SOA – laboratorium nr 5

Temat: Tworzenie EJB oraz aplikacji klienckich.

EJB 3.1 — nowe funkcjonalności

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

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 3.1 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:

Przykładowe zadanie

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

Singletonowy komponent EJB ma zawierać metody obsługujące pamięć podręczną miejsc w teatrze. Dodajmy do projektu kilka ziaren sesyjnych związanych z logiką biznesową, takich jak bezstanowe ziarno sesyjne odpowiedzialne za widok miejsc w teatrze i stanowe ziarno sesyjne działające jako pośrednik systemu płatności.

Krok 1. Przykład - Tworzenie singletonowych komponentów EJB

Jak sama nazwa wskazuje, javax.ejb.Singleton to ziarno sesyjne dopuszczające istnienie co najwyżej jednej instancji na poziomie aplikacji.

Aby wymusić traktowanie EJB jako singleton, wystarczy zastosować dla niego adnotację @javax.ejb.Singleton. Inną adnotacją wartą poznania jest @javax.ejb.Startup. Powoduje ona utworzenie ziarna przez kontener w momencie startu aplikacji. Jeśli w EJB został zdefiniowany kod powiązany z adnotacją @javax.annotation.PostConstruct, zostanie on automatycznie wykonany.

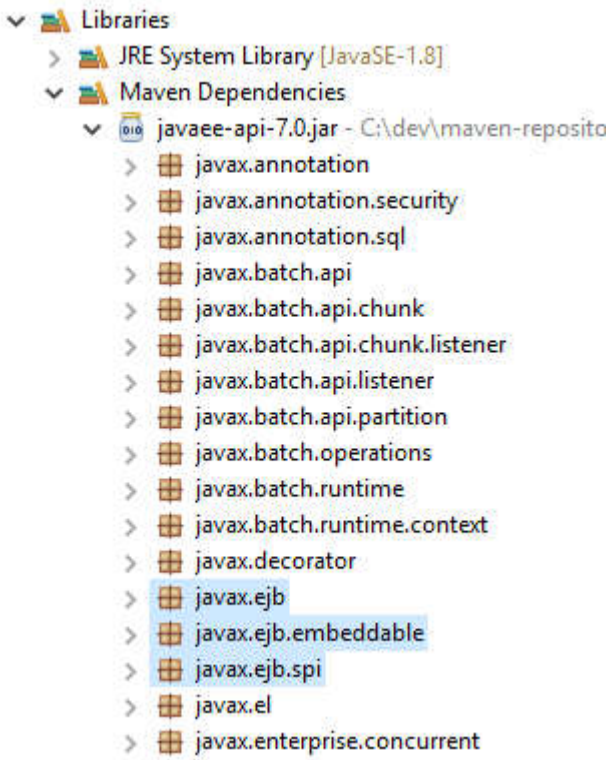
Jeśli zamierzamy użyć API EJB 3.1 wraz z adnotacjami, potrzebne będzie **Common Annotations API** (JSR-250) i zależności **API EJB 3.1**. Jeśli do tworzenia projektu użyjemy Mavena to dodajemy poniższy blok poniżej sekcji pom.

```
<dependencies>
<dependency>
<groupId>org.jboss.spec.javax.annotation</groupId>
<artifactId>jboss-annotations-api_1.1_spec</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.jboss.spec.javax.ejb</groupId>
<artifactId>jboss-ejb-api_3.1_spec</artifactId>
<scope>provided</scope>
</dependency>
```

Jeśli chcielibyśmy korzystać z wersji EJB 3.2 to wystarczy w pliki konfiguracyjnym Mavena pom.xml zdefiniować następujący wpis:

```
<dependencies>
        <dependency>
                <groupId>javax</groupId>
                <artifactId>javaee-api</artifactId>
                <version>7.0</version>
                <scope>provided</scope>
        </dependency>
</dependencies>
```

Warto takze upewnić sie czy w pakiecie javaee-api-7.0.jar w Eclipse znajduja sie pakiety obsługujace EJB:



Można również upewnić się czy nasza wersja WildFly obsługuje np standard 3.2. W tym celu należy sprawdzić czy zawiera następujący plik:

${WildFly10_HOME}\modules\system\layers\base\javax\ejb\api\main\jboss-ejb-api_3.2_spec-1.0.0.Final.jar

Tworzenie kodu aplikacji EJB

Tworzenie klas EJB nie wymaga wyrafinowanych kreatorów — wystarczy dodać zwykłe klasy Javy. Można użyć polecenia *New/Java Class* z menu *File* i wpisać np. TheatreBox jako nazwę klasy i pl.agh.test.ejb jako nazwę pakietu

Klasa zawiera następującą implementację.

```
@Singleton
@Startup
public class TheatreBox {
    private ArrayList<Seat> seatList;
    private static final Logger logger =
    Logger.getLogger(TheatreBox.class);
    @PostConstruct
    public void setupTheatre(){
    seatList = new ArrayList<Seat>();
    int id = 0;
    for (int i=0;i<5;i++) {
    Seat seat = new Seat(++id, "Parter",40);
    seatList.add(seat);
    }
    for (int i=0;i<5;i++) {
```

```
    Seat seat = new Seat(++id, "Balkon I",20);
    seatList.add(seat);
    }
    for (int i=0;i<5;i++) {
    Seat seat = new Seat(++id, "Balkon II",10);
    seatList.add(seat);
    }
    logger.info("Utworzono listę miejsc.");
    }
    @Lock(READ)
    public ArrayList<Seat> getSeatList() {
    return seatList;
    }
    @Lock(READ)
    public int getSeatPrice(int id) {
    return getSeatList().get(id).getPrice();
    }
    @Lock(WRITE)
    public void buyTicket(int seatId ) {
    Seat seat = getSeatList().get(seatId);
    seat.setBooked(true);
    }
}
```

Metoda setupTheatre zostaje wywołana tuż po wdrożeniu aplikacji i ma za zadanie przygotowanie listy miejsc przechowywanej w obiektach Seat. Każdy obiekt Seat wykorzystuje konstruktor z trzema polami: identyfikatorem miejsca, jego opisem i ceną.

```
public class Sea {
  public Seat(int id, String seat, int price) {
       this.id = id;
      this.seatName = seat;
     this.price = price;
  }
     // Metody pobierania i ustawiania – na razie pominołem.
}
```

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.

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

Dostęp w celu odczytu i zapisu ma w danym momencie tylko jeden klient. Istnieje jednak możliwość zdefiniowania bardziej szczegółowego poziomu obsługi współbieżności przez użycie adnotacji.

Adnotacja @Lock pozwala określić, jaki rodzaj współbieżnego dostępu jest dla danej metody dozwolony. Adnotacja @Lock z przekazanym typem javax.ejb.LockType.READ umożliwia wielowątkowy dostęp do ziarna.

```
@Lock(READ)
public ArrayList<Seat> getSeatList() {
   return seatList;
}
```

Z drugiej strony, użycie typu javax.ejb.LockType.WRITE powoduje, że w danym momencie dostęp do ziarna ma tylko jeden z wątków.

```
@Lock(WRITE)
public void buyTicket(Seat seat) {
 seat.setBooked(true);
}
```

Ogólna zasada jest prosta: typ READ służy do oznaczania metod, które jedynie odczytują wartości z pamięci podręcznej. Typu WRITE należy używać dla metod, które zmieniają wartości elementów zawartych w pamięci podręcznej.

Istnieje jeszcze inne podejście do zarządzania współbieżnością. Użycie adnotacji **@javax.ejb.ConcurrencyManagement** z argumentem ConcurrencyManagementType.BEAN umożliwia wyłączenie stosowanych do tej pory adnotacji @Lock i przeniesienie odpowiedzialności za prawidłową obsługę współbieżności na kod ziarna singletonowego. Od tego momentu w celu zapewnienia odpowiedniej ochrony przed uszkodzeniem danych przez współbieżny zapis należy w przypadku metody buyTicket stosować słowo kluczowe synchronized.

```
@Singleton
@Startup
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class TheatreBox {
. . . .
 public void buyTicket(Seat seat) {
 synchronized (this){
 seat.setBooked(true);
 }
 }
}
```

Ponieważ współbieżny dostęp jest blokowany w momencie wejścia wątku do bloku synchronized, żadna inna metoda nie będzie mogła zmienić obiektu, dopóki nie opuści go aktualnie wykonywany wątek

Krok 2. Przygotowanie ziaren sesyjnych

1. Dodanie ziarna bezstanowego

Pierwszym tworzonym ziarnem będzie pl.agh.test.ejb.TheatreInfoBean, które w zasadzie zawiera jedynie minimalny kod odpowiedzialny za przygotowanie listy miejsc na podstawie zawartości tablicy. W praktyce stanowi fasadę dla ziarna singletonowego.

```
@Stateless
@Remote(TheatreInfo.class)
public class TheatreInfoBean implements TheatreInfo {
 @EJB TheatreBox box;
 @Override
 public String printSeatList() {
 ArrayList<Seat> seats= box.getSeatList();
 StringBuffer sb = new StringBuffer();
 for (Seat seat: seats) {
 sb.append(seat.toString());
 sb.append("\n");
 }
 return sb.toString();
 }
}
```

Ponieważ chcę wywoływać komponent EJB z poziomu klienta zdalnego, przy użyciu adnotacji @Remote(TheatreInfo.class) dodaliśmy dla niego zdalny interfejs.

Przyjrzyjmy się dokładniej wierszowi @EJB TheatreBox, dzięki któremu możliwe jest bezpieczne wstrzyknięcie EJB do klasy bez ręcznego przeprowadzania wywołania JNDI. W praktyce oznacza to zwiększoną przenośność aplikacji, ponieważ różne serwery aplikacji mogą stosować odmienne reguły JNDI.

Interfejs zdalny ziarna jest wyjątkowo prosty.

```
public interface TheatreInfo {
        public String printSeatList();
}
```

Jeśli planowane jest udostępnienie komponentu EJB jedynie lokalnym klientom (na przykład serwletowi), można pominąć definicję interfejsu zdalnego i po prostu użyć adnotacji @Stateless. Serwer aplikacji utworzy dla ziarna sesyjnego widok no-interface, który można bezpieczne wstrzykiwać do lokalnych klientów, takich jak serwlety lub inne komponenty EJB.

2. Dodanie ziarna sesyjnego

Aby kontrolować zawartość portfela klienta, potrzebny będzie komponent przechowujący dane sesji z klientem. Zamiana klasy Javy na stanowe ziarno sesyjne wymaga jedynie dodania adnotacji @Stateful do zawartości klasy pl.agh.test.ejb.TheatreBookerBean.

```
 @Stateful
@Remote(TheatreBooker.class)
 public class TheatreBookerBean implements TheatreBooker {
        private static final Logger logger =
```

```
        Logger.getLogger(TheatreBookerBean.class);
        int money;
        @EJB TheatreBox theatreBox;
        @PostConstruct
        public void createCustomer() {
        this.money=100;
        }
        @override
        public String bookSeat(int seatId) throws SeatBookedException,
        NotEnoughMoneyException {
        Seat seat = theatreBox.getSeatList().get(seatId);
        // Logika biznesowa.
        if (seat.isBooked()) {
        throw new SeatBookedException("To miejsce jest już zarezerwowane!");
        }
        if (seat.getPrice() > money) {
        throw new NotEnoughMoneyException("Nie masz wystarczających środków,
        aby kupić ten bilet!");
        }
        theatreBox.buyTicket(seatId);
        money = money -seat.getPrice();
        logger.info("Rezerwacja przyjęta.");
        return "Rezerwacja przyjęta.";
        }
    }
```

Ziarno zawiera adnotację @PostConstruct zapewniającą inicjalizację zmiennej sesyjnej (money), która odpowiada za sprawdzanie, czy klient ma wystarczającą ilość środków, by zakupić bilet.

Głównym celem klasy 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. Aby przykład zadziałał prawidłowo, stosowane klasy wyjątków powinny dziedziczyć po ogólnej klasie Exception.

```
        public class SeatBookedException extends Exception {
        . . . .
        }
```

Użycie wyjątku wykonania (na przykład EJBException) spowodowałoby usunięcie instancji, a co za tym idzie, przerwanie komunikacji między klientem zdalnym i serwerem. Kiedy używamy komponentów EJB, musimy odpowiednio dobierać rodzaj stosowanych wyjątków — wyjątek wykonania powinien być stosowany w sytuacjach, w których nie można już nic zrobić (na przykład zostało przerwane połączenie z bazą danych). W przeciwnym razie warto zgłosić wyjątek sprawdzany (lub nie zgłaszać żadnych wyjątków), jeśli wyjątek dotyczyć ma przewidzianej sytuacji biznesowej, na przykład wcześniejszej rezerwacji miejsca przez inną osobę.

Krok 3. Wdrożenie aplikacji EJB

(proponuje użycie Mavena)

W obecnej postaci projekt mógłby nadawać się do spakowania jako projekt EJB po wydaniu odpowiedniego polecenia Maven w wierszu poleceń z poziomu głównego folderu projektu.

mvn install

Powyższe polecenie spowodowałoby skompilowanie i spakowanie aplikacji, którą wystarczy później skopiować do folderu deployments serwera aplikacji. Skonfigurujmy Maven w taki sposób, by korzystało z dodatku JBoss, dodając poniższe wpisy w pliku pom.xml.

```
<build>
 <finalName>${project.artifactId}</finalName>
 <plugins>
 <plugin>
 <groupId>org.jboss.as.plugins</groupId>
 <artifactId>jboss-as-maven-plugin</artifactId>
 <version>${version.jboss.maven.plugin}</version>
 <configuration>
 <filename>${project.build.finalName}.jar</filename>
 </configuration>
 </plugin>
 <plugin>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>2.3.1</version>
 <configuration>
 <source>1.7</source>
 <target>1.7</target>
</configuration>
 </plugin>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-ejb-plugin</artifactId>
 <version>2.3</version>
 <configuration>
 <ejbVersion>3.1</ejbVersion>
 <generateClient>true</generateClient>
 </configuration>
 </plugin>
 </plugins>
</build>
```

Pierwsza część kodu XML określa atrybut finalName dla projektu. Zawiera on nazwę spakowanego artefaktu (w naszym przypadku jest to nazwa odpowiadająca identyfikatorowi projektu, co pozwoli odnieść się do pliku ticket-agency-ejb.jar).

Artefakt jboss-as-maven-plugin spowoduje włączenie dodatku JBoss dla Maven i wdrożenie projektu.

W pliku XML powinien znaleźć się również artefakt maven-compiler-plugin wymuszający zgodność z Javą 1.8 i aktywujący procesory adnotacji. Artefakt maven-ejb-plugin znajdował się w pliku pom.xml już wcześniej, ponieważ wybraliśmy archetyp EJB. Ustawiamy opcję generateClient na wartość true (domyślną wartością jest false), by utworzyć klasy klientów EJB. Po skonfigurowaniu dodatku JBoss wdrożenie aplikacji można przeprowadzić automatycznie z poziomu projektu po wpisaniu polecenia:

mvn jboss-as:deploy

Ponieważ w trakcie prac nad aplikacją wdrożenia występują wyjątkowo często, warto wykonywać wspomnianą operację z poziomu środowiska Eclipse. W tym celu należy utworzyć nową konfigurację Run Configuration, wybierając z menu Run polecenie Run Configurations. Wpisujemy folder bazowy projektu (narzędzie Browse workspace pomoże wybrać projekt z listy) i w polu Goals wpisujemy cel Maven.

Krok 4. Tworzenie zdalnego klienta EJB

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

Oznacza to tyle, że dowiązanie JNDI dla komponentu EJB TheatreInfo umieszczonego w pliku ticket-agency-ejb.jar będzie miało postać:

ejb:/ticket-agency-ejb//TheatreInfoBean!com.test.ejb.TheatreInfo

Stanowe komponenty EJB będą zawierały dodatkowy atrybut — ?stateful — na końcu tekstu JNDI, więc pełny adres będzie miał postać:

ejb:<app-n me>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-interface>?stateful

Oto przykład dowiązania dotyczącego klasy TheatreBookerBean.

ejb:/ticket-agency-ejb//TheatreBookerBean!com.test.ejb.TheatreBooker?stateful

Po utworzeniu tekstu dowiązania JNDI można przystąpić do tworzenia klienta EJB. Najlepsza strategia polega na wykonaniu dla klienta osobnego projektu Maven, by nie zaśmiecać projektu serwerowego zależnościami specyficznymi dla klienta. Podobnie jak w przypadku projektu ticket-agency-ejb, tworzymy nowy projekt Maven.

Sugeruje użycie prostego archetypu *maven-archetype-quickstart*, który spowoduje utworzenie prostej struktury projektu zgodnej z Maven.

Konfiguracja pliku pom.xml projektu klienta

Konfiguracja zależności klienta w zasadzie wymaga wskazania wszystkich bibliotek odpowiedzialnych za połączenie i transport danych do serwera, a także podania wszystkich zależności klienta EJB. Najpierw jednak, podobnie jak to było w przypadku projektu serwera, dodajmy zależności klienta EJB w postaci BOM.

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.jboss.as</groupId>
<artifactId>jboss-as-ejb-client-bom</artifactId>
<version>7.2.0.Final</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Następnie musimy dodać zestaw zależności niezbędnych do prawidłowego rozwiązania interfejsów EJB (artefakt ticket-agency-ejb) i API transakcyjnego JBoss (to niezbędny element, bo EJB to komponenty świadome systemów transakcyjnych). Pozostałe zależności to: API jboss-ejb-api i ejb-client, org.jboss.xnio i API org.jboss.xnio (niskopoziomowa implementacja wejścia-wyjścia), org.jboss.remoting3 (podstawowy protokół transportowy), który to wymaga org.jboss.sasl (bezpieczeństwo warstwy transportowej) i na końcu API org.jboss.marshalling (serializacja obiektów wysyłanych z i do serwera).

```
<dependencies>
<dependency>
<groupId>com.packtpub.as7development.chapter3</groupId>
<artifactId>ticket-agency-ejb</artifactId>
<type>ejb-client</type>
<version>${project.version}</version>
</dependency>
<dependency>
<groupId>org.jboss.spec.javax.transaction</groupId>
<artifactId>jboss-transaction-api_1.1_spec</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.jboss.spec.javax.ejb</groupId>
<artifactId>jboss-ejb-api_3.1_spec</artifactId>
<scope>runtime</scope>
</dependency>

<dependency>
<groupId>org.jboss</groupId>
<artifactId>jboss-ejb-client</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.jboss.xnio</groupId>
<artifactId>xnio-api</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.jboss.xnio</groupId>
<artifactId>xnio-nio</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
```

```
<groupId>org.jboss.remoting3</groupId>
<artifactId>jboss-remoting</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.jboss.sasl</groupId>
<artifactId>jboss-sasl</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.jboss.marshalling</groupId>
<artifactId>jboss-marshalling-river</artifactId>
<scope>runtime</scope>
</dependency>
</dependencies>
```

Tworzenie kodu klienta EJB

Zakończyliśmy konfigurację. Tworzenie kodu zaczniemy od dodania klasy pl.agh.test.client.RemoteEJBClient, która będzie komunikowała się z aplikacją biletową EJB.

```
public class RemoteEJBClient {
private final static java.util.logging.Logger logger = Logger.
getLogger(RemoteEJBClient.class .getName());
private final static Hashtable jndiProperties = new Hashtable();
public static void main(String[] args) throws Exception {
    testRemoteEJB();
}
private static void testRemoteEJB() throws NamingException {
jndiProperties.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
final TheatreInfo info = lookupTheatreInfoEJB(); [1]
final TheatreBooker book = lookupTheatreBookerEJB(); [2]
String command = "";
/* Kod metody został pominięty. Wyświetla informację powitalną. */
dumpWelcomeMessage();
while (true) {
command = IOUtils.readLine("> "); [3]
if (command.equals("book")) { [4]
int seatId = 0;
try {
seatId = IOUtils.readInt("Wpisz id miejsca");
} catch (NumberFormatException e1) {
logger.info("Niewłaściwy format!");
continue;
}
try {
String retVal = book.bookSeat(seatId-1);
}
catch (SeatBookedException e) {
logger.info(e.getMessage());
continue;
}
```

```
 catch (NotEnoughMoneyException e) {
 logger.info(e.getMessage());
 continue;
 }
 }
 else if (command.equals("lista")) { [5]
 logger.info(info.printSeatList().toString());
 continue;
 }
else if (command.equals("koniec")) { [6]
logger.info("Żegnam");
break;
}
else {
logger.info("Nieznane polecenie"+command);
}
}
}
private static TheatreInfo lookupTheatreInfoEJB() throws NamingException {
final Context context = new InitialContext(jndiProperties);
return (TheatreInfo) context.lookup("ejb:/ticket-agency-
ejb//TheatreInfoBean!com.test.ejb.TheatreInfo");
}
private static TheatreBooker lookupTheatreBookerEJB() throws
NamingException {
final Context context = new InitialContext(jndiProperties);
return (TheatreBooker) context.lookup("ejb:/ticket-agency-
ejb/TheatreBookerBean!com.test.ejb.TheatreBooker?stateful");
}
}
```

Wyszukujemy SLSB TheatreInfo [1] i SFSB TheatreBrooker [2] , używając omówionych wcześniej reguł JNDI.
Główna pętla programu odczytuje dane z konsoli [3] i na podstawie przekazanej wartości przeprowadza odpowiednie operacje: rezerwuje bilet [4] , wyświetla listę biletów [5] lub kończy działanie aplikacji [6] .

Dodanie konfiguracji klienta EJB

Przedstawiony kod nie zawiera żadnej informacji na temat lokalizacji serwera z komponentami EJB. Jeśli nie chce się podawać tego rodzaju informacji w kodzie, najlepiej utworzyć plik *jboss-ejb-client.properties* i udostępnić go w ścieżce wyszukiwania klas.
Domyślnie archetyp Maven dotyczący szybkiego startu nie zawiera folderu resources dla plików konfiguracyjnych projektu. Trzeba go dodać, klikając prawym przyciskiem myszy projekt, z menu kontekstowego wybierając New/Source folder i wpisując w kreatorze src/main/resources. Po wykonaniu tego zadania wystarczy już tylko utworzyć plik jboss-ejb-client.properties.

Zawartość pliku *jboss-ejb-client.properties* powinna być następująca.

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port = 4447
```

Oto krótkie wyjaśnienie jego zawartości. Opcjonalna właściwość endpoint.name reprezentuje nazwę wykorzystywaną do utworzenia końcówki klienckiej. Jeśli nie zostanie wskazana, system użyje domyślnej nazwy config-based-ejb-client-endpoint.

Właściwość remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED włącza szyfrowanie połączenia XNIO. W przeciwnym razie w połączeniu zostałby zastosowany jawny tekst.
Właściwość remote.connections służy do zdefiniowania listy nazw logicznych wykorzystywanych później przez właściwości remote.connection. *[nazwa]*.host i remote.connection. *[nazwa]*.port.
Jeśli zdefiniuje się więcej niż jedno połączenie (tak jak w przykładzie poniżej), klient będzie się komunikował tak samo z każdą z lokalizacji:

```
remote.connections=host1,host2
remote.connection.host1.host=192.168.0.1
remote.connection.host2.host=192.168.0.2
remote.connection.host1.port = 4447
remote.connection.host2.port = 4547
```

Domyślnym portem stosowanym przez system połączeń zdalnych jest 4447. Możliwe jest wskazanie własnego portu przy użyciu interfejsu CLI. Zmiana portu na 4457 wymaga użycia następującego polecenia.

```
/socket-binding-group=standard-sockets/socket-binding=remoting/:write-attribute ∑(name=port,value=4457)
```

Uruchomienie aplikacji klienckiej

Do uruchomienia aplikacji klienckiej potrzebne będą jeszcze moduły dodatkowe Maven odpowiedzialne za uruchomienie zdalnego klienta EJB.

```
<build>
<finalName>${project.artifactId}</finalName>
<plugins>
<!-- tu umieść maven-compiler-plugin -->
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>exec-maven-plugin</artifactId>
<version>${version.exec.plugin}</version>
<executions>
<execution>
<goals>
<goal>exec</goal>
</goals>
</execution>
</executions>
<configuration>
<executable>java</executable>
<workingDirectory>${project.build.directory}/
∑exec-working-directory</workingDirectory>
<arguments>
<argument>-classpath</argument>
<classpath />
<argument>com.packtpub.as7development.chapter3.client.RemoteEJBClient
∑</argument>
</arguments>
</configuration>
</plugin>
</plugins>
</build>
```

Powyższy fragment konfiguracji dodaje (poza pominiętym w celu zwiększenia czytelności maven-compiler-plugin, bo był już omawiany w części serwerowej) moduł exec-maven-plugin, który umożliwia wykonywanie programów Javy przy użyciu celu exec.
Gdy wszystkie dodatki znajdują się w konfiguracji, do skompilowania i uruchomienia projektu wystarczy użyć takiego celu Maven:

mvn install exec:exec

Zadanie do realizacji

1. Przeanalizuje i samodzielnie wykonaj na podstawie opisu znajdującego się na stronie Eclipsie tutorial pokazujący jak stworzyć aplikacje wykorzystującą EJB.
   http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jst.ejb.doc.user%2Ftopics%2Ftecrtpro.html

   Inne przykładowe tutoriale które mogą pomóc w nauce.

http://docs.jboss.org/ejb3/docs/tutorial/1.0.7/html_single/

http://www.laliluna.de/articles/posts/ejb-3-tutorial-jboss.html

http://www.roseindia.net/javabeans/sessionbeantutorial.shtml

2. Przeanalizować tutorial znajdujące się w materiałach: 1_Aplikacja EJB _

Zadania do samodzielnego wykonania (do oddania)

3. Utwórz komponent sesyjny `Konwerter` Fahrenheita zgodnie z wzorami:
   - $T_C = 5/9 * (T_F - 32)$
   - $T_F = 9/5 * T_C + 32$

Komponent powinien mieć interfejs zdalny udostepniający metody Fahr2Cels(double temp) i Cels2Fahr(**double temp)**. Następnie, napisz statyczną stronę HTML z formularzem do wprowadzania temperatury do konwersji. Formularz woła serwlet, który czyta wartość przesłanych parametrów (temperatura, kierunek konwersji) i wywołuje metode komponentu EJB, a następnie prezentuje wynik..

4. Zadanie:

Stwórz aplikacje wspierającą pracę biblioteki wykorzystując do realizacji warstwy logiki biznesowej - komponenty EJB 3.1 Możliwe funkcje do modułu bibliotecznego to: wypożyczanie książki, rezerwacja, zwracanie.

Informacje o książkach przechowujesz w pliku XML.
Aplikacja kliencka może zostać zrealizowana w postaci aplikacji tekstowej lub prostej aplikacji Webowej -> do zrealizowania jako aplikacja JSF.