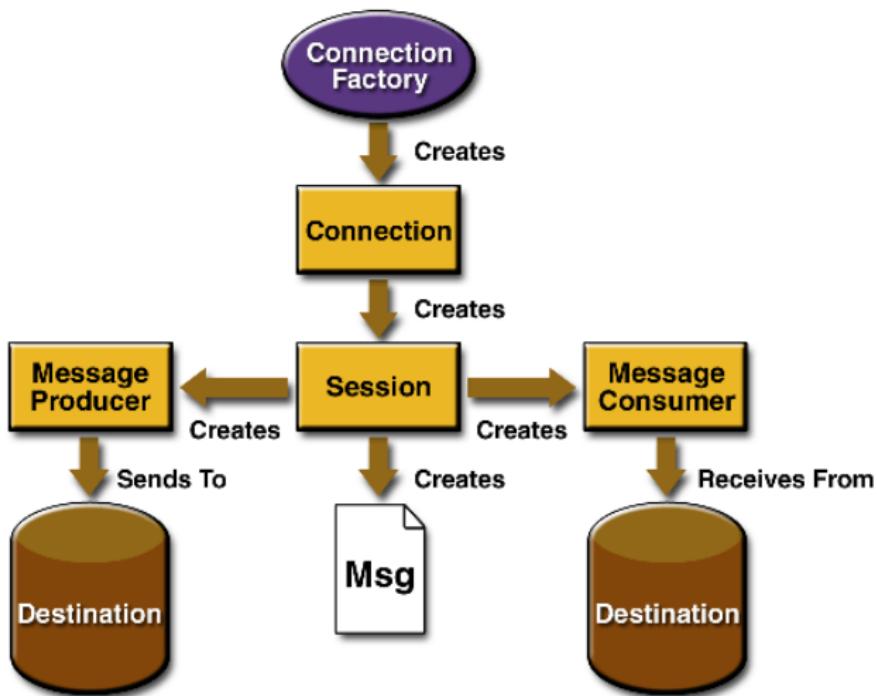
- Jest to przykład modelu wpychania komunikatów (*push-based model*) – komunikaty są automatycznie rozsyłane do wszystkich konsumentów
- Producent nie jest w żaden sposób uzależniony od konsumentów odbierających komunikat

Model punkt-punkt



- Umożliwia synchroniczne i asynchroniczne przekazywanie komunikatów poprzez kanały wirtualne zwane **kolejkami** (*queue*)
- Jest to przykład modelu wyciągania komunikatów (*pull-based model*) – komunikaty są wydobywane z kolejki na żądania klienta
- Dla pojedynczej kolejki może istnieć wielu odbiorców, ale dany komunikat może trafić tylko do jednego z nich

Model JMS API



Wysyłanie komunikatów do kolejki

```
QueueConnection conn = null;
QueueSender sender = null;
QueueSession sess = null;
Queue queue = null;
try {
    InitialContext ctx = new InitialContext();
    queue = (Queue) ctx.lookup("queue/myQueue");
    QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    conn = factory.createQueueConnection();
    sess = conn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
} catch (Exception e) {
    e.printStackTrace();
}
TextMessage msg;
try {
    msg = sess.createTextMessage(request.getParameter("message"));
    sender = sess.createSender(queue);
    sender.send(msg);
    sess.close();
} catch (JMSEException e) {
    e.printStackTrace();
}
```

Wysyłanie komunikatów do tematu

```
TopicConnection conn = null;
TopicPublisher sender = null;
TopicSession sess = null;
Topic topic = null;
try {
    InitialContext ctx = new InitialContext();
    topic = (Topic) ctx.lookup("topic/myTopic");
    TopicConnectionFactory factory =
        (TopicConnectionFactory) ctx.lookup("ConnectionFactory");
    conn = factory.createTopicConnection();
    sess = conn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
} catch (Exception e) {
    e.printStackTrace();
}
TextMessage msg;
try {
    msg = sess.createTextMessage(request.getParameter("message"));
    sender = sess.createPublisher(topic);
    sender.send(msg);
    sess.close();
} catch (JMSEException e) {
    e.printStackTrace();
}
```

Typy komunikatów

- Wszystkie komunikaty mają postać obiektów złożonych z nagłówka (*header*) i ciała komunikatu (*message body*)
- Nagłówek pozwala na dostarczenie wiadomości do odpowiedniego adresata oraz zawiera informacje dodatkowe – metadane
- Ciało reprezentuje właściwą treść wiadomości, która może przybierać różną formę, np. *TextMessage*, *MapMessage*, *ObjectMessage*, itd.

Komponenty sterowane komunikatami

- Komponenty, których przeznaczeniem jest asynchroniczne przetwarzanie wiadomości otrzymywanych za pośrednictwem JMS
- Podobnie, jak pozostałe składniki EJB, działają w środowisku dostarczonym przez kontener, które zapewnia bezpieczeństwo, zarządzanie zasobami, współbieżność, itd.
- Ponieważ MDB mają reagować wyłącznie na odbierane komunikaty, nie implementują interfejsów lokalnych ani zdalnych
- Zamiast tego bazują na interfejsach definiujących metody pozwalające na przetwarzanie wiadomości. Najczęściej jest to *javax.jms.MessageListener* z metodą *onMessage(Message msg)*

Komponenty sterowane komunikatami – przykład

```
@MessageDriven(  
    mappedName="queue/myQueue",  
    activationConfig = {  
        @ActivationConfigProperty(propertyName="destinationType",  
            propertyValue="javax.jms.Queue")  
    }  
)  
public class MDBQueue implements MessageListener {  
    public void onMessage(Message msg) {  
        try {  
            TextMessage tmsg = (TextMessage) msg;  
            System.out.println("Received message: " + tmsg.getText());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Komponenty sterowane komunikatami – przykład

```
@MessageDriven(  
    mappedName="topic/myTopic",  
    activationConfig = {  
        @ActivationConfigProperty(propertyName="destinationType",  
            propertyValue="javax.jms.Topic")  
    }  
)  
public class MDBTopic implements MessageListener {  
    public void onMessage(Message msg) {  
        try {  
            TextMessage tmsg = (TextMessage) msg;  
            System.out.println("Received message: " + tmsg.getText());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Komponenty sterowane komunikatami

- Zgodnie z zaprezentowanym przykładem wszystkie MDB powinny być oznaczone specjalną adnotacją `@javax.ejb.MessageDriven` lub opisane odpowiednio za pomocą elementów XML
- Dodatkowo, istnieje możliwość definiowania zestawu właściwości typu *klucz-wartość* konfigurujących sposób zachowania komponentu
- W przypadku adnotacji należy użyć atrybutu `activationConfig` przyjmującego jako wartość tablicę adnotacji `@javax.ejb.ActivationConfigProperty`

Komponenty sterowane komunikatami – selektor komunikatów

- Mechanizm pozwalający na filtrowanie odbieranych wiadomości na podstawie zdefiniowanej wcześniej właściwości
- Definicja wygląda następująco:

```
@ActivationConfigProperty(propertyName="messageSelector",
                           propertyValue="id='ZK140'")
```

-
- Użyte właściwości mają postać dodatkowych nagłówków dołączanych do obiektu wiadomości w czasie jej tworzenia, np.:

```
Message msg = session.createTextMessage();
msg.setStringProperty("id", "ZK140");
```

-
- Mogą być typu *String* lub dowolnego typu prostego

Komponenty sterowane komunikatami – tryb potwierdzania

- Tryb potwierdzania dostarczania wiadomości przez kontener może zostać zdefiniowany w następujący sposób:

```
@ActivationConfigProperty(propertyName="acknowledgeMode",
                           propertyValue="Auto-acknowledge")
```

- Ustawienie wartości na *Auto-acknowledge* nakazuje wysyłanie potwierdzenia do dostawcy usługi JMS natychmiast po przekazaniu komunikatu do właściwego komponentu MDB
- Z kolei *Dups-ok-acknowledge* pozwala kontenerowi na wysyłanie potwierdzeń w czasie późniejszym

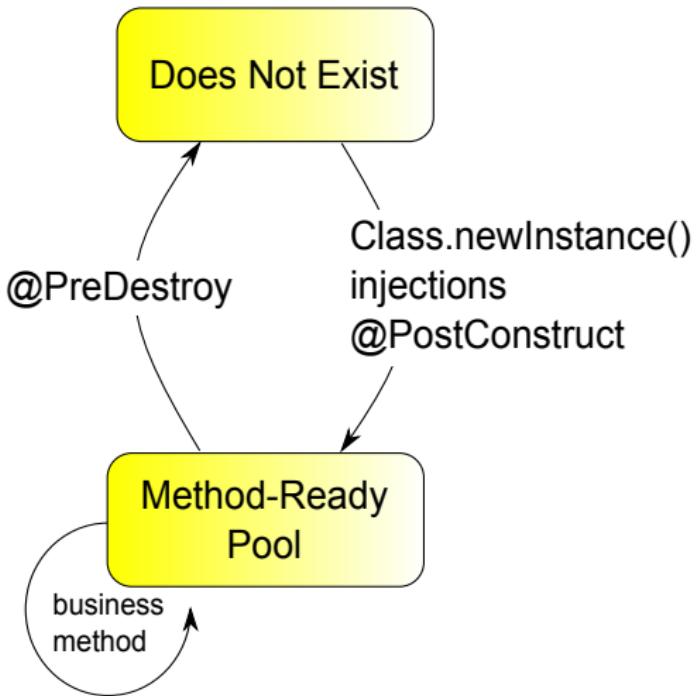
Komponenty sterowane komunikatami – określenie trwałości subskrypcji

- W przypadku komponentów korzystających z *javax.jms.Topic* można określić trwałość subskrypcji, np.:

```
@ActivationConfigProperty(propertyName="subscriptionDurability",
                           propertyValue="Durable")
```

- Kiedy atrybut przyjmuje wartość *Durable*, subskrypcja utrzymywana jest nawet, gdy połączenie kontenera z dostawcą JMS zostanie przerwane, natomiast przy *Non-Durable* – nie (w przypadku awarii komunikaty nie będą dostarczone)

Komponenty sterowane komunikatami – cykl życia



Komponenty sterowane komunikatami – cykl życia

- Stan “*nie istnieje*” – egzemplarz komponentu nie został jeszcze utworzony
- Stan “*w puli gotowych komponentów*” – podczas przejścia do tego stanu tworzony jest właściwy egzemplarz komponentu, wstrzykiwane są odpowiednie zasoby, generowane jest zdarzenie wywołujące metodę zwrotną oznaczoną przez `@PostConstruct` (spełniającą odpowiednie wymagania), a komponent trafia do puli, gdzie bierze udział w obsłudze komunikatów
- Kiedy serwer nie potrzebuje już komponentów, przechodzą one do stanu “*nie istnieje*”. Przejście poprzedza wywołanie metody oznaczonej przez `@PreDestroy`, która może np. zamykać otwarte zasoby

Ćwiczenia

- Ćwiczenie 9.1:
praktyczne wykorzystanie MDB



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService**
- 11 Obsługa transakcji
- 12 Bezpieczeństwo aplikacji

Plan modułu

10

Usługa TimerService

- omówienie usługi systemowej TimerService

Usługa TimerService

- Usługa udostępniania przez kontener, przeznaczona do powiadamiania komponentów EJB o osiągnięciu danego terminu, zakończeniu limitu czasowego lub nadejściu kolejnego okresu cyklu czasowego
- Komponent, który chce otrzymywać informacje o powyższych zdarzeniach musi spełnić jeden z dwóch warunków:
 - klasa komponentu musi implementować interfejs `javax.ejb.TimedObject` definiujący metodę
`public void ejbTimeout(Timer timer)`
 - odpowiednia metoda (zwracająca typ `void` i przyjmująca jeden argument `javax.ejb.Timer`) musi być oznaczona adnotacją
`@javax.ejb.Timeout`
- Kiedy nadejdzie zaplanowany termin lub upłynie określony przedział czasowy kontener automatycznie wywoła wskazaną metodę

Usługa TimerService

- Ponieważ to kontener odpowiada za powiadamianie komponentów o zdarzeniach zachodzących w czasie, musi posiadać dane o wszystkich planowanych terminach
- Ich rejestracja odbywa się za pomocą referencji do specjalnego obiektu *TimerService*, która może być uzyskana z kontekstu *EJBContext* lub wstrzyknięta za pomocą adnotacji
@javax.annotation.Resource

Usługa TimerService

```
@Stateless
public class Test implements TestRemote {
    @Resource
    private TimerService ts;

    @Timeout
    public void wykonajZadanie(Timer timer) {
        // logika wykonywana w momencie
        // wyzerowania licznika
    }

    public void ustawTermin(Date date) {
        ts.createTimer(date, null);
    }
}
```

Interfejs TimerService

- Definiuje metody pozwalające na dostęp do usług *TimerService* udostępnianych przez kontener
- Wśród metod wyróżniamy:

```
createTimer(Date expiration, Serializable info)
```

- tworzy licznik czasowy, który wyzeruje się tylko raz – w dniu określonym przez parametr *expiration*

```
createTimer(long duration, Serializable info)
```

- tworzy licznik czasowy, który wyzeruje się tylko raz – po upływie czasu określonego przez *duration*

Interfejs TimerService

- Metody interfejsu (cd.):

```
createTimer(Date initialExpiration, long intervalDuration,  
           Serializable info)
```

- tworzy licznik czasowy, który wyzeruje się pierwszy raz w dniu określonym przez *initialExpiration* i po każdym kolejnym odstępie czasu wyznaczonym przez *intervalDuration*

```
createTimer(long initialDuration, long intervalDuration,  
           Serializable info)
```

- tworzy licznik czasowy, który wyzeruje się pierwszy raz po upływie czasu *initialDuration* oraz po każdym kolejnym odstępie czasu wyznaczonym przez *intervalDuration*

Interfejs *TimerService*

- W momencie utworzenia licznika jest on utrwalany w dodatkowej pamięci, co zapobiega sytuacjom związanym z błędami systemowymi, takimi jak np. awaria serwera – liczniki uaktywnią się w czasie kolejnego uruchomienia serwera i spowodują wygenerowanie zdarzeń odpowiednią liczbę razy

Interfejs TimerService

- Interfejs posiada również metodę:

```
public java.util.Collection getTimers()
```

zwracającą wszystkie liczniki czasowe (obiekty typu *javax.ejb.Timer*) utworzone dla egzemplarza danego komponentu

- metoda jest często wykorzystywana do zarządzania istniejącymi licznikami, np. do ich anulowania

```
@Resource private TimerService ts;  
...  
public void anulujLiczniiki() {  
    for (Object timer: ts.getTimers()) {  
        ((Timer)timer).cancel();  
    }  
}
```

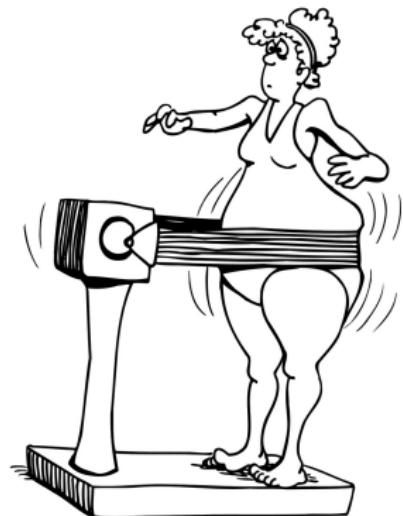
Obiekt javax.ejb.Timer

- Każdy obiekt tego typu reprezentuje dokładnie jedno zaplanowane zadanie dla określonego komponentu
- Może być używany do:
 - anulowania licznika czasowego
 - pozyskania informacji związanych z licznikiem
- Obiekt udostępnia następujące metody:

```
public void cancel()
public java.io.Serializable getInfo()
public java.util.Date getNextTimeout()
public long getTimeRemaining()
```

Ćwiczenia

- Ćwiczenie 10.1:
implementacja mechanizmu sprawdzającego cyklicznie ważność kont klientów



Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji**
- 12 Bezpieczeństwo aplikacji

Plan modułu

11

Obsługa transakcji

- pojęcie transakcji
- deklaratywne zarządzanie transakcjami
- jawne zarządzanie transakcjami

Transakcje

- Przez pojęcie **transakcji** rozumie się zbiór zależnych od siebie czynności podejmowanych w celu realizacji założonego zadania, z których każda musi zakończyć się sukcesem – innymi słowy: powinny być one ukończone wszystkie razem albo wcale

Transakcje

- Transakcje są **bezpieczne**, jeśli spełniają kryteria *ACID*, czyli są:
 - atomowe (*atomic*) — każda z czynności składających się na realizowane zadanie musi zostać wykonana bezbłędnie i w całości, w przeciwnym przypadku transakcja jest przerywana, a wykonane zmiany są cofane
 - spójne (*consistent*) — składowe systemu muszą zachować integralność w czasie i po zakończeniu transakcji
 - izolowane (*isolated*) — dane wykorzystywane podczas transakcji nie mogą być wykorzystywane przez inne elementy systemu w czasie trwania transakcji
 - trwałe (*durable*) — zmiany przeprowadzone podczas dokonywania transakcji muszą zostać utrwalone w pamięci fizycznej

Deklaratywne zarządzanie transakcjami

- Opiera się na definiowaniu atrybutów transakcji dla poszczególnych metod
- Odbywa się przy użyciu adnotacji **@javax.ejb.TransactionAttribute** lub elementów XML deskryptora wdrożenia *ejb-jar.xml*
- Jest to zalecany sposób obsługi transakcji w technologii EJB
- Pozwala na:
 - modyfikację ustawień bez konieczności dokonywania zmian w logice biznesowej
 - redukuje stopień złożoności
 - upraszcza tworzenie aplikacji wykorzystujących mechanizm transakcji i zmniejsza ryzyko występowania błędów

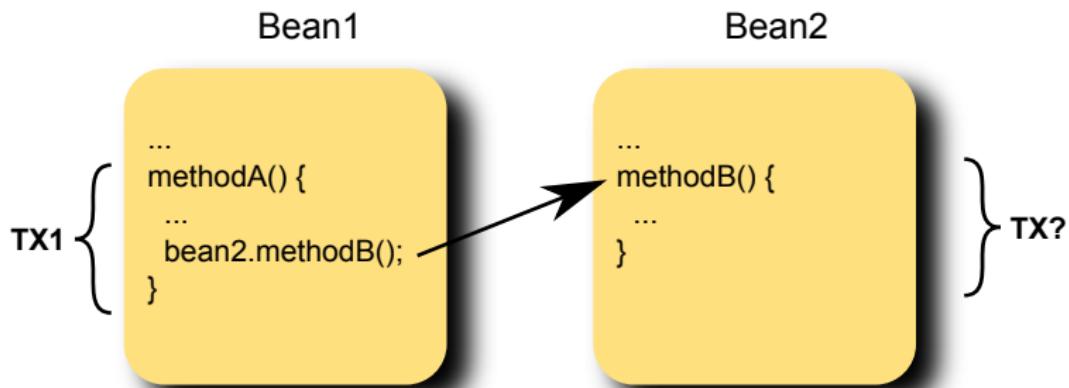
Atrybuty transakcji

- Atrybuty transakcji ustalane na poziomie adnotacji lub pliku XML mogą przyjmować następujące wartości:
 - **NotSupported**
 - **Supports**
 - **Required**
 - **RequiresNew**
 - **Mandatory**
 - **Never**
- Mogą definiować sposób zachowania transakcyjnego całego komponentu lub poszczególnych metod
- Jeżeli nie określi się żadnych atrybutów, domyślną wartością będzie **Required**

Atrybuty transakcji – przykład deklaracji w deskryptorze

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" ...  
    version=3.0>  
    <assembly-descriptor>  
        <container-transaction>  
            <method>  
                <ejb-name>nazwa komponentu</ejb-name>  
                <method-name>*</method-name>  
            </method>  
            <trans-attribute>atrybut transakcji</trans-attribute>  
        </container-transaction>  
        <container-transaction>  
            <method>  
                <ejb-name>nazwa komponentu</ejb-name>  
                <method-name>nazwa metody</method-name>  
            </method>  
            <trans-attribute>atrybut transakcji</trans-attribute>  
        </container-transaction>  
    </assembly-descriptor>  
</ejb-jar>
```

Atrybuty transakcji



Atrybuty transakcji – NotSupported

- Wywołanie metody zawiesza bieżącą transakcję do momentu jej zakończenia
- Zakres transakcji nie jest propagowany do komponentu, metody, ani żadnych wywołań, które mają tam miejsce
- Po zakończeniu metody działającej w trybie **NotSupported** pierwotna transakcja zostaje wznowiona

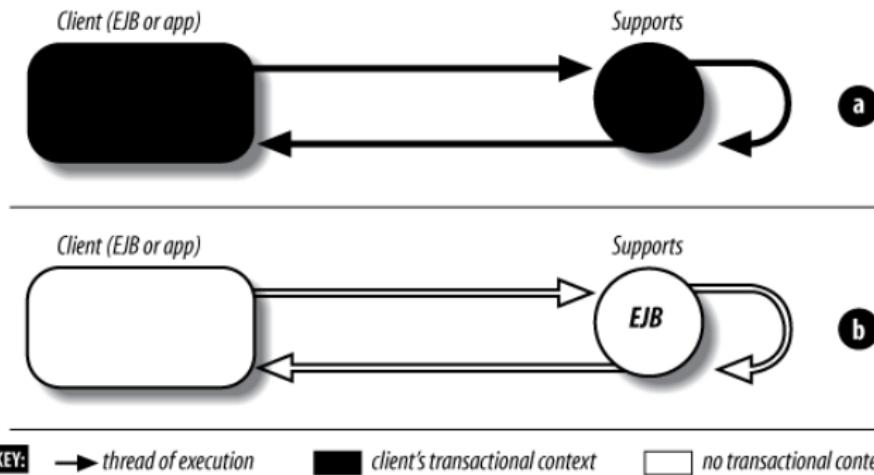
TRANSAKCJA	
Klienta	Metody biznesowej
–	–
<i>T1</i>	–



Atrybuty transakcji – Supports

- Jeżeli wywołanie metody pochodzi od elementu będącego częścią jakiejś transakcji metoda wywoływana zostanie w niej ujęta
- W przeciwnym przypadku będzie wykonywana poza transakcjami

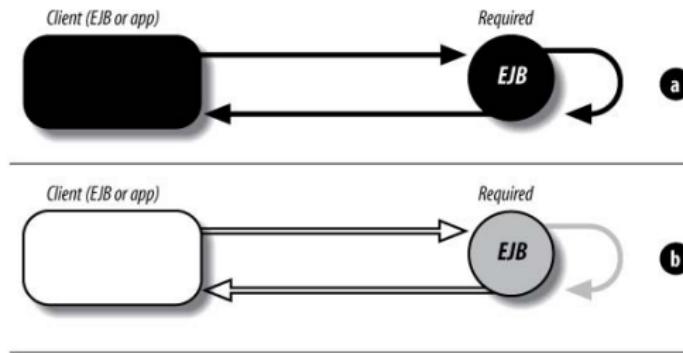
TRANSAKCJA	
Klienta	Metody biznesowej
-	-
T1	T1



Atrybuty transakcji – Required

- Wywołana metoda musi być wykonana wewnątrz zakresu transakcji
- Jeśli wywołujący jest częścią transakcji, metoda zostanie w niej uwzględniona
- W innym przypadku stworzona zostanie nowa transakcja, która zakończy działanie wraz z zakończeniem wywołanej metody

TRANSAKCJA	
Klienta	Metody biznesowej
–	T2
T1	T1

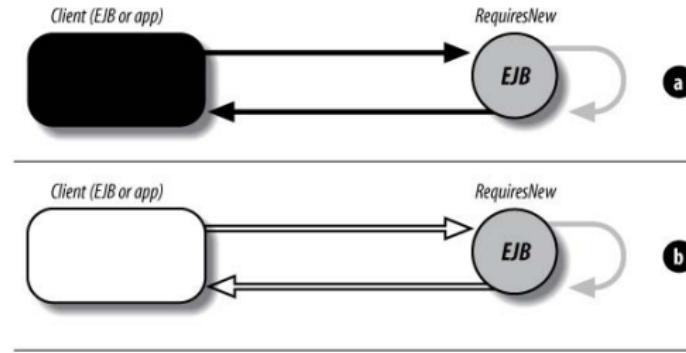


KEY: → thread of execution [black] client's transactional context [grey] EJB's transactional context [white] no transactional context

Atrybuty transakcji – RequiresNew

- Powoduje każdorazowe utworzenie transakcji dla wywoływanej metody
- Ewentualna transakcja wywołującego jest zawieszana do czasu zakończenia metody

TRANSAKCJA	
Klienta	Metody biznesowej
-	T2
T1	T2

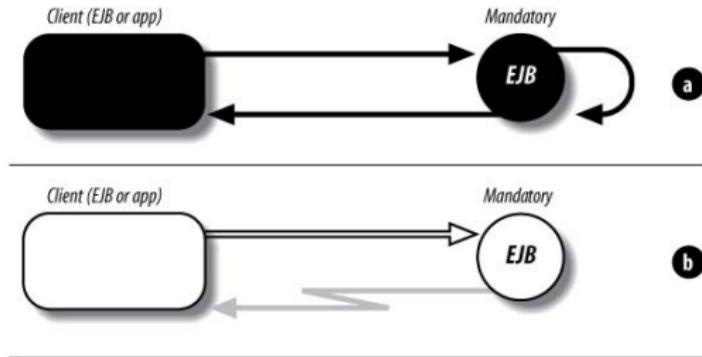


KEY: → thread of execution [black] client's transactional context [grey] EJB's transactional context [white] no transactional context

Atrybuty transakcji – Mandatory

- Wywołana metoda musi być częścią zakresu transakcji wywołującej
- Nie może zostać utworzona nowa transakcja
- W sytuacji, kiedy wywołujący nie jest częścią transakcji wystąpi wyjątek
javax.ejb.EJBTransactionRequiredException

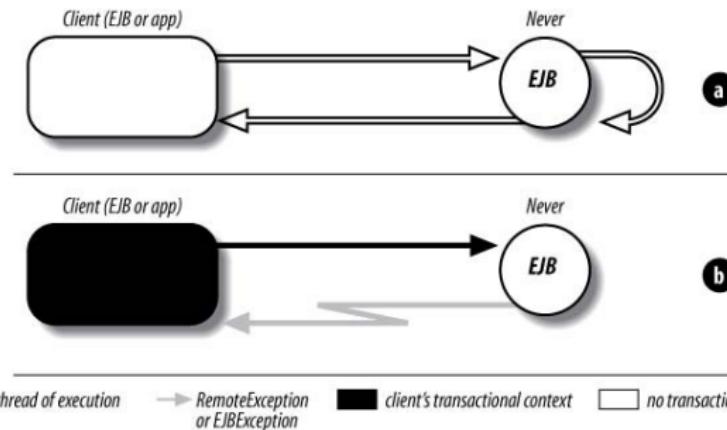
TRANSAKCJA	
Klienta	Metody biznesowej
–	wyjątek
T1	T1



Atrybuty transakcji – Never

- Metoda nie może być wykonana wewnątrz żadnej transakcji
- Jeśli wywołujący jest częścią jakiejś transakcji dojdzie do wyjątku *EJBException*

TRANSAKCJA	
Klienta	Metody biznesowej
–	–
<i>T1</i>	wyjątek



Atrybuty transakcji

- W celu zapewnienia bezpiecznego zarządzania dostępem do bazy danych przez kontener zaleca się stosowanie odwołań do usługi menedżera encji wewnątrz metod działających w trybie **Required** , **RequiresNew** , **Mandatory**
- W przypadku komponentów sterowanych komunikatami można stosować jedynie **NotSupported** i **Required**

Jawne zarządzanie transakcjami

- Jawne zarządzanie transakcjami w EJB realizowane jest przez JTA, czyli *Java Transaction API*
- W najprostszym przypadku sprowadza się do wykorzystania interfejsu *javax.transaction.UserTransaction* definiującego takie metody, jak: *begin* czy *commit*
- Aby komponent EJB mógł jawnie zarządzać transakcjami musi być oznaczony przez **@javax.ejb.TransactionManagement** (**TransactionManagementType.BEAN**)
- Jawne zarządzanie transakcjami nie jest możliwe w przypadku komponentów encyjnych

Uzyskanie obiektu typu *UserTransaction*

```
@Stateless  
@TransactionManagement(TransactionManagementType.BEAN)  
public class MyBean implements MyBeanRemote {  
    @Resource  
    private EJBContext ejbContext;  
  
    public void test() {  
        UserTransaction ut = ejbContext.getUserTransaction();  
        ...  
    }  
}
```

Uzyskanie obiektu typu *UserTransaction*

```
@Stateless  
@TransactionManagement(TransactionManagementType.BEAN)  
public class MyBean implements MyBeanRemote {  
    @Resource  
    private UserTransaction ut;  
  
    public void test() {  
        ...  
    }  
}
```

Uzyskanie obiektu typu *UserTransaction*

```
@Stateless  
@TransactionManagement(TransactionManagementType.BEAN)  
public class MyBean implements MyBeanRemote {  
  
    public void test() {  
        InitialContext jndiCtx;  
        try {  
            jndiCtx = new InitialContext();  
            UserTransaction ut = (UserTransaction)  
                jndiCtx.lookup("java:comp/env/UserTransaction");  
            ...  
        } catch (NamingException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Użycie obiektu typu *UserTransaction*

```
UserTransaction ut;  
...  
try {  
    ut.begin();  
    ...  
    ut.commit();  
} catch (NotSupportedException e) {  
    e.printStackTrace();  
} catch (SystemException e) {  
    e.printStackTrace();  
} catch (SecurityException e) {  
    e.printStackTrace();  
} catch (IllegalStateException e) {  
    e.printStackTrace();  
} catch (RollbackException e) {  
    e.printStackTrace();  
} catch (HeuristicMixedException e) {  
    e.printStackTrace();  
} catch (HeuristicRollbackException e) {  
    e.printStackTrace();  
}
```

Jawne zarządzanie transakcjami

- Metody udostępniane przez *UserTransaction*:

<i>begin()</i>	tworzy nową transakcję
<i>commit()</i>	kończy transakcję związaną z bieżącym wątkiem
<i>rollback()</i>	wycofuje transakcję i wszystkie wprowadzone zmiany
<i>setRollbackOnly()</i>	oznaczenie transakcji do wycofania
<i>setTransactionTimeout(int sec)</i>	ustala czas życia transakcji, musi być wywołana po <i>begin()</i>
<i>getStatus()</i>	zwraca aktualny stan transakcji

Zasięg transakcji

- Zależy od typu komponentu
- W przypadku bezstanowych komponentów sesyjnych ogranicza się do jednej metody
- Dla stanowych komponentów sesyjnych może rozciągać się na kilka metod
- Dla komponentów MDB zasięg trwa tak długo, jak metoda *onMessage()*

Plan szkolenia

- 1 Wprowadzenie
- 2 Podstawy architektury
- 3 Utrwalanie i usługa Entity Manager
- 4 Odwzorowania obiektowo-relacyjne
- 5 Zależności zachodzące między encjami
- 6 Podstawy języka JPQL
- 7 Wywołania zwrotne i klasy nasłuchujące
- 8 Komponenty sesyjne
- 9 Komponenty sterowane komunikatami
- 10 Usługa TimerService
- 11 Obsługa transakcji
- 12 **Bezpieczeństwo aplikacji**

Plan modułu

12

Bezpieczeństwo aplikacji

- podstawowe pojęcia związane z bezpieczeństwem
- deklaratywne zabezpieczanie aplikacji
- programowe zabezpieczanie aplikacji

Podstawowe pojęcia

- **Autentykacja** (*authentication*) – to identyfikacja użytkownika realizowana zazwyczaj przy użyciu loginu i hasła
- **Autoryzacja** (*authorization*) – pozwala sprawdzić, czy dany użytkownik ma prawo wykonywania żądanego przez niego operacji
- **Integralność danych** (*data integrity*) – polega na zagwarantowaniu spójności danych przekazywanych podczas komunikacji między nadawcą i odbiorcą
- **Poufność** (*confidentiality*) – proces ograniczający dostęp do istotnych informacji niepowołanym osobom

Uwierzytelnianie

- Obejmuje sprawdzenie, czy dany klient jest tym, za kogo się podaje
- Podczas logowania do systemu zdalny klient jest kojarzony z określonym identyfikatorem bezpieczeństwa
- Podczas próby wywołania metody, identyfikator propagowany jest przez kontener do obiektu EJB, który może dokonać jego weryfikacji
- Przykład przekazania loginu i hasła klienta jako atrybutów połączenia

```
SecurityClient client = SecurityClientFactory.getSecurityClient();  
client.setSimple("admin", "admin");  
client.login();
```

Uwierzytelnianie

- Proces uwierzytelniania wymaga stworzenia tzw. **Domeny Bezpieczeństwa**, na którą składają się deklaracje zbioru użytkowników, haseł i odpowiednich ról
- W przypadku serwera JBoss za konfigurację domeny odpowiada plik:
JBOSS_HOME/server/default/conf/login-config.xml
- Zastosowanie określonych reguł zabezpieczeń dla danego komponentu odbywa się na poziomie klasy przez adnotację:
`@org.jboss.ejb3.annotation.SecurityDomain ("nazwa domeny")`
- W momencie, kiedy podany login lub hasło będą niewłaściwe wyrzucony zostanie odpowiedni wyjątek

Uwierzytelnianie

- Przykład konfiguracji domeny bezpieczeństwa na serwerze JBoss:

```
<policy>
    <application-policy name="MojaDomena">
        <authentication>
            <login-module
                code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                flag="required">
                <module-option name="usersProperties">
                    users.properties
                </module-option>
                <module-option name="rolesProperties">
                    roles.properties
                </module-option>
            </login-module>
        </authentication>
    </application-policy>
</policy>
```

Uwierzytelnianie

- Przykład konfiguracji dla serwera GlassFish:

```
<glassfish-ejb-jar>
  ...
  <enterprise-beans>
    <ejb>
      <ejb-name>BankServiceBean</ejb-name>
      <ior-security-config>
        <as-context>
          <auth-method>USERNAME_PASSWORD</auth-method>
          <realm>default</realm>
          <required>true</required>
        </as-context>
      </ior-security-config>
    </ejb>
  </enterprise-beans>
</glassfish-ejb-jar>
```

Autoryzacja

- W momencie, kiedy użytkownik został uwierzytelniony następuje proces weryfikujący, czy posiada uprawnienia niezbędne do wywołania żądanej metody
- Proces autoryzacji opiera się na kojarzeniu ról z danym użytkownikiem, a także przypisywaniu uprawnień metody w oparciu o te role
- Role użytkowników określane są najczęściej w czasie wdrażania komponentu
- Uprawnienia metod nadawane są w czasie tworzenia komponentu za pomocą adnotacji lub składni XML

Autoryzacja

- W celu przypisania uprawnień dla metod EJB można wykorzystać adnotację `@javax.annotation.security.RolesAllowed`
- Wspomniana adnotacja pozwala na zdefiniowanie jednej lub kilku logicznych ról, które posiadają prawo dostępu do danej metody
- W przypadku użycia na poziomie klasy definiuje domyślny zestaw ról, mających dostęp do metod komponentu, przy czym każda z metod może przełonić to zachowanie za pomocą tej samej adnotacji
- Adnotacja `@javax.annotation.security.PermitAll` pozwala na dostęp do metody wszystkim użytkownikom, którzy zostali uwierzytelnieni. Może być używana na poziomie klasy lub metody. Wykorzystanie jest tożsame z sytuacją, kiedy zasady autoryzacji nie zostały określone

Autoryzacja w XML

- Konfiguracja autoryzacji dostępu umieszczona w pliku `ejb-jar.xml` odbywa się wewnątrz elementu `<assembly-descriptor>`
- Opiera się o:
 - definicje ról przy pomocy znaczników `<security-role>`, których obowiązkowym podelementem jest `<role-name>`
 - uprawnienia metod definiowane wewnątrz `<method-permission>`, w skład których wchodzą:
 - `<role-name>` określa dopuszczalne role, użycie `<unchecked>` odpowiada adnotacji `@PermitAll`
 - `<method>` definiuje metody do zabezpieczenia przy pomocy `<method-name>`, `<method-params>`, `<ejb-name>`, `<method-intf>`

Autoryzacja w XML

```
<ejb-jar version="3.0">
  <assembly-descriptor>
    <security-role>
      <description>opcjonalny opis roli</description>
      <role-name>nazwa roli</role-name>
    </security-role>
    <method-permission>
      <role-name>nazwa roli</role-name>
      <method>
        <ejb-name>nazwa komponentu</ejb-name>
        <method-name>nazwa metody</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

Programowe zabezpieczanie aplikacji

- Interfejs `javax.ejb.EJBContext` udostępnia metody mogące posłużyć do programowego zabezpieczenia aplikacji. Są to:

<code>getCallerPrincipal()</code>	zwraca obiekt typu <i>Principal</i> reprezentujący użytkownika aktualnie korzystającego z beana
<code>isCallerInRole(String rola)</code>	zwraca wartość logiczną mówiącą o tym, czy bieżący użytkownik przynależy do wskazanej roli

Ćwiczenia

- Ćwiczenie 12.1:
*implementacja autentykacji i autoryzacji
dostępu do wybranych metod aplikacji*

