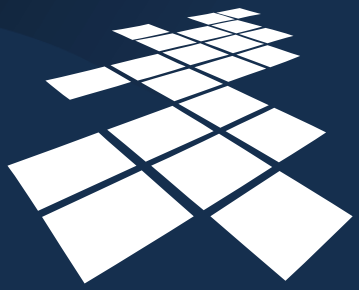
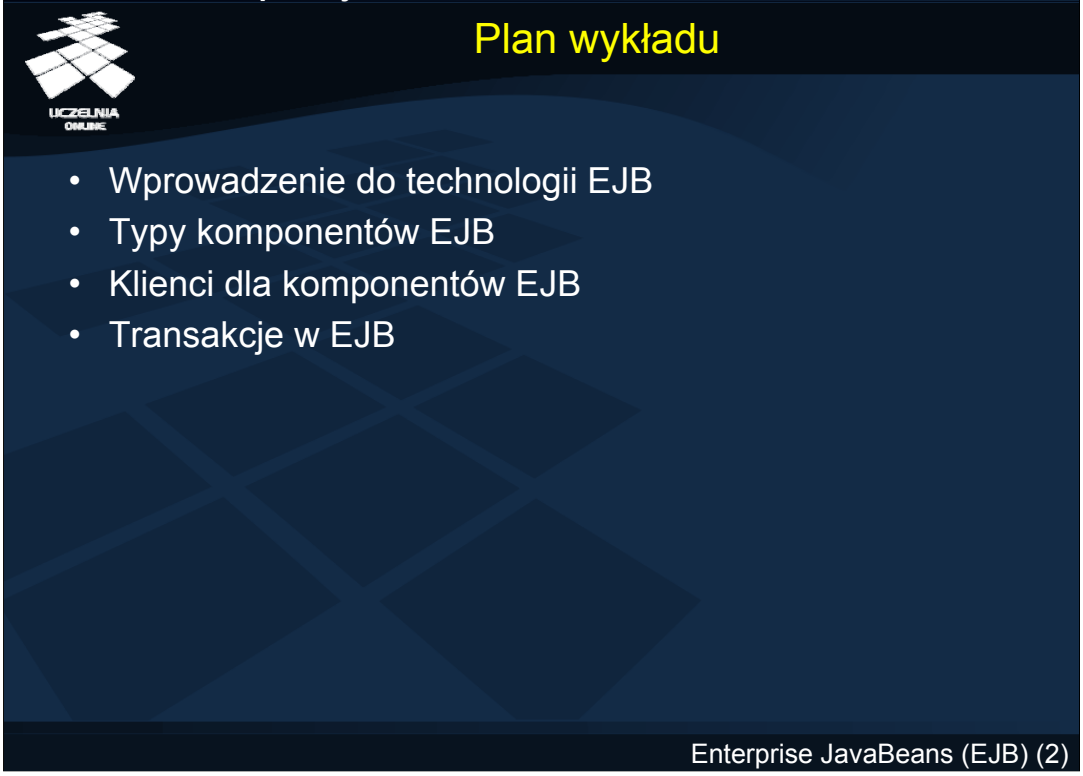
Enterprise JavaBeans (EJB)

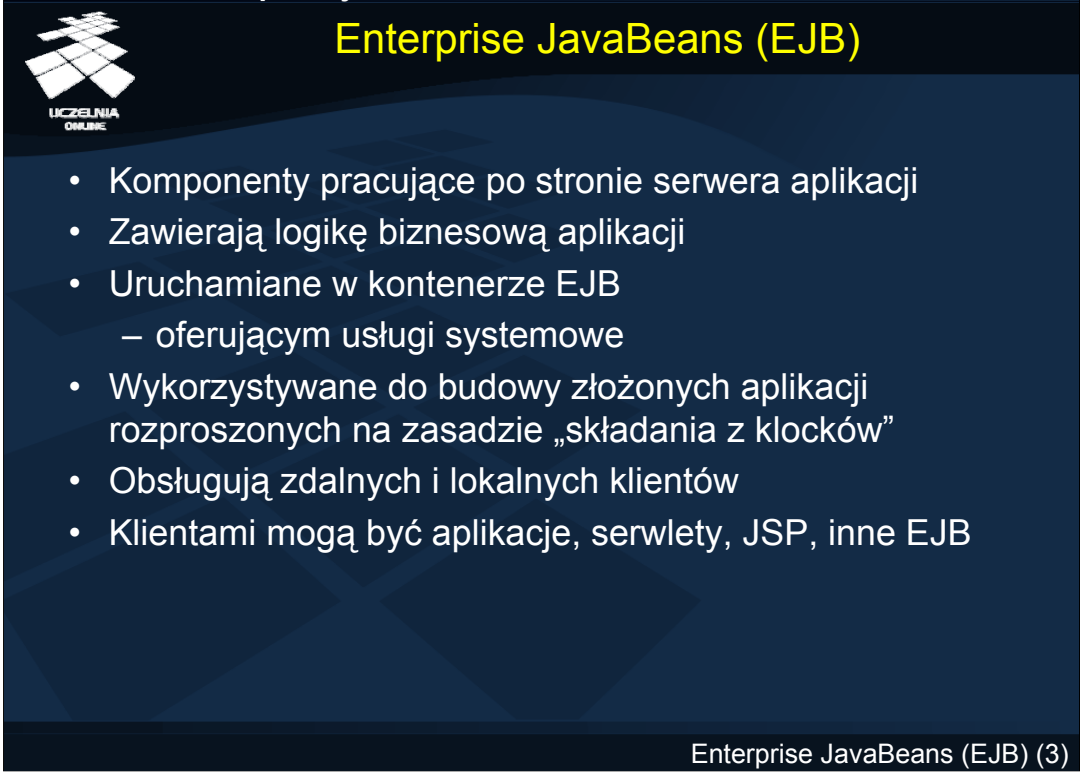
Wykład prowadzi:
Marek Wojciechowski



UCZELNIA
ONLINE



Celem wykładu jest przedstawienie technologii Enterprise JavaBeans (EJB). Wykład rozpocznie się od wprowadzenia do technologii EJB i przedstawienia jej roli w ramach platformy Java EE. Następnie przedstawione będą dwa istniejące typy komponentów EJB: komponenty sesyjne i komunikatowe. Omówione będą sposoby korzystania z komponentów EJB z rozróżnieniem na klientów zdalnych i lokalnych. Na zakończenie omówione będzie przetwarzanie transakcyjne na poziomie komponentów EJB.



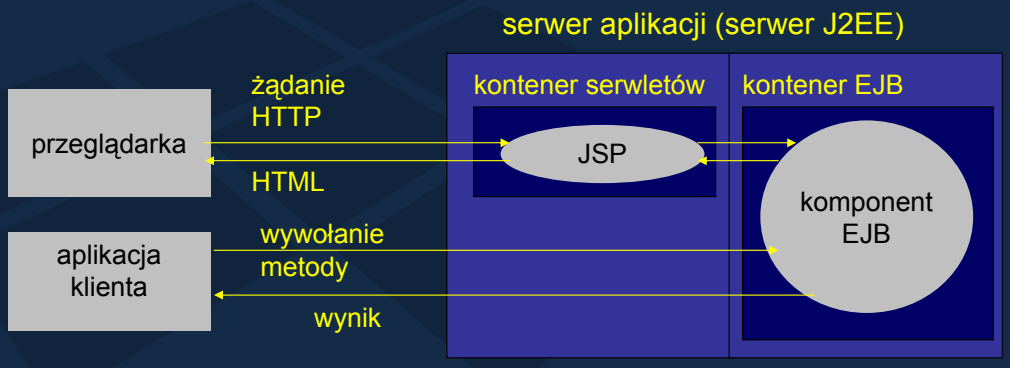
Enterprise JavaBeans (EJB) to komponenty pracujące po stronie serwera aplikacji, zawierające logikę biznesową, czyli to co jest celem aplikacji.

Komponenty EJB są uruchamiane w kontenerze EJB (ang. EJB container), środowisku uruchomieniowym stanowiącym część serwera aplikacji. Kontener dostarcza gotowe implementacje usług systemowych takich jak bezpieczeństwo czy obsługa transakcji, pozwalając twórcom aplikacji skupić się na logice biznesowej.

Komponenty EJB służą do budowy złożonych aplikacji rozproszonych na zasadzie „składania z klocków”. EJB obsługują zdalnych i lokalnych klientów. Klient lokalny to komponent działający w tej samej maszynie wirtualnej Java, a klient zdalny to komponent działający w innej maszynie wirtualnej, być może na innym komputerze. Klientami dla EJB mogą być aplikacje Java, serwlety, JSP lub inne komponenty EJB.

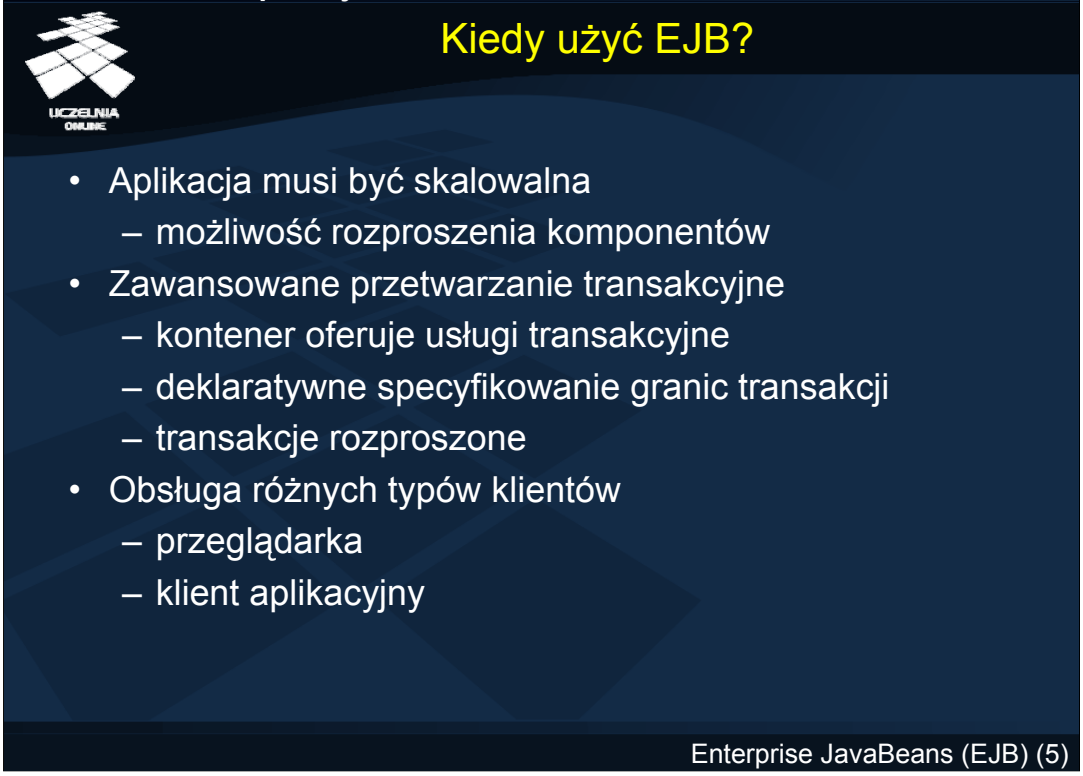
Rola EJB na platformie Java EE

- Nie wszystkie aplikacje Java EE wykorzystują technologię EJB
- EJB realizują logikę biznesową, zostawiając innym elementom aplikacji logikę prezentacji



Nie wszystkie aplikacje Java EE wykorzystują technologię EJB. Jeśli EJB są użyte, to odpowiadają za logikę biznesową, zostawiając innym elementom aplikacji logikę prezentacji.

Rysunek na slajdzie przedstawia architekturę aplikacji Java EE, w której zostały wykorzystane komponenty EJB. Aplikacja może posiadać dwa typy klientów: przeglądarkę i klienta aplikacyjnego, najczęściej z okienkowym interfejsem graficznym. Komponenty EJB stanowią wspólne dla wszystkich typów klientów miejsce implementacji logiki biznesowej. Klienci aplikacyjni korzystają z komponentów EJB bezpośrednio. Przeglądarka komunikuje się z serwletami i JSP, odpowiadającymi za generację interfejsu użytkownika w języku HTML i pośredniczącymi w komunikacji z EJB. Serwlety i JSP mogą pracować na tym samym serwerze aplikacji co EJB lub na innym.



Użycie w aplikacji komponentów EJB do implementacji logiki biznesowej należy rozważyć gdy aplikacja musi spełniać co najmniej jedno z wymienionych na slajdzie wymagań.

1. Gdy aplikacja musi być skalowalna. Technologia EJB umożliwia rozproszenie komponentów aplikacji poprzez umieszczenie komponentów odpowiadających za logikę prezentacji i EJB, odpowiadających za logikę biznesową, na różnych serwerach. Można również, w przypadku dużego obciążenia, rozproszyć komponenty EJB między serwerami. W ten sposób można zmniejszyć obciążenie pojedynczych serwerów, ale ceną są zwiększone koszty komunikacji między komponentami aplikacji.

2. Gdy aplikacja wymaga zawansowanego przetwarzanie transakcyjnego. Kontener EJB oferuje usługi transakcyjne z możliwością deklaratywnego specyfikowania granic transakcji. Transakcje mogą obejmować wywołania wielu metod różnych komponentów. EJB jest również odpowiednią technologią dla transakcji rozproszonych.

3. Gdy aplikacja ma mieć klientów różnych typów np. przeglądarka internetowa, urządzenia mobilne, aplikacja graficzna. Komponenty EJB implementują logikę biznesową i pozwalają na dostęp różnym typom klientów odpowiadającym za interfejs użytkownika.

Historia technologii EJB

- Od początku ważny element technologii Java EE
- Do wersji EJB 2.1 skomplikowane i nieefektywne
  - wiele interfejsów
  - obowiązkowe deskryptory instalacji w XML
  - interfejsy i klasy zależne od EJB API
  - nieefektywne komponenty encyjne
- W wersji EJB 3.0 znaczące uproszczenie
  - mniej plików źródłowych (POJO, POJI)
  - adnotacje zamiast deskryptorów XML
  - wstrzykiwanie zależności (ang. dependency injection)
  - komponenty encyjne zastąpione Java Persistence

Technologia EJB od początku historii Java EE stanowi ważny element Java EE 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. Sytuacje, w których EJB są szczególnie przydatne zostały omówione na poprzednim slajdzie.

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 (ang. deployment descriptor) 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, 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.

Niniejszy wykład oparty jest na wersji 3.0 technologii EJB. Opracowany wraz z EJB 3.0 standard Java Persistence będzie omówiony w ramach następnego wykładu.

Typy komponentów EJB (EJB 3.0)

- Sesyjne (Session Beans):
  - realizują konkretne zadanie dla klienta
  - w danej chwili nie są współdzielone
  - ich stan nie wykracza poza sesję
- Komunikatowe (Message-Driven Beans):
  - asynchroniczni konsumenci komunikatów
  - najczęściej wykorzystują technologię Java Message Service (JMS) i nasłuchują nadejścia komunikatu JMS
  - klienci nie odwołują się do nich bezpośrednio

W wersji 3.0 technologii EJB występują dwa główne typy komponentów: sesyjne i komunikatowe. Komponenty sesyjne można dalej podzielić na sesyjne stanowe i sesyjne bezstanowe.

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.

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.

Typy sesyjnych EJB

- Stanowe (ang. stateful)
  - zmienne instancji komponentu reprezentują stan dla konkretnej sesji z klientem
  - stan określany jako „stan konwersacji”, utrzymywany podczas sesji z klientem, obejmującej wiele wywołań metod
- Bezstanowe (ang. stateless)
  - nie pamiętają stanu konwersacji z klientem
  - użyteczny stan tylko na czas pojedynczego wywołania metody
  - mogą implementować Web Service

Sesyjne komponenty EJB mogą być stanowe (ang. stateful) lub bezstanowe (ang. stateless).

Stanowy komponent sesyjny pamięta stan dla konkretnej sesji z klientem (stan komponentu jest rozumiany jako stan zmiennych instancji). Stan ten jest określany jako „stan konwersacji”, gdyż utrzymywany jest podczas całej sesji z klientem, obejmującej wiele wywołań metod.

Bezstanowy komponent sesyjny nie pamięta stanu konwersacji z klientem. Jego stan jest użyteczny dla klienta tylko przez czas trwania pojedynczego wywołania metody, gdyż serwer nie gwarantuje, że przy kolejnym wywołaniu metody klientowi będzie przydzielona ta sama instancja komponentu.

Bezstanowy komponent sesyjny może implementować Web Service, co zostanie omówione w ramach wykładu na temat Web Services.

Stanowy czy bezstanowy?

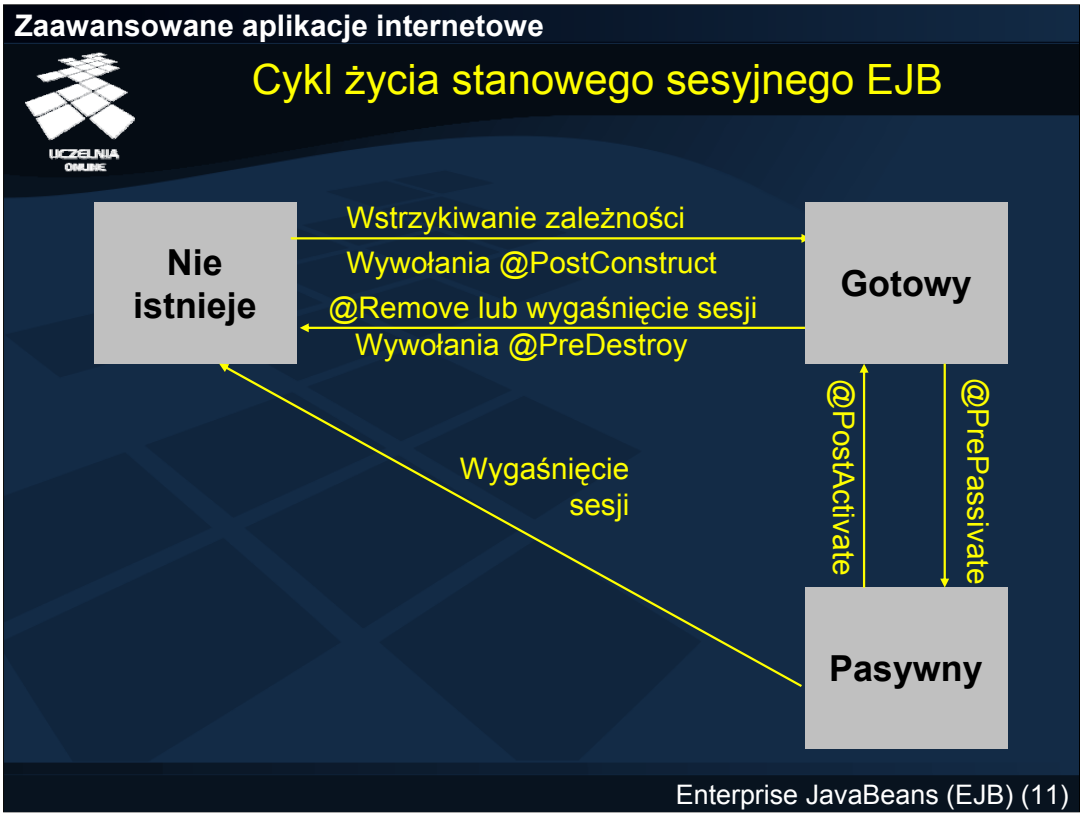
- Odpowiedź nie jest oczywista, ponieważ bezstanowe mogą być efektywniejsze
- Stanowy odpowiedni gdy:
  - komponent reprezentuje interakcję z klientem
  - komponent musi zachowywać stan między wywołaniami metod
  - komponent zarządza przepływem sterowania
- Bezstanowy odpowiedni gdy:
  - komponent realizuje ogólne zadanie
  - komponent nie pamięta danych dla konkretnego klienta

Wybór między oferującym większą funkcjonalność komponentem stanowym a bezstanowym nie jest oczywisty, gdyż bezstanowe komponenty mogą być efektywniejsze. Dla komponentów stanowych w szczególnych sytuacjach kontener musi zapisać ich stan w pamięci trwałej, a następnie go odtworzyć, co jest czasochłonne. Ogólnie preferowane są komponenty bezstanowe, a stanowe powinny być używane gdy pamiętanie stanu jest niezbędne.

W szczególności, stanowy komponent sesyjny jest odpowiedni gdy: komponent reprezentuje interakcję z konkretnym klientem; komponent musi zachowywać stan między wywołaniami metod; komponent zarządza przepływem sterowania między innymi komponentami.

Bezstanowy komponent sesyjny jest odpowiedni gdy komponent realizuje ogólne zadanie np. konwersję danych, wysłanie potwierdzenia pocztą elektroniczną itp. i nie przechowuje danych dla konkretnego klienta.

Cykl życia bezstanowego sesyjnego komponentu EJB jest bardzo prosty i występują w nim tylko dwa stany. Instancja komponentu albo nie istnieje albo jest gotowa do użycia. Kontener tworzy instancję komponentu gdy klient żąda referencji do komponentu, a nie ma wolnej gotowej instancji. Po utworzeniu instancji, a przed jej udostępnieniem klientowi, kontener wstrzykuje zależności (np. obiekt reprezentujący transakcję lub referencje do innych komponentów EJB), a następnie wywołuje metody oznaczone adnotacją @PostConstruct. W stanie gotowości klient może wywoływać metody biznesowe na rzecz komponentu. Gdy instancja komponentu nie jest potrzebna, kontener może podjąć decyzję o jej usunięciu. Przed udostępnieniem instancji dla mechanizmu garbage collection, kontener wywołuje metody oznaczone adnotacją @PreDestroy.



W cyklu życia stanowego sesyjnego komponentu EJB występują trzy stany: nie istnieje, gotowy i pasywny. Kontener tworzy instancję komponentu gdy klient żąda referencji do komponentu. Po utworzeniu instancji, a przed jej udostępnieniem klientowi, kontener wstrzykuje zależności, a następnie wywołuje metody oznaczone adnotacją @PostConstruct. W stanie gotowości klient może wywoływać metody biznesowe na rzecz komponentu. Gdy instancja jest w stanie gotowości, kontener może podjąć decyzję o pasywacji instancji. Pasywacja polega na wymieceniu instancji z pamięci i zachowaniu jej stanu w pamięci trwałej (np. na dysku). Gdy klient wywoła metodę na rzecz pasywnej instancji, zostanie ona aktywowana przez kontener, a jej stan odtworzony. Przed pasywacją kontener wywołuje metody oznaczone adnotacją @PrePassivate, a po aktywacji – metody oznaczone adnotacją @PostActivate. Gdy klient skończy pracę z komponentem, może jawnie zlecić kontenerowi usunięcie instancji komponentu poprzez wywołanie metody oznaczonej adnotacją @Remove. (Jest to jedyna metoda związana z cyklem życia jawnie wywoływana w kodzie. Pozostałe są w odpowiednich momentach wywoływane przez kontener.) Kontener usunie instancję komponentu również w sytuacji gdy wygaśnie sesja (zarówno dla pasywnej jak i aktywnej instancji). Przed udostępnieniem instancji dla mechanizmu garbage collection w związku z jej usuwaniem, kontener wywołuje metody oznaczone adnotacją @PreDestroy.

Struktura kodu sesyjnego EJB

- Kod sesyjnego komponentu EJB stanowią:
  - klasa komponentu
  - interfejsy biznesowe
  - klasy pomocnicze
- Konwencje nazw dla EJB:
  - nazwa komponentu: <nazwa>Bean, np. BasketBean
  - nazwa klasy komponentu <nazwa>Bean, np. BasketBean
  - interfejs biznesowy: <nazwa>, np. Basket

Kod źródłowy sesyjnego komponentu EJB obejmuje:

(a) klasę komponentu, implementującą interfejs biznesowy komponentu i metody cyklu życia jeśli są wykorzystywane;

(b) interfejsy biznesowe, deklarujące metody udostępniane przez komponent klientom zdalnym i lokalnym, tworzone w formie zwykłego interfejsu Java opatrzonego adnotacjami;

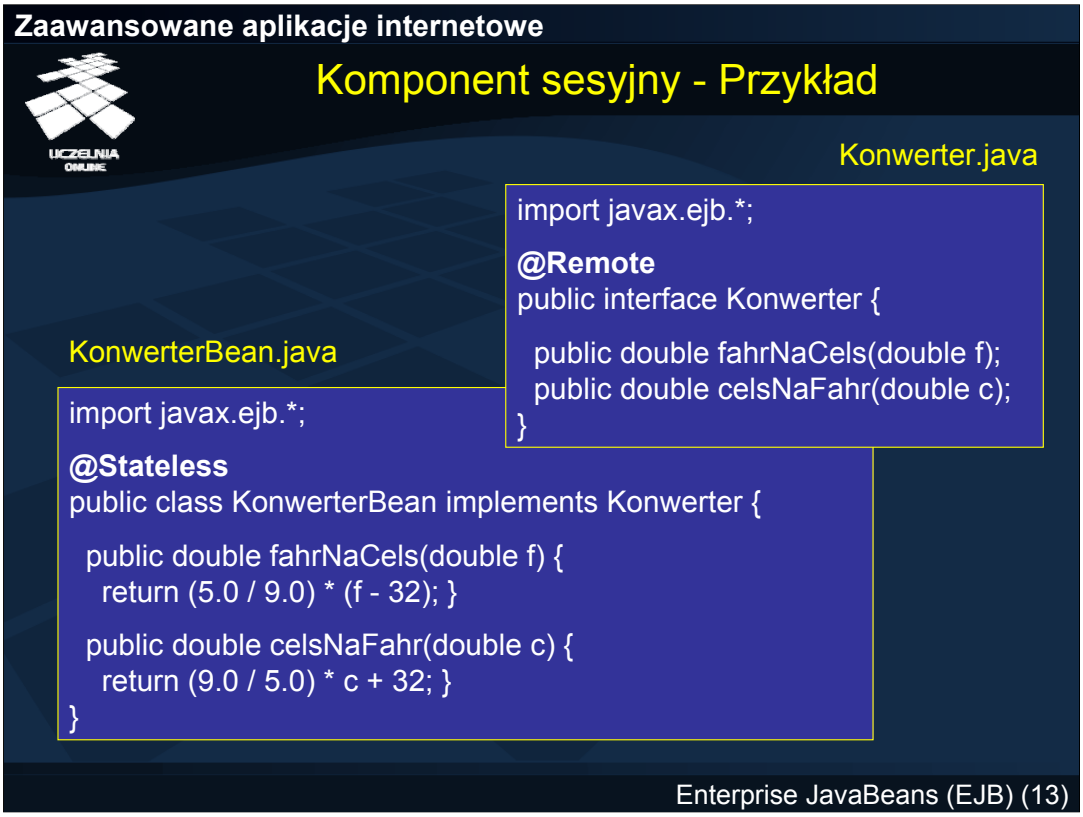
(c) opcjonalnie klasy pomocnicze, wykorzystywane przez klasę komponentu.

Dla ułatwienia orientacji w plikach źródłowych zaproponowano następującą konwencję nazewnictwa:

(a) nazwa komponentu: <nazwa>Bean, np. BasketBean;

(b) nazwa klasy komponentu <nazwa>Bean, np. BasketBean;

(c) interfejs biznesowy: <nazwa>, np. Basket.



Slajd przedstawia pliki z kodem źródłowym bezstanowego sesyjnego komponentu EJB, służącego 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).

Klienci dla sesyjnych EJB

- Klient zdalny (ang. remote client)
  - może (ale nie musi!) działać na innej JVM i innym serwerze
  - może nim być serwlet, JSP, EJB lub klient aplikacyjny
  - lokalizacja EJB dla klienta jest transparentna
  - interfejs biznesowy EJB oznaczony @Remote
- Klient lokalny (ang. local client)
  - musi działać na tej samej JVM
  - może nim być serwlet, JSP lub EJB
  - lokalizacja EJB dla klienta nie jest transparentna
  - interfejs biznesowy EJB oznaczony @Local

Klient zdalny (ang. remote client) może (ale nie musi!) działać na innej maszynie wirtualnej Java i innym serwerze. Klientem zdalnym może być serwlet, JSP, EJB lub klient aplikacyjny. Lokalizacja EJB dla klienta jest transparentna. Klient uzyskuje referencję do tzw. pieńka (ang. stub) i wywołuje na jego rzecz metody biznesowe. Pieniek w sposób transparentny dla klienta obsługuje komunikację z instancją komponentu w kontenerze EJB. Interfejs biznesowy EJB dla klientów zdalnych jest oznaczony adnotacją @Remote.

Klient lokalny (ang. local client) musi działać na tej samej maszynie wirtualnej Java co komponent EJB. Klientem lokalnym może być serwlet, JSP lub EJB. Lokalizacja EJB dla klienta nie jest transparentna, klasa EJB i klasa klienta muszą być załadowane do maszyny wirtualnej przez ten sam class loader. Interfejs biznesowy EJB dla klientów lokalnych jest oznaczony adnotacją @Local.

Dostęp lokalny czy zdalny?

- Związki między komponentami: ścisłe czy luźne?
- Typ klientów: aplikacje czy serwlety/JSP/EJB?
- Planowane rozproszenie dla skalowalności?

- Uwagi:
  - dostęp zdalny daje elastyczność
  - dostęp lokalny efektywniejszy
  - wpływ rozproszenia na wydajność nie jest jednoznaczny
  - interfejs biznesowy może być jednocześnie oznaczony jako @Local i @Remote

W celu podjęcia decyzji czy komponent EJB powinien umożliwiać dostęp zdalny czy lokalny, należy odpowiedzieć sobie na kilka pytań.

Po pierwsze, czy związki między poszczególnymi komponentami EJB są ścisłe czy raczej luźne? Jeśli komponenty często wywołują nawzajem swoje metody w celu realizacji złożonego zadania, odpowiedni może być dla nich dostęp lokalny, ze względu na większą efektywność komunikacji.

Po drugie, jakiego typu klienci będą korzystać z komponentu? Jeśli klienci aplikacyjni, to komponent musi pozwalać na dostęp zdalny. Jeśli tylko komponenty, które mogą być zainstalowane na tym samym serwerze aplikacji, to można rozważyć dostęp lokalny.

Po trzecie, czy szczególnie istotna jest skalowalność aplikacji? Jeśli tak, to dostęp zdalny daje więcej możliwości, gdyż umożliwia w razie potrzeby rozproszenie komponentów między serwerami aplikacji.

Ogólnie można stwierdzić, że dostęp zdalny daje większą elastyczność. W roli zdalnego klienta może wystąpić klient działający na tej samej maszynie wirtualnej, a dostęp zdalny umożliwi łatwą zmianę lokalizacji komponentów względem siebie gdy zajdzie taka potrzeba. Zaletą dostępu lokalnego jest większa efektywność.

Analizując wpływ rozproszenia komponentów na wydajność aplikacji, należy stwierdzić, że nie jest on jednoznaczny. Z jednej strony, rozproszenie zmniejsza obciążenie serwerów i zwiększa całkowitą dostępną moc obliczeniową. Z drugiej strony, umieszczenie komponentów aplikacji na tym samym serwerze redukuje koszty komunikacji i umożliwia efektywny dostęp lokalny do EJB.

Na zakończenie dyskusji należy zwrócić uwagę, że interfejs biznesowy komponentu EJB może mieć dwie adnotacje: @Remote i @Local, choć jest to rzadko stosowane.

Uzyskanie referencji do EJB przez klienta - dependency injection

- Domyślna nazwa w JNDI
  - klient typu serwlet/JSP/EJB

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

  - klient aplikacyjny

```
@EJB
static Konwerter konw;
```

- Dowolna nazwa w JNDI
  - klient typu serwlet/JSP/EJB

```
@EJB(name="ejb/Konwerter")
Konwerter konw;
```

  - klient aplikacyjny

```
@EJB(name="ejb/Konwerter")
static Konwerter konw;
```

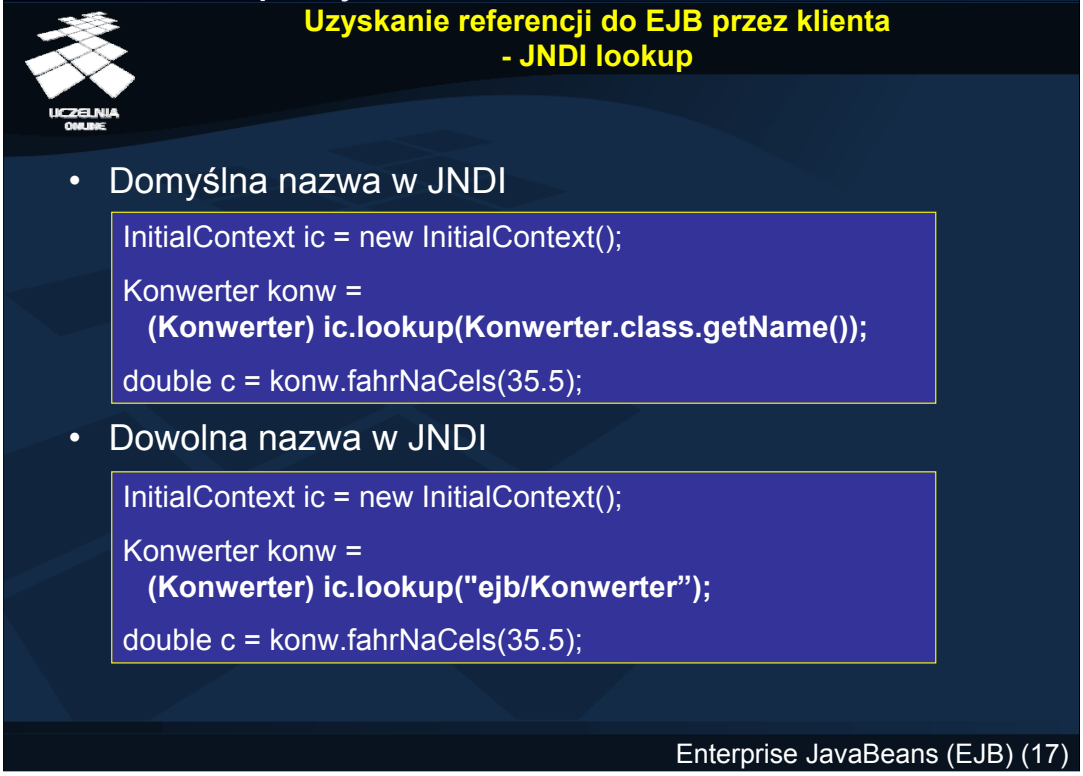

Gdy sesyjny komponent EJB jest instalowany (ang. deployed) w kontenerze EJB, jego interfejs biznesowy jest rejestrowany w rejestrze JNDI kontenera. Istnieją dwa sposoby uzyskania przez klienta referencji do komponentu EJB: wstrzyknięcie referencji mechanizmem dependency injection lub jawne wyszukanie komponentu w rejestrze JNDI operacją lookup. Pierwszy, nowszy sposób jest przedstawiony na niniejszym slajdzie, drugi będzie zaprezentowany na slajdzie następnym. Niezależnie od sposobu jej uzyskania, referencja do komponentu jest typu interfejsu biznesowego komponentu (zdalnego lub lokalnego).

W przypadku wstrzykiwania referencji mechanizmem dependency injection do pola w klasie klienta, należy pamiętać o tym, że w przypadku klienta aplikacyjnego pole w klasie, do którego będzie wstrzykiwana referencja musi być statyczne. Jest to spowodowane tym, że działanie klienta aplikacyjnego rozpoczyna się w kontekście statycznym (od statycznej metody main()).

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”.

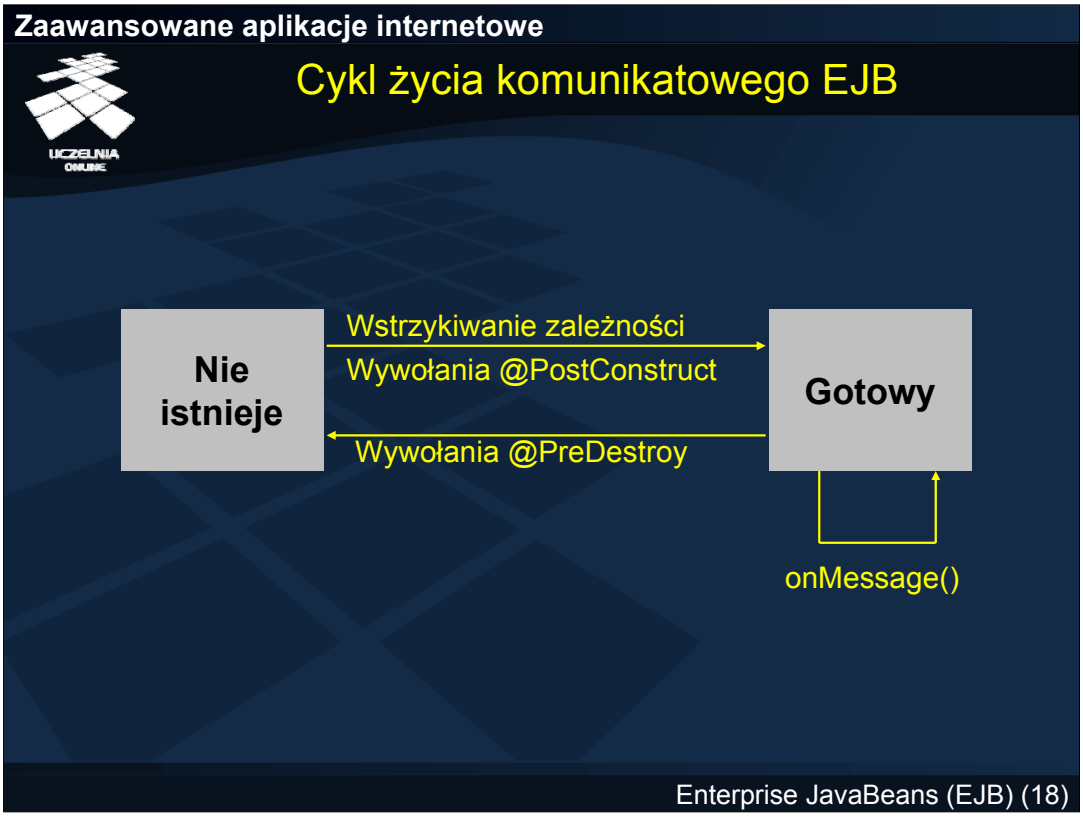
Po uzyskaniu referencji do komponentu, klient korzysta z niego wywołując na rzecz referencji metody zawarte w interfejsie biznesowym, co zostało zilustrowane w pierwszym przykładowym fragmencie kodu na slajdzie.



Slajd pokazuje sposób uzyskania referencji do komponentu EJB przez klienta dowolnego typu poprzez jawne wyszukanie referencji w rejestrze JNDI operacją lookup. W celu uzyskania dostępu do rejestru JNDI klient tworzy obiekt InitialContext. Bezargumentowy konstruktor InitialContext łączy klienta z rejestrem JNDI kontenera, w którym wykonuje się klient.

W przypadku gdy komponent jest zarejestrowany w JNDI pod domyślną nazwą, parametrem metody lookup() obiektu InitialContext jest w pełni kwalifikowana nazwa klasy interfejsu biznesowego komponentu. Można ją uzyskać wywołując metodę getName() na rzecz atrybutu „class” interfejsu, co ilustruje pierwszy fragment kodu na slajdzie.

W przypadku gdy komponent został zarejestrowany w JNDI pod inną nazwą niż domyślna, należy tę nazwę podać jako parametr metody lookup() obiektu InitialContext, co ilustruje drugi fragment kodu na slajdzie.



Podobnie jak komponenty sesyjne bezstanowe, komponenty komunikatowe nie podlegają pasywacji i ich cykl życia obejmuje tylko dwa stany: instancja komponentu albo nie istnieje albo jest gotowa do użycia. Kontener zazwyczaj przygotowuje pulę gotowych instancji komponentów komunikatowych i przydziela je do obsługi przychodzących komunikatów. Po utworzeniu instancji, a przed jej wykorzystaniem, kontener wstrzykuje zależności, a następnie wywołuje metody oznaczone adnotacją @PostConstruct. W stanie gotowości instancja może być przydzielona do obsługi komunikatu. Wywoływana wtedy jest jej metoda onMessage(). Po obsłużeniu komunikatu instancja wraca do puli i może w przyszłości obsłużyć kolejne komunikaty. Gdy instancja komponentu nie jest potrzebna, kontener może podjąć decyzję o jej usunięciu. Przed udostępnieniem instancji dla mechanizmu garbage collection, kontener wywołuje metody oznaczone adnotacją @PreDestroy.

Struktura kodu komunikatowego EJB

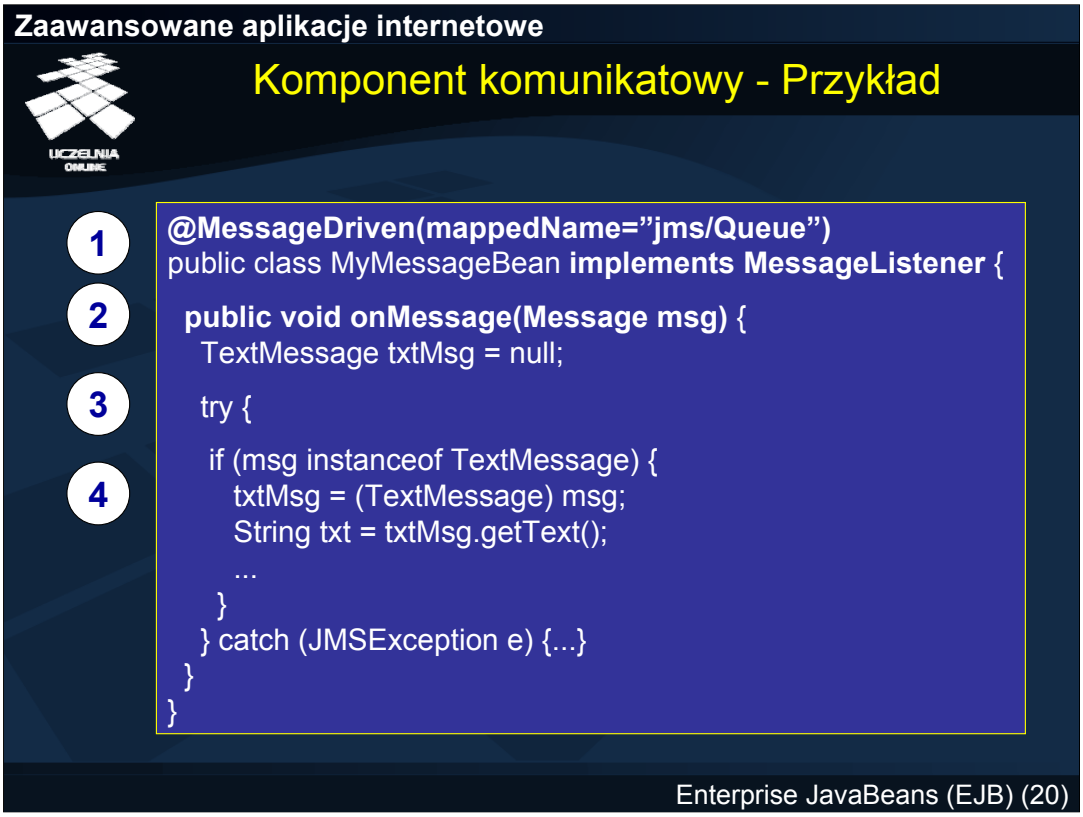
- Komunikatowe EJB nie są bezpośrednio wywoływane przez klientów, więc nie posiadają interfejsów biznesowych
- Kod komunikatowego EJB stanowi jedynie klasa komponentu:
  - implementująca javax.jms.MessageListener
  - posiadająca implementację metody onMessage()
  - oznaczona adnotacją @MessageDriven
  - posiadająca publiczny bezargumentowy konstruktor

Cechą wyróżniającą kod komunikatowych EJB od kodu sesyjnych EJB jest brak interfejsów biznesowych. Komunikatowy EJB ich nie posiada, gdyż nie jest bezpośrednio wykorzystywany przez klientów.

Kod komunikatowego EJB stanowi jedynie klasa komponentu. Klasa ta musi implementować interfejs javax.jms.MessageListener. Interfejs ten zawiera metodę onMessage(), której implementacja musi w związku z tym być zawarta w klasie komponentu. Metoda ta będzie wywoływana przez kontener w odpowiedzi na nadejście komunikatu.

Klasa komunikatowego EJB jest oznaczana adnotacją @MessageDriven, zawierającą nazwę JNDI miejsca przeznaczenia komunikatów (np. kolejki), z którego komponent ma obsługiwać komunikaty. Alternatywą dla adnotacji jest wykorzystanie deskryptora instalacji EJB w formie pliku XML.

Klasa komunikatowego EJB dodatkowo musi spełniać szereg warunków tj. musi posiadać publiczny bezargumentowy konstruktor, musi być publiczna, nie może być abstract ani final, nie może mieć zdefiniowanej metody finalize().



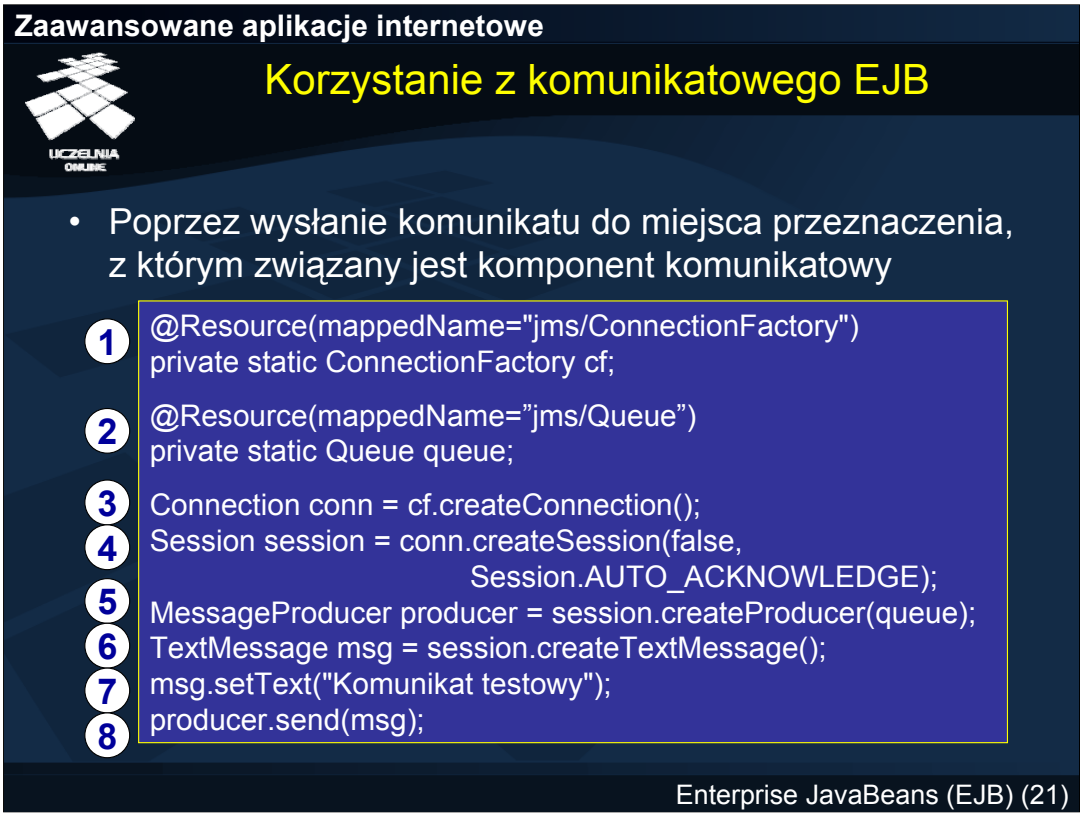
Slajd pokazuje przykład kodu komunikatowego komponentu EJB (pominięte zostały dyrektywy import). Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Klasa jest oznaczona adnotacją @MessageDriven, zawierającą nazwę kolejki, z której komponent będzie obsługiwał komunikaty. Klasa implementuje interfejs MessageListener.

2. Klasa posiada metodę onMessage() przewidzianą przez interfejs MessageListener. Metoda ta zawiera kod przetwarzający odebrany komunikat.

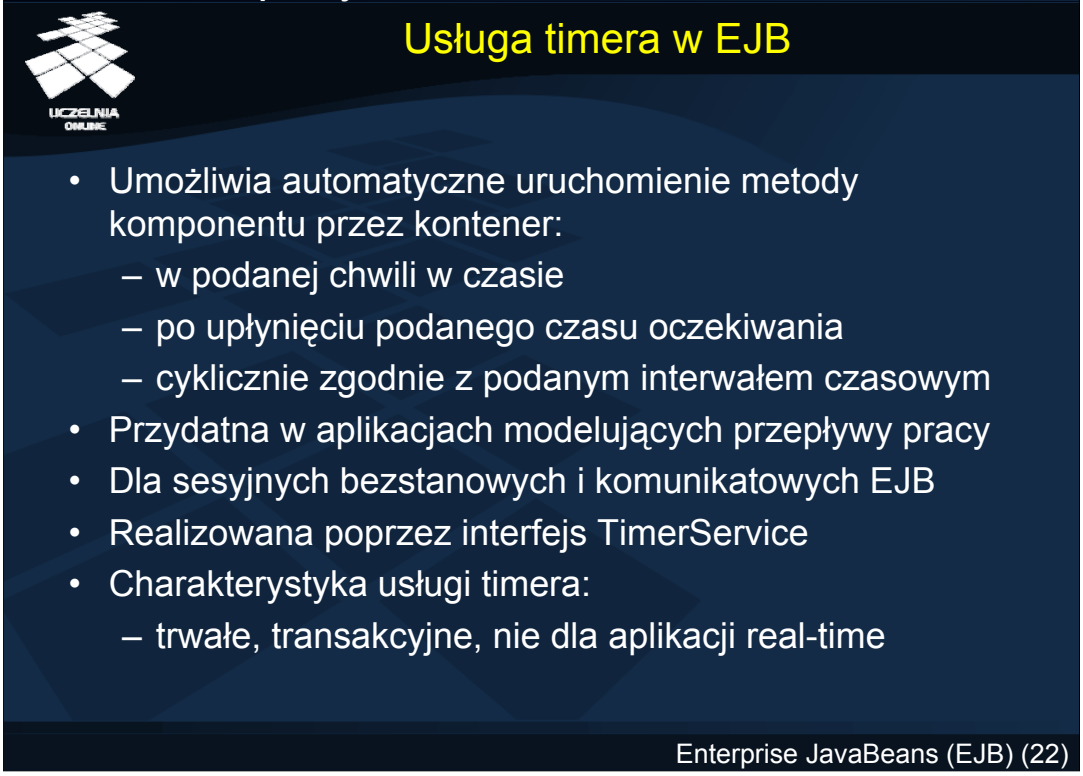
3. Operacja odczytu treści komunikatu może rzucić wyjątek JMSException. Stąd blok try z sekcją catch przechwytującą ten wyjątek.

4. Przykładowy komponent ma przetwarzać przychodzące komunikaty tekstowe. Najpierw sprawdzany jest typ odebranego komunikatu i jeśli jest on typu TextMessage, to metodą getText() odczytywana jest z niego treść komunikatu, która następnie może być wykorzystana zgodnie z przeznaczeniem komponentu.



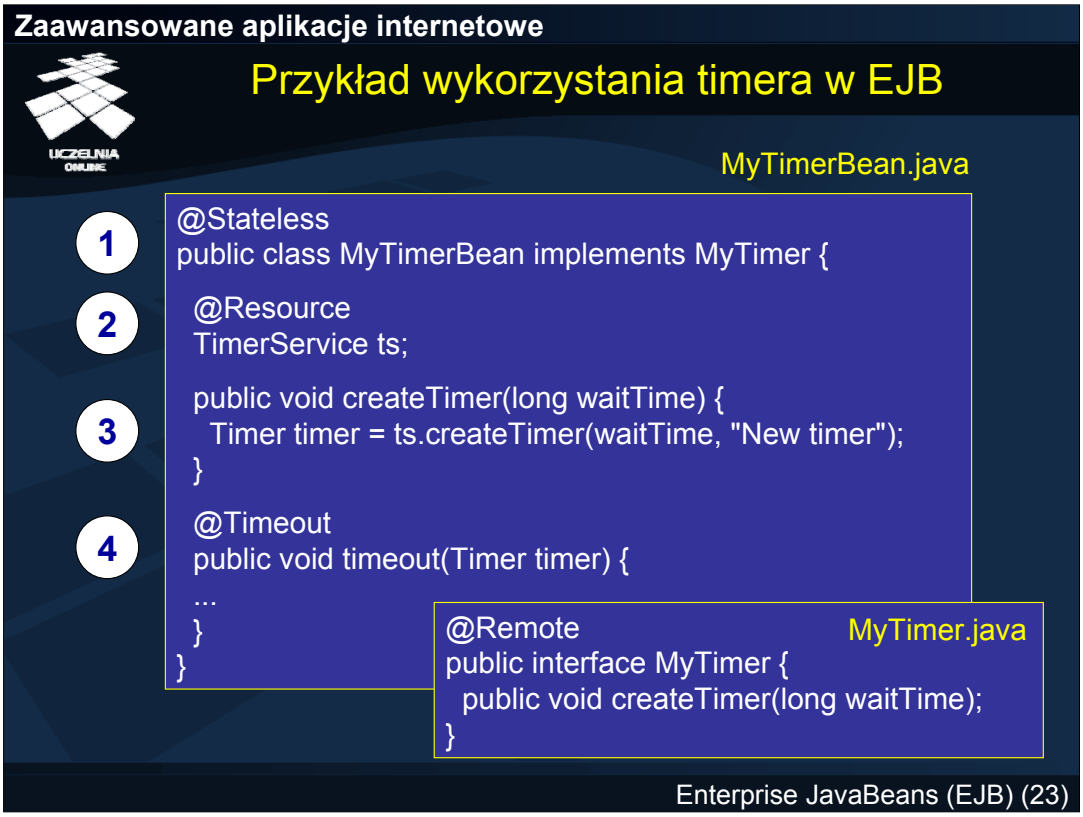
Klient korzysta z komunikatowego komponentu EJB poprzez wysłanie komunikatu do miejsca przeznaczenia, z którym związany jest komponent. Slajd pokazuje przykład wysłania komunikatu tekstowego do kolejki. Kod klienta w żaden sposób nie wskazuje na to, że odbiorcą komunikatu będzie komunikatowy EJB. Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Wstrzyknięcie wskazanego obiektu ConnectionFactory, poprzez który klient uzyska połączenie z systemem przekazywania komunikatów.
2. Wstrzyknięcie obiektu reprezentującego wskazaną kolejkę komunikatów.
3. Utworzenie połączenia z systemem przekazywania komunikatów.
4. Utworzenie sesji w ramach połączenia.
5. Utworzenie obiektu producenta komunikatów dla wskazanej kolejki.
6. Utworzenie komunikatu typu tekstowego.
7. Umieszczenie tekstu w komunikacie.
8. Wysłanie komunikatu do kolejki.



Technologia EJB udostępnia usługę timera, umożliwiającą automatyczne uruchomienie metody komponentu przez kontener: w podanej chwili w czasie, po upłynięciu podanego czasu oczekiwania lub cyklicznie zgodnie z podanym interwałem czasowym. Usługa timera może być przydatna np. w aplikacjach modelujących przepływy pracy i jest dostępna dla sesyjnych bezstanowych i komunikatowych EJB. Ustawianie timera w kodzie metody komponentu EJB jest realizowane poprzez interfejs TimerService.

Timery w EJB są trwałe tj. odporne na wyłączenia i awarie serwera. Operacje ustawiania i kasowania timera są transakcyjne - jeśli stanowią element transakcji, która jest wycofywana, zostaną również anulowane. Mimo, że czas dla timera jest specyfikowany z dokładnością do milisekundy, taka precyzja momentu ich uruchomienia nie jest gwarantowana, przez co usługa timera w EJB nie jest odpowiednia dla aplikacji czasu rzeczywistego.



Slajd pokazuje przykład wykorzystania timera w sesyjnym bezstanowym komponencie EJB. U góry pokazany jest kod źródłowy klasy komponentu, a u dołu kod źródłowy interfejsu biznesowego. Znaczenie wyróżnionych fragmentów kodu klasy komponentu jest następujące:

1. Klasa komponentu jest oznaczona adnotacją @Stateless i implementuje swój interfejs biznesowy. Interfejs przewiduje tylko jedną metodę udostępnianą klientom: createTimer().

2. Wstrzyknięcie obiektu TimerService, poprzez który metoda komponentu ustawi timer.

3. Metoda biznesowa createTimer(). Jej działanie sprowadza się do ustawienia jednorazowego timera, który wygaśnie po przekazanej jako parametr metody liczbie milisekund. Do ustawiania timera służy przeciążona metoda createTimer() interfejsu TimerService. W przykładzie została użyta jedna z jej wersji.

4. Metoda, która będzie wywołana po wygaśnięciu timera. Taka metoda musi być oznaczona adnotacją @Timeout, posiadać jeden parametr typu Timer i typ zwrotny void.

Transakcje w EJB

- Zarządzane przez kontener (CMT)
  - specyfikowane deklaratywnie: atrybut transakcyjny
  - cały kod metody w jednej transakcji
- Zarządzane przez komponent (BMT)
  - granice transakcji wyznaczane programowo
  - transakcje JDBC
    - w ramach połączenia JDBC
  - transakcje JTA (Java Transaction API)
    - zarządzane przez zarządcę transakcji Java EE
    - mogą być rozproszone

Transakcje w EJB mogą być zarządzane przez kontener (CMT – ang. container-managed transactions) lub przez komponent (BMT – ang. bean-managed transactions).

Granice transakcji w transakcjach zarządzanych przez kontener są specyfikowane deklaratywnie poprzez tzw. atrybut transakcyjny. Atrybut ten określa sposób zachowania się metody komponentu EJB względem transakcji. Cały kod metody zawsze wykonuje się w ramach jednej transakcji. Jedna transakcja może obejmować działanie wielu metod różnych komponentów.

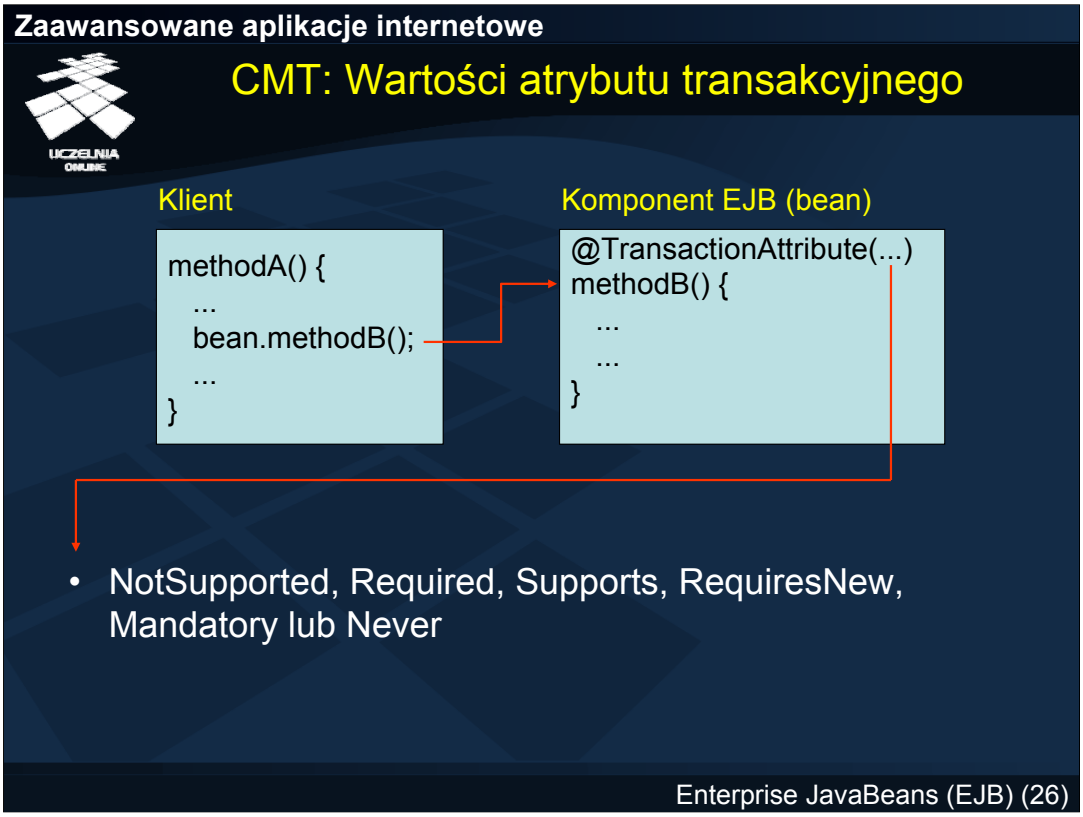
W transakcjach zarządzanych przez komponent granice transakcji są wyznaczane programowo poprzez wywołania odpowiednich metod – tak jak w serwletach. Podobnie jak w serwletach, transakcje zarządzane przez komponent mogą być realizowane poprzez interfejs JDBC lub JTA (Java Transaction API). Transakcje JDBC odbywają się w ramach otwartego połączenia z bazą danych (poprzez obiekt Connection). Transakcje JTA są zarządzane przez zarządcę transakcji Java EE w ramach serwera aplikacji i są „odporne” na zamykanie i otwieranie połączeń z bazą danych w trakcie ich trwania. JTA umożliwia też realizację transakcji rozproszonych.

CMT: Atrybut transakcyjny

- Specyfikowany za pomocą adnotacji @TransactionAttribute na poziomie metody lub klasy komponentu EJB

```
@TransactionAttribute(REQUIRED)
@Stateful
public class MyTransBean implements MyTrans {

  public void methodA {...}

 @TransactionAttribute(REQUIRES_NEW)
  public void methodB {...}

  public void methodC {...}

}
```


Atrybut transakcyjny dla transakcji zarządzanych przez kontener jest specyfikowany za pomocą adnotacji @TransactionAttribute na poziomie metody lub klasy komponentu EJB. Atrybut na poziomie klasy odnosi się do wszystkich metod komponentu, dla których nie podano jawnie innej wartości atrybutu.



Atrybut transakcyjny określa czy i w jakiej transakcji ma się wykonać wywołana metoda komponentu EJB, w zależności od tego czy klient wywołujący metodę jest w trakcie transakcji czy nie. Dopuszczalne wartości atrybutu transakcyjnego dla metod komponentu EJB to: NotSupported, Required, Supports, RequiresNew, Mandatory i Never. Znaczenie poszczególnych wartości jest następujące:

NotSupported – jeśli klient nie jest w transakcji, to kontener nie rozpoczyna nowej transakcji przed wywołaniem metody komponentu. Jeśli klient jest w transakcji, to transakcja jest zawieszana na czas działania metody.

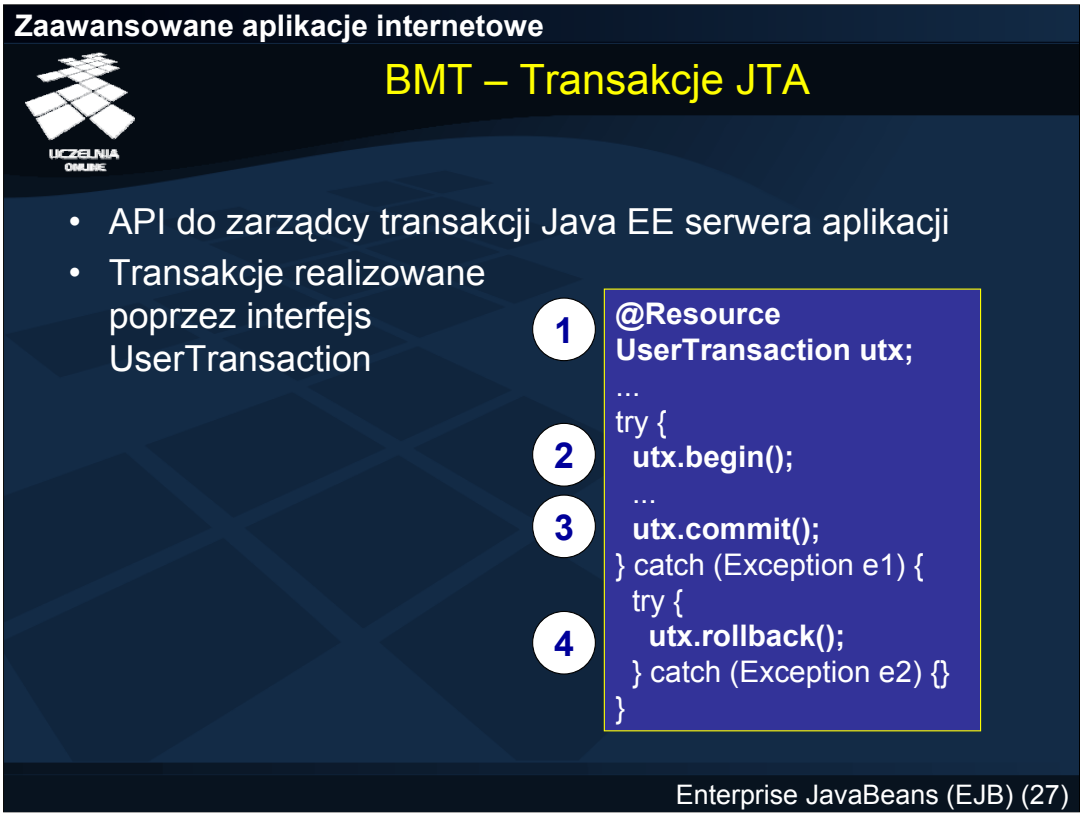
Required - jeśli klient nie jest w transakcji, to kontener rozpoczyna nową transakcję przed wywołaniem metody komponentu. Jeśli klient jest w transakcji, to metoda komponentu wykona się w ramach tej transakcji.

Supports – jeśli klient nie jest w transakcji, to kontener nie rozpoczyna nowej transakcji przed wywołaniem metody komponentu. Jeśli klient jest w transakcji, to metoda komponentu wykona się w ramach tej transakcji.

RequiresNew – kontener rozpoczyna nową transakcję przed wywołaniem metody komponentu niezależnie od tego czy klient jest w transakcji czy nie. Jeśli klient jest w transakcji, to transakcja klienta jest zawieszana na czas wykonywania metody komponentu.

Mandatory – jeśli klient nie jest w transakcji, to kontener zgłasza wyjątek. Jeśli klient jest w transakcji, to metoda komponentu wykona się w ramach tej transakcji.

Never – jeśli klient nie jest w transakcji to metoda komponentu wykona się poza transakcją. Jeśli klient jest w transakcji, to kontener zgłasza wyjątek.



Transakcje, których granice są wyznaczane programowo przez komponent mogą być realizowane poprzez JTA lub JDBC. JTA (Java Transaction API) to interfejs programistyczny pozwalający aplikacji Java EE na wyznaczanie granic transakcji w sposób niezależny od konkretnej implementacji zarządcy transakcji w ramach serwera aplikacji.

Transakcje JTA są realizowane poprzez zasób UserTransaction, stanowiący interfejs do zarządcy transakcji. Schemat korzystania z transakcji JTA został pokazany na slajdzie. Znaczenie wyróżnionych fragmentów kodu jest następujące:

1. Wstrzyknięcie zasobu UserTransaction.

2. Rozpoczęcie transakcji poprzez wywołanie metody begin() na rzecz obiektu UserTransaction.

3. Zatwierdzenie transakcji poprzez wywołanie metody commit() na rzecz obiektu UserTransaction.

4. Wycofanie transakcji poprzez wywołanie metody rollback() na rzecz obiektu UserTransaction w wypadku przechwycenia wyjątku rzuconego przez kod transakcji. Operacja wycofywania transakcji też może rzucić wyjątek, więc została umieszczona w zagnieżdżonym bloku try.

Podsumowanie

- Technologia EJB umożliwia implementację logiki biznesowej w aplikacjach Java EE
- Komponenty EJB są uruchamiane w kontenerze EJB
- Dwa typy komponentów: sesyjne i komunikatowe
- Zalety EJB:
  - skalowalność aplikacji
  - wsparcie dla różnych typów klientów
  - silne wsparcie dla transakcji

Technologia EJB umożliwia implementację logiki biznesowej w aplikacjach Java EE. Komponenty EJB są uruchamiane w kontenerze EJB, który stanowi element serwera aplikacji i oferuje komponentom gotowe implementacje usług przydatnych w złożonych aplikacjach internetowych.

Obecnie istnieją dwa typy komponentów EJB: sesyjne i komunikatowe. Komunikacja klientów z sesyjnymi EJB odbywa się na zasadzie wywoływania ich metod biznesowych. Komunikatowe EJB są asynchronicznymi konsumentami komunikatów wysyłanych do systemów przekazywania komunikatów.

Wykorzystanie EJB jest zalecane gdy aplikacja ma być skalowalna, gdy komponenty zawierające logikę biznesową mają być dostępne dla klientów różnych typów lub gdy aplikacja realizuje zaawansowane przetwarzanie transakcyjne.

Materiały dodatkowe

- The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/

•The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/