WI

# Programowanie komponentowe

## Enterprise JavaBeans

Piotr Błaszyński

Wydział Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego

5 czerwca 2019

# Wprowadzenie



- Enterprise JavaBeans to podstawowa technologia komponentowa platformy Java Enterprise Edition.
- EJB jest infrastrukturą systemów korporacyjnych zaprojektowaną z myślą o zapewnieniu programistom mechanizmów automatycznego zarządzania wieloma usługami, które mają kluczowe znaczenie dla funkcjonowania ich aplikacji.
- Podstawowym elementem tej architektury jest kontener EJB, czyli bezpośrednie środowisko komponentów Enterprise JavaBeans i dostawca zarządzanych usług.
- W specyfikacji Enterprise JavaBeans 3.0 unowocześniono technologię EJB i uproszczono zadania realizowane przez programistów.
- Aktualna wersja - 3.2 (2014 r.). Następca – Spring.

# Wprowadzenie - ogólne



- Model komponentów po stronie serwera może nam pomóc zdefiniować architekturę dla rozproszonych obiektów biznesowych, które łączą w sobie dostępność rozproszonych systemów obiektowych z płynnością logiki biznesowej implementowanej w formie hermetycznie zamkniętych obiektów.
- Modele komponentów serwera są wykorzystywane na poziomie serwerów aplikacji (warstwa pośrednia).
- Warstwa pośrednia odpowiada za zarządzanie komponentami w czasie wykonywania i udostępnianie ich zdalnym klientom.
- Podstawowa funkcjonalność tych modeli komponentowych ułatwia wytwarzanie rozproszonych obiektów biznesowych łączonych w ramach szerszych rozwiązań biznesowych.

# Wprowadzenie - ogólne



- Model komponentów po stronie serwera może nam pomóc zdefiniować architekturę dla rozproszonych obiektów biznesowych, które łączą w sobie dostępność rozproszonych systemów obiektowych z płynnością logiki biznesowej implementowanej w formie hermetycznie zamkniętych obiektów.
- Modele komponentów serwera są wykorzystywane na poziomie serwerów aplikacji (warstwa pośrednia).
- Warstwa pośrednia odpowiada za zarządzanie komponentami w czasie wykonywania i udostępnianie ich zdalnym klientom.
- Podstawowa funkcjonalność tych modeli komponentowych ułatwia wytwarzanie rozproszonych obiektów biznesowych łączonych w ramach szerszych rozwiązań biznesowych.

# Wprowadzenie - definicja



Definicja architektury Enterprise JavaBeans według Sun Microsystems:

- Enterprise JavaBeans jest architekturą komponentową stworzoną z myślą o wytwarzaniu i wdrażaniu rozproszonych aplikacji biznesowych złożonych z komponentów. Aplikacje tworzone z wykorzystaniem architektury Enterprise JavaBeans są skalowalne i transakcyjne, a także oferują bezpieczeństwo w środowiskach wieloużytkownikowych. Raz napisana aplikacja może być wdrażana na wielu platformach serwerów obsługujących specyfikację Enterprise JavaBeans. [Enterprise JavaBeans Specification, v3.0, Copyright© 2002 Sun Microsystems, Inc.]

# JavaBeans - co to jest?



JavaBeans to pewien standard pisania klas, opierający się na 3 prostych cechach:

1. wszystkie właściwości są prywatne (używanie getters/setters),
2. klasa posiada publiczny konstruktor bezargumentowy,
3. klasa powinna (nie musi!) implementować interfejs Serializable.

# JavaBeans - inne definicje



POJOs i POJIs:

- Plain Old Java Objects.
- Plain Old Java Interfaces.

# Java Beans - przechowywanie danych



- Utrwalanie danych można traktować jako abstrakcję bezpośrednio nad interfejsem JDBC.
- Warstwa utrwalania danych odpowiada za takie odwzorowywanie obiektów w bazie danych, które umożliwia ich przeszukiwanie, odczytywanie, aktualizowanie i usuwanie bez konieczności korzystania z konkretnego API, np. JDBC.
- Począwszy od wersji 3.0, abstrakcję dostępu do danych przeniesiono do specyfikacji nazwanej Java Persistence API.

# JavaBeans - przechowywanie danych



- Interfejs Java Persistence API definiuje sposób odwzorowywania w bazie danych zwykłych, „tradycyjnych" obiektów Java nazywanych obiektami POJO.
- Te tradycyjne, zwykłe obiekty Javy są nazywane komponentami encyjnymi (ang. entity beans).
- Komponenty encyjne różnią się od pozostałych klas Javy tylko tym, że są odwzorowywane na reprezentację w bazie danych z wykorzystaniem metadanych Java Persistence.
- Oznacza to, że można je zapisywać i odczytywać z bazy danych bez konieczności umieszczania w kodzie źródłowym konstrukcji nawiązujących połączenie za pośrednictwem interfejsu JDBC ani przeszukujących zwracane zbiory wynikowe.
- Interfejs Java Persistence API definiuje też język zapytań, który został stworzony przede wszystkim z myślą o obiektach Javy.

# JavaEE - inne możliwości



- Oprócz obsługi rozproszonych obiektów biznesowych zgodnych ze standardem RMI technologia Enterprise JavaBeans obsługuje także asynchroniczne przesyłanie komunikatów.
- W aktualnej wersji istnieją możliwości budowy komponentów sterowanych komunikatami nie ograniczane tylko do standardu JMS (Java Message Service) - możliwa jest współpraca z dowolnym systemem przesyłania komunikatów (np. SMTP, SNMP, Jabber itp.).
- Specyfikacja EJB 3.0 dopuszcza możliwość udostępniania komponentów biznesowych w formie Web Services i tym samym umożliwienie wywoływania tych komponentów zarówno przez pozostałe aplikacje JavaEE, jak i przez aplikacje napisane w innych językach programowania.

# Komponenty encyjne



- Komponenty encyjne (ang. entity beans) budowane według specyfikacji Java Persistence 1.0 są dostępne wyłącznie w formie zwykłych, „tradycyjnych” obiektów Javy (POJO) i mogą być odwzorowywane w tabelach relacyjnej bazy danych. W przeciwieństwie do pozostałych typów komponentów EJB komponenty encyjne mogą być serializowane i przesyłane za pośrednictwem sieci tak jak wszystkie inne obiekty POJO.
- Komponenty encyjne modelują te elementy (również działania) biznesowe, które można wyrazić w formie rzeczowników. Przykładowo pojedynczy komponent encyjny może reprezentować klienta, element wyposażenia lub nawet miejsce.
- Czyli komponenty encyjne modelują obiekty świata rzeczywistego, które przeważnie są utrwalane w formie rekordów relacyjnej bazy danych.

# Komponenty encyjne - przykład



```
import javax.persistence.*;
@Entity
@Table(name="CAR")
public class Car {
        private int id;
        private String brand;
        private String vin;

        @Id
        @GeneratedValue
        @Column(name="CAR_ID")
        public int getId( ) { return id; }
        public void setId(int pk) { this.id = pk; }
```

# Komponenty encyjne - przykład, cd.



```
        @Column(name="BRAND")
        public String getBrand( ) { return brand; }
        public void setBrand(String brand) { this.
            brand = brand; }

        @Column(name="VIN")
        public String getVin( ) { return vin; }
        public void setVin(String vin) { this.vin =
             vin; }
}
```

# Komponenty encyjne - omówienie



- Interakcja z komponentami encyjnymi odbywa się z wykorzystaniem usługi EntityManager.
- Do zdefiniowania komponentu encyjnego, wystarczające jest stworzenie klasy komponentu z odpowiednią adnotacją (Entity).
- Specyfikacja Java Persistence przewiduje, że komponenty encyjne są konkretnymi (nie jak wcześniej abstrakcyjnymi) obiektami, które w dodatku nie muszą implementować żadnego interfejsu technologii Enterprise JavaBeans.

# Komponenty encyjne - omówienie



- Komponenty encyjne zgodne ze specyfikacją Java Persistence zawierają pola reprezentujące stan oraz metody do ustawiania (*setters*) i zwracania wartości (*getters*) odpowiedzialne za zapewnianie dostępu do tych pól.
- Klasy komponentów oznacza się adnotacją @javax.persistence.Entity.

# Komponenty encyjne - omówienie



- Specyfikacja Java Persistence definiuje kompletny mechanizm odwzorowań obiektowo relacyjnych (ORM) w formie adnotacji, które mogą być stosowane przez programistów w kodzie komponentów encyjnych.
- Adnotacja @javax.persistence.Table służy do oznaczania tabel bazy danych, w których dana encja ma być odwzorowywana.
- Adnotację @javax.persistence.Column stosuje się do oznaczania metod zwracających wartości właściwości danego komponentu encyjnego i definiuje kolumnę tabeli relacyjnej bazy danych, w której dana właściwość ma być odwzorowywana.

# Komponenty encyjne - omówienie



- Adnotacja @javax.persistence.Id wskazuje na właściwość klucza głównego
- Adnotacja @javax.persistence.GeneratedValue określa, że dany kontener lub baza danych ma automatycznie generować odpowiedni identyfikator w chwili utworzenia egzemplarza encji Car w bazie danych.

# Komponenty biznesowe



Komponenty pracujące po stronie serwera Enterprise JavaBeans można podzielić na dwie podstawowe kategorie:

- komponentów sesyjnych (*session beans*),
- oraz komponentów sterowanych komunikatami (*message-driven beans*).

# Komponenty biznesowe



- Dostęp do komponentów sesyjnych można uzyskiwać za pomocą rozmaitych protokołów obiektów rozproszonych.
- Komponenty sterowane komunikatami przetwarzają w sposób asynchroniczny komunikaty pochodzące z takich systemów jak JMS, istniejących systemów informatycznych oraz usług Web Services.
- Wszystkie serwery EJB muszą obsługiwać przynajmniej komponenty sterowane komunikatami generowanymi przez system JMS, jednak z reguły obsługują także inne rodzaje komponentów sterowanych komunikatami.

# Komponenty sesyjne



- Komponenty sesyjne są rozszerzeniami aplikacji klienckiej zarządzającej procesami lub zadaniami.
- Przykładowo komponent encyjny Car oferuje metody obsługujące pewne działania wykonywane bezpośrednio na encji modelującej samochód, ale nie mówi niczego o kontekście, w którym te działania są podejmowane.

# Komponenty sesyjne



- Proces składania zamówienia na samochód wymaga nie tylko dostępu do encji Car, ale też do pewnych operacji i danych, które z samym samochodem nie mają bezpośrednio nic wspólnego — każda taka operacja musi uwzględniać cennik, dostawy, stan placu, itp.
- Za koordynację tych wszystkich działań musi odpowiadać właściwy komponent sesyjny.
- Komponenty sesyjne z reguły zarządzają określonymi rodzajami działań, np. związanymi wyłącznie z wykonywaniem zamówienia lub przygotowaniem kosztorysu.

# Komponenty sesyjne



- Komponenty sesyjne wykonują wiele operacji związanych z relacjami łączącymi poszczególne komponenty encyjne.
- Przykładowo, może się okazać, że złożenie zamówienia przez komponent sesyjny OrderAgent będzie wymagało współpracy z aż czterema komponentami encyjnymi: CustomerOrder, Order, Car oraz Customer.

# Komponenty sterowane komunikatami



- Komponenty sterowane komunikatami odpowiadają za koordynację zadań, w które są zaangażowane pozostałe komponenty sesyjne i encyjne.
- Komponenty sterowane komunikatami różnią się od komponentów sesyjnych przede wszystkim sposobem, w jaki możemy uzyskiwać dostęp do oferowanych funkcji.
- O ile komponent sesyjny oferuje interfejs zdalny określający, które z jego metod mogą być przedmiotem wywołań, komponent sterowany komunikatami rejestruje się jako adresat określonych komunikatów.

# Komponenty sterowane komunikatami



- Komponenty sterowane komunikatami realizują swoje zadania, przetwarzając otrzymywane komunikaty i zarządzając działaniami podejmowanymi przez pozostałe komponenty.
- Przykładowo komponent sterowany komunikatami OrderProcessor otrzymuje w sposób asynchroniczny komunikaty (pochodzące np. z istniejącego systemu zamówień) i na tej podstawie koordynuje działanie komponentów Car, Order, CustomerOrder oraz Customer, aby ostatecznie dokonać zamówienia.

# Komponenty biznesowe



Działania podejmowane przez komponenty sesyjne i komponenty sterowane komunikatami mają charakter przejściowy (chwilowy):

- rozpoczynamy proces składania zamówienia, wykonujemy określoną liczbę zadań składowych i kończymy bieżącą operację.
- Same komponenty sesyjne i sterowane komunikatami nie reprezentują informacji składowanych w bazie danych, chociaż ich działania mają wpływ na zawartość bazy danych.
- Przykładowo, wskutek zamówienia samochodu może zostać utworzony nowy komponent Order reprezentujący związek pomiędzy komponentami encyjnymi Customer, CustomerOrder i Car (reprezentującymi odpowiednio klienta, zamówienie klienta i samochód).

# Komponenty biznesowe



- Wszystkie te zmiany zostaną odwzorowane w bazie danych przez operacje wykonywane na właściwych komponentach encyjnych.
- Okazuje się, że także komponenty sesyjne i komponenty sterowane komunikatami, np. komponenty OrderAgent i OrderProcessor odpowiedzialne za składanie zamówienia, mogą uzyskiwać bezpośredni dostęp do bazy danych i wykonywać operacje odczytu, aktualizacji i usuwania danych.
- Z drugiej strony, baza danych nie zawiera rekordów reprezentujących komponenty OrderAgent i OrderProcessor – po wykonaniu bieżącego zamówienia wspomniane komponenty oczekują na przetworzenie kolejnego żądania.

# Komponenty biznesowe



Specyfikacje Enterprise JavaBeans i Java Persistence określają to, że:

- Komponenty encyjne charakteryzują się trwałym stanem.
- Komponenty sesyjne i komponenty sterowane komunikatami modelują interakcje, ale ich stan nie jest trwały.

# Komponenty biznesowe - Interfejsy



Aby zaimplementować komponent sesyjny lub komponent sterowany komunikatami, należy zdefiniować zarówno odpowiednie interfejsy, jak i klasę samego komponentu:

- Interfejs zdalny definiuje metody biznesowe komponentu sesyjnego, które mają być dostępne z poziomu aplikacji spoza kontenera EJB, czyli metody biznesowe udostępniane przez dany komponent aplikacjom zewnętrznym żądającym realizacji określonych zadań.
- Interfejs zdalny ma co prawda postać zwykłego interfejsu Javy, ale jego identyfikacja w roli interfejsu zewnętrznego wymaga oznaczenia adnotacją @javax.ejb.Remote.

# Komponenty biznesowe - Interfejsy



- Interfejs lokalny definiuje metody biznesowe komponentu sesyjnego, które mogą być wykorzystywane przez pozostałe komponenty pracujące w tym samym kontenerze EJB – metody biznesowe udostępniane przez dany komponent sesyjny pozostałym komponentom działającym w tej samej wirtualnej maszynie Javy.
- Interfejs lokalny umożliwia interakcję pomiędzy komponentami bez konieczności stosowania pośrednictwa kosztownego protokołu obiektów rozproszonych i podnosi efektywność takiej lokalnej współpracy.
- Interfejs lokalny ma co prawda postać zwykłego interfejsu Javy, ale jego identyfikacja w roli interfejsu lokalnego wymaga oznaczenia adnotacją @javax.ejb.Local.

# Komponenty biznesowe - Interfejsy



- Interfejs punktu końcowego definiuje metody biznesowe, które mogą być wywoływane z poziomu aplikacji spoza danego kontenera EJB, za pośrednictwem protokołu SOAP.
- Interfejs punktu końcowego bazuje na interfejsie Java API for XML-RPC (JAX-RPC) i został zaprojektowany z myślą o współpracy z takimi standardami jak SOAP i WSDL.
- Interfejs punktu końcowego ma co prawda postać zwykłego interfejsu Javy, ale jego identyfikacja w roli interfejsu punktu końcowego wymaga oznaczenia adnotacją @javax.jws.WebService.

# Komponenty biznesowe - Interfejsy



- Komponenty sterowane komunikatami muszą implementować tzw. interfejs komunikatów definiujący metody, za pośrednictwem których systemy przesyłania komunikatów, np. JMS, mogą do tych komponentów dostarczać komunikaty.

# Komponenty biznesowe - Interfejsy



- Klasa komponentu sesyjnego zawiera logikę biznesową i przynajmniej jeden interfejs(zdalny, lokalny lub punktu końcowego).
- Tego rodzaju klasy z reguły implementują wymienione wcześniej interfejsy, chociaż specyfikacja tego wprost nie wymaga.
- Pojedyncza klasa komponentu może zawierać więcej niż jeden interfejs tego samego typu.

# Komponenty biznesowe - Interfejsy



- Kontener EJB z reguły określa, czy dany komponent sesyjny ma charakter zdalny i (lub) lokalny na podstawie implementowanych przez ten komponent interfejsów.
- Klasa komponentu sesyjnego musi być oznaczona albo adnotacją @javax.ejb.Stateful, albo adnotacją @javax.ejb.Stateless, ponieważ tylko w oparciu o te adnotacje kontener EJB może określić rodzaj danego komponentu sesyjnego.
- Każdy komponent sterowany komunikatami implementuje jedną lub wiele metod odpowiedzialnych za dostarczanie komunikatów (nazwanych np. onMessage()) i zdefiniowanych przez interfejs komunikatów.
- Kontener wywołuje te metody w chwili otrzymania nowego komunikatu.

# Komponenty biznesowe - Interfejsy



- Sama klasa komponentu sterowanego komunikatami musi być oznaczona adnotacją @javax.ejb.MessageDriven.
- Wszystkie kontenery EJB 3.0 muszą obsługiwać komponenty sterowane komunikatami w systemie JMS, czyli takie, które implementują interfejs javax.jms.MessageListener.
- Specyfikacja EJB przewiduje też obsługę komponentów sterowanych komunikatami, które przetwarzają komunikaty pochodzące z systemów przesyłania komunikatów innych typów (charakteryzujących się własnymi interfejsami komunikatów).

# Komponenty biznesowe - interfejsy



- Interfejsy lokalne umożliwiają efektywną współpracę komponentów sesyjnych znajdujących się w tym samym kontenerze EJB.
- Wywołania metod interfejsu lokalnego nie wymagają stosowania protokołu obiektów rozproszonych.
- Komponent sesyjny nie musi jednak oferować interfejsu lokalnego, jeśli projektant danego komponentu przewiduje współpracę tylko z klientami zdalnymi lub usługami Web Services.
- Podobnie, komponent sesyjny nie musi oferować interfejsu zdalnego ani interfejsu punktu końcowego, jeśli wiadomo, że jego metody będą wywoływane wyłącznie przez komponenty sesyjne pracujące w tym samym kontenerze EJB.

# Komponenty biznesowe - interfejsy



- Budując komponenty sesyjne, możemy więc stosować dowolne kombinacje interfejsów lokalnych, zdalnych i punktów końcowych.
- Oprogramowanie klienckie współpracujące z komponentem sesyjnym nigdy nie ma bezpośredniego dostępu do jego klasy (dotyczy to także tych klientów, którzy sami są komponentami sesyjnymi).
- Zamiast tego oprogramowanie klienckie musi korzystać z metod wchodzących w skład interfejsów oferowanych przez dany komponent sesyjny.

# Komponenty biznesowe - interfejsy



- Oprogramowanie klienckie korzystające z interfejsów danego komponentu w praktyce współpracuje z pośrednikami lub tzw. namiastkami (*stubs*) wygenerowanymi automatycznie przez kontener EJB.
- Mimo, że korzystanie z metod interfejsów lokalnych nie wymaga stosowania protokołu obiektów rozproszonych, tego rodzaju interfejsy i tak reprezentują pośredników lub namiastki właściwe dla danej klasy komponentu. Wynika to z faktu, że – choć nie jest konieczna komunikacja za pośrednictwem sieci — pośrednik lub namiastka **umożliwiają** kontenerowi monitorowanie interakcji pomiędzy komponentem a jego klientem i — tym samym — stosowanie reguł bezpieczeństwa oraz właściwą obsługę transakcji.

# Komponenty biznesowe - interfejsy



- Należy pamiętać, że komponenty sterowane komunikatami nie obsługują interfejsów zdalnych,lokalnych ani punktów końcowych, co nie oznacza, że nie mogą występować w roli klientów innych komponentów sesyjnych i komunikować się z nimi za pośrednictwem ich własnych interfejsów.
- Komponenty sesyjne współpracujące z komponentem sterowanym komunikatami mogą się znajdować albo w tym samym kontenerze EJB (wówczas komponent sterowany komunikatami wykorzystuje ich interfejsy lokalne), albo w innej przestrzeni adresowej i innym kontenerze EJB (wówczas komponent sterowany komunikatami wykorzystuje ich interfejsy zdalne lub interfejsy punktu końcowego).

# Komponenty biznesowe - interfejsy



- Wiele operacji interakcji ma miejsce także pomiędzy komponentami EJB a samym kontenerem. (terminów kontener i serwer można w tym kontekście używać zamiennie).
- Kontener EJB odpowiada za tworzenie nowych egzemplarzy komponentów, zapewnianie ich właściwego składowania przez serwer itp. Narzędzia tworzone przez producentów kontenerów wykonują w tle bardzo wiele często złożonych i czasochłonnych zadań.

# Komponenty biznesowe - interfejsy



- Co najmniej jedno narzędzie jest odpowiedzialne za tworzenie komponentów encyjnych i ich odwzorowywanie w rekordach bazy danych.
- Pozostałe narzędzia generują kod zarówno na podstawie interfejsów komponentów, jak i samych klas komponentów.
- Generowany kod odpowiada z kolei za takie działania jak tworzenie komponentu, składowanie go w bazie danych itp.

# Komponenty - nazewnictwo



- Kiedy mówimy o komponencie EJB jako całości, a więc o jego interfejsach, klasie komponentu itp., stosujemy jego nazwę biznesową poprzedzoną słowem komponent i skrótem EJB.
- Przykładowo gotowy komponent opracowany z myślą o przetwarzaniu płatności dokonywanych z użyciem kart kredytowych będziemy nazywali komponentem EJB ProcessPayment.

# Komponenty - nazewnictwo



- Terminy komponent biznesowy lub komponent EJB odnoszą się do dowolnego rodzaju komponentu, zatem może chodzić zarówno o komponent sesyjny, jak i o komponent sterowany komunikatami.
- Zamiast nazwy komponent sterowany komunikatami często stosuje się wygodny akronim MDB (*Message Driven Bean*).

# Komponenty - nazewnictwo



- Do wyróżniania definiowanych przez komponenty interfejsów lokalnych, zdalnych i punktów końcowych stosujemy odpowiednie przyrostki.
- Kiedy mówimy o interfejsie zdalnym lub lokalnym komponentu sesyjnego OrderAgent, łączymy nazwę komponentu biznesowego odpowiednio ze słowem Remote lub Local.

# Komponenty - nazewnictwo



- Przykładowo, interfejs zdalny komponentu EJB OrderAgent nazywamy interfejsem OrderAgentRemote.
- Podobnie interfejs lokalny komponentu EJB OrderAgent nazwalibyśmy interfejsem OrderAgentLocal.
- Interfejs punktu końcowego komponentu – usługi sieciowej EJB OrderAgent nazwalibyśmy interfejsem OrderAgentWS (gdzie przyrostek WS reprezentuje Web Service).
- Nazwa klasy komponentu zawsze jest połączeniem nazwy samego komponentu i przyrostka Bean. Przykładowo klasę komponentu EJB OrderAgent nazwalibyśmy OrderAgentBean.

# Komponent bezstanowy - przykład



Poniżej przedstawiony jest sposób budowy prostego, bezstanowego komponentu sesyjnego z pojedynczym interfejsem zdalnym. Komponent ten to EJB Echo, czyli komponent sesyjny udostępniający w formie usługi proste funkcje odpowiadania. Funkcję interfejsu zdalnego komponentu EJB Echo będzie pełnił interfejs Javy nazwany EchoRemote i definiujący dwie proste metody. Do oznaczania interfejsów zdalnych służy adnotacja @javax.ejb.Remote. Interfejs zdalny tego komponentu:

```
import javax.ejb.Remote;
@Remote
public interface EchoRemote {
        public String answer(String question);
        public String justEcho(String message);
}
```

# Komponent bezstanowy - przykład



- Chociaż mamy tutaj do czynienia z interfejsem zdalnym komponentu sesyjnego EJB — przedstawiony kod nie zawiera odwołań do interfejsów Java RMI ani żadnych innych API.
- Brak tego rodzaju odwołań jest jednym z najważniejszych elementów odróżniających specyfikację Enterprise JavaBeans 3.0 od starszych wersji tej technologii.
- Specyfikacja EJB przewiduje co prawda konieczność stosowania przynajmniej jednego protokołu Java RMI-IIOP w roli sieciowego protokołu transportowego, jednak wszelkie odwołania do mechanizmu RMI wyeliminowano z kodu źródłowego, aby ułatwić pracę z innymi protokołami wybranymi przez producentów kontenerów.

# Komponent bezstanowy - przykład



Właściwa klasa bezstanowego komponentu sesyjnego wygląda następująco:

```
import javax.ejb.*;
@Stateless
public class CalculatorBean implements
    CalculatorRemote {
        public String answer(String question);
                return "Yes - " + question;
        }
        public String justEcho(String message);
                return message;
        }
}
```

# Komponent bezstanowy - przykład



- Klasa EchoBean musi nie tylko implementować przynajmniej jeden interfejs zdalny lub lokalny, ale też zawierać adnotację @javax.ejb.Stateless.
- Przedstawiona klasa nie implementuje żadnego interfejsu właściwego dla technologii EJB ani żadnych mechanizmów powiadomień zwrotnych.
- Zgodnie ze specyfikacją EJB 3.0, wszystkie komponenty EJB mają przypominać (na tyle, na ile jest to możliwe) zwykłe obiekty Javy.
- Nie oznacza to , że w nowej specyfikacji pominięto mechanizmy związane z powiadomieniami zwrotnymi — zmieniono tylko ich status z elementów wymaganych na rozwiązania stosowane jedynie w razie konieczności.

# Komponenty sterowane komunikatami



- Także dla komponentów sterowanych komunikatami należy definiować klasy implementujące interfejsy komunikatów, jednak tego rodzaju komponenty z natury rzeczy nie implementują interfejsów zdalnych, lokalnych ani punktów końcowych.
- Klasy komponentów sterowanych komunikatami oznacza się za pomocą adnotacji @javax.ejb.MessageDriven.
- Rodzaj implementowanych przez komponent MDB metod odpowiedzialnych za dostarczanie komunikatów zależy od typu obsługiwanej usługi przesyłania komunikatów.

# Komponenty sterowane komunikatami



- Przykładowo komponenty sterowane komunikatami zgodne ze standardem JMS (który musi być obsługiwany przez wszystkie kontenery EJB) muszą implementować metodę onMessage() wywoływaną za każdym razem, gdy do kontenera dociera asynchroniczny komunikat JMS.
- Komponenty sterowane komunikatami nie mają kluczy głównych z dokładnie tego samego powodu co komponenty sesyjne.
- Oba rodzaje komponentów nie są trwałe, zatem nie potrzebują kluczy identyfikujących odpowiednie rekordy w bazach danych.

# Komponenty sterowane komunikatami



- Jak kontener EJB obsługuje mechanizmy bezpieczeństwa, transakcji itp.?
- Kontener EJB uzyskuje tego rodzaju informacje (niezbędne do prawidłowego wykonywania aplikacji) z ustawień domyślnych, adnotacji i (lub) deskryptorów wdrożenia w formacie XML.

# Komponenty sterowane komunikatami



- Aby uprościć proces wytwarzania komponentów EJB, nowa specyfikacja Enterprise JavaBeans 3.0 definiuje szereg intuicyjnych ustawień domyślnych, które zwalniają programistów z obowiązku dodawania do tworzonych plików źródłowych niezliczonych adnotacji czy samodzielnego pisania deskryptorów wdrożenia w języku XML.
- Przykładowo domyślną wartością właściwości reprezentującej przetwarzanie transakcyjne jest REQUIRED.
- Domyślna semantyka zabezpieczeń jest reprezentowana przez wartość UNCHECKED.
- Istnienie tych i innych wartości domyślnych umożliwia programiście koncentrowanie się na pisaniu logiki biznesowej zamiast tracić czas na tworzenie niepotrzebnych metadanych.

# Adnotacje, deskryptory wdrożenia



- Aby uprościć proces wytwarzania komponentów EJB, nowa specyfikacja Enterprise JavaBeans 3.0 definiuje szereg intuicyjnych ustawień domyślnych, które zwalniają programistów z obowiązku dodawania do tworzonych plików źródłowych niezliczonych adnotacji czy samodzielnego pisania deskryptorów wdrożenia w języku XML.
- Przykładowo domyślną wartością właściwości reprezentującej przetwarzanie transakcyjne jest REQUIRED.
- Domyślna semantyka zabezpieczeń jest reprezentowana przez wartość UNCHECKED.

# Adnotacje, deskryptory wdrożenia



- Istnienie tych i innych wartości domyślnych umożliwia programiście koncentrowanie się na pisaniu logiki biznesowej zamiast tracić czas na tworzenie niepotrzebnych metadanych.
- Jeśli jednak ustawienia domyślne z jakiegoś powodu okażą się niewystarczające, programista może użyć bezpośrednio w swoim kodzie adnotacji Javy.
- Adnotacje umożliwiają programistom komponentów EJB definiowanie (bezpośrednio w plikach klas komponentów) metadanych decydujących o funkcjonowaniu odpowiednich mechanizmów w takich obszarach jak bezpieczeństwo, przetwarzanie transakcyjne czy odwzorowywanie encji w bazach danych.

# Adnotacje, deskryptory wdrożenia



- Mechanizm adnotacji w pewnym sensie łagodzi przynajmniej część negatywnych skutków zależności od konkretnych narzędzi EJB, ponieważ programista nie musi wykorzystywać do definiowania niezbędnych metadanych złożonych plików XML.
- Większość zintegrowanych środowisk programistycznych (Integrated Development Environments — IDE) oferuje funkcje automatycznego podpowiadania kodu, a adnotacje technologii EJB wprost idealnie nadają się do tego rodzaju działań.

# Adnotacje, deskryptory wdrożenia



- Adnotacje są jak dotąd najprostszym sposobem definiowania metadanych EJB.
- Zdarza się jednak, że programista z pewnych względów chce lub jest zmuszony przykryć ustawienia reprezentowane przez adnotacje ustawieniami stosowanymi na poziomie całego wdrożenia.
- Co więcej, niektórzy programiści Javy w ogóle rezygnują z dobrodziejstw adnotacji, ponieważ nie chcą rezygnować z dotychczasowych przyzwyczajeń.
- W związku z tym twórcy nowych specyfikacji (od EJB 3.0) przewidzieli możliwość stosowania tzw. deskryptorów wdrożenia (*deployment descriptor*).

# Adnotacje, deskryptory wdrożenia



- Deskryptory wdrożenia są plikami w formacie XML umożliwiającymi programiście dostosowywanie zachowań komponentów EJB w czasie wykonywania aplikacji bez konieczności modyfikowania i ponownego kompilowania samego oprogramowania.
- Deskryptory wdrożenia umożliwiają nam opisywanie atrybutów decydujących o sposobie wykonywania komponentów serwera (w takich obszarach jak bezpieczeństwo, kontekst transakcyjny itp.).

# Adnotacje, deskryptory wdrożenia



- Już po zdefiniowaniu klasy komponentu i jego interfejsów programista może utworzyć i wypełnić danymi odpowiedni deskryptor wdrożenia — nowe ustawienia przykryją lub uzupełnią metadane reprezentowane przez adnotacje użyte w kodzie źródłowym.
- Środowiska IDE, które oferują możliwość wytwarzania komponentów Enterprise JavaBeans, często umożliwiają programistom generowanie deskryptorów wdrożenia na podstawie ustawień wybranych w oknach podobnych do arkuszy właściwości.

# Adnotacje, deskryptory wdrożenia



- Po ustawieniu wszystkich właściwości danego komponentu środowisko IDE zapisuje deskryptor wdrożenia w pliku dyskowym.
- Dopiero po skonstruowaniu i zapisaniu pliku deskryptora można spakować wszystkie pliki danego komponentu w pojedynczym pliku JAR, który zostanie później wdrożony.

# Adnotacje, deskryptory wdrożenia



- Wybór pomiędzy adnotacjami a deskryptorem wdrożenia w formacie XML zależy od osobistych preferencji programisty.
- Z reguły niechęć do deskryptorów wdrożenia wynika z faktu, że pliki w formacie XML są dużo obszerniejsze.
- Adnotacje mogą też służyć jako elementy dokumentujące pliki z kodem źródłowym.

# Adnotacje, deskryptory wdrożenia



- Analizując plik z kodem klasy lub interfejsu, można od razu stwierdzić (właśnie na podstawie użytych adnotacji), z którym składnikiem aplikacji EJB mamy do czynienia.
- Z drugiej strony część programistów uważa metadane stosowane bezpośrednio w kodzie źródłowym za rozwiązanie zbyt inwazyjne i dąży do całkowitego oddzielania swojej logiki biznesowej (zapisanej w „czystej” Javie) od mechanizmów EJB.
- Wielu programistów chciałoby też zachować możliwość modyfikowania metadanych bez konieczności wprowadzania jakichkolwiek zmian w plikach z kodem źródłowym.

# Adnotacje, deskryptory wdrożenia



- Takie podejście jest szczególnie uzasadnione w przypadku aplikacji, których metadane EJB są w jakimś stopniu uzależnione od docelowego środowiska.
- Ponieważ metadane reprezentowane przez adnotacje mogą być w części lub całości przykrywane przez deskryptor wdrożenia XML, dyskusja pomiędzy zwolennikami obu rozwiązań w praktyce okazuje się zupełnie niepotrzebna.
- Można przecież błyskawicznie tworzyć prototypy komponentów EJB z wykorzystaniem adnotacji, po czym — w razie konieczności — przykryć nieaktualne ustawienia metadanymi zawartymi w deskryptorze wdrożenia.

# Adnotacje, deskryptory wdrożenia, pliki JAR



- Pliki JAR w praktyce są zwykłymi archiwami ZIP zawierającymi zarówno klasy Javy, jak i wszystkie zasoby niezbędne do prawidłowego funkcjonowania określonej aplikacji.
- Pliki JAR mogą służyć do pakowania apletów, aplikacji Javy, komponentów JavaBeans, aplikacji internetowych (serwletów i stron JSP) oraz komponentów Enterprise JavaBeans.
- Pojedynczy plik JAR zawierający jeden lub wiele komponentów EJB musi zawierać klasy samych komponentów, interfejsy tych komponentów oraz klasy pomocnicze dla każdego ze swoich komponentów.

# Adnotacje, deskryptory wdrożenia, pliki JAR



- Archiwum JAR może też zawierać deskryptor wdrożenia w postaci pliku XML, jeśli programista komponentów zdecydował się na tę formę definiowania potrzebnych metadanych.
- W czasie wdrażania komponentu należy wskazać odpowiednim narzędziom wdrożeniowym kontenera EJB miejsce składowania gotowego pliku JAR.
- Kiedy kontener EJB otwiera wskazany plik JAR, w pierwszej kolejności poszukuje klas oznaczonych (za pomocą adnotacji) jako komponenty EJB i (lub) odczytuje zawartość deskryptora wdrożenia (jeśli taki zdefiniowano), aby właściwie wykryć wszystkie komponenty i uzyskać informacje na temat żądanego sposobu zarządzania tymi komponentami w czasie wykonywania.

# Adnotacje, deskryptory wdrożenia, pliki JAR



- Na podstawie adnotacji i (lub) deskryptora wdrożenia narzędzia kontenera odpowiedzialne za wdrażanie komponentów mogą określić, jakiego rodzaju komponenty są składowane w danym pliku JAR (mogą to być komponenty sesyjne lub komponenty sterowane komunikatami) oraz jak należy nimi zarządzać w ramach transakcji, kto powinien mieć dostęp do tych komponentów w czasie wykonywania itp.
- Osoba wdrażająca komponent może oczywiście zmienić część tych ustawień (np. dotyczących przetwarzania transakcyjnego i bezpieczeństwa), aby nowy komponent lepiej pasował do specyfiki całej aplikacji.

# Kontener EJB



- Komponenty sesyjne deklarują interfejsy, które mogą być wykorzystywane przez oprogramowanie klienckie do uzyskiwania dostępu do tych komponentów.
- Specyfikacja EJB 3.0 przewiduje, że oprogramowanie klienckie spoza systemu kontenera Enterprise JavaBeans zawsze musi wykorzystywać interfejsy zdalne odpowiednich komponentów EJB.

# Kontener EJB



- Aplikacje klienckie spoza kontenera EJB mają też możliwość uzyskiwania dostępu do bezstanowych komponentów sesyjnych dokładnie tak jak do usług Web Services.
- Aplikacje klienckie wchodzące w skład tego samego systemu Javy EE (komponentów EJB, serwletów lub stron JSP) mogą korzystać z pośrednictwa interfejsów lokalnych pod warunkiem, że działają w ramach tej samej maszyny wirtualnej.

# Kontener EJB



- Poniżej przedstawione zostały sposoby łączenia w czasie wykonywania interfejsów komponentów sesyjnych z egzemplarzami klas komponentów.
- Kontener EJB serwera JBoss implementuje interfejs komponentu sesyjnego w sposób umożliwiający klientom (mającym postać albo aplikacji spoza tego kontenera, albo komponentów pracujących w tym samym środowisku) interakcję i wywoływanie metod klasy danego komponentu.

# Kontener EJB



- Trzy elementy istotne dla używanej architektury: namiastka pośrednicząca, kontener EJB oraz egzemplarz samego komponentu.
- Większość programistów nigdy nie miała okazji do zapoznania się z wewnętrzną architekturą wykorzystywanego kontenera EJB, ponieważ faktycznym zadaniem oprogramowania pośredniczącego jest na tyle skuteczne ukrywanie tego rodzaju mechanizmów, aby twórcy aplikacji mogli się koncentrować wyłącznie na pisaniu logiki biznesowej.

# Kontener EJB



- Takie podejście ma wiele zalet, ponieważ pozwala wyodrębnić obszary odpowiedzialności w zależności od poziomu wiedzy i doświadczenia programistów zaangażowanych w poszczególne działania.
- Programiści aplikacji muszą dysponować szczegółową wiedzą o sposobie pracy środowiska biznesowego i technikach jego modelowania, zatem powinni się koncentrować na tworzeniu aplikacji i komponentów opisujących odpowiednie wycinki działalności danego przedsiębiorstwa.

# Kontener EJB



- Programiści systemowi, czyli osoby odpowiedzialne za pisanie serwerów EJB, co prawda nie muszą rozumieć działań biznesowych modelowanych przez programistę aplikacji, ale ich praca wymaga wiedzy, jak należy pisać kontenery i pomocnicze obiekty rozproszone.
- Oznacza to, że programiści systemowi powinni wykorzystywać swoje umiejętności do implementowania rozwiązań zarządzających obiektami rozproszonymi pozostawiając implementowanie logiki biznesowej programistom aplikacji.

# Namiastka pośrednika, kontener EJB



- Kiedy zaimplementowana logika biznesowa współpracuje z komponentem sesyjnym, taka współpraca nigdy nie ma charakteru bezpośredniego (nie dotyczy egzemplarzy samej klasy komponentu), ponieważ wymaga zaangażowania interfejsu zdalnego lub lokalnego danego komponentu.
- Kiedy wywołujemy metody za pośrednictwem zdalnego lub lokalnego interfejsu, w praktyce korzystamy z egzemplarza obiektu nazywanego namiastką pośrednika (ang. proxy stub).
- Właśnie namiastka pośrednika implementuje interfejs zdalny lub lokalny komponentu sesyjnego i odpowiada za przekazywanie naszych żądań (wywołań metod komponentu sesyjnego) do zdalnego kontenera EJB lub do kontenera EJB pracującego w lokalnej maszynie wirtualnej Javy (JVM).

# Namiastka pośrednika, kontener EJB



- Namiastka pośrednika może być albo generowana przez prekompilator (jak w przypadku prekompilatora rmic technologii RMI), albo generowana dynamicznie w czasie wdrażania z wykorzystaniem mechanizmów JDK zawartych w klasie java.lang.reflect.Proxy (jak w przypadku serwera JBoss).
- Namiastka pośrednika kieruje wywołania metod do kontenera EJB na właściwym serwerze (lub w ramach tego samego serwera, jeśli wywołanie następuje za pośrednictwem interfejsu lokalnego).

# Kontener EJB i egzemplarz komponentu



- Zarówno za zarządzanie egzemplarzami klasy komponentu, jak i za izolowanie transakcji oraz zapewnianie bezpieczeństwa odpowiada sam kontener EJB.
- Kontener dysponuje informacjami zawartymi w metadanych zdefiniowanych w formie adnotacji i (lub) zapisanych w deskryptorze wdrożenia w formacie XML.
- Na podstawie tych metadanych kontener EJB inicjuje transakcje oraz wykonuje takie działania jak uwierzytelnianie i autoryzacja.

# Kontener EJB i egzemplarz komponentu



- Do zadań kontenera należy także zarządzanie cyklem życia egzemplarzy komponentów oraz kierowanie żądań otrzymywanych od pośrednika do egzemplarzy klas właściwych komponentów.
- Wywołanie może zostać przekazane do właściwego egzemplarza komponentu dopiero po tym, jak kontener EJB zainicjuje zarządzanie cyklem życia tego egzemplarza, rozpocznie przetwarzanie transakcji i wykona wszystkie testy wymagane przez politykę bezpieczeństwa.

# Przykład – zamówienie



- Przykład jak oprogramowanie klienckie może korzystać z rozwiązań implementowanych przez komponenty EJB.
- Zostanie utworzony przykładowy komponent sesyjny nazwany OrderAgent, który będzie odpowiedzialny za tworzenie zamówienia w odpowiedzi na żądania zdalnych klientów.
- Komponent EJB OrderAgent musi współpracować zarówno z różnymi komponentami encyjnymi, jak i z pozostałymi komponentami sesyjnymi.

# Przykład – zamówienie



- Komponenty encyjne są wygodnym narzędziem do reprezentowania danych i opisywania tych pojęć biznesowych, które można wyrażać w formie rzeczowników, ale nie sprawdzają się w modelowaniu procesów czy zadań.
- Przykładowo, komponent EJB Car oferuje metody i zachowania umożliwiające wykonywanie operacji bezpośrednio na reprezentacji samochodu, ale nie definiuje kontekstu, w którym te operacje są wykonywane.
- Koncepcja architektury wielowarstwowej powstała wskutek dążenia do wyeliminowania sytuacji, w której logika biznesowa jest definiowana w aplikacjach klienckich.
- Podobnie nie chcemy, aby tego rodzaju logika znajdowała się na poziomie komponentów encyjnych reprezentujących dostawców lub klientów.

# Przykład – zamówienie



- Wykonywanie zamówień należy do tych czynności (funkcji) biznesowych, która w żadnym razie nie powinna być implementowana przez komponenty Car lub Customer, ponieważ nie stanowi bytu biznesowego, tylko określony proces lub zadanie.
- Komponenty sesyjne pełnią funkcję agentów zarządzających procesami i zadaniami biznesowymi w imieniu oprogramowania klienckiego i jako takie stanowią właściwe miejsce dla logiki biznesowej.
- Komponenty sesyjne nie są trwałe, zatem żadne elementy tych komponentów nie są bezpośrednio odwzorowywane w bazie danych ani utrwalane pomiędzy kolejnymi sesjami.

# Przykład – zamówienie



- Komponenty sesyjne z reguły współpracują z komponentami encyjnymi oraz wykorzystują dane i inne zasoby do sterowania przepływem zadań (ang. taskflow).
- Przepływ zadań jest istotą wszystkich systemów biznesowych, ponieważ wyraża sposób, w jaki poszczególne encje ze sobą współpracują, modelując funkcjonowanie rzeczywistej organizacji biznesowej.
- Komponenty sesyjne kontrolują zadania i zasoby, mimo że same nie reprezentują żadnych danych.

# Przykład – zamówienie



- Poniższy fragment kodu demonstruje sposób, w jaki komponent sesyjny zaprojektowany z myślą o wykonywaniu zamówienia może sterować przepływem zadań realizowanych przez pozostałe komponenty encyjne i sesyjne.
- Fragment oprogramowania klienckiego, w tym przypadku interfejs użytkownika, uzyskuje zdalną referencję do komponentu sesyjnego OrderAgent.
- Wykorzystując informacje wpisane przez użytkownika w polach tekstowych odpowiedniego formularza aplikacja kliencka zamawia samochód dla pojedynczego klienta:

# Przykład – zamówienie



```
String creditCard = textCreditCard.getText( );
int carID = Car.FindCarIdByVin(textCarVin.getText(
    )));
Customer customer = new Customer(name, address,
    phone);
// korzystamy z mechanizmu JNDI do uzyskania
    potrzebnej referencji.
OrderAgentRemote orderAgent = ...;
OrderAgent.setCustomer(customer);
OrderAgent.setCarID(cabinID);
Order res = OrderAgent.createOrder(creditCard,
    price);
```

# Przykład – zamówienie



- W pierwszej kolejności uzyskujemy zdalną referencję do zdalnego interfejsu komponentu EJB OrderAgent.
- Potrzebujemy zdalnej referencji (nie interfejsu lokalnego), ponieważ w tym przypadku klient ma postać aplikacji działającej poza danym kontenerem EJB.
- Warto pamiętać, że — chociaż powyższy przykład tego nie demonstruje — referencje do zdalnych komponentów sesyjnych uzyskuje się z wykorzystaniem mechanizmu JNDI, czyli funkcjonalnego interfejsu API stworzonego z myślą o lokalizowaniu takich zasobów jak zdalne obiekty.

# Przykład – zamówienie



- Dalsza część kodu klienckiego współpracuje z komponentem EJB OrderAgent celem właściwego ustawienia komponentów Customer i Car dla żądanego zamówienia.
- W ostatnim kroku zapisujemy zamówienie (w istocie proces jest jeszcze bardziej złożony, wywołując metodę createOrder(), która zwraca egzemplarz klasy Order udostępniający dane przydatne podczas prezentacji użytkownikowi stosownego pokwitowania (np. za pośrednictwem graficznego interfejsu użytkownika).

# Przykład – zamówienie



- Mamy w tym przypadku do czynienia z dość zgrubną abstrakcją ponad procesem składania zamówienia dla klienta — większość szczegółowych działań wchodzących w skład tego procesu jest ukryta przed oprogramowaniem klienckim.
- Skuteczne ukrywanie szczegółów związanych z przepływem zadań jest o tyle ważne, że umożliwia elastyczność ewoluującego systemu informatycznego.
- Na tym etapie wiemy, że zawsze będzie istniała potrzeba zamawiania samochodów dla klientów, ale nie możemy być pewni, czy sam proces dokonywania tego rodzaju zamówień nie będzie modyfikowany.

# Przykład – zamówienie



- Poniżej przedstawiono kod klasy komponentu OrderAgent: OrderAgentBean.
- Metoda createOrder() współpracuje z czterema komponentami encyjnymi: Customer, Car i Order oraz z innym komponentem sesyjnym: ProcessPayment.
- Komponent EJB ProcessPayment oferuje szereg metod obsługujących proces dokonywania płatności z wykorzystaniem czeków, gotówki i kart kredytowych.
- W tym przypadku wykorzystujemy komponent ProcessPayment do obsługi płatności z użyciem karty kredytowej.
- Po wykonaniu tej operacji utworzony egzemplarz komponentu Order jest automatycznie odłączany od pamięci trwałej i zwracany klientowi.

# Przykład – zamówienie



```
@Stateful
public class OrderAgentBean implements
    OrderAgentRemote {
  @PersistenceContext private EntityManager
      entityManager;
  @EJB private ProcessPaymentRemote process;
  private Customer customer;
  private Car car;
  public void setCustomer(Customer cust) {
    entityManager.create(cust);
    customer = cust;
  }
  public void setCarID(int id) {
    car = entityManager.find(Car.class, id);
  }
```

# Przykład – zamówienie



```
  public Order createOrder(String card, double
      price) throws IncompleteConversationalState {
    if (customer == null || car == null) {
      throw new IncompleteConversationalState( );
    }
    try {
      Order order = new Order(customer, car, price,
          new Date( ));
      entityManager.persist(order);
      process.byCredit(customer, card, price);
      return order;
    } catch(Exception e) {
      throw new EJBException(e);
    }
  }
}
```