

XV Seminarium PLOUG
Warszawa
Maj 2007

Tworzenie aplikacji dla Oracle Application Server 10g R3 w technologii EJB 3.0

Marek Wojciechowski
Politechnika Poznańska, PLOUG

e-mail: Marek.Wojciechowski@cs.put.poznan.pl

Abstrakt. Jedną z nowych cech serwera Oracle Application Server 10g R3 jest wsparcie dla standardu EJB 3.0. Komponenty Enterprise JavaBeans (EJB) od początku istnienia platformy Java Enterprise Edition były lansowane jako podstawowa technologia implementacji logiki biznesowej. Niestety wcześniejsze wersje tej technologii cechowały się nadmierną złożonością, a także mogły prowadzić do nieefektywnych rozwiązań, szczególnie w przypadku wykorzystania jej do komunikacji z bazą danych. W wersji 3.0 technologia EJB została znacząco uproszczona, a komunikacja z bazą danych została wyodrębniona jako odrębny standard o nazwie Java Persistence API. Celem niniejszego artykułu jest omówienie technologii EJB 3.0 i tworzenia w niej aplikacji dla serwera Oracle Application Server 10g R3 w środowisku Oracle JDeveloper 10g.

1. Wprowadzenie

Java Platform, Enterprise Edition (Java EE, dawniej J2EE) to najpoważniejsza platforma do tworzenia nowoczesnych, wielowarstwowych aplikacji klasy „enterprise”, wymagających efektywności, skalowalności i bezpieczeństwa. Java EE cechuje się dostępnością serwerów aplikacji i narzędzi programistycznych pochodzących od wielu renomowanych producentów oprogramowania, w tym m.in. Oracle i IBM, a także wielu dopracowanych i odpowiednich dla zastosowań produkcyjnych środowisk open source. Za zaletę Java Platform, Enterprise Edition można uznać również bogactwo technologii składowych i szkieletów aplikacji, z których skorzystać mogą twórcy aplikacji. Podstawowym zarzutem stawianym platformie Java EE była zawsze jej złożoność. Problem ten dotyczył w szczególności technologii Enterprise JavaBeans (EJB), lansowanej od początku przez firmę Sun jako podstawowa technologia implementacji logiki biznesowej na tej platformie. Technologia EJB aż do wersji 2.1, stanowiącej jedną ze specyfikacji J2EE 1.4, była nie tylko nadmiernie skomplikowana, ale do tego w zakresie komunikacji z bazą danych nienaturalna i nieefektywna. Należy tu wspomnieć, że wady EJB stały się bezpośrednią przyczyną opracowania najpoważniejszej spośród rozwiązań niepochodzących od firmy Sun technologii szkieletowej dla platformy Java (nie tylko Enterprise Edition) o nazwie Spring Framework [Spring].

Najnowsza, wersja Java Platform, Enterprise Edition, oznaczana skrótowo jako Java EE 5.0, wprowadza szereg istotnych, a w niektórych obszarach wręcz rewolucyjnych zmian, stanowiących odpowiedź na powszechnie wytykane wady wersji wcześniejszych. Najważniejszą ze zmian jest znaczące uproszczenie technologii Enterprise JavaBeans w ramach specyfikacji EJB 3.0 wraz z wyodrębnieniem z niej obsługi trwałości obiektów aplikacji (czyli komunikacji z bazą danych) w formie nowej specyfikacji o nazwie Java Persistence. O tym jak istotny przełom w rozwoju platformy Java EE stanowi pojawienie się EJB 3.0 wraz z Java Persistence świadczy fakt dostarczenia przez wielu producentów serwerów aplikacji Java EE wersji oprogramowania wspierających te dwie specyfikacje tak szybko jak było to możliwe, nawet bez pełnego wsparcia dla Java EE 5.0. Oracle Application Server wspiera EJB 3.0 i Java Persistence od wersji 10g R3 10.1.3.1, na szczeblu całej platformy Java EE deklarując zgodność z wersją J2EE 1.4. Pełne wsparcie dla Java EE 5.0 planowane jest w wersji 11g serwera aplikacji.

Celem niniejszego artykułu jest prezentacja technologii EJB i Java Persistence, a także ilustracja sposobu tworzenia aplikacji w tych technologiach w środowisku Oracle JDeveloper 10g 10.1.3.2.

2. EJB 3.0

Technologia EJB od początku historii Java Platform, Enterprise Edition stanowi ważny element tej platformy i była lansowana jako preferowana technologia do implementacji logiki biznesowej. Doświadczenie pokazało, że EJB nie są wymagane we wszystkich aplikacjach i z powodzeniem można implementować logikę biznesową w klasach towarzyszących serwletom i JSP. EJB są szczególnie przydatne gdy aplikacja musi być skalowalna, gdy realizuje zaawansowane przetwarzanie transakcyjne (np. transakcje rozproszone) lub gdy ma obsługiwać klientów różnych typów tj. umożliwiać zarówno dostęp z przeglądarki internetowej, jak i poprzez klienta aplikacyjnego.

Do wersji EJB 2.1 tworzenie komponentów EJB było skomplikowane, a aplikacje je wykorzystujące często były nieefektywne. Implementacja komponentu typowo obejmowała dwa interfejsy, klasę komponentu i deskryptor instalacji w formacie XML. Do tego, interfejsy i klasy komponentów musiały dziedziczyć z interfejsów i klas bibliotecznych EJB, co pociągało za sobą konieczność implementacji wielu niewykorzystywanych metod i obsługiwania wielu wyjątków. Największym problemem EJB do wersji 2.1 włącznie były jednak zdecydowanie komponenty encyjne, służące do komunikacji z bazą danych. Encyjne EJB były nienaturalne, nieprzenaszalne i nieefektywne.

Wraz z wersją EJB 3.0 nastąpiło znaczące uproszczenie technologii EJB. Wymagana jest mniejsza liczba plików źródłowych, a do tego są one „zwykłymi” interfejsami i klasami Java (tzw. POJI i POJO). Deskryptory XML nie są już obowiązkowe i zostały zastąpione przez adnotacje (ang. anno-

tations), które pojawiły się w wersji 1.5 (5.0) języka Java i stanowią podstawowy mechanizm konfiguracji aplikacji Java EE 5.0. Obiekty potrzebne do działania komponentu EJB (podobnie jak w wielu innych typach komponentów aplikacji Java EE 5.0) są wstrzykiwane mechanizmem dependency injection, zamiast wyszukiwania ich przez JNDI. Encyjne komponenty EJB zostały tak dalece uproszczone, że przestały być komponentami EJB i stanowią odrębny standard o nazwie Java Persistence, oparty o odwzorowanie obiektowo-relacyjne, omówiony w następnym rozdziale.

Wspomniane wyżej adnotacje są nowym elementem języka Java, wprowadzonym w J2SE 5.0 i mają postać metadanych o prostej składni, zagnieżdżanych w kodzie Java. Adnotacje są powszechnie uznawane za wygodniejszy i bardziej naturalny niż zewnętrzne pliki XML mechanizm konfiguracji aplikacji, ponieważ ustawienia konfiguracyjne w formie adnotacji bezpośrednio poprzedzają w kodzie klasy i metody, do których się odnoszą.

Wstrzykiwanie zależności (ang. dependency injection) pozwala na luźne wiązanie komponentów ze sobą, bez zaszywania w kodzie jawnych odwołań do innych obiektów, pozostawiając związanie współpracujących ze sobą komponentów aplikacji środowisku uruchomieniowemu, czyli w przypadku komponentów EJB – kontenerowi EJB. Wcześniej mechanizm wstrzykiwania zależności pojawił się w niemającym statusu standardu, wspomnianym we wstępie szkielecie aplikacji Spring Framework, przyczyniając się do jego sukcesu. Java EE 5 zaadaptowała ten wzorzec projektowy, pozwalając programistom korzystać z jego zalet bez konieczności uciekania się do niestandardowych rozwiązań. Do komponentów EJB wstrzykiwane mogą być np. zarządca encji (`EntityManager`) do komunikacji z bazą danych lub referencje do innych komponentów EJB.

W wersji 3.0 technologii EJB występują dwa główne typy komponentów: sesyjne i komunikatowe. Sesyjny komponent EJB (Session Bean) realizuje konkretne zadanie dla klienta i może być postrzegany jako logiczne rozszerzenie kodu aplikacji klienta umieszczone po stronie serwera aplikacji. Klient zleca komponentowi wykonanie zadania poprzez wywołanie metody na jego rzecz. Sesyjny komponent w danej chwili może mieć tylko jednego klienta i nie jest współdzielony (podobnie jak sesja dotyczy jednego użytkownika – stąd nazwa typu komponentu). Stan sesyjnego komponentu nie wykracza poza sesję i nie jest reprezentowany w sposób trwały np. w bazie danych. Komponenty sesyjne, podobnie jak w poprzednich wersjach EJB, można dalej podzielić na sesyjne stanowe i sesyjne bezstanowe.

Komunikatowy komponent EJB (Message-Driven Bean) jest asynchronicznym konsumentem komunikatów (wiadomości). Najczęściej komunikatowe EJB wykorzystują technologię Java Message Service (JMS) i nasłuchują nadejścia komunikatu JMS. Nadejście komunikatu inicjuje wywołanie metody komponentu. Klienci nie odwołują się do komunikatowych EJB bezpośrednio. Klient zleca wykonanie zadania poprzez wysłanie komunikatu do systemu komunikatów (np. do kolejki). System po nadejściu komunikatu przydziela do jego obsługi instancję komunikatowego EJB. Taka architektura ma na celu asynchroniczną obsługę żądania klienta. Klient wysyła żądanie w formie komunikatu i kontynuuje pracę, nie czekając na zakończenie realizacji zadania.

Uproszczenie sposobu implementacji komponentu w wersji EJB 3.0 dotyczy w szczególności komponentów sesyjnych. Kod źródłowy sesyjnego komponentu EJB 3.0 obejmuje:

- klasę komponentu, implementującą interfejs biznesowy komponentu i metody cyklu życia jeśli są wykorzystywane;
- interfejsy biznesowe, deklarujące metody udostępniane przez komponent klientom zdalnym i lokalnym, tworzone w formie zwykłego interfejsu Java opatrzonego adnotacjami;
- opcjonalnie klasy pomocnicze, wykorzystywane przez klasę komponentu.

Nie są już wymagane interfejsy `Home` i `LocalHome` wykorzystywane w EJB 2.1 do zarządzania cyklem życia komponentu. Klasa komponentu jest zwykłą klasą POJO i nie rozszerza żadnej klasy bibliotecznej, jak miało to miejsce w EJB 2.1. Dzięki temu, klasa komponentu EJB nie musi dostarczać implementacji wielu metod typu „callback”, które były wymagane w EJB 2.1, mimo że bardzo często miały puste ciała. Należy w tym momencie podkreślić, że w dalszym ciągu istnieje możliwość wskazania metod komponentu jako metod „callback”, które będą w odpowiednim momencie cyklu

życia wywoływane przez kontener, za pomocą adnotacji `@PostConstruct`, `@PreDestroy`, `@PrePassivate` i `@PostActivate`.

Rozważmy poniższy przykład kodu komponentu sesyjnego w wersji EJB 3.0, obejmującego interfejs biznesowy i klasę komponentu.

Konwerter.java

```
import javax.ejb.*;

@Remote
public interface Konwerter {
    public double fahrNaCels(double f);
    public double celsNaFahr(double c);
}
```

KonwerterBean.java

```
import javax.ejb.*;

@Stateless
public class KonwerterBean implements Konwerter {
    public double fahrNaCels(double f) {
        return (5.0 / 9.0) * (f - 32);
    }
    public double celsNaFahr(double c) {
        return (9.0 / 5.0) * c + 32;
    }
}
```

Powyższy kod definiuje sesyjny bezstanowy komponent EJB, służący do konwersji wartości temperatur między skalami Celsjusza i Fahrenheita. Plik `Konwerter.java` zawiera interfejs biznesowy, obejmujący dwie metody `fahrNaCels()` i `celsNaFahr()`. Interfejs biznesowy jest w tym wypadku interfejsem zdalnym, specyfikującym metody udostępniane przez komponent klientom zdalnym. Decyduje o tym adnotacja `@Remote`. Plik `KonwerterBean.java` zawiera klasę komponentu. Klasa implementuje interfejs biznesowy `Konwerter` i zawiera przewidziane w nim metody. O tym, że klasa jest klasą bezstanowego sesyjnego komponentu EJB świadczy adnotacja `@Stateless` (dla komponentów stanowych używana jest adnotacja `@Stateful`).

Gdy sesyjny komponent EJB jest osadzany (ang. *deployed*) w kontenerze EJB, jego interfejs biznesowy jest rejestrowany w rejestrze JNDI kontenera. Istnieją dwa sposoby uzyskania przez klienta referencji do komponentu EJB. Nowszym, dostępnym od EJB 3.0 sposobem jest wstrzyknięcie referencji mechanizmem *dependency injection*, ale możliwe jest również jawne wyszukanie komponentu w rejestrze JNDI operacją `lookup()`. Wg specyfikacji EJB 3.0, domyślnie komponent EJB jest rejestrowany w rejestrze JNDI pod nazwą będącą w pełni kwalifikowaną nazwą interfejsu biznesowego. W takim wypadku pole w klasie klienta, do którego wstrzykiwana jest referencja do komponentu, jest oznaczane adnotacją `@EJB` bez żadnych atrybutów. Gdy komponent został zarejestrowany w JNDI pod inną nazwą niż domyślna, należy w adnotacji `@EJB` podać tę nazwę poprzez atrybut `name`. Po uzyskaniu referencji do komponentu, klient korzysta z niego wywołując na rzecz referencji metody zawarte w interfejsie biznesowym. Poniższy fragment kodu ilustruje sposób wstrzyknięcia referencji do komponentu EJB w kodzie klienta typu *serwlet/JSP/EJB* (klient niebędący aplikacją *stand-alone*), przy założeniu, że wykorzystywany komponent EJB został zarejestrowany w JNDI pod domyślną nazwą.

```
@EJB
Konwerter konw;
...
double c = konw.fahrNaCels(35.5);
```

3. Java Persistence

Implementacja aplikacji Java pracujących na relacyjnej bazie danych na poziomie interfejsu JDBC jest czasochłonna i uciążliwa. Problem stanowi niski poziom abstrakcji interfejsu JDBC i różnice w organizacji danych między obiektowym językiem Java, a relacyjnymi bazami danych. Lansowana przez specyfikację Java EE do wersji 1.4 jako rozwiązanie tego problemu technologia encyjnych EJB okazała się nienaturalna i nieefektywna. Jako alternatywę, różne środowiska zaproponowały technologie automatyzujące odwzorowanie obiektów na poziomie programu Java w struktury relacyjne. Technologie te są określane jako technologie odwzorowania obiektowo-relacyjnego (Object-Relational Mapping – w skrócie O/RM). Można z nich korzystać również w celu uzyskania obiektowej reprezentacji danych dla istniejącego schematu relacyjnej bazy danych. Najpopularniejsze implementacje technologii odwzorowania obiektowo-relacyjnego dla aplikacji Java to Hibernate [Hibernate] (rozwiązanie Open Source firmy JBoss) i Oracle Toplink [Toplink] (rozwiązanie firmowe firmy Oracle). Mniejszą popularność zyskała technologia JDO [JDO] (firmy Sun).

Java Persistence to nowy, opracowany razem z EJB 3.0 standard zapewniania trwałości obiektów w aplikacjach Java EE i Java SE, stanowiący część specyfikacji Java EE od wersji 5.0. Został on opracowany razem z EJB 3.0 w odpowiedzi na niepowodzenie lansowanej do tej pory koncepcji encyjnych EJB i niewątpliwy sukces technologii odwzorowania obiektowo-relacyjnego takich jak Hibernate czy Oracle Toplink. Technologie te, mimo że oparte o te same idee, różnią się jeśli chodzi o API. Standard Java Persistence jest oparty o odwzorowanie obiektowo-relacyjne i definiuje standardowe API do obsługi trwałości obiektów. Hibernate i Oracle Toplink stanowią obecnie implementacje standardu Java Persistence, przy czym Toplink ma status implementacji referencyjnej.

Elementy standardu Java Persistence to:

- interfejs programistyczny Java Persistence API (JPA), obejmujący interfejs do zarządcy trwałości `EntityManager` (skrót JPA jest powszechnie używany jako skrócona nazwa standardu Java Persistence jako całości);
- język zapytań Java Persistence Query Language (JPQL), o składni przypominającej SQL, umożliwiający tworzenie przenaszalnych zapytań;
- metadane o odwzorowaniu obiektowo-relacyjnym, najczęściej umieszczone w kodzie w formie adnotacji, z możliwością dodatkowej konfiguracji w środowisku produkcyjnym poprzez XML-owe pliki konfiguracyjne.

Podstawowym pojęciem w Java Persistence jest encja (ang. entity). Encja to lekki obiekt służący do reprezentacji trwałych danych. Typowo, encja reprezentuje tabelę z relacyjnej bazy danych, ale istnieje również możliwość odwzorowania encji na kilka tabel. Encja definiowana jest w formie klasy encji. Niekiedy klasa encji jest uzupełniana o klasy pomocnicze np. klasę definiującą strukturę złożonego klucza głównego.

Klasa encji to zwykła klasa POJO (Plain Old Java Object), spełniająca reguły JavaBeans tj. dostęp do pól klasy tylko przez metody klasy `setXXX()/getXXX()` i bezargumentowy publiczny lub zabezpieczony konstruktor. Klasa encji nie może być `final`. Klasa encji nie musi dziedziczyć z żadnej konkretnej klasy ani implementować konkretnego interfejsu. W praktyce klasy encji są tworzone jako implementujące interfejs `Serializable`, gdyż jest to wymagane gdy obiekty klasy mają być odłączane od kontekstu trwałości np. gdy są parametrami metod zdalnego interfejsu EJB. Poniżej przedstawiono przykład definicji klasy encji do reprezentacji informacji o błędach.

Blad.java

```
@Entity
@Table(name="BLEDY")
public class Blad implements Serializable {
    @Id
    private Long id;
    private String kod;
    private String opis;
    public Blad() { }
    public Long getId() { return id; }
```

```
public void setId(Long id) { this.id = id; }
public String getKod() { return kod; }
public void setKod(String kod) { this.kod = kod; }
public String getOpis() { return opis; }
public void setOpis(String opis) { this.opis = opis; }
}
```

Aby klasa była klasą encji musi być oznaczona adnotacją `@Entity`. Domyślnie klasa jest odwzorowywana na tabelę o nazwie takiej samej jak nazwa klasy. Adnotacja `@Table` zmienia to odwzorowanie, wskazując jawnie nazwę tabeli. Jest to szczególnie przydatne gdy schemat bazy danych już istnieje. Pole `id` zostało wskazane adnotacją `@Id` jako klucz główny dla encji. Każda encja musi posiadać klucz główny. Gdy, tak jak w przykładzie, obejmuje on jedno pole standardowego typu języka Java, nie jest konieczne definiowanie klasy pomocniczej.

Cyklem życia encji zarządza tzw. zarządca encji (`EntityManager`). Zarządcy encji mogą być zarządzani przez kontener, co jest dostępne dla komponentów EJB i komponentów managed bean w JSF, lub zarządzani przez aplikację, co jest wykorzystywane w serwetach i aplikacjach Java SE. Zarządca encji zarządzany przez kontener jest wstrzykiwany do komponentu aplikacji adnotacją `@PersistenceContext`. Zarządca encji zarządzany przez aplikację jest tworzony i niszczone przez aplikację za pośrednictwem obiektu `EntityManagerFactory` wstrzykiwanego do komponentu adnotacją `@PersistenceUnit` lub tworzonego statyczną metodą `createEntityManagerFactory()` klasy `Persistence`. Poniżej przedstawiono przykład wstrzyknięcia zarządcy encji zarządzanego przez kontener.

```
@PersistenceContext
EntityManager em;
```

Zbiór klas encji zarządzanych przez `EntityManager` w aplikacji jest definiowany jako tzw. jednostka trwałości (`Persistence Unit`). Zbiór klas encji w ramach jednej jednostki trwałości reprezentuje dane z jednej bazy danych. Jednostka trwałości jest definiowana w pliku konfiguracyjnym `persistence.xml`. Jeden plik `persistence.xml` może zawierać definicje kilku jednostek trwałości. Każda jednostka trwałości musi posiadać nazwę unikalną w zasięgu widzialności jednostki trwałości. Nazwa ta wykorzystywana jest np. w adnotacji wstrzykującej obiekt `EntityManagerFactory` w przypadku gdy w pliku `persistence.xml` zdefiniowano więcej niż jedną jednostkę trwałości. Poniżej przedstawiono przykładową zawartość pliku `persistence.xml`.

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" ...>
  <persistence-unit name="AlbumyJPPU" transaction-type="JTA">
    <provider>
      oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider
    </provider>
    <class>encje.Blad</class>
    <jta-data-source>jdbc/sample</jta-data-source>
    <properties>
      <property name="toplink.ddl-generation"
        value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Plik ten zawiera definicję jednej jednostki trwałości o nazwie `AlbumyJPPU`, obejmującą jedną klasę: `encje.Blad`. Definicja jednostki trwałości wskazuje `JTA` jako typ transakcji, `Oracle Toplink` jako implementację zarządcy encji oraz `jdbc/sample` jako nazwę JNDI źródła danych. Ponadto, ustawiona została wartość jednej z właściwości dla `Oracle Toplink`, która określa, że gdy w momencie uruchomienia aplikacji nie będą w bazie danych istniały wymagane tabele, to zostaną one automatycznie utworzone przez `Toplink`.

Instancja encji wg standardu Java Persistence może znajdować się w jednym z czterech stanów:

- nowa (ang. new) – nieposiadająca trwałej tożsamości i niezwiązana jeszcze z kontekstem trwałości;
- zarządzana (ang. managed) - posiadająca trwałą tożsamość i związana z kontekstem trwałości;
- odłączona (ang. detached) - posiadająca trwałą tożsamość, a niezwiązana w danym momencie z kontekstem trwałości;
- usunięta (ang. removed) - posiadająca trwałą tożsamość, związana z kontekstem trwałości i zarezerwowana do usunięcia z bazy danych.

Pierwszy przedstawiony poniżej fragment kodu ilustruje utworzenie nowej instancji encji, a następnie związanie jej z kontekstem trwałości metodą `persist()` obiektu `EntityManager`, a drugi wyszukanie trwałej instancji poprzez klucz główny metodą `find()`, a następnie jej usunięcie metodą `remove()`.

```
@PersistenceContext
EntityManager em;
...
Blad b = new Blad();
b.setKod("b001");
b.setOpis("Niedozwolona operacja w module X");
em.persist(b);
```

```
@PersistenceContext
EntityManager em;
...
Blad b = em.find(Blad.class, new Long(13));
em.remove(b);
```

Zapytania do bazy danych w standardzie Java Persistence są reprezentowane przez obiekty Query tworzone metodami obiektu `EntityManager`. Standard przewiduje trzy rodzaje zapytań: dynamiczne w języku JPQL (Java Persistence Query Language), dynamiczne natywne i nazwane (w JPQL lub natywne). Zapytania nazwane mają taką przewagę nad dynamicznymi, że mogą być pre-kompilowane i lepiej optymalizowane, a przez to efektywniejsze. Poniżej przedstawiono przykład definicji sparаметryzowanego zapytania nazwanego w klasie encji za pomocą adnotacji `NamedQuery`, a następnie przykład wykonania tego zapytania. Wykonanie zapytania nazwanego obejmuje następujące kroki: utworzenie obiektu zapytania ze wskazaniem go poprzez nazwę, ustalenie wartości ewentualnych parametrów i pobranie kolekcji wyników zapytania.

Blad.java

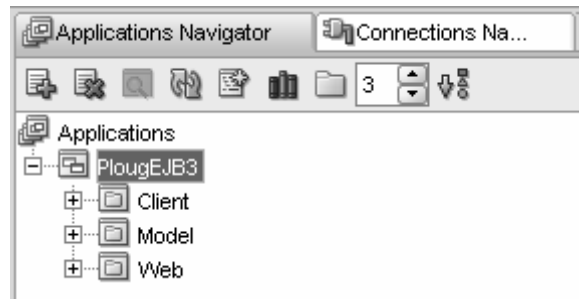
```
@Entity
@Table(name="BLEDY")
@NamedQuery(name = "findByKeyword",
            query = "SELECT b FROM Blad b WHERE b.opis LIKE :keyword")
public class Blad implements Serializable {...}
}
```

```
@PersistenceContext
EntityManager em;
...
List<Blad> wyn = null;
wyn = em.createNamedQuery("findByKeyword")
        .setParameter("keyword", "%krytyczny%")
        .getResultList();
```

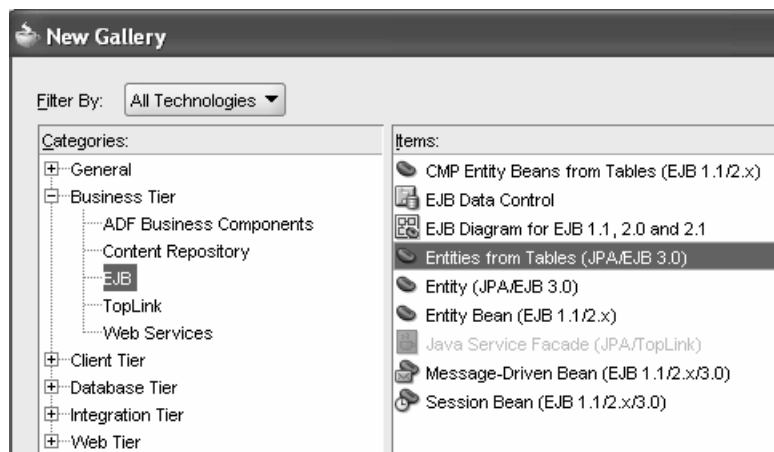
4. Tworzenie aplikacji opartych o EJB 3.0 w środowisku Oracle JDeveloper 10g

Pierwszą wersją JDevelopera wspierającą EJB 3.0 i JPA była wersja 10.1.3.1. Należy jednak zwrócić uwagę, że początkowo wsparcie to zostało zaoferowane dla roboczej wersji specyfikacji EJB 3.0, przez co kreator posługiwał się jeszcze pojęciem „Entity EJB 3.0”, a plik persistence.xml nie był używany. Poniższy przykład tworzenia aplikacji dotyczy wersji 10.1.3.2 JDevelopera, oferującej wsparcie dla ostatecznej wersji specyfikacji EJB 3.0 i JPA i posługującej się właściwym nazewnictwem.

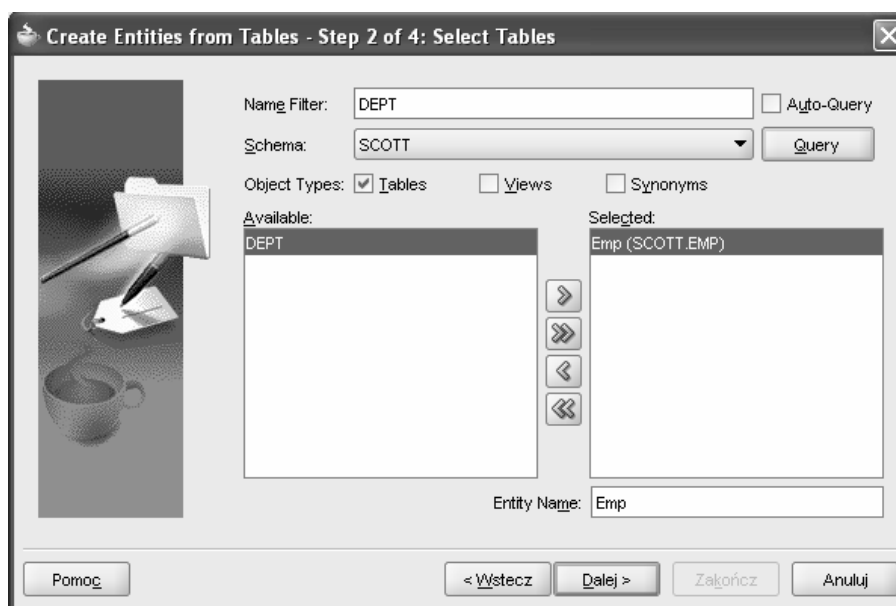
Przykładowa aplikacja będzie umożliwiała wyświetlenie w przeglądarce internetowej tabelki z informacjami o pracownikach pobieranych z tabeli w bazie danych Oracle (tabeli EMP). Aplikacja będzie podzielona na 3 projekty: Model – dla komponentów EJB, Web – dla warstwy interfejsu użytkownika opartego o ADF Faces (podstawowa obecnie technologia tworzenia warstwy widoku aplikacji webowych w JDeveloper) oraz Client – dla konsolowego klienta EJB umożliwiającego przetestowanie komponentów EJB.



Aby utworzyć encje JPA na podstawie tabel z bazy danych uruchomimy z poziomu projektu Model odpowiedni kreator:



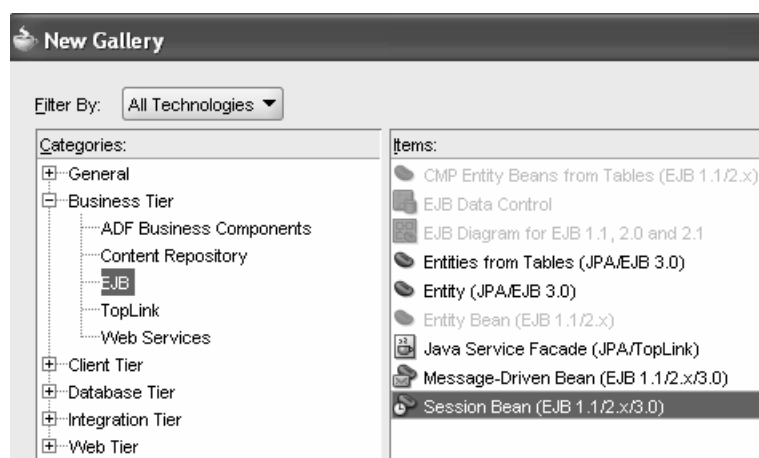
W kreatorze „Entities from Tables (JPA/EJB 3.0)” wybierzemy jedynie tabelę EMP (w pozostałych krokach kreatora pozostawiając wartości domyślne):

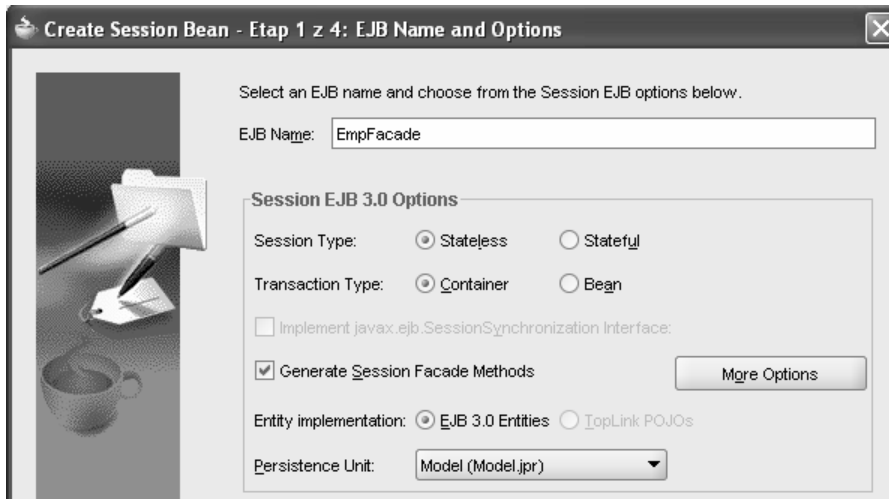


Wynikiem działania kreatora będzie klasa encji Emp zawierająca nazwane zapytanie wyszukiujące wszystkie wystąpienia encji:

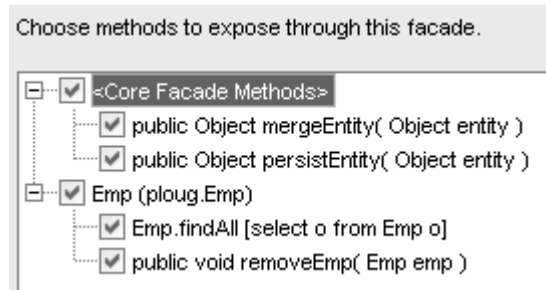
```
@Entity
@NamedQuery(name = "Emp.findAll", query = "select o from Emp o")
public class Emp implements Serializable {
    @Id
    @Column(nullable = false)
    private Long empno;
    private String ename;
    private Timestamp hiredate;
    private String job;
    ...
}
```

W kolejnym kroku utworzymy sesyjny bezstanowy komponent EJB 3.0 pełniący funkcję fasady buforującej klientów od warstwy biznesowej reprezentowanej przez encje. Podejście to uniezależnia klienta od modelu danych i może zredukować ruch w sieci w przypadku zdalnych klientów warstwy usług biznesowych:

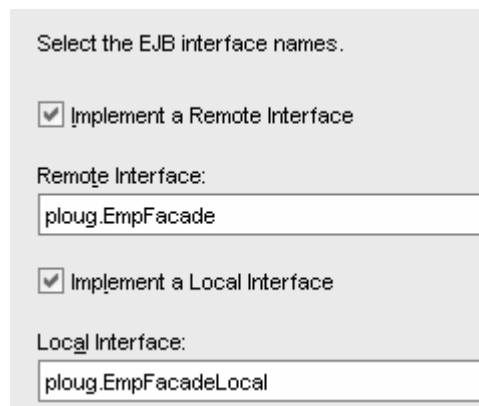




W kolejnym ekranie kreatora EJB zlecimy udostępnienie wszystkich metod JPA poprzez fasadę:



Na zakończenie zlecimy utworzenie dla komponentu sesyjnego utworzenie zarówno interfejsu zdalnego (z myślą o testowym kliencie konsolowym) jak i lokalnego (z myślą o aplikacji webowej):

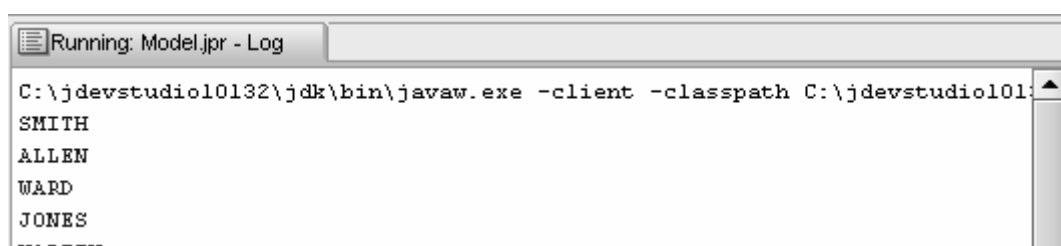


Następnie wygenerujemy testowego klienta konsolowego aby sprawdzić funkcjonowanie warstwy EJB/JPA, wybierając z menu kontekstowego dla węzła komponentu fasadowego w nawigatory aplikacji opcję „New Sample Java Client”. Jako docelowy projekt w oknie dialogowym wskażemy projekt Client. W wygenerowanej klasie klienta zmienimy fragment kodu realizujący wywołanie metody komponentu fasadowego, tak aby aplikacja wyświetlała nazwiska wszystkich pracowników:

```
public class EmpFacadeClient {
    public static void main(String [] args) {
```

```
try {
    final Context context = getInitialContext();
    EmpFacade empFacade = (EmpFacade) context.lookup("EmpFacade");
    for (Emp e : empFacade.queryEmpFindAll())
        System.out.println(e.getEname());
} catch (Exception ex) {
    ex.printStackTrace();
}
}
```

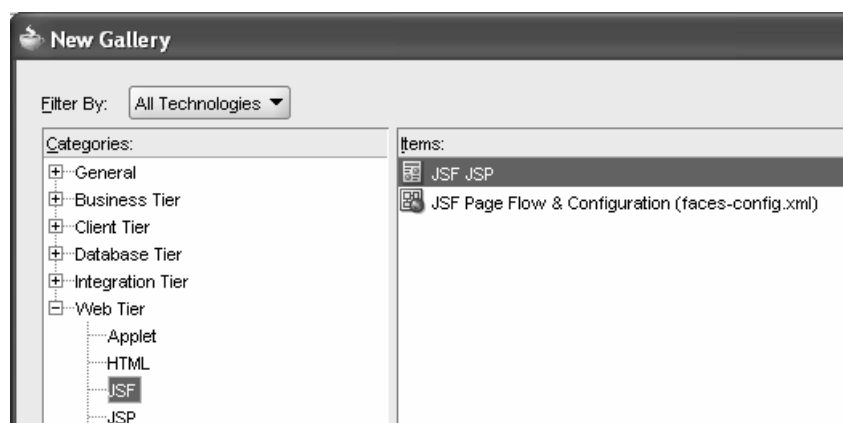
Aby przetestować aplikację, należy najpierw uruchomić komponent fasadowy, co spowoduje osadzenie komponentów EJB/JPA projektu na wbudowanym serwerze OC4J, a następnie uruchomić klienta konsolowego:



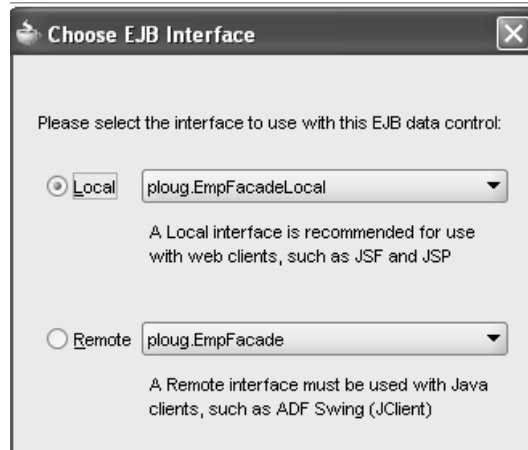
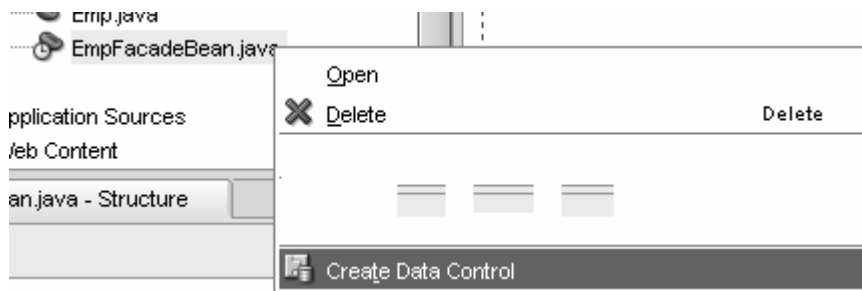
```
Running: Model.jpr - Log
C:\jdevstudio10132\jdk\bin\javaw.exe -client -classpath C:\jdevstudio10132\j2ee\server\lib\oc4j.jar
SMITH
ALLEN
WARD
JONES
MARTIN
```

Po pozytywnym zweryfikowaniu poprawności działania warstwy EJB/JPA aplikacji można przystąpić do implementacji interfejsu użytkownika w formie aplikacji webowej. Wykorzystamy do tego celu technologię ADF Faces stanowiącą obecnie podstawową technologię tworzenia warstwy widoku w ramach Oracle Application Development Framework [ADF].

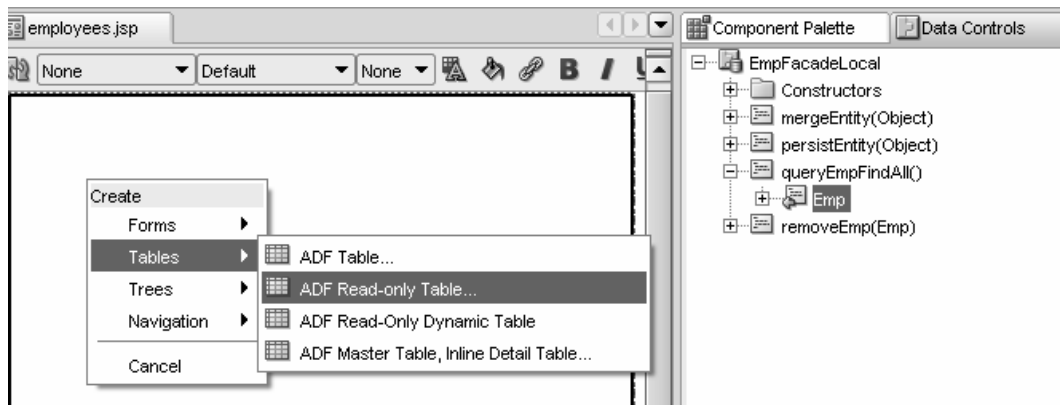
W projekcie Web utworzymy kreatorem nową stronę JSF, pozostawiając domyślne wartości wszystkich parametrów kreatora poza nazwą pliku, którą zmienimy na employees.jsp:

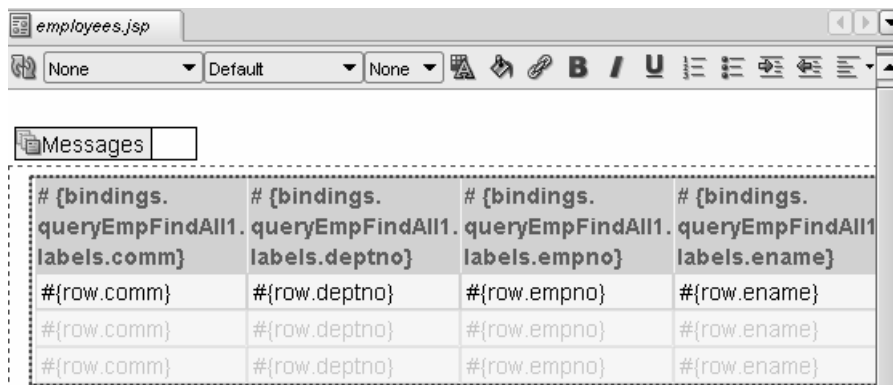


Po utworzeniu pustej strony JSF powrócimy na chwilę do projektu Model aby udostępnić komponent EJB pełniący funkcję fasady poprzez kontrolkę danych ADF, wskazując że kontrolka ma wykorzystywać lokalny interfejs komponentu EJB:

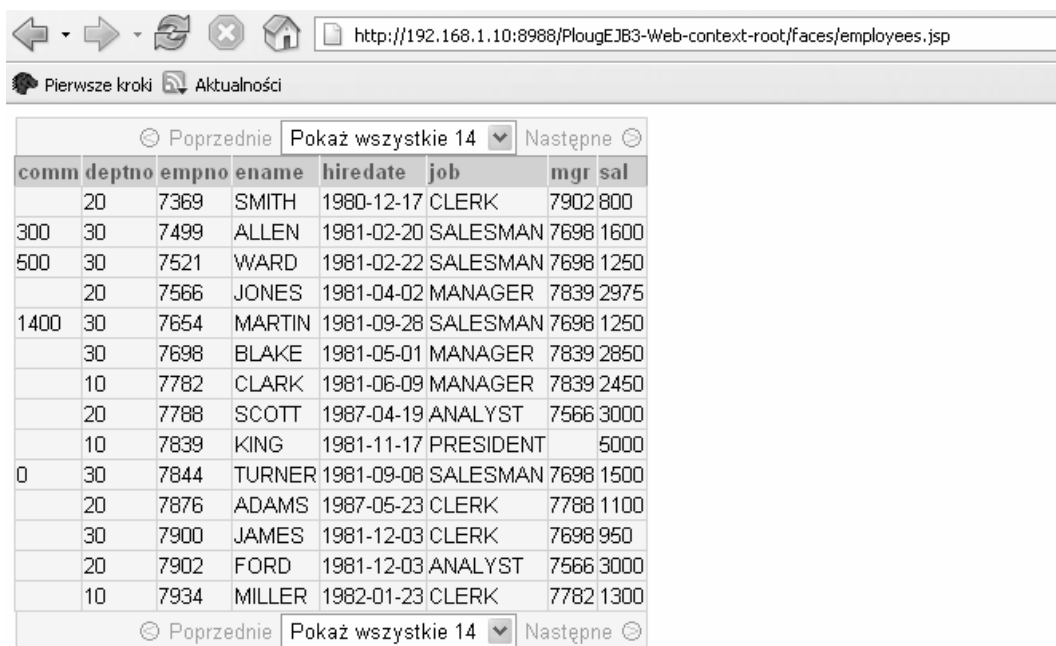


Po powrocie do edycji strony JSF, z palety Data Controls techniką drag-and-drop za pomocą prawego klawisza myszy umieścimy na stronie dostawcę danych reprezentującego wynik metody fasady zwracającej kolekcję wszystkich pracowników. Jako typ umieszczanego na stronie elementu interfejsu wybierzemy z pojawiającego się po upuszczeniu kontrolki menu kontekstowego opcję „ADF Read-only Table”.





Powyższy etap kończy proces tworzenia naszej prostej aplikacji. Aplikacja jest gotowa do uruchomienia. Po uruchomieniu strony JSF, nastąpi jej osadzenie na wbudowanym serwerze OC4J i automatyczne otwarcie okna przeglądarki z adresem URL wywołującym stronę. Możliwe jest oczywiście przygotowanie w środowisku JDeveloper archiwum EAR z aplikacją i jej instalacja na zewnętrznym kontenerze OC4J lub serwerze aplikacji Oracle Application Server 10g R3. Możliwe jest również zainstalowanie aplikacji na innym serwerze aplikacji wspierającym JSF, EJB 3.0 i JPA, pod warunkiem dostarczenia z aplikacją bibliotek ADF.



5. Podsumowanie

Technologia EJB 3.0 wraz z Java Persistence to najbardziej wyczekiwany składnik Java EE 5.0. Stanowi ona znaczące uproszczenie w stosunku do poprzednich wersji, jednocześnie opierając się o powszechnie zaakceptowaną koncepcję odwzorowania obiektowo-relacyjnego w zakresie komunikacji z bazą danych.

Firma Oracle brała aktywny udział w pracach nad EJB 3.0 i Java Persistence, co zaowocowało nadaniem Oracle Toplink statusu referencyjnej implementacji Java Persistence. Doceniając znaczenie praktyczne nowej wersji EJB, Oracle zaoferował wsparcie dla EJB 3.0 i JPA jako wybranych specyfikacji Java EE 5.0 w wersji R3 10.1.3.1 serwera Oracle Application Server 10g, generalnie zgodnego z J2EE 1.4, i w odpowiadających mu wersjach środowiska JDeveloper 10g.

Pełną zgodność z Java EE 5.0 będzie oferował Oracle Application Server 11g. Obecnie (maj 2007) dostępne są już wersje Technology Preview w pełni zgodnego z Java EE 5.0 kontenera Java EE OC4J 11g (11.1.1) oraz umożliwiającego tworzenie dla niego aplikacji środowiska JDeveloper 11g (11.1.1). JDeveloper 11g rozszerza wsparcie dla tworzenia aplikacji opartych o EJB oferując szereg drobnych ułatwień oraz możliwość reprezentacji komponentów EJB 3.0 na diagramach (JDeveloper 10g oferuje diagramy EJB tylko dla wersji 2.1 i wcześniejszych).

Bibliografia

5. [ADF] Oracle Application Development Framework, <http://www.oracle.com/technology/products/adf/>
6. [Hibernate] Hibernate, <http://www.hibernate.org/>
7. [JavaEE] Java EE At a Glance, <http://java.sun.com/javaee/>
8. [JavaEE5Tut] The Java EE 5 Tutorial, <http://java.sun.com/javaee/5/docs/tutorial/doc/>
9. [JDO] Java Data Objects (JDO), <http://java.sun.com/products/jdo/>
10. [SCS06] Stearns J., Roberto Chinnici R., Sahoo: Update: An Introduction to the Java EE 5 Platform, http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/, May 2006
11. [Spring] Spring Framework, <http://www.springframework.org/>
12. [Toplink] Oracle Toplink, <http://www.oracle.com/technology/products/ias/toplink/>