

XII Konferencja PLOUG  
Zakopane  
Październik 2006

# Co nowego w Java EE?

Marek Wojciechowski

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

## **Abstrakt**

Najnowsza wersja specyfikacji Java Platform Enterprise Edition wprowadza szereg istotnych zmian wpływających na sposób tworzenia aplikacji w tej technologii. W artykule zostaną przedstawione i zilustrowane przykładami najistotniejsze zmiany i nowe elementy specyfikacji, ze szczególnym uwzględnieniem EJB 3.0 i nowego standardu Java Persistence.



## 1. Wprowadzenie

Java Platform, Enterprise Edition to obok Microsoft .NET najpoważniejsza platforma do tworzenia nowoczesnych, wielowarstwowych aplikacji klasy „enterprise”, wymagających efektywności, skalowalności i bezpieczeństwa. W przeciwieństwie do Microsoft .NET, Java Platform, Enterprise Edition 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. Obszarem, w którym Java Platform, Enterprise Edition dotychczas zdecydowanie przegrywała z Microsoft .NET była łatwość tworzenia aplikacji. Po pierwsze, Java Platform, Enterprise Edition długo nie mogła doznać się technologii upraszczającej i porządkującej sposób implementacji interfejsu użytkownika. Po drugie, lansowana od początku jako podstawowa technologia implementacji logiki biznesowej na tej platformie technologia Enterprise JavaBeans (EJB) była nadmiernie skomplikowana, a do tego w zakresie komunikacji z bazą danych nienaturalna i nieefektywna.

Najnowsza, oznaczona numerem „5” wersja Java Platform, Enterprise Edition wprowadza szereg istotnych, a w niektórych obszarach wręcz rewolucyjnych zmian, stanowiących odpowiedź na powszechnie wytykane wady wersji wcześniejszych. Oczywiście pierwszymi, rzucającymi się w oczy zmianami są nowy skrót nazwy platformy i nowa strategia numerowania jej wersji. Z dotychczasowej nazwy („Java 2 Platform, Enterprise Edition”) wyrzucono dwójkę, co pociągnęło za sobą zmianę skrótu „J2EE” na „Java EE”. Zdecydowano się na skrót „Java EE”, a nie „JEE”, aby podkreślić ścisły związek platformy z językiem Java. Nowa wersja została oznaczona numerem „5.0”, co oznacza zmianę sposobu numerowania wersji po poprzedniej J2EE 1.4. Wcześniej system numerowania wersji został zmieniony dla Java Platform, Standard Edition, której następnie również zmieniono skrót nazwy (na „Java SE”). Dywagacje na temat nazwy platformy i jej skrótu mają oczywiście dla twórców aplikacji znikome znaczenie. Istotne są zmiany „merytoryczne”, a tych w wersji 5.0 nie brakuje.

Celem niniejszego artykułu jest dokonanie przeglądu zmian i nowych elementów w wersji 5.0 platformy Java EE. Ponadto, szczegółowo omówione i zilustrowane przykładami będą technologie EJB i Java Persistence. EJB to technologia składowa Java EE, która została poddana najbardziej znaczącym zmianom, a Java Persistence to nowy, wyodrębniony z EJB standard, stanowiący przełom w dostępie aplikacji Java EE do baz danych.

## 2. Przegląd zmian i nowości w Java EE 5

Podstawowym celem Java EE 5 było ułatwienie tworzenia złożonych aplikacji korporacyjnych w języku Java i zwiększenie wydajności programistów. Cele te osiągnięto głównie poprzez dopracowanie i często uproszczenie technologii składowych dostępnych już w J2EE 1.4 oraz ich lepszą integrację. Najważniejsze zmiany w Java EE 5 w porównaniu z J2EE 1.4 to [SCS06]:

- uczynienie większości XML-owych plików konfiguracyjnych, takich jak np. `ejb-jar.xml`, opcjonalnymi;
- więcej sensownych wartości domyślnych dla opcji konfiguracyjnych aplikacji;
- wstrzykiwanie zależności w celu umożliwienia dostępu do zasobów;
- uproszczenie implementacji Web Services i wsparcie dla większej liczby standardów z nimi związanych;

- lepsza integracja technologii JavaServer Faces (JSF) z JSP i JSTL;
- znaczące uproszczenie technologii EJB;
- nowy standard dostępu do bazy danych o nazwie Java Persistence, zastępujący encyjne komponenty EJB, oparty o odwzorowanie obiektowo-relacyjne.

Deskryptory instalacji (ang. deployment descriptors) w formie plików XML do wersji J2EE 1.4 miały kluczowe znaczenie, stanowiąc miejsce konfiguracji aplikacji i jej komponentów oraz integracji poszczególnych komponentów ze sobą. W Java EE 5 deskryptory te stały się opcjonalne, gdyż podstawowy mechanizm konfiguracji aplikacji stanowią adnotacje (ang. annotations). 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. Adnotacje na platformie Java EE 5 są wykorzystywane m.in. do:

- definiowania Web Services;
- definiowania komponentów EJB;
- opisu sposobu odwzorowania klas Java na tabele w bazie danych;
- wstrzykiwania referencji do wykorzystywanych komponentów oraz różnych zasobów;
- konfiguracji ustawień bezpieczeństwa aplikacji.

Konfiguracja aplikacji została uproszczona nie tylko ze względu na możliwość zastąpienia plików XML adnotacjami zagnieżdżonymi w kodzie Java, ale również ze względu na większą „domyślność” serwera. Dla większości opcji konfiguracyjnych aplikacji przyjmowane są wartości domyślne. Uproszczeniu uległ również sposób „pakowania” aplikacji do dystrybucji i instalacji na serwerze, gdyż nie jest już wymagany plik konfiguracyjny aplikacji `application.xml`. Funkcją tego pliku było wskazanie modułów składowych aplikacji i ich roli. W wersji Java EE 5 serwer sam odnajdzie moduły składowe zawarte w archiwum instalacyjnym aplikacji i domyśli się ich roli w oparciu o rozszerzenia nazw plików i zawartość archiwów poszczególnych modułów.

Powszechnie wykorzystywaną techniką w różnego typu komponentach jest w wersji Java EE 5 wstrzykiwanie zależności (ang. dependency injection). Mechanizm ten umożliwia komponentowi aplikacji dostęp do wymaganych przez niego zasobów, stanowiąc alternatywę dla jawnego wyszukania zasobu poprzez interfejs JNDI. Wcześniej mechanizm wstrzykiwania zależności pojawił się w niemającym statusu standardu szkielecie aplikacji Spring Framework [Spring], 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. Wstrzykiwanie zależności pozwala na luźne wiązanie komponentów ze sobą, bez zasywania w kodzie jawnych odwołań do innych obiektów, pozostawiając związanie współpracujących ze sobą komponentów aplikacji środowisku uruchomieniowemu, czyli w przypadku Java EE – kontenerom serwletów i EJB w ramach serwera aplikacji. Wstrzykiwanie zależności w aplikacjach Java EE 5 jest możliwe m.in. w komponentach EJB, serwletach i zarządzanych komponentach JavaBean wykorzystywanych przez JavaServer Faces. Wstrzykiwane mogą być m.in. takie zasoby jak: źródła danych, obiekt `UserTransaction`, zarządca encji (`EntityManager`) do komunikacji z bazą danych, kolejki wiadomości, referencje do komponentów EJB i Web Services.

Tworzenie Web Services zostało uproszczone dzięki opracowaniu nowego interfejsu programistycznego JAX-WS. JAX-WS z jednej strony upraszcza tworzenie Web Services sprowadzając przekształcenie klasy Java w `WebService` do umieszczenia adnotacji `@WebService` w kodzie

klasy, a z drugiej strony oferuje nowe możliwości takie jak np. możliwość asynchronicznej komunikacji klienta z komponentem Web Service.

Technologia JavaServer Faces (JSF), która pojawiła się w J2EE 1.4 doczekała się w Java EE 5 drobnych udoskonaleń np. w zakresie zachowywania stanu komponentów interfejsu użytkownika oraz została w pełni zintegrowana z JavaServer Pages (JSP) i biblioteką JSTL. Technologia JSF od początku była opracowywana z założeniem, że najczęściej będzie wykorzystywana w połączeniu z JSP. Jednakże pierwsza wersja JSP nie wykorzystywała w pełni możliwości JSP 2.0, zakładając jedynie mechanizmy dostępne już w JSP 1.2. W szczególności, dla JSF opracowano własny język wyrażeń, niezależny od języka wyrażeń dla JSP 2.0, choć identyczny z nim składniowo. W Java EE 5 języki wyrażeń JSP i JSF zostały zunifikowane w jeden język o nazwie Unified Expression Language, z zachowaniem różnic w notacji i znaczeniu wyrażeń  $\${...}$  i  $\#{...}$ . Notacja  $\${...}$ , wykorzystywana w JSP, oznacza wyrażenia wartościowane natychmiast, umożliwiające jedynie odczyt właściwości obiektów. Notacja  $\#{...}$ , wykorzystywana w JSF, oznacza wyrażenia wartościowane z opóźnieniem, umożliwiające również ustawianie właściwości obiektów i wywoływanie na ich rzecz metod. Integracja języków wyrażeń JSP i JSF nie stanowi jedynie „ideologicznej nadbudowy”. Przykładowo, dzięki niej możliwe są odwołania do komponentów JSF z poziomu znacznika `forEach` z biblioteki JSTL.

Z pewnością największe zmiany nastąpiły w Java EE 5 w technologii Enterprise JavaBeans (EJB). Technologia ta została znacząco uproszczona, a ponadto zarzucone zostały encyjne komponenty EJB i zastąpione przez nowy standard Java Persistence. Technologiom tym będą poświęcone dwa kolejne rozdziały.

### 3. 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 obsługiwanie 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, które pojawiły się w wersji 1.5 (5.0) języka Java. Obiekty potrzebne do działania komponentu 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.

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 instalowany (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 serwet/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);
```

## 4. 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 encyjnnych 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.

Elementy standardu Java Persistence to:

- interfejs programistyczny Java Persistence API, obejmujący interfejs do zarządcy trwałości `EntityManager`;
- 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.

Cykiem 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>
      ora-
cle.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 zaszeregowana 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 prekompilowane i lepiej optymalizowane, a przez to efektywniejsze. Poniżej przedstawiono przykład definicji sparametryzowanego 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, ustawienie 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();
```

## **5. Podsumowanie**

Najnowsza wersja platformy Java Platform, Enterprise Edition: Java EE 5 wprowadza szereg zmian i nowości w porównaniu z wcześniejszą wersją oznaczaną jako J2EE 1.4. Część zmian ma charakter ewolucyjny i polega na udoskonaleniu dotychczasowych rozwiązań i lepszej ich integracji. Fundamentalne, a może nawet rewolucyjne zmiany to (1) wprowadzenie adnotacji jako podstawowego sposobu konfiguracji aplikacji i jej komponentów, co w wielu przypadkach pozwala na wyeliminowanie XML-owych deskryptorów instalacji, (2) przyjęcie wstrzykiwania zależności jako mechanizmu dostępu do zasobów oraz (3) znaczące uproszczenie technologii EJB, z wyodrębnieniem z niej standardu Java Persistence dotyczącego komunikacji z bazą danych, opartego o koncepcje odwzorowania obiektowo-relacyjnego.

### **Bibliografia**

- [Hibernate] Hibernate, <http://www.hibernate.org/>
- [JavaEE] Java EE At a Glance, <http://java.sun.com/javae/>
- [JavaEE5Tut] The Java EE 5 Tutorial, <http://java.sun.com/javae/5/docs/tutorial/doc/>
- [JDO] Java Data Objects (JDO), <http://java.sun.com/products/jdo/>
- [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/](http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/), May 2006
- [Spring] Spring Framework, <http://www.springframework.org/>
- [Toplink] Oracle Toplink, <http://www.oracle.com/technology/products/ias/toplink/>

