Numer 1/2010(7)

Java eXPress

CZASOPISMO DLA DEWELOPERÓW JAVA

Glassfish Enterprise 5 9's z HADB

Transakcje w systemach Java Enterprise

Scala - wprowadzenie

Aplikacje Flex z BlazeDS

>> Spring - wprowadzenie

>> Log4j a komunikatory internetowe

JEP.getTeam().sayThankYou();

Już siódmy raz przyszło mi podzielić się z Wami tym, co dostaję od innych. Myślałem, że po kilku numerach przyjdzie kryzys i nie będzie chętnych do napisania artykułów. A jest zupełnie przeciwnie. Składając numer siódmy miałem materiały na cały numer ósmy. Czy to nie cudownie? Gratulacje dla wszystkich, którzy chcą w ten sposób podzielić się swoją wiedzą.

JAVA exPress z numeru na numer robi się coraz ciekawsze. Coraz więcej ludzi widzi sens w publikowaniu swoich artykułów na łamach tego pisma. To dobrze. Wiedza, którą się nie dzielimy z innymi jest mało warta.

Teraz pora na szerszą reklamę czasopisma. Nie chce mi się wierzyć, że w Polsce jest tylko 2 tyś. developerów Java, którzy chcą rozwijać się i dokształcać. Pomóżcie w naszej misji i prześlijcie na grupy dyskusyjne w pracy, czy na uczelni informacje o JAVA exPress.

Wielkie podziękowania należą się nie tylko autorom tekstów. JAVA exPress to zespół ludzi, którzy mniej lub bardziej pomagają. Zaczynając od Marka Podsiadłego i Jakuba Sosińskiego, którzy od samego początku pomagają tworzyć stronę JAVA exPress i samo pismo. Cóż wiele tu mówić. Właściwie, to oni są odpowiedzialni za wszystko co widzicie na stronach javaexpress.pl i dworld.pl. Tylko im zawdzięczamy, że mamy miejsce w sieci.

Podziękowania należą się także całej grupie tłumaczy oraz patronom. Od tego numeru odszedł jeden patron, ale w zamian przyszedł nowy. Firma Adobe będzie wspierała JAVA exPress przez co najmniej 1 rok. Mam nadzieję, że na tym się nie skończy, bo zaczęło się bardzo obiecująco.

Na koniec jak zwykle apel. Jeśli chcesz napisać artykuł, pomóc w tłumaczeniach lub tworzeniu stron www, napisz do nas na kontakt@dworld.pl.

Do zobaczenia w czerwcu,
Grzegorz Duda

Rozkład jazdy

Redaktor: Grzegorz Duda
Korekta: Agnieszka Wadowska-Duda
Strony www i teksty html: Marek Podsiadły
Grafika: Jakub Sosiński

Co w trawie piszczy...

Grzegorz Duda

ScreenSnipe

Czy to bugzilla, czy Jira, czy inny bug tracker, to wiele od jakości buga zależy od testerów. Oni podają kroki do reprodukcji buga i załączają niezbędne informacje. Bardzo często załączają zrzut ekranu. Ale co tester, to inny format i wiele razy zdarzało się, że zrzut w bmp zajmował kilka MB.

Marzyło mi się narzędzie, które zautomatyzuje proces tworzenia buga. No i jest. Co lepsze, to powstało ono w Polsce, a konkretniej w grupie Spartez z Gdańska.

ScreenSnipe umożliwia nie tylko pobranie zrzutu ekranu, ale także dodanie adnotacji, powiększeń, zaznaczeń itp. A co lepsze integruje się z JIRĄ.

Warto przyglądnąć się temu narzędziu na http://my.spartez.com/software-for-agile-developers/screensnipe.html.

Code Bubbles

Ostatnio wiele o tym się mówi. Jedni uważają to za przyszłość IDE, inni za zabawkę, a jeszcze inni za dobry materiał na pracę doktorską. Ja jestem gdzieś pomiędzy drugą i trzecią grupą, ale z całą pewnością warto przyjrzeć się Code Bubbles. Chociażby po to, aby zobaczyć jak inaczej można podejść do tematu IDE. Zamiast tworzyć kolejny klon Eclipse'a, warto zrobić krok milowy i wyprzedzić całą konkurencję.

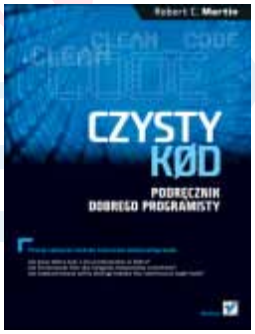
Pewnie nie tym razem, ale któraś kolejna próba na pewno się powiedzie.

Pamiętacie Wolfram Alfa? Też miało zwojować świat, a póki co nadal to jest jedynie ciekawostka jak można inaczej zrobić wyszukiwarkę.

Czy w swoich projektach podarzasz za innymi, czy szukasz nowych rozwiązań?

Polecamy książki

Jeśli JAVA exPress to dla Was za mało wiedzy, to chcielibyśmy polecić Wam 2 książki Wydawnictwa Helion. Pierwszą z nich powinniście znać pod angielskim tytułem „Effective Java", której autorem jest Joshua Bloch. Polski tytuł to Java. Efektywne Programowanie. Wydanie II.



Druga książka to nowość. Autora, wójka Boba (Uncle Bob), nie trzeba przedstawiać. Z czystym sumieniem polecamy Czysty Kod. Podręcznik dobrego programisty.

Konferencje

Zakończyła się konferencja 4Developers, a niedługo już kolejna. W Poznaniu odbędzie się w dniach 12-14 maja konferencja GeeCON, a 26 czerwca, w Warszawie Javarsovia. Mam nadzieję, że JAVA exPress pojawi się w Warszawie tym razem. Do trzech razy sztuka.

Ale to nie koniec w tym roku. W październiku jak zwykle można liczyć na JDD. Będzie fajnie jak zwykle.

Dodatkowo COOLuary nabiorą nieco rumieńców. Już w drugiej połowie maja odbędzie się czwarta edycja COOLuarów. Tym razem poprzedzona będzie całodzienną konferencją w standardowej formule. Ale jak to na Developers World przystało, spodziewajcie się niecodziennego podejścia do tematu. Więcej wkrótce na dworld.pl.

Pierwsze kroki w Scali

Łukasz Kuczera

> *If I were to choose language other than Java it would be Scala*
>
> *James Gosling*
> *Creator of Java*
>
> *I can honestly say if someone had shown me the Programming in **Scala book by Martin Odersky, Lex Spoon & Bill Venners** back in 2003 I'd probably have never created Groovy.*
>
> *James Strachan*
> *Creator of Groovy*

Historia

Scala to nowy bardzo ciekawy język programowania. Łączy ze sobą dwa „światy" programowania – świat programowania obiektowego i świat funkcyjny. Historia Scali zaczyna się trochę wcześniej niż w 2001 na Politechnice w Lozannie w Szwajcarii (EPFL), gdzie Martin Odersky wraz z grupą studentów zakłada projekt nowego języka. Wszystko zaczęło się od eksperymentalnego języka Pizza, który został utworzony przez Odersky'iego i Philip'a Wadler'a w 1998 roku. Wtedy obaj panowie pracowali w Sun'ie nad rozwojem Javy i zastanawiali się jak wprowadzic do języka Generics (JSR-014). Wpadli wtedy na pomysł, że zrobią nowy język, który posłuży im jako pole doświadczalne. Napisanie nowego języka to dużo pracy, zysk był jednak spory, oswobodzeni z Javy mogli próbować nowych rzeczy, ograniczał ich tylko JVM. Pizza ma Generics, domknięcia i pattern matching. Wiosną 1998 do Odersky'iego i Wadler'a dołącza David Stoutamire i Gilad Bracha bazując na doświadczeniach Pizzy tworzą kolejny język – GJ (Generics Java). Implementacja GJ została wcielona do Javy pod postacią generics w 1.5 (praktycznie niezmieniona). Martin Odersky napisał kompilator do GJ, który stał się podstawą javac, to było już w wersji 1.3 (2000), mimo to Generics były niedostępne w języku, aż do wersji 1.5 (2004). Orginalna wersja Odersky'iego została rozszerzona o „wildcards" czyli „? extends T". W 2001 Odersky przenosi się do Szwajcarii na uniwersytet w Lozannie, gdzie zajmuje stanowisko profesora metod programowania i rozpoczyna pracę nad nowym językiem.

Instalacja.

Do rozpoczęcia pracy wystarczy ściągnąć dystrubucję ze strony domowej – http://www.scala-lang.org/downloads i zainstalować. Do dyspozycji są dwie wersje – pod konkretną platformę bądź uniwersalny instalator. Jeżeli nie korzystamy z instalatora po rozpakowaniu należy ustawić zmienną środowiskową SCALA_HOME na katalog ze Scalą i do zmiennej PATH dodać katalog bin dystrybucji. Po tych zabiegach wpisując „scala" w linii poleceń uruchamia się interaktywny interpreter. Do małych zadań, skryptowania, zabawy interpreter jest doskonałym narzędziem, przy tworzeniu większego projektu do dyspozycji są trzy zintegrowane środowiska programistyczne: Netbeans, IntelliJ IDEA oraz Eclipse. W chwili pisania artykułu pluginy IDE są piętą achillesową Scali, da się pracować, trzeba jednak przygotować się na to że nie wszystko będzie działało tak samo dobrze jak dla Javy.

Scala jest językiem w pełni obiektowym to znaczy wszystko jest obiektem, jest też ję-

> Póki co kod wygląda jak w języku
> z dynamicznym typowaniem.
> Nic bardziej mylnego. Scala jest statyczna

zykiem funkcyjnym, pozwala na tworzenie metod wyższych rzędów, domknięć i preferuje obiekty niemutowalne. Kompiluje się do bytecode'u Javy oraz do CLI .NET, pokazuje to że twórcom zależy na przenośności języka i nie wiążą się ściśle z JVM'em.

Pierwszy kod

Wewnątrz interpretera, nie musimy deklarować zmiennych. Zatem wpisując:

```
scala> 2+2
```

Otrzymamy:

```
res0: Int = 4
```

Interpreter przypisuje to co wpisujemy do kolejnych zmiennych o nazwie res[n]. W Scali nie ma też operatorów w tradycyjnym sensie. To znaczy operatory są zaimplementowane w bibliotece, nie jest to element języka. Dlatego powyższy zapis jest skrótem od takiego:

```
scala> (2).+(2)
```

Oznacza to nic innego jak wywołanie metody „+" na obiekcie typu Int z argumentem Int w notacji tzw. operatorowej. Jeżeli metoda przyjmuje tylko jeden argument, można ją wywoływać bez kropki i nawiasów. Jest to jeden z elementów który pozwala na szybkie i łatwe tworzenie DSL'i (Domain Specific Language) w Scali. Poniższy zapis przypisuje cztery do zmiennej x (właściwie wartości, ale o tym za chwilę).

```
scala> val x = 2+2
```

Póki co kod wygląda jak w języku z dynamicznym typowaniem. Nic bardziej mylnego. Scala jest statyczna, dzięki inferencji (wnioskowania) typów nie musimy ich jawnie podawać, kompilator zrobi to za nas. Oczywiście nie zawsze otrzymany typ jest taki jakbyśmy chcieli dlatego możemy go podać jawnie:

```
scala> val x: Int = 2+2
scala> val y: Double = 2+2
```

Tam gdzie to możliwe Scala wykorzystuje obiekty Javy. Powyższe wyrażenia zostaną przez kompilator przetłumaczone najprawdopodniej na Javowe int i double. Znamiennym przykładem jest String:

```
scala> val hello = „Hello, world !"
hello: java.lang.String = Hello, world !
```

Dla rozbudzenia apetytu, możesz spróbować wpisać w interpreterze:

```
scala> „Hello, world".foreach(println(_))
```

Powyższy zapis korzysta z kilku funkcjonalności języka: pełnej obiektowości, implicit conversions (domyślnych konwersji), funkcji wyższych rzędów oraz czegoś co nazywa się placeholder syntax (składnia zastępnikowa ?). W tej części opiszę m.in. implicit conversions.

Definicja funkcji:

```
scala> def double(x: Int): Int = x*2
```

Powyższy zapis definuje funkcję o nazwie double z jednym parametrem Int, zwracającą typ Int. W funkcjach ostatnia zapis jest zwracany dzięki temu nie ma obowiązku pisania słówka kluczowego returns, jest to opcjonalne. Co instotne Inferencer jest w stanie wypełnić za nas typ zwracany przez funkcję:

```
scala> def double(x: Int) = x*2
double: (Int)Int
```

Jeżeli funkcja, metoda jest zbyt długa aby

> wszystkie metody z predef
> są automatycznie importowane
> i dostępne w aktualnym zasięgu

zmieścić się w jednej linii możemy użyć nawiasów klamrowych:

```
def double(x: Int) = {
  x*2
}
```

Istotny jest znak „=” bez niego funkcja będzie zwracała Unit, który jest podobny do „void” w Javie.

W Scali prawie wszystkie wyrażenia coś zwracają np. if:

```
def even(x: Int) = if((x%2) == 0)
true else false
```

Tutaj if-else jest wykorzystane jak operator trójargumentowy w Javie “?”. Istotny jest też sposób w jaki Scala interpretuje operator porównania „==”. Działa on odwrotnie niż w Javie, nie porównuje referencji obiektów, działa jak equals w Javie. Jest to bardziej intuicyjne i pozwala zapobiec błędom w rodzaju:

Java:

```
Public boolean isEqual(
  Integer x,Integer y) {
    return x == y;
  }
boolean b1 = isEqual(2, 2);
boolean b2 = isEqual(
  new Integer(2),
  new Integer(2));
System.out.println(b1+” “+b2);
true false
```

Wracając do funkcji even, lepiej by wyglądała tak:

```
def even(x: Int) = x%2==0
```

Myślę, że jest to odpowiednia chwila na odpalenie zintegrowanego środowiska programistycznego (jeżeli jeszcze tego nie zrobiłeś). Ja korzystam z eclipse. Tworzymy nowy projekt, nowy pakiet nazwa dowolna i nowy „obiekt” (Scala Object). Nie będę zgłębiał czym jest obiekt w Scali, po krótce można go potraktow jać jak puszkę na elementy statyczne klas. Nam jest potrzebny do napisania pierwszej aplikacji w Scali.

```
package example
object Main extends Application {
  println(2+2)
}
```

Kilka istotnych szczegółów. W Scali istnieje obiekt o nazwie Predef, wszystkie metody z predef są automatycznie importowane i dostępne w aktualnym zasięgu (scope). Stąd mamy dostępne println(), czyż nie wygląda to znacznie lepiej niż: System.out.println(); ☺ Średniki są opcjonalne tam gdzie nie są konieczne można je pominąć. `Application` jest specjalnym rodzajem Trait (cecha), cały kod wewnątrz obiektu, który rozszrza `Application` trafia do main Javy. Jeżeli nasza aplikacja miałaby czytać parametry wejściowe wtedy powinniśmy zdefiniować metodę main:

```
package example
object Main2 {
   def main(args : Array[Strin-
g]):Unit= {
println(2+2)
// Wypisuje argumenty na konsoli
args.foreach(println(_))
```

> W Scali wywołania metod i odwołania do pól są traktowane identycznie

```
// W javie wygląda to mniej więcej tak
  for(arg <- args) {
    println(arg);
  }
 }
}
```

Przeciążanie ovperatorów, implicit conversions

Zdefiniujmy klasę ułamka, która jest dobrym przykładem na pokazanie kilku cech Scali:

```
class Rational(num: Int,
  denom: Int)
```

I to wszystko ? Tak! Aczkolwiek taka definicja ułamka nie na wiele się zda. Powyższy zapis definiuje nową klasę która przyjmuje dwa parametry. Za kulisami jest tworzony konstruktor, nazywany głównym konstruktorem (primary). W Javie klasa Rational wygląda tak:

```
public class Rational {
  public Rational(
    int num, int denom) {}
}
```

Stwórzmy pola do przechowywania licznika i mianownika, dalej w Javie:

```
public class Rational {
  private int num;
  private int denom;
  public Rational(int num,
    int denom) {
    this.num = num;
    this.denom = denom;
  }
}
```

Jest to codzienność w Javie, tworzymy konstruktor bierzemy argumenty i przypisujemy do pól. Eclipse może nas w tym wyręczyć, lecz mi osobiście nie chce się klikać szukać w menu kontekstowym refactor itd. Zajmuje to podobny czas co napisanie tego z tzw. „palca”.

Spójrzmy na ten sam kod w Scali:

```
class Rational(val nom: Int,
  val denom: Int)
```

I to wszystko ? Tak ! Nawet więcej. Taka definicja tworzy główny konstruktor oraz dwa pola, dzięki słowu kluczowemu val przed argumentami. Dodatkowo podczas tworzenia instancji klasy Rational argumenty są automatycznie przypisywane do pól.

```
val half = new Rational(1,2)
println(half.num+"/"+half.denom)
```

Powyższy kod wypisze na konsoli:

```
1/2
```

No tak, ale teraz odwołuję się bezpośrednio do pól, jak będę chciał zmienić implementację pobierania licznika, mianownika to będę miał problem. W Scali dostęp jest jednorodny, to znaczy że wywołania metod i odwołania do pól są traktowane identycznie, w dowolnym momencie możemy podstawić zamiast pola metodę, która oblicza jego wartość.

Klasa Rational pozwala już na przechowywanie licznika i mianownika, a także na dostęp do nich. Niestety nie jest poprawna z punktu widzenia matematycznego, gdyż możemy zdefiniować ułamek o mianowniku 0 ! Pamiętasz obiekt Predef ? Jest w nim bardzo przydatna metoda require, jest bardzo podobna do assert w Javie, tylko na szczęście nie jest domyślnie wyłączona.

```
class Rational(val nom: Int,
  val denom: Int) {
    require(denom != 0)
}
```

> W Scali operatory to zwyczajne metody

Jeżeli teraz spróbujemy utworzyć nową instancję Rational z mianownikiem 0 zostanie wyrzucony IllegalArgumentException:

```
scala> new Rational(1,0)
java.lang.IllegalArgumentExcep-
tion: requirement failed
at   scala.Predef$.require(Predef.
scala:107)
at Rational.<init>(<console>:5)
at .<init>(<console>:6)
at .<clinit>(<console>)
at    RequestResult$.<init>(<conso
le>:3)
at  RequestResult$.<clinit>(<conso
le>)
at RequestResult$result(<console>)
at   sun.reflect.NativeMethodAcces-
sorImpl.invoke0(Native Method)
at sun.reflect.NativeMetho...
```

Kolejna rzecz o którą, aż się prosi żeby zaimplementować to metoda toString. Działanie tej metody jest identyczne jak w Javie.

```
class Rational(val nom: Int,
  val denom: Int) {
  require(denom != 0)
override def toString(): String =
  (num+"/"+denom)
}
```

Teraz po przekazaniu obiektu typu Rational do println bądź sklejeniu go ze String'iem zostanie wywołana metoda toString. Rzecz warta uwagi to słówko kluczowe override. Jest ono konieczne gdy nadpisujemy metodę z klasy nadrzędnej, jeżeli nie nadpisujemy żadnej metody, to kompilator zgłosi błąd. Jest to rozmiękczenie problemu znanego jako problem ułomnej klasy bazowej. Sprowadza się on do klas rozszerzających inne klasy, w szczególności jest on dotkliwy jeżeli nie mamy dostępu do kodu klasy bazowej. Jeżeli w Javie nadpiszemy metodę z klasy bazowej i w kolejnej wersji zostanie ona z niej usunięta (klasy bazowej) to nie mamy żadnej szansy się o tym dowiedzieć jeżeli nie śledzimy zmian w kodzie. Nasza metoda zostanie potraktowana jako definicja nowej metody, niekoniecznie będzie to efekt pożądany. W Scali taki kod nie skompiluje się. W Javie 6 zostało to lekko poprawione poprzez anotację @Overrides, niestety anotacja nie jest obowiązkowa.

Na tym etapie możemy utworzyć klasę w interpreterze i ujrzymy wywołanie metody toString:

```
scala> new Rational(1,2)
res10: Rational = 1/2
```

Jest nieźle, ale liczby o mianowniku 1 to są tak naprawdę liczby całkowite. Pisanie jawnie mianownika nie jest wygodne. Dodajmy dodatkowy konstruktor:

```
class Rational(val num: Int,
  val denom: Int) {
  require(denom != 0)
   override def toString(): String
= (num+"/"+denom)
   def this(x: Int) = this(x,1);
}

scala> new Rational(3);
res11: Rational = 3/1
```

Ostatnia funkcjonalność której nie może zabraknąć to działania na ułamkach. Żeby ograniczyć miejsce, zdefiniuję tylko dodawanie. W Javie dodałbym w tym celu metodę add w klasie Rational, która brałaby argument Rational i zwracałaby sumę obu ułamków. W Scali operatory to zwyczajne metody, zatem mogę zdefiniować nowy operator dla typu Rational:

```
class Rational(val num: Int,
  val denom: Int) {
  require(denom != 0)
   override def toString(): String
= (num+"/"+denom)
   def this(x: Int) = this(x,1);
```

> Świetnie, dodawanie ułamków działa

```
    def +(that: Rational) =
     new Rational(
       num*that.denom +
       that.num*denom,
       denom*that.denom)
}

scala> val half = new Rational(1,2)
half: Rational = 1/2

scala> half + half
res0: Rational = 4/4
```

Świetnie, dodawanie ułamków działa i mogę korzystać z notacji operatorowej pozwoli to na tworzenie bardziej intuicyjnego i zwięzłego kodu. Spróbuj przepisać to samo w Javie, kod będzie znacznie rozleglejszy a co za tym idzie trudniejszy w utrzymaniu. Klasa ma już wszystko co trzeba (pomijając brakujące operacje) jednak próba dodania liczby całkowitej do ułamka, nie powiedzie się, mimo że jest to całkowicie poprawne:

```
scala> 1 + half
<console>:7:   error:   overloaded
method value + with alternatives
(Double)Double <and> (Float)Float
<and> (Long)Long <and>
  (Int)Int  <and>  (Char)Int  <and>
(Short)Int  <and>  (Byte)Int  <and>
(java.lang.String)java.lang.
String cannot be applied to
 (Rational)   1 + half
              ^
```

W typie Int nie ma metody „+” biorącej Rational jako argument i raczej nieprędko się pojawi. Kompilator jednak próbował ją odnaleźć w kilku typach: Double, Float, Long, Char, Short, Byte i String aż w końcu się poddał. Owo wyszukiwanie to imlicit conversions, jeżeli kompilator nie może odnaleźć metody w danym obiekcie to szuka czy nie ma dostępnej konwersji do typu, który tą metodę posiada. Niemalże dynamiczne typowanie ! Tak naprawdę na codzień spotykamy się z obcym kodem i nic nie możemy z nim zrobić, możemy z tym żyć i pamiętać aby nie dodawać Int'ów do ułamków, ewentualnie pozostaje nam wzorzec Adaptera, zamiast Int'a musiałbym korzystać z klas adaptujących, jeżeli klasa adaptowana ma duży interfejs utrzymywanie go będzie problematyczne. Definicja imiplicit conversion jest bardzo prosta:

```
object Main extends Application {
   implicit def intToRational(x:
Int): Rational = new Rational(x);
  val half = new Rational(1,2);
  println(half)
  println(1+half)
}
```

Powyższy kod wypisze na konsoli:

```
1/2
3/2
```

Istotne jest to aby definicja implicit była w zasięgu. Jeżeli zdefiniuje ją w klasie Rational, nie zostanie odnaleziona. Kompilator nie importuje konwersji automatycznie, wyobraź sobie całą masę konwersji dziejących się dookoła bez wiedzy programisty. Jest to potężne narzędzie, ale nadużyte powoduje ciężkie dni z debugerem, gdy niewiadomo dlaczego kod wykonuje „magiczne” operacje. Konwersje są aplikowane także jeżeli spodziewany typ jest inny niż przekazany. W Scali istnieją konwersje pomiędzy typami prostymi w górę to znaczy rozszerzające. Dzięki konwersji ze String można pisać np. taki kod:

```
for(char <- „Hello world !”) {
    println(char)
}
```

Podsumowanie

Ostatnią rzeczą jaką chciałem opisać to integracja Javy ze Scalą. Na nasze szczęście jest ona praktycznie niezauważalna. W istocie String w Scali to nic innego jak java.lang.String. Klasy Javy z których bę-

> Scala to bardzo elegancki język,
> który pozwala programować na wysokim poziomie.

dziemy korzystać w naszym kodzie należy najpierw zaimportować (o ile są w innym pakiecie).

Mechanizm importowania jest nieco rozszerzony w stosunku do Javy. Importowane mogą być całe pakiety w ten sposób importowanie jest hierarchiczne. Jeżeli mamy pakiety o strukturze first->second, to:

```
import first._;
import second._;
```

Spowoduje zaimportowanie wszystkich elementów pakietu first (_ jest traktowane jak * w Javie) oraz wszystkich elementów pakietu first.second.

Podsumowując Scala to bardzo elegancki język, który pozwala programować na wysokim poziomie. Dzięki pełnej kompilacji do bytecodu Javy działa tak samo szybko. Jest statycznie typowany, dlatego kompilator wychwyci za nas proste błędy, a środowisko podpowie nam dokumentację dla danego obiektu. Jednocześnie cechy takie jak wnioskowanie typów, implicit conversions i duck typing pozwala na pisanie kodu, który wygląda prawie jak język dynamiczny. Elementy funkcyjne języka o których nie pisałem wcale idealnie nadają się do pisania bardzo zwięzłego kodu. Pełna integracja z Javą umożliwia pisanie projektów hybrydowych Scala-Java i uruchamianie kodu Javy w Scali jak i na odwrót. Zachęcam do spróbowania Scali nawet jeżeli nie zamierzasz pisać w niej to przynajmniej zdobędziesz wizję tego co w Javie można poprawić i przygotujesz się lepiej na domknięcia w JDK 7.

Źródła:

Java Language Specification (3rd edition)

Programming In Scala – Odersky, Spoon, Venners

http://scala-lang.org/

VRILTEK.COM

Logotypy

Portale

Plakaty

Wizytówki

Strony WWW

DTP

Sklepy internetowe

Ulotki

Banery

Spring – kontener wstrzykiwania zależności

Marcin Świerczyński

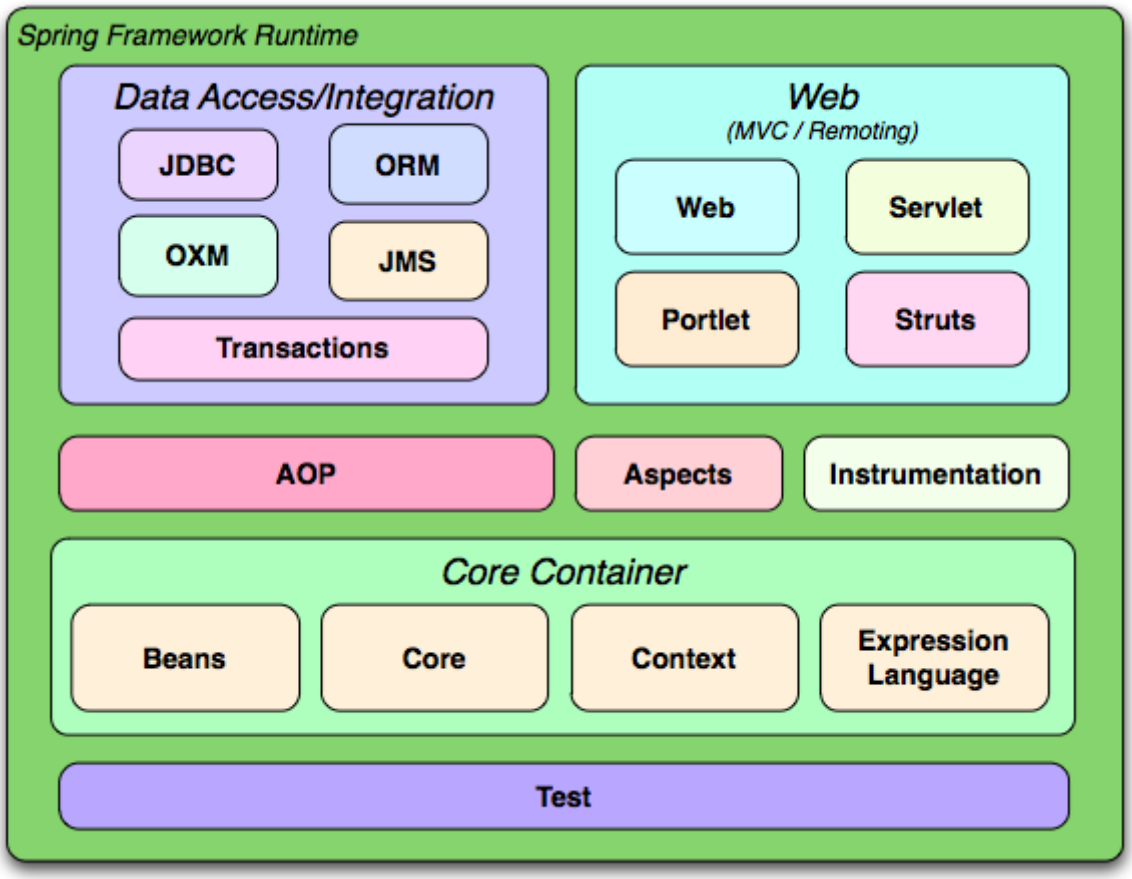
Czym jest Spring?

Spring Framework jest to platforma, której głównym celem jest uproszczenie procesu tworzenia oprogramowania klasy enterprise w technologii Java/J2EE. Rdzeniem Springa jest kontener wstrzykiwania zależności, który zarządza komponentami i ich zależnościami. Umożliwia on automatyczne wykrywanie tych zależności bez większego udziału programisty. Nie ma także problemu z własnoręczną konfiguracją – jeśli taki sposób pracy bardziej nam odpowiada. Cel jest jednak jeden – zmniejszenie stopnia związania klas. Artykuł ten ma za zadanie zaprezentować Springa właśnie w tym kontekście. Ale o tym za chwilę – najpierw kilka słów o architekturze frameworka.

Architektura Spring Framework

Spring jest rozwiązaniem modułowym. Bez problemu możemy wykorzystać jedynie te części, których potrzebujemy. W skrócie omówię niektóre z nich.

- Core Container
  - Core i Beans – podstawowe moduły, zawierają funkcjonalność Inversion of Control i Dependency Injection, będące tematem przewodnim niniejszego artykułu. To dzięki nim możliwe jest oddzielenie konfiguracji i specyfikacji zależności od logiki biznesowej.
  - Context – zapewnia dostęp do obiektów na poziomie frameworka w sposób analogiczny do JNDI. Do-



Ilustracja 1: Architektura Spring Framework, źródło: Spring Framework Reference

> Sposobem na wyeliminowanie zależności z klasy jest stworzenie zależności poza klasą i wstrzyknięcie jej do środka.

daje wsparcie dla internacjonalizacji i propagowania zdarzeń, a także EJB i JMX.

- DAO – zapewnia wsparcie dla metod utrwalnia obiektów, w szczególności JDBC, ORM, OXM (mapowanie XML), JMS (tworzenie i przetwarzanie wiadomości). Dostarcza gotową do wykorzystania pulę połączeń, a także możliwość deklaratywnego definiowania transakcji. Pozwala na łatwe mapowanie ResultSetów na listę obiektów klas domenowych.
- Web – zawiera własny framework webowy – Spring Web MVC. Pozwala także wykorzystywać inne technologie, np. Struts, JSF, Velocity. Wspomaga proces ładowania plików na serwer.
- AOP – wspiera programowanie zorientowane aspektowo zarówno w wydaniu prostszym (Spring AOP), jak i bardziej rozbudowanym (AspectJ).
- Test – zawiera mechanizmy służące do testowania aplikacji (JUnit lub TestNG). W szczególności dostarcza mocków.

O wstrzykiwaniu zależności

Zrozumienie celowości Dependency Injection i Inversion of Control często stanowi problem dla początkujących programistów. Spróbujmy zatem omówić to na przykładzie.

Wady ścisłego wiązania klas

Na początku uczyliśmy się, że obiektowe języki programowania pozwalają na modelowanie otaczającego nas świata. To prawda, ale niestety obiekty domenowe nie są zbyt użyteczne bez otaczającego je kontekstu. Rozważmy związek pomiędzy obiektem dostępu do danych (DAO), a obiektem klasy DataSource, reprezentującym bazę danych. Oczywiście obiekt DAO zależy od bazy danych. W klasycznym podejściu wyglądałoby to tak:

```
public class JdbcBookDao
  implements BookDao {
  private BasicDataSource
    dataSource;

  public JdbcBookDao() {
    dataSource =
      new BasicDataSource();
    dataSource.setDriverClassName(
      "com.mysql.jdbc.Driver");
    dataSource.setUrl(
      "jdbc:mysql://localhost/
      javaexpress?autoReconnect=
      true");
    dataSource.setUsername(
      "username");
    dataSource.setPassword(
      "password");
  }
  // ...inne metody
}
```

Problemem, który pierwszy rzuca się w oczy jest tutaj zaszycie parametrów dostępu do bazy danych w kodzie. Można to rozwiązać poprzez wykorzystanie klasy java.util.Properties, ale to dopiero początek. Kolejną bolączką jest uzależnienie się od konkretnej implementacji (org.apache.commons.dbcp.BasicDataSource), co wpływa ujemnie na elastyczność rozwiązania. Zwróćmy uwagę, że posługiwanie się samym interfejsem niczego nie zmieni, ponieważ do użycia przedstawionych metody potrzebujemy konkretnej klasy.

Wstrzykiwanie zależności

Sposobem na wyeliminowanie konkretnej zależności z klasy JdbcBookDao jest stworzenie zależności poza klasą i wstrzyknięcie jej do środka. To zapewnia większą elastyczność, ponieważ konfigurację możemy

> Teraz obiekt DAO nie jest ściśle związany z żadną klasą, co jest na pewno krokiem naprzód

modyfikować nie naruszając klasy, która z niej korzysta. Nie dość, że minimalizujemy w ten sposób ryzyko powstania błędu, to unikamy konieczności modyfikacji bytu, do którego źródeł nie musimy mieć przecież dostępu! W kodzie wyglądałoby to tak:

```
public class JdbcBookDao
  implements BookDao {
  private DataSource dataSource;

  public void setDataSource(
    DataSource dataSource) {
    this.dataSource = dataSource;
  }
  // ...inne metody
}
```

Teraz obiekt DAO nie jest ściśle związany z żadną klasą, co jest na pewno krokiem naprzód, ale w tym momencie tylko przenieśliśmy konstrukcje obiektu wyżej, do klienta:

```
public class BookService {
  private JdbcBookDao bookDao;

  public BookService() {
    try {
      Properties props =
        new Properties();
      props.load(
        new FileInputStream(
        "dataSource.properties"));

      BasicDataSource dataSource =
       new BasicDataSource();
      dataSource.
        setDriverClassName(
          props.getProperty(
            "driverClassName"));
      dataSource.setUrl(
       props.getProperty("url"));
      dataSource.setUsername(
        props.getProperty(
        "username"));
      dataSource.setPassword(
        props.getProperty(
        "password"));
      bookDao = new JdbcBookDao();
      bookDao.setDataSource(
        dataSource);

    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Co gorsza, w ten sposób stworzyliśmy zależność między BookService, a dwoma klasami konkretnymi: BasicDataSource oraz JdbcBookDao, co właściwie oznacza początek do punktu wyjścia. Na szczęście nie oznacza to, że nasze eksperymenty poprowadziły nas w złym kierunku. Teraz musimy tylko przenieść proces wstrzykiwania do zakresu odpowiedzialności innego bytu.

Inversion of Control

W tradycyjnym programowaniu, to w zakresie odpowiedzialności klienta jest pobieranie wymaganych przez niego komponentów. W przypadku IoC, kontener zajmuje się procesem wstrzykiwania zależności. Klient prosi tylko o pewien byt, ale to kontener decyduje jak i kiedy go dostarczyć. Jak ma się to naszego przykładu?

Otóż klient nie musi bezpośrednio inicjalizować obiektu klasy JdbcBookDao. Zamiast tego, odpowiedni obiekt jest wstrzykiwany do klasy BookService.

```
public class BookService {
  private BookDao bookDao;

  public void setBookDao(
    BookDao bookDao) {
    this.bookDao = bookDao;
  }
}
```

A gdzie kończy się łańcuch zależności? W pliku konfiguracyjnym Springa:

> Najważniejsze jest to, iż cała konfiguracja jest w pełni transparentna do klasy usługi, co zapewnia luźnie wiązanie klas.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"> <!-- (1) -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url"
              value="jdbc:mysql://localhost/javaexpress?
                     autoReconnect=true" />
    <property name="username" value="username" />
    <property name="password" value="password" />
  </bean>

  <bean id="bookDao" class="dao.JdbcBookDao"> <!-- (2) -->
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="bookService" class="service.BookService"> <!-- (3) -->
    <property name="bookDao" ref="bookDao" />
  </bean>
</beans>
```

W miejscu oznaczonym przez (1) definiujemy źródło danych. Na uwagę zasługuje tu atrybut „destroy-method", w którym podajemy nazwę metody, która zostanie wywołana w chwili usuwania obiektu DataSource. W (2) deklarujemy obiekt klasy JdbcBookDao. Jak pamiętamy, klasa ta ma jedno pole – dataSource – oraz publiczny setter je ustawiający. Dzięki nim możliwe jest wstrzyknięcie uprzednio określonego źródła danych do obiektu. Powiązanie między beanem DataSource oraz JdbcBookDao zapewnia ten sama fraza będąca odpowiednio identyfikatorem komponentu i wartością atrybutu ref własności dataSource. Analogiczna procedura zachodzi w (3).

W tym momencie łańcuch zależności został zakończony i wszystkie niezbędne obiekty dostarczone do klasy BookService. Najważniejsze jest to, iż cała konfiguracja jest w pełni transparentna do klasy usługi, co zapewnia luźnie wiązanie klas.

Przejdźmy teraz do konkretnej implementacji i stwórzmy w pełni działający przykład.

IoC w Springu – podstawy

Nasza aplikacja będzie bardzo prostą implementacją narzędzia do obsługi domowej biblioteczki. Pozycje, które posiadamy na półce są zapisane w pliku o odpowiednim formacie. W szczególności przechowywana jest tam data wypożyczenia książki. Zadaniem aplikacji będzie wyszukanie pozycji, które wypożyczyliśmy ponad 30 dni temu.

Tworzenie klasy domenowej

Aplikacja zawiera tylko jedną klasę domenową, reprezentującą książkę:

> Powierzymy zadanie wstrzykiwania tej nazwy Springowi.

```
package model;

//... pominięto importy
public class Book {
  private String title;
  private String author;
  private Date lendDate;

  public Book(String title,
    String author, Date lendDate) {
    this.title = title;
    this.author = author;
    this.lendDate = lendDate;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(
    String title) {
    this.title = title;
  }

  public String getAuthor() {
    return author;
  }

  public void setAuthor(
    String author) {
    this.author = author;
  }

  public Date getLendDate() {
    return lendDate;
  }

  public void setLendDate(
    Date lendDate) {
    this.lendDate = lendDate;
  }
}
```

Jak wspomniano, nasze książki przechowywać będziemy w pliku CSV, np. takim:

Spring in Action,Craig Walls,20091028

Java Persistence with Hibernate,Christian Bauer,20091230

Spring in Practice,Willie Wheeler,20091127

Pierwsze pole to tytuł książki, drugie autor, a ostatnie – data wypożyczenia w formacie yyyyMMdd.

Tworzenie warstw DAO i usług

Kolejnym etapem jest konstrukcja interfejsu DAO. Będzie on zawierał tylko jedną metoda – wyciąganie wszystkich książek.

```
package dao;

//... pominięto importy
public interface BookDao {
  List<Book> findAll() throws
    Exception;
}
```

Teraz przyszła kolej na implementacje interfejsu w postaci klasy odczytującej dane z pliku. Istotne z naszego punktu widzenia jest założenie, że nazwa i ścieżka do tego pliku może się zmieniać, więc nie chcemy przechowywać jej na stałe w kodzie. Powierzymy zadanie wstrzykiwania tej nazwy Springowi.

```
package dao;

//... pominięto importy
public class CsvBookDao implements
  BookDao {
  private String csvBooksFile;

  public void setCsvBooksFile(
    String csvBooksFile) {
    this.csvBooksFile =
      csvBooksFile;
  }

  @Override
  public List<Book> findAll() throws
    Exception {
    List<Book> booksList =
      new ArrayList<Book>();

    DateFormat df =
      new SimpleDateFormat(
        "yyyyMMdd");

    BufferedReader br =
```

> Które rozwiązanie stosować?
> To tak naprawdę kwestia gustu i wyczucia

```
      new BufferedReader(
      new FileReader(
      csvBooksFile));
    String line;
    while ((line = br.readLine())
      != null) {
      String[] fields =
        line.split(",");
      // pomijamy kwestie blednego
formatu pliku
      String title = fields[0];
      String author = fields[1];
      Date lendDate =
        df.parse(fields[2]);
      Book book =
        new Book(title, author,
          lendDate);
      booksList.add(book);
    }
    br.close();
    return booksList;
  }
}
```

Jak widzimy, klasa zawiera pole prywatne przechowujące ścieżkę do pliku. Ponadto stworzyliśmy publiczny mutator (setter), który aktualizuje wartość tego pola. Dzięki temu Spring może wstrzyknąć odpowiednią wartość z plików konfiguracyjnych.

Implementacja metody findAll() nie ma tak naprawdę wiele wspólnego z samym Springiem i polega na odczytywaniu kolejnych linii pliku, a następnie dzieleniu tych linii na części odpowiadające poszczególnym polom klasy Book.

Ostatnią niemal klasą, którą musimy zbudować jest klasa usługi – Book Service. Jej zadaniem będzie wybranie tych książek spośród wszystkich w bazie, które pożyczyliśmy co najmniej 30 dni temu.

Wstrzykiwanie przez setter lub konstruktor

Metoda, której używamy tutaj do wstrzykiwania zależności nazywa się wstrzykiwaniem przez setter. Spring pozwala także na wykorzystanie wstrzykiwania przez konstruktor.

Zamiast setterów, które ustawiają poszczególne pola klasy, możemy wykorzystać konstruktor z argumentami odpowiadającymi tym polom. W naszej klasie BookService moglibyśmy zastąpić setter przez następujący konstruktor:

```
public BookService(BookDao bookDao) {
  this.bookDao = bookDao;
}
```

Musielibyśmy też zmodyfikować plik konfiguracyjny:

```
<bean id="bookService" class="service.BookService">
  <constructor-arg ref="bookDao" />
</bean>
```

Które rozwiązanie stosować? To tak naprawdę kwestia gustu i wyczucia. Ogólna zasada mówi, żeby stosować wstrzykiwanie przez konstruktor dla pól, które są niezbędne dla poprawnego działania klasy, a wstrzykiwanie przez setter dla wszystkich pozostałych. Z drugiej strony używanie wyłącznie setterów pozwala na modyfikowanie wartości pól już po utworzeniu obiektu, co wpływa na elastyczność. Problem ten został dogłębniej omówiony w źródle (4).

> Konfiguracja aplikacji opartej o Springa najczęściej odbywa się w pliku XML

```
package service;

//... pominięto importy
public class BookService {
  private BookDao bookDao;

  public void setBookDao(
    BookDao bookDao) {
    this.bookDao = bookDao;
  }

  public List<Book>
    findBooksLent30DaysAgo()
      throws Exception {
    List<Book> booksLent30DaysAgo
      = new ArrayList<Book>();
    List<Book> allBooks =
      bookDao.findAll();

    Date thirtyDaysAgo =
      daysAgo(30);
    for (Book book : allBooks) {
      boolean bookWasLent30DaysAgo
        = book.getLendDate()
          .compareTo(
            thirtyDaysAgo)
          <= 0;
      if (bookWasLent30DaysAgo)
        booksLent30DaysAgo.
          add(book);
    }
    return booksLent30DaysAgo;
  }

  private Date daysAgo(int days) {
    GregorianCalendar gc =
      new GregorianCalendar();
    gc.add(Calendar.DATE, -days);
    return gc.getTime();
  }
}
```

Implementacja jest dość oczywista. Na początku deklarujemy pole typu BookDao, do którego obiekt zostanie wstrzyknięty dzięki odpowiedniemu setterowi. Następnie używamy obiektu DAO, żeby wyciągnąć wszystkie książki z pliku. Ostatnim już etapem jest skopiowanie obiektów spełniających określone warunki do nowej listy.

Czas na ostatni krok – konfigurację.

Konfiguracja warstw DAO i usług w Springu

Konfiguracja aplikacji opartej o Springa najczęściej odbywa się w pliku XML. Konwencja zakłada, że plik taki nosi nazwę applicationContext.xml, ale nie jest to regułą. Co więcej, nie jesteśmy wcale ograniczeni do jednego pliku konfiguracyjnego – możemy np. stworzyć osobne pliki dla warstw DAO, usług, czy bezpieczeństwa. W naszym prostym przykładzie w zupełności wystarczy jednak jeden plik o konwencjonalnej nazwie.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframe-
work.org/schema/beans" (
   xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
      xsi:schemaLocation="http://www.
springframework.org/schema/beans
http://www.springframework.org/sche-
ma/beans/spring-beans.xsd">

  <bean id="bookDao"
        class="dao.CsvBookDao">
    (2)
    <property name="csvBooksFile"
    value="/home/marcin/books.txt" />
    (3)
  </bean>

  <bean id="bookService"
        class="service.BookService">
    (4)
    <property name="bookDao"
              ref="bookDao" />
  </bean>
</beans>
```

Spring dostarcza wiele różnych schematów do konfigurowania różnych elementów. Tutaj (1) wykorzystaliśmy podstawowy schemat beans, służący do konfiguracji komponentów. W (2) definiujemy komponent klasy CsvBookDao o identyfikatorze bookDao. Klasa ta ma jedno pole – csvBookFile typu String. W (3) ustalamy wartość tego pola. Spring używa refleksji, żeby utworzyć obiekt żądanej klasy, a także zainicjować jej pole. Dzięki temu zabiegowi

> Jesteśmy właściwie u celu.
> Pozostaje przetestowanie zaimplementowanego rozwiązania.

będziemy mogli w każdej chwili zmienić lokalizację pliku z książkami, co było naszym celem.

Analogicznie, w (4) definiujemy komponent klasy BookService, a jego pole bookDao inicjalizujemy referencją do uprzednio zdefiniowanego komponentu. Zwróćmy uwagę, że w atrybucie ref używamy tego samego ciągu znaków, co w atrybucie id wykorzystywanego obiektu.

Testujemy!

Jesteśmy właściwie u celu. Pozostaje przetestowanie zaimplementowanego rozwiązania. Ja wykorzystam bibliotekę JUnit 4, ale tak naprawdę nie ma to większego znaczenia – można nawet stworzyć prostą klasę, która po prostu wypisze oczekiwane pozycje książkowe.

```
package service;

//...pominięto importy
public class BookServiceTest {
  @Test
  public void getBooksLent30DaysAgo()
    throws Exception {
    ApplicationContext appCtx =
    new
      ClassPathXmlApplicationContext(
      "META-INF/spring/
       applicationContext.xml");
    BookService bookService =
      (BookService) appCtx.
      getBean("bookService");

    List<Book> booksLent30DaysAgo =
      bookService.
        findBooksLent30DaysAgo();

    assertEquals(2,
      booksLent30DaysAgo.size());
    assertEquals("Spring in Action",
      booksLent30DaysAgo.get(0).
      getTitle());
    assertEquals(
      "Spring in Practice",
      booksLent30DaysAgo.get(1).
        getTitle());
  }
}
```

Najważniejsze są tutaj dwie pierwsze linie metody testującej. Na początku tworzymy obiekt klasy ClassPathXmlApplicationContext, do konstruktora której podajemy względną ścieżkę do pliku konfiguracyjnego. Dzięki tej klasie, możemy uzyskać referencję do każdego komponentu zdefiniowanego w pliku XML za pomocą wartości jego atrybutu id. Następnie za jej pomocą uzyskujemy referencję do naszej klasy usług, którą już możemy się posługiwać w standardowy sposób.

Podsumowanie

Zbudowaliśmy prostą aplikację wykorzystującą Spring Framework. Już teraz widać, że użycie DI sprawiło, iż jej moduły są lepiej odseparowane, a całość łatwiejsza w utrzymaniu. A to przecież tylko banalny przykład i raptem wierzchołek góry lodowej zwanej DI w Springu. Przed nami korzystanie z PropertyPlaceholderConfigurer, automatyczne wiązanie zależności oraz skanowanie komponentów, adnotacje i wiele więcej. Ale to temat na kolejny artykuł. Zainteresowanych poszerzeniem wiedzy już teraz zapraszam do ramki źródła.

Źródła

- Spring in Practice, Willie Wheeler, John Wheeler, Manning Publications – książka będąca inspiracją i podstawą do napisania tego artykułu
- Spring in Action, Craig Walls, Ryan Breidenbach, Manning Publications
- http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/
- http://martinfowler.com/articles/injection.html#ConstructorVersusSetterInjection

GlassFish Enterprise: 5 9's z HADB

Mirosław Dąbrowski

Wprowadzenie. Kilka słów o GlassFish'u,

Serwer aplikacyjny dostępny od Sun Microsystems pierwotnie miał swe początki jako alians iPlanet (kooperacja inżynierów Sun Microsystems z NetScape). Po okresie wspólnej pracy z NetScape, Sun projekt iPlanet rozwijał pod swoimi skrzydłami już pod nazwą SunONE, Sun Open Net Environment. Nazwa ta została wprowadzona w 2002 roku i była przewidziana na grupę produktów, które powstały w wyniku aliansu. SunONE z kolei przekształcił się w linię produktów odpowiedzialnych za warstwę webową (serwer webowy, aplikacyjny, portal etc.) obecnych w ofercie pod nazwami Sun Java System. Debiut SJS nastąpił podczas konferencji SunNetwork we wrześniu 2003 roku. W ramach SJS powstał Sun Java System Application Server, który obecnie jest starą nazwa na wspierany serwer aplikacyjny oferowany od Sun Microsystems. Sun Java System Application Server v9.1 to ostatnia wersja z działu aplikacyjnych serwerów z nazwa SJSAS a jego odpowiednikiem (bit w bit identyczna implementacja, jedynie różny instalator i kilka graficznych plików) jest Sun GlassFish Enterprise Application Server v2.1.

Dziś GlassFish Enterprise jest aktualną nazwą dla tego serwera (wprowadzona w kwietniu 2008 roku). Glass w nazwie oznacza przejrzystość kodu, gdyż pierwotnie serwer aplikacyjny był OpenSource, ale z uwagi na jej popularność Sun postanowił oferować ten produkt w wersji Enterprise. W wersji 2.x (obecnie najnowsza 2.1.1) Sun GlassFish Enterprise Application Server jest serwerem aplikacyjnym, będącym implementacją specyfikacji JSR 244 rozwijanej w ramach JCP, szerzej znanej pod nazwa Java Enterprise Edition 5 lub w skrócie JEE5. Co oznacza, że tak jak RedHat JBoss, IBM WebSphere czy Oracle Weblogic, GlassFish jest jednym z kontenerów, służących jako środowisko do hostowania aplikacji/usług utworzonych przede wszystkim (choć nie tylko) w technologii JEE. Jest nie tylko jednym z serwerów, ale obecnie serwerem o najniższych kosztach

> GlassFish w swojej ofercie posiada dwa mechanizmy do przetrzymywania sesji.

wsparcia 7x24x365 (stałe, niezależne od ilości CPU, rdzeni, socket'ów), serwerze o dostępność na poziomie 5 9's (realizowana przez HADB), serwerem o najwyższej wydajność (aktualne dane niezależnego konsorcjum SPEC - www.spec.org), serwerem o największej ilości pobrań (ponad 4 mln) oraz przede wszystkim serwerem będącym od wielu lat platformą referencyjną (pierwszy certyfikowany, w pełni zgodny ze specyfikacją serwer aplikacyjny JEE5, oraz JEE6 w wersji v3).

Te i inne fakty sprawiają, że coraz częściej GlassFish używany jest w środowiskach produkcyjnych. Wraz z HADB, zapewnia bardzo wysoki poziom dostępności usług na poziomie 5 9's. Oznacza to dostępność na poziomie 99.999% a innymi słowy maksymalny dozwolony downtime naszych usług na poziomie 5 minut w ciągu roku. 5 9's jest standardem dla usług telekomunikacyjnych, i wszędzie tam, gdzie tolerancja utraty sesji jest bardzo niska. A oprócz samej sesji, HADB pozwala na stałość usługi JMS, usługi do asynchronicznego przesyłania wiadomości.

O artykule

Poniższy artykuł omawia w teorii technologię HADB jak również krok po kroku pokazuje budowę w oparciu o zwirtualizowane środowisko VMware. Wymagana jest jedynie podstawowa wiedza na temat środowisk Unix'owych, aby móc swobodnie się w nim poruszać. Wszystkie czynności przeprowadzane podczas tego artykuł są niezmiernie szczegółowe, aby każdy mógł bez problemu poradzić ze wszystkimi etapami.

Sekcje komend w artykule opatrzone kolorem oznaczają uruchomienie danej komendy. Natomiast sekcje opatrzone kolorem oznaczają przebieg danej komendy (odpowiedzi na pytania, podsumowania po wykonaniu komendy, etc.).

Przygotowania do pracy

Aby móc przeprowadzić ćwiczenia wymagane jest pobranie ze stron Sun Microsystems obrazu Solaris'a 10 z utworzonymi zonami. Obraz tem posiada 12 skonfigurowanych zon – 12 zwirtualizowanych Solaris'ów z których 4 będą pełnić w naszym środowisku rolę instancji GlassFish'a będących w jednym klastrze. Link do obrazu poniżej (login: „**root**", hasło: „**cangetin**").

Po co, dlaczego? Czym jest HADB ?

GlassFish w swojej ofercie posiada dwa mechanizmy do przetrzymywania sesji. **In-memory replication** jako szybkość i open-source, oraz **HADB**, stabilność na wysoką skalę, wsparcie oraz replikacja JMS. Oba mechanizmy dają podstawową funkcjonalność – replikacja sesji. Do przetrzymywania samej sesji **in-memory replication** ma przewagę wydajności (RAM) i jest lepszym rozwiązaniem na małą skalę. Jeśli jednak ilość naszych sesji przekracza (zależne od środowiska, aplikacji, zasobów i innych współczynników) 100000 aktywnych sesji, mechanizm HADB staje się niezastąpiony w przypadku tak dużego obciążenia.

HADB, High Availiability Database jest to mechanizm rozproszonej bazy danych, której repozytorium są w plikach przechowywane na każdym z nodów. Baza HADB służy jako repozytorium sesji i wiadomość JMS i przewidziana jest do dużych systemów w których tolerancja utraty sesji oraz wysoka dostępność mają spełnić standardy 5 9's.

> Domain Administration Server (DAS)
> jest to centralny serwer w architekturze GlassFish'a

GlassFish a GlassFish Enterprise oraz historia wersji.

Serwer GlassFish jest dostępny w dwóch wersjach – **OpenSource** (https://glassfish.dev.java.net/) oraz **Enterprise** (http://www.sun.com/software/products/appsrvr/index.jsp). Mowa tu o GlassFish Application Server oraz GlassFish Enterprise Application Server. **Obydwie wersie posiadają** podobną implementację z tą różnicą, że Enterprise posiada oficjalny suport od strony Sun Microsystems, moduły do HADB oraz zestaw trzech, dodatkowych, płatnych narzędzi dostępnych w pakiecie **Enterprise Manager – Performance Advisor**, **Performance Monitor** oraz **SNMP Monitor**. Z uwagi na HADB będziemy pracować z wersją Enteprise.

Rozwój każdej z edycji podzielony jest na 3 fazy:

- **Concept Creation** – zbieranie kluczowych wymogów, określanie ram czasowych, prototypowanie.
- **Active Development** – implementacja w kierunku Milestone aż do wersji finalnej.
- **Maintenance** – poprawki finalnej edycji, wydania produktu z poprawkami.

Pierwszy GlassFish v1 miał swoja premierę w maju 2006 roku i jego odpowiednikiem w ofercie Sun'a był „Sun Java System Application Server 9.0 PE". Był on implementacja specyfikacji JEE5, jednak z kilkoma brakami jak możliwość klastrowania. GlassFish v2 miał premierę we wrześniu 2007 roku wraz z wersją wspieraną przez Sun'a o nazwie „Sun Java System Application Server 9.1". Te wersje wspierały już klaster, load balancing jak również replikacja sesji przez pamieć. Wersia SJSAS 9.1 wspierała HADB, która od kwietna 2008 roku zmieniła nazwa na GlassFish Enterprise v2.1.v2.1 w grudniu 2009 otrzymała update do wersji v2.1.1 prezentujący drobne poprawki w produkcie.

Wersja v3 GlassFish'a dostępna od grudnia 2009 roku jest pierwsza na w pełni zgodną implementacja serwera aplikacyjnego z JEE6. GlassFish v3 jest serveram modułowym (bazując na OSGI, implementacja Apache Felix) w którym moduły instaluje się łącząc się przez telnet do DAS'a na port 6666. GlassFish v3 jest dostępny w dwóch profilach: WEB – Servlets, JSP, JavaBeans, POJO, oraz Full – Servlets, JSP, JavaBeans, POJO, EJB.

Suszona rybka. Wpierw trochę teorii.

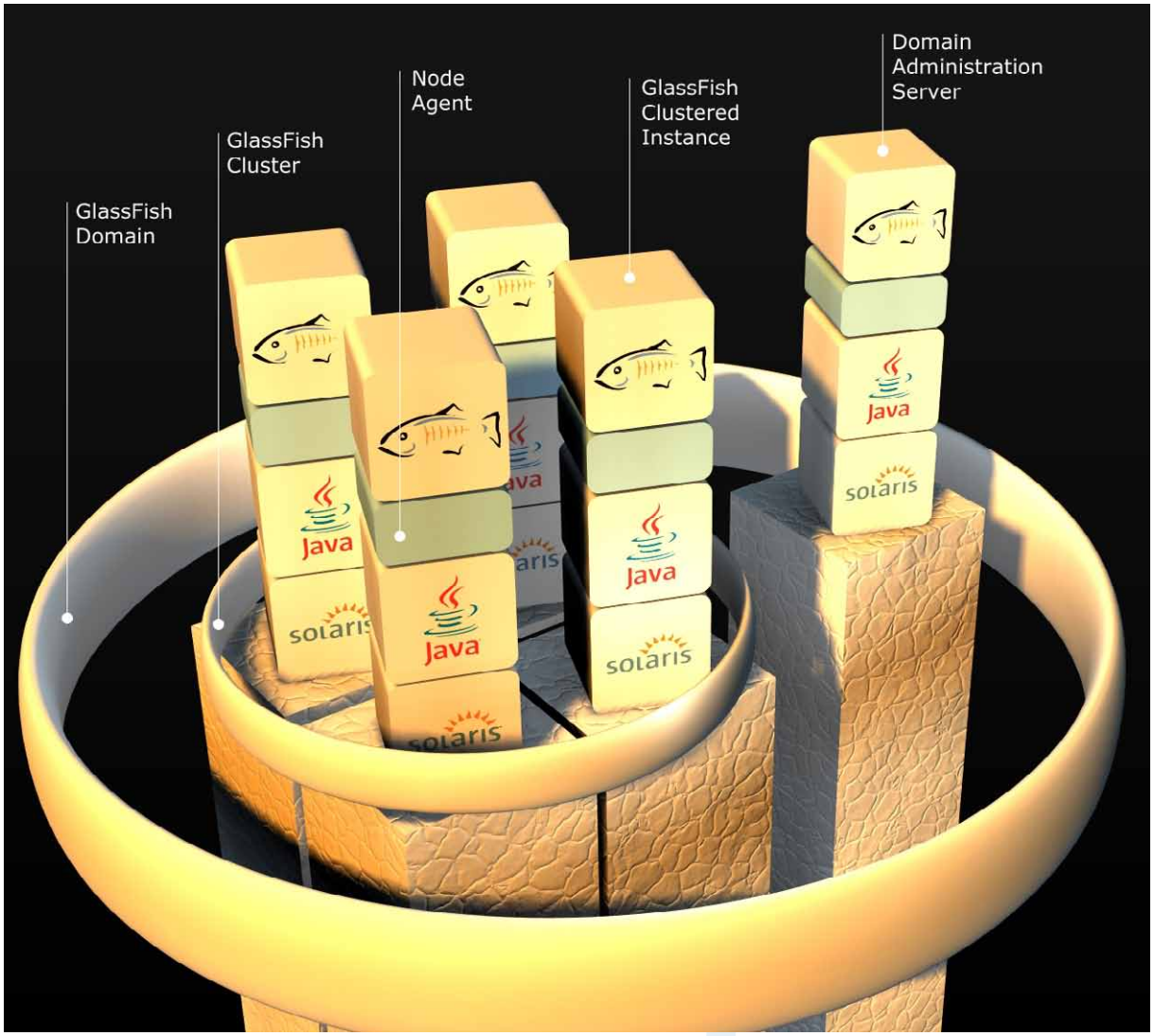
W architekturze serwera aplikacyjnego GlassFish występuje kilka pojęć, które warto przypomnieć zanim zaczniemy pracę, abyśmy zrozumieli podstawowe słownictwo występujące podczas instalacji oraz konfiguracji naszego środowiska.

Na rysunku 1 mamy graficzną prezentację przykładowego środowiska na którym będziemy pracować. W celu wyjaśnienia wszystkich elementów wchodzących w jego sład, kolejno opiszemy każdy z nich.

Domain Administration Server (DAS) - jest to centralny serwer w architekturze GlassFish'a. DAS to instancja administracyjna o nazwie server, odpowiadająca za centralny monitoring, logowanie, wdrażania aplikacji, konfiguracje zasobów dla klastrów i instancji, utworzonych w obrębie domeny. DAS jest miejscem gdzie administrujemy domeną.

Uwaga: Zalecany rozmiar heap dla tego procesu to 512MB.

> Klaster - jest to nazwany zbiór homogenicznych instancji.



Rysunek 1 - Przykładowa architektura. 1 domena, 1 DAS, 1 klaster, 4 agenty, 4 instancje. Całość działa na systemie Solaris.

Domena - z kolei jest to zbiór instancji, klastrów wchodzących w jej skład mających wspólną konfigurację i udostępnione zasoby w jej obrębie. Każda domena ma swego DAS'a (ma którym trzymane jest centralne repozytorium aplikacji) oraz profil domeny, który określa jej możliwości. Jest on ustalany podczas jej tworzenia. Dostępne są trzy przedefiniowane profile: **Developer, Cluster, Enterprise**. Ostatni jako jedyny dostępny jest w wersji komercyjnej (GlassFish Enterprise) i jako jedyny ma wsparcie do HADB. Konfiguracja domeny jest całościowo przeprowadzana na DAS'ie i przechowywana w pliku domain.xml również na DAS'ie.

Klaster - jest to nazwany zbiór homogenicznych instancji. Homogeniczność określana jest jedynie po wersji aplikacyjnego serwera. Każdy klaster istniejący w domenie (może być ich kilka) musi mieć unikatową nazwę. Jest tak ponieważ każdy klaster swą konfigurację przechowuje w katalogu, który ma nazwę określoną przez nazwę klastra. Każdy klaster należy zawsze i tylko do jednej domeny.

> Instalacja agenta jest jedyną czynnością jaką trzeba przeprowadzić na nowej maszynie w celu dodania jej do domeny GlassFish'a.

Instacja - klastrowa (**clustered instance**) bądź nie (**stand-alone instance**) jest to działający proces GlassFish'a. Proces JVM hostujący aplikacje (będące kopią aplikacji z centralnego repozytorium w DAS). Na jednej instalacji GlassFish'a może działać kilka jego instancji, każda nasłuchująca na innych portach HTTP, HTTPS, JMX, oraz IIOP.

Raz stworzona instancja stand-alone nie może być później dodana do klastra.

Uwaga: Zalecany rozmiar heap dla pojedynczego procesu instancji to 512MB.

Instancje podobnie jak klastry, muszą mieć unikatową nazwą w obrębie domeny i należą zawsze i tylko do jednej domeny. Jest tak z uwagi na ich konfigurację, przechowywaną w odrębnych katalogach. Jakakolwiek komunikacja z instancją następuje poprzez **agenta**.

Node Agent - jest to lekki proces uruchamiany ręcznie, lub skonfigurowany jako usługa w systemie operacyjnym (Windows 2000 - Service Contol, Windows XP i wyżej gotowy skrypt ma.cfg, Solaris do wersji 9 – init.d, Solaris 10 – usługa Service Management Facility), który odpowiada za komunikacje oraz synchronizację instancji wchodzących w skład domeny z DAS'em (synchronizacje lokalnych repozytoriów z centralnym). Proces ten jest wymagany aby móc zainstalować instancję. Jedyna instancja, która nie wymaga agenta to DAS.

Instalacja agenta jest jedyną czynnością jaką trzeba przeprowadzić na nowej maszynie w celu dodania jej do domeny GlassFish'a. Po jego instalacji dodawanie/instalacji instancji na nowej maszynie przeprowadzane jest zdalnie z DAS'a.

Proces agenta jest aktywny jedynie podczas cyklu życia instancji obsługiwanych przez niego (tworzenie, uruchamiania, usuwanie instancji, etc.).

Uwaga: Zalecany rozmiar heap dla procesu agenta to 256MB.

Tyle teorii czas przejść do praktyki.

Główno dowodzący ławicy. Instalacja DAS.

Budowę naszej architektury zaczniemy od instalacji DAS'a w tym celu pobierzemy instalatora. Aby móc korzystać z mechanizmu HADB musimy pobrać instalator GalssFish v2.x Enterprise (wersja Open-Source nie posiada wsparcia do HADB) w wersji z modułami do bazy wysokiej dostępności HADB (dopisek with HADB). Jest to oddzielna instalacja dostępna na stronach Sun Microsystems pod nazwa: **Sun GlassFish Enterprise Server v2.1.1 with HADB**. Jest do pobrania pod adresem podanym poniżej (stan na luty 2010):

http://www.sun.com/software/products/appsrvr/get_it.jsp

GlassFish v2.1.1 dostępny jest w wersjach językowych: English (**sges_ee-2_1_1-solaris-i586.bin**) oraz Multi Language (**sges_ee-2_1_1-solaris-i586-ml.bin**) z dopiskiem ml w nazwie. Obydwie wersje są odpowiednia do naszych ćwiczeń.

Po pobraniu instalatora przeprowadźmy instalacje naszej domeny a mówiąc konkretnie instancję DAS'a. Wpierw uruchamiamy i logujemy się do zony w której będziemy instalować DAS'a. W naszym przypadku to zona o nazwie **zone01**.

Uruchamiamy zonę zone01:

```
global # zoneadm -z zone01 boot
```

Logujemy:

> Po zalogowaniu do zony i po wgraniu naszego instalatora musimy dać na nim uprawniania.

```
global # zlogin zone01
[Connected to zone ,zone01' pts/14]
Last login: Thu Feb 11 07:36:54 on
pts/4
Sun Microsystems Inc.   SunOS 5.10
Generic January 2005
zone01 #
```

Po zalogowaniu do zony i po wgraniu naszego instalatora musimy dać na nim uprawniania. Do tego użyjemy narzędzia **chmod** dając stosowane uprawnienia na pliku. Po dodaniu uprawnień z uwagi, że pracujemy z zonach instalator graficzny nie zostałby wyświetlony (wymaga to ustawienie zmiennej DISPLAY) instalujemy z przełącznikiem **-console**.

```
zone01 # cd glassfish/
zone01 # cd sges_v2.1.1_with_hadb/
zone01  #  chmod  744  ./sges_ee-
2_1_1-solaris-i586-ml.bin
zone01 # ./sges_ee-2_1_1-solaris-
i586-ml.bin - console
```

Po uruchomieni instalatora w trybie CLI ...

```
zone01 # ./sges_ee-2_1_1-solaris-
i586-ml.bin -console
Checking available disk space...
Checking Java(TM) 2 Runtime Envi-
ronment...
Extracting Java(TM) 2 Runtime En-
vironment files...
```

po krótkiej chwili uruchomi się instalator ...

```
Launching Java(TM) 2 Runtime Envi-
ronment...
Java Accessibility Bridge for GNO-
ME loaded.

You are running the installation
program for Sun GlassFish Enter-
prise Server
with HADB. This program asks you
to supply configuration preference
settings
that it uses to install the se-
rver.

The installation program consi-
sts of one or more selections that
provide you
with information and let you enter
preferences that determine how Sun
GlassFish
Enterprise Server with HADB is in-
stalled and configured.

When you are presented with the
following question, the installa-
tion process
pauses to allow you to read the
information that has been presen-
ted. When you
are ready, press Enter to conti-
nue.

 <Press ENTER to Continue>
Some questions require more deta-
iled information that you are re-
quired to type.
The question may have a default
value that is displayed inside of
brackets [].
For example, the following qu-
estion has a default answer of yes:

Are you sure? [yes]

If you want to accept the default
answer, press only the Enter key
(which on
some keyboards is labeled Return).

If you want to provide a diffe-
rent answer, type it at the com-
mand prompt and
then press Enter.

 <Press ENTER to Continue>
```

Enter.

```
Welcome to the Sun GlassFish Enterprise Server
with HADB Installation Program

<Press ENTER to Continue>
```

Enter.

```
Before you install this product,
you must read and accept the en-
tire
```

> Na jednej fizycznie maszynie może być kilka instalacji, każda innej wersji GlassFish'a

```
Software License Agreement under
which this product is licensed for
your use.
<Press ENTER to display the Soft-
ware License Agreement>
```

Enter. Zostanie wyświetlona na licencja produktu.

```
…
If you have read and accept all
the terms of the entire Software
License
Agreement, answer ‚yes', and the
installation will continue.

If you do not accept all the terms
of the Software License Agreement,
answer
‚no', and the installation program
will end without installing the
product.

Have you read, and do you accept,
all of the terms of the preceding
Software
License Agreement [no] {„<" goes
back, „!" exits}?
```

Wpisujemy „**yes**". Enter. Akceptujemy licencję. Po akceptacji licencji zaczyna się parametryzacja instalacji. Instalator poprosi o podanie miejsca instalacji.

```
The Sun GlassFish Enterprise Serv-
er with HADB components will be
installed in
the following directory, which is
referred to as the „Installation
Directory".
To use this directory, press only
the Enter key. To use a different
directory,
type in the full path of the di-
rectory to use followed by press-
ing the Enter
key.

Installation Directory [/opt/SUN-
Wappserver] {„<" goes back, „!"
exits}:
```

Prośba o podanie katalogu instalacji. Wybierzmy domyślny klikając Enter, czyli w przypadku dystrybucji na Solaris'a **/opt/SUNWappserver/.** Instalując kilka Instalacji (nie instancji) robimy to podając inne katalogi, natomiast instancje są instalowane w obrębie konkretnej instalacji. Co za tym idzie na jednej fizycznie maszynie może być kilka instalacji, każda innej wersji GlassFish'a, pracująca na tej same lub innej wersji JDK a w ramach tych instalacji kilka instancji klastrowanych lub nie podpiętych pod różne domeny.

Nasz domyślny katalog nie jest jeszcze stworzony w systemie więc instalator zapyta nas o to czy chcemy zmienić miejsce docelowe instalacji, czy utworzyć nieistniejący folder. Klikamy Enter co spowoduje stworzenie tego katalogu.

```
The directory „/opt/SUNWappserv-
er" does not exist.
Do you want to create it now or
choose another directory?

1. Create Directory
2. Choose New

Enter the number corresponding to
your choice  [1] {„<" goes back,
„!"  exits}
```

Następnie kolejno będziemy pytani o instalacji elementów GlassFisha.

```
Please choose components.
Do you want to install Node Agent
[yes] {„<" goes back, „!" exits}?
```

Enter. Instalacja agenta w tym przypadku jest na przyszłość. Gdybyśmy chcieli, aby na maszynie na którym działa DAS również działały inne instancje (klastrowane lub nie) wtedy wymagany byłby agent.

Z uwagi, że nasze środowisko jest oparte o zony i każda instancja wchodząca w skład klastra działa w obrębie innej zony,

> Na tym etapie nie będziemy przeprowadzać Load Balancing.

a maszyna na której działa DAS nie posiada innych instancji, agent jest całkowicie opcjonalny.

Jednak zalecam jego instalację ponieważ po nazwie jaką on otrzyma od instalatora możemy sprawdzić, czy mamy poprawnie skonfigurowany plik **/etc/hosts** z wpisem **FQDN**. Instalator pobiera z pliku hosts FQDN dla nazwy agenta. Jeśli po instalacji serwera nazwa agenta odpowiada FQDN naszej maszyny oznacza to, że środowisko na którym pracuje GlassFish (Solaris) jest poprawnie skonfigurowane.

```
Do you want to install High Availability Database Server [no] {„<” goes back, „!” exits}?
```

Wpisujemy **„yes”** Enter.

```
Do you want to install Load Balancing Plugin [no] {„<” goes back, „!” exits}?
```

Enter. Na tym etapie nie będziemy przeprowadzać Load Balancing.

```
Do you want to install Domain Administration Server [yes] {„<” goes back, „!” exits}?
```

Tak chcemy zainstalować DAS. Enter.

```
Do you want to install Sample Applications [yes] {„<” goes back, „!” exits}?
```

Opcjonalne. Enter. Po tej operacji instalator zapyta o JDK.

```
The Sun GlassFish Enterprise Server requires a Java 2 SDK.
Please provide the path to a Java 2 SDK 5.0 or greater.

1. Install Java 2 SDK (6.0)
2. Reuse existing Java 2 SDK
3. Exit

What would you like to do  [1] {„<” goes back, „!” exits}?
```

Wskażmy zainstalowane JDK z systemu. Wybieramy opcję **„2”**. Enter.

```
What would you like to do  [1] {„<” goes back, „!” exits}? 2

Please enter the path to an existing, compatible Java 2 SDK Version 1.5.0 or
above          [/usr/jdk/instances/jdk1.5.0] {„<” goes back, „!” exits}
```

Instalator bazując na zmiennej **JAVA_HOME** pobierze ścieżkę do JDK z systemu.

```
Supply the admin user’s password and override any of the other initial
configuration settings as necessary.

Admin User [admin] {„<” goes back, „!” exits}:
```

Nazwa dla administratora GlassFish’a. Trzymajmy się prostoty, nazwijmy do **„admin”**. Klikając enter przyjmujemy domyślną nazwę.

```
Admin User’s Password (8 chars minimum):
```

Hasło dla administratora: „**adminadmin”**.

```
Re-enter Password:
```

Powtórzenie Hasła dla administratora: **„adminadmin”**.

```
Master Password will be used as SSL certificate database password.
Master Password (8 chars minimum):
```

Master password. Hasło do bazy certyfikatów SSL GlashFisha. **„adminadmin”**.

```
Re-enter Master Password:
```

Powtórzenie Hasła Master: **„adminad-**

> Rozpoczynamy instalację.

min”.

Następnie podajemy porty na których będzie pracować nasz DAS. Wszystkie wybieramy domyślnie.

```
Admin Port [4848] {„<” goes back,
„!” exits}:
```

Port na którym będziemy łączyć się do konsoli administracyjnej, webowej o nazwie **Admin Console**. Dla domeny enterprises łączymy się przez HTTPS.

```
HTTP Port [8080] {„<” goes back,
„!” exits}:
```

Port HTTP. Enter.

```
HTTPS Port [8181] {„<” goes back,
„!” exits}:
```

Port HTTPS. Enter.

```
Please choose installation op-
tions.
Do you want to enable Updatecenter
client [yes] {„<” goes back, „!”
exits}?
```

Pytanie o instalacje UpdateCenter – aplikacji stand-alone (w wersji GlassFish’a v3 już zintegrowanej z Admin Console), służącej do aktualizacji zainstalowanych pluginów. Enter.

```
Do you want to upgrade from pre-
vious Application Server version
[no] {„<”goes back, „!” exits}?
```

Nie przeprowadzamy upgrade’u. Enter.

```
Checking disk space...

The following items for the prod-
uct Sun GlassFish Enterprise Serv-
er with HADB
will be installed:

Product: Sun GlassFish Enterprise
Server with HADB
Location: /opt/SUNWappserver
Space Required: 500.20 MB
--------------------------------
----------------
Java 2 SDK, Standard Edition 6.0
Uninstall
Sun Java System Message Queue 4.4
Application Server
Sample Applications
High Availability Database Server
High Availability Database Admin-
istration Client
Startup

Ready to Install

1. Install Now
2. Start Over
3. Exit Installation

What would you like to do [1] {„<”
goes back, „!” exits}?
```

Podsumowanie przed instalacją. Enter. Rozpoczynamy instalację.

```
Installing Sun GlassFish Enter-
prise Server with HADB
|-1%--------------25%------------
-----50%-----------------75%-----
---------100%|
Installation Successful.
Next Steps:

   1. Access the About Application
Server 2.1 welcome page at:
        file:///opt/SUNWappserver/
docs/about.html

   2. Start the Application Server
by executing:
          /opt/SUNWappserver/bin/
asadmin start-domain --user admin
domain1

    3. Start the Admin Console:
       https://localhost:4848

    4. Access sample applications:
        http://localhost:8080/sam-
ples/index.html

Please press Enter/Return key to
```

> Czas aby uruchomić DAS'a.

```
exit the installation program.
{"!" exits}
```

Na koniec zostaniemy poinformowani podsumowaniem instalacji.

Uruchamianie Domain Administration Server.

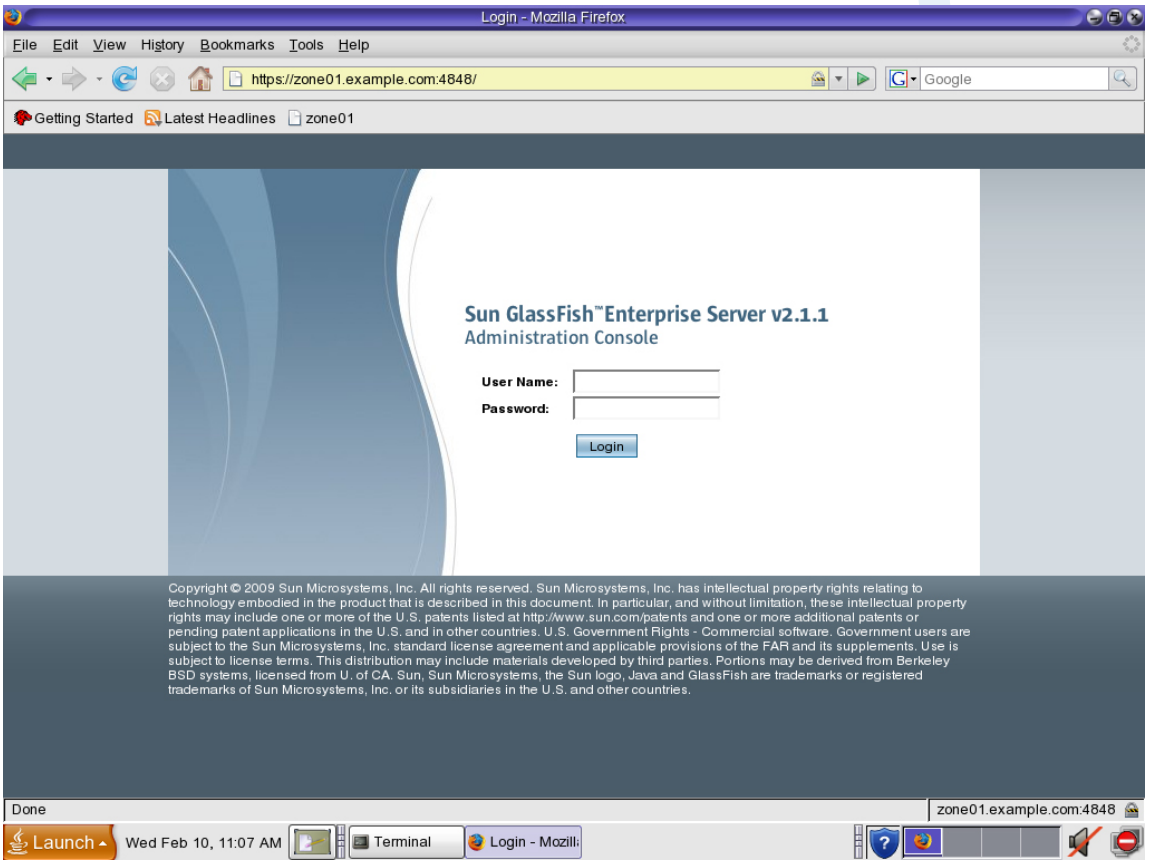
W tej chwili mamy zainstalowanego DAS'a oraz domenę **domain1** (instalowaną zawsze przy instalacji DAS'a) o profilu enteprise (z uwagi na wersje instalatora GlassFish Enterprise) wraz z modułami do HADB. Całość jest obecna w zonie **zone01**. Czas aby uruchomić DAS'a. W tym celu:

```
zone01 # cd /opt/SUNWappserver/bin/
zone01 # ./asadmin start-domain domain1
```

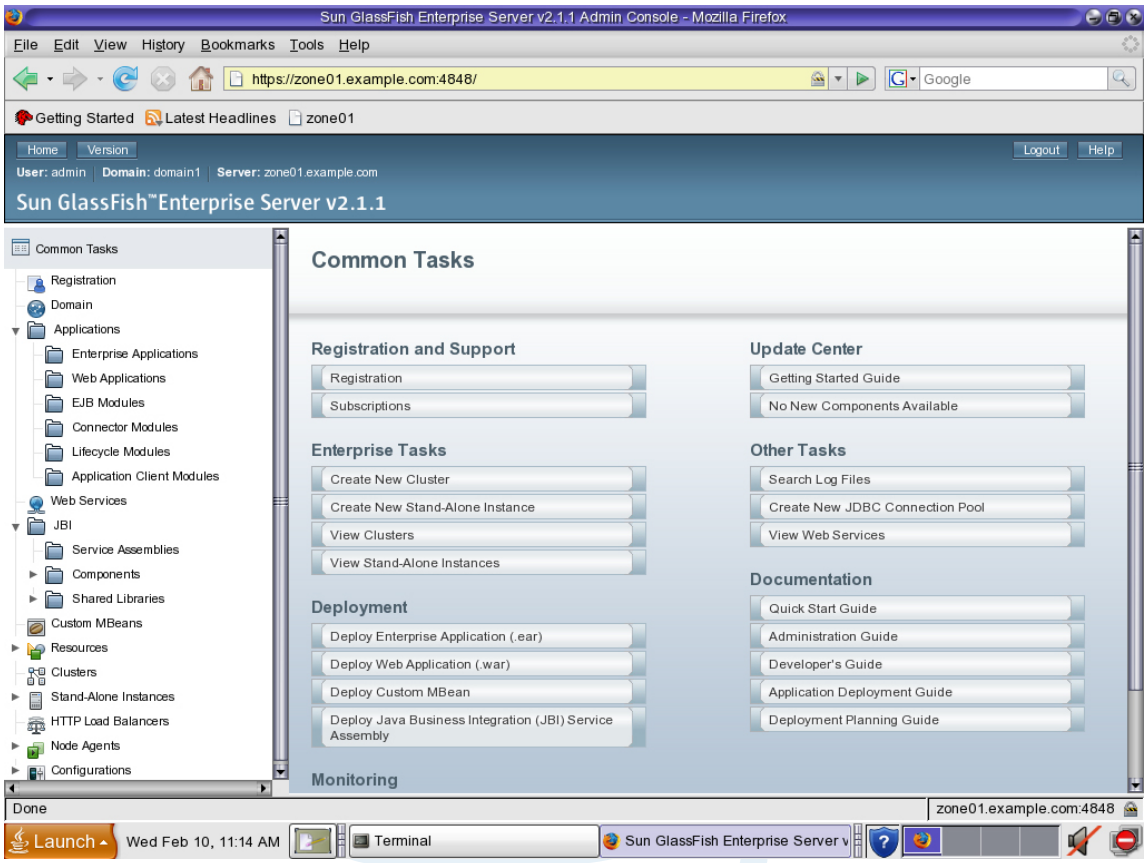
Narzędzie **asadmin** jest to narzędzie służące do administracji GlassFish'em. Narzędzie CLI w którym przeprowadzamy całkowitą konfigurację naszej domeny. Parametr **start-domain** uruchamia domenę, DAS'a, czyli mówiąc konkretnie instancję administracyjną **server.**

Zostaniemy poproszeni o podanie hasła Maser. W naszym przypadku to **adminadmin**.

```
Domain domain1 started.
Domain [domain1] is running [Sun GlassFish Enterprise Server v2.1.1 ((v2.1 Patch06)(9.1_02 Patch12)) (build b31g-fcs)] with its configuration and logs at: [/opt/SUNWappserver/domains].
Admin Console is available at [https://localhost:4848].
Use the same port [4848] for „asadmin" commands.
```



Rysunek 2 - Ekran logowania do Admin Console.



Rysunek 3 - Strona startowa Admin Console. Common Tasks. Po zalogowaniu do Admin Console.

```
User web applications are avail-
able at these URLs:
[http://localhost:8080    https://
localhost:8181 ].
Following web-contexts are avail-
able:
[/web1  /__wstx-services ].
Standard JMX Clients (like JCon-
sole) can connect to JMXServi-
ceURL:
[service:jmx:rmi:///jndi/rmi://
learning:8686/jmxrmi] for domain
management purposes.
Domain listens on at least follow-
ing ports for connections:
[8080  8181  4848  3700  3820  3920
8686 ].
Domain supports application server
clusters and other standalone in-
stances.
```

Po uruchomieniu instancji **server** (DAS), mamy powyższe podsumowanie. Zalogujmy się na Admin Console w celu praktycznego przetestowania DAS'a. W tym celu uruchamiamy przeglądarkę w zonie **globalnej** jako URL podając zonę **zone01** na której działa nasz DAS. Z uwagi na komunikację z użyciem FQDN użyjemy p

https://zone01.example.com:4848/

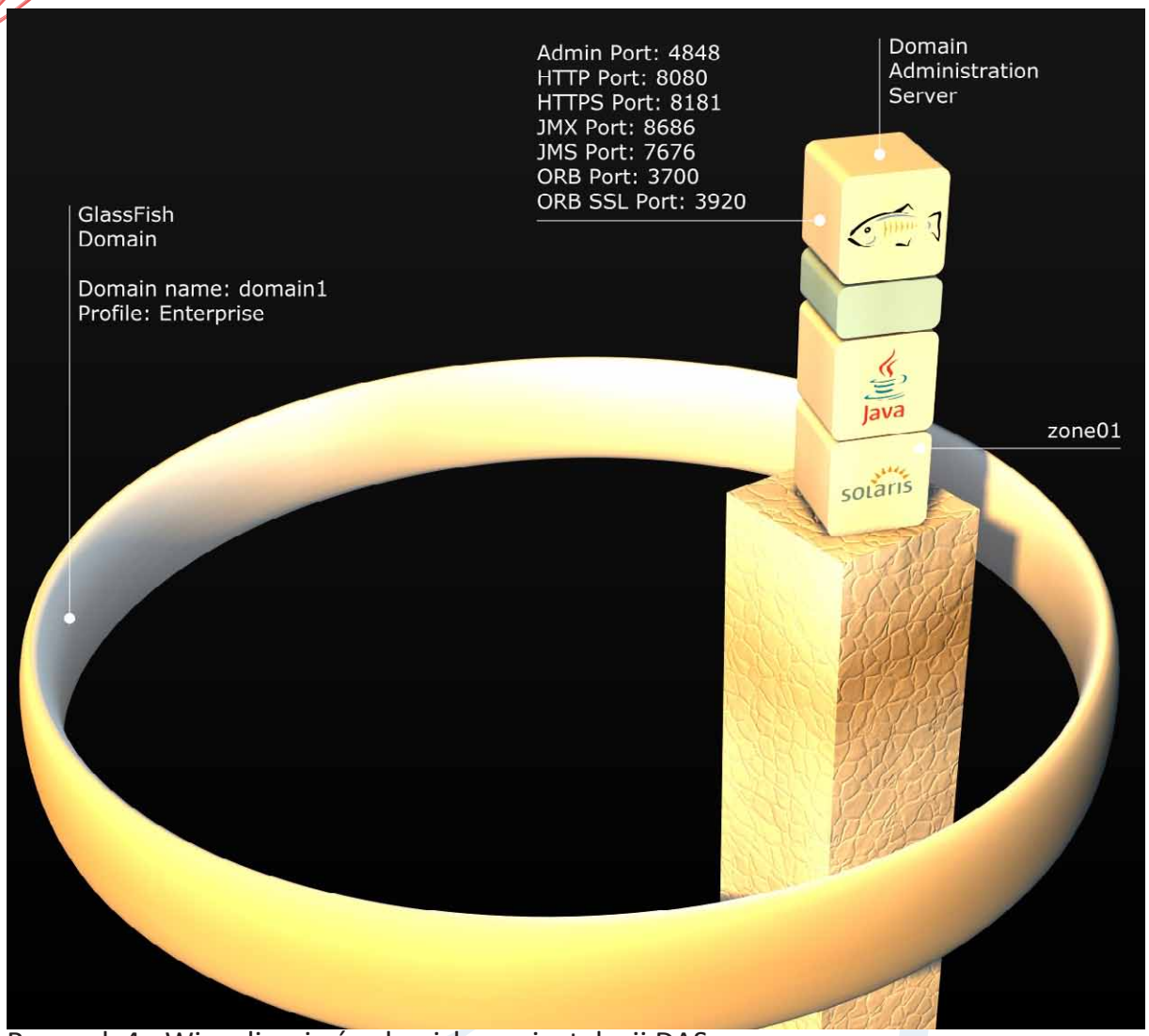
Powyższy URL jest adresem pod którym logujemy się do **Admin Console**. Admin Console jest to webowe, graficzne narzędzie administracyjne, które podobnie jak **asadmin** służy da administracji domeną.

Budowa akwarium. Klaster GlassFish.

Po zainstalowaniu DAS czas stworzyć przykładowy klaster. Wpierw stworzymy konfigurację naszego klastra na DAS'ie do której będziemy podczepiać nowe instancje. Nowo dodane instancje synchronizują się z centralnym repozytorium DAS'a kopiując do repozytorium lokalnego aplikację które będą hostować - instancje w klastrze są homogeniczne. Do stworzenia klastra wystarczy jedna prosta komenda:

```
zone01 # ./asadmin create-cluster
clusterA
```

Po chwili otrzymamy odpowiedź.



Rysunek 4 - Wizualizacja środowiska po instalacji DAS.

```
Command executed successfuly
```

Po wykonaniu tej komendy w katalogu **/opt/SUNWappserver/domain/domain/config** utworzył się katalog **clusterA-config** w wewnątrz którym w katalogu **lib** możemy dodać biblioteki tylko na potrzeby naszych instancji w klastrze. Utworzyliśmy klaster o nazwie **clusterA**. Oprócz utworzenia powyższych katalogów został zaktualizowany plik **domain.xml** opisujący naszą domenę **domain1** o wpis na temat klastra. Po utworzeniu konfiguracji klastra czas dodać do nie go instancje.

Budowa akwarium ciąg dalszy. Instalacja agentów.

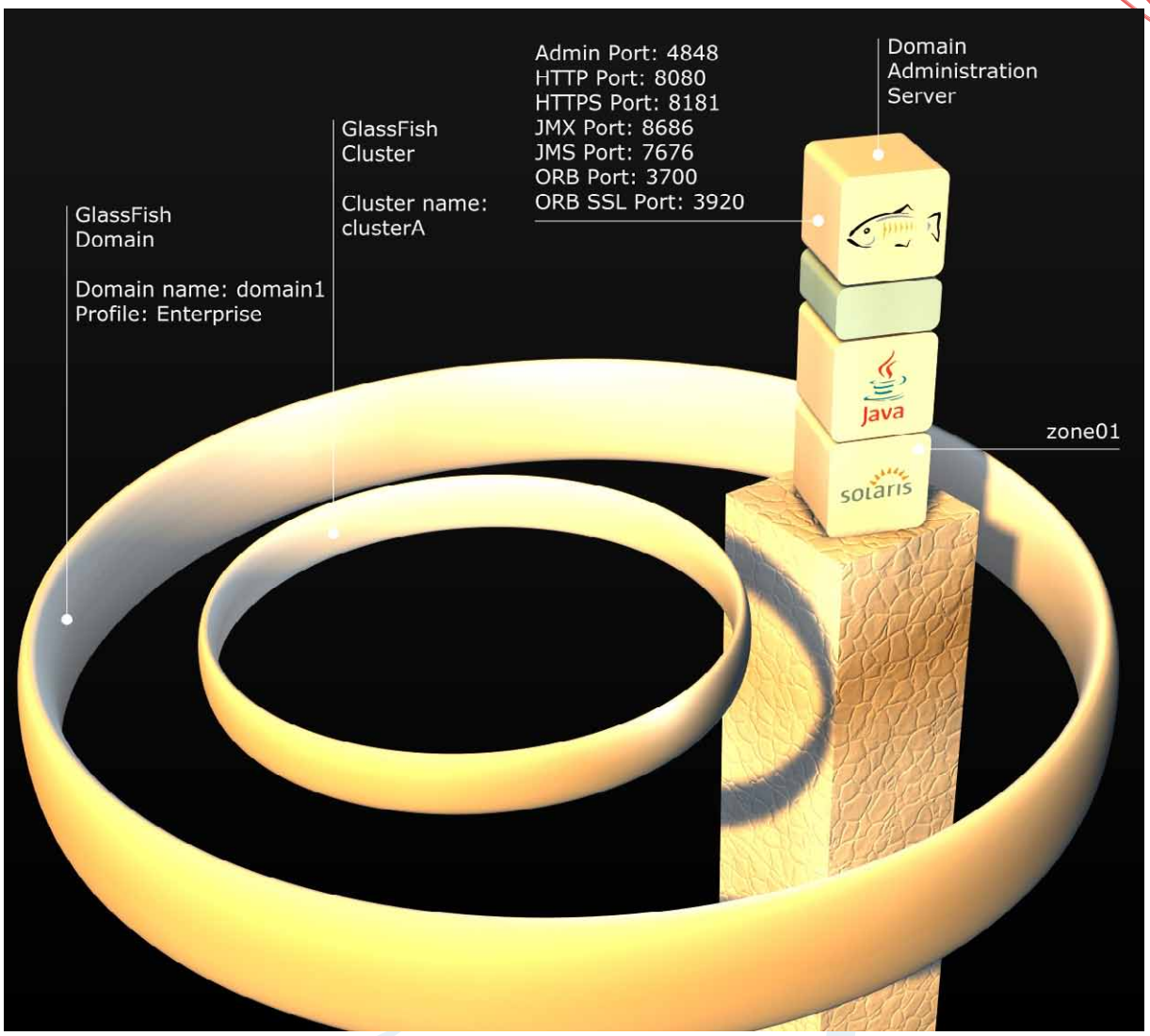
Naszą architekturę zbudujemy z czterech instancjach. Każda z nich będzie zainstalowana w oddzielnej zonie: **zone02, zone03, zone04, zone05**. Instalacja jest podobna jak w przypadku DAS. Z taką różnicą, że nie instalujemy komponentów:

- Domain Administration Server.
- Administration tools.

Z uwagi na fakt, że całe środowisko jest w zonach wpierw musimy uruchomić dane zony. W tym celu:

```
global # zoneadm -z zone02 boot
global # zoneadm -z zone03 boot
global # zoneadm -z zone04 boot
global # zoneadm -z zone05 boot
```

Powyższe komendy spowoduje uruchomieni 4 następnych zon. Na każdej z nich przeprowadzimy identyczne instalacje GlassFish'a. Na każdej stworzymy po jednym agencie do którego będziemy podczepiać instancje.



Rysunek 5 – Wizualizacja środowiska po stworzeniu klastra.

Kolejno dla każdej z zon:

1. Logujemy się:

```
global # zlogin zone02
[Connected to zone ‚zone02’ pts/14]
Last login: Thu Feb 11 07:38:06 on
pts/6
Sun Microsystems Inc.   SunOS 5.10
Generic January 2005
zone02 #
```

2. Instalujemy GlassFish’a (pomijając instalacje Domain Administration Server oraz Administration Tools, ale nie zapominając o komponentach HADB. Inne parametry instalacji robimy identyczne jak w przypadku instalacji naszego DAS’a):

```
zone02 # ./sges_ee-2_1_1-solaris-
i586-ml.bin -console
```

3. Tworzymy agenta komendą **create-node-agent** w asadmin w zonie, podając jako host DAS’a w formacie FQDN (**zone01.example.com**) a jako port **4848**, na którym działa DAS:

```
zone02 # cd /opt/SUNWappserver/
bin/
zone02 # ./asadmin create-node-
agent --host zone01.example.com
--port 4848 zone02-node-agent
```

Podczas tworzenia agenta zostaniemy poproszeni o podanie loginu (**admin**):

```
Please enter the admin user name>
```

Oraz hasła (**adminadmin**):

```
Please enter the admin password>
```

Po podaniu wymaganych danych po chwili otrzymamy odpowiedź o utworzeniu agenta.

```
Command create-node-agent executed successfully.
```

W tym momencie DAS otrzymał wpis w domenie o istnieniu agenta na oddzielnej maszynie, jednak abyśmy mieli z nią komunikację przez JMX musimy wpierw uruchomić naszego agenta.

4. Uruchamiamy agenta komendą **start-node-agent** w asadmin:

```
zone02 # ./asadmin start-node-agent zone02-node-agent
```

Zostaniemy poproszeni o podanie loginu (**admin**):

```
Please enter the admin user name>
```

Hasła (**adminadmin**):

```
Please enter the admin password>
```

Oraz hasła Master (**adminadmin**):

```
Please enter the master password [Enter to accept the default]:>
```

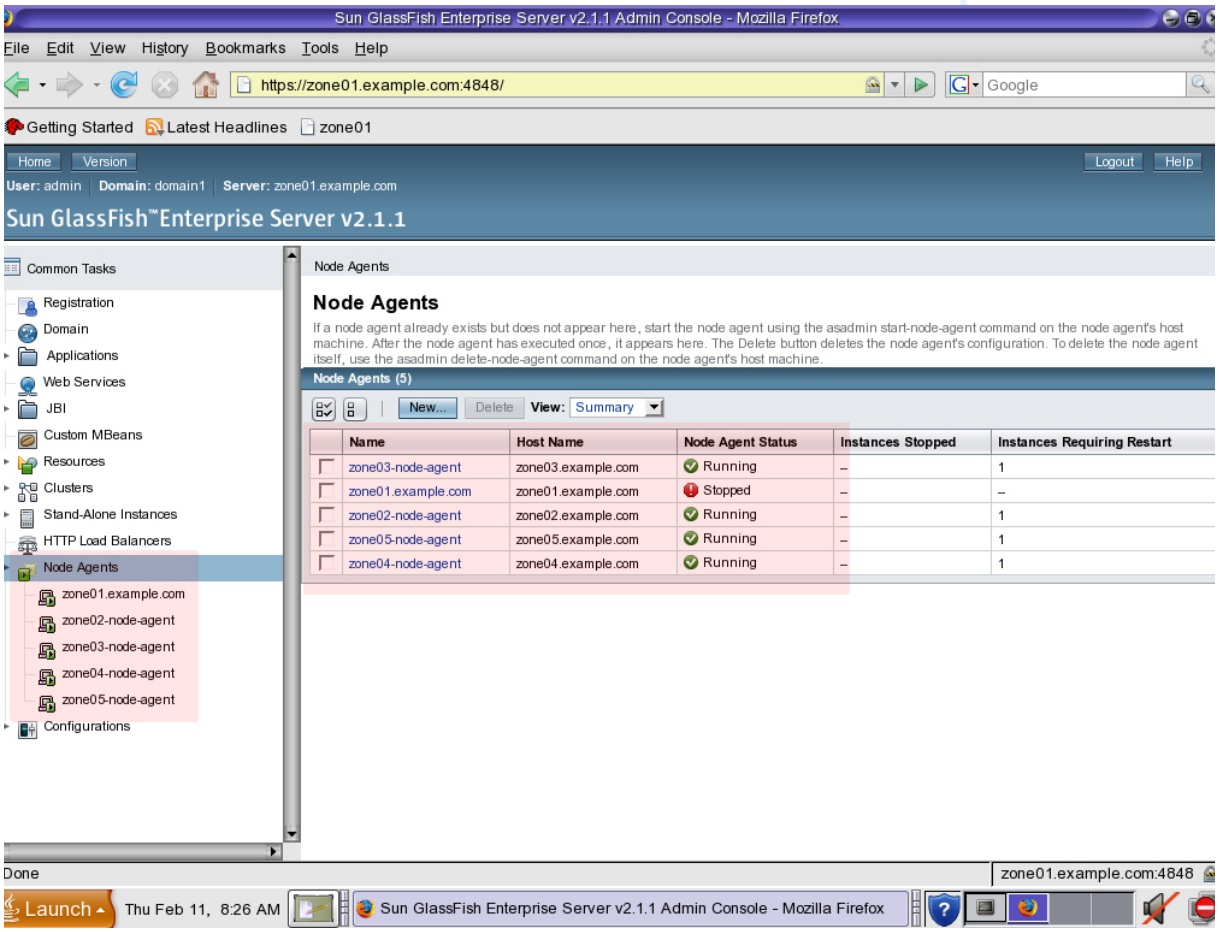
Po uruchomieniu agent otrzymujemy wpis z informacją do logów jak poniżej.

```
Redirecting output to /opt/SUNWappserver/nodeagents/zone01-node-agent/agent/logs/server.log
Redirecting application output to /opt/SUNWappserver/nodeagents/zone01-node-agent /agent/logs/server.log
Command start-node-agent executed successfully.
```

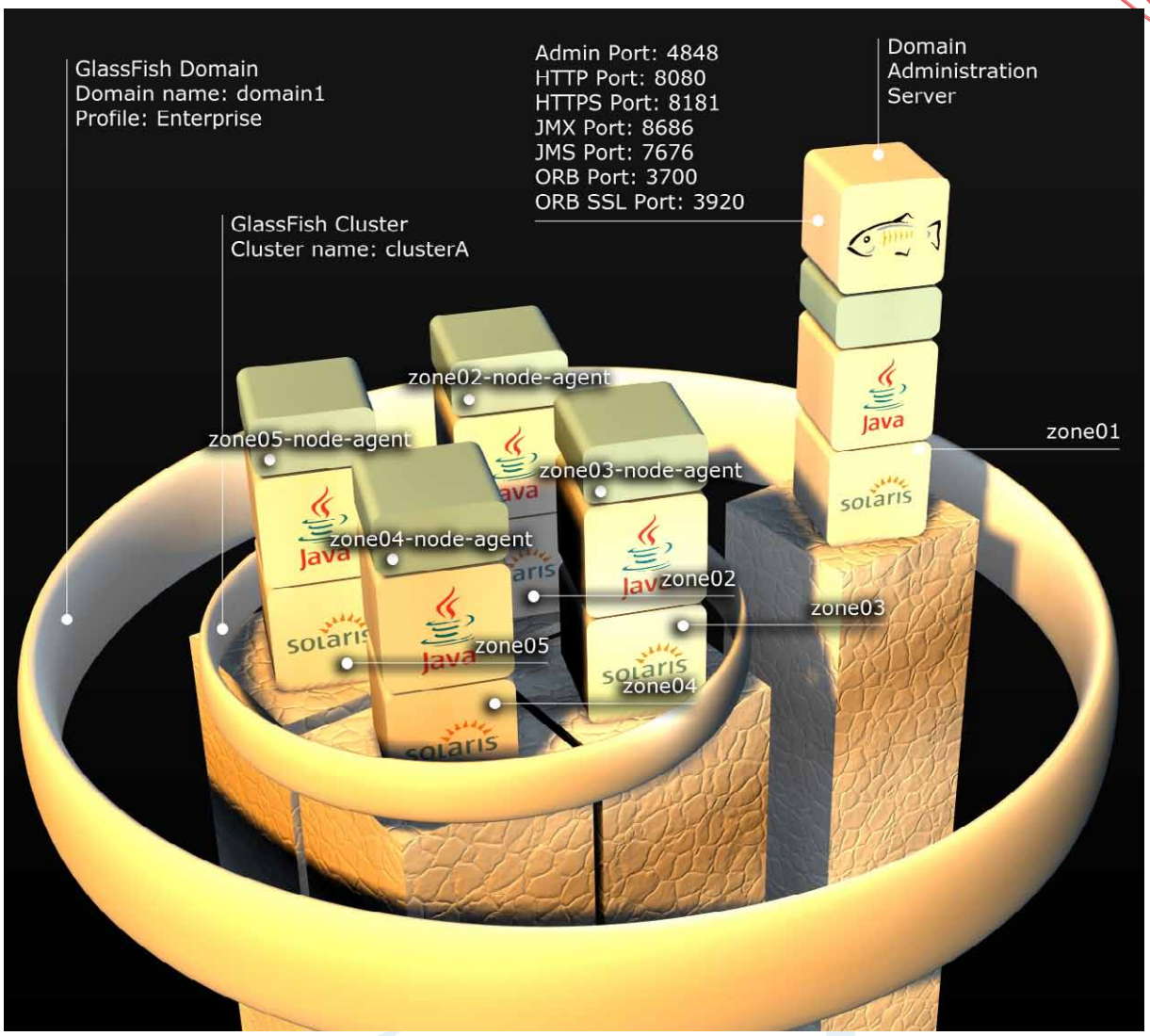
Po dokonaniu tych czynności na każdej z zon, sprawdzimy czy DAS ma komunikację za wszystkimi agentami. Logując się na **Admin Console** (https://zone01.example.com:4848) w drzewie **Node Agents** zobaczymy listę naszych agentów w domenie (Rysunek 6).

Budowa akwarium ciąg dalszy. Instancje w klastrze.

Obecnie utworzyliśmy nasz klaster **clusterA** dodając do niego cztery hosty. Czas



Rysunek 6 – Podgląd stanu agentów z poziomu Admin Console.



Rysunek 7 - Wizualizacja środowiska po utworzeniu agentów.

utworzyć na hostach instancje GlassFish'a, które będą brać udział w mechanizmie HADB. W tym celu na każdej z zon: **zone02**, **zone03**, **zone04**, **zone05** zainstalujemy instancje już bezpośrednio z DAS. Mamy taką możliwość ponieważ uruchomiliśmy już agentów.

Instancje utworzymy używając komendy **create-instance** w **asadmin**.

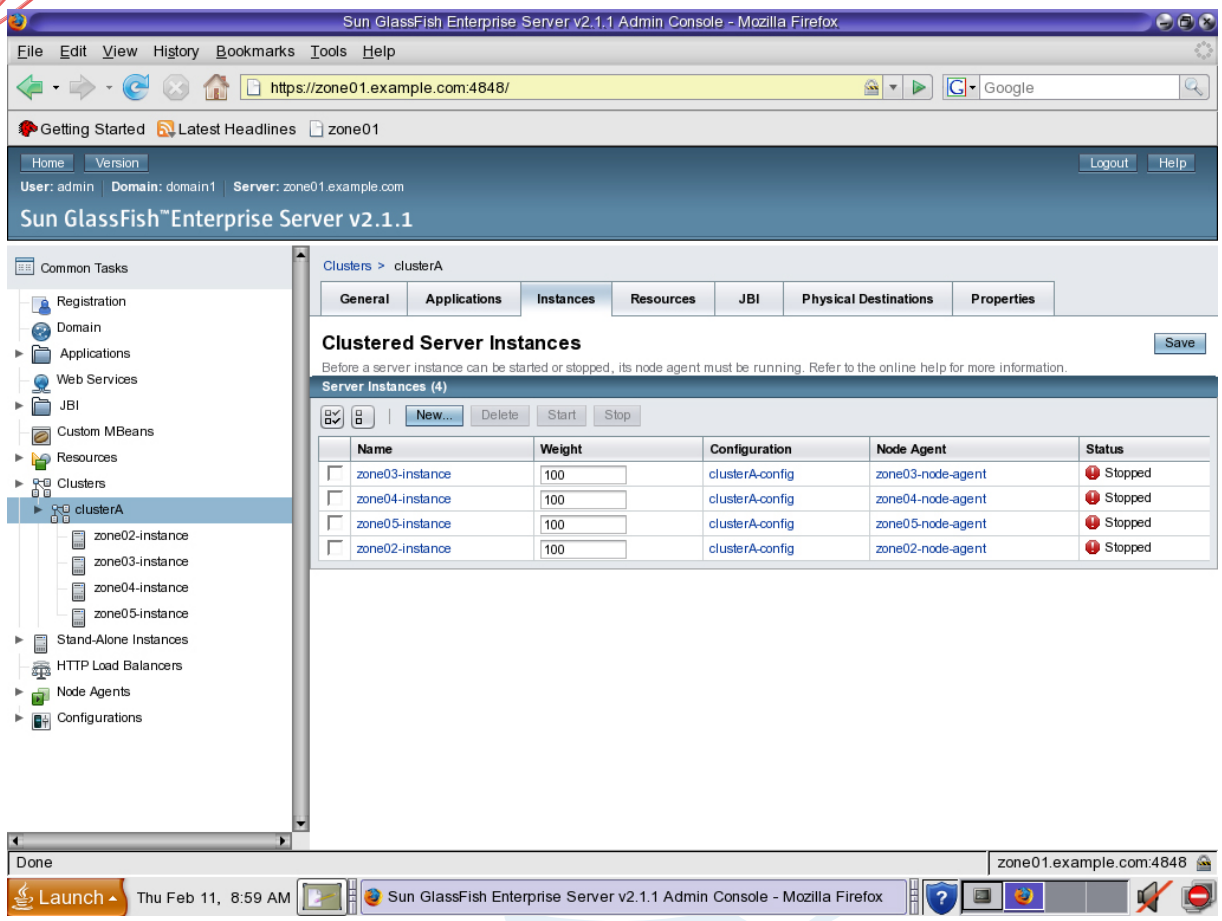
```
zone01 # ./asadmin create-instance --cluster clusterA --nodeagent zone02-node-agent instance
```

Po utworzeniu instancji otrzymamy podsumowanie na jakich portach działa utworzona instancja.

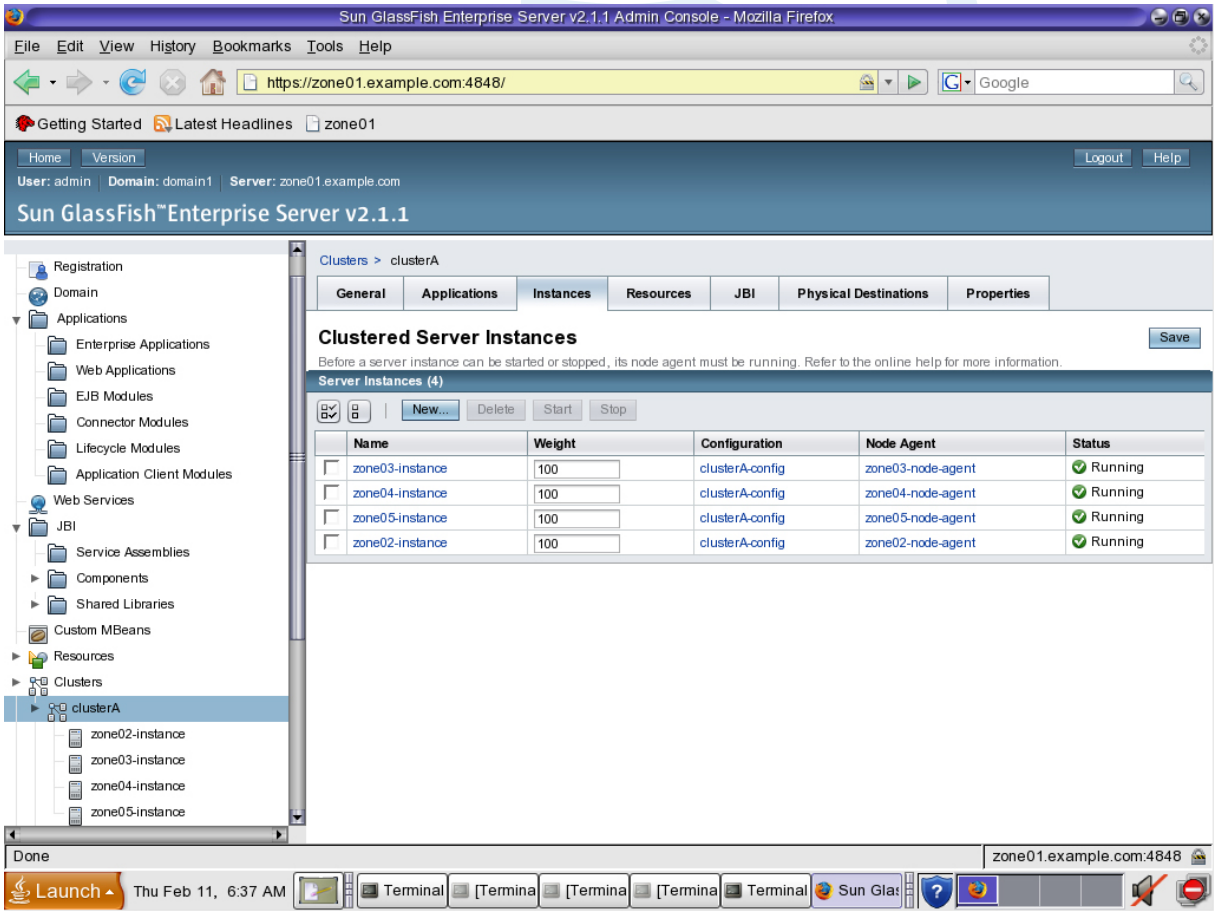
```
Using 38,081 for HTTP_LISTENER_PORT.
Using 38,182 for HTTP_SSL_LISTENER_PORT.
Using 33,821 for IIOP_SSL_LISTENER_PORT. Using 37,677 for JMS_PROVIDER_PORT.
Using 33,701 for IIOP_LISTENER_PORT.
Using 38,687 for JMX_SYSTEM_CONNECTOR_PORT.
Using 33,921 for IIOP_SSL_MUTUALAUTH_PORT.
Command create-instance executed successfully.
```

Analogicznie tworzymy instancje dla każdej z zon. Po utworzeniu wszystkich instancji i zalogowaniu się Admin Console (https://zone01.example.com:4848) zobaczymy jak wygląda nasz klaster. Drzewo **Clusters -> clusterA** - zakładka **Instances**.

Aby uruchomić klaster wystarczy nam prosta komenda **start-cluster** w asadmin na DAS:



Rysunek 8 - Stan klastra po utworzeniu instancji.



Rysunek 9 - Stan klastra po jego uruchomieniu.

```
zone01 # ./asadmin start-cluster clusterA

Command start-cluster executed successfully.
```

Po uruchomieniu tej komendy w drzewie **Clusters –> clusterA** zobaczymy nasze 4 aktywne instancje wraz z ich stanem (Rysunek 9).

Na końcu mamy prezentacje graficzną naszego środowiska. Zakończyliśmy etap budowy klastra czas przygotować HADB (Rysunek 10).

Topologie HADB

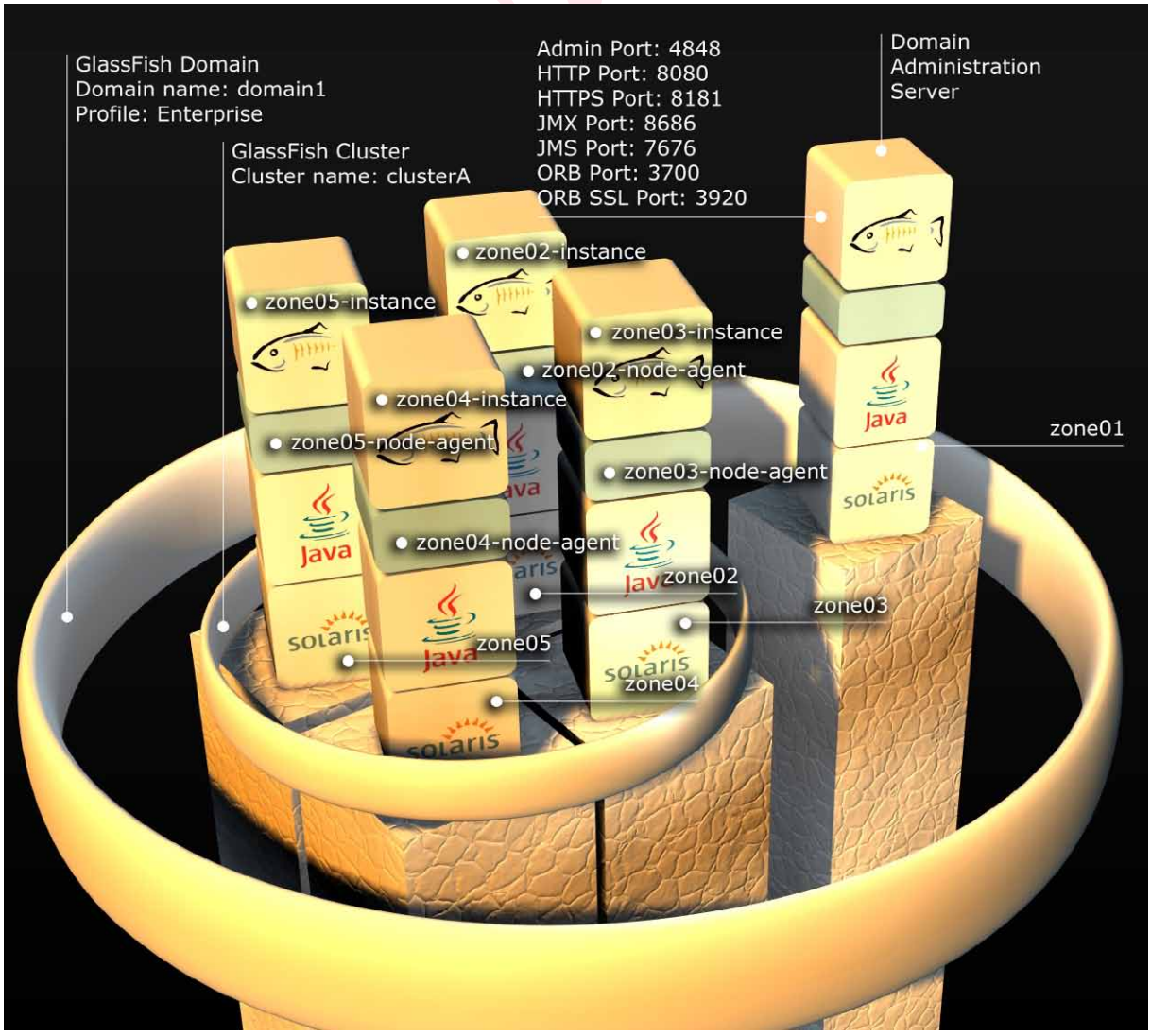
Mechanizm rozproszonej bazy wysokiej dostępności pozwala na stworzenie dwóch topologii:

- **Colocated** (wraz z instancjami klastra)
- **Separate tier** (na oddzielnych nodach)

Pierwsza bazuje na użyciu tych nod'ów na których pracują instancje w celu przechowywania replikacji sesji. Druga przewiduje użycie w tym celu odrębnych maszyn. Zwiększa to oczywiście wydajność instancji, ale nakłada wymogi na przepustowość sieci, oraz wymaga dodatkowego sprzętu.

Z uwagi na prostotę naszego środowiska będziemy trzymać się pierwszej topologii – **colocated**.

Obie topologie przedstawione są na Rysunkach 11 i 12.



Rysunek 10 - Stan środowiska po uruchomieniu klastra.

> Dwie przedstawione topologie charakteryzuje identyczna logika działania HADB.

Teoria w HADB.

Dwie przedstawione topologie charakteryzuje identyczna logika działania HADB. Warto wpierw omówić kilka pojęć związanych z HADB aby w pełnie zrozumieć obie topologie.

Omówimy:

- Domena HADB
- DRU
- Management Agent
- Active node
- Spare node

Domena HADB - W obu topologiach mechanizm HADB tworzy odrębną domenę. Nie jest to domena w rozumieniu tradycyjnych domen GlassFish'a, ale domena skupiająca w sobie nod'y odpowiadające za repozytorium i replikacje sesji. Domena HADB skupia w sobie pary nod'ów, każdy para replikuje sesje między sobą (A w B i B w A) w grupach zwanych **DRU**.

DRU (Data Redundancy Unit) – Są zawsze dwie. Każda z nich zawiera 100% replikację sesji. Nod wchodzący w skład jednego DRU ma w drugim DRU swego mirror'a, drugiego nod'a z którym replikuje sesje. Oba nod'y tworzą poziom. Do każdej domeny nod'y dodaje się lub usuwa parami, co skutkuje usunięciem bądź dodaniem nod'ów z obydwu DRU. Każdy DRU zawiera w sobie aktywne (**active**) oraz wolne (**spare**) nod'y.

Management Agent – Proces JVM, agenta HADB hostujący sesje dla klasterze GlassFish. Może być **active** lub **spare**.

Active node – Nod aktywny wchodzący w skład DRU. Jest to działający proces Management Agent (MA), który bierze aktywny udział w domenie (hostuje oraz replikuje sesje).

Spare node – Jest to nod „zapasowy". Spare node jest podpięty pod dane DRU i pełni rolę backup'u. Jeśli z jakiś powodów proces MA przestaje działać w jego miejsce wchodzi Spare Node, który synchronizuje się pobierając replikacje sesji z mirror'a zakończonego procesu MA. W architekturze HADB Spare Nod'y są opcjonalne. Jeśli ich nie wykorzystamy to w przypadku awarii danego MA jego mirror hostuje swoje sesje plus replikacje sesji mirrora, co sprawia, że musi mieć zapasową moc obliczeniową.

Cała baza działa dopóki dopóty na każdym z poziomów działa przynajmniej jeden nod.

Na Rysunku 13 przedstawiona jest przykładowa prezentacja architektury HADB z użyciem 4 nodów, 2 active, 2 spare.

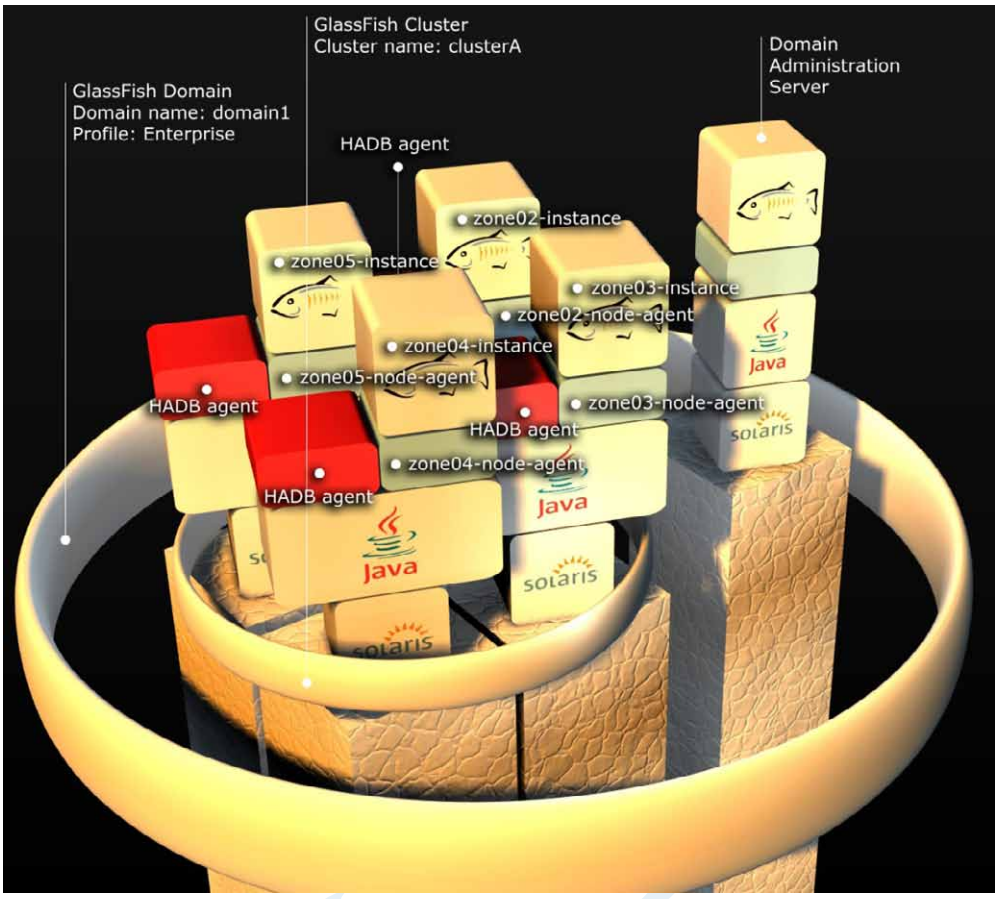
Dodatkowo na Rysunku 14 dołączony jest diagram struktur połączonych reprezentujący architekturę, którą będziemy budować.

Budowa HADB. Uruchamianie agentów HADB.

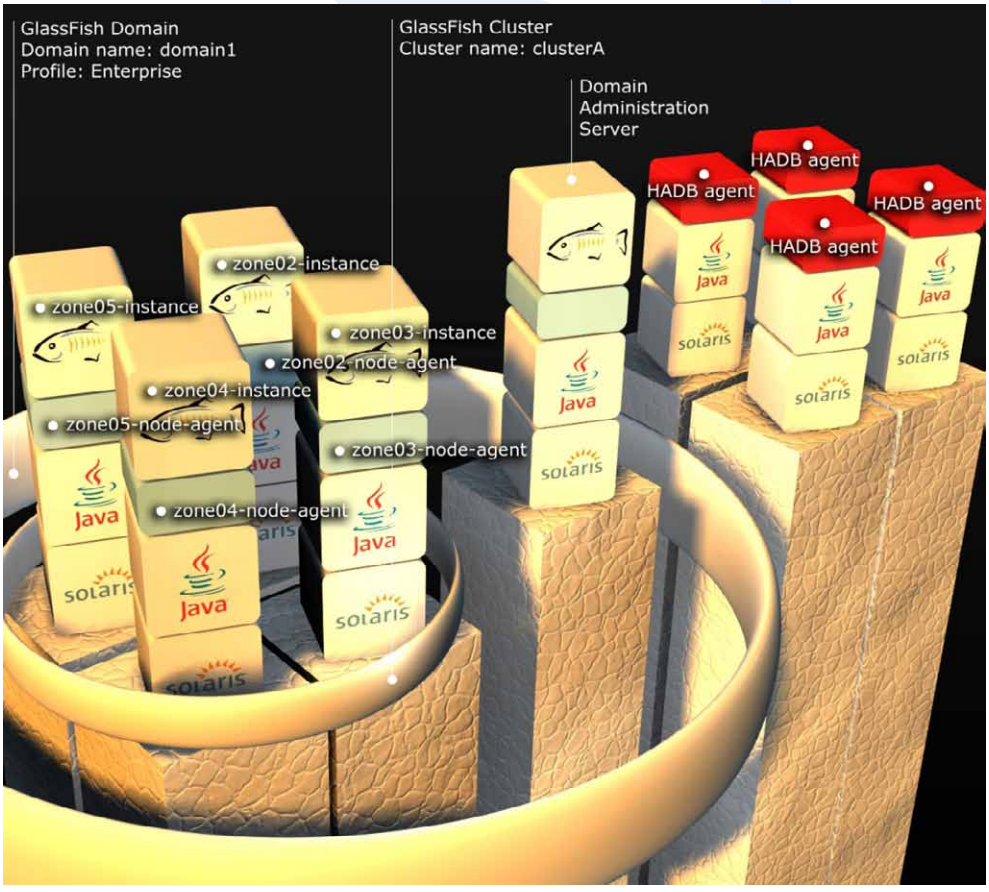
Aby móc skorzystać z mechanizmu HADB w środowisku Solaris wymogiem jest parametryzacja kernela. Są to cztery komendy które wykonujemy w terminalu w zonie globlanej:

```
global # set shmsys:shminfo_shmmax=0x80000000
global # set semsys:seminfo_semmni=16
global # set semsys:seminfo_semmns=128
global # set semsys:seminfo_semmnu=1000
```

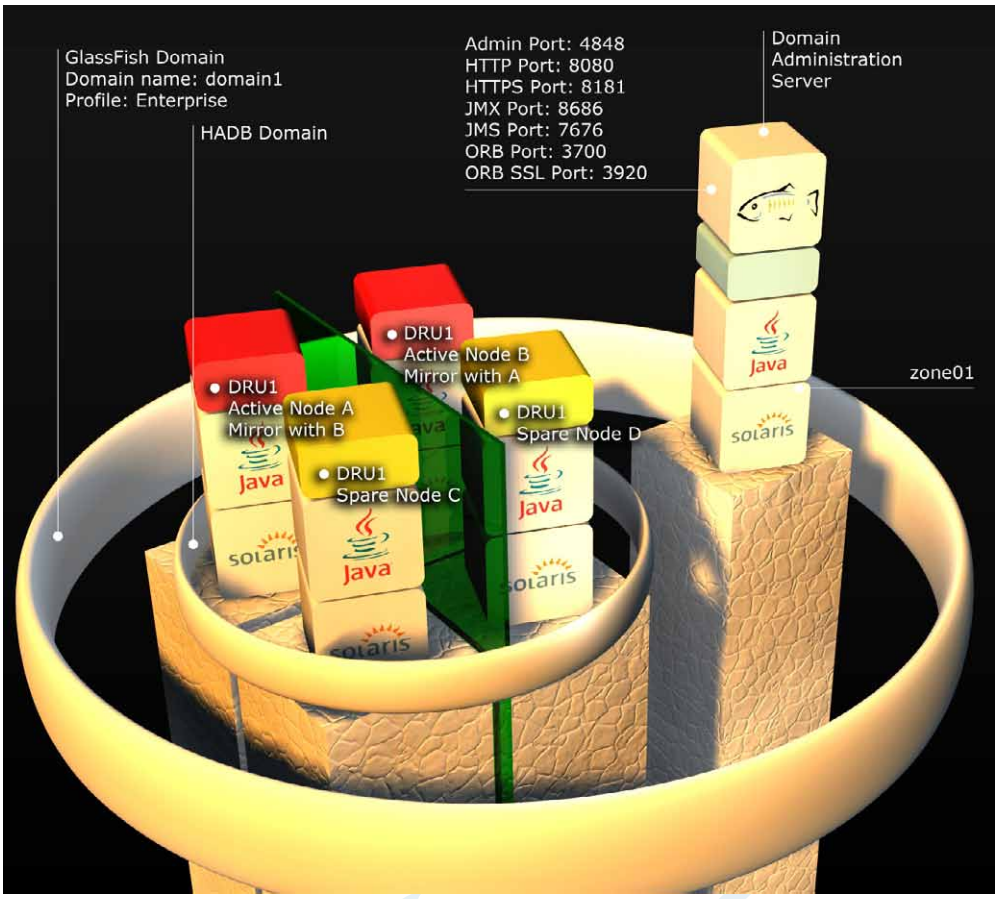
Uwaga: Powyższe parametry różnią się oczywiście od środowiska na jakim będzie-



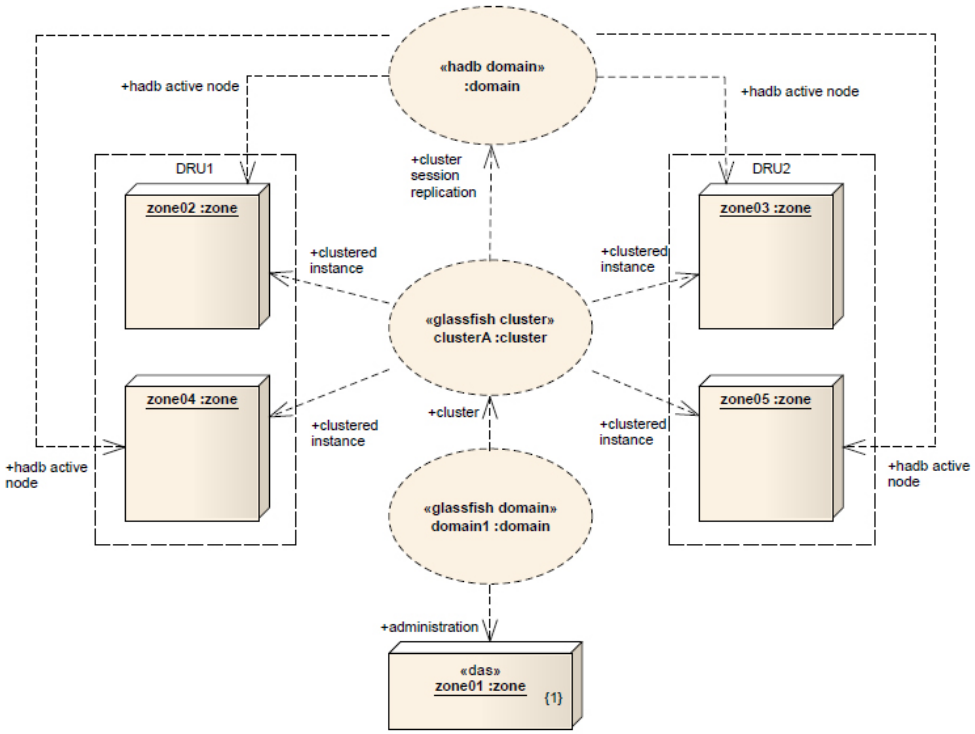
Rysunek 11 - Topologia coolocated w HADB.



Rysunek 12 - Topologia separate tier w HADB.



Rysunek 13 - Przykładowa architektura prostego HADB w topologii separate tier. W celu przejrzystości pominięto reprezentację klastra clusterA.



Rysunek 14 - Diagram struktur połączonych zależności między domeną a hadb.

> Pierwszą czynnością,
> którą przeprowadzamy podczas tworzenia HADB
> jest instalacja modułów HADB do każdej z instancji.

my stawiać nasze usługi. Jako referencje polecam Sun GlassFish Enterprise Server v2.1 High Avaliabiliy Administration Guide, gdzie znajdziemy wytyczne odnoście konfiguracji środowiska pod HADB.

Po ustawieniu parametrów robimy restart maszyny. Formalnie Solaris pozwala na konfigurowania parametrów bez restartowania maszyny, ale z uwagi na prostotę przeprowadzamy ponowne uruchomienie systemu. W tym celu użyjemy komendy **init 6**.

```
global # init 6
```

Po ponownym uruchomianiu maszyny, zalogowaniu do Solarisa, musimy na nowo:

1. Uruchomić zone zone01 a w niej DAS'a
2. Uruchomić zony zone02, zone03, zone04, zone05 i w nich ich agentów
3. Uruchomić klastra z DAS

Wszystkie powyższe procedury były już omawiane, więc nie powinno być problemu z ich powtórzeniem.

Pierwszą czynnością, którą przeprowadzamy podczas tworzenia HADB jest instalacja modułów HADB do każdej z instancji. Tą czynność już przeprowadziliśmy instalując moduły wraz z instancjami na samym początku.

Drugim krokiem jest uruchomienie **agentów HADB.** Agent HADB nosi nazwę **Management Agent** i odpowiada za obsługę tej części bazy za którą odpowiada dany nod z bazie HADB. Dla przypomnienia dodam, że baza jest rozproszona, co za tym idzie jej **schema** jest podzielona względem ilości nod'ów, a każdy z nod'ów pracuje jedynie na swojej części bazy.

W celu uruchomienia Management Agent'a w katalogu instalacji GlassFisha mamy podkatalog hadb. To właśnie w nim znajdują się moduły do HADB, które w tej dystrybucji są dostępne w dwóch wersjach 4 oraz 4.4.3-21 jak widać po nazwach katalogów w katalogu hadb. Do naszych ćwiczeń użyjemy modułów 4.4.3-21. Uruchamiamy więc agenta (na każdej z zon: **zone02**, **zone03**, **zone04**, **zone05**).

```
zone02 # cd /opt/SUNWappserver/
hadb/4.3.3-21/bin/
zone02 # ./ma ma.cfg
```

Plik **ma.cfg** jest to plik konfiguracyjny agenta.

```
Management Agent version 4.4.3.21
[V4-4-3-21 2009-04-25 03:09:43
pakker@hel04] ( SunOS_5.9_ix86)
starting
Logging to /opt/SUNWappserver/
hadb/4.4.3-21/log/ma.log
2010-02-14 18:04:26.964 INFO Ma-
nagement Agent version 4.4.3.21
[V4-4-3-21 200 9-04-25 03:09:43
pakker@hel04] (SunOS_5.9_ix86)
starting
2010-02-14 18:04:26.965 INFO
Using property: jgroups.fragsi-
ze=8096
2010-02-14 18:04:26.966 INFO
Using property: ma.server.type=j-
mxmp
2010-02-14 18:04:26.966 INFO
Using property: ma.server.main-
ternal.interfaces=
2010-02-14 18:04:26.966 INFO
Using property: ma.server.dbhi-
storypath=/opt/SUN Wappserver/
hadb/4.4.3-21/history
2010-02-14 18:04:26.966 INFO
Using property: ma.server.dbcon-
figpath=/opt/SUNW appserver/config/
hadb
2010-02-14 18:04:26.967 INFO
Using property: jgroups.logleve-
l=OFF
2010-02-14 18:04:26.967 INFO
Using property: console.logleve-
```

> I w ten oto prosty sposób utworzyliśmy krok po kroku architekturę HADB.

```
l=INFO
2010-02-14    18:04:26.967    INFO
Using property: logfile.name=/opt/
SUNWappserver/ hadb/4.4.3-21/log/
ma.log
2010-02-14    18:04:26.967    INFO
Using property: ma.server.jmxmp.
port=1862
2010-02-14    18:04:26.967    INFO
Using property: logfile.logleve-
l=INFO
2010-02-14    18:04:26.968    INFO
Using property: ma.server.dbde-
vicepath=/opt/SUNW      appserver/
hadb/4.4.3-21/device
2010-02-14 18:04:26.968 INFO Using
property:     repository.dr.path=/
opt/SUNWapps erver/hadb/4.4.3-21/
rep
2010-02-14 18:04:27.203 INFO   Li-
stening for client connections on
port 1862
```

Powyższy wydruk na konsole jest podsumowaniem po uruchomieniu agenta. Jak widać zgodnie z konfiguracją w pliku **ma-.cfg** agent nasłuchuje na porcie JMX 1862, repozytorium przetrzymuje w podkatalogu **/rep,** logi agenta są zapisywane do pliku **ma.log** a historia operacji na bazie jest przechowywana w podkatalogu **/history**.

Oprócz skryptu **ma**, który uruchamia nam agenta w katalog **/bin** znajdziemy:

```
zone01 # ls
clusql    hadbm     ma          ma-
initd  ma.cfg
```

- **clustersql** – SQL Utility, narzędzie SQL do ręcznej pracy na bazie.
- **hadbm** – (High Avaliability Database Manager) narzędzie do zarządzania bazą HADB.
- **ma** – Management Agent, skrypt do uruchomienia agenta.
- **ma-initd** – skrypt do zarejestrowania agenta jako usługi w Solarisie do wersji 9 (od wersji 10 rejestrujemy do sami jako usługę w SMF).
- **ma.cfg** – plik konfiguracyjna agenta.

Po uruchomieniu agentów HADB, czas na budowę bazy.

Budowa HADB. Tworzenie bazy.

Do stworzenia bazy wystarczy w przypadku topologii colocated wystarczy nam jedna prosta komenda (w przełączniku hosts nie ma spacji między kolejnymi maszynami oraz nazwy maszyn podajemy używając FQDN):

```
zone01 # ./asadmin configure-ha-
cluster --hosts zone02.example.
com,zone03.example.com,zone04.
example.com,zone05.example.com
clusterA
```

Po dłuższej chwili (w zależności od sprzętu na jakim pracujemy 5 minut lub więcej) otrzymamy odpowiedź:

```
Command configure-ha-cluster exe-
cuted successfully.
```

I w ten oto prosty sposób utworzyliśmy krok po kroku architekturę HADB.

To co dzieje się przez tą komendę to uruchomienie w tle narzędzia **hadbm** i kolejno utworzenie domeny HADB, stworzenie schema dla bazy i dodanie nod'ów do DRU, wszystkich jako aktywne. Dzięki tej komendzie mamy najszybszy i najprostszy sposób to tworzenia HADB jednak tylko w topologii colocated. W przypadku topologii z użyciem oddzielnych maszyn musielibyśmy używać narzędzia **hadbm** i ręcznie krop po kroku tworzyli całą strukturę.

Uwaga: Dzięki narzędziu **hadbm** możemy zarządzać całą domeną HADB. Dodawać nod'y, zmieniać rozmiar używanej prze-

> Konfigurację HADB
> przeprowadzamy z poziomu Admin Console.

strzenie na dysku przez nod'y, czyścić bazę, refragmentować w przypadku dodania czy usunięcia nod'ów z DRU etc.

Aby korzystać z tego mechanizmu wystarczy wdrożyć aplikację, domyślni DAS korzysta z HADB. Dzięki HADB będziemy mieli zapewnioną dostępność i replikację komponentów SFSB. K

Konfigurację HADB przeprowadzamy z poziomu Admin Console.

Podsumowanie

Poniższy artykuł pokazał jak w prosty sposób zbudować HADB. Więc na temat wysokiej dostępność w oparciu o GlassFisha polecam dokument - „Sun GlassFish Enterprise Server 2.1 High Availability Administration Guide" gdzie szczegółowo opisany jest Load Balancing, HADB oraz replikacja JMS.

http://docs.sun.com/app/docs/doc/820-4341

Mirosław Dąbrowski
Certyfikowany instruktor
Sun Microsystems
miroslaw.dabrowski@euler.pl

Log4j a komunikatory internetowe

Michał Szynkaruk

Logi informujące o błędach w trakcie działania aplikacji można zapisywać zarówno lokalnie (jako pliki tekstowe, krotki w bazie danych itd.) jak i wysyłać na inną maszynę, np. w postaci e-maila. Niewątpliwe wielu z nas ma na stałe uruchomiony jakiś komunikator internetowy niezależnie od tego czy jesteśmy w pracy, w domu czy w podróży. Skoro chcemy być w stałym kontakcie z naszymi znajomymi, to dlaczego nie możemy być z naszą aplikacją?

W tym artykule pokażę jak można wykorzystać bibliotekę Log4j i najpopularniejsze komunikatory w Polsce: Skype, GaduGadu oraz Google Talk.

Skype i Log4j

Dzięki API o nazwie Skype4Java które jest oficjalnie udostępnione poprzez stronę:

https://developer.skype.com/wiki/Java_API

możemy w łatwy sposób wysyłać logi na podane konto. Aby to uczynić należy:

- posiadać aktywne konto Skype
- dołączyć bibliotekę skype_full.jar
- dołączyć bibliotekę log4j

Na początku konstruujemy własny appender w oparciu o abstrakcyjną klasę AppenderSkeleton (jeśli nie masz podstawowej wiedzy dotyczącej biblioteki log4j, to polecam przeczytanie mojego artykułu który znajduje się w poprzednim numerze Java Express oraz oficjalnej dokumentacji na stronie http://logging.apache.org/log4j/1.2/index.html). Nazwijmy naszą klasę SkypeAppender i umieśćmy w niej tylko jedną zmienną wraz ze standardowym getterem i setterem:

```
private String receiver;

public String getReceiver() {
  return receiver;
}

public void setReceiver(
    String receiver) {
  this.receiver = receiver;
}
```

Następnie zdefiniujmy funkcję odpowiedzialną za wysyłanie wiadomości:

```
public void sendMessage(
    String content) {
  try {
    Skype.chat(receiver).
      send(content);
  } catch (SkypeException ex) {
  }
}
```

oraz nadpiszmy metody dziedziczone z klasy AppenderSkeleton:

```
@Override
protected void append(
    LoggingEvent event) {
  sendMessage(getLayout().
    format(event));
}

@Override
public boolean requiresLayout() {
  return true;
}

@Override
public void close() {
}
```

Konfigurację możemy ustawić przykładowo w pliku właściwości:

```
log4j.appender.skype=log4jtests.
SkypeAppender
log4j.appender.skype.layout=org.
apache.log4j.PatternLayout
log4j.appender.skype.layout.Co-
nversionPattern=[%p] %c - %m
log4j.appender.skype.receiver=
TU_PODAJESZ_NAZWE_ODBIORCY
```

> Gadu-Gadu wykorzystuje własny protokół komunikacji,

```
log4j.rootLogger=DEBUG, skype
```

Pozostało nam już tylko stworzyć obiekt typu Logger i wysłać wpis.

Przykładowe wywołanie w statycznej funkcji **main()**:

```
Logger logger =
  Logger.getRootLogger();
logger.info(
  „Czesc, tu aplikacja :)”);
```

Ważne jest by w trakcie działania aplikacji uruchomiony był Skype i umożliwiał on „ingerencję” naszej aplikacji w Javie.

Wysyłamy loGGi

Gadu-Gadu wykorzystuje własny protokół komunikacji, w którym m. in. każdy użytkownik jest jednoznacznie identyfikowany za pomocą unikalnego numeru. Aby móc wysyłać logi w postaci wiadomości musimy:

- posiadać aktywne konto GG z którego dane logi wysyłamy
- dołączyć bibliotekę JGGApi która implementuje wymieniony protoków (szczegółowe informacje można znaleźć na stronie: http://jggapi.sourceforge.net/)
- dołączyć bibliotekę Jakarta-Commons Logging która jest wykorzystywana przez JGGApi
- dołączyć bibliotekę log4j

W tym przykładzie skorzystałem z JGGApi w wersji 1.6, log4j w wersji 1.2.15, Commons Logging w wersji 1.1.1.

Podobnie jak w poprzednim przykładzie konstruujemy własny appender w oparciu o abstrakcyjną klasę AppenderSkeleton. Niech nasza klasa przyjmie nazwę GGAppender. Umieścimy w niej następujące zmienne:

```
private int number;
private String password;
private int receiver;
private boolean isReady;
private boolean isFirst = true;
private ISession session;
private LoginContext loginContext;
```

Gdzie number i password odnoszą się do naszego konta gg, receiver jest numerem GG na który chcemy wysłać naszą wiadomość.

Zmienna isReady mówi nam kiedy mamy połączenie z serwerem Gadu-Gadu, jesteśmy zalogowani i gotowi wysyłać wiadomości. Zmienna isFirst określa czy potrzebujemy połączenia z serwerem Gadu-Gadu, session jak sama nazwa wskazuje jest sesją, a loginContext zawiera dane o naszym koncie GG. Dla wszystkich wyżej wymienionych zmiennych definiujemy standardowe gettery i settery (tak by móc przypisać im wartości określone w konfiguracji, o czym za chwile będzie mowa). Następnie definiujemy funkcję **connect()** dzięki której połączymy się z serwerem Gadu-Gadu:

```
public void connect()
    throws GGException {
  loginContext =
    new LoginContext(number,
      password);
  session = SessionFactory.
    createSession();
  session.
    addSessionStateListener(
      new SessionStateListener() {
    public void sessionStateChan-
ged(
  SessionState oldSessionState,
  SessionState newSessionState) {
  if (newSessionState.equals(
    SessionState.
      AUTHENTICATION_AWAITING)) {
        login();
```

> Musimy także zaimplementować funkcję append

```
      } else if (newSessionState.
        equals(
        SessionState.LOGGED_IN))
      {
        isReady = true;
      } else {
        isReady = false;
      }
    }
  });
  IConnectionService
    connectionService =
      session.
        getConnectionService();
  connectionService.
    addConnectionListener(
      new ConnectionListener.
        Stub() {
    @Override
    public void connectionEstabli-
shed() {
      System.out.println(
        "Dokonano połączenia");
    }

    @Override
    public void connectionError(
        Exception ex) {
      System.out.println(
        "Błąd połączenia: " +
        ex.getMessage());
    }

    @Override
    public void connectionClosed()
    {
      System.out.println(
        "Połączenie zakończone");
    }
  });
  IServer server = session.
    getConnectionService().
      lookupServer(
        loginContext.getUin());
  connectionService.connect(
    server);
}
```

Teraz czas na funkcję login() :

```
private void login() {
  ILoginService loginService =
    session.getLoginService();
  loginService.
    addLoginListener(
      new LoginListener.Stub() {
    @Override
    public void loginOK() {
      System.out.println(
        „Zalogowano");
    }

    public void loginFailed() {
      System.out.println(
        „Nieudane logowanie");
    }
  });
  try {
    loginService.login(
      loginContext);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

oraz funkcję sendMessage która w argumencie przyjmuje treść wiadomości:

```
public void sendMessage(
    String content) {
  IMessageService messageService =
    session.getMessageService();
  OutgoingMessage outMessage =
    OutgoingMessage.
      createNewMessage(receiver,
        content);
  try {
    messageService.sendMessage(
      outMessage);
  } catch (GGException e) {
    e.printStackTrace();
  }
}
```

Musimy także zaimplementować funkcję **append** w której definiujemy gdzie log jest zapisywany, funkcje **close** oraz **requiresLayout**:

```
@Override
protected void append(
    LoggingEvent event) {
  if (isFirst) {
    try {
      connect();
```

> Podobny efekt uzyskamy jeśli stworzymy plik właściwości

```
    } catch (GGException e) {
      e.printStackTrace();
    }
    while (!isReady) {

    }
    isFirst = false;
  }
  sendMessage(getLayout().
    format(event));
}

@Override
public boolean requiresLayout() {
  return true;
}

@Override
public void close() {
}
```

Dzięki tej linii:

```
sendMessage(getLayout().
  format(event));
```

wysyłamy sformatowaną wiadomość według szablonu określonego w pliku konfiguracji.

Zdecydowałem się na konfigurację XML'ową w pliku log4j.xml:

```
<?xml version="1.0"
      encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration
  SYSTEM "log4j.dtd">
<log4j:configuration
      xmlns:log4j="http://jakarta.
apache.org/log4j/">
  <appender
    class="log4jtests.GGAppender"
    name="gg">
    <param name="number"
           value="TWÓJ_NUMER_GG"
    />
    <param name="password"
           value="TWOJE_HASŁO_GG"
    />
    <param name="receiver"
        value="NUMER_GG_ODBIORCY"
    />
    <layout
      class="org.apache.log4j.Pat-
ternLayout">
      <param
        name="ConversionPattern"
        value="[%p] %c - %m" />
    </layout>
  </appender>
  <root>
    <priority value="debug">
    </priority>
    <appender-ref ref="gg" />
  </root>
</log4j:configuration>
```

gdzie:

%p to priorytet zdarzenia

%c to kategoria zdarzenia

%m to treść komunikatu

a poziom od którego logi są wysyłane ustawiamy na debug.

Podobny efekt uzyskamy jeśli stworzymy plik właściwości (log4j.properties) :

```
log4j.appender.gg=
  log4jtests.GGAppender
log4j.appender.gg.layout=
  org.apache.log4j.PatternLayout
log4j.appender.gg.layout.Conver-
sionPattern=
  [%p] %c - %m
log4j.appender.gg.number=
  TWÓJ_NUMER_GG
log4j.appender.gg.password=
  TWOJE_HASŁO_GG
log4j.appender.gg.receiver=
  NUMER_GG_ODBIORCY
log4j.rootLogger=
  DEBUG, gg
```

Przykładowy test który możemy umieścić w statycznej funkcji main():

```
Logger logger =
  Logger.getRootLogger();
try {
  int i = 5 / 0;
} catch (ArithmeticException ex) {
  logger.error(
```

> Podobnie jak w poprzednich dwóch przykładach tworzymy własny appender

```
    „Wystapil blad w programie”);
}
```

Google Talk

Wysyłanie wiadomości na konto Google Talk jest równie łatwe, gdyż dysponujemy darmowymi klientami protokołu xmpp z którego korzysta ten komunikator. Osobiście skorzystałem z biblioteki Smack w wersji 3.1.0 którą można pobrać ze strony:

http://www.igniterealtime.org/downloads/index.jsp#smack

Podobnie jak w poprzednich dwóch przykładach tworzymy własny appender, który u mnie prezentuje się następująco:

```
public class GtalkAppender
    extends AppenderSkeleton {
  private String user;
  private String password;
  private String receiver;
  // tu umieść gettery i settery
  // dla powyższych trzech
  // zmiennych

  private boolean isFirst = true;
  XMPPConnection connection;
  ConnectionConfiguration
    connConfig;

  public void sendMessage(
      String content) {
    try {
      Skype.chat(receiver).
        send(content);
    } catch (SkypeException ex) {
    }
  }

  @Override
  protected void append(
      LoggingEvent event) {
    if (isFirst) {
      connConfig =
       new ConnectionConfiguration(
          “talk.google.com”, 5222,
           “gmail.com”);
      connection =
        new XMPPConnection(
          connConfig);
      try {
        connection.connect();
        connection.login(
          user, password);
      } catch (XMPPException ex) {
      }
      while (!connection.
        isConnected()) {
      }
      isFirst = !isFirst;
    }
    Message msg = new Message(
      receiver,
      Message.Type.chat);
    msg.setBody(getLayout().
      format(event));
    connection.sendPacket(msg);
  }

  @Override
  public boolean requiresLayout()
  {
    return true;
  }

  @Override
  public void close() {
  }
}
```

Przykładowa konfiguracja poprzez plik właściwości:

```
log4j.appender.gtalk=
  log4jtests.GtalkAppender
log4j.appender.gtalk.layout=
  org.apache.log4j.PatternLayout
log4j.appender.gtalk.layout.Co-
nversionPattern=
  [%p] %c - %m
log4j.appender.gtalk.user=
  TWÓJ_ADRES_GMAIL
log4j.appender.gtalk.password=
  TWOJE_HASŁO
log4j.appender.gtalk.receiver=
  ADRES_GMAIL_ODBIORCY
log4j.rootLogger=
  DEBUG, gtalk
```

> Sposób monitorowania działania naszej aplikacji przy wykorzystaniu tych trzech komunikatorów wydaje się być niezwykle ciekawy i skuteczny.

Podsumowanie

Możliwości Log4j oraz istnienie javowych bibliotek dla Skype, Gadu-Gadu oraz Gtalk powodują, że każdy z nas ma możliwość szybkiego odbioru informacji o wystąpieniu ewentualnych błędów w trakcie działania naszej aplikacji. Dzięki temu szybko i skutecznie możemy rozpocząć akcje naprawcze. Sposób monitorowania działania naszej aplikacji przy wykorzystaniu tych trzech komunikatorów wydaje się być niezwykle ciekawy i skuteczny.

O autorze

Michał jest studentem IV roku informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej oraz certyfikowanym programistą Java.

Aplikacje Flex z BlazeDS

Sujit Reddy G (tłumaczenie Paweł Cegła)

Niniejszy artykuł przedstawia, jak stworzyć aplikację Flex w środowisku programistycznym Flash Builder 4 beta 2, która metodę z klasy Java na serwerze używając zdalnej usługi BlazeDS. Po skonfigurowaniu serwera wymaganego dla przykładowej aplikacji, użyjemy środowiska Flash Builder 4, żeby wygenerować klasy usługowe w języku ActionScript i zbudować aplikację Flex, która wyświetli wyniki wywołania zdalnej metody.

Wymagania

Aby wykonać kroki przedstawione w artykule, potrzebne będzie następujące oprogramowanie i pliki:

Flash Builder 4 beta:

- Pobierz
- Dowiedz się więcej

BlazeDS 4.0 nightly build:

- Pobierz

Apache Tomcat:

- Dowiedz się więcej

Przykładowe pliki:

- flashbuilder4_blazeds_source.zip (ZIP, 35 KB)

Wymagana wiedza

Wiedza o kontenerach J2EE, Adobe Flex i platformie Java będzie przydatna.

Konfiguracja serwera

Pierwszym krokiem jest utworzenie klasy Java, którą będziemy wywoływać z aplikacji Flex. Przykładowa aplikacja w niniejszym artykule używa klasy SimpleCustomerService.class. Klasa ta zawiera metodę o nazwie getAllCustomers(), która zostanie wywołana z aplikacji Flex:

```
public class
    SimpleCustomerService {
  public ArrayList<SimpleCustomer>
    getAllCustomers() {
    ArrayList<SimpleCustomer>
      customers = null;
    // code to create ArrayList
    // containing SimpleCustomer
    // objects
    return customers;
  }
}
```

Rozpoczniemy od skompilowania plików SimpleCustomerService.java i SimpleCustomer.java w folderze <SampleZipFile>/java_src (można użyć już skompilowanych klas z folderu <SampleZipFile>/java_classes).

Aby utworzyć aplikację internetową, która korzysta z powyższych klas, należy wykonać poniższe kroki:

1. Jeśli Tomcat nie jest jeszcze zainstalowany, należy zrobić to teraz. Informacje o tym, skąd go pobrać znajdują się pod adresem http://tomcat.apache.org/.
2. Wyszukaj folder webapps w folderze Tomcata. W systemie Windows domyślnym miejscem jest C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\.
3. Utwórz folder o nazwie samplewebapp w folderze webapps, aby utworzyć nową aplikację internetową.
4. Skopiuj klasy SimpleCustomerService.class i SimpleCustomer.class do folderu webapps/samplewe-

> Osiągnięcie rzeczy prostych
> jest w Gradle nie tylko łatwe,
> ale i daje się elegancko wyrazić.

bapp/WEB-INF/classes, do podfolderu odpowiadającemu hierarchii pakietów Java.

Konfiguracja BlazeDS

Zanim nastąpi możliwość wywołania klasy Java z aplikacji Flex należy ją wystawić jako zdalną usługę BlazeDS. Żeby skonfigurować BlazeDS dla przykładowej aplikacji internetowej należy skopiować pliki JAR BlazeDS tak, żeby znalazły się na ścieżce źródłowej aplikacji. Postępuj wg poniższych kroków, aby skonfigurować BlazeDS:

1. Ściągnij najnowszą dystrybucję binarną (nightly build) BlazeDS, która zawiera wszystkie pliki JAR i pliki konfiguracyjne, które są potrzebne. Jeśli potrzebujesz wcześniejszą wersję BlazeDS albo LiveCycle Data Services ES, należy postępować wg kroków znajdujących się na moim blogu w tym temacie.

2. Rozpakuj archiwum zip, które ściągnąłeś do folderu. W nim znajduje się plik blazeds.war, który zawiera wymagane pliki JAR i pliki konfiguracyjne potrzebne do uruchomienia BlazeDS oraz blazeds-bin-readme.htm, który zawiera warunki, postanowienia i szczegóły licencyjne.

3. Rozpakuj zawartość blazeds.war do oddzielnego folderu o nazwie blazeds (możesz użyć narzędzia WinZip lub podobnego do rozpakowania archiwum WAR.)

4. Skopiuj wszystkie pliki JAR z folderu blazeds/WEB-INF/lib do folderu samplewebapp/WEB-INF/lib.

5. Skopiuj folder blazeds/WEB-INF/flex do folderu samplewebapp/WEB-INF. Ten folder zawiera pliki konfiguracyjne BlazeDS, które służą do skonfigurowania usług zdalnych (Remoting), komunikacyjnych i proxy.

6. Jeśli jeszcze nie masz pliku web.xml, który chciałbyś użyć, skopiuj plik blazeds/WEB-INF/web.xml do folderu samplewebapp/WEB-INF.

```
<!-- Http Flex Session attribute and binding listener support -->
<listener>
<listener-class>flex.messaging.HttpFlexSession</listener-class>
</listener>
<!-- MessageBroker Servlet -->
<servlet>
<servlet-name>MessageBrokerServlet</servlet-name>
<display-name>MessageBrokerServlet</display-name>
<servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
<init-param>
<param-name>services.configuration.file</param-name>
<param-value>/WEB-INF/flex/services-config.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>MessageBrokerServlet</servlet-name>
<url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

> Osiągnięcie rzeczy prostych
> jest w Gradle nie tylko łatwe,
> ale i daje się elegancko wyrazić.

Następnie należy zmapować serwlet BlazeDS o nazwie MessageBrokerServlet tak, żeby BlazeDS zostało uruchomione przy każdym wywołaniu usług zdalnych, komunikacyjnych albo proxy poprzez każdy obsługiwany kanał.

Jeśli używasz własnego pliku web.xml, musisz dodać do niego poniższy kod. Możesz również skopiować go z pliku blazeds/WEB-INF/web.xml.

Flash Builder 4 korzysta z serwletu RDSDispatchServlet (znajdujący się w BlazeDS i LiveCycle Data Services ES2), żeby otrzymać szczegóły usługi w aplikacji internetowej. Jeśli korzystasz z własnego pliku web.xml, dodaj mapowanie dla RDSDispatchServlet do swojej aplikacji poprzez wklejenie poniższego fragmentu kodu do pliku samplewebapp/WEB-INF/web.xml w węźle <web-app>.

Utworzenie miejsca docelowego

Żeby wystawić klasę Java jako zdalne miejsce docelowe, należy dodać węzeł <destination> w pliku samplewebapp/WEB-INF/flex/remoting-config.xml pod węzłem <service>, jak pokazano niżej (pliki remoting-config.xml i services-config.xml użyte w przykładowej aplikacji można znaleźć w folderze <SampleZipFile>/config.)

```
<destination id="SimpleCustomer-
ServiceDestination">
  <properties>
    <source>
      com.adobe.services.
      SimpleCustomerService
    </source>
  </properties>
</destination>
```

Kiedy uruchomisz serwer Tomcat, BlazeDS wystawi klasę Java jako zdalne miejsce docelowe o identyfikatorze SimpleCustomerServiceDestination.

Stworzenie aplikacji klienckiej

Skoro serwer jest skonfigurowany, jesteśmy gotowi do użycia Flash Builder 4 i wygenerowania kodu, który skorzysta ze zdalnego miejsca docelowego. Aplikacja Flex wywoła metodę getAllCustomers() w miejscu docelowym, które zostało utworzone.

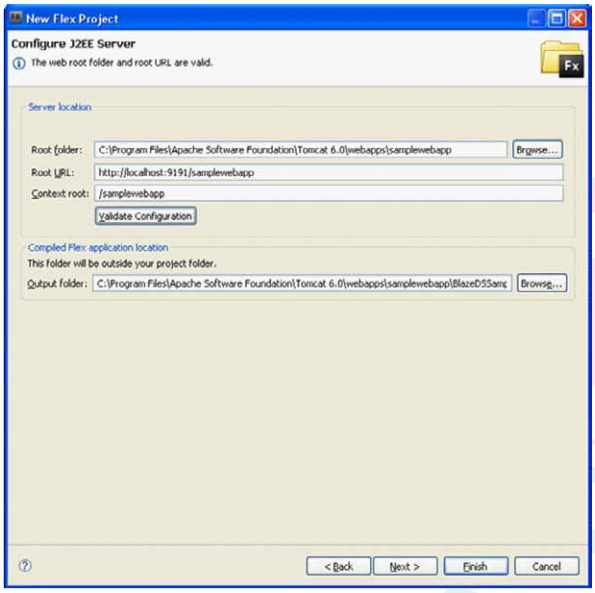
Żeby wygenerować kod ActionScript, które skorzysta z miejsca docelowego, Flash Builder zażąda od aplikacji internetowej BlazeDC albo Adobe LiveCycle Data Services ES2, którą skonfigurowano wcześniej i odczyta szczegóły wystawionej usługi.

```
<servlet>
  <servlet-name>RDSDispatchServlet</servlet-name>
  <display-name>RDSDispatchServlet</display-name>
      <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-
class>
  <init-param>
    <param-name>useAppserverSecurity</param-name>
    <param-value>false</param-value>
  </init-param>
  <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping id="RDS_DISPATCH_MAPPING">
  <servlet-name>RDSDispatchServlet</servlet-name>
  <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
</servlet-mapping>
```

> Postępuj zgodnie z poniższymi krokami, aby utworzyć nową usługę

Aby utworzyć nowy projekt Flex należy:

1. Wybrać File > New > Flex Project. (Rysunek 1)



Rys. 1. Ustawienie właściwości projektu

2. Jako nazwa projektu wpisz BlazeDS-Sample.
3. Wybierz Web (Runs In Adobe Flash Player) jako typ aplikacji.
4. Wybierz J2EE jako typ serwera aplikacyjnego.
5. Wybierz Use Remote Object Service a następnie wybierz BlazeDS (patrz rys. 1).
6. Kliknij Next, aby kontynuować.

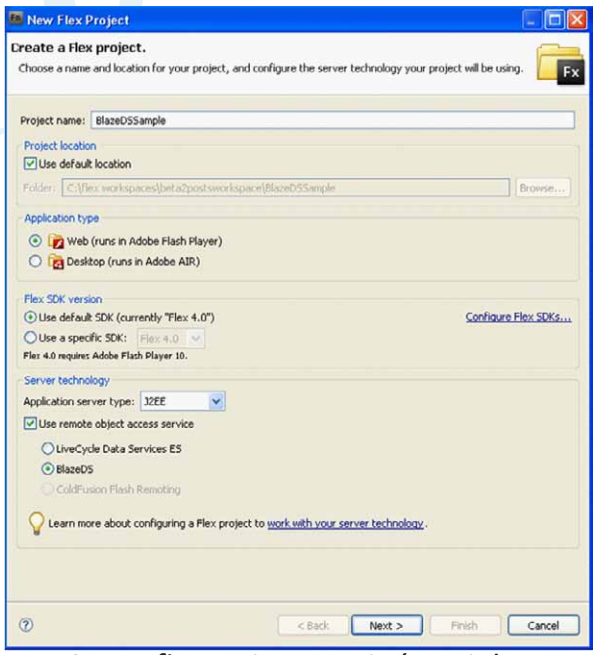
Ponieważ w ustawieniach projektu wybrano J2EE jako typ serwera, Flash Builder zapyta się o ustawienia serwera.

7. Aby skonfigurować serwer J2EE, ustaw jako Root Folder ścieżkę do głównego folderu aplikacji internetowej, która została skonfigurowana z BlazeDS.
8. Ustaw jako Root URL adres aplikacji internetowej; np. http://localhost:9191/samplewebapp.
9. Ustaw Context Root aplikacji internetowej; np. /samplewebapp.
10. Zostaw wartość domyślną dla Output Folder, wskazuje ona na serwer.
11. Kliknij Validate Configuration, żeby sprawdzić, czy konfiguracja serwera jest poprawna. Jeśli nie – sprawdź ustawienia i spróbuj jeszcze raz.
12. Jeśli konfiguracja jest poprawna, kliknij Finish, żeby utworzyć projekt.

Utworzenie nowej usługi

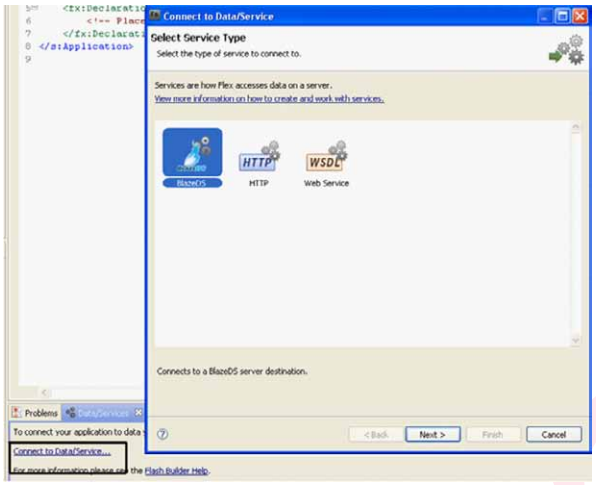
Postępuj zgodnie z poniższymi krokami, aby utworzyć nową usługę we Flash Builder 4:

1. Znajdź widok Data/Services we Flash Builder 4; jeśli nie jest otwarty, wybierz Window > Date/Services.



Rys. 2. Konfiguracja ustawień projektu

> Flash Builder 4 generuje klasę ActionScript, która reprezentuje klasę Java skojarzoną z wybranym zdalnym miejscem docelowym.



Rys. 3. Wybierz typ serwisu`

2. Kliknij Connect to Data/Service (albo wybierz Date > Connect to Data/Service). Flash Builder 4 wyświetli listę typów serwisów, które mogą zostać odpytane z bieżącego projektu Flex.

3. W tym przykładzie korzystamy z usługi BlazeDS, więc wybierz BlazeDS, jak pokazano na rys. 3.

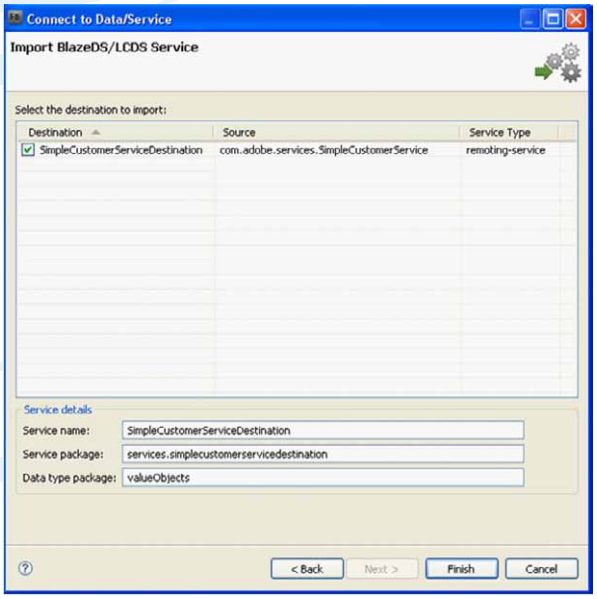
4. Kliknij Next.

5. Jeśli Flash Builder 4 poprosi o podanie hasła RDS, wybierz po prostu No Password Required i kliknij OK (to zadziała, gdyż parametr useAppSecurity serwletu RDSDispatcherServlet jest ustawiony na false w pliku web.xml.)

Flash Builder połączy się z serwerem i wyświetli listę wystawionych miejsc docelowych usługi, jak pokazano na rys. 4.

6. Select SimpleCustomerServiceDestination from the list and click Finish. If your application has more than one destination, you can select any destination from the list for which you want code to be generated.

Flash Builder 4 generuje klasę ActionScript, która reprezentuje klasę Java skojarzoną z wybranym zdalnym miejscem docelowym. W tym przypadku Flash Builder 4 wygenerował klasę o nazwie SimpleCustomerServiceDestination, która posiada funkcje odpowiadające publicznym metodom wystawionym w SimpleCustomerService.java. Żeby wywołać metodę SimpleCustomerService.java na serwerze, wystarczy wywołać odpowiednią funkcję SimpleCustomerServiceDestination. as. Usługę (SimpleCustomerServiceDestination.as) i jej wylistowane operacje (reprezentujące funkcje w klasie usługi) można zobaczyć na widoku Data/Services; można również zobaczyć kod w widoku Package Explorer. Jeżeli któryś z argumen-



Rys. 4. Lista dostępnych miejsc docelowych

tów lub wartość zwracana przez metody klas Java jest typem stworzonym przez użytkownika, Flash Builder 4 wygeneruje klasy ActionScript reprezentujące te typy razem klasami usług. W tym przykładzie getAllCustomers() zwraca listę ArrayList zawierającą obiekty typu SimpleCustomer. ArrayList jest typem wbudowanym i

> Flash Builder 4 potrafi również generować kod wywołujący usługę i wiążący wyniki z kontrolkami.

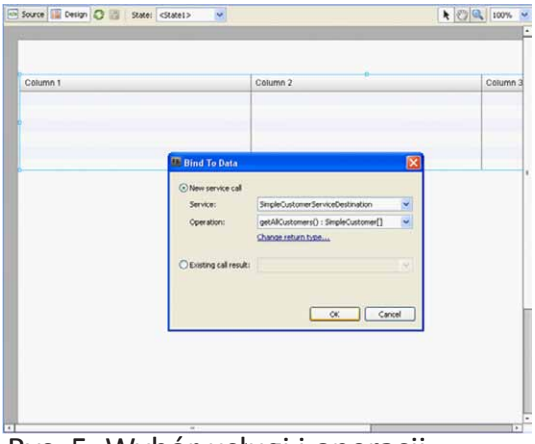
domyślnie przekonwertowano go na typ ArrayCollection po stronie klienta. SimpleCustomer jest jednak typem użytkownika, więc Flash Builder 4 wygenerował SimpleCustomer.as z właściwościami dla każdej publicznej właściwości w SimpleCustomer.java, żeby reprezentować obiekty zwrócone przez serwer. Zwróć uwagę, że typy zwracane przez usługę są także poprawnie skonfigurowane w widoku Data/Services na podstawie typów metod w klasie Java.

Kod potrzebny do użycia wartości zwróconej przez zdalną usługę został wygenerowany, następna część przedstawia kroki wiodące do wyświetleni odpowiedzi w kontrolce interfejsu użytkownika.

Powiązanie wyników wywołania usługi z kontrolkami interfejsu użytkownika.

Flash Builder 4 potrafi również generować kod wywołujący usługę i wiążący wyniki z kontrolkami. W niniejszym przykładzie powiązane zostaną wyniki z kontrolką DataGrid.

1. Dodaj kontrolkę DataGrid do aplikacji w widoku Design.
2. Kliknij prawym klawiszem na kontrolce i wybierz Bind To Data.



Rys. 5. Wybór usługi i operacji

3. W oknie dialogowym Bind To Data, które się pojawi, wybierz New Service Call.
4. Wybierz SimpleCustomerServiceDestination z listy usług i wybierz getAllCustomers():SimpleCustomer[] z listy operacji jak pokazano na rys. 5.
5. Kliknij OK.

Ponieważ wybrano New Service Call, Flash Builder utworzy nową instancję klasy SimpleCustomerServiceDestination i klasy CallResponder w bieżącym pliku MXML. Jeśli instancja klasy SimpleCustomerServiceDestination już istnieje, tylko instancja klasy CallResponder zostanie utworzona. Klasa CallResponder pomaga w zarządzaniu wynikami wywołań asynchronicznych usług RPC. Więcej na temat klasy CallResponder można znaleźć we Flex Language Reference.

Uwaga: Jeśli wywołanie usługi istnieje już w bieżącym pliku MXML i chcesz powiązać wyniki z kontrolką, wybierz Existing Call Result w oknie dialogowym Bind To Data oraz istniejące wywołanie usługi.

Zapisz aplikację i ją uruchom. Po tym jak aplikacja Flex wystartuje w oknie przeglądarki internetowej, wywoła metodę getAllCustomers() w klasie SimpleCustomerService na serwerze i wyświetli obiekty SimpleCustomer zwrócone przez serwer w kontrolce DataGrid.

Co dalej

Teraz, gdy użyłeś Flash Builder 4, żeby stworzyć aplikację Flex, która wysyła żądania do zdalnej usługi BlazeDS i wyświetla wyniki w kontrolce DataGrid, może chciałbyś poszukać możliwości użycia tego po-

> Możesz wystawić dowolną klasę Java
> jako usługę BlazeDS albo LiveCycle Data Services ES2

dejścia we własnych aplikacjach. Możesz wystawić dowolną klasę Java jako usługę BlazeDS albo LiveCycle Data Services ES2 oraz zbudować aplikację Flex we Flash Builder 4, żeby skorzystać z nich.

Możesz również użyć Flash Builder 4 do stworzenia aplikacji dla usług HTTP, web services i usług zarządzających danymi LiveCycle Data Services ES2. Dowiesz się więcej z artykułu Tima Buntela pt. Data-centric development with Flash Builder 4 beta.

Jeśli tworzysz aplikacje zorientowane na dane, powinieneś poświęcić trochę czasu na zrozumienie możliwości zarządzania danymi i programowaniu zorientowanym modelowo LiveCycle Data Services ES2. The LiveCycle Data Services ES2 Quick Starts jest doskonałym zasobem na rozpoczęcie pracy.

O autorze

Sujit Reddy G pracuje na stanowisku Technical Evangelist for Flex w Adobe. Posiada ogromne doświadczenie w technologiach Flex, J2EE i PHP. Specjalizuje się w tworzeniu aplikacji korporacyjnych na platformie Adobe Flash i prowadzi blog skupiając się na integracji platformy Flash z Adobe LiveCycle Data Services (oraz BlazeDS) pod adresem http://sujitreddyg.wordpress.com.

Transakcje w systemach Java Enterprise: Wprowadzenie

Jarosław Błąd

Artykułem tym rozpoczynam cykl związany z szeroko pojętą tematyką budowy systemów transakcyjnych w środowisku Java Enterprise. Temu podstawowemu zagadnieniu związanemu bezpośrednio z tworzeniem solidnych systemów informatycznych poświęcono do tej pory niewiele książek i artykułów, a zdobycie praktycznej wiedzy w tym zakresie jest stosunkowo trudne. Z tego powodu w ramach cyklu w kolejnych artykułach postaram się w sposób systematyczny przedstawić następujące zagadnienia:

- Podstawowe pojęcia i mechanizmy związane z budową systemów transakcyjnych.
- Sposób obsługi transakcji w serwerze aplikacji.
- Korzystanie z baz danych i systemów kolejkowania a transakcje w serwerze aplikacji.
- Transakcje w komponentach EJB.
- Strategie obsługi transakcji w aplikacjach JEE.
- Problemy i ograniczenia związane z budową transakcyjnych aplikacji w technologii JEE.

Szczególną uwagę będę starał się poświęcić rzeczywistym problemom na jakie możemy się natknąć tworząc systemy transakcyjne i praktycznym rozwiązaniom, które możemy zastosować w codziennej pracy.

Wprowadzenie

Zastosowanie systemów transakcyjnych w aplikacjach bankowych, e-commerce, czy innych, w których w grę wchodzą pieniądze w zasadzie nie podlegają dyskusji. Ale można się zastanawiać, czy warto do prostszych systemów internetowych, systemów zarządzania treścią czy np. aplikacji forum internetowego dokładać jeszcze dodatkowy aspekt w postaci transakcji. Według mnie warto, co najmniej z jednego powodu. Dzięki transakcjom możemy zachować spójność danych w systemie. Nie ma nic gorszego niż próba naprawy niespójności danych. Dodajmy, że próba bez gwarancji sukcesu. Jednym słowem używając transakcji gwarantujemy sobie pewien poziom spokoju.

Rozważania na temat transakcji zacznę od przypomnienia kilku podstawowych definicji.

Transakcja - zestaw operacji na danych, który traktujemy jaką jedną całość i który cechuje się następującymi właściwościami:

- Jest niepodzielny (*atomicity*), czyli albo wszystkie operacje w ramach transakcji zostaną wykonane albo żadna.
- Zachowuje spójność danych (*consistency*). To znaczy, że jeśli na dane obrabiane przez system nałożone są pewne warunki logiczne, to warunki te muszą być spełnione zarówno przed jak i po wykonaniu transakcji. Należy zwrócić uwagę, że nie ma wymagania, aby w trakcie trwania transakcji dane były spójne!
- Jest izolowany (*isolation*). Jeśli w jednym systemie wiele transakcji wykonywane jest współbieżnie to z punktu widzenia pojedynczej transakcji powinno to wyglądać tak, jakby wszystkie transakcje były wykonywane po kolei. Mówimy wtedy, że transakcje wykonywane są we wzajemnej izolacji. Ta właściwość, szczególnie w odniesieniu do baz danych przysparza sporo problemów.

> Wymienione wyżej właściwości określane są skrótem *ACID*

Jest to spowodowane potrzebą zwiększenia wydajności kosztem niepełnej izolacji transakcji. Temat ten postaram się szczegółowo omówić w części związanej z bazami danych.

- Jest trwały (*durability*). Zmiana danych, których dokonała transakcja muszą być trwałe, nawet w przypadku awarii systemu tuż po zakończeniu transakcji.

Wymienione wyżej właściwości określane są skrótem *ACID* pochodzącym od pierwszych liter ich angielskich nazw. A samą transakcję zazwyczaj określa się jako *transaction*, ale często również używa terminu *unit of work* (UOW).

- **System transakcyjny** (*transactional system*) – system, w którym wszystkie operacje na danych grupowane są w transakcje.
- **Zasób transakcyjny** (*transactional resource*) – system (podsystem), który umożliwia operowanie na danych w sposób transakcyjny (np. baza danych).
- **Zakres transakcji** (*transaction scope*) – obszar działania programu od momentu rozpoczęcia transakcji do jej zatwierdzenia lub wycofania.
- **Zatwierdzanie transakcji** (*transaction commit*) – operacja trwałego wprowadzenia zmian w danych, które zaszły od momentu rozpoczęcia transakcji.
- **Wycofanie transakcji** (*transaction rollback*) – operacje wycofania wszystkich zmian w danych, które zaszły od momentu rozpoczęcia transakcji.

Typowym przykładem systemu transakcyjnego (a jednocześnie zasobu transakcyjnego) są systemy relacyjnych baz danych. Przez wiele lat systemy baz danych były w praktyce synonimem systemów transakcyjnych. Wymagania związane z budową złożonych systemów, w szczególności systemów, które muszą się integrować z innymi systemami zmieniły świat systemów transakcyjnych. Nie da się jednak zaprzeczyć, że bazy danych grają w nim jedną z głównych ról.

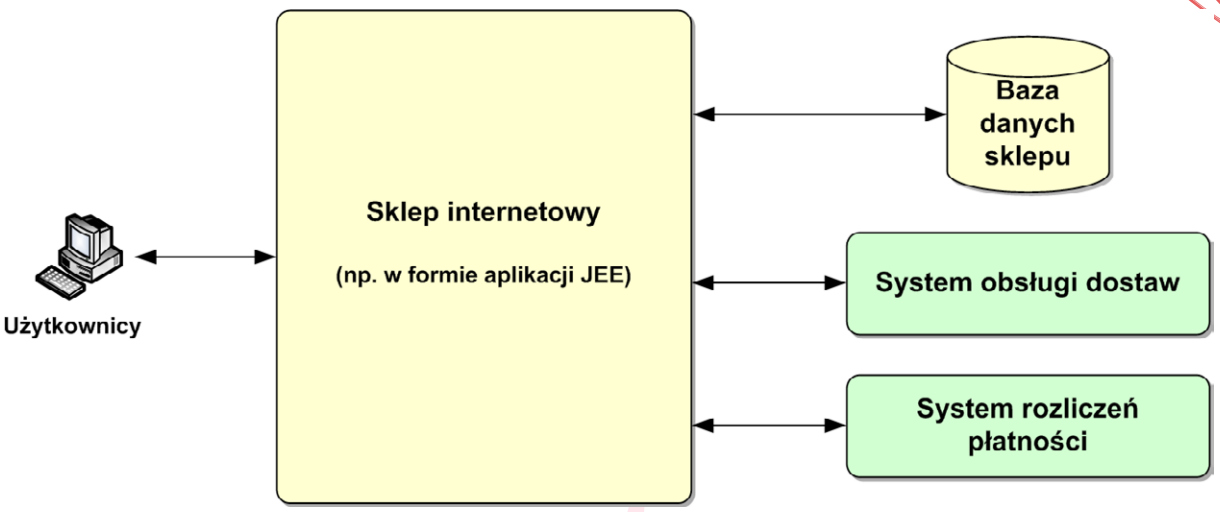
Transakcje rozproszone

Aby zagadnienie integracji systemów dokładniej zilustrować przyjrzyjmy się dość typowej konstrukcji współczesnego sklepu internetowego poglądowo przedstawionej na rysunku 1. System sklepu jest wyposażony we własną bazę danych, w której gromadzone są zarówno informacje o ofercie sklepu jak i o zamówieniach składanych przez klientów.

Ale to nie wszystko. Zamówione towary trzeba jakoś dostarczać. Obsługą tego procesu zajmuje się zazwyczaj odrębny system wspierających logistykę dostaw. Zamówienia trzeba również rozliczać, fakturować itd. Dochodzi więc system rozliczeń płatności, czy system finansowo księgowy. Oczywiście można to dalej komplikować przez dodawanie kolejnych systemów, takich jak obsługa płatności elektronicznych, system gospodarki magazynowej, systemy powiadomień o statusie zamówień (mail, sms) itd.

Oczywiście końcowego użytkownika nie interesuje to, ile systemów w rzeczywistości stoi za aplikacją sklepu internetowego. Składając zamówienie spodziewa się on, że dostanie towar i fakturę, czyli traktuje swoje działanie jako pojedynczą transakcję. To na aplikacji sklepu ciąży zadanie doprowadzenia do sytuacji, w której wszystkie systemy zagrają razem.

Taki mniej więcej jest obraz współczesnych



Rysunek 1 Przykład sklepu internetowego w architekturze rozproszonej

systemów informatycznych, które nam przychodzi w ostatnim czasie tworzyć, a w których występuje nieustająca potrzeba integracji różnych systemów.

Rodzi to z kolei innego rodzaju problem, mianowicie problem transakcji rozproszonej, czyli takiej w której pojedyncza aplikacja wykonuje operacje na różnych systemach, które nic o sobie nie wiedzą, jednocześnie zachowując wszystkie właściwości transakcji, o których wspominałem na początku (*ACID*).

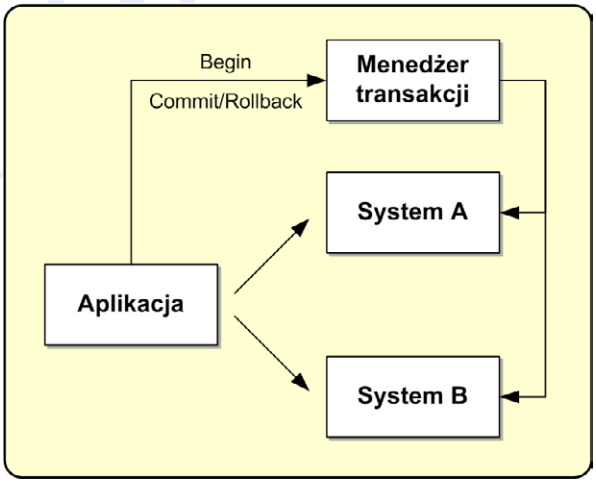
Oczekiwany przebieg transakcji rozproszonej wygląda następująco:

- Rozpoczęcie transakcji.
- Użycie kilku zasobów transakcyjnych.
- Zatwierdzenie lub wycofanie transakcji, co powinno spowodować określone skutki we wszystkich zasobach uczestniczących w transakcji.

Implementacja pojedynczego systemu transakcyjnego, takiego jak na przykład relacyjna baza danych nie jest zadaniem trywialnym. W przypadku systemu, który musi wspierać transakcje rozproszone skala trudności znacząco rośnie. Podstawową trudnością jest rozwiązanie problemu komunikacji między systemami, które często były tworzone jako całkowicie niezależne produkty. Poradzono sobie z tym konstruując oprogramowanie menedżera transakcji i wymyślając dwufazowy protokół zatwierdzania transakcji, o którym teraz kilka słów.

Dwufazowy protokół zatwierdzania transakcji

Przyjrzyjmy się jak działa dwufazowy protokół zatwierdzania transakcji. Przykład na rysunku 2 pokazuje dwa niezależne systemy A i B, które muszą uczestniczyć w transakcji rozproszonej. Wprowadzony jest również menedżer transakcji, który w imieniu aplikacji korzystającej z systemów A i B zarządza transakcją. Po wykonaniu operacji na systemach A i B, aplikacja żąda od menedżera zatwierdzenia transakcji.



Rysunek 2 Infrastruktura systemu zarządzania transakcjami rozproszonymi

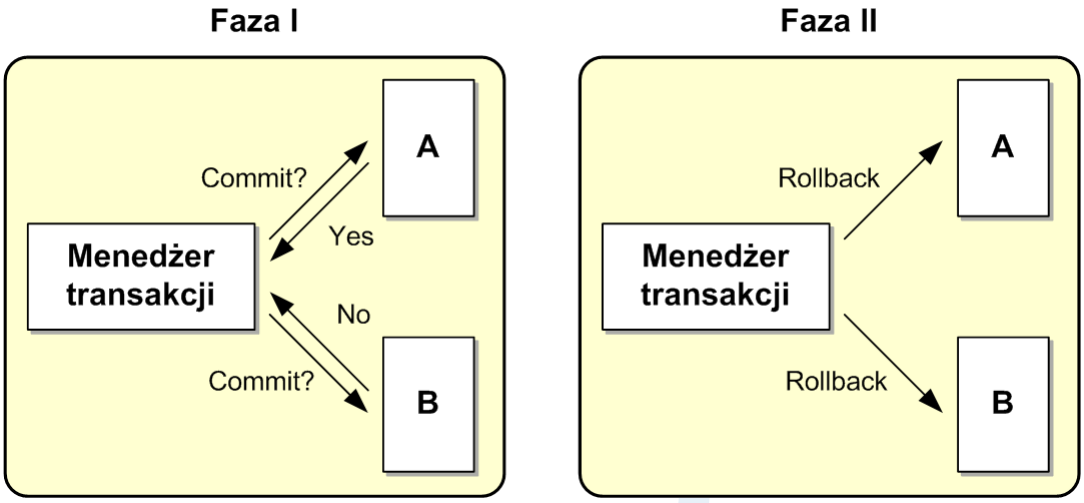
Menedżer transakcji w pierwszej fazie, którą określa się fazą przygotowania (*prepare*), pyta wszystkie systemy, czy są go-

> Implementacja pojedynczego systemu transakcyjnego, takiego jak na przykład relacyjna baza danych nie jest zadaniem trywialnym.

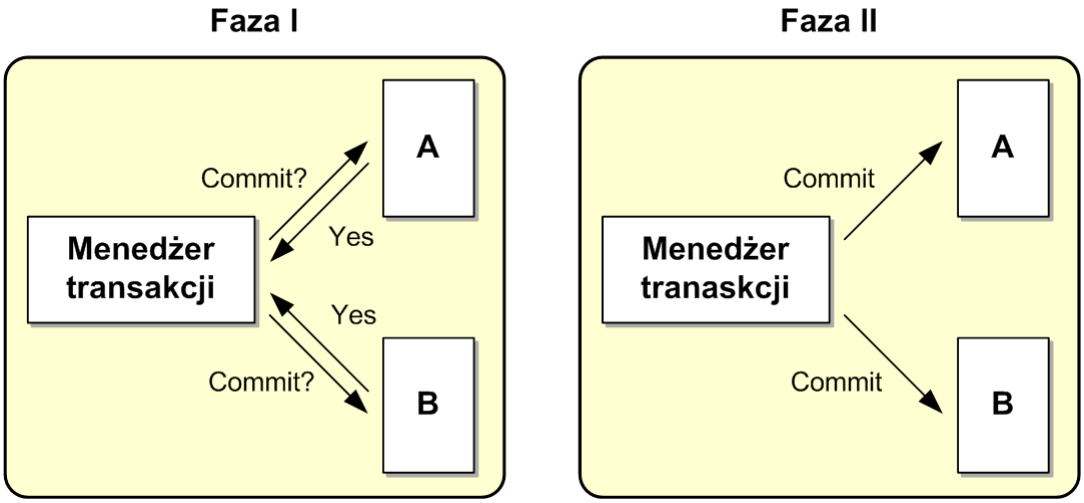
towe do zatwierdzenia swoich lokalnych zmian. Każdy system może udzielić jednej z dwóch odpowiedzi: tak lub nie. Jeśli choć jeden z systemów nie zgodzi się na zatwierdzenie transakcji menedżer transakcji w fazie drugiej do wszystkich systemów wysyła rozkaz wycofania transakcji (tą sytuację ilustruje rysunek 3). Jeśli wszystkie systemy odpowiedzą, że są gotowe do zatwierdzenia transakcji menedżer transakcji w fazie drugiej do wszystkich systemów wysyła rozkaz zatwierdzenia transakcji (tą sytuację ilustruje rysunek 4). Po zatwierdzeniu lub wycofaniu transakcji sterowanie wraca do aplikacji.

W rzeczywistych implementacjach, gdzie mamy do czynienia z systemami rozproszonymi w sensie logicznym lub fizycznym zachodzi szereg warunków brzegowych, z którymi zarówno systemy i menedżer transakcji musi sobie radzić. Przede wszystkim występuje szereg możliwości wystąpienia awarii:

- jednego z systemów,
- komunikacji między systemem a menedżerem transakcji,
- wreszcie samego menedżera transakcji.



Rysunek 3 Dwufazowy protokół zatwierdzenia transakcji – wycofanie transakcji



Rysunek 4 Dwufazowy protokół zatwierdzenia transakcji – zatwierdzenie transakcji

> Architektura systemu,
> w którym realizowane są transakcje rozproszone
> składa się z trzech głównych elementów

System transakcyjny musi sobie z tymi problemami umieć radzić. Temat związany z odtwarzaniem systemów po awarii jest bardzo obszerny i nie mamy tutaj miejsca na jego choćby skromne omówienie. Warto według mnie zwrócić uwagę jedynie na to, że w większości przypadków przywrócenie systemu do działania po awarii może następować automatycznie. Jedynie część specyficznych awarii związanych z awarią samego menedżera transakcji uniemożliwia automatyczne odtworzenie i wymaga interwencji operatora.

Istotą dwufazowego protokołu zatwierdzania transakcji jest to, że jeżeli w pierwszej fazie system zgłosił gotowość do zatwierdzenia transakcji, to w drugiej fazie nie może się już z tej decyzji wycofać. Takie zachowanie musi gwarantować implementacja systemu uczestniczącego w transakcji rozproszonej. Jest to więc swoistego rodzaju kontrakt między menedżerem transakcji rozproszonej a systemem w niej uczestniczącym.

Uczestnicy transakcji rozproszonej

Architektura systemu, w którym realizowane są transakcje rozproszone składa się z trzech głównych elementów (patrz rysunek 5):

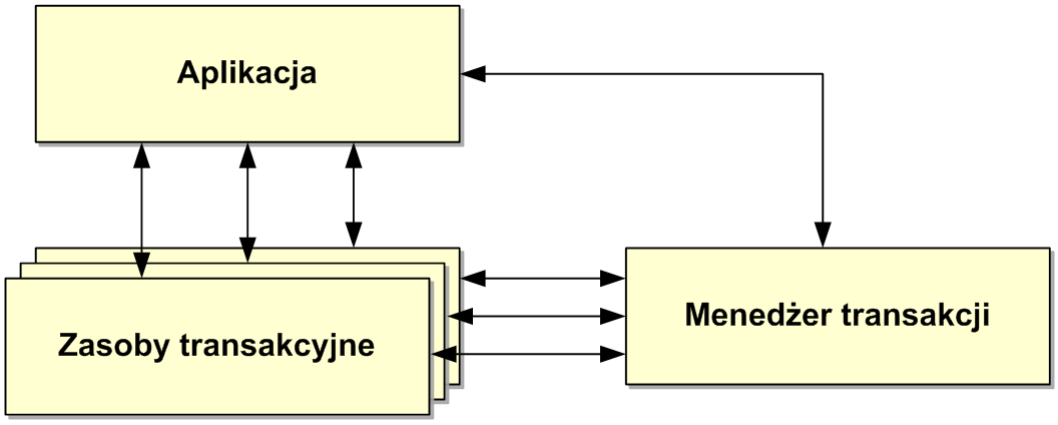
- Aplikacji, czyli tego co zazwyczaj piszemy jako deweloperzy systemów.
- Zasobów transakcyjnych, z których nasza aplikacja korzysta (np. baz danych).
- Menedżera transakcji, który zarządza transakcją rozproszoną w imieniu aplikacji (zwykle dostarczany przez serwer aplikacji).

Każdy z tych elementów ma swoje specyficzne zadania do zrealizowania. Zacznijmy od zadań, które ciążą na aplikacji:

- Aplikacja (a więc my) przede wszystkim zarządza zakresem transakcji, czyli decyduje gdzie transakcja ma się zacząć, a gdzie zakończyć.
- Przeprowadza operacje na zasobach transakcyjnych, takich jak bazy danych, systemy kolejkowania czy transakcyjne systemy plików.

Przyjrzyjmy się teraz zadaniom menedżera transakcji:

- Menedżer transakcji przede wszystkim tworzy transakcję i zarządza kontekstem transakcji.
- Kojarzy również transakcję z zasobami w niej uczestniczącymi.



Rysunek 5 Uczestnicy transakcji rozproszonej

> W czasie trwania transakcji jej kontekst przekazywany jest poszczególnym uczestnikom transakcji

- Wreszcie prowadzi operacje zatwierdzania lub wycofywania transakcji bazując na dwufazowym protokole zatwierdzania.

Na koniec zadania zasobów transakcyjnych:

- Podstawową ich rolą jest umożliwienie aplikacji operowanie na danych, które przechowują.
- Dodatkowo zasoby transakcyjne muszą umieć współpracować z menedżerem transakcji, w szczególności w ramach dwufazowego protokołu zatwierdzania.

Kilka zdań wyjaśnienia należy się pojęciu kontekstu transakcji (*transaction context*). Kontekst transakcji to nic innego jak bieżący stan transakcji, w szczególności identyfikator transakcji oraz informacje o zasobach uczestniczących w transakcji. W czasie trwania transakcji jej kontekst przekazywany jest poszczególnym uczestnikom transakcji, co jest znane pod pojęciem propagacji kontekstu transakcji.

Na tym chciałbym zakończyć ogólne wprowadzenie do systemów transakcyjnych. Więcej szczegółów można znaleźć w materiałach pomocniczych [3,5]. Teraz przejdźmy do omówienia jak to jest zrealizowane w serwerze JEE.

Transakcje w środowisku serwera aplikacyjnego JEE

Zacznijmy od omówienia modeli transakcji jakie serwer aplikacji udostępnia deweloperem piszącym aplikacje. W praktyce mamy możliwość pisania systemów transakcyjnych na trzy różne sposoby.

Transakcje lokalne. Serwer aplikacji umożliwia wykonywania operacji w sposób transakcyjny na pojedynczym zasobie, np. na bazie danych. Przy czym zarządzanie transakcjami realizuje programista korzystając z właściwości konkretnego zasobu, np. transakcje w obrębie pojedynczej bazy danych realizujemy z poziomu API zdefiniowanego przez specyfikację JDBC. Rola serwera aplikacji ogranicza się tutaj jedynie do udostępnienia zasobów, na których operuje aplikacja, np. źródła danych do bazy danych (`javax.sql.DataSource`).

Transakcje zarządzane przez programistę z wykorzystaniem interfejsów zdefiniowanych przez specyfikację JTA. Są to takie transakcje, w których programista pisząc kod, jawnie określa początek i koniec transakcji korzystając z JTA. Musi również umieć obsłużyć szereg sytuacji brzegowych, czy wyjątkowych związanych z korzystaniem z tych interfejsów.

Transakcje zarządzane przez kontener, tzw. deklaratywne. W tym przypadku programista tworzy komponenty EJB i w deskryptorze komponentów określa jakie ma być zachowanie transakcyjne poszczególnych metod. Całością obsługi transakcji zajmuje się serwer aplikacji, a ściślej kontener komponentów EJB we współpracy z menedżerem transakcji.

Transakcje lokalne oraz transakcję zarządzane przez kontener omówię w oddzielnych artykułach. Natomiast w tym artykule przedstawię dokładniej transakcje zarządzane przez programistę, które moim zdaniem najlepiej ilustrują wsparcie serwera aplikacji w tworzeniu systemów transakcyjnych.

Środowisko transakcyjne dla aplikacji JEE jest określone przez dwie specyfikacje:

- **Java Transaction API (JTA)** - definiuje

> JTA ma się mniej więcej tak do JTS
> jak specyfikacja JDBC do sterownika do bazy danych.

ona sposób zarządzania transakcjami z punktu widzenia programisty. Określa również sposób współpracy z zasobami uczestniczącymi w transakcji rozproszonej (jest to odwzorowanie fragmentu standardu X/Open DTP - XA Interface).

- **Java Transaction Service (JTS)** - określa sposób implementacji menedżera transakcji (w szczególności wspierającego JTA), aczkolwiek specyfikacja JEE nie wymaga, żeby JTA było koniecznie implementowane w postaci JTS. Na rynku istnieje sporo implementacji JTA nie korzystających z JTS. JTS jest wymagane, jeśli myślimy o współpracy menedżerów transakcji w środowisku rozproszonym (transakcja rozproszona pomiędzy kilkoma serwerami aplikacji różnych dostawców). JTS jest tak naprawdę mapowaniem CORBA Object Transaction Service na język Java.

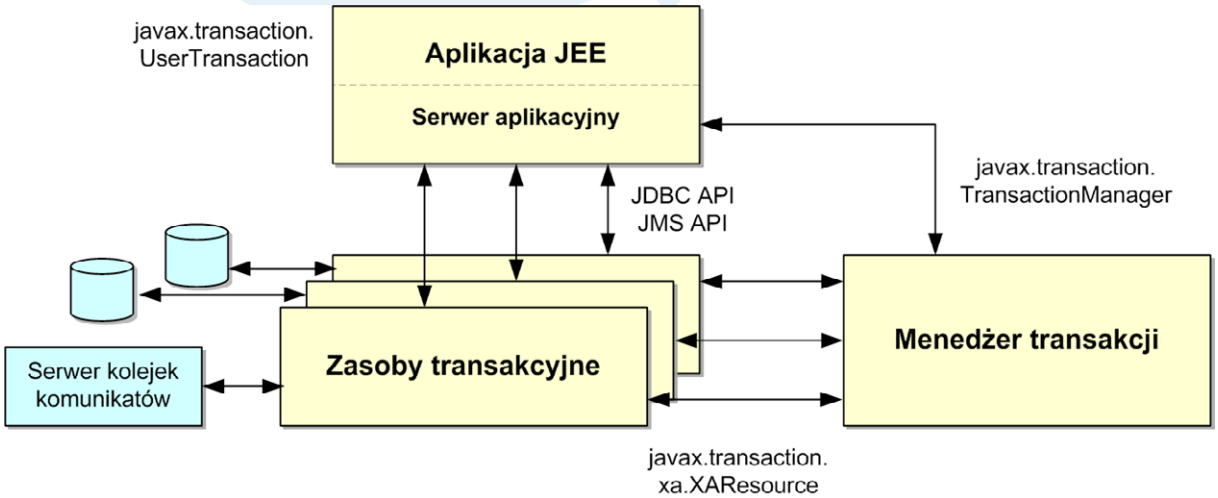
Najłatwiej zrozumieć związek pomiędzy tymi specyfikacjami przez analogię. JTA ma się mniej więcej tak do JTS jak specyfikacja JDBC do sterownika do bazy danych. Z punktu widzenia programisty praktyczne znaczenie ma JTA i tym będę się wyłącznie zajmował.

Przyjrzyjmy się teraz interfejsom jakie w serwerze aplikacji używane są do współpracy poszczególnych elementów uczestniczących w przetwarzaniu transakcji. Szczegóły przedstawiłem na rysunku 6.

Dla aplikacji pracującej w serwerze aplikacji, która chce zarządzać transakcją został przygotowany interfejs `javax.transaction.UserTransaction`. Wywołania tego interfejsu serwer aplikacji deleguje do interfejsu `javax.transaction.TransacationManager`, który stanowi reprezentację menedżera transakcji w serwerze aplikacji (implementacja tego interfejsu stanowi serce menedżera transakcji).

Aplikacja z poszczególnymi zasobami transakcyjnymi komunikuje się przez interfejsy specyficzne dla danych zasobów, np. przez interfejs JDBC albo JMS. Zachowanie transakcyjne zasobów jest przezroczyste z punktu widzenia aplikacji. Oczywiście sterowniki do zasobów muszą być odpowiednio zaimplementowane, ale zadanie to spoczywa na dostawcy sterownika, a nie na programiście aplikacji.

Pozostaje jeszcze interfejs `XAResource` za pomocą którego odbywa się komunikacja między menedżerem transakcji a zasoba-



Rysunek 6 Uczestnicy transakcji rozproszonych w środowisku serwera aplikacji JEE

> W praktyce dość często można zaobserwować przypadki korzystania z baz danych w transakcjach rozproszonych z nieprawidłowo skonfigurowanym zasobem.

mi transakcyjnymi. Aby móc uczestniczyć w transakcji rozproszonej zasoby te muszą implementować `javax.transaction.xa.XAResource`. Interfejs ten stanowi kontrakt określający sposób współpracy menedżera transakcji z zasobami transakcyjnymi.

Jeżeli używamy transakcji JTA **ważne jest upewnienie się, czy zasoby na których operujemy implementują `XAResource`** oraz czy są odpowiednio skonfigurowane w serwerze. W praktyce dość często można zaobserwować przypadki korzystania z baz danych w transakcjach rozproszonych z nieprawidłowo skonfigurowanym zasobem. Wynika to z tego, że zazwyczaj sterowniki JDBC w jednym fizycznym archiwum `jar` zawierają zarówno klasy implementujące zwykły dostęp jak i przystosowany do pracy w transakcji rozproszonej. Łatwo zatem o pomyłkę, zwłaszcza, że serwery nie ostrzegają przed użyciem zasobu nie będącego XAResource'em w transakcji JTA. Jest to zresztą zrozumiałe, gdyż nie wszystkie zasoby, na których operujemy są w stanie uczestniczyć w takiej transakcji - np. wysyłanie poczty elektronicznej.

Zarządzanie transakcjami za pomocą interfejsu UserTransaction

Przejdźmy do omówienia najważniejszego interfejsu z punktu widzenia programisty zainteresowanego zarządzaniem transakcjami. W serwerze aplikacji jest to interfejs `UserTransaction`. Jest to skromny interfejs o następujących metodach (pominąłem specyfikację wyjątków, które mogą być rzucone przez te metody):

```
void begin();
void commit();
void rollback();

void setRollbackOnly();
int getStatus();
void setTransactionTimeout(
    int seconds);
```

Jak widać interfejs składa się z dwóch grup metod. Pierwszej pozwalającej zarządzać zakresem transakcji, czyli rozpocząć transakcję (`begin`) oraz ją zakończyć zatwierdzając (`commit`) lub wycofując (`rollback`). Drugiej niejako pomocniczej pozwalającej sterować pewnymi elementami transakcji a także uzyskać informacje o aktualnym stanie transakcji.

Pierwsza grupa metod moim zdaniem nie wymaga szerszego komentarza. Natomiast druga grupa metod jest bardziej intrygująca i wymagająca dodatkowych objaśnień ponad to, co można wyczytać z suchego Java Transaction API.

Zacznijmy od metody `setRollbackOnly`. Pozwala ona na oznaczenie transakcji do wycofania. Oznacza to, że jedynym dopuszczalnym działaniem, które kończy transakcję jest jej wycofanie. Próba zatwierdzenia takiej transakcji zakończy się wyjątkiem. Zwróćmy uwagę na to, że metoda ta nie przerywa działania naszego programu - ustawia jedynie odpowiedni status transakcji. Zazwyczaj używamy tej metody w przypadku, gdy z pewnych warunków biznesowych lub współpracy z innymi systemami w sposób nietransakcyjny wynika, że musimy transakcję wycofać, ale powinniśmy wykonać jeszcze szereg innych działań (czyli nie przerywać wykonania naszego programu).

Z kolei za pomocą metody `getStatus` możemy zbadać w jakim obecnie stanie znajduje się nasza transakcja. Specyfikacja JTA za pomocą stałych z interfejsu `javax.transaction.Status` definiuje szereg stanów, w których może znaleźć się transakcja:

> Większość z tych stanów związana jest z realizacją dwufazowego protokołu zatwierdzania transakcji

```
NO_TRANSACTION
ACTIVE
PREPARING
PREPARED
COMMITTED
COMMITTING
ROLLING_BACK
ROLLEDBACK
MARKED_ROLLBACK
UNKNOWN
```

Większość z tych stanów związana jest z realizacją dwufazowego protokołu zatwierdzania transakcji i jest wykorzystywana wewnętrznie przez menedżer transakcji. Z praktycznego punktu widzenia dla programisty istotne są poniższe stany:

- `ACTIVE` - oznaczający, że transakcja trwa (została rozpoczęta).
- `NO_TRANSACTION` - oznaczający, że z aktualnym wątkiem nie jest związana żadna transakcja. Status ten można wykorzystać do zweryfikowania, czy w metodach, które wymagają do prawidłowego działania rozpoczętej wcześniej transakcji, została ona faktycznie rozpoczęta.
- `MARKED_ROLLBACK` - oznaczająca, że transakcja została oznaczona do wycofania. Status ten sprawdzamy przy samodzielnym zarządzaniu transakcjami, aby podjąć decyzję o zatwierdzeniu czy wycofaniu transakcji. Często również na jego podstawie możemy zaprezentować użytkownikowi odpowiednią informację o rezultacie transakcji, którą próbował wykonać.

Na koniec omawiania interfejsu `UserTransaction` kilka słów o timeout'ach transakcji. Metoda `setTransactionTimeout(...)` pozwala na ustawienie maksymalnego dopuszczalnego czasu trwania transakcji. Jednak jej działanie jest chyba najmniej oczywiste ze wszystkich metod związanych z zarządzaniem transakcjami (nie tylko w obrębie `UserTransaction`). Przyjrzyjmy się jej dokładnie:

- Po pierwsze z ustawienia maksymalnego czasu trwania transakcji wcale nie wynika, że transakcja zakończy się po tym czasie. Co więcej, nie możemy oczekiwać, że nawet w dowolnym momencie po przekroczeniu tego czasu działanie naszej aplikacji zostanie przerwane. W rzeczywistości po przekroczeniu tego czasu serwer aplikacji jedynie oznacza tą transakcję do wycofania (`MARKED_ROLLBACK`). Tak więc transakcja wykonuje się cała (nawet jeśli trwałaby godzinami), a dopiero na końcu jest wycofywana z powodu przekroczenia tego czasu. Jest to zachowanie sprzeczne z zazwyczaj wyrobioną intuicją programisty, która podpowiada, że działanie aplikacji powinno zostać przerwane (tak jak jest to na przykład przy obsłudze socket'ów). Takie zachowanie transakcji rodzi wiele problemów, które szerzej będę omawiał w oddzielnym artykule.
- Po drugie należy pamiętać, że jeśli samodzielnie ustawiamy maksymalny czas trwania transakcji musimy to zrobić przed wywołaniem metody `begin`.
- Po trzecie należy pamiętać, że w serwerze aplikacji zawsze jest zdefiniowany jakiś domyślny czas trwania transakcji. Jest on specyficzny dla danego serwera, zazwyczaj jest ustawiony na poziomie 60-120 sekund. Dość łatwo o tym zapomnieć. Dlatego jeśli aplikacja wydaje się działać poprawnie, ale dane w bazie danych się nie pojawiają proponuję sprawdzić, czy czasami transakcja nie została wycofana z powodu przekroczenia maksymalnego czasu jej trwania.

> Przyjrzyjmy się teraz pierwszemu przykładowi wykorzystania interfejsu do zarządzania transakcjami

Przyjrzyjmy się teraz pierwszemu przykładowi wykorzystania interfejsu do zarządzania transakcjami z poziomu prostej aplikacji serwlet'owej operującej na dwóch bazach danych w ramach pojedynczej transakcji rozproszonej. Pominąłem tutaj właściwą obsługę wyjątków i sytuacji brzegowych, którą omówię w kolejnym przykładzie.

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletException;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;
...

public class SimpleTransactionDemoServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res)
      throws ServletException, IOException {
    ...
    try {
      Context ctx = new InitialContext();
      UserTransaction ut = (UserTransaction) ctx
          .lookup("java:comp/UserTransaction");
      ;

      ut.begin();

      DataSource ds1 = (DataSource) ctx
          .lookup("java:comp/env/jdbc/TransactionDemoDS1");
      DataSource ds2 = (DataSource) ctx
          .lookup("java:comp/env/jdbc/TransactionDemoDS2");

      doSomethingInFirstDatabase(ds1);
      doSomethingInSecondDatabase(ds2);

      ut.commit();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
    ...
  }
}
```

Przeanalizujemy ten krótki przykład. Cała zabawa rozpoczyna się od pobrania z JNDI obiektu `UserTransaction`. W większości serwerów aplikacji znajduje się on pod kluczem `java:comp/UserTransaction`, ale są serwery, które udostępniają go pod innym kluczem (np. WebSphere v4 pod kluczem `jta/usertransaction`). Dlatego należy się najpierw upewnić w dokumentacji serwera, którego używamy, gdzie w JNDI można znaleźć obiekt `UserTransaction`.

Następnie rozpoczynamy transakcję za pomocą operacji `begin`, pobieramy skonfigurowane wcześniej źródła danych i wykonujemy działania na bazach danych.

Po wykonaniu operacji na bazach danych

zatwierdzamy transakcję, dane trafiają na trwałe do odpowiednich baz danych. Jednym słowem lekko, łatwo i przyjemnie. Niestety nie do końca, jak to będę starał się pokazać na kolejnym przykładzie.

Niemal identycznie wygląda zarządzanie transakcjami w przypadku pisania komponentów EJB, które samodzielnie zarządzają transakcjami (*bean managed transcations*). Jedyną różnicą jest sposób dostępu do interfejsu `UserTransaction`, zamiast pobierać go bezpośrednio z JNDI korzystamy z metody `getUserTransaction()` z interfejsu `javax.ejb.EJBContext`.

Obsługa wyjątków i sytuacji brzegowych przy samodzielnym zarządzaniu transakcjami

Niestety, przedstawiony wcześniej przykład zarządzania transakcjami poza swoją zachęcającą prostotą ma jedną zasadniczą wadę. Mianowicie w praktyce nie działa, gdyż nie obsługuje sytuacji brzegowych i wyjątkowych, które mogą się wydarzyć podczas wykonywania poszczególnych operacji. Pokazuje jedynie tzw. ścieżkę pozytywną, czyli taką, w której wszystko co sobie zamierzyliśmy wykonało się poprawnie.

Przyjrzyjmy się teraz w jaki sposób można podejść do pełnej obsługi procesu zarządzania pojedynczą transakcją:

```
Context ctx = null;
UserTransaction ut = null;

try {
  ctx = new InitialContext();
  ut = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
  ;
} catch (NamingException e) {
  throw new RuntimeException(e);
}

try {
  ut.begin();
} catch (NotSupportedException e) {
  throw new RuntimeException(e);
} catch (SystemException e) {
  throw new RuntimeException(e);
}

boolean ok = false;
Exception exception = null;

try {
  try {
    DataSource ds1 = (DataSource) ctx
        .lookup("java:comp/env/jdbc/TransactionDemoDS1");
    DataSource ds2 = (DataSource) ctx
        .lookup("java:comp/env/jdbc/TransactionDemoDS2");

    doSomethingInFirstDatabase(ds1);
    doSomethingInSecondDatabase(ds2);

    ok = true;
  } catch (Exception e) {
    exception = e;
  }
} finally {
  Exception cleanupException = null;
  if (ok) {
```

```
    try {
      int transactionStatus = ut.getStatus();
      if (transactionStatus == Status.STATUS_ACTIVE) {
        ut.commit();
      } else if (transactionStatus == Status.STATUS_MARKED_ROLLBACK) {
        ut.rollback();
      } else {
        if (log.isWarnEnabled()) {
          log.warn("Unexpected transaction status: " + ut.getStatus());
        }
      }
    } catch (SystemException e) {
      cleanupException = e;
    } catch (RollbackException e) {
      cleanupException = e;
    } catch (HeuristicMixedException e) {
      cleanupException = e;
    } catch (HeuristicRollbackException e) {
      cleanupException = e;
    }
  } else {
    try {
      int transactionStatus = ut.getStatus();
      if (transactionStatus == Status.STATUS_ACTIVE
          || transactionStatus == Status.STATUS_MARKED_ROLLBACK) {
        ut.rollback();
      } else {
        if (log.isWarnEnabled()) {
          log.warn("Unexpected transaction status: " + ut.getStatus());
        }
      }
    } catch (SystemException e) {
      cleanupException = e;
    }
  }

  if (cleanupException != null) {
    if (exception != null) {
      throw new RuntimeException(doubleErrorMessage(exception),
          cleanupException);
    } else {
      throw new RuntimeException(cleanupException.getMessage(),
          cleanupException);
    }
  } else if (exception != null) {
    throw new RuntimeException(exception.getMessage(), exception);
  }
}
```

Prześledźmy kolejno najważniejsze elementy tej obsługi.

- Pierwszą rzeczą, która może się nie udać to pobranie obiektu `UserTransaction`. W tym przypadku nie pozostaje nam nic innego jak przerwanie działania aplikacji i rzucenie wyjątku czasu wykonania.
- Następną rzeczą, która może się nie udać, jest rozpoczęcie transakcji. Wbrew pozorom nie jest to rzadki problem. Najczęściej wynika on z próby rozpoczęcia transakcji w sytuacji, gdy już jakaś inna transakcja jest przypisana do wątku. Zazwyczaj otrzymujemy wtedy komunikat o tym, że serwer aplikacji nie wspiera transakcji zagnieżdżonych. W tym przypadku również

> Mamy wówczas do czynienia z tzw. podwójnym błędem.

niewiele możemy zrobić poza rzuceniem `RuntimeException`, gdyż błąd wynika albo bezpośrednio z błędu w naszej aplikacji albo zawodzi serwer aplikacji (co wbrew pozorom nie jest znowu takie wyjątkowe).

- Jeśli już uporamy się z rozpoczęciem transakcji, musimy obsłużyć sytuację, w której podczas wykonywania operacji biznesowych w ramach rozpoczętej transakcji wystąpił wyjątek. Ponieważ kontrakt współpracy z serwerem aplikacji narzuca nam konieczność zakończenia transakcji (zatwierdzenia lub wycofania) to w zasadzie jedynym wyjściem jest przechwycenie wyjątku, jego zapamiętanie, obsłużenie zakończenia transakcji i na końcu jego ponowne rzucenie dalej. Jeśli nie zrobimy tego w ten sposób transakcja pozostanie przypięta do wątku i będziemy mieli w kolejnych żądaniach wyjątek w rodzaju „Nested transactions not supported".

- Przejdźmy teraz do najtrudniejszej części, czyli obsługi zamykania transakcji. Do obsłużenia mamy trzy główne przypadki:

  - Operacje na zasobach przebiegły bez wyjątku, sama transakcja ma status `ACTIVE`. W takim przypadku zatwierdzamy transakcję (`commit`).

  - Operacje na zasobach przebiegły bez wyjątku, ale transakcja ma status `MARKED_ROLLBACK`. W takim przypadku musimy wycofać transakcję (`rollback`).

  - Podczas wykonywania operacji na zasobach wystąpił wyjątek. Wtedy niezależnie od tego czy transakcja jest w stanie `ACTIVE` czy `MARKED_ROLLBACK` musimy wycofać transakcję (`rollback`).

- Niestety podczas zatwierdzania lub wycofywania transakcji również może wystąpić wyjątek. W takim przypadku wyjątek ten musimy zapamiętać (w przykładzie zmienna `cleanupException`) a następnie na samym końcu obsługi go rzucić. Szczególna sytuacja zachodzi wówczas, gdy zarówno podczas wykonywania operacji biznesowych jak i podczas zamykania transakcji pojawią się wyjątki. Mamy wówczas do czynienia z tzw. podwójnym błędem. Należy zwrócić uwagę, żeby błędy te wzajemnie się nie maskowały powodując trudne do odszyfrowania problemy. Dlatego w przykładzie w takim przypadku w sposób specjalny konstruowany jest komunikat wyjątku tak, aby zawierał informację o wyjątku pierwotnym (metoda `doubleErrorMessage`). Przykładowa implementacja tej metody może wyglądać następująco:

```
public String doubleErrorMessage-
(Throwable exception) {
  StringWriter sw = new StringW-
riter();
  PrintWriter pw = new PrintWri-
ter(sw);
  exception.printStackTrace(pw);
  pw.close();
  return "double error; original
cause:\n[\n" + sw.toString() +
"]";
}
```

Chwilę uwagi chciałbym jeszcze poświęcić wyjątkom typu `Heuristic...Exception`. Kiedy się pierwszy raz z nimi zetknąłem wydawały mi się czymś magicznym, nie bardzo przystającym do rzeczywistości. Jednak po dokładniejszym rozeznaniu się tematyce systemów transakcyjnych uświadomiłem sobie, że nie są one czymś nadzwyczajnym. Sygnalizują jedynie pew-

> Czy to jakaś magiczna sztuczka serwera aplikacji?

ne sytuacje wyjątkowe, które mogą się wydarzyć na styku współpracy menedżera transakcji i menedżera konkretnego zasobu w ramach dwufazowego protokołu zatwierdzania. I tak na przykład wyjątek `HeuristicRollbackException` możemy zobaczyć, jeśli w pierwszej fazie zasób transakcyjny zgłosił gotowość zatwierdzenia transakcji, ale pomiędzy zakończeniem pierwszej fazy a rozpoczęciem drugiej samodzielnie podjął decyzję o wycofaniu zmian, które zaszły w tej transakcji. Dokładne omówienie przypadków związanych z tymi wyjątkami można znaleźć w [4].

Jak można zauważyć kompletna obsługa samodzielnego zarządzania transakcjami wymaga niemało wysiłku, a powstały kod jest niezbyt czytelny (mizerny stosunek prawdziwej logiki biznesowej do logiki związanej z obsługą wyjątków). Na szczęście stosunkowo łatwo można go zamknąć w pojedynczej klasie usługowej i nie powielać w innych częściach aplikacji.

Bezpośrednie korzystanie z menedżera transakcji

Korzystając jedynie z interfejsu `UserTransaction` nie jesteśmy w stanie wykorzystać wszystkich możliwości jakie daje nam model transakcji zdefiniowany przez serwer aplikacji JEE. W szczególności nie możemy zawieszać oraz wznawiać wykonania bieżącej transakcji.

W praktyce taka możliwość czasami się przydaje. Wyobraźmy sobie aplikację bankowości elektronicznej, w której chcielibyśmy logować do oddzielnej bazy danych wszystkie próby wykonywanych przez użytkowników transakcji niezależnie od tego czy zakończyły się one zatwierdzeniem czy wycofaniem. Jeśli zapis do logów umieścimy w tej samej transakcji to system będzie się zachowywał prawidłowo jedynie w przypadku zatwierdzenia transakcji. W przypadku jej wycofanie w bazie danych odpowiedzialnej za logi nic nie zobaczymy.

Taką funkcjonalność można zrealizować odwołując się bezpośrednio do menedżera transakcji, co jest jak najbardziej dopuszczane przez specyfikację JTA. Tak jak wcześniej wspomniałem w serwerze aplikacji menedżer transakcji jest reprezentowany przez interfejs `TransactionManager`, który poza metodami, które znajdziemy w interfejsie `UserTransaction` posiada jeszcze dwie dodatkowe:

- `suspend()` - pozwalającą zawiesić bieżącą transakcję - metoda zwróci wtedy obiekt `Transaction`, który reprezentuje zawieszoną transakcję.
- `resume(Transaction t)` - pozwalający wznowić zawieszoną wcześniej transakcję.

Podobnie jak w przypadku `UserTransaction` obiekt `TransactionManager` znajduje się w JNDI pod odpowiednim kluczem. Klucz ten jest specyficzny dla danego serwera aplikacji.

Poniżej przedstawiłem przykład realizujący opisaną wyżej funkcjonalność. Dla jego czytelności pominąłem obsługę wyjątków.

Jako ciekawostkę można zauważyć, że zarówno obiekt `UserTransaction` jak i `TransactionManager` są pobierane z pod tego samego klucza JNDI (`java:comp/UserTransaction`). Czy to jakaś magiczna sztuczka serwera aplikacji? Otóż nie, przykład ten testowałem w serwerze aplikacji JOnAS, w którym jako menedżer transakcji używany jest komponent JOTM. W tej konkretnej implementacji menedżera

```
Context ctx = new InitialContext();
UserTransaction ut = (UserTransaction) ctx
    .lookup("java:comp/UserTransaction");
TransactionManager tm = (TransactionManager) ctx
    .lookup("java:comp/UserTransaction");

ut.begin();

DataSource ds1 = (DataSource) ctx
    .lookup("java:comp/env/jdbc/TransactionDemoDS1");
// wykonujemy operacje na pierwszej bazie danych (ds1)

Transaction t = tm.suspend();
DataSource dslog = (DataSource) ctx
    .lookup("java:comp/env/jdbc/TransactionDemoDSLog");
// wykonujemy niezależną od otworzonej wcześniej transakcji
// operację na bazie danych (dslog)
tm.resume(t);

DataSource ds2 = (DataSource) ctx
    .lookup("java:comp/env/jdbc/TransactionDemoDS2");
// wykonujemy operacje na drugiej bazie danych (ds2)

ut.commit(); // lub ut.rollback() jeśli coś poszło nie tak
```

transakcji obiekt, który znajduje się pod wskazanym kluczem w JNDI po prostu implementuje oba interfejsy `UserTransaction` i `TransactionManager`. Jeśli spojrzymy na te oba interfejsy i pokrywające się w nich metody to takie rozwiązanie wydaje się zrozumiałe.

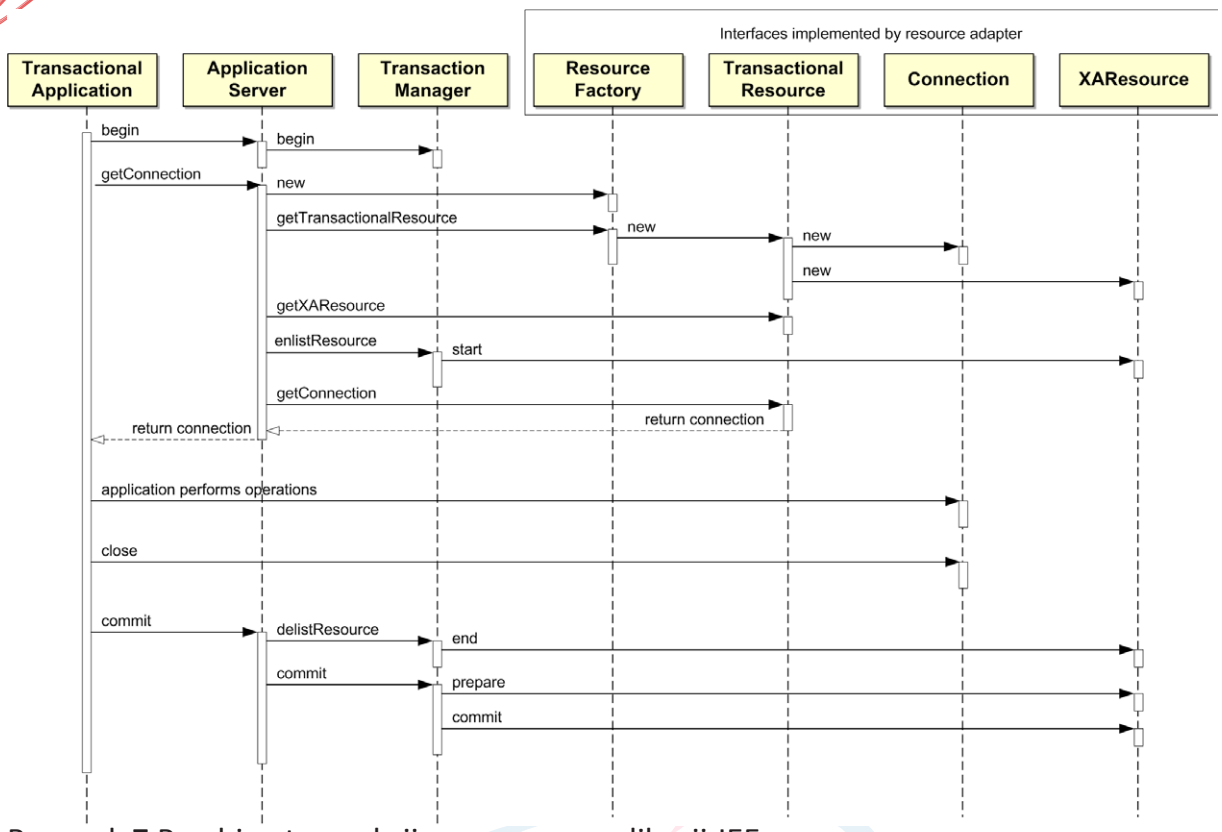
Powyższą funkcjonalność można również zrealizować za pomocą komponentów EJB odpowiednio sterując zachowaniem transakcyjnym poszczególnych ich metod. Przedstawię taki przykład w jednym z kolejnych artykułów.

Przebieg transakcji w serwerze aplikacji

Jako podsumowanie omawianych w tym artykule zagadnień chciałbym przedstawić całościowy obraz przebiegu pojedynczej transakcji w serwerze aplikacji JEE. Szczegółowe interakcje pomiędzy poszczególnymi elementami uczestniczącymi w transakcji przedstawia rysunek 7. Obrazuje on przebieg transakcji zarządzanej przez programistę. Przebieg transakcji zarządzanej przez kontener różni się tylko tym, że rozpoczęcie i zakończenia transakcji zamiast aplikacji wykonuje serwer aplikacji.

Zacznijmy od omówienie poszczególnych obiektów uczestniczących w transakcji:

- `Transactional Application` - reprezentuje naszą aplikację, z poziomu której zarządzamy transakcją.
- `Application Server` - udostępnia środowisko zarządzania transakcjami przez udostępnienie aplikacji interfejsu `UserTransaction`.
- `Transaction Manager` - realizuje faktyczne zarządzanie transakcjami w imieniu naszej aplikacji.
- `Resource Adapter` - reprezentuje zasób transakcyjny, na którym operuje nasza aplikacja. Zasób transakcyjny z punktu widzenia zarządzania transakcjami przez serwer aplikacji musi składać się z następujących elementów:
  - `Resource Factory` - czyli obiektu umiejącego powołać do życia zasób transakcyjny.
  - `Transactional Resource` - będący podstawowym interfejsem reprezentującym zasób transakcyjny.
  - `Connection` - interfejs reprezentu-



Rysunek 7 Przebieg transakcji w serwerze aplikacji JEE

jący fizyczne połączenie do zasobu transakcyjnego (np. połączenie do bazy danych, serwera JMS itp.).

- `XAResource` - interfejs umożliwiający realizację dwufazowego protokołu zatwierdzania transakcji.

Prześledźmy teraz interakcje jakie zachodzą pomiędzy poszczególnymi elementami.

- Nasza aplikacja rozpoczyna transakcję wołając `begin` na interfejsie `UserTransaction` pobranym z serwera aplikacji. Powoduje to utworzenie nowej transakcji w menedżerze transakcji (powstaje nowy kontekst transakcji) oraz skojarzenie wątku aktualnie wykonującego kod naszej aplikacji z nowoutworzoną transakcją.
- W następnym kroku nasza aplikacja musi uzyskać połączenie do zasobu na którym chce operować. Odbywa się to przez wywołanie metody `getConnection` na obiekcie, który reprezentuje zasób transakcyjny w serwerze aplikacji i który jest przez niego udostępniany (w przypadku bazy danych jest to obiekt `DataSource`). Żądanie przez aplikację dostępu do zasobu uruchamia w serwerze aplikacji całą lawinę działań:
  - Tworzona jest fabryka zasobu, a następnie za jej pomocą tworzony jest zasób transakcyjny, co w konsekwencji prowadzi do utworzenia fizycznego połączenia i obiektu `XAResource`.
  - Po pobraniu zasobu transakcyjnego jest on przypisywany do danej transakcji (`enlistResource`) oraz informowany, że rozpoczyna uczestnictwo w transakcji rozproszonej (metoda start).
  - Dopiero na samym końcu zwracany jest do aplikacji obiekt reprezentujący fizyczne połączenie do zasobu transakcyjnego.
- Następnie nasza aplikacja operuje na zasobie transakcyjnym. Po czym zamyka połączenie. Zazwyczaj wywołanie metody `close` w rzeczywistości nie

> Jak widać całość nie jest taka banalna, jakby się mogło wydać po zapoznaniu się z interfejsem `UserTransacion`.

wywołuje żadnego fizycznego skutku (np. zamknięcia socket'a połączenia do bazy danych). Stanowi tylko informację dla serwera aplikacji, że aplikacja zakończyła swoje działanie na danym połączeniu.

- Po wykonaniu operacji nasza aplikacja znów za pośrednictwem `UserTransaction` zaczyna zatwierdzanie transakcji (wywołanie metody `commit`), co uruchamia następującą sekwencję działań w serwerze aplikacji:
  - Za pośrednictwem menedżera transakcji zasób transakcyjny jest informowany, że w danej transakcji nie będą już wykonywane żadne działania poza zatwierdzeniem lub wycofaniem transakcji (metoda `delistResource` w menedżerze transakcji, która z kolei wywołuje metodę `end` na `XAResource`).
  - Następnie również za pośrednictwem menedżera transakcji realizowane jest dwufazowe zatwierdzanie transakcji, co na rysunku widoczne w postaci operacji `prepare` i `commit` na obiekcie `XAResource`.

Jak widać całość nie jest taka banalna, jakby się mogło wydać po zapoznaniu się z interfejsem `UserTransacion`. Na szczęście prostota interfejsu `UserTransaction` skutecznie ukrywa przed programistą aplikacji złożone interakcje, które zachodzą we wnętrzu serwera aplikacji podczas obsługi transakcji. Niestety czasami rzeczywiste problemy, które nam się przytrafiają w konkretnych systemach korzystających często z bardzo różnych zasobów transakcyjnych, uruchomionych w różnych serwerach aplikacji (a co za tym idzie różniących się szczegółami implementacji menedżera transakcji) czy w złożonych środowiskach produkcyjnych mocno kontrastują z prostym wyglądem interfejsu `UserTransaction`. Aby pokonać te problemy znajomość szczegółów interakcji zachodzących we wnętrzu serwera aplikacji staje się niezbędna.

Na tym chciałbym zakończyć wprowadzenie w transakcje w serwerze aplikacji JEE. W następnym odcinku postaram się przedstawić szczegóły korzystania z baz danych i systemów kolejkowania w kontekście tworzenia systemów transakcyjnych na platformie JEE.

Literatura

[1] Java Transaction API Specification, http://java.sun.com/javaee/technologies/jta/index.jsp

[2] Mark Little, Jon Maron, Greg Pavlik, Java Transaction Processing, Prentice Hall, 2004

[3] Jim Gray, Andreas Reuter, Transacations Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993

[4] Mark Richards, Java Transaction Design Strategies, C4Media 2006

[5] Nuts and Bolts of Transaction Processing, http://www.theserverside.com/tt/articles/article.tss?l=Nuts-and-Bolts-of-Transaction-Processing

Mistrz Programowania: Refaktoryzacja, cz. III

Mariusz Sieraczkiewicz

Dzisiaj kolejny odcinek książki-niespodzianki o refaktoryzacji. Mariusz Sieraczkiewicz zgodził się opublikować w odcinkach na łamach JAVA exPress swoją książkę "Jak całkowicie odmienić sposób programowania używając refaktoryzacji". Jest to pierwsza książka z serii Mistrz Programowania i dotyczy... no tak - refaktoryzacji.

Pierwsza część książki jest dostępna za darmo na stronie http://www.mistrzprogramowania.pl/. Tam także możesz zakupić pełną wersję, bez konieczności czekania 3 miesięcy na kolejną część w JAVA exPress. No i będziesz miał całość w jednym pdf-ie.

W każdym razie zapraszam nawet jeśli możesz czekać. Wspomagajmy samych siebie. Może jutro Ty będziesz chciał coś sprzedać...

A książka Mariusza jest warta swej ceny ;)

Grzegorz Duda

Jednak to nie koniec naszej podróży. Nie poddając się w naszych zmaganiach z refaktoryzacją, będziemy dalej analizować przykład. Zauważymy, że klasa `SearchWordService` zajmuje się dwoma czynnościami:

- przechodzeniem struktury strony HTML,
- realizowaniem algorytmu formułowania tłumaczeń.

Moglibyśmy zostawić kod w tej postaci, jednak zrobimy kolejny krok do przodu — rozdzielimy te dwie odpowiedzialności.

Refaktoryzacja: Zastąpienie metody poruszania się po złożonej strukturze wzorcem Iteratora

Metody `hasNextWord` i `moveToNextWord` to metody umożliwiające nawigację po analizowanej stronie HTML, a dokładniej po słowach wchodzących w składa tłumaczenia. Operacje te odbywają się z użyciem obiektu klasy `BufferedReader`. Mamy więc do czynienia z **sekwencyjnym dostępem do złożonej struktury, przy czym złożoność wewnętrznej struktury jest ukryta**. Jest to nic innego jak intencja stojąca za wzorcem Iteratora.

Refaktoryzacja: Przeniesienie metody i przeniesienie pola

Używając kilku prostych refaktoryzacji wydzielmy wzorzec iteratora. Po pierwsze stwórzmy nową klasę o nazwie `PageIterator` — jej odpowiedzialnością będzie umożliwienie poruszania się po kolejnych wyrazach tłumaczenia. Wszelkie operacje dokonywane na obiekcie klasy `BufferReader` przeniesiemy do klasy `PageIterator`. Użyjemy zatem refaktoryzacji *Przeniesienie metody* dla metody `prepareBufferedReader`, `hasNextWord` oraz `moveToNextWord`. Należy pamiętać o odpowiednim dopasowaniu modyfikatorów dostępu. Efekt tych przekształceń przedstawia poniższy listing.

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PageIterator {

    private BufferedReader bufferedReader = null;

    private BufferedReader prepareBufferedReader()
        throws IOException, MalformedURLException {

        String[] commandParts = command.split(" ");
        String wordToFind = commandParts[1];

        String urlString = "http://www.dict.pl/dict?word=" + wordToFind
                + "&words=&lang=PL";

        return new BufferedReader(new InputStreamReader(
                    new URL(urlString).openStream()));
    }

    public boolean hasNextWord(String word) {
        return (word != null);
    }

    private boolean hasNextLine(String line) {
        return (line != null);
    }

    public String moveToNextWord() {
        try {
            String line = bufferedReader.readLine();
            Pattern pattern = Pattern
                .compile(".*<a href=\"dict\\?words?=(.*)&lang.*");

            while (hasNextLine(line)) {

                Matcher matcher = pattern.matcher(line);

                if (matcher.find()) {
                    String foundWord
                        = matcher.group(matcher.groupCount());

                    return new String(foundWord.getBytes(), "UTF8");
                } else {
                    line = bufferedReader.readLine();
                }
            }
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        }
```

```
        return null;
    }
}
```

Kod ten oczywiście się jeszcze nie kompiluje. Celowo metodę `prepareBufferedReader` pozostawiłem prywatną. Metoda ta zajmuje się inicjacją iteratora. Jeśli ustanowiłbym ją publiczną, użytkownik klasy mógłby ją wywołać. Nam natomiast zależy na tym, aby inicjacja nastąpiła zawsze. Dlatego stworzymy konstruktor, który będzie wywoływał tę metodę. Przyjrzyjmy się również wierszom

```
    String [] commandParts = command . split (" ");
    String wordToFind = commandParts [1];
```

Jest to analiza komendy przychodzącej od użytkownika. Z pewnością nie jest do czynność, którą powinien zajmować się iterator. Dlatego z punktu widzenia klasy `PageIterator` założymy, iż klient tej klasy dostarczy konkretnego słowa bazowego. W tym celu konstruktor będzie miał parametr, który będzie przyjmował wyszukiwane słowo. Przy okazji zmienimy nieco obsługę wyjątków — zamiast wypisywać stos wywołań, będziemy wyrzucać własny wyjątek niekontrolowany. Kod po zmianach wygląda następująco:

```
    public PageIterator(String wordToFind) {
        bufferedReader = prepareBufferedReader(wordToFind);
    }

    private BufferedReader prepareBufferedReader(String wordToFind) {
        BufferedReader result = null;

        String urlString = "http://www.dict.pl/dict?word=" + wordToFind
                + "&words=&lang=PL";

        try {
            result = new BufferedReader(new InputStreamReader(
                        new URL(urlString).openStream()));
        } catch (MalformedURLException e) {
            throw new WebDictionaryException(e);
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        }

        return result;
    }
```

Ponadto w klasie iteratora przyda się jeszcze metoda zwalniająca zasoby — jej zadaniem będzie zamknięcie strumienia `bufferedReader`. Do tej pory zwolnienie zasobów następowało w klasie `SearchWordService`.

```
    public void dispose() {
        try {
            if (bufferedReader != null ) {
                bufferedReader.close();
            }
        } catch (IOException ex) {
            throw new WebDictionaryException(ex);
        }
    }
```

Po powyższych zmianach klasa `SearchWordService` korzystająca z klasy `PageIterator` będzie wyglądać następująco:

```
package pl.bnsit.webdictionary;

import java.util.ArrayList;
import java.util.List;

public class SearchWordService {

    public List<DictionaryWord> search(String wordToFind) {
        List<DictionaryWord> result = new ArrayList<DictionaryWord>();

        PageIterator pageIterator = new PageIterator(wordToFind);

        String currentWord = pageIterator.moveToNextWord();
        int counter = 1;

        while (pageIterator.hasNextWord(currentWord)) {
            DictionaryWord dictionaryWord = new DictionaryWord();
            dictionaryWord.setPolishWord( currentWord );
            currentWord = pageIterator.moveToNextWord();
            dictionaryWord.setEnglishWord( currentWord );
            currentWord = pageIterator.moveToNextWord();

            result.add( dictionaryWord );

            System.out.println( counter + ") "
                    + dictionaryWord.getPolishWord()
                    + " => " + dictionaryWord.getEnglishWord());
            counter = counter + 1;
        }

        pageIterator.dispose();
```

```
        return result;
    }


}
```

Zmiany doprowadziły do znaczącego uproszczenia konstrukcji klasy `SearchWordService`, a co najważniejsze — klasa ta realizuje obecnie tylko i wyłącznie algorytm wyszukiwania tłumaczeń.

Zauważmy również, że z klasy `SearchWordService` zniknęło pole `command`. Wydawało się ono zbędne i tylko niepotrzebnie czyniło klasę stanową tzn. przechowującą pewien stan. Równie dobrze informacja o komendzie mogłaby być przyjmowana jako parametr tej metody:

```
    // ...
    public List<DictionaryWord> search(String command) {
        List<DictionaryWord> result = new ArrayList<DictionaryWord>();

        String[] commandParts = command.split(" ");
    // ...
```

Można się zastanowić, czy do metody `search` powinniśmy przekazywać parametr `command`, a nie słowo do wyszukania. Jeśli tak uczynimy, uzyskamy klasę niezależną od czynników zewnętrznych, którą możemy równie dobrze użyć w innym kontekście — nie tylko w naszej aplikacji konsolowej. A przecież w obiektowości jednym z celów jest uzyskanie możliwości wielokrotnego użycia tego samego kodu. Zatem metoda ta ostatecznie wygląda następująco:

```
    // ...
    public List<DictionaryWord> search(String wordToFind) {
        List<DictionaryWord> result = new ArrayList<DictionaryWord>();

        PageIterator pageIterator = new PageIterator(wordToFind);
    // ...
```

Zaś metoda `searchWord` w klasie `WebDictionary`, która korzysta z klasy `SearchWordService`, wygląda następująco:

```
    private void searchWord(String command) {
        SearchWordService searchWordService
            = new SearchWordService();
```

```
        String[] commandParts = command.split(" ");
        String wordToFind = commandParts[1];
        lastSearchWords = searchWordService.search(wordToFind);
    }
```

Voila! Możemy się teraz nacieszyć stworzonym kodem, który stworzyliśmy. Udało się zrobić kilka istotnych kroków naprzód.