SOA – laboratorium nr 4

Temat: Tworzenie EJB oraz aplikacji klienckich.

Celem tego zestawu ćwiczeń jest zapoznanie z sesyjnymi komponentami Enterprise JavaBeans. Zilustrowane będą różnice między komponentami stanowymi i bezstanowymi. Pokazane będzie tworzenie klientów aplikacyjnych i webowych.

Na początek proponuje zaznajomić się z dokumentacja znajdującą się pod tym adresem -> https://javaee.github.io/tutorial/partentbeans.html#BNBLR

Czym są EJB?

Według specyfikacji, Enterprise JavaBeans (EJB) to komponenty, których podstawowym zadaniem w aplikacjach Java Enterprise Edition (JEE) jest implementacja logiki biznesowej i dostępu do danych.

Najważniejsze fakty:

- Pracują po stronie serwera
- Zawierają logikę biznesową aplikacji
- Wykorzystywane do budowy złożonych aplikacji rozproszonych na zasadzie „składania z klocków”
- beany zarządzane przez kontener serwera aplikacji (kontener EJB oferujący usługi systemowe),
- kontener udostępnia programiście dodatkowe usługi jak zarządzanie transakcjami, bezpieczeństwem, itd,
- potrafi równoważyć obciążenie na kilka serwerów aplikacji (skalowalność),
- klientem dla EJB może być wiele typów aplikacji (nie tylko moduł war),
- bardzo dobrze nadają się do dużych aplikacji rozproszonych,
- dzięki uproszczonemu tworzeniu i wdrażaniu EJB dobrze wspomagają tworzenie mniejszych aplikacji.

W zasadzie istnieją trzy rodzaje komponentów EJB:

- **bezstanowe ziarna sesyjne** (SLSB — *Stateless Session Beans*) — obiekty, których instancje nie zawierają żadnych informacji o stanie konwersacji, więc gdy nie obsługują aktualnie konkretnego klienta, w zasadzie są sobie równoważne;
- **stanowe ziarna sesyjne** (SFSB — *Stateful Session Beans*) — obiekty obsługujące usługi konwersacyjne dotyczące silnie powiązanych klientów; stanowe ziarno sesyjne wykonuje zadania dla konkretnego klienta i przechowuje stan przez cały czas trwania sesji z klientem; po zakończeniu sesji stan nie jest dłużej przechowywany;

- **ziarna sterowane komunikatami** (MDB — *Message-Driven Beans*) — rodzaj komponentu EJB mogący asynchronicznie przetwarzać komunikaty przesyłane przez dowolnego producenta JMS

Poza standardowymi komponentami EJB, serwer aplikacji obsługuje również nowe odmiany EJB 3.2 wprowadzone wraz z Javą EE 6.

- **Singletonowy komponent EJB** — przypomina bezstanowe ziarno sesyjne, ale do obsługi żądań klientów wykorzystywana jest tylko jedna instancja, co gwarantuje użycie tego samego obiektu we wszystkich wywołaniach. Singletony mogą korzystać z bogatszego cyklu życia dla pewnego zbioru zdarzeń, a także ze ściślejszych zasad blokad, by prawidłowo obsłużyć współbieżny dostęp do instancji.
- **Bezinterfejsowy komponent EJB** — to nieco inne spojrzenie na standardowe ziarno sesyjne, bo od lokalnych klientów nie wymaga się osobnego interfejsu, czyli wszystkie metody publiczne klasy ziarna są dostępne dla kodu wywołującego.
- **Asynchroniczne komponenty EJB** — umożliwiają przetwarzanie żądań klientów w sposób asynchroniczny (podobnie jak w przypadku MDB), ale udostępniają typowany interfejs i stosują nieco bardziej wyrafinowane podejście do obsługi żądań klientów, które dzieli się na dwa etapy:

Sesyjne EJB mogą udostępniać dwa rodzaje interfejsów:

- lokalny:
    - może być spakowany w module EJB jak i w module webowym,
    - klienty muszą działać w tej samej maszynie wirtualnej co serwer,
    - mogą z nich korzystać aplikacje webowe i inne EJB,
    - ich położenie nie jest przezroczyste dla klienta;
- zdalny:
    - muszą być spakowane w module EJB,
    - klienty mogą działać w innej maszynie wirtualnej niż EJB,
    - mogą z nich korzystać aplikacje webowe, inne EJB, oddzielne aplikacje klienckie,
    - ich położenie jest przezroczyste dla klienta.

Pisząc aplikację wykorzystującą interfejsy zdalne należy pamiętać o kilku ważnych aspektach:

- argumenty i rezultaty metod EJB muszą być serializowalne,
- nie jest możliwe przekazywanie z warstwy EJB obiektów encyjnych, których pola są pobierane z opóźnieniem (tzw. lazy fetch):
- można wymusić stosowanie natychmiastowego pobierania wszystkich pól,
- można udostępnić zestaw funkcji pozwalających na pobieranie dodatkowych danych z opóźnieniem;
- może być wymagane spakowanie definicji interfejsów EJB i klas encyjnych zarówno z modułem EJB jak i z wykorzystującymi go modułami webowymi lub klienckimi:
- można wyłączyć te elementy do oddzielnego archiwum JAR.

Na EJB składają się:

- klasy implementujące funkcjonalność EJB,
- opcjonalne interfejsy biznesowe:
  - ✓ remote – dla dostępu zdalnego do EJB,
  - ✓ local – dla dostępu lokalnego do EJB;
- inne klasy z których korzystają EJB,
- opcjonalne deskryptory opisujące EJB:
  - ✓ ejb-jar.xml ,
  - ✓ jboss-ejb3.xml .

Komponenty Sesyjne:

Wymagania co do klasy beana:

- ✓ musi posiadać odpowiednią adnotację (@Stateless , @Statefull , @Singleton),
- ✓ musi być publiczna i nie może być abstrakcyjna ani finalna,
- ✓ musi implementować metody biznesowe,
- ✓ musi mieć publiczny, bezparametrowy konstruktor,
- ✓ nie może definiować metody finalize() ,
- ✓ może posiadać metody asynchroniczne (@Asynchronous),
- ✓ nazwy metod nie mogą zaczynać się od prefiksu ejb .

Informacje co do metody:

- Możliwość definiowania metod wywoływanych na określonych etapach cyklu życia EJB poprzez odpowiednie adnotacje na metodach o sygnaturze void ...() :
  @PostConstruct – wywołana po wstrzyknięciu zależności,
  @PreDestroy – wywołana przed usunięcie beana przez kontener,
  @PrePassivate – wywołana przed „zatrzymaniem" beana stanowego,
  @PostActivate – wywołana po aktywowaniu „zatrzymanego" beana stanowego.

Przykłady kodu:

Bezstanowy sesyjny bean EJB nie dostępny przez interfejs (no-interface view):

@Stateless
public class KompService {
...
}
	Komponet

@ViewScoped
@Named
public class **AppView** {
@EJB
private KompService service;
}
	Użycie komponentu

Bezstanowy sesyjny bean EJB dostępny przez Interfejs lokalny:

tak		albo tak

@Local
public interface KompLocal {
...
}
	Interfejs lokalny:	**public interface KompLocal** {
		...
		}

@Stateless
public class KompService implements KompLocal {
...
}
	Komponet	@Local(KompLocal.class)
		@Stateless
		public class KompService implements KompLocal {
		...
		}

@ViewScoped
@Named
public class AppView {
@EJB
private KompLocal service;
}
	Użycie komponentu	@ViewScoped
		@Named
		public class AppView {
		@EJB
		private KompLocal service;
		}

Bezstanowy sesyjny bean EJB nie dostępny przez interfejs:

————	Interfejs lokalny:

@LocalBean
@Stateless
public class KompService {
...
}
	Komponet

@ViewScoped
@Named
public class AppView {
@EJB
private KompService service;
}
	Użycie komponentu

Bezstanowy sesyjny bean EJB dostępny przez zdalny interfejs oraz lokalnie bez interfejsu:

@Remote **public interface KompRemote** {	Interfejs zdalny:

... }	

@Stateless
public class KompService implements KompRemote {
...
}
	Komponet

@ViewScoped
@Named
public class AppView {
@EJB(lookup = "java:global/komp-ear/komp-ejb/KompService!pl.agh.kis.KompRemote")
private KompRemote service;
}
	Użycie komponentu

Gdy wstrzykiwanie nie jest możliwe można wyszukać komponent tak:

KompRemote service = InitialContext.doLookup(" java:global/komp-ear/komp-ejb/KompService!pl.agh.kis.KompRemote ");

Adresy JNDI - Java Naming and Directory Interface - usługa katalogowa pozwalający klientom na wyszukiwanie obiektów za pomocą nazw:

- java:global - wyszukiwanie zdalnych beanów:
  java:global[/application name]/module name /enterprise bean name[/interface name] ;
- java:module - wyszukiwanie beanów w tym samym module:
  java:module/enterprise bean name/[interface name] ;
- java:app - wyszukiwanie beanów w tej samej aplikacji:
  java:app[/module name]/enterprise bean name [/interface name] .

Sesyjne EJB - singleton:

@Singleton
public class KompSingleton {
...
}
	Komponet

@ViewScoped
@Named
public class KompView {
@EJB
private KompSingleton singleton;
}
	Użycie komponentu

@Singleton
@Startup
public class InitKomp {
}
	Singleton uruchamiany automatycznie z aplikacją:

- singleton w EJB został zaprojektowany z myślą o dostępie wielowątkowym,
- dwa tryby zarządzania synchronizacja ustawiane za pomocą **@ConcurrencyManagement**:
  - ✓ kontener (adnotacja **@Lock**),
  - ✓ bean (**synchronized**);
- kontener pozwala na dwa rodzaje synchronizacji:
  LockType.WRITE - wykluczający dostęp do metod singletonu,
  LockType.READ - współdzielony dostęp do metod singletonu;
- domyślnie synchronizacja jest realizowana przez kontener
  (**@Lock(LockType.WRITE)** na każdej metodzie)).

Timer Service:

- usługa serwera aplikacji, która umożliwia wywoływanie określonych działań o określonym czasie i co określony czas,
- zdarzenia, które wysyła usługa timera mogą odbierać EJB:
  - sesyjne bezstanowe,
  - message-driven;
- EJB z przypisanym timerem musi:
  - implementować bezargumentową lub jednoargumentową metodę adnotowaną **@Timeout** lub **@Schedule** , lub implementować interfejs **javax.ejb.TimedObject** ;
- domyślnie uruchomione timery są zapamiętywane pomiędzy uruchomieniami serwera.

Timer programistyczny:

```
@Singleton
@Startup
public class DelayedInit {
        @Resource
        private TimerService timerSerivce;

        @PostConstruct
        private void init() {
                timerService.createTimer(1000, "ping");
        }
        @Timeout
        public void execute(Timer timer) {
                System.out.println(timer.getInfo());
        }
}
```

Automatyczna akcja wykonywana co minutę:

```
@Singleton
@Startup
```

```
public class Scheduled {
	@Schedule(minute = "*/1", hour = "*", persistent = false)
	public void execute() {
		...
	}
}
```

Autoryzacja dostępu w EJB

Bezpieczeństwo deklaratywne w EJB

- moduł EJB – wymagane role definiowane za pomocą adnotacji na klasach/metodach biznesowych:
  - @RolesAllowed(...) - dostęp dla użytkowników o podanych rolach,
  - @PermitAll - dostęp dla wszystkich użytkowników,
  - @DenyAll - niedostępne dla nikogo;
- zasięg:
  - adnotacja na poziomie klasy dotyczy wszystkich metod tej klasy,
  - adnotacje na poziomie metod nadpisują definicje na poziomie klasy.
- wymagane jest użycie adnotacji @DeclareRoles w połączeniu z @RolesAllowed.

```
@Stateless
@DeclareRoles({"ADMIN", "USER"})
public class BookService {
	@RolesAllowed("USER")
	public List<Book> findAll() {
		...
	}
	@RolesAllowed("ADMIN")
	public void deleteAll() {
		...
	}
}
```

Bezpieczeństwo programistyczne w EJB

- obiekt klasy SessionContext udostępnia informacje o użytkowniku:
  - isCallerInRole() - sprawdzenie roli,
  - getCallerPrincipal() – uzyskanie tożsamości;
- wstrzyknięcie za pomocą adnotacji @Resource .

```
@Stateless
@DeclareRoles({"ADMIN", "USER"})
public class BookService {
	@Resource
	private SessionContext ctx;

	@RolesAllowed({"ADMIN", "USER"})
	public void saveBook(Book book) {
		boolean isAdmin = ctx.isCallerInRole("ADMIN");
```

```
                String login = ctx.getCallerPrincipal().getName();
                boolean isOwner = book.getOwner().getLogin().equals(login);
                if (isAdmin || isOwner) {
                        ...
                } else {
                        throw new SecurityException("brak dostepu");
                }
}
}
```

Tworzenie aplikacji klienckiej Java EE:

- klientem EJB może być dowolna aplikacja napisana w Javie
- w aplikacji klienckiej można korzystać wyłącznie z interfejsów zdalnych EJB a dostęp do EJB uzyskuje się przy pomocy adnotacji lub JNDI:
- adnotacje działają tylko w klasie głównej aplikacji;
- aplikacja kliencka może mieć deskryptor application-client.xml,
- klasa główna aplikacji i zależności od innych bibliotek są podawane w manifeście JAR-a,
- do uruchomienia aplikacji klienckiej potrzebny jest dostęp do:
    - bibliotek klienckich serwera aplikacji,
    - bibliotek z interfejsami zdalnymi EJB oraz klasami (interfejsami)
    - współdzielonymi w aplikacji klienckiej i module EJB.

Uruchamianie aplikacji klienckiej Java EE

- aplikacja kliencka uruchamiana jest w specjalnym kontenerze klienckim,
- sposób uruchamiania aplikacji jest zależny od serwera aplikacji:
- często dołączany jest skrypt lub program uruchamiający;
- jeśli aplikacja kliencka jest uruchamiana na innym komputerze niż serwer aplikacji, to musi mieć otwarty dostęp do odpowiednich portów serwera aplikacji, m.in.:
    - usługi JNDI,
    - portów pozwalających na wołanie EJBContext,
    - ...
- jest możliwe uruchomienie aplikacji klienckiej bez kontenera klienckiego, ale wymaga to dodatkowych działań konfiguracyjnych oraz dołączenia bibliotek klienckich serwera aplikacji.

---------- **czesc praktyczna** ----------

Pierwsza komponent sesyjny w JavaEE.

Tworzenie komponentów EJB w JAVA EE nie jest zbyt skomplikowane. Sprowadza się w zasadzie do stworzenia zwykłych klas, które posiadając odpowiednie adnotacje.

Stwórzmy więc dwa komponenty: Bezstanowy – zwracający zawsze jakąś wartość np. niech to będzie wynik dodawania dwóch liczb oraz Singleton – będący licznikiem wykonanych obliczeń.

Nasza aplikacja składać się będzie z 3 niezależnych części:

- implementacji komponentów

- specyfikacji interfejsów

oraz implementacji klienta wykorzysującego nasze komponenty w postaci apliakacji webowej w postaci servleta lub pliku JSF. .

Tworzymy wiec 3 projekty: (Do ich tworzenia można użyć mavena lub stworzyć je samodzielnie)

ejb3-server-api	ejb3-server-impl	ejb3-server-war
Rodzaj deploy : jar	**Rodzaj deploy:** jar	**Rodzaj deploy:** war

Projekt **ejb3-server-api** zawiera tylko specyfikacje interfejsów: w zależności od planowanego użycia należy stworzyć interfejsy lokalne lub zdalne (lub takie i takie)

ITestAddBean :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface ITestAddBean {

        int add(int a,int b);
}
```

- ILocalTestAddBean :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface ILocalTestAddBean extends ITestAddBean {}
```

IRemoteTestAddBean :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface IRemoteTestAddBean extends ITestAddBean {}
```

ITestBeanCounter :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface ITestBeanCounter {
```

```
        void increment();
        long getNumber();
}
```

ILocalTestBeanCounter :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface ILocalTestBeanCounter extends ITestBeanCounter {}
```

IRemoteTestBeanCounter :

```
package pl.agh.kis.soa.ejb3.server.api;

public interface IRemoteTestBeanCounter extends ITestBeanCounter{}
```

Projekt **ejb3-server-imp** zawiera implementacje poszczególnych interfejsów

```
package pl.agh.kis.soa.ejb3.server.impl;

import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class TestAddBean implements ILocalTestAddBean {

   /**
    * Default constructor.
    */
   public TestAddBean () {
      // TODO Auto-generated constructor stub
   }

   public int add(int a,int b){
    int r=a+b;
    return r;
   }

}
```

```
package pl.agh.kis.soa.ejb3.server.impl;

import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Singleton;

import pl.agh.kis.soa.ejb3.server.api.ILocalTestBeanCounter;
import pl.agh.kis.soa.ejb3.server.api.IlTestBeanCounter;
import pl.agh.kis.soa.ejb3.server.api.IRemoteTestBeanCounter;

@Singleton
@Remote(IRemoteTestBeanCounter.class)
@Local(ILocalTestBeanCounter.class)
public class TestBeanCounter implements ITestBeanCounter{
```

```
    long counterNumber = 0;

@Override
public void increment() {
counterNumber ++;
}

@Override
public long getNumber() {
return counterNumber;
}

}
```

Zamiast deklaracji lokalnego interfejsu możliwe jest zastosowanie @LocalBean :

```
@Singleton
@Remote(IRemoteTestBeanCounter.class)
@LocalBean
public class TestBeanCounter implements ITestBeanCounter{
        //...
}
```

Dzieki temu w aplikacji klienckiej możliwe jest bezpośrednie wstrzyknięcie komponentu poprzez uzycie:

```
@EJB
TestBeanCounter
```

Innym sposobem dostępu do komponenty z poziomu aplikacji klienckiej jest użycie JNDI i jej metody lookup.

Poniższa tabela pozwoli zorientować się w znaczeniu poszczególnych elementów.

Element	**Opis**
app-name	To nazwa aplikacji typu enterprise (bez elementu .ear), jeśli komponent EJB znajduje się w pakiecie EAR.
module-name	To nazwa modułu (bez elementu .jar lub .war), w którym znajduje się komponent EJB
distinct-name	Można opcjonalnie ustawić nazwę wyróżniającą dla każdej jednostki wdrożenia. bean-name To nazwa klasy ziarna.
fully-qualified-classname-of-the-remote-interface	To w pełni kwalifikowana nazwa klasy interfejsu zdalnego

Przykładowy kod pozwalający na wyszukiwanie komponentów stanowych wyglądałby mniej więcej tak:

```
import java.util.Properties;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class LookerUp {

private Properties prop = new Properties();
private String jndiPrefix;

public LookerUp(){
prop.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
}
public Object findLocalSessionBean(String moduleName, String beanName, String interfaceFullQualified
Name) throws NamingException{

final Context context = new InitialContext(prop);
Object object = context.lookup("java:global/"+moduleName+"/"+beanName+"!"+interfaceFullQualifiedNa
me);
context.close();

return object;
}

public Object findSessionBean(String jndiName) throws NamingException{

final Context context = new InitialContext(prop);
Object object = context.lookup(jndiName);
context.close();

return object;
}

}
```

Wywołanie w kodzie klienta:

```
//--- EJB Lookup w tym samym  WAR
String moduleName = "ejb3-server-client-war"; // WAR name
String beanName = "TestBean";
String interfaceQualifiedName = ILocalTestBean.class.getName();
LookerUp wildf9Lookerup = new LookerUp();
proxy = (ILocalTestBean) wildf9Lookerup.findLocalSessionBean(moduleName,beanName,interfaceQualif
iedName);
```

Samodzielnie dokończ projekt implementując aplikacje webową, która korzystać będzie z obu komponentów.

Przykładowy projekt dołączony do materiałów lab w katalogu Bean – zawiera prosty przelicznik temperatury.

Na wszelki wypadek podaje namiar na dokumentacje wspierającą tworzenie projektów EJB w środowisku Intelij.

https://www.jetbrains.com/help/idea/ejb.html

Zadanie zaliczeniowe

1. Napisać aplikacje do zakupu biletów do teatru w oparciu o różnego rodzaju komponenty EJB.

Singletonowy komponent EJB ma zawierać metody obsługujące zarządzanie miejscami w teatrze. Dodajmy do projektu kilka ziaren sesyjnych związanych z logiką biznesową, takich jak bezstanowe ziarno sesyjne odpowiedzialne za informacje o dostępności poszczególnych miejsc w teatrze i stanowe ziarno sesyjne działające jako pośrednik systemu płatności – pozwalające na zakup biletu na określone miejsce. Zakup wiąze się z zmniejszeniem stanu konta poszczególnego użytkownika. .

Ziarno singletonowe udostępnia trzy metody publiczne. Metoda getSeatList zwraca listę obiektów Seat, które zostaną wykorzystane do wskazania użytkownikowi, czy podane miejsce zostało zarezerwowane.

Metoda getSeatPrice to metoda pomocnicza, która zwraca cenę za miejsce jako typ int, co umożliwia szybkie sprawdzenie, czy użytkownika stać na zakup wskazanego miejsca.

Ostatnia z metod, buyTicket, odpowiada za zakup biletu i oznaczenie miejsca jako zarezerwowanego.

Oprócz tego Singleton ma stworzyć liste miejsc z przypisanymi im cenami w momencie stworzenia komponentu.

Ziarno nad metodami dotyczącymi obsługi obiektów Seat powinno zawiera adnotację **@Lock**. Służy ona do sterowania współbieżnością singletonu**.** Współbieżny dostęp do singletonowego EJB jest domyślnie kontrolowany przez kontener.

Aby kontrolować zawartość portfela klienta, potrzebny będzie komponent przechowujący dane sesji z klientem. Głównym celem klasy sesyjnej jest wywołanie metody buyTicket singletonu po przeprowadzeniu kilku prostych testów związanych z logiką biznesową. Jeśli w trakcie sprawdzeń pojawi się sytuacja niedozwolona, aplikacja zgłosi wyjątek. Dotyczy to między innymi sytuacji, w których miejsce zostało już zarezerwowane lub gdy klient nie posiada wystarczających środków na zakup biletu

Klientem aplikacji niech będzie aplikacja webowa stworzona w JSF. Zakres i projekt pozostawiam do Państwa uznania.