Enterprise JavaBeans 3.0

Inżynieria Oprogramowania
Altkom Akademia

Plan szkolenia

Plan modułu

1. **Nowości w EJB 3.1**
   - interfejsy biznesowe
   - przenośne nazwy JNDI
   - singletony
   - harmonogramowanie
   - asynchroniczne beany sesyjne
   - uproszczone pakowanie aplikacji

Interfejsy biznesowe – dostęp lokalny

- Bean EJB dostępny **lokalnie** może:
  - implementować interfejs opisany adnotacją `@Local`
  - implementować zwykły interfejs i być opisanym adnotacją `@Local` wskazującą klasę interfejsu, np.:

```
@Stateless
@Local(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

  - **nie implementować interfejsu biznesowego**, ale być opisanym adnotacją `@javax.ejb.LocalBean`, np.:

```
@Stateless  // lub @Stateful
@LocalBean
public class BeanName { ... }
```

Interfejsy biznesowe – dostęp zdalny

- Bean EJB dostępny **zdalnie** może:
  - implementować interfejs opisany adnotacją `@Remote`
  - implementować zwykły interfejs i być opisanym adnotacją `@Remote` wskazującą klasę interfejsu, np.:

```
@Stateless
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

Nazwy JNDI

- Specyfikacja EJB 3.1 definiuje trzy przestrzenie nazw, które określają ich zasięgi:
  - globalny
  - aplikacji
  - modułu
- Kontener zgodny ze specyfikacją rejestruje beany sesyjne za pomocą nazw JNDI o określonej strukturze
- Tak ustandaryzowane nazwy **są przenośne**, dzięki czemu beany mogą ustanawiać referencje do innych beanów EJB na innych serwerach aplikacyjnych

Nazwy JNDI

- **Nazwy w globalnej przestrzeni nazw** będą dostępne dla kodu w dowolnej aplikacji
- Mają postać:

  java:global[/<app-name>]/<module-name>/<bean-name>[!<interface-FQN>]
  - **<app-name>** – nazwa aplikacji (lub EAR)
  - **<module-name>** – nazwa modułu (JAR lub WAR)
- Kontener musi zarejestrować jedną globalną nazwę dla każdego interfejsu lokalnego i zdalnego oraz widoku bez interfejsu
- Jeżeli sesyjny EJB posiada tylko jeden interfejs biznesowy lub jest widokiem bez interfejsu, kontener zarejestruje go za pomocą nazwy:

  java:global[/<app-name>]/<module-name>/<bean-name>

Nazwy JNDI

- **Nazwy w przestrzeni nazw aplikacji** będą dostępne tylko dla kodu w tej samej aplikacji
- Mają postać:

  java:app/<module-name>/<bean-name>[!<interface-FQN>]
- Stosuje się te same reguły publikacji nazw, co poprzednio (dla nazw globalnych)

Nazwy JNDI

- **Nazwy w przestrzeni nazw modułu** będą dostępne tylko dla kodu w tym samym module
- Mają postać:

  java:module/<bean-name>[!<interface-FQN>]
- Stosuje się te same reguły publikacji nazw, co poprzednio (dla nazw globalnych)

Nazwy JNDI

- Warto zauważyć, że mimo, iż globalne nazwy JNDI dla lokalnych interfejsów i widoków bez interfejsów są publikowane, to nie oznacza to, że taki interfejs będzie mógł być dostępny dla komponentów działających w innej JVM

Singletony

- **Singleton** to nowy typ komponentu sesyjnego
- Jest klasą POJO, dla której kontener gwarantuje utworzenie pojedynczej, współdzielonej instancji
- Singletony są bezpieczne wielowątkowo i korzystają z transakcji
- Są dla nich dostępne wszelkie usługi, jak dla innych komponentów (bezpieczeństwo, dostęp zdalny, DI, itp.)
- Są opisywane adnotacją **`@javax.ejb.Singleton`**:

```
@Singleton
public class SingletonBean {
  // ...
}
```

Singletony

Uwaga!

- Specyfikacja EJB 3.1 stwierdza, że każda aplikacja będzie miała jedną instancję singletona dla każdej maszyny wirtualnej
- Tak więc w środowisku klastrowym będzie jedna instancja na każdy węzeł klastra, a nie jedna dla całego klastra :-)

Współbieżny dostęp

- Specyfikacja EJB 3.1 umożliwia proste zarządzanie współbieżnym dostępem do stanu, z użyciem adnotacji
  `@javax.ejb.ConcurrencyManagement`
- Można jej podać jedną ze stałych z klasy *ConcurrencyManagementType*:
  - *CONTAINER*
  - *BEAN*
  - *CONCURRENCY_NOT_SUPPORTED*
- Współbieżny dostęp do singletonów kontroluje domyślnie kontener
- Każda metoda domyślnie uzyskuje blokadę zapisu (*write lock*) – może to powodować pogorszenie wydajności

Współbieżny dostęp

- Można samodzielnie określić dostęp do metod beana, dzięki użyciu adnotacji **`@javax.ejb.Lock`**:

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class SingletonBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() {
    return state;
  }

  @Lock(LockType.WRITE)
  public void setState(String state) {
    this.state = state;
  }
}
```

Współbieżny dostęp

- Żądania nie powinny się blokować ze względu na zbyt długie oczekiwanie na zwolnienie blokady
- Można określić maksymalny czas oczekiwania za pomocą adnotacji **`@javax.ejb.AccessTimeout`**, np.:

```
@Lock(LockType.READ)
@AccessTimeout(timeout=15, unit=TimeUnit.SECONDS)
public String concurrentReadOnlyMethod(){
  // ...
}
```

- Upłynięcie zadanego czasu spowoduje wyrzucenie wyjątku *javax.ejb.ConcurrentAccessTimeoutException*

Współbieżny dostęp

- W trybie zarządzania współbieżnością *ConcurrencyManagement.BEAN* programista musi zapewnić bezpieczeństwo wielowątkowe samodzielnie
- Może wykorzystać dowolne mechanizmy, np. *synchronized*, *volatile*, typy atomowe, itd.

Nasłuch startu i kończenia aplikacji

- Wprowadzenie singletonów umożliwiło prosty sposób realizacji metod zwrotnych uruchamianych przy starcie i kończeniu aplikacji EJB
- Domyślnie kontener decyduje o momencie instancjonowania singletona
- Można wymusić utworzenie instancji singletona w czasie inicjalizacji aplikacji poprzez użycie adnotacji `@javax.ejb.Startup`
- To umożliwia beanowi zdefiniowanie:
  - metody `@PostConstruct` wykonywanej w trakcie jego inicjalizacji
  - metody `@PreDestroy` wykonywanej, gdy aplikacja będzie zamykana

Nasłuch startu i kończenia aplikacji – przykład

```
@Singleton
@Startup
public class PropertiesBean {

   @PostConstruct
   private void startup() { ... }

   @PreDestroy
   private void shutdown() { ... }

   ...

}
```

Kolejność inicjalizacji singletonów

- Kolejnością inicjalizacji singletonów można sterować za pomocą adnotacji `@javax.ejb.DependsOn`
- Można w ten sposób wprowadzić zależności między singletonami
- Adnotacją opisujemy bean zależny, tzn. ten, który ma być zainicjalizowany jako ostatni
- Za pomocą jednej adnotacji można wskazać kilka zależnych beanów

```
// beany PlayerBean i EnemyBean musza byc zainicjowane
// przed GameBean (w dowolnej kolejnosci)
@Singleton
@DependsOn("PlayerBean, EnemyBean")
public class GameBean {
  // ...
}
```

Kolejność inicjalizacji singletonów – przykład

```
@Singleton              // inicjowany jako trzeci
@DependsOn("EnemyBean")
public class GameBean {
  // ...
}

@Singleton              // inicjowany jako drugi
@DependsOn("PlayerBean")
public class EnemyBean {
  // ...
}

@Singleton              // inicjowany jako pierwszy
public class PlayerBean {
  // ...
}
```

Harmonogramowanie

- Usługa *TimerService* umożliwia planowanie zadań/powiadomień w określonych chwilach czasowych dla wszystkich typów beanów EJB oprócz stanowych
- Nowością jest możliwość deklaratywnego planowania zadań (uruchamiania metod) w stylu *cron'a*
- Przydatna jest adnotacja **`@javax.ejb.Schedule`**:

```
@Stateless
public class TimerBean {
  @Schedule(minute="15", hour="6", dayOfWeek="Mon")
  public void sendNotification() {
    // ...
  }
}
```

Harmonogramowanie

- Adnotacja `@Schedule` może przyjąć następujące atrybuty określające datę i czas:

Atrybut	Dopuszczalne wartości	Wartość domyślna
second	$[0, 59]$	0
minute	$[0, 59]$	0
hour	$[0, 23]$	0
dayOfMonth	$[1, 31]$ $[-7, -1]$, (*x dni przed ostatnim dniem miesiąca*) "Last" (*ostatni dzień miesiąca*)	*
month	$[1, 12]$ {"Jan", "Feb", . . . , "Dec"}	*
dayOfWeek	$[0, 7]$ {"Sun", "Mon", . . . , "Sat"} (*0 i 7 oznaczają "Sun"*)	*
year	*rok 4-cyfrowy*	*

Harmonogramowanie

- Każdy atrybut może przyjąć następujące formaty:
  - pojedyncze wartości, np.:
    - *second = "10"*
    - *month = "Sep"*
  - dzikie karty (* oznacza wszystkie dopuszczalne wartości)
  - listy wartości (wyliczenia), np.:
    - *second = "10,20,30"*
    - *dayOfWeek = "Mon,Wed,Fri"*
  - zakresy, np.:
    - *dayOfMonth = "27-3" (oznacza: "27-Last, 1-3")*
  - inkrementacje – wyrażenia postaci *x/y*, gdzie *x* oznacza wartość początkową, a *y* oznacza przyrost wartości w danej jednostce czasu, np.:
    - *second = "30/10"* oznacza *"30,40,50"*
    - *minute = "*/5"* oznacza każdą co piątą minutę w godzinie

Harmonogramowanie

- Definiując harmonogramowanie należy pamiętać o wartościach domyślnych, np.:

```
@Schedule(second="*/5", info="Every 5 seconds")
```

nie oznacza bynajmniej zadania uruchamianego co 5 sekund...

- Faktycznie zadanie zostanie uruchomione o północy następnego dnia (zerowa minuta i godzina)

Harmonogramowanie

- Adnotacje **`@Schedule`** można grupować za pomocą adnotacji **`@javax.ejb.Schedules`**, np.:

```
@Schedules({
  @Schedule(hour="*", minute="*", second="*/15"),
  @Schedule(hour="*", minute="*", second="*/25")
})
```

Asynchroniczne EJB

- Asynchroniczne wykonywanie metod umożliwiały dotychczas beany MDB – jednakże ich użycie wymagało konfiguracji JMS
- Teraz można użyć adnotacji **`@javax.ejb.Asynchronous`** do wskazania, że EJB ma zwrócić sterowanie natychmiast po wywołaniu metody, np.:

```
@Stateless
public class AsynchBean {
  @Asynchronous
  public void fireAction(Customer c) {
    try {
      sendInvoice(c);
      sendEmail(c);
    } catch (Exception e) { /* ... */ }
  }
}
```

Asynchroniczne EJB

- W przypadku, gdy klient jest zainteresowany wynikiem z metody wykonywanej asynchronicznie, może do odebrania wyniku wykorzystać interfejs *java.util.concurrent.Future*
- Wystarczy wynik opakować w implementację tego interfejsu o nazwie *javax.ejb.AsynchResult*
- Przykład:

```
@Asynchronous
public Future<Integer> calcAsync(int input) {
  // obliczenia ...
  Integer result = ...
  return new AsyncResult<Integer>(result);
}
```

Asynchroniczne EJB

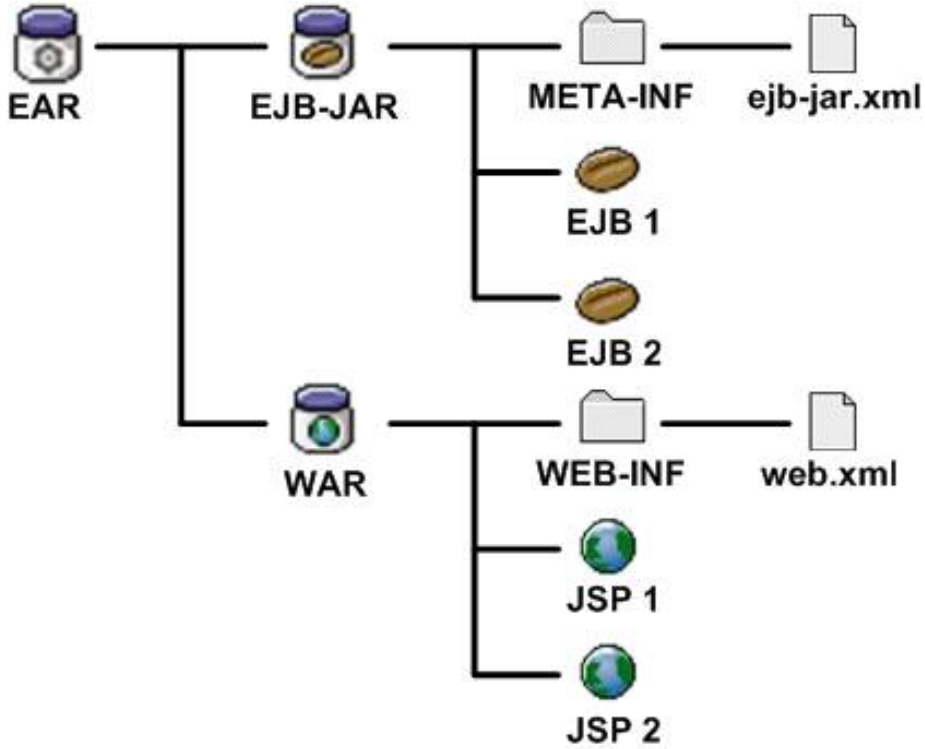
- Wywołanie asynchroniczne i odebranie wyniku (cd. przykładu):

```
Future<Integer> calcFuture = ejb.calcAsync(input);
// teraz mozna zajac sie czyms innym
// ...
// nadszedl czas, aby odebrac wynik
// czekamy maks. 10s
Integer result = calcFuture.get(10, TimeUnit.SECONDS);
```

- Interfejs *Future* umożliwia także anulowanie zadania

Pakowanie aplikacji

- Nawet najprostsza aplikacja EJB 3.0 wymagała, aby:
  - moduł webowy był spakowany do pliku *.war*
  - beany EJB znalazły się w pliku *.jar*
  - oba powyższe spakowane do *.ear* i wdrożone na serwerze

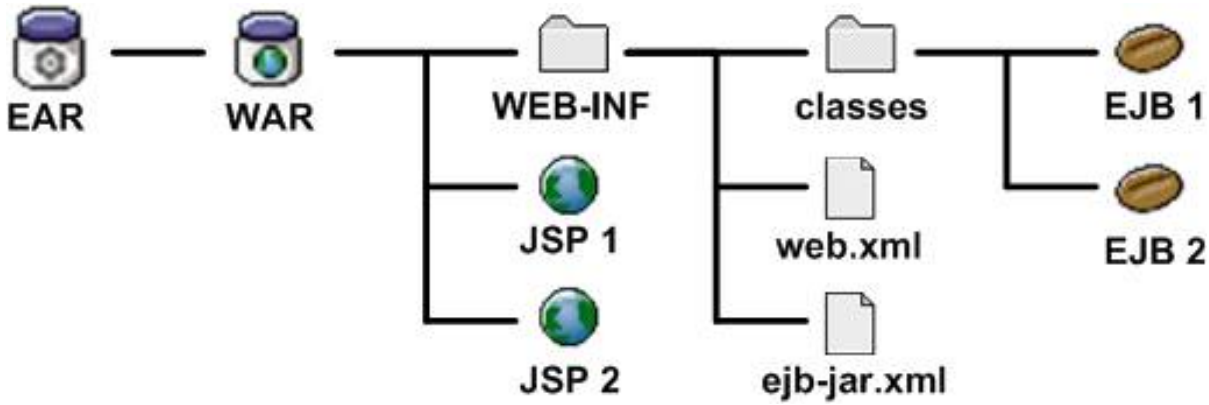


Pakowanie aplikacji

- U podstaw takiego modelu pakowania jest idea, że moduły EJB reprezentują zmodularyzowane usługi biznesowe, które są konsumowane przez klientów – moduły webowe
- Tego typu modularyzacja nie ma uzasadnienia (zbędny narzut) w przypadku prostych aplikacji webowych, dla których jest mało prawdopodobne, aby usługi biznesowe były współdzielone przez wiele innych modułów JEE
- Uproszczone pakowanie EJB dla aplikacji webowych wychodzi temu naprzeciw

Pakowanie aplikacji

- Teraz wystarczy beany EJB umieścić w katalogu *WEB-INF/classes* (tam gdzie serwlety) i wdrożyć jak aplikację webową
- Opcjonalny deskryptor *ejb-jar.xml* można umieścić w katalogu *WEB-INF* (tam, gdzie *web.xml*)



Pakowanie aplikacji

- Taki sposób pakowania jest wykorzystywany przez *EJB Lite*
- Dla wielu aplikacji technologia EJB oferuje więcej funkcjonalności niż one faktycznie potrzebują
- *EJB Lite* to okrojony (a więc lekki!) podzbiór możliwości EJB 3.1 przeznaczony do budowy niewielkich aplikacji
- *EJB Lite* oferuje:
  - beany sesyjne (stanowe, bezstanowe i singletony)
  - interfejsy lokalne EJB lub widoki bez interfejsów
  - klasy przechwytujące (interceptory)
  - transakcje CMT oraz BMT
  - bezpieczeństwo (deklaratywne i programowe)