Java exPress
Numer 2/2010(8)
CZASOPISMO DLA DEWELOPERÓW JAVA

CouchDB — bo dane to nie zawsze tabele

Transakcje w systemach Java EE, cz.II

Notatki o testowaniu: WebDriver

Alokacja czasu

>> Architektura aplikacji Flex i Java
>> db4o – obiektowa baza danych
>> BONUS: Refaktoryzacja cz.IV

Patroni:
Adobe
e-point
Lider biznesowych zastosowań technologii Java

JavaExPress www = New Page();

Kolejny numer oddany w ręce czytelników. To już ósmy. Pierwszy pojawił się 2 lata temu. Dzięki ciągłemu zaangażowaniu polskiej społeczności javowej od dwóch lat możemy się cieszyć nowym numerem JAVA exPress co 3 miesiące.

W drugie urodziny udało nam się odświeżyć i przebudować stronę czasopisma http://javaexpress.pl. Mamy nadzieję, że w nowej oprawie graficznej stanie się ona punktem centralnym w dyskusjach na tematy około javowe. Szczególnie te poruszane w artykułach znajdujących się w nowych numerach JAVA exPress.

Wiele dyskusji, czy JAVA exPress powinien być wydawany w formie papierowej zakończy prawdopodobnie informacja o SDJ, który to z formy papierowej przeniósł się na darmowego pdfa. Jednak „tradycyjna" forma jest zbyt kosztowna nawet dla takiego pisma jak SDJ.

Jak zwykle wielkie podziękowania dla Marka Podsiadłego, który wiele swojego czasu poświęca na rozwój stron http://dworld.pl oraz http://javaexpress.pl. Wielkie podziękowania należą się także autorom tekstów oraz tłumaczom - Magdzie i Grześkowi.

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

Do zobaczenia na konferencjach,

Grzegorz Duda

Rozkład jazdy

Redaktor: Grzegorz Duda
Korekta: Agnieszka Wadowska-Duda
Strony www i teksty html: Marek Podsiadły
Grafika: Jakub Sosiński
Tłumaczenie: Magdalena Rudny, Grzegorz Kurka

Relacja z konferencji Javarsovia

Kapituła Konferencji

Wszystko zakończyło się w sobotę w nocy, gdy ostatni uczestnicy konferencji w dobrych humorach, żegnali się, opuszczając SPOINĘ. Była to impreza integracyjna, ostatni gwóźdź programu, dla osób, które za aktywność zgarnęły zaproszenia.

Tak właśnie dobiegła końca 4. edycja największej, bezpłatnej konferencji społeczności javowej w Polsce - Javarsovia 2010, organizowanej przez Warszawski JUG.

I choć się skończyła, to wciąż rozbrzmiewają jej echa w społeczności i dociera do nas dużo pozytywnych opinii na jej temat. Jest również trochę uwag krytycznych, ale to dobrze, bo dzięki nim wiemy co poprawić na przyszłość.

W tym roku ponad 650 uczestników brało udział w konferencji. Jest to o 150 więcej niż rok wcześniej. Wliczając sponsorów i prelegentów, możemy powiedzieć, że nasze szacunki mówiące o 700 osobach okazały się trafione. Dzięki temu udało nam się zaserwować wszystkim obiad, a i koszulek wystarczyło prawie dla wszystkich.

Od strony merytorycznej Javarsovia wypadła rekordowo. Uczestnicy mogli wybierać spośród 24 prezentacji, odbywających się w 4 równoległych sesjach. Jak zwykle nie zabrakło technologicznych nowości, a odwiedzający mieli szansę posłuchać o najnowszych chmurach, szkieletach, fasolkach i nierelacyjnych bazach danych. Na prezentacjach można było poznać nie tylko najnowsze technologie, ale także zgłębić arkana programistycznego rzemiosła. Mogliśmy dowiedzieć się jak pisać lepszy kod, jak dbać o testy oraz poznać zasady kontraktów i współpracy z klientem przy zastosowaniu zwinnego podejścia. Najaktywniejsi mogli liczyć na ciekawe nagrody w postaci unikalnych kubków, książek, licencji na oprogramowanie oraz zaproszeń na SPOINĘ.

W holu można było wymienić doświadczenia z uczestnikami, którzy przyjechali do nas z całej Polski. Gwar jaki panował może świadczyć o tym, że większość ludzi w pełni wykorzystało tą możliwość. Jesteśmy bardzo zadowoleni z osiągniętego efektu.

Wszystkim przybyłym uczestnikom oraz chojnym sponsorom bardzo dziękujemy.

Jeżeli nie byliście na Javarsovii 2010, nic straconego - wszystkie sesje były rejestrowane i ukażą się w serwisie parleys.com. Wystarczy, że będziecie śledzić informacje na www.javarsovia.pl

Liczymy, że konferencja na stałe wpisze się do Waszych “kalendarzy konferencyjnych”, obok innych, świetnych polskich konferencji.

Już na jesieni “warsztatowa” Warsjava, a za rok - jubileuszowa Javarsovia 2011!

db4o - Obiektowa baza danych

Paweł Cegła

Wstęp

Wszędobylski tandem: baza danych + ORM

Myśląc „przechowywanie danych” w kontekście aplikacji komputerowych prawie na pewno myślimy „relacyjna baza danych i mapowanie obiektowo-relacyjne”. Bazy danych, które są dojrzałymi i niezawodnymi produktami zapewniają nam trwałe przechowywanie danych. Natomiast rozwiązania typu ORM umożliwiają nam dostęp do tych danych w postaci użytecznej dla programów napisanych w sposób obiektowy. Podejście to zostało spopularyzowane przez framework Hibernate a następnie ustandaryzowane jako Java Persistence API stanowiące element specyfikacji Java EE 5.

Podejście takie izoluje w dużym stopniu projektanta/programistę od czynności związanych z relacyjnymi bazami danych (takimi jak zakładanie indeksów, rozmiary kolumn itp.) Jednak w przypadku bardziej złożonych aplikacji kod w obiektowym języku programowania zostaje „zaśmiecony” pewnymi elementami relacyjnymi, np. adnotacjami JPA; @Column, @ManyToMany.

Obiektowa baza danych

Alternatywą mogą być obiektowe bazy danych. Obiekty są w nich przechowywane tak, jak używamy ich w programach napisanych zgodnie z paradygmatem obiektowym. Nie trzeba ich mapować na model relacyjny ani modelować takich podstawowych zależności jak dziedziczenie i kompozycja.

Nie mogą one jeszcze zagrozić pozycji relacyjnych baz danych, jednak warto przyjrzeć się możliwościom, jakie oferują. db4o jest bazą danych dostępną na platformy Java i .NET. Możemy jej używać na licencji GPL oraz komercyjnej.

Instalacja

Instalacja sprowadza się do ściągnięcia paczki ze wszystkimi niezbędnymi komponentami. Później należy tylko dodać odpowiedni plik JAR do zmiennej CLASSPATH (np. dla db4o w wersji 7.4 i javy 1.5 - db4o-7.4.121.14026-java5.jar).

Można również dodać db4o jako zależność w projektach zarządzanych przez Mavena, odpowiednie artefakty znajdują się pod adresem http://source.db4o.com/maven.

Podstawowe interfejsy

Db4o

Klasa-fabryka, która służy do rozpoczęcia korzystania z bazy w jednym z dostępnych trybów, jak również zarządzaniem konfiguracją.

ObjectContainer

Interfejs, przy pomocy którego wykonujemy wszystkie podstawowe operacje CRUD na bazie. Każdy obiekt typu ObjectContainer implementuje również interfejs rozszerzony ExtObjectContainer (metoda ObjectContainer.ext()), który zapewnia funkcjonalność dodatkową (np. dostęp do wewnętrznych identyfikatorów obiektów).

Interfejs ten w db4o zapewnia podobną funkcjonalność jak EntityManager w Java Persistence API.

ObjectSet

Obiekty tego typu są zwracane jako wyniki zapytań. Interfejs ten rozszerza kilka innych interfejsów z biblioteki standardowej Javy; Collection, List, Iterator i Iterable. Możemy go zrzutować na taki, który aktualnie potrzebujemy w naszej aplikacji.

> Nie mogą one jeszcze zagrozić pozycji relacyjnych baz danych, jednak warto przyjrzeć się możliwościom, jakie oferują

Tryby działania

Baza danych db4o można używać w dwóch trybach; wbudowanym i klient-serwer. Pierwszy z nich nadaje się do mniejszych aplikacji, bez przetwarzania współbieżnego, natomiast tryb klient-serwer będzie idealny dla aplikacji wielowątkowych, z wieloma transakcjami wykonującymi się równolegle. Czyli m.in. w popularnych aplikacjach internetowych. Działanie bazy w trybie klient-serwer możemy ograniczyć tylko do jednej maszyny wirtualnej a komunikacja pomiędzy klientami a serwerem polegać będzie na wymianie referencji do obiektów Javy (analogia do interfejsów lokalnych w EJB). Oczywiście klienty i serwer mogą znajdować się w różnych maszynach wirtualnych. Wtedy komunikacja odbywać się będzie po sieci TCP/IP (tym razem analogia do interfejsów zdalnych w EJB).

Tryb wbudowany

```
// Listing 1
ObjectContainer oc = Db4o.openFile(PATH_TO_DB4O_FILE);
// operacje na bazie
oc.close();
```

Tryb klient-serwer

Jedna maszyna wirtualna

```
// Listing 2
// serwer
ObjectServer server = Db4o.openServer(PATH_TO_DB4O_FILE, 0);
// klient
ObjectContainer oc = server.openClient();
```

Wiele maszyn wirtualnych

```
// Listing 3
// serwer
ObjectServer server = Db4o.openServer(PATH_TO_DB4O_FILE, PORT);
server.grantAccess(USERNAME, PASSWORD);
// klient
ObjectContainer oc = Db4o.openClient(HOST, PORT, USERNAME, PASSWORD);
```

Podstawowe operacje

```
// ObjectContainer oc - uzyskany w którymkolwiek z trybów działania
```

Zapisywanie

```
oc.store(new Person("Jan", "Kowalski"));
```

Prościej chyba się już nie da...

> Zapytania natywne są głównym i preferowanym sposobem wyszukiwania w db4o

Usuwanie

```
oc.delete(person);
```

Modyfikowanie

```
// person – obiekt pobrany z bazy
// nawet jeśli będzie zawierał te same wartości pól,
// to db4o uzna go jako nowy
oc.store(person);
```

Wyszukiwanie

W relacyjnych bazach używamy języka SQL do wykonywania wszystkich operacji, również wyszukiwania. Baza db4o oferuje trzy rodzaje obiektowego API do wyszukiwania obiektów.

Wyszukiwanie przez przykład

Jest to naprostszy sposób, który jednocześnie obarczony jest największymi ograniczeniami. Jako kryteria wyszukiwania należy podać obiekt z ustawionymi pewnymi właściwościami. Baza znajdzie obiekty tej samej klasy, które będą miały takie same wartości atrybutów podanego przykładu.

```
// Listing 4
Person example = new Person();
example.setLastname("Kowalski");
example.setAge(34);
List<Person> results =
  oc.queryByExample(example);
```

Metoda queryByExample() zwraca obiekt typu ObjectSet, który implementuje m.in. interfejs List. Otrzymamy listę osób, które będą miały na nazwisko „Kowalski" i będą miały 34 lata. W ten sposób bardzo łatwo możemy konstruować proste „zapytania". Ma on jednak kilka niedogodności. Dostęp do pól klasy db4o uzyskuje poprzez refleksję – jest to więc dosyć wolna metoda. Nie można tworzyć rozbudowanych warunków zawierających operatory logiczne, wywołania metod i operatorów innych niż ==. Powyższy przykład odpowiada warunkowi lastname.equals("Kowalski") && age == 34. Niemożliwe jest również wyszukiwanie obiektów, których pola mają mieć wartości domyślne (0 dla int i long, false dla boolean, null dla referencji), gdyż db4o potraktuje to jako brak kryterium dla danego atrybutu. Obiekty, które chcemy użyć jako przykład dla wyszukiwania muszą mieć możliwość pozostawienia pewnych pól niezainicjalizowanych lub ustawienia wartości domyślnych. Jest tak zazwyczaj dla obiektów typu JavaBean.

Zapytania natywne

Zapytania natywne są głównym i preferowanym sposobem wyszukiwania w db4o, gdyż pozwalają na sprawdzanie typów w trakcie kompilacji i łatwą refaktoryzację kodu. Przetwarzanie zapytań natywnych polega na sprawdzeniu wyrażenia dla wszystkich obiektów pewnej klasy i zwróceniu tych, dla których wyrażenie jest prawdziwe. Wyrażenia te kodujemy implementując interfejs Predicate. Parametryzowany jest typem obiektów, które chcemy wyszukać i zawiera jedną metodę match(), która zwraca wartość logiczną true dla obiektów spełniających założone kryteria.

```
// Listing 5
List<Person> people =
  oc.query(new Predicate<Person>() {
  @Override
  public boolean match(
    Person candidate) {
    return candidate
      .getLastname()
      .startsWith("K");
  }
});
```

Powyższe zapytanie natywne zwróci wszystkie osoby w bazie o nazwiskach zaczynających się od „K". Wszystkie zmiany w klasie Person, np. zmianę atrybutu „lastname" z typu String na jakikolwiek inny zostanie od razu zauważone przez kompilator.

> SODA jest niskopoziomowym API do tworzenia zapytań w db4o.

Metoda query() przyjmująca jako pierwszy argument obiekt typu Predicate<T> posiada jeszcze dwie dodatkowe odmiany pozwalające na uporządkowanie listy wyników; jedna przyjmuje komparator typu java.util.Comparator<T> a druga com.db4o.query.QueryComparator<T>.

db4o próbuje optymalizować wyrażenia typu Predicate tak, żeby użyć wewnętrzne indeksy bazy i utworzyć jak najmniej instancji. Prace nad optymalizatorem ciągle trwają a ulepszenia pojawiają się w każdym kolejnym wydaniu bazy.

Zapytania SODA

SODA jest niskopoziomowym API do tworzenia zapytań w db4o. Za jego pomocą możemy dowolnie modelować zapytanie:

```
// Listing 6
Query q = oc.query();
q.constrain(Person.class);
// ograniczamy zapytanie od
// obiektów klasy Person
q.descend(„lastname”);
// przechodzimy do atrybutu
// „lastname”
q.constrain(„K”)
  .startsWith(true);
// i ograniczamy go do wartości
// rozpoczynających się od „K”
```

Metoda query tworzy zapytanie, które zwraca wszystkie obiekty w bazie. Kolejne metody dodają ograniczenia. Już w tym przykładzie widać, że to API nie zapewnia bezpieczeństwa typów; w trakcie kompilacji nie można sprawdzić, czy atrybut „lastname” w ogóle istnieje oraz czy wartość „K” jest poprawnym ograniczeniem dla niego. Sposób ten ma jednak dużą zaletę; dzięki niemu można generować zapytania dynamicznie.

Prosta aplikacja internetowa typu CRUD

Rozpoczynając naukę jakiegokolwiek języka programowania pierwszą napisaną aplikacją jest zazwyczaj „Hello World!” W świecie aplikacji i frameworków webowych odpowiednikiem jest prosta aplikacja typu CRUD (Create, Read, Update, Delete). W niniejszym artykule ograniczymy się tylko do operacji Create i Read; stworzymy prostą listę zadań.

```
// Listing 7
public class ClientFactory {

  private static final
    ObjectServer server;

  static {
    File home = new File(
      System.getProperty(
        “user.home”));
    File db = new File(home,
      “crud.db4o”);
    server = Db4o.openServer(
      db.getPath(), 0);
  }

  public static ObjectContainer
      openClient() {
    return server.openClient();
  }

  public static void close() {
    server.close();
  }
}
```

Klasa ClientFactory zawiera statyczną referencję do serwera db4o a metoda getClient() tworzy nam nową instancję typu ObjectContainer. Będziemy ją pobierać przed wykonaniem operacji na bazie i zamykać po jej zakończeniu.

```
// Listing 8
public class Db4oListener
implements ServletContextListener {
  public void contextInitialized(
    ServletContextEvent sce) {
    try {
      Class.forName(
“eu.pawelcegla.db4ocrud.ClientFacto-
ry”);
    } catch (
      ClassNotFoundException ex) {
      throw new RuntimeException(
“Error starting db4o client factory”,
ex);
    }
  }

  public void contextDestroyed(
```

> Niniejszy artykuł starał się przedstawić jedynie absolutnie niezbędne podstawy używania obiektowej bazy danych db4o

```
      ServletContextEvent sce) {
    ClientFactory.close();
  }
}

// Listing 9
<listener>
  <listener-class>eu.pawelcegla.
db4ocrud.Db4oListener
  </listener-class>
</listener>
```

Db4oListener obsługuje zdarzenia zainicjalizowania i usunięcia kontekstu aplikacji webowej – czyli po prostu uruchomienia i zatrzymaniu aplikacji. Przy starcie wymuszamy załadowanie klasy ClientFactory a jednocześnie wykonania jej bloku static. W nim zaś uruchamiamy serwer db4o, który będzie pracował na pliku crud.db4o umieszczonym w katalogu domowym użytkownika. Wraz z zatrzymaniem aplikacji zamykamy serwer. Żeby zaimplementowany listener obsługiwał te zdarzenia należy dodać podany na listingu 9 fragment do pliku web.xml.

```
// Listing 10
public class Task {

  private final String description;
  private final Date due;

  public Task(String description,
    Date due) {
    this.description =
      description;
    this.due = due;
  }

  @Override
  public String toString() {
    DateFormat df =
      new SimpleDateFormat(
        "yyyy-MM-dd");
    return "(" + df.format(due) +
      ") " + description;
  }

  public static Comparator<Task>
      getComparator() {
   return new Comparator<Task>() {
     public int compare(Task o1,
         Task o2) {
       return o1.due.compareTo(
         o2.due);
     }
   };
  }
}
```

Klasa reprezentująca zadanie do wykonania. Zawiera tekstowy opis i termin zadania. Statyczna metoda zwraca komparator sortujący zadania wg terminu.

Serwlet TaskServlet (listing 11) zawiera praktycznie całą logikę aplikacji. Obsługuje dwa zgłoszenia:

- wyświetlenie listy zadań; metoda GET, URL /task/list,
- dodanie nowego zadania; metoda POST, URL /task/add.

Listing 12 przedstawia niezbędną konfigurację do dodania w pliku web.xml.

Na listingu 13 znajduje się prosta strona JSP służąca do wyświetlenia listy zadań oraz formularza do dodawania nowych.

Źródła aplikacji w postaci spakowanego projektu NetBeans dostępne są pod adresem http://pawelcegla.eu/java-express/Db4o-Crud.zip.

Podsumowanie

Niniejszy artykuł starał się przedstawić jedynie absolutnie niezbędne podstawy używania obiektowej bazy danych db4o (samouczek dostępny razem z bazą ma 171 stron). Autor ma nadzieję, że czytelnicy spróbują jej użyć w swoich rozwiązaniach i sprawdzić, czy może zastąpić znany duet – relacyjną bazę danych i ORM.

Odnośniki

http://db4o.com – główna strona projektu

http://developer.db4o.com – zasoby dla programistów

http://pawelcegla.eu/category/db4o - kilka wpisów o db4o na moim blogu

```
// Listing 11

public class TaskServlet extends HttpServlet {

  protected void processRequest(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
        IOException {
    ObjectContainer oc = ClientFactory.openClient();
    try {
      String pathInfo = request.getPathInfo();

      if ("/add".equals(pathInfo)
          && "post".equalsIgnoreCase(request.getMethod())) {

        String description = request.getParameter("description");
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        Date dueDate = df.parse(request.getParameter("dueDate"));
        oc.store(new Task(description, dueDate));
        oc.commit();
        response.sendRedirect(
          request.getContextPath() + "/task/list");

      } else if ("/list".equals(pathInfo)) {

        List<Task> tasks = oc.query(new Predicate<Task>() {

          @Override
          public boolean match(Task candidate) {
            return true;
          }
        }, Task.getComparator());
        request.setAttribute("tasks", tasks);
        request.getRequestDispatcher("/WEB-INF/jsp/tasks.jsp").
          forward(request, response);
      }
    } catch (ParseException ex) {
      throw new ServletException("Cannot parse due date", ex);
    } finally {
      oc.close();
    }
  }

  @Override
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }

  @Override
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }
}

// Listing 12
<servlet>
  <servlet-name>TaskServlet</servlet-name>
  <servlet-class>eu.pawelcegla.db4ocrud.TaskServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>TaskServlet</servlet-name>
  <url-pattern>/task/*</url-pattern>
</servlet-mapping>
```

```
// Listing 13

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>db4o crud</title>
</head>
<body>
<h1>db4o crud</h1>
<table>
  <tr>
    <th>Task</th>
  </tr>
  <c:forEach var="task" items="${tasks}">
    <tr>
      <td>${task}</td>
    </tr>
  </c:forEach>
</table>
<strong>New task</strong>
<form action="add" method="post">
Description: <input name="description" /> <br />
Due date (yyyy-mm-dd): <input name="dueDate" />
<input type="submit" value="Add" />
</form>
</body>
</html>
```

CouchDB — bo dane to nie zawsze tabele

Paweł Stawicki

CouchDB jest nowym rodzajem bazy danych. Nie jest to baza relacyjna, ani też obiektowa. CouchDB przechowuje dokumenty. Dokument jest czymś w rodzaju znanej z Javy kolekcji Map, ma klucze i wartości. Kluczem zawsze jest String (oczywiście unikalny w ramach jednego dokumentu), co do wartości możliwości jest znacznie więcej, o czym dalej. Kolejną ciekawą cechą CouchDB jest to, że dokumenty te są dostępne przez RESTowy interfejs i protokół HTTP, zapisane w JSONie, znanym JavaScriptowcom formacie zapisu złożonych danych (a więc obiektów). Językiem jaki został użyty do napisania CouchDB jest Erlang, autorzy wybrali go bo świetnie nadaje się do pisania aplikacji wielowątkowych, więc CouchDB jest bardzo dobrze skalowalna na wiele procesorów/rdzeni.

Instalacja

Na stronie http://couchdb.apache.org dostępny jest plik .tar.gz z bazą, dodatkowo w Ubuntu dostępny jest pakiet couchdb. W trakcie instalacji pakietu, utworzony zostaje użytkownik couchdb. Aby odpalić bazę z tym użytkownikiem:

```
sudo -i -u couchdb couchdb -b
```

Wraz z bazą uruchamia się serwer HTTP z narzędziem administracyjnym pomocnym przy wykonywaniu operacji. Otwarcie w przeglądarce strony http://localhost:5984/ powinno skutkować pojawieniem się napisu powitalnego:

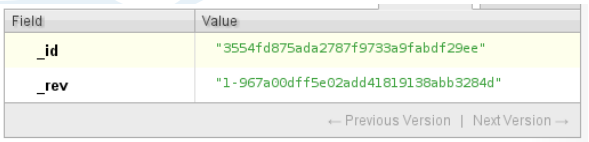
```
{„couchdb”:”Welcome”,
„version”:”0.10.0”}
```

Wspomniane narzędzie znajduje się pod adresem http://localhost:5984/_utils/ Nazywa się Futon. Pozwala na tworzenie i usuwanie baz danych, rekordów (dokumentów) oraz ich modyfikację.

Dokumenty

Futon jest dość intuicyjny, dlatego nie będę opisywał w jaki sposób utworzyć lub skasować bazę danych, ale przejdę od razu do opisu dokumentu. Każdy dokument w CouchDB ma id w polu o nazwie _id. Domyślnie ID są generowane jako UUID, jednak nic nie stoi na przeszkodzie, żeby wpisać inne ID podczas tworzenia dokumentu. Pamiętajmy jednak, że raz nadanego ID nie można później zmienić, więc w polu tym nie powinny być trzymane żadne informacje. Innymi słowy pole ID powinno być używane tylko do jednoznacznego określenia dokumentu, i do niczego innego.

Drugim polem którego nie można edytować po utworzeniu dokumentu jest _rev. Jak nietrudno się domyślić, oznacza ono rewizję. Każda zmiana dokumentu w CouchDB jest zapamiętywana, tzn. możemy zawsze dostać się do dokumentu sprzed zmiany, właśnie przez rewizję.



Rys. 1. Najprostszy możliwy dokument, zawierający tylko pola _id i _rev

Oczywiście do dokumentu możemy dodać też dowolną ilość innych pól. Dodawać można Stringi, liczby, listy, mapy a nawet binarne załączniki. Możliwości jest więc sporo. Wszystko to jest dość proste i intuicyjne w Futonie, trzeba tylko wiedzieć jaki format mają poszczególne typy danych.

typ danych	format (przykład)
String	“artykuł”
liczba	20
tablica	[“kot”, “pies”, “sikorka”]
mapa	{ “imię”: “Paweł”, “nazwisko”: “Stawicki” }

Dokumenty można tworzyć, przeglądać i edytować przez Futona, ale jak wspomniałem na wstępie, CouchDB posiada RESTowy interfejs HTTP. Aby go użyć, wykorzystamy narzędzie curl. (http://curl.haxx.se/) - w Ubuntu dostępne jako pakiet. Za pomocą

> Dokument jest czymś w rodzaju znanej z Javy kolekcji Map, ma klucze i wartości.

curla możemy wysyłać komendy HTTP. Np: curl http://localhost:5984

Powinno pokazać nam znajomy tekst powitalny.

```
{„couchdb”:”Welcome”,
”version”:”0.10.0”}
```

Jeśli mamy bazę danych o nazwie tryme, komenda

```
curl http://localhost:5984/tryme
```

pokaże nam informacje o niej:

```
{„db_name”:”tryme”,”doc_coun-
t”:9,”doc_del_count”:0,”upda-
te_seq”:54,”purge_seq”:0,”com-
pact_running”:false,”disk_size-
”:106585,”instance_start_time-
”:”1265675480690743”,”disk_format_
version”:4}
```

Możemy też sprawdzić, jakie bazy są dostępne:

```
curl http://localhost:5984/_all_dbs

[„tryme”,”loads”,”training”]
```

lub ściągnąć konkretny dokument:

```
curl      http://localhost:5984/tryme
/61fe9c6c226b978f74b76329191806b3

{„_id”:”61fe9c6c226b-
978f74b76329191806b3”,”_rev”:”3-f1f-
a64bf24600ba2ab5ad508f5bc1ab6”,”first-
name”:”Leszek”,”surname”:”Kowalski”,
”type”:”person”,”position”:”develo-
per”,”salary”:3000}
```

To wszystko robimy poleceniem GET, zgodnie z konwencją REST. Możemy też oczywiście użyć polecenia PUT, jak nietrudno się domyślić, w celu utworzenia dokumentu lub (jeśli mamy uprawnienia administratora) bazy danych:

```
curl  -X  PUT  http://localhost:5984/
tryme/id1 -d ,{ „firstname”: „Leszek”,
„surname”: „Gruchała” }’
{„ok”:true,”id”:”id1”,”rev”:”1-12f22
9b1e1754562785b1caa6b52a819”}
```

Utworzyliśmy właśnie rekord z dwoma polami. Jego _id to „id1». Tworząc dokumenty za pomocą Futona, nie musimy podawać ID, Futon potrafi wygenerować je za nas (a właściwie pobrać z CouchDB). W przypadku korzystania z interfejsu HTTP, musimy _id podać explicite. Nie jesteśmy jednak zmuszeni do generowania go własnymi siłami. Możemy pobrać identyfikatory z CouchDB:

```
curl http://localhost:5984/_uuids

{„uuids”:[„e0e8bb72f42f6359f9210fa-
4a0bb0136”]}
```

Możemy też pobrać (wygenerować) więcej ID na raz:

```
curl http://localhost:5984/_uuids?co-
unt=3

{„uuids”:[„db9649f84b61cc57eb880dd34
e8a0fc8”,”db2945c01d421e37af230544e4
b10dcc”,”864c0c5ab116b95dd55222ab8b5
bca61”]}
```

Identyfikatory te są unikalne. Jeśli ktoś inny w tym samym czasie pobierze identyfikator, dostanie inny niż my.

Widok przez map/reduce

JSON pozwala na zapisanie w dokumencie różnych typów danych, w tym także... funkcji. W CouchDB jest specjalny rodzaj dokumentów nazwany „design documents». W takim dokumencie możemy definiować widoki właśnie za pomocą dwóch funkcji na wzór znanego programującym w językach funkcyjnych paradygmatu map/reduce. Do utworzenia widoku wystarczy funkcja map, reduce jest konieczne do wyliczania pewnych sumarycznych wartości dla wielu dokumentów. (Może nie brzmi to zbyt jasno, ale myślę że rozjaśni się po przykładzie). Na początek zajmiemy się właśnie samą funkcją map. Jej jedynym parametrem jest dokument. W funkcji tej wołamy inną funkcję, emit, do której przekazujemy klucz i wartość. I właśnie to co wyemitujemy, znajdzie się w widoku. Myślę, że nie ma co zwlekać i najwyższy czas na przykład. Załóżmy, że mamy bazę

> JSON pozwala na zapisanie w dokumencie różnych typów danych, w tym także... funkcji

danych z trzema prostymi dokumentami:

```
firstname: Paweł
surname: Stawicki
position: developer
salary: 1000

firstname: Leszek
surname: Gruchała
position: developer
salary: 2000

firstname: Ryszard
surname: Stawicki
position: manager
salary: 2500
```

Załóżmy, że chcemy znaleźć wszystkich deweloperów, i zobaczyć ich dane, funkcja map będzie wyglądała tak:

```
function(doc) {
  if (position == „developer”) {
    emit(doc._id, doc);
  }
}
```

Możemy też zrobić to inaczej, o ile nie potrzebujemy żeby kluczem było `_id`. Jeśli kluczem będzie `position`, możemy ograniczyć zwracany wynik właśnie do określonych kluczy. W tym celu jednak nasz widok trzeba zapisać pod określoną nazwą w *design document*. (Jest to analogiczne do tworzenia widoków w bazach SQLowych) W Futonie tworzymy „Temporary view...», z następującą funkcją `map`:

```
function(doc) {
  emit(doc.position, doc);
}
```

Widok ten zapisujemy („Save As...»), podając nazwę *design documentu* i widoku. Załóżmy, że zapisaliśmy ten widok w design documencie o nazwie `it`, zaś sam widok nazwany został `by-position`. Dostać się do niego możemy przez następujący URL: *http://localhost:5984/tryme/_design/it/_view/by-position*

```
{„total_rows”:3,
 „offset”:0,
 „rows”:[
                    {„id”:”61fe9c6c226b-
978f74b76329191806b3”,
   „key”:”developer”,
   „value”:
                    {„_id”:”61fe9c6c226b-
978f74b76329191806b3”,
     „_rev”:”4-7d0ecd6ca4c65ee368fc88
e51fa6a3b5”,
     „firstname”:”Leszek”,
     „surname”:”Grucha\u0142a”,
     „type”:”person”,
     „position”:”developer”,
     „salary”:2000
    }
  },
                  {„id”:”eb3873f48bb581d-
f13762324b8ec0313”,
   „key”:”developer”,
   „value”:
                  {„_id”:”eb3873f48bb581d-
f13762324b8ec0313”,
     „_rev”:”6-b2c4d3dd7da321d286b166
62ecb1f082”,
     „firstname”:”Pawe\u0142”,
     „surname”:”Stawicki”,
     „type”:”person”,
     „position”:”developer”,
     „salary”:1000
    }
  },
  {„id”:”1”,
   „key”:”manager”,
   „value”:
   {„_id”:”1”,
     „_rev”:”9- e3a554d649e755e1bed-
0d10537971e9b”,
     „surname”:”Stawicki”,
     „firstname”:”Ryszard”,
     „type”:”person”,
     „position”:”manager”,
     „salary”:2500
    }
  }
 ]}
```

Teraz mamy stanowisko jako klucz. Jeśli chcemy wyświetlić samych deweloperów, dostępni będą pod adresem *http://localhost:5984/tryme/_design/it/_view/by-position?key=”developer”*

```
{„total_rows”:3,
 „offset”:0,
 „rows”:[
                    {„id”:”61fe9c6c226b-
978f74b76329191806b3”,
   „key”:”developer”,
   „value”:
                    {„_id”:”61fe9c6c226b-
```

> A cóż to za parametr `group_level=1`?

```
978f74b76329191806b3",
     „_rev":"4-7d0ecd6ca4c65ee368fc88
e51fa6a3b5",
     „firstname":"Leszek",
     „surname":"Grucha\u0142a",
     „type":"person",
     „position":"developer",
     „salary":2000
    }
  },
              {„id":"eb3873f48bb581d-
f13762324b8ec0313",
   „key":"developer",
   „value":
             {„_id":"eb3873f48bb581d-
f13762324b8ec0313",
     „_rev":"6-b2c4d3dd7da321d286b166
62ecb1f082",
     „firstname":"Pawe\u0142",
     „surname":"Stawicki",
     „type":"person",
     „position":"developer",
     „salary":1000
    }
  }
 ]}
```

Powiedzmy, że w bazie danych mamy wielu developerów, managerów, administratorów i innych pracowników IT oraz ich wynagrodzenia. Aby poznać łączne wynagrodzenie dla każdej grupy zawodowej musimy wykorzystać funkcję `reduce`. Funkcja ta przyjmuje 3 parametry: klucze, wartości i parametr typu boolean `rereduce`. Z `reduce` zwracamy wynik, który powinien być skalarem, najczęściej liczbą. Czyli dla określonych kluczy, zwracamy liczbę, wyliczoną na podstawie wartości związanych z tymi kluczami. Klucze i wartości które trafiają do `reduce`, to te same klucze i wartości, które wyemitowaliśmy z map. Jest jeszcze trzeci parametr, `rereduce`, ale o tym później.

Nasza funkcja `reduce` do badania wynagrodzeń w poszczególnych grupach, powinna wyglądać tak:

```
function(keys, values, rereduce) {
  var salaries = 0;
  for(i = 0; i < values.length; i++) {
    salaries += values[i].salary;
  }
  return salaries;
}
```

Za pomocą takiej funkcji `reduce` możemy też obliczyć łączne wynagrodzenia wszystkich pracowników. Jeśli dodamy ją do widoku `it`, i zapiszemy go pod nową nazwą `salaries`, będzie można się do niego dostać przez url:

http://localhost:5984/tryme/_design/it/_view/salaries?group_level=1

```
{„rows":[
 {„key":"developer", „value":3000},
 {„key":"manager", „value":2500}
]}
```

A cóż to za parametr `group_level=1`? Oznacza on, na jakim poziomie grupować klucze. W tym wypadku mamy tylko jeden poziom kluczy, dodamy zatem naszym pracownikom działu IT jeszcze projekty. Dodajmy pole `project` z wartością dla Pawła „travel», dla Leszka „houses", i dla Ryszarda też „houses". Dodajmy jeszcze jednego pracownika do projektu „houses":

```
firstname: Maciej
surname: Majewski
position: developer
project: houses
salary: 1800
```

Teraz zmodyfikujemy nieco funkcję `map` tak, żeby emitować klucz będący tablicą:

```
emit([doc.position,     doc.project],
doc);
```

Klucz jest teraz dwuelementową tablicą, ma dwa poziomy. I możemy go według tych poziomów grupować. Najlepiej zilustrować to na przykładzie. Wywołanie widoku z `group_level=2` daje następujący wynik:

```
{„rows":[
   {„key":[„developer","houses"],"va-
lue":3800},
   {„key":[„developer","travel"],"va-
lue":1000},
    {„key":[„manager","houses"],"va-
lue":2500}
]}
```

A to wynik dla group_level=1:

> Widać zatem, że wynik zwracany z `reduce` odnosi się do grupy kluczy, i to zgrupowanych na różnych poziomach.

```
{„rows”:[
 {„key”:[„developer”],”value”:4800},
 {„key”:[„manager”],”value”:2500}
]}
```

A co się stanie jak pominiemy parametr `group_level`? Dostaniemy wynik zredukowany do zerowego poziomu, czyli łączne wynagrodzenie wszystkich pracowników.

```
{„rows”:[
 {„key”:null,”value”:7300}
]}
```

Widać zatem, że wynik zwracany z `reduce` odnosi się do grupy kluczy, i to zgrupowanych na różnych poziomach. A co robi parametr `rereduce`? Tak naprawdę, nasza funkcja `reduce` zawiera błąd i nie zadziałałaby dla dużej liczby dokumentów. To z tego powodu, że może być ona wywoływana rekurencyjnie, to znaczy najpierw przekazywana jest do niej jakaś grupa dokumentów, a potem jej wynik znowu przekazywany jest do `reduce`. `Rereduce` ma wartość `false` w pierwszym wypadku, a w kolejnych wywołaniach `true`. Teoretycznie mogłoby się tak stać w przypadku tutaj rozpatrywanym kiedy klucze są zgrupowane (`group_level=1`). Kolejne wywołania `reduce` mogłyby wyglądać następująco:

1. reduce (['developer'], <cały dokument Paweł>, false) => 1000

2. reduce (['developer'], <cały dokument Leszek>, false) => 2000

3. reduce (['developer'], [1000, 2000], **true**) => 3000

W pierwszym i drugim przypadku do `reduce` jako parametr `values` trafiają dokumenty, ale w trzecim przypadku trafiają tam wyniki wcześniejszych wywołań. Nie są to dokumenty ale tablica liczb, więc trzeba ją inaczej obsłużyć. Poprawna funkcja `reduce` powinna wyglądać tak:

```
function(keys, values, rereduce) {
  var salaries = 0;
  if (rereduce) {
    salaries += sum(values);
  } else {
    for(i = 0; i < values.length; i++)
{
      salaries +=
        values[i].salary;
    }
  }
  return salaries;
}
```

Oprócz `group_level`, do dyspozycji mamy jeszcze między innymi parametry:

- `startkey` - klucz pierwszego dokumentu w widoku
- `endkey` - analogicznie, klucz ostatniego dokumentu. Przy pomocy tych dwóch parametrów możemy ograniczyć liczbę zwracanych w widoku danych.
- `key` - za pomocą tego parametru pobierzemy tylko jeden dokument, o podanym kluczu
- `group=true` - w ten sposób możemy ustawić `group_level` na maksimum
- `revs_info=true` - wyświetla informacje o rewizjach dokumentu. Ma zastosowanie tylko kiedy pobieramy pojedynczy dokument (parametr `key`)

Design documents

Design documents służą do czegoś więcej niż tylko przechowywanie widoków map/reduce. Dzięki design documents możemy pobierane z bazy obiekty JSON „otoczyć” odpowiednim HTMLem tak, żeby zamiast kodu JavaScript zobaczyć pod adresem obiektu miłą dla oka stronę. Dobry przykład można znaleźć pod adresem http://caprazzi.net:5984/fortytwo/_design/fortytwo/index.html Jest to w pełni funkcjonalna aplikacja webowa, w której jedynym serwerem jest serwer CouchDB. W design documencie, oprócz widoków, mogą być funkcje walidujące zapisywane dokumenty, dokonujące dodatkowych operacji na do-

> W design documencie
> możemy zdefiniować funkcję walidującą.

kumencie zapisywanym do bazy, formatujące dokument do miłego dla oka HTMLa. Zajmiemy się nimi wszystkimi po kolei.

Walidacja

W design documencie możemy zdefiniować funkcję walidującą. Nie jest ona konieczna. Jeśli jej nie będzie, każdy dokument będzie uznawany za poprawny. Design documentów może być więcej, a w każdym z nich może być funkcja walidująca. Jeśli tak jest, dokument musi być zaakceptowany przez wszystkie z nich. Funkcja taka ma nazwę `validate_doc_update`, i następującą deklarację:

```
function(newDoc, oldDoc, userCtx) {}
```

Funkcja ta nie zwraca żadnej wartości. Jeśli uznamy dokument za niepoprawny, powinniśmy rzucić wyjątek:

```
throw({forbidden : „Dokument niepoprawny”});
```

Jeśli dokument próbuje zapisać użytkownik, któremu nie chcemy na to pozwolić ze względu na autoryzację, możemy też rzucić:

```
throw({unauthorized : „Dokument niepoprawny”});
```

W tym drugim przypadku baza poprosi użytkownika o nazwę i hasło.

Zajmijmy się teraz parametrami funkcji. Pierwszych dwóch nietrudno się domyślić, jest to nowa i stara wersja dokumentu. Jeśli funkcja wywoływana jest dla nowego dokumentu, którego jeszcze nie ma w bazie, `oldDoc` przyjmuje wartość null. `userCtx` to obiekt zawierający dane dotyczące użytkownika, np. `userCtx.name`.

„Dekorowanie» dokumentu

Czasem dobrze by było przesłać nie tyle dokument w formacie JSON, ale np. część dokumentu, w XMLu, CSV lub innym formacie tekstowym. CouchDB i to nam umożliwia, udostępniając funkcję `show`, dzięki której możemy określić co ma być zwracane zamiast „surowego» JSONa. Parametry funkcji to dokument i request:

```
function(doc, req) {}
```

Funkcja ta może zwracać zwykły tekst, wtedy jest on ładowany jako treść odpowiedzi, np:

```
function(doc, req) {
  return '<h1>' + doc.name +
    '</h1><p>' + doc.bio + '</p>';
}
```

Możemy też określić inne parametry, np. nagłówki

```
function(doc, req) {
  return {
    body : '<h1>' + doc.title + '</
h1>',
    headers : {
      „Content-Type» : „application/
xml»,
      „X-My-Own-Header»: „you can set
your own headers»
    }
  }
}
```

Z założenia funkcja show dla takich samych parametrów, zwraca zawsze takie same wyniki, dzięki czemu można je cache'ować, co może znacznie przyspieszyć działanie bazy.

W jednym design documencie może być więcej funkcji show, różniących się nazwami. (Rys. 2).

Pobrać dokument „udekorowany” przez taką funkcję można za pomocą następującego URLa:

```
http://localhost:5984/tryme/_de-
sign/it/_show/show_id/133fec564f0b-
68f30a7b7d63fe8235d2
```

Gdzie _design/it określa design document, a _show/show_id funkcję o nazwie show_id, zaś 133fec564f0b68f30a7b7d63fe8235d2 to ID dokumentu, który chcemy sobie obejrzeć.

Ciekawym rozwiązaniem jest to, że funkcję

> CouchDB implementuje też mechanizm template'ów, i można ich używać w funkcjach `show`.

show można także wywołać bez ID dokumentu. Wtedy na miejsce argumentu `doc` trafia do funkcji null. Drugim parametrem funkcji show jest request, z niego możemy wyciągnąć parametry przekazane w requeście HTTP. Robi się to w sposób najprostszy z możliwych:

```
req.param1
```

Wyciągnie nam parametr o nazwie `param1`.

CouchDB implementuje też mechanizm template'ów, i można ich używać w funkcjach `show`. Możemy trzymać w osobnych plikach wzór HTMLa jakim chcielibyśmy otoczyć dokument i użyć ich w naszej funkcji. Dokładny opis korzystania z tego mechanizmu wykracza jednak poza zakres tego artykułu, odsyłam do książki online (*http://books.couchdb.org/relax/*)

„Dekorowanie» widoku

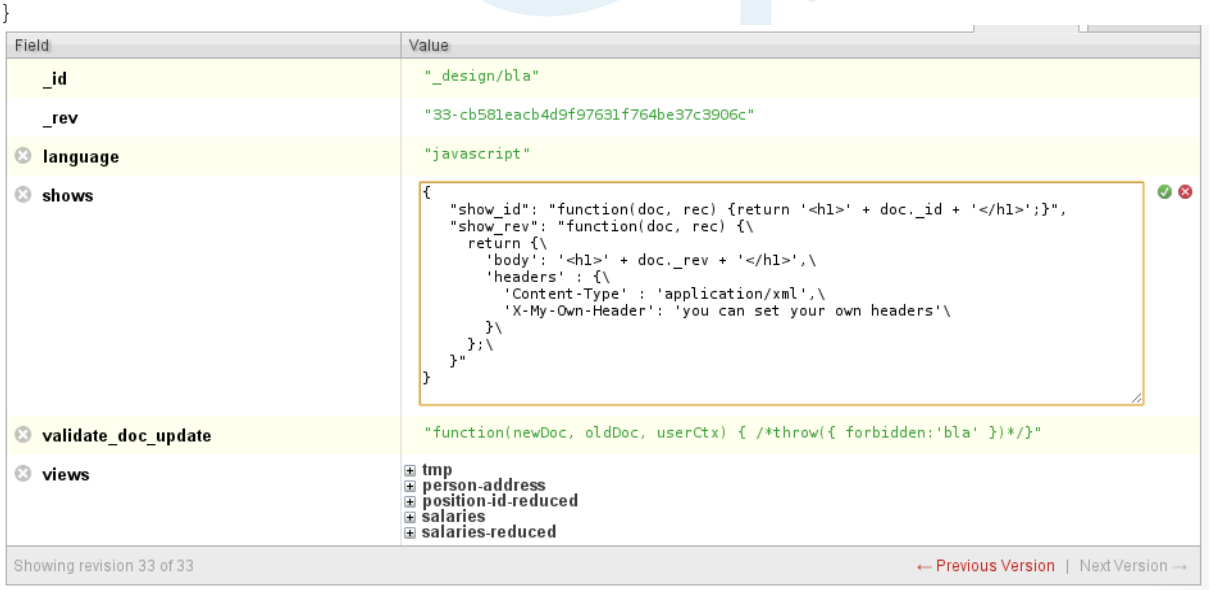
Innym rodzajem funkcji w CouchDB jest funkcja `list`. Pełni ona podobne zadanie co `show`, ale nie dla pojedynczego dokumentu, ale całego widoku. Przyjmuje dwa parametry:

```
function(head, req) {
  while(getRow()) {
    send(...)
  }
}
```

`head` zawiera informacje potrzebne do page-'ingu, czyli ile dokumentów (wierszy) pobrać oraz od którego zacząć.

`req` to natomiast wszelkie dane o requeście i nie tylko. Zawiera następujące parametry:

- `info` - informacje o bazie (te same co pod URLem bazy danych, np. http://localhost:5984/tryme/)
- `verb` - GET lub POST
- `query` - parametry dołączone do wywołania
- `path` - ścieżka wywołania rozbita na części (separator: „/»)
- `headers`
- `cookie`
- `body` - przydatne jeśli wysłane było POSTem
- `form` - to co w body, ale w formacie JSON
- `userCtx` - informacje o użytkowniku, takie same jak przesłane do funkcji `validate_doc_update`



Rys. 2. Wiele funkcji show w jednym design documencie.

> Kolejną zaletą CouchDB,
> o której dotąd nie wspominałem,
> jest łatwość replikacji.

`getRow()` pobiera następny dokument z dekorowanego widoku (null po pobraniu wszystkich dokumentów), a `send()` wysyła HTML do przeglądarki.

Na końcu funkcja list zwraca String, który również jest wysyłany do przeglądarki (może to być więc np. footer).

Aby udekorować sobie widok funkcją list, wysyła się żądanie pod następujący URL:

http://localhost:5984/tryme/_design/it/_list/basic/info

Gdzie basic to nazwa funkcji list, a info to nazwa widoku.

Bezpieczeństwo

Bezpośrednio po instalacji, z bazą danych każdy może zrobić wszystko. Pod warunkiem, że pracuje na tej samej maszynie na której stoi baza. Możemy jednak udostępnić ją światu, jednocześnie tworząc użytkowników z prawami administratorów. Jeśli jest choć jeden admin, inni użytkownicy mają już znacznie ograniczone uprawnienia. Aby utworzyć konto administratora, należy wyedytować plik `/etc/couchdb/local.ini`, i w sekcji `[admins]` dodać:

```
franek = haslo
```

Po uruchomieniu bazy, hasło zostanie zahashowane. Administratora można też dodać odpowiednim poleceniem HTTP:

```
curl -X PUT http://localhost:5984/_config/admins/franek -d '"haslo"'
```

Jeśli administrator jest już dodany, następnych może dodawać tylko on.

Jeśli chcemy wykonać jakąś operację jako administrator, nazwa użytkownika i hasło muszą być dodane do URLa:

```
curl http://franek:haslo@localhost:5984/tryme
```

Oczywiście nie jest to bezpieczne. Nazwa użytkownika i hasło, niezaszyfrowane są przekazywane przez sieć do serwera. Wystarczy podsłuchać...

Jeśli chcielibyśmy aby dostęp do naszej bazy danych mieli także użytkownicy z innych komputerów (nie tylko localhosta), trzeba jeszcze wyedytować `bind_address` w pliku `/etc/couchdb/local.ini` Jeśli ustawimy tam wartość `0.0.0.0`, do bazy będzie można się podłączyć z dowolnego komputera.

Obecnie rozwiązania bezpieczeństwa są intensywnie rozwijane w projekcie CouchDB, należy się więc spodziewać że mechanizmy te zostaną znacznie ulepszone w niedalekiej przyszłości. Prawdopodobnie zwiększona zostanie elastyczność i będzie można ustawić jaki użytkownik do czego ma dostęp i co może z danym zasobem zrobić.

Replikacja

Kolejną zaletą CouchDB, o której dotąd nie wspominałem, jest łatwość replikacji. Za pomocą Futona można skopiować zawartość bazy danych do innej bazy praktycznie bez żadnego przygotowania. Replikację można robić zarówno z bazy lokalnej na zdalną, jak i ze zdalnej na lokalną. Nie ma tu żadnych ograniczeń (Rys. 3).

Replikacja jest jednokierunkowa. To znaczy, że jeśli replikujemy z bazy A do bazy B, to wszystkie dokumenty które mamy w bazie A zostaną skopiowane do B, ale jeśli w B mieliśmy jakieś dokumenty których nie ma w bazie A, to nie zostaną one skopiowane do A, ani skasowane. Wynika z tego, że nie zawsze po replikacji obie bazy będą identyczne. Aby tak było, musielibyśmy zrobić dwie replikacje: A -> B i B -> A. Co ciekawe, jeśli jakiś dokument był na bazie źródłowej, ale został skasowany, przy replikacji zostanie on także skasowany na bazie docelowej.
Jeśli na bazie docelowej jest ustawione konto administratora, warto dodać jego nazwę/hasło do URLa. Inaczej dokumenty do utwo-

> Jedną z ciekawszych funkcji replikacji w CouchDB jest „replikacja ciągła”.

rzenia których uprawnienia ma tylko administrator (np. *design document*) nie zostaną skopiowane. W takim przypadku URL powinien wyglądać tak: *http://franek:haslo@remote.com:5984/tryme*

Replikację możemy przeprowadzać także lokalnie, tzn. replikować bazę danych na inną bazę danych na tym samym serwerze, tworząc w ten sposób kopię zapasową. Dzięki temu bez obaw można testować „ryzykowne» operacje, bo gdyby coś poszło nie tak, zawsze możemy łatwo i szybko przywrócić stan poprzedni.

Jedną z ciekawszych funkcji replikacji w CouchDB jest „replikacja ciągła” (*continous replication*). Po skopiowaniu danych CouchDB nadal nasłuchuje na zmiany na bazie źródłowej. Kiedy takowe się pojawiają, wysyła je na bazę docelową. Nie wysyła jednak od razu, tylko w najbardziej dogodnym momencie. Dlatego nie można nigdy zakładać że jeśli mamy ciągłą replikację, to bazy zawsze będą takie same. Tym niemniej jeśli jesteśmy w stanie zaakceptować niewielkie opóźnienia w replikacji, i niewielkie różnice w danych, możliwość ta może być bardzo przydatna. Aby włączyć ciągłą replikację:

```
curl -X POST http://127.0.0.1:5984/_
replicate -d ,{„source”:”tryme”,
„target”:”http://remote.com:5984/
tryme”, „continuous”:true}’
```

CouchDB a Java

Do CouchDB powstało kilka javowych bibliotek. Jedną z nich jest CouchDB4J, jednak za jej pomocą nie udało mi się pobrać dokumentów z widoku. Być może moje dokumenty były specyficzne, jednak biblioteka nie jest rozwijana od czerwca 2009, więc prawdopodobne jest że nie będzie działać z nowymi wersjami CouchDB :(

JRelax

Druga biblioteka jaką wypróbowałem to JRelax. Tutaj bolączką jest brak dokumentacji, ale podglądając źródła i dysponując IDE z podpowiadaniem da się to przeżyć. Tak wygląda pobranie wszystkich baz za pomocą tej biblioteki:

```
DefaultResourceManager manager = new
DefaultResourceManager(„http://lo-
calhost:5984”);
List<String> dbs =
  manager.listDatabases();
for(String dbName : dbs) {
    System.out.println(dbName);
}
```

Dość zaskakujący był dla mnie brak metody typu `listDocuments(String dbName)`, aby pobrać wszystkie dokumenty (ich id) z danej bazy, trzeba użyć widoku tymczasowego:

```
ViewResult<String, Object> res =
  manager.executeTemporaryView(
    „tryme”, „function(doc) {
    emit(doc._id, null)}”, null,
    String.class, Object.class);

for(ViewResult.ViewResultRow<String,
Object> row :
  res.getResultRows()) {
    System.out.println(
      row.getKey());
}
```

Parametry metody `executeTemporaryView` to kolejno: baza danych, funkcja `map`, funkcja `reduce`, klasa klucza widoku, klasa wartości widoku. Typ klucza i wartości widoku musi być też podany jako typ generyczny `ViewResult<K, V>`, będącego rezultatem wywołania tej metody. Można tam wstawić odpowiednio przygotowane POJO, czy to zgodne z konwencją javowych beanów, czy też z annotacjami. Do mapowania JRelax uży-

Rys. 3. Replikacja za pomocą narzędzia „Futon”

> Pierwszy, dość oczywisty przykład zastosowania, to dane, które nie zawsze są takie same.

wa Jacksona (parser JSONa), i na jego stronę odsyłam po szczegóły, na jakie klasy potrafi on JSONa przemapować (*http://wiki.fasterxml.com/JacksonDataBinding*, *http://wiki.fasterxml.com/JacksonAnnotations*).

Tak natomiast można zapisać dokument do bazy:

```
Document doc = new Document(„tryme”,
„2”, „{\”firstname\”: \”pawel\”}”);
manager.saveDocument(doc);
```

Nie znalazłem niestety możliwości podania nazwy i hasła użytkownika, więc JRelax można używać tylko do „otwartych» baz, lub tylko do odczytu. Nie znalazłem też możliwości wygenerowania id z bazy, ani przekazania parametrów `startKey`, `endKey` do wywołania widoku. Wygląda więc na to, że możliwości JRelax są dość ograniczone.

jcouchdb

Kolejną Javową biblioteką do CouchDB jest jcouchdb. Podobnie jak w przypadku JRelax, jest ona umieszczona w repozytorium Mavena, które wystarczy dodać do poma, aby Maven za nas załatwił wszystkie zależności. Pobieranie baz danych jest proste i wygodne:

```
Server srv =
  new ServerImpl(„localhost”);
List<String> databases =
  srv.listDatabases();
for(String dbName : databases) {
    System.out.println(dbName);
}
```

Równie proste jest pobranie id wszystkich dokumentów w bazie:

```
Database db = new Database(
  „localhost”, „tryme”);
ViewResult<Map> res =
  db.listDocuments(null, null);
for(ValueRow<Map> doc :
  res.getRows()) {
    System.out.println(
      doc.getId());
}
```

Metoda `listDocuments` przyjmuje dwa parametry, pierwszy z nich to obiekt `Options`, w którym możemy ustawić parametry takie jak `group`, `startKey`, `endKey` itd. Drugi parametr to `JSONParser`, co oznacza, że możemy przekazać własny parser, który zmapuje nam JSONa na nasze obiekty, a nie tylko na mapę parametrów. Właśnie, używając domyślnego parsera możemy w dość prosty sposób odczytać dokument jako mapę:

```
Map<String, Object> doc =
  db.getDocument(Map.class, „1”);
for(String paramName :
  doc.keySet()) {
    System.out.println(paramName +
    „: „ + doc.get(paramName));
}
```

Równie wygodnie można pobrać widok:

```
ViewResult<Map> viewRes =
  db.queryView(„it/by-position”, Map.
class, null, null);
for(ValueRow<Map> row :
  viewRes.getRows()) {
    System.out.println(
      row.getKey() + „: „ +
      row.getValue());
}
```

Muszę przyznać, że właśnie ta biblioteka przypadła mi osobiście najbardziej do gustu.

Dlaczego CouchDB?

Rozważmy, dlaczego ktoś miałby używać nierelacyjnej bazy danych, takiej jak CouchDB, zamiast tradycyjnej, relacyjnej SQLowej. Pierwszy, dość oczywisty przykład zastosowania, to dane, które nie zawsze są takie same. Np. jeśli mamy bazę danych pacjentów, każdy ma przypisanego lekarza, ale tylko niektórzy położną, niektórzy mają przepisane leki, inni dietę, inni określone zabiegi itd. Oczywiście w bazie relacyjnej także dałoby się to osiągnąć, ale byłoby to bardziej skomplikowane, trzebaby użyć wielu tabel, obsłużyć powiązania pomiędzy nimi itd.

Inny argument przemawiający za CouchDB to elastyczność. Zwłaszcza na etapie developmentu schemat bazy danych często się zmie-

> Kolejnym argumentem przemawiającym na korzyść CouchDB jest prosty protokół HTTP/REST

nia. W CouchDB wprowadzanie takich zmian jest znacznie łatwiejsze - w każdej chwili możemy dopisać coś do dokumentu, nie trzeba martwić się o relacje pomiędzy tabelami.

Kolejnym argumentem przemawiającym na korzyść CouchDB jest prosty protokół HTTP/REST. Nie trzeba korzystać z żadnych sterowników (*driverów*) JDBC, wystarczy request HTTP żeby wykonywać operacje na bazie.

Przydatna może się także okazać funkcja wersjonowania. Przypadkowe usunięcie lub nadpisanie danych z dokumentu nie jest problemem, wystarczy przywrócić wcześniejszą wersję.

CouchDB ma też inne zalety, o mniejszym znaczeniu dla programistów pracujących z bazą danych, lecz ułatwiające życie administratorom. Jest to skalowalność i łatwa replikacja. Wszystko wskazuje na to, że kolejne procesory będą miały coraz więcej rdzeni. CouchDB nie powinna mieć problemu z wykorzystaniem dużej ich ilości. Możliwość ciągłej replikacji natomiast ułatwia zbudowanie klastra.

CouchDB, a także inne nierelacyjne bazy danych, stają się coraz bardziej popularne. Czas pokaże, czy zagoszczą na dobre na naszych serwerach.

O autorze

Paweł Stawicki zawodowo zajmuje się Javą od 2003 roku. Od tego też czasu związany ze szczecińską firmą NCDC. Oprócz Javy interesuje się Javą FX, Scalą, i różnymi innymi ciekawymi technologiami ;) Jest co-leaderem Szczecińskiego JUGa (http://szczecin.jug.pl) i autorem bloga http://pawelstawicki.blogspot.com. Współorganizator szczecińskiej konferencji java4people w 2009 roku. Z zamiłowania żeglarz.

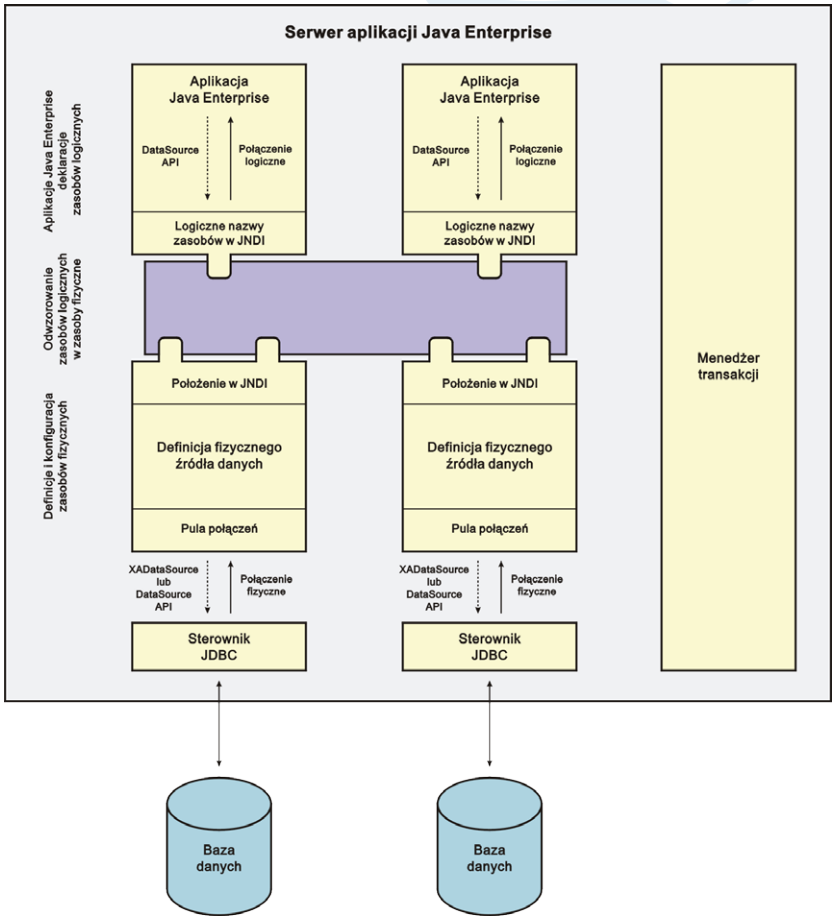
Transakcje w systemach Java EE: Korzystanie z baz danych

Jarosław Błąd

Dostęp do baz danych w środowisku serwera aplikacji

Korzystanie z baz danych w aplikacjach Java Enterprise różni się nieco od korzystania z nich w aplikacjach konsolowych, czy desktopowych. Różnice te polegają przede wszystkim na odmiennym sposobie uzyskiwania połączenia do baz danych. W standardowych aplikacjach pisanych w środowisku Java Standard Edition korzystamy bezpośrednio ze sterownika JDBC, z którego uzyskujemy fizyczne połączenie do bazy danych. W aplikacjach Java Enterprise dostęp do bazy danych jest o wiele bardziej złożony. Przyjrzyjmy mu się dokładniej.

Na rysunku 1 zostało przedstawione środowisko jakie serwer aplikacji tworzy dla aplikacji korzystających z baz danych.



Rysunek 1 Dostęp do baz danych w środowisku tworzonym przez serwer aplikacji

Środowisko to składa się z następujących elementów:

- Deskryptora aplikacji, w którym twórca aplikacji definiuje logiczne zasoby, z których będzie korzystał (np. bazy danych). Dla aplikacji internetowych zasoby te są definiowane za pomocą elementów `<resource-ref>` w pliku `web.xml`. Aplikacje mogą się odwoływać jedynie do zasobów, które zostały zdefiniowane w deskryptorze. Z punktu widzenia aplikacji zasoby te są widoczne w drzewie JNDI pod zdefiniowanymi w deskryptorze nazwami.
- Definicji fizycznych zasobów (np. baz danych) wraz z ich parametrami konfiguracyjnymi takimi jak: adres serwera bazy danych, nazwa bazy danych, użytkownik, hasła. Zasoby te mogą być dostępne dla wszystkich aplikacji zainstalowanych na serwerze. Każdy zasób jest widoczny pod odpowiednią nazwą w drzewie JNDI. Ze względów wydajnościowych obiekty reprezentujące zasoby fizyczne utrzymują pulę otwartych połączeń do bazy danych. Sposób konfiguracji zasobu fizycznego jest specyficzny dla danego serwera aplikacji.
- Odwzorowania zdefiniowanych w aplikacjach zasobów logicznych na zasoby fizyczne zdefiniowane w konfiguracji serwera. Odwzorowanie to jest realizowane za pomocą deskryptorów specyficznych dla danego serwera aplikacji i jest ustanawiane w momencie instalacji aplikacji na serwerze.
- Menedżera transakcji, który w imieniu aplikacji potrafi zarządzać transakcjami

> W aplikacjach Java Enterprise
> dostęp do bazy danych jest o wiele bardziej złożony.

rozproszonymi, w których może uczestniczyć wiele zasobów transakcyjnych (np. baz danych).

Taki sposób organizacji dostępu do bazy danych ma następujące konsekwencje:

- Aplikacja nie zawiera konfiguracji dostępu do bazy danych i nie zarządza połączeniem do bazy danych.
- Aplikacja operuje na połączeniu logicznym udostępnianym przez serwer aplikacji.
- Konfiguracja parametrów fizycznego połączenia do bazy danych znajduje się na poziomie serwera aplikacyjnego.
- Wiele aplikacji może korzystać z tego samego fizycznego zasobu. Wiąże się to również ze współdzieleniem puli połączeń.

Wszystko to powoduje, że w serwerze aplikacyjnym mamy bardzo elastyczną strukturę dostępu do bazy danych, niestety okupioną dużą złożonością jej konfiguracji.

Przyjrzyjmy się jak taka konfiguracja może wyglądać w praktyce na przykładzie prostej aplikacji internetowej, która chce skorzystać z bazy danych.

Tworząc naszą aplikację musimy zadeklarować w jej deskryptorze (plik `web.xml`), że będziemy korzystać z bazy danych i że obiekt umożliwiający dostęp do bazy danych (`javax.sql.DataSource`) powinien się znajdować w drzewie JNDI pod określoną nazwą (tutaj `jdbc/SampleDS`):

```
<resource-ref>
  <description>Sample DS
  </description>
  <res-ref-name>jdbc/SampleDS
  </res-ref-name>
  <res-type>javax.sql.DataSource
  </res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

W serwerze aplikacji musimy zdefiniować zasób fizyczny wraz z konfiguracją parametrów połączenia do bazy danych oraz konfiguracją parametrów puli połączeń (np. wielkość puli). W każdym serwerze konfigurowanie zasobu fizycznego wygląda nieco inaczej. Dla serwera JBoss w naszym przykładzie będzie to plik `sample-ds.xml` umieszczony w katalogu `$JBOSS_HOME/server/$PROFILE_NAME/deploy`:

```
<?xml version="1.0"
  encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/PhysicalSampleDS
    </jndi-name>
    <connection-url>jdbc:postgresql:
//localhost:5432/sampledb
    </connection-url>
    <driver-class>
      org.postgresql.Driver
    </driver-class>
    <user-name>testuser</user-name>
    <password>123456</password>
    <new-connection-sql>select 1
    </new-connection-sql>
    <use-java-context>false
    </use-java-context>
    <metadata>
      <type-mapping>PostgreSQL 8.3
      </type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Ostatnią rzeczy, którą musimy przygotować jest deskryptor specyficzny dla serwera, który zawiera odwzorowanie logicznej nazwy zasobu zdefiniowanej w aplikacji na jego fizyczny odpowiednik zdefiniowany w serwerze. W przypadku JBoss'a takie mapowanie odbywa się za pomocą pliku `jboss-web.xml` i może wyglądać następująco

```
<?xml version="1.0"
  encoding="UTF-8"?>
<jboss-web>
  <resource-ref>
    <res-ref-name>jdbc/SampleDS
    </res-ref-name>
    <jndi-name>jdbc/PhysicalSampleDS
    </jndi-name>
  </resource-ref>
</jboss-web>
```

Warto również pamiętać, że sterowniki JDBC

> Tworząc aplikacje Java Enterprise trzeba pamiętać, żeby nigdy nie korzystać bezpośrednio ze sterownika JDBC do uzyskiwania połączeń do bazy danych

powinny być umieszczone w odpowiednich katalogach serwera aplikacyjnego, a nie w archiwach aplikacji (war, ear). Przykładowo w JBoss'ie powinniśmy je umieszczać w katalogu:

`$JBOSS_HOME/server/$PROFILE_NAME/lib`

gdzie `JBOSS_HOME` to ścieżka, w której zainstalowaliśmy serwer a PROFILE_NAME to nazwa profilu, w ramach którego ma pracować nasza aplikacja. Umieszczenie sterownika JDBC w archiwum aplikacji to proszenie się o trudne do wychwycenia problemy związane z hierarchią ładowania klas w serwerze aplikacji.

Tworząc aplikacje Java Enterprise trzeba pamiętać, żeby nigdy nie korzystać bezpośrednio ze sterownika JDBC do uzyskiwania połączeń do bazy danych (`DriverManager.getConnection(...)`). Zawsze należy korzystać z mechanizmów dostarczanych przez serwer aplikacji.

Transakcje lokalne

Rozważania związane z transakcjami w dostępie do baz danych zacznijmy od tzw. transakcji lokalnych. Są to transakcje realizowane na poziomie menedżerów poszczególnych zasobów (np. baz danych czy kolejek JMS). Zarządzając takimi transakcjami korzystamy wyłącznie z właściwości udostępnianych przez interfejsy specyficzne dla danego zasobu. Dla baz danych będzie to interfejs JDBC. Korzystamy tutaj z tego, że menedżer zasobów (np. silnik bazy danych) jest jednocześnie menedżerem transakcji. W transakcjach lokalnych nie jest wykorzystywany menedżer transakcji serwera aplikacji. Rola serwera aplikacji w przypadku transakcji lokalnych sprowadza się jedynie do umożliwienia dostępu do zasobu.

Transakcje lokalne możemy wykorzystać jeżeli w naszym systemie mamy jedną bazę danych, co jest dość typowym przypadkiem.

Zarządzanie transakcjami bazadanowymi odbywa się na poziomie połączenia do bazy danych, reprezentowanego w interfejsie JDBC przez obiekt `java.sql.Connection`. W serwerze aplikacji obiekty te uzyskujemy z obiektu `javax.sql.DataSource` pobranego przez naszą aplikację z JNDI. Obiekt java.sql.Connection udostępnia następujące metody służące do zarządzania transakcjami:

- `void setAutoCommit(boolean autoCommit)` – umożliwia wyłączenie trybu autocommit, co jest równoznaczne z rozpoczęciem transakcji. Domyślnie wszystkie zwracane przez serwer połączenia są ustawiane w tryb autocommit, co oznacza, że każde zapytanie do bazy danych wykonywane jest w oddzielnej transakcji.
- `void commit()` – powoduje zatwierdzenie bieżącej transakcji. Jednocześnie powoduje rozpoczęcie nowej transakcji.
- `void rollback()` – powoduje wycofanie bieżącej transakcji. Jednocześnie powoduje rozpoczęcie nowej transakcji.
- `void setTransactionIsolation(int level)` – ustawia wskazany poziom izolacji transakcji. Więcej na ten temat w dalszej części artykułu.
- `Savepoint setSavepoint(String name)` – umożliwia ustawienie punktu kontrolnego o danej nazwie, do którego będzie się można wycofać bez wycofywania całej transakcji.
- `void releaseSavepoint(Savepoint savepoint)` – powoduje usunięcie danego punktu kontrolnego z bieżącej transakcji.
- `void rollback(Savepoint savepoint)` – powoduje wycofanie zmian do podanego punktu kontrolnego.

W praktyce tworząc aplikacje Java Enterprise, które zazwyczaj mają charakter aplikacji OLTP (Online Transaction Processing) bardzo rzadko korzysta się z punktów kontrolnych transakcji. Mają one głównie zastosowanie w aplikacjach, w których występują długotrwające transakcje. W takich aplikacjach występują zazwyczaj złożone i kosztowne operacje, któ-

> Transakcje lokalne możemy wykorzystać jeżeli w naszym systemie mamy jedną bazę danych, co jest dość typowym przypadkiem.

rych wycofywanie w całości byłoby nieopłacalne. W aplikacjach internetowych ten przypadek jest bardzo rzadko spotykany. W praktyce mamy więc do czynienie tylko z trzema metodami – `setAutocommit(...)`, `commit()` oraz `rollback()`.

Przyjrzyjmy się jak za pomocą tych metod zrealizować transakcję bazodanową wpisującą dane do dwóch różnych tabel z poziomu serwletu (pominąłem w tym przykładzie właściwą obsługę wyjątków):

```
public class JDBCTransactionDemoServlet extends
HttpServlet {
  protected void doGet(HttpServletRequest req,
    HttpServletResponse res)
      throws ServletException, IOException {
    ...
    try {
      Context ctx = new InitialContext();

      // uzyskanie połączenia
      DataSource ds1 = (DataSource) ctx
        .lookup(
      "java:comp/env/jdbc/TransactionDemoDS1");
      Connection conn = ds1.getConnection();

      // rozpoczęcie transakcji
      conn.setAutoCommit(false);

      // wykonanie operacji
      PreparedStatement stmt1 = conn
        .prepareStatement(
         "insert into a values (1, 'dane_x'");
      PreparedStatement stmt2 = conn
        .prepareStatement(
         "insert into b values (2, 'dane_y'");
      stmt1.executeUpdate();
      stmt2.executeUpdate();
      stmt1.close();
      stmt2.close();

      // zatwierdzenie transakcji i
      // zamknięcie połączenia
      conn.commit();
      conn.close();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
    ...
  }
}
```

Jak można zauważyć koncepcyjnie zarządzanie transakcjami z poziomu interfejsu JDBC nie wydaje się szczególnie trudne i składa się z następujących kroków:

- Uzyskania połączenia z serwera aplikacji
- Rozpoczęcia transakcji.
- Wykonania operacji na bazie danych w ramach otwartej transakcji.
- Zatwierdzenia transakcji.
- Zamknięcia połączenia. W serwerze aplikacji połączenie nie jest fizycznie zamykane, a jedynie wraca do puli połączeń i może być ponownie wykorzystane – niekoniecznie przez naszą aplikację.

Jednak jak to zwykle bywa, diabeł tkwi w szczegółach. W naszym przypadku jest to właściwa obsługa sytuacji wyjątkowych. W trakcie działania przedstawionego wyżej kodu może dojść do wielu sytuacji, które wymagają specjalnego obsłużenia, w szczególności:

- Może nie udać się pobranie połączenia (np. wyczerpana pula połączeń).
- Mogą nie udać się operacje wykonywane na danych w bazie (np. błąd w aplikacji albo naruszenie więzów integralności danych na poziomie bazy danych). Powinno to skutkować wycofaniem transakcji.
- Może nie udać się zatwierdzenie transakcji (np. wskutek wykrytego konfliktu podczas modyfikacji danych w bazie – przypadek typowy dla baz z optymistycznym blokowaniem).
- Problemy komunikacyjne pomiędzy serwerem aplikacji a bazą danych na każdym z wymienionych eta-

> Teraz nasuwa się pytanie gdzie taką obsługę wyjątków umieścić w kodzie naszej aplikacji.

pów. Mój ulubiony przykład z życia to zrywanie połączeń w związku z przekroczeniem zdefiniowanej liczby stanów w zaporze ogniowej (*statefull firewall*) pracującej pomiędzy serwerem aplikacji a serwerem bazy danych.

Zobaczmy jak można sobie z tymi sytuacjami poradzić:

```
Context ctx = new InitialContext();
DataSource ds1 = (DataSource) ctx
  .lookup("java:comp/env/jdbc/TransactionDemoDS1");

Connection conn = null;
boolean ok = false;

try {
  conn = ds1.getConnection();
} catch (SQLException e) {
  throw new RuntimeException(e);
}

RuntimeException exception = null;
try {
  conn.setAutoCommit(false);
  // tutaj wykonujemy operacje na bazie w rozpoczętej transakcji
  ok = true;
} catch (SQLException e) {
  exception = new RuntimeException(e);
} catch (RuntimeException e) {
  exception = e;
} finally {
  if (ok) {
    try {
      conn.commit();
    } catch (SQLException e) {
      if (exception != null) {
        exception = new RuntimeDoubleException(exception, e);
      } else {
        exception = new RuntimeException(e);
      }
    }
  } else {
    try {
      conn.rollback();
    } catch (SQLException e) {
      if (exception != null) {
        exception = new RuntimeDoubleException(exception, e);
      } else {
        exception = new RuntimeException(e);
      }
    }
  }

  try {
    conn.close();
  } catch (SQLException e) {
    if (exception != null) {
      exception = new RuntimeDoubleException(exception, e);
    } else {
      exception = new RuntimeException(e);
    }
  }
}

if (exception != null) {
  throw exception;
}
```

> W praktyce najlepiej sprawdza się podejście trzecie uzupełniane podejściem drugim.

Jak widać pełna obsługa tego typu transakcji wymaga sporej ilości kodu, którego działanie nie jest już takie łatwe do prześledzenia.

Teraz nasuwa się pytanie gdzie taką obsługę wyjątków umieścić w kodzie naszej aplikacji.

W pierwszym podejściu wydawałoby się, że w każdym miejscu gdzie realizujemy nasze transakcje. Jeżeli jednak zadanie prawidłowego zarządzania transakcjami i połączeniami oddamy w ręce programistów poszczególnych modułów czy ekranów aplikacji to nie możemy oczekiwać, że nasz system będzie odpowiednio wysokiej jakości. Nawet dobremu programiście zdarzy się jakiś element tej obsługi przeoczyć lub zrealizować nieprawidłowo. W najlepszym przypadku będzie to prowadziło do wycieków z puli połączeń, a w najgorszym do ulotnych, ciężkich do zlokalizowania błędów w logice naszej aplikacji. Poza tym takie podejście powoduje gigantyczną duplikację kodu. Nawet w niewielkich systemach miejsc, w których należałoby umieścić przedstawiony wyżej kod byłoby pewnie z kilkadziesiąt. Prowadzi to w oczywisty sposób do pogorszenia czytelności kodu (obsługa wyjątków zajmuje więcej niż logika operacji na bazie) i kłopotów w jego utrzymaniu.

Podejście drugie polega na zamknięciu powyższego kodu w jedną klasę usługową i wywoływaniu go z tych miejsc naszej aplikacji, gdzie występuje zarządzanie transakcjami.

Sytuacja komplikuje się wtedy, gdy budujemy złożony system, w którym liczba możliwych interakcji pomiędzy fragmentami kodu realizującym logikę transakcyjną jest trudna do oszacowania, a co za tym idzie do oprogramowania. Wtedy zostaje nam podejście trzecie, czyli scentralizowane zarządzania transakcjami.

Podejście to polega na umieszczeniu przetwarzania całego żądania do naszej aplikacji w jednej transakcji JDBC. Można to zrealizować na poziomie metod `doGet(...)` czy `doPost(...)` serwleta lub kodu kontrolera z modelu MVC a połączenie do bazy danych umieścić w parametrach żądania HTTP, przez co zawsze mamy do niego dostęp z dowolnego miejsca w naszej aplikacji. Takie podejście bardzo upraszcza zarządzanie transakcjami i połączeniami do bazy danych w naszej aplikacji. Oczywiście jak wszystkie „złote środki" takie podejście ma swoje ograniczenia. Jeśli zdecydujemy się na takie rozwiązanie należy przede wszystkim pamiętać, że w przypadku wykonywania przez nas długotrwałych operacji (np. skomplikowany rendering treści) cały czas blokujemy jedno połączenie na poziomie serwera aplikacji oraz zasoby (wiersze/tabele) na poziomie bazy danych.

W praktyce najlepiej sprawdza się podejście trzecie uzupełniane podejściem drugim. To znaczy standardowo zawsze wszystko obejmujemy transakcją, a tylko w pewnych krańcowych sytuacjach (np. ze względów wydajnościowych) odstępujemy od tej reguły i obsługujemy zachowanie transakcyjne naszego systemu na poziomie poszczególnych komponentów.

Transakcje zarządzane przez menedżer transakcji serwera aplikacji (transakcje rozproszone)

Jeśli w systemie operujemy na więcej niż jednym zasobie transakcje lokalne przestają wystarczać. Musimy wtedy skorzystać z dobrodziejstwa menedżera transakcji rozproszonej, który jest składową serwera aplikacji. Przyjrzyjmy się jak w takim przypadku wygląda korzystanie z baz danych. Poniższy przykład pokazuje fragment aplikacji, która operuje na dwóch bazach danych w ramach pojedynczej transakcji.

```
...
Context ctx = new InitialContext();
UserTransaction ut = (UserTransac-
tion) ctx
      .lookup("java:comp/UserTransac-
tion");

ut.begin();

DataSource ds1 = (DataSource) ctx
```

> Zarządzanie transakcjami przez menedżer transakcji serwera aplikacji równie dobrze sprawdza się w przypadku, gdy korzystamy z jednej bazy danych.

```
   .lookup("java:comp/env/jdbc/Trans-
actionDemoDS1");
DataSource ds2 = (DataSource) ctx
   .lookup("java:comp/env/jdbc/Trans-
actionDemoDS2");

Connection conn1 =
  ds1.getConnection();
Connection conn2 =
  ds2.getConnection();

doSomethingInFirstDatabase(conn1);
doSomethingInSecondDatabase(conn2);

conn1.close();
conn2.close();

ut.commit();
...
```

Jak widać z poziomu kodu uczestnictwo naszych operacji na bazach danych w transakcji rozproszonej nie wymaga żadnych specjalnych zabiegów. Widzimy również, że nie korzystamy tutaj z żadnych metod interfejsu JDBC, które służyły nam do zarządzania transakcjami lokalnymi. Wszystkie te szczegóły ukrywa przed nami menedżer transakcji i odpowiednie implementacje sterowników do baz danych.

Dla czytelności pominąłem obsługę wyjątków. Oczywiście operacje pobierania i zamykania połączeń powinny być nią objęte. Również kod całej transakcji powinien być obsłużony w taki sposób, w jaki przedstawiłem to w pierwszej części artykułu (rozdział „Obsługa wyjątków i sytuacji brzegowych przy samodzielnym zarządzaniu transakcjami").

Aby móc operować na bazach danych w ramach transakcji rozproszonej musi zostać spełnionych kilka warunków, których nie widać z przedstawionego wyżej przykładu:

- Przede wszystkim silnik bazy danych, którego używamy musi wspierać dwufazowy protokół zatwierdzania transakcji (na przykład w tak popularnej bazie danych jak PostgreSQL wsparcie dla tego protokołu pojawiło się dopiero od wersji 8.1).
- Po drugie sterownik JDBC do bazy danych musi implementować odpowiednie interfejsy pozwalające na współpracę z menedżerem transakcji (XADataSource, XAConnection, XAResource). W przypadku wspomnianej wyżej bazy PostgresSQL funkcjonalność ta pojawiła się od wersji 8.1dev-403 sterownika.
- Wreszcie musimy we właściwy sposób skonfigurować źródło danych w serwerze aplikacji (czyli jako źródło XA). Ponieważ sterowniki JDBC do danej bazy danych przychodzą zazwyczaj w jednej paczce zawierającej zarówno sterowniki XA jak i nie XA to dość łatwo o pomyłkę. Sprzyja temu większość przykładów użycia sterownika, które pokazują konfigurację źródła danych nie wspierającego uczestnictwa bazy danych w transakcji rozproszonej. Skonfigurowanie źródła danych, które nie wspiera transakcji rozproszonych to jeden z najczęstszych błędów popełnianych podczas tworzenia systemów transakcyjnych.

Korzystając z baz danych w ramach transakcji rozproszonej (JTA) nie wolno nam używać następujących metod zdefiniowanych w obiekcie `java.sql.Connection`:

- `setAutoCommit(...)`
- `commit()`
- `rollback()`
- `setSavepoint()`

Zarządzanie transakcjami przez menedżer transakcji serwera aplikacji równie dobrze sprawdza się w przypadku, gdy korzystamy z jednej bazy danych. Tak więc możemy zarządzać transakcjami w całym systemie w jednolity sposób, korzystając z interfejsu `UserTranscation`, bez potrzeby wnikania w aspekt zarządzania transakcjami z poziomu JDBC. Zachęcam do takiego podejścia nawet w systemach z jednym źródłem danych.

Poziomy izolacji transakcji

Omówienie zastosowania baz danych w systemach transakcyjnych nie może się obyć bez

> W idealnym świecie transakcje nie widzą żadnych efektów działań wykonywanych przez inne transakcje dopóki tamte nie zostaną zatwierdzone.

poruszenia tematyki poziomów izolacji transakcji w bazach danych. Być może jest to nawet najważniejszy aspekt budowy systemów transakcyjnych korzystających z baz danych. W pierwszej części artykułu opisując właściwości izolacji transakcji wspominałem, że w odniesieniu do baz danych własność ta nastręcza wielu kłopotów. Przyjrzyjmy się dlaczego tak jest.

W idealnym świecie transakcje nie widzą żadnych efektów działań wykonywanych przez inne transakcje dopóki tamte nie zostaną zatwierdzone. Obserwując taki świat z zewnątrz mielibyśmy wrażenie, że wszystkie transakcje wykonują się po kolei (są uszeregowane). Ponieważ taki idealny świat działał by zbyt wolno, to zaczęto poszukiwać sposobów na złagodzenie tego wymagania i zwiększenie wydajności. W ten sposób narodziły się w bazach danych różne poziomy izolacji transakcji. W zależności od poziomu dopuszczają one istnienie określonych anomalii przy współbieżnym wykonywaniu transakcji. Aby dobrze zrozumieć poziomy izolacji transakcji musimy najpierw przyjrzeć się tym anomaliom.

Brudne odczyty (*ang. dirty reads*)

Brudny odczyt oznacza, że transakcje mogą widzieć zmiany wykonywane przez inne transakcje zanim zmiany te zostaną zatwierdzone. W takim przypadku istnieje możliwość, że w przypadku wycofania zmian inne transakcje będą dalej pracowały na niewłaściwych danych. Sytuację brudnego odczytu ilustruje poniższy diagram:

Transakcja A	Czas	Transakcja B
begin	t0	begin
-		-
update p	t1	-
-		-
-	t2	retrieve p
-		-
rollback	t3	-
-		-
-	t4	?

Transakcje A i B rozpoczynają się w tej samej chwili t0. Następnie transakcja A aktualizuje wiersz p, który transakcja B odczytuje w chwili t2. Niestety w chwili t3 transakcja A wycofuje wykonane wcześniej zmiany, co oznacza, że transakcja B używa wartości p, która tak naprawdę nigdy nie znalazła się w bazie. Najłatwiej sobie wyobrazić jakie „zniszczenia" może powodować takie zachowanie systemu jeśli p oznacza wysokość oprocentowania naszego rachunku bankowego.

Niepowtarzalne odczyty (*ang. non-repeatable reads*)

Niepowtarzalny odczyt oznacza, że transakcja, która wielokrotnie odczytuje ten sam wiersz, może otrzymać różne wyniki chociaż z punktu widzenia spójności danych oba poprawne, czyli zatwierdzone przez inne transakcje. Sytuację niepowtarzalnego odczytu ilustruje diagram poniżej:

Transakcja A	Czas	Transakcja B
begin	t0	begin
-		-
retrieve p	t1	-
-		-
-	t2	update p
-		-
-	t3	commit
-		-
retrieve p?	t4	-

Transakcja A pobiera dwa razy wiersz p (t1 i t4), za każdym razem otrzymując inne wyniki. Jeśli inne dane są modyfikowane w oparciu o wartość p, może to prowadzić do niespójności, które jest trudno wykryć.

Fantomy (ang. phantom reads)

Anomalia ta polega na tym, że jeżeli transakcja dwa razy odczytuje zbiór danych według tych samych warunków to może ona otrzymać dwa różne wyniki. Sytuację taką ilustruje diagram poniżej:

> Przyjrzyjmy się teraz poziomom izolacji transakcji zdefiniowanych w specyfikacji JDBC

```
Transakcja A          Czas        Transakcja B
-------------------------------------------------
    begin                 t0          begin
      -                                   -
    retrieve q -> a,b     t1              -
      -                                   -
      -                   t2          insert c into q
      -                                   -
      -                   t3          commit
      -                                   -
    retrieve q -> a,b,c!  t4              -
      -                                   -
```

Jak można zauważyć transakcja A dwa razy odczytuje tabelę q, za każdym razem otrzymując inne dane, co jest związane z zatwierdzeniem w międzyczasie transakcji B.

W oparciu o eliminację powyższych anomalii stworzono model poziomów izolacji transakcji. Model zdefiniowano w standardzie języka SQL. Specyfikacja JDBC również opiera się na tym modelu wprowadzając jeden dodatkowy poziom. Przyjrzyjmy się teraz poziomom izolacji transakcji zdefiniowanych w specyfikacji JDBC (stałe w `java.sql.Connection`), w kolejności od najmniej do najbardziej restrykcyjnych:

- `TRANSACTION_NONE` – wskazuje, że sterownik nie wspiera transakcji. Ten poziom nie ma nic wspólnego z omawianymi wyżej anomaliami dostępu do danych. Może on być używany w sytuacji, kiedy za pomocą interfejsu JDBC udostępniane są dane lub systemy, w których nie da się zdefiniować żadnego sensownego zachowania transakcyjnego. Takim przykładem jest biblioteka csvjdbc (http://csvjdbc.sourceforge.net/), która za pomocą interfejsu JDBC umożliwia dostęp do plików CSV.

- `TRANSACTION_READ_UNCOMMITED` - poziom, na którym mogą występować wszystkie wymienione anomalie.

- `TRANSACTION_READ_COMMITED` - poziom oznaczający, że zmiany wykonywane przez transakcje nie są widoczne dla innych transakcji do momentu jej zatwierdzenia, czyli mamy tutaj ochronę przed brudnymi odczytami. Na tym poziomie dalej mogą występować niepowtarzalne odczyty i fantomy.

- `TRANSACTION_REPEATABLE_READ` - ten poziom chroni zarówno przed brudnymi odczytami jak i przed niepowtarzalnymi odczytami. Fantomy dalej mogą na nim występować.

- `TRANSACTION_SERIALIZABLE` - na tym poziomie nie mogą występować żadne z opisywanych anomalii.

W kontekście tak przyjętej definicji poziomów izolacji chciałbym zwrócić uwagę na jeden znaczący fakt. Ustawienie poziomu izolacji SERIALIZABLE wcale nie oznacza, że mamy zagwarantowane wykonywanie transakcji w sposób uszeregowany (jedna po drugiej). Niestety przez nieszczęśliwy, moim zdaniem, dobór nazwy najwyższego poziomu izolacji wiele osób jest przekonanych, że tak się właśnie dzieje.

Piękny świat poziomów izolacji zdefiniowanych w JDBC API i standardzie SQL psują dostępne implementacje baz danych, które czasami niektórych poziomów nie wspierają albo stosują inną nomenklaturę. Przykładowo w bazie PostgreSQL są tylko dwa poziomy: READ_COMMITED i SERIALIZABLE. W DB2 są cztery, ale inaczej się nazywają, a dodatkowo te same nazwy używane są do określenia różnych poziomów. Tą złożoną sytuację ilustruje tabela poniżej.

DB2 Transaction Isolation Level	JDBC Transaction Isolation Level
Uncommitted Read	READ_UNCOMMITED
Cursor stability	READ_COMMITED
Read stability	REPEATABLE_READ(*)
Repeatable read (*)	SERIALIZABLE

Z tego względu zawsze przy nowej bazie danych czeka nas lektura dokumentacji sterownika JDBC jak i samej bazy danych.

> zasadniczo podział przebiega między algorytmami wykorzystującymi pesymistyczne i optymistyczne blokowanie zasobów.

Niestety na tym nie kończą się komplikacje związane z poziomami izolacji transakcji. Dostawcy baz danych implementują poziomy izolacji korzystając z różnych mechanizmów i algorytmów. Zasadniczo podział przebiega między algorytmami wykorzystującymi pesymistyczne i optymistyczne blokowanie zasobów. Przyjrzymy się jaki może mieć to wpływ na transakcje wykonywane na tym samym poziomie izolacji w bazach danych, które stosują odmienne podejścia implementacyjne (PostgreSQL i IBM DB2).

Dla obu baz danych będziemy chcieli wykonać jednocześnie dwie transakcje (symulowane za pomocą dwóch oddzielnych konsol) na tym samym poziomie izolacji (`READ COMMITTED`):

- jedna odczytująca zawartość tabeli `foo` (transakcja A),
- druga modyfikująca wybrany wiersz tabeli `foo` (transakcja B).

W obu przypadkach sprawdzamy jakie jest zachowanie odczytu w transakcji A w przypadku, w którym doszło już do aktualizacji danych przez transakcję B.

Przed wykonaniem tych transakcji zawartość tabeli `foo` była następująca:

```
id       data
---------------
  1      dane 1
  2      dane 2
  3      dane 3
```

Transakcja A

```
test=> BEGIN;
test=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;


test=> SELECT * FROM foo;
 id |  data
----+--------
  1 | dane 1
  2 | dane 2
  3 | dane 3
(3 rows)
```

Odczyt daje się wykonać chociaż transakcja B zmieniała dane w tabeli.

```
test=> COMMIT;
test=>
```

Transakcja B

```
test=> BEGIN;
test=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=> UPDATE foo SET data='dane 2 zmienione' WHERE id=2;









test=> COMMIT;
test=>
```

PostgreSQL (baza danych z optymistycznym blokowaniem z wykorzystaniem mechanizmu Multi-Version Concurrency Control)

Jak widać mimo, że wykonujemy identyczne transakcje na takim samym poziomie izolacji, to bazy zachowują się odmiennie - chociaż w obu przypadkach założenie co do braku odpowiednich anomalii jest spełnione. Niestety ma to bezpośredni wpływ na działanie naszego systemu, szczególnie w zakresie jego wydajności. Musimy taki aspekt uwzględnić przy projektowaniu i implementacji naszego systemu.

Kolejnym dość złożonym zagadnieniem jest ustawienie żądanego poziomu izolacji z poziomu aplikacji Java Enterprise. Niestety tutaj każdy serwer aplikacji zachowuje się inaczej:

- Pierwsze rozczarowanie przeżywamy nie mogąc na poziomie deskryptora aplikacji ustawić poziomu izolacji źródła danych. Takie rozwiązanie wydaje się najbardziej naturalne, gdyż poziom izolacji transakcji wynika z charakteru aplikacji. Niestety

Transakcja A

```
db2 => SET ISOLATION cs


db2 => SELECT * FROM foo
```

W tym momencie konsola zawisa czekając na zwolnienie blokady nałożonej przez transakcję B. Wynik pojawi się dopiero po wykonaniu operacji COMMIT na transakcji B.

```
ID          DATA

----------- ------------------------
          1 dane 1

          2 dane 2 - zmienione

          3 dane 3


  3 record(s) selected.
db2 => COMMIT
db2 =>
```

Transakcja B

```
db2 => SET ISOLATION cs
db2 => UPDATE foo SET data='dane 2 -
zmienione' WHERE id = 2




db2 => COMMIT
db2 =>
```

IBM DB2 (baza danych z pesymistycznym blokowaniem)

specyfikacja Java Enterprise pomija całkowicie ten istotny aspekt aplikacji. Konfigurowanie tego na poziomie konfiguracji fizycznego zasobu w serwerze wydaje się pomysłem karkołomnym, stawiającym pod dużym znakiem zapytania całą koncepcję rozdzielenia zasobów logicznych od fizycznych.

- Zazwyczaj mamy możliwość ustawienia domyślnego poziomu izolacji na poziomie konfiguracji zasobu fizycznego w serwerze aplikacji. Ale jeśli mamy sytuację, w której chcemy skorzystać z zasobu na różnych poziomach izolacji mamy problem, bo musimy konfigurować więcej niż jedno fizyczne źródło danych do tej samej bazy danych. Niestety często jest to jedyne rozwiązanie.

- W niektórych serwerach aplikacji daje się ustawić poziom izolacji na poziomie logicznego źródła danych za pomocą specyficznego deskryptora deploymentu. Np. w serwerze aplikacji IBM Websphere można to zrobić tak (fragment pliku `ibm-web-ext.xmi`):

```
<resourceRefExtensions
  xmi:id="ResourceRef_ext_1"
  isolationLevel=
    "TRANSACTION_READ_COMMITTED">
  <resourceRef
   href="WEB-INF/web.xml#ResRef_1" />
</resourceRefExtensions>
```

gdzie do zasobów wymienionych w deskryptorze `web.xml` dodajemy dodatkowe własności, które będą uwzględniane w momencie, gdy zażądamy dostępu do danego zasobu.

- Ostatnią deską ratunku wydaje się metoda `setTransactionIsolation` wywołana bezpośrednio na uzyskanym z serwera aplikacji połączeniu (`java.sql.Connection`). Niestety nie mamy gwarancji, że metoda ta zadziała w przypadku gdy transakcja jest już rozpoczęta, a z taką sytuacją mamy do czynienia gdy korzystamy z transakcji JTA. Niektóre bazy danych i sterowniki do nich pozwalają zmienić poziom izolacji w trakcie trwania transakcji (np. baza IBM DB2 pozwala zmienić poziom izolacji transakcji na poziomie pojedynczego zapytania SQL).

Jak wynika z powyższych rozważań JDBC API w zakresie poziomów izolacji w praktyce nie jest w stanie ukryć przed programistą rzeczywistych implementacji silników baz danych i sterowników do nich. Zawsze musimy wniknąć w szczegóły działania bazy danych, konfiguracji sterownika do niej, konfiguracji serwera aplikacji i procesu instalacji aplikacji.

Można się jeszcze zastanawiać w jakich przypadkach stosować konkretne poziomy izolacji. Ciężko na to pytanie jednoznacznie odpowiedzieć nie znając wymagań dla konkretnego

> Można się jedynie kierować pewnymi heurystykami wynikającymi z dotychczasowego doświadczania.

systemu. Można się jedynie kierować pewnymi heurystykami wynikającymi z dotychczasowego doświadczania. I tak z mojej praktyki wynika, że:

- Najlepiej zacząć od ustawienia domyślnego poziomu izolacji transakcji na `READ_COMMITTED`. Zapewnia on rozsądny kompromis między wydajnością systemu, a zapewnieniem spójności danych.
- Poziomu `READ_UNCOMMITTED` możemy użyć przy wyciąganiu mniej istotnych danych na potrzeby prezentacji. Ma to szczególne znaczenie w przypadku korzystania z bazy danych z pesymistycznym blokowaniem.
- Poziomu `SERIALIZABLE` można użyć do realizacji semafora na bazie danych oraz oczywiście do implementacji operacji transferu środków pomiędzy kontami bankowymi.

Pozostałe aspekty korzystania z baz danych z poziomu serwera aplikacji

Na zakończenie chciałbym przedstawić kilka problemów/aspektów, na które można się natknąć w codziennej praktyce zawodowej, a co do których dość ciężko znaleźć wyjaśnienie czy rozwiązanie w dokumentacji produktów czy w Internecie.

Timeout transakcji JTA a timeout'y na poziomie bazy danych

Pomiędzy timeout'em transakcji JTA a działaniami wykonywanymi przez nas na bazie danych nie ma żadnego związku, poza tym, że po przekroczeniu timeout'u JTA wszystkie wykonane przez nas operacje na bazie danych powinny zostać wycofone. Wynika to z przyczyn, o których pisałem w pierwszej części artykułu.

Co więcej, znane mi bazy danych nie oferują funkcjonalności, która by umożliwiała ustawienie maksymalnego czasu trwania transakcji w bazie danych.

Wiem, że wiele osób poszukuje takiego rozwiązania. Pewne przybliżone rozwiązanie można uzyskać korzystając ze specyficznych własności konkretnych silników baz danych. Możemy tu mówić w zasadzie o dwóch typach rozwiązań:

- Część baz danych (np. PostgreSQL) umożliwiają ustawienie maksymalnego czasu trwania pojedynczego zapytania. Odbywa się to na poziomie konfiguracji serwera bazy danych (`statement_timeout` w PostgreSQL) lub za pomocą JDBC API - `java.sql.Statement.setQueryTimeout` (niestety w większości sterowników niezaimplementowane, gdyż silnik bazy tego nie wspiera).
- Część baz danych (np. Oracle, IBM DB2, MS SQL Server) umożliwia ustawienie maksymalnego czasu utrzymywania blokady na zasobie (wiersz, tabela). Zazwyczaj znane jest to pod pojęciem `lock-timeout`. Rozwiązanie to jednak nie daje żadnej gwarancji i ma duże ograniczenia. Na przykład w przypadku pełnego przeglądania dużej tabeli (*ang. full scan*) blokada może się zmieniać przy przechodzeniu z wiersza na wiersz, przez co nigdy nie uzyskamy wyjątku oznaczającego `lock-timeout` a jednocześnie dostęp do takiej tabeli będzie mocno utrudniony.

Korzystanie z bazy danych poza otwartą transakcją JTA

Czasami chcemy zrealizować jakieś operacje na bazie danych poza aktualnie trwającą transakcją. W takim przypadku możemy skorzystać bezpośrednio z menedżera transakcji. Przykład takiego rozwiązania przedstawiłem w pierwszej części artykułu w sekcji „Bezpośrednie korzystanie z menedżera transakcji".

Korzystanie z połączenia przed rozpoczęciem transakcji JTA

Spotykam się czasami z pytaniem, jak zachowuje się połączenie do bazy wzięte przed roz-

> Tak więc zawsze trzeba sprawdzić jaki jest domyślny poziom izolacji w naszym specyficznym przypadku

poczęciem transakcji JTA i co się dzieje jeśli po rozpoczęciu transakcji JTA będę chciał go dalej używać? Co się stanie jeśli po rozpoczęciu transakcji JTA wezmę połączenie z tego samego źródła danych? Mamy więc do czynienia z przykładem kodu jak poniżej:

```
Context ctx = new InitialContext();
DataSource ds1 = (DataSource) ctx
  .lookup("java:comp/env/jdbc/
    TransactionDemoDS1");
Connection conn1 =
  ds1.getConnection();

// zrób coś na połączeniu conn1

UserTransaction ut =
  (UserTransaction) ctx
    .lookup(
      "java:comp/UserTransaction");

ut.begin();

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

Connection conn2 =
  ds1.getConnection();

// zrób coś na połączeniu conn2

// zrób coś na połączeniu conn1

ut.commit();
```

Zachowanie systemu w takim przypadku jest następujące:

- Z połączeniem wziętym przed transakcją nic się nie dzieje. Można go dalej używać, tak jak byśmy go używali w transakcji lokalnej. Transakcja JTA nie ma na to połączenie żadnego wpływu.
- Pobrania jeszcze raz połączenia z tego samego źródła danych powoduje przydzielenie nowego połączenia i przypisanie go do transakcji JTA.
- Serwer aplikacyjny zapewnia więc prawidłową pracę na obu połączeniach, które są wzajemnie odseparowane.

Domyślne poziomy izolacji

Każdy serwer aplikacji stosuje własne domyślne poziomy izolacji transakcji, co więcej mogą się one różnić w zależności od bazy danych lub użytego sterownika JDBC. Na przykład:

- IBM Websphere w wersji 5.1 dla większości baz danych stosuje domyślny poziom REPEATABLE_READ, ale dla bazy Oracle poziome READ_COMMITED.
- Weblogic korzysta z ustawień domyślnych bazy danych.

Tak więc zawsze trzeba sprawdzić jaki jest domyślny poziom izolacji w naszym specyficznym przypadku, a najlepiej nadpisać konfigurację własnymi ustawieniami, co chroni nas przed sytuacją, w której dostawca serwera w drobnej poprawce aktualizującej zmienia domyślny poziom, a nasza produkcyjna aplikacja dotychczas świetnie działająca załamuje się nawet pod niewielkim obciążeniem.

Ze względu na obszerność zagadnienia związanego z wykorzystaniem baz danych przy budowie systemów transakcyjnych obiecany w poprzednim artykule temat wykorzystania systemów kolejkowania postanowiłem przenieść do kolejnego artykułu.

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] Specyfikacja oraz API JDBC, http://java.sun.com/javase/technologies/database/
[4] Dokumentacja do bazy danych PostgreSQL
[5] Dokumentacja do bazy danych IBM DB2
[6] Dokumentacja do serwera aplikacji JBoss
[7] Dokumentacja do serwera aplikacji JOnAS
[8] Dokumentacja do serwera aplikacji IBM Websphere
[9] Dokumentacja do serwera aplikacji BEA Weblogic (obecnie Oracle)

Zarządzanie sobą w czasie - alokacja czasu

Łukasz Lechert

Przeglądając oferty firm szkoleniowych można spotkać szkolenia z zarządzania czasem i zarządzania sobą w czasie. Analizując zawartość programów, zadałem sobie pytanie czym się różnią się te dwa podejścia? Przyjmując, że czas i człowiek są zasobami w projektach, można założyć, że zasób jakim jest dostępny czas płynie w sposób ciągły, nieprzerwany. Zarządzanie czasem odnosić się może do planowania w zarządzaniu. W dostępnym czasie można rozplanować realizację projektu, podzielić go na fazy, zadania, podzadania oraz zdefiniować czas realizacji. Do tak przygotowanego planu przydziela się następnie zasoby ludzkie i narzędzia. Ludzie odpowiedzialni za realizację poszczególnych zadań zobowiązani są do osiągnięcia celów, oraz doprowadzenie zadań do końca oraz dostarczenie ustalonych wcześniej, wypracowanych przez siebie efektów. Takie działanie można okreslić mianem zarządzania czasem. Specjaliści pracujący nad poszczególnymi zadaniami, na podstawie dostępnych informacji, priorytetów, określają kolejności realizacji poszczególnych czynności w ramach zadań, planują swój dzień pracy oraz czas wolny. W tym aspekcie ludzie ci zarządzają sobą i swoimi działaniami w celu najbardziej efektywnego wykorzystania zasobu jakim jest czas. Przyglądając się temu procesowi można dojść do wniosku, że tworzywem jest nie tylko czas, ale i człowiek, a człowiek i jego decyzje generują konkretną wartość dodaną np. dla projektu. Takie działanie można określić mianem zarządzania sobą w czasie, które jest tematem artykułów.

Kompetencje z zakresu zarządzania sobą w czasie zależą od dwóch czynników - wiedzy na temat dostępnych metod i narzędzi oraz postaw, które są przeważającym czynnikiem sukcesu. Postawa ma związek z aktywnością oraz planowaniem proaktywnym, czyli takim które wyprzedza bieg wydarzeń, powoduje podejmowanie wcześniejszej inicjatywy. W przeciwieństwie do planowania proaktywnego, planowanie reaktywne cechuje się reakcjami na skutek nieprzewidzianych zdarzeń, koncentracją na spełnianiu próśb i poleceń innych osób. Praca wymaga zarówno planowania proaktywnego, jak i reaktywnego. Skuteczne zarządzanie sobą w czasie opiera się przede wszystkim na planowaniu proaktywnym oraz konsekwentnym realizowaniu planu. Różnice można zauważyć w sposobie komunikacji osób, które działają w sposób reaktywny oraz proaktywny. Osoba mówiąca głosem dziecka lub rodzica, a zarazem ukierunkowana reaktywnie stwierdzi, że musi cos zrobić, nie ma na coś czasu, narzeka, że nie może sobie z czymś poradzić. Osoby proaktywne, biorące życie w swoje ręce mówią głosem dorosłego, decyduje, postanawiam, wolę lub wybieram. Są to wyrażenia, które słyszymy wiele razy codziennie, jednak niosą one dodatkową informację o człowieku oraz organizacji jego życia.

W życiu wykonujemy wiele różnych czynności, począwszy od prostej wycieczki z dziećmi na basen, skończywszy na obowiązkach w pracy. Alokacja to planowanie zagospodarowania czasu w bliskiej przyszłości w taki sposób, żeby osiągnąć pozytywne efekty w przyszłości. A jak wygląda Twoja idealna doba, na jakie czynności alokujesz swój dostępny czas? Odpowiedzi na te pytania będą z całą pewnością zróżnicowane. Idąc dalej, poszczególne aktywności można pogrupować. Obszarami w których możemy alokować czas są.

- zadania,
- więzi z innymi,
- własne Ja,
- alokacja bezkierunkowa.

W obszarze „zadania” mieszczą się wszystkie czynności, które traktujemy m.in. jako obowiązek, czyli porządki w domu, zadania w pracy oraz aktywności, których podjelismy się w celu rozwoju osobistego, lecz traktujemy je jako obowiązek, a nie przyjemność np. jogging w celu zrzucenia zbędnych kilogramów. W drugim obszarze znajdują się spotkania z przyjaciółmi, znajomymi i rodziną. Aktywności te powinny być relacyjne, również dla przyjemności. Ważnym warunkiem jest, że spotkania

> Alokacja to planowanie zagospodarowania czasu w bliskiej przyszłości w taki sposób, żeby osiągnąć pozytywne efekty w przyszłości.

te nie mają służyć tylko i wyłącznie załatwianiu spraw i interesów. W takich przypadkach stają się zadaniami np. kolacja z klientem. Kolejnym obszarem jest „własne Ja". Wielokrotnie przeznaczamy czas na hobby, pasje lub rozwój własnej kondycji. Czynności te charakteryzują się spontanicznością, potrzebą chwili np. w celu poprawienia złego nastroju lub pozbycia się stresu po pracy. Ostatnim najmniej produktywnym obszarem jest bezkierunkowa alokacja czasu. Są to czynności dające nam poczucie, że wykonujemy je dla własnego relaksu, np. granie w gry komputerowe, wycieczki do galerii handlowych, nie mające celu nabycia potrzebnych produktów, rozmowy za pomocą komunikatorów internetowych. Te aktywności daja jedynie wrażenie relaksu, ale w rzeczywistosci nim nie są, gdyż wykonując je, nie jesteśmy w stanie skupić się na bieżącej chwili, tylko wciaż myślimy o tym, co będzie, stanie się za moment. W ten sposób tracimy poczucie i kontrolę szybko upływającego czasu. Czynności te zyskały przydomek pożeraczy czasu.

Oprócz pożeraczy czasu istnieje również wiele czynników negatywnie wpływających na proces gospodarowania czasem. Zjawiska te nie tylko negatywnie wpływają na nas samych, ale również na naszych bliskich lub kolegów z pracy. Nawarstwiając się mogą prowadzić do patologii oraz dotkliwie dezorganizować proces zarządzania sobą w czasie.

Informacje o autorze

Autor jest absolwentem specjalizacji Inżynieria Oprogramowania Politechniki Wrocławskiej oraz Studium Podyplomowego Zarządzanie Projektem na Poznańskim Uniwersytecie Ekonomicznym. Interesuje się systemami wspierającymi procesy logistyczne, zagadnieniami zarządzania projektem oraz oprogramowaniem o otwartym kodzie.

Kontakt: ***lukasz.lechert@gmail.com***

Architektura aplikacji Flex i Java

Jeanette Stallons (Tłum. Magdalena Rudny, Grzegorz Kurka)

Platforma Java EE jest wiodącym rozwiązaniem dla technologii webowych. Natomiast Adobe Flash jest wiodącym rozwiązaniem wśród technologii RIA (Rich Internet Application). Dzięki użyciu obu tych technologii, twórcy aplikacji są w stanie dostarczyć atrakcyjne wizualnie, zorientowane na dane (ang. *data centric*) aplikacje, które łączą w sobie zalety stabilnej i wydajnej części serwerowej z zaletami intuicyjnego interfejsu użytkownika znanego z wielu stron www.

Poniższy artykuł omawia architekturę aplikacji opartej o połączenie technologii Flex i Java, skupiając się na przedstawieniu:

- architektury klient-serwer,
- różnych sposobów komunikacji między klientem i serwerem,
- zaletach i sposobach użycia Flash Remotingu,
- sposobie zapewnienia bezpieczeństwa aplikacji Flex,
- budowie aplikacji z użyciem zdarzeń, stanów, komponentów MXML i modułów

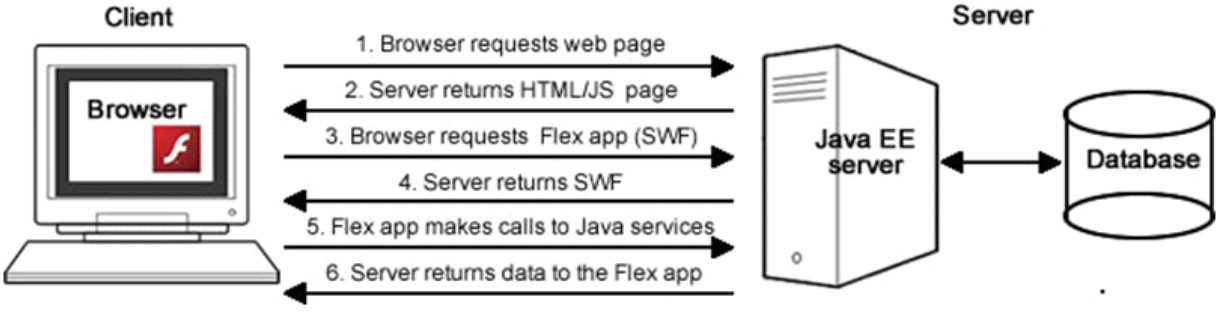
Dla lepszego zrozumienia tych zagadnień warto zapoznać się z materiałem video przygotowanym przez Adobe: Introduction to Flex 4 and Java integration.

Dla osób, które chcą dodatkowo zgłębić temat omawianych technologii przeznaczony jest artykuł: The technologies for building Flex and Java applications.

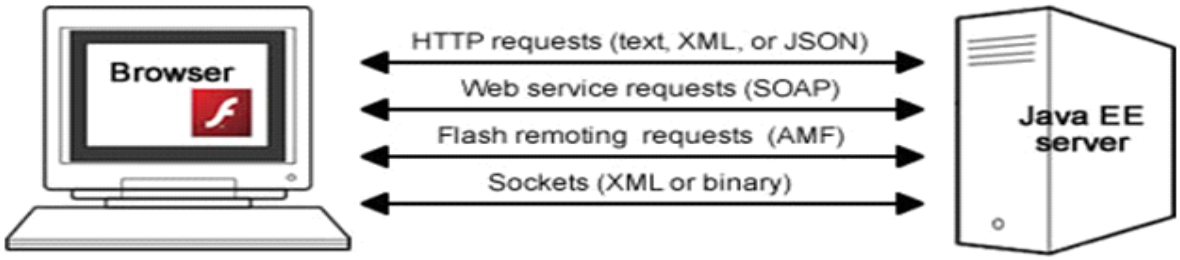
Architektura klient/serwer

Aplikacje Flex i Java wykorzystują architekturę trójwarstwową, w której warstwą prezentacji jest aplikacja Flex, warstwą biznesową jest działający na serwerze kod Java, a warstwą przechowywania danych jest baza danych. Sposób tworzenia kodu części serwerowej aplikacji nie odbiega od sposobu w jaki zwykle tworzy się aplikacje w języku Java. Obejmuje on projektowanie struktury obiektów, bazy danych, obiektów mapujących bazą danych (np. Hibernate lub EJB 3) oraz logikę biznesową operującą na stworzonych strukturach. Dostęp do całości aplikacji zapewnia warstwa prezentacji (Flex) poprzez protokół HTTP. Za sprawne przetwarzanie, przesyłanie i utrwalanie wprowadzonych przez użytkownika danych odpowiada logika zawarta w kodzie Java.

Typowe aplikacje HTML składają się ze zbioru stron między którymi użytkownik może swobodnie nawigować, jednak informacje o danych i stanach nie są przechowywane pomiędzy kolejnymi wywoływania tej samej strony. Aplikacje Flex, wprost przeciwnie, są z natury stanowe (ang. *stateful*). Flexowa aplikacja jest osadzona w kodzie strony HTML, gdzie Flash Player generuje kolejne widoki aplikacji bez przechodzenia oraz odświeżania strony w której aplikacja została osadzona. Generowanie widoków następuje dynamicznie z jednoczesną asynchroniczną wymianą danych między warstwą widoku a warstwą logiki biznesowej (patrz Rysunek 1) (podobnie jak to się odbywa w przypadku funkcjonalności XMLHttpRequest API w JavaScript).



Rys. 1: Architektura klient/serwer



Rysunek 2. Metody łączenia Flexa i Javy.

Komunikacja klient-serwer

Aplikacje Flex mogą wymieniać dane z warstwą logiki biznesowej przy użyciu socketów lub, co częstsze, za pomocą protokołu HTTP. Framework Flex udostępnia trzy rodzaje zdalnych wywołań procedur (RPC) po HTTP do komunikacji z serwerem: HTTPService, WebService i RemoteObject. Wszystkie te procedury opakowują połączenie HTTP nawiązane przez Flash Playera, które to natomiast korzysta z bibliotek zawartych w przeglądarce. Żadna z tych metod komunikacji nie umożliwia aplikacji Flex bezpośredniego sięgnięcia do zdalnej bazy danych.

HTTPService umożliwia wysyłanie żądań HTTP do plików JSP, XML, usług RESTful web serwisu lub do innych usług na serwerze, które zwrócą odpowiedź po HTTP. Chcąc użyć HTTPService musimy zdefiniować URL do klasy endpointu, funkcje nasłuchujące (ang. listener functions) - przeznaczone do asynchronicznego przetwarzania rezultatów wywołań HTTPService - oraz typ zwracanych danych (określający typ danych na który ma zostać przetłumaczona odpowiedz na wywołanie przesłana do aplikacji Flex). Możliwe jest przypisanie odpowiedzi, otrzymywanej w postaci tekstowej, do zmiennej typu String, konwersja do XML, E4X lub zwyczajnego obiektu ActionScript .W sytuacji gdy odpowiedź jest w formacie JSON można zastosować do deserializacji zewnętrzną bibliotekę Adobe Flex corelib, która zapewni przekształcanie obiektów JSON w obiekty ActionScriptowe. Do wywołania serwisu webowego opartego o SOAP możemy użyć HTTPService API lub bardziej specializowanego WebService API, który to automatycznie dokonuje serializacji i deserializacji pomiędzy tekstowym komunikatem protokołu SOAP a obiektami ActionScript.

Trzecią możliwością zdalnych wywołań procedur (RPC) jest RemoteObject API. Wykonuje ono wywołanie Flash Remoting do klasy Javowej znajdującej się po stronie serwera, otrzymując za pośrednictwem HTTP binarny strumień Action Message Format. Jeśli to tylko możliwe, wydajniej jest stosować Flash Remoting, który dzięki binarnemu formatowi przesyłania danych umożliwia do 10 razy szybszy transfer niż "rozgadane" formaty tekstowe takie jak JSON lub SOAP (patrz Rysunek 2). Porównanie AMF z technologiami opartymi o tekstowy format można przeczytać na stronie James Ward's Census RIA Benchmark application.

Flash Remoting

Flash Remoting to połączenie funkcjonalności zlokalizowanej po stronie klienta i po stronie serwera, które tworzy mechanizm dający możliwość pracy na obiektach zlokalizowanych po stronie serwera w taki sposób jakby były one obiektami lokalnymi platformy Flash. Remoting zapewnia transparentny transfer danych pomiędzy ActionScript, a danymi na serwerze, wykonując serializacje danych do Action Message Format (AMF), deserializacje oraz konwersje pomiędzy typami po stronie klienta i serwera.

Flash Remoting wykorzystuje mechanizmy wbudowane w Flash Player oraz mechanizmy po stronie serwera wbudowane w niektóre serwery (takie jak ColdFusion i Zend) lub moduły, które możemy doinstalowywać do większości popularnych serwerów (dla platformy Java EE są to BlazeDS i LiveCycle Data Services, dla .NET WebORB for .NET lub FluorineFX, dla PHP Zend framework lub amfphp i wiele innych). Więcej informacji o BlazeDS i LiveCycle Data Services można znaleźć w artykule The technologies for building Flex and Java applications.

BlazeDS i LiveCycle Data Services dostarczają mechanizmy do komunikacji po obu stronach kanału komunikacyjnego. Po stronie klienckiej oba te rozwiązania używają framework opar-

> AMF jest binarnym formatem danych służącym do serializacji obiektów ActionScript i przesyłania ich za pośrednictwem Internetu

ty o komunikaty (ang. message-based framework), który zapewnia interakcje z serwerem. Framework po stronie klienta wystawia kanały komunikacyjne, które opakowują połączenie pomiędzy Flexem a serwerową częścią BlazeDS lub LiveCycle Data Services. Kanały są grupowane w zbiory, które odpowiadają za nawiązywanie połączeń i obsługę błędów. Część serwerowa zawarta jest w aplikacji webowej J2EE. Aplikacja kliencka Flex wysyła żądanie za pośrednictwem kanału komunikacyjnego, które jest przekierowywane do endpointu zlokalizowanego w serwerze BlazeDS lub LiveCycle DS. Z endpointu żądanie jest przesyłane przez szereg obiektów Java, poczynając od obiektu MessageBroker, poprzez obiekt serwisu i obiekt docelowy (ang. destination object), a kończąc na obiekcie adaptera. Obiekty docelowe udostępniają serwisy Remoting, Proxying i Messaging, a LiveCycle DS dodatkowo serwis Data Management. Realizacja żądania następuje w adapterze (patrz Rysunek 3).

AMF

AMF jest binarnym formatem danych służącym do serializacji obiektów ActionScript i przesyłania ich za pośrednictwem Internetu między aplikacją Flash a zdalnym serwisem. Twórcą tego protokołu jest Adobe, który jako ostatnią opublikował specyfikacje AMF 3 dla ActionScript 3. Pod linkiem: link, można znaleźć tabelę konwersji pomiędzy typami ActionScript i Java oraz konwersji odwrotnej między Java i ActionScript.

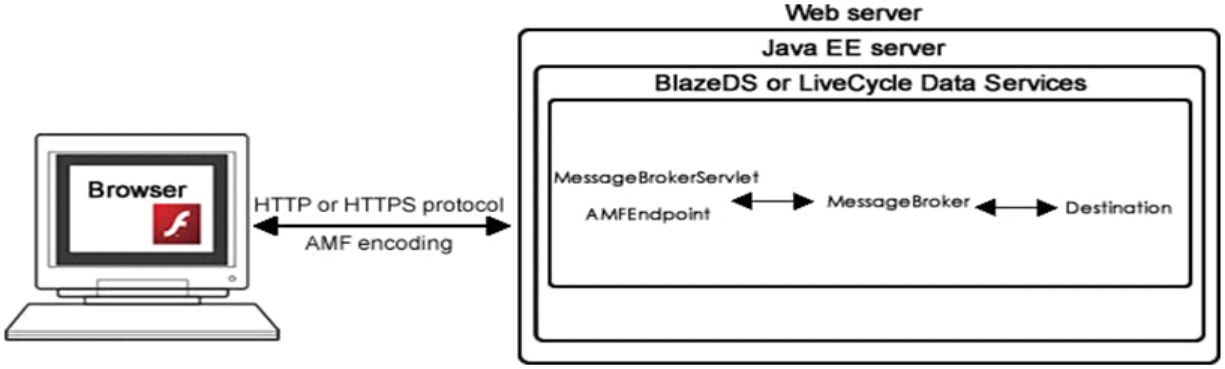
Dla obiektów użytkownika jak i dla obiektów silnie typowanych, własności z modyfikatorem public (włączając w to te zdefiniowane z metodami get i set) są serializowane i wysyłane z aplikacji Flex do serwera lub z serwera do aplikacji Flex, jako własności generalnego obiektu Object. Aby umożliwić mapowanie pomiędzy odpowiadającymi sobie obiektami po stronie klienta i serwera, należy zastosować te same nazwy własności w klasach Java i ActionScript. W dalszej kolejnośc w klasie ActionScript należy zastosować metatag [RemoteClass], aby utworzyć obiekt ActionScriptowy, który będzie bezpośrednio mapowany do obiektu Java.

Poniżej przykład klasy ActionScript o nazwie Employee, która mapuje serwerowy obiekt DTO o tej samej nazwie znajdujący się w pakiecie services.

```
package valueobjects.Employee{
  [Bindable]
      [RemoteClass(alias="services.Em-
ployee")]
  public class Employee {
    public var id:int;
    public var firstName:String;
    public var lastName:String;
    (...)
  }
}
```

Instalacja BlazeDS i LiveCycle Data Services

Chcąc używać Flash Remotingu z wykorzystaniem BlazeDS lub LiveCycle Data Services niezbędne jest zainstalowanie i skonfigurowanie

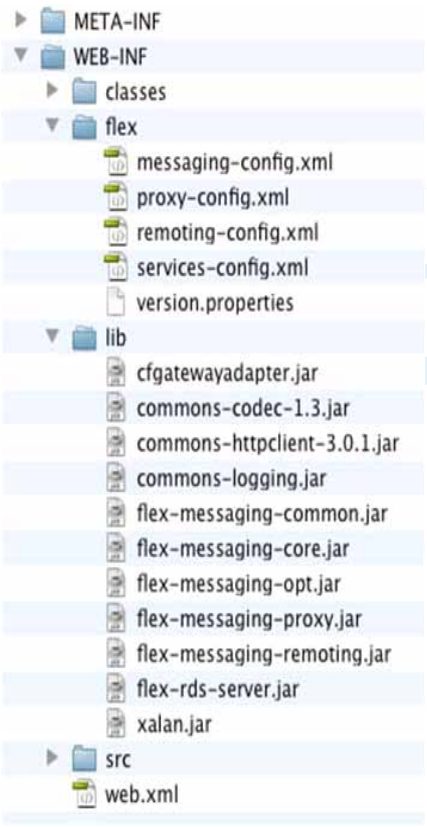


Rysunek 3. Architektura Flash Remoting.

> Chcąc używać Flash Remotingu z wykorzystaniem BlazeDS lub LiveCycle Data Services niezbędne jest zainstalowanie i skonfigurowanie dodatkowej aplikacji.

dodatkowej aplikacji na serwerze. Dla BlazeDS dostępne są do ściągnięcia wersje w postaci archiwum WAR, które można zainstalować podobnie jak aplikacje webową lub w postaci w pełni gotowego rozwiązania. W tym przypadku gotowym rozwiązaniem jest Tomcat wraz z osadzonym archiwum WAR, skonfigurowany i posiadający zbiór różnych przykładowych aplikacji. Instalator LiveCycle Data Service w podobny sposób umożliwia wybór pomiędzy instalacją rozwiązania wraz z integrowanym Tomcatem a instalacją aplikacji webowej na serwerze wybranym przez instalującego.

W takim scenariuszu na serwerze aplikacyjnym posiadać będziemy aplikację webową o nazwie blazeds lub lcds (często z dołączonym numerem wersji). Aplikacje tą można modyfikować i rozbudowywać własnym kodem, choć częstsze jest przekopiowanie JARów i konfiguracji aplikacji blazeds lub lcds do istniejącej aplikacji webowej Java.



Rysunek 4.Wymagane pliki BlazeDS lub LiveCycle Data Services.

Modyfikowanie web.xml

Chcąc przekopiować zawartość blazeds lub lcds do innej aplikacji webowej, często niezbędne okazuje się zmodyfikowanie web.xml'a, aby zdefiniować w nim funkcje nasłuchującą dla HttpFlexSession i servlet mapujący MessageBroker, który odbiera wszystkie żądania i przekazuje je do odpowiednich Javowych endpointów. Zawartość takiego web.xml'a można skopiować i przekleić z oryginalnego pliku web.xml zawartego w projekcie blazeds lub lcds.

```
<!-- Http Flex Session attribute and
binding listener support -->
<listener>
 <listener-class>flex.messaging.Http-
FlexSession</listener-class>
</listener>
<!-- MessageBroker Servlet -->
<servlet>
 <servlet-name>MessageBrokerServlet</
servlet-name>
 <display-name>MessageBrokerServlet</
display-name>
  <servlet-class>flex.messaging.Messa-
geBrokerServlet</servlet-class>
 <init-param>
   <param-name>services.configuration.
file</param-name>
     <param-value>/WEB-INF/flex/servi-
ces-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>
```

Opcjonalnie można skopiować i wkleić (i odkomentować) mapowanie dla RDSDispatchServlet, który używany jest przez funkcje Flash Builder 4 dla połączenia się i odczytania szczegółów wystawionych usług aby wygenerować przykładowy kod aplikacji klienckiej. Więcej szczegółów o technologiach modelujących Adobe można znaleźć w artykule The technologies for building Flex and Java applications i kursie Building a Flex application that connects to a BlazeDS Remoting destination

> Możliwy jest dostęp do stanowych i bezstanowych obiektów Java poprzez ustawienie atrybutu scope

using Flash Builder 4.

```
<servlet>
   <servlet-name>RDSDispatchServlet</
servlet-name>
   <display-name>RDSDispatchServlet</
display-name>
    <servlet-class>flex.rds.server.se-
rvlet.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>
```

Przegląd services-config.xml

Stosując Flash Remoting klient wysyła żądanie do serwera, gdzie jest ono przetwarzane, a następnie tworzona jest odpowiedz, która jest odsyłana do klienta. Konfiguracja żądań klienta zawarta jest w plikach services-config.xml i remoting-config.xml zlokalizowanych w katalogu /WEB-INF/flex/

Plik services-config.xml zawiera definicje różnych kanałów używanych do wysyłania żądań. Każda definicja kanału określa typ protokołu sieciowego i format wiadomości, który będzie stosowany w żądaniu i endpointu do którego ma ono dotrzeć. Javowy endpoint deserializuje wiadomość zgodnie z regułami protokołu określonego w konfiguracji i przekazuje ją w postaci Javowej do MessageBrokera, który przesyła ją dalej do właściwego serwisu docelowego (następny rozdział pokazuje jak należy go zdefiniować).

```
<channels>
 <channel-definition id="my-amf" clas-
s="mx.messaging.channels.AMFChan-
nel">
    <endpoint url="http://{server.na-
me}:{server.port}/{context.root}/
messagebroker/amf"  class="flex.messa-
ging.endpoints.AMFEndpoint"/>
  </channel-definition>
   <channel-definition  id="my-secure-
amf"    class="mx.messaging.channels.
SecureAMFChannel">
  <endpoint  url="https://{server.na-
me}:{server.port}/{context.root}/
messagebroker/amfsecure"
   class="flex.messaging.endpoints.Se-
cureAMFEndpoint"/>
  </channel-definition>
   (...)
</channels>
```

Definiowanie miejsc docelowych (ang. *destinations*)

W pliku remoting-config.xml definiowane są miejsca docelowe (nazywane mapowaniem na klasy Javowe) do których MessageBroker przekazuje komunikaty. W tagu source umieszcza się pełną (wraz z nazwą pakietu) nazwe javowej klasy POJO, która musi posiadać bezargumentowy konstruktor i być widoczna na classpath. Zwykle klasy umieszcza się w katalogu /WEB-INF/classes/ lub w archiwach JAR zlokalizowanym w /WEB-INF/lib/. Możliwy jest również dostęp do EJB i innych obiektów zarejestrowanych w Java Naming and Directory Interface (JNDI) poprzez wywoływanie metod na miejscach docelowych, które są klasami fasady serwisu. Klasy te wyszukują obiekty w JNDI i wywołują ich metody.

Możliwy jest dostęp do stanowych i bezstanowych obiektów Java poprzez ustawienie atrybutu scope na jedną z wartości: application, session lub request (wartość domyślna). Tworzenie i zarządzanie obiektami referencyjnymi po stronie serwera obsługuje BlazeDS lub LiveCycle Data Services.

```
<service id="remoting-service" clas-
s="flex.messaging.services.Remoting-
Service">
  <adapters>
     <adapter-definition id="java-ob-
ject"    class="flex.messaging.servic-
es.remoting.adapters.JavaAdapter"
default="true"/>
```

> W większości aplikacji dostęp do części lub wszystkich zasobów zgromadzonych na serwerze zastrzeżony jest dla określonych użytkowników.

```
  </adapters>
  <default-channels>
    <channel ref="my-amf"/>
  </default-channels>
  <destination id="employeeService">
    <properties>
        <source>services.EmployeeSer-
vice</source>
     <scope>application</scope>
    </properties>
  </destination>
</service>
```

Możliwe jest określenie kanału dla wybranego miejsca docelowego.

```
<destination id="employeeService "
channels="my-secure-amf">
```

Na koniec, należy użyć tych miejsc docelowych przy definiowaniu instancji RemoteObject w aplikacji Flex.

```
<s:RemoteObject id="employeeSvc" de-
stination="employeeService"/>
```

Bezpieczeństwo

W większości aplikacji dostęp do części lub wszystkich zasobów zgromadzonych na serwerze zastrzeżony jest dla określonych użytkowników. Wiele aplikacji Java EE używa kontenera zarządzania bezpieczeństwem (ang. *Container-Managed Security*), który dokonuje autentykacji (sprawdzenia tożsamości użytkownika) i autoryzacji (określenie zasobów do których użytkownik ma dostęp – często wynika to ze zdefiniowanej w systemie roli użytkownika) w oparciu o Realm, kolekcją nazw użytkowników, wraz z ich hasłami i rolami. Realm skonfigurowany jest na serwerze Java EE i może mieć postać relacyjnej bazy danych, usługi katalogowej LDAP, dokumentu XML lub wykorzystywać dedykowany framework do autentykacji i autoryzacji.

Aby zintegrować aplikacje Flex z Javowym frameworkiem bezpieczeństwa, w celu odpowiedniego ograniczenia dostępu do zasobów serwera, należy dodać wpisy do plików konfiguracyjnych BlazeDS lub LiveCycle Data Services (szczegóły poniżej), a następnie, jak to ma miejsce w większości tworzonych aplikacji Flex, stworzyć formularz, który zostanie wypełniany danymi uwierzytelniającymi przez użytkownika i przesyłany do serwera w celu autentykacji. Dane uwierzytelniające użytkownika są przekazywane do serwera automatycznie wraz z wszystkimi późniejszymi żądaniami.

Modyfikacja services-config.xml

W pliku *services-config.xml* znajdującym się w BlazeDS lub LiveCycle Data Services należy zdefiniować *login command* dla stosowanego serwera aplikacji wewnątrz taga *security.* BlazeDS i LiveCycle Data Services dostarczają następujące *login command:* TomcatLoginCommand (dla Tomcata jak i dla Jbossa), JRunLoginCommand, WeblogicLoginCommand, WebSphereLoginCommand, OracleLoginCommand. Wszystkie one są zdefiniowane w pliku XML, w którym wystarczy jedynie odkomentować właściwy wpis.

Często pojawia się potrzeba zdefiniowania wymogów bezpieczeństwa (tag security-constraint), do czego można zastosować autentykację Basic lub Custom i jeśli jest to pożądane, jedną lub więcej ról. Do stworzenia autentykacji Custom przy użyciu Tomcat lub JBoss, często niezbędne jest dodanie dodatkowych klas do aplikacji webowej w celu integracji z frameworkiem bezpieczeństwa wykorzystywanym przez serwer aplikacji Java EE i zmodyfikowanie kilku plików konfiguracyjnych. Szczegółowe przykłady znajdują się pod następującym linkiem: link.

```
<services-config>
  <security>
    <login-command class="flex.messag-
ing.security.TomcatLoginCommand"
    server="Tomcat">
     <per-client-authentication>false</
per-client-authentication>
  </login-command>
  <security-constraint id="trusted">
    <auth-method>Custom</auth-method>
    <roles>
      <role>employees</role>
      <role>managers</role>
    </roles>
```

> Możliwe jest również zdefiniowanie domyślnych wymogów bezpieczeństwa dla wszystkich miejsc docelowych

```
  </security-constraint>
 </security>
 ...
</services-config>
```

Modyfikacja remoting-config.xml

W następnej kolejności, w definicji miejsca docelowego, niezbędne jest dodanie odniesienia do uprzednio zdefiniowanych wymogów bezpieczeństwa:

```
<destination id="employeeService">
 <properties>
  <source>services.EmployeeService</
source>
 </properties>
 <security>
 <security-constraint ref="trusted"/>
 </security>
</destination>
```

Możliwe jest również zdefiniowanie domyślnych wymogów bezpieczeństwa dla wszystkich miejsc docelowych i/lub ograniczeń dostępu jedynie dla wybranych metod, które będą stosowały inne wymogi bezpieczeństwa.

Domyślny kanał *my-amf* używa HTTP. Możliwa jest zmiana jednego lub więcej miejsc docelowych tak aby używały kanału *my-secure-amf* wykorzystującego HTTPS:

```
<destination id="employeeService">
 <channels>
  <channel ref="my-secure-amf"/>
 </channels>
 ...
</destination>
```

Poniżej definicja my-secure-amf znajdujaca się w services-config.xml.

```
<!-- Non-polling secure AMF -->
<channel-definition id="my-secure-amf"
   class="mx.messaging.channels.Secu-
reAMFChannel">

  <endpoint     url="https://{server.
name}:{server.port}/{context.root}/
messagebroker/amfsecure"
   class="flex.messaging.endpoints.Se-
cureAMFEndpoint"/>
</channel-definition>
```

Rozbudowa aplikacji Flex

Spójrzmy na aplikacje od strony serwera. Jeśli nasza aplikacja wykorzystuje autentykacje Custom, niezbędne jest stworzenie w aplikacji Flex formularza, który służy do wprowadzania przez użytkownika loginu i hasła, który przekazuje je do serwera poprzez wywołanie metody ChannelSet.login() i nasłuchuje zdarzeń result i fault. Pojawienie się zdarzenia result oznacza, że logowanie zakończyło się pomyślnie, zdarzenie fault oznacza niepowodzenie. Wprowadzone dane uwierzytelniające wykorzystywane są we wszystkich serwisach podłączonych za pomocą tego samego ChannelSet. Dla podstawowej autentykacji nie ma potrzeby dodawania czegokolwiek do aplikacji Flex. Otwiercie dialogu do logowania następuje w przeglądarce w sytuacji gdy aplikacja po raz pierwszy próbuje się połączyć do miejsca docelowego.

Zabezpieczona aplikacja może wysłać żądanie Flash Remoting do miejsca docelowego podobnie jak to było wykonywane wcześniej, z ta różnicą, że dane uwierzytelniające użytkownika są wysyłane automatycznie z każdym wysyłanym żądaniem (dla obu typów autentykacji Basic i Custom). Jeśli miejsce docelowe lub metoda miejsca docelowego wymaga innego poziomu uprawnień niż ma zalogowany użytkownik wywołanie zwróci zdarzenie fault. W celu usunięcia danych uwierzytelniających i wylogowania użytkownika należy wywołać metodę ChannelSet.logout().

Arcitektura aplikacji Flex

Po zapoznaniu się z Flash Remoting od strony serwera oraz sposobem w jaki tworzymy instancje RemoteObject w Flex przyjrzyjmy się sposobowi w jaki można zbudować aplikacje aby wykorzystać te obiekty.

Stosowanie zdarzeń (ang. *events*)

Typowa aplikacja Flex składa się z kodu MXML tworzącego interfejs użytkownika i kodu ActionScript zawierającego logikę. Ta para tech-

> Data binding jest mocną stroną frameworka Flex

nologii, tak jak ma to miejsce w przypadku JavaScript i przeglądarki obiektów DOM, jest powiązana za pomocą zdarzeń i procedur ich obsługi. Aby użyć w aplikacji *RemoteObjectów,* należy stworzyć ich instancje, wywołać zdalne metody miejsc docelowych, zdefiniować metody odbierające i przetwarzające otrzymane z serwera zdarzenia *result i fault*.

Poniżej przestawiony jest przykład prostej aplikacji, w której dane o pracownikach są wyszukiwane w bazie danych i wyświetlane w komponencie Flex DataGrid. Po zainicjalizowaniu aplikacji, następuje wywołanie metody getEmployees() z miejsca docelowego employeeService zdefiniowanego w pliku remoting-config.xml i jeśli uda się poprawnie odczytać dane z serwera następuje wypełnienie zmiennej employees , w przeciwnym wypadku wiadomość o niepowodzeniu zostaje wyświetlona w Alert box, nie jest istotne z jakiej przyczyny odczyt się nie powiódł. Do powiązania zmiennej employees do własności dataProvider DataGridu stosuje się Data binding .

```
<s:Application      xmlns:fx="http://
ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/
flex/spark"
    xmlns:mx="library://ns.adobe.com/
flex/mx"
   initialize="employeeSvc.getEmploy-
ees()">
   <fx:Script>
      <![CDATA[
         import mx.collections.Array-
Collection;
         import mx.controls.Alert;
          import mx.rpc.events.Faul-
tEvent;
          import mx.rpc.events.Resul-
tEvent;
        [Bindable]private var employ-
ees:ArrayCollection;
         private function onResult(e-
:ResultEvent):void{
            employees=e.result as Ar-
rayCollection;
         }
          private function onFault(e-
:FaultEvent):void{
           Alert.show("Error retrie-
ving data.","Error");
         }
      ]]>
   </fx:Script>
   <fx:Declarations>
     <s:RemoteObject id="employeeSvc"
destination="employeeService"
             result="onResult(event)"
fault="onFault(event)" />
   </fx:Declarations>
     <mx:DataGrid  dataProvider="{em-
ployees}"/>
</s:Application>
```

Stosując RemoteObject możliwe jest definiowanie procedur obsługi zdarzeń result i fault na poziomie serwisu:

```
<s:RemoteObject      id="employeeSvc"
destination="employeeService"
result="onResult(event)"
fault="onFault(event)"/>
```

lub na poziomie metody:

```
<s:RemoteObject      id="employeeSvc"
destination="employeeService">
    <s:method    name="getEmployees"
result="onResult(event)"
fault="onFault(event)"/>
   <s:method    name="getDepartments"
result="onResult2(event)"
fault="onFault2(event)"/>
</RemoteObject>
```

lub dla każdego wywołania:

```
<s:Application      xmlns:fx=http://
ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/
flex/spark"
    xmlns:mx="library://ns.adobe.com/
flex/mx"
   initialize="getEmployeesResult.to-
ken=employeeSvc.getEmployees()">
  <fx:Declarations>
    <s:RemoteObject  id="employeeSvc"
destination="employeeService"/>
     <s:CallResponder  id="getEmploy-
eesResult" result="onResult(event)"
    fault="onFault(event)"/>
  </fx:Declarations>
```

Zastosowania data bindingu

Data binding jest mocną stroną frameworka Flex pozwalającą zaktualizować interfejs użytkownika, w sytuacji gdy wyświetlane dane ulegną zmianie, bez jawnego tworzenia w tym

> Wraz z rozrostem aplikacji niezbędne okaże się porozdzielanie logiki na pakiety klas ActionScriptu i widoków na oddzielne pliki MXML

celu rejestrów i obserwatorów zdarzeń (ang. listeners). We wcześniej zaprezentowanym kodzie, tag [Bindable] poprzedzający deklarację zmiennej jest dyrektywą kompilatora, która definiuje zachowanie translatora podczas kompilacji pliku, skutkuje ona automatyczną generacją kodu ActionScript, dzięki któremu zdarzenia są rozgłaszane za każdym razem, gdy zmienna employees ulegnie zmianie.

```
[Bindable]private var employees:ArrayCollection;
```

Nawiasy klamrowe wystepujące w przypisaniu dla własności dataProvider w deklaracji DataGridu realnie generują kod, który odpowiada za nasłuchiwanie zmian zmiennej employees i właściwe aktualizowanie widoku DataGridu.

```
<mx:DataGrid dataProvider="{employees}"/>
```

W tej aplikacji zmienna employees jest inicjalizowana wartością null, a DataGrid pozostaje pusty, jednak jak tylko zmienna employees zostanie wypełniona danymi otrzymanymi z serwera, widok DataGridu zostanie zaktualizowany i wyświetli otrzymane dane.

Zastosowanie statusów widoku

Chcąc dokonać bardziej radykalnych zmian w interfejsie użytkownika podczas działania aplikacji, na przykład chcąc dodać, usunąć, przenieść lub zmodyfikować komponenty, należy użyć statusów widoku (ang. view status). Dla każdego widoku lub komponentu Flex możliwe jest zdefiniowanie różnych statusów, a następnie zdefiniowanie statusu(-ów) dla każdego obiektu w widoku, jakie powinien zawierać ten obiekt oraz wyglądu i zachowania które powinno być powiązane z danym statusem. Zmiany statusów odbywają się poprzez ustawianie własności currentState na nazwę jednego ze zdefiniowanych statusów.

```
<s:states>
 <s:State name="employees"/>
 <s:State name="departments"/>
</s:states>
<mx:DataGrid dataProvider="{employees}" includeIn="employees"/>
<s:Button label.employees="Switch to departments"
 label.departments="Switch to employees"
   click.employees="currentState='departments'"
     click.departments="currentState='employees'"/>
```

Zastosowanie komponentów MXML

Wraz z rozrostem aplikacji niezbędne okaże się porozdzielanie logiki na pakiety klas ActionScriptu i widoków na oddzielne pliki MXML (nazywane komponentami MXML). Każdy komponent MXML rozszerza istniejący komponent i może być jedynie włączony do aplikacji, ale nie może być uruchomiony samodzielnie. Chcąc użyć komponentu w MXML'u należy zainstancjonować komponent (nazwa jego klasy jest taka sama jak nazwa pliku w którym się znajduje) oraz dołączyć właściwą przestrzeń nazw (ang. *namespace*) tak aby kompilator był wstanie go zlokalizować.

Poniżej przedstawiono kod MXML'owego komponentu MasterView zapisanego w pliku MasterView.mxml zlokalizowanego w pakiecie *com.adobe.samples.views*.

```
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/flex/spark" >
  <fx:Metadata>
   [Event(name="masterDataChange",type="flash.events.Event")]
  </fx:Metadata>
  <fx:Script>
   <![CDATA[
    import mx.collections.ArrayList;
            [Bindable]private var masterData:ArrayList=new ArrayList(["data1", "data2", "data3"]);
    public var selectedData:String;
                private function onChange(e:Event):void{
       selectedData=dataList.selectedItem;
              this.dispatchEvent(new Event("masterDataChange"));
    }
```

> Luźno powiązane komponenty, jak ten nasz przykładowy, które definiują i rozgłaszają własne zdarzenia są podstawą modułowej konstrukcji aplikacji Flex.

```
  ]]>
 </fx:Script>
    <s:DropDownList    id="dataList"
dataProvider="{masterData}"
change="onChange(event)"/>
</s:Group>
```

Natomiast ten kod przedstawia przykład instancjonowania i używania uprzednio stworzonego przez nas komponentu MasterView.

```
<s:Application      xmlns:fx="http://
ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/
flex/spark"
      xmlns:views="com.adobe.samples.
views.*">
   <fx:Script>
      <![CDATA[
        import mx.controls.Alert;
         private function onMasterDa-
taChange(e:Event):void{
          Alert.show(e.currentTarget.
selectedData,"Master data changed");
        }
      ]]>
   </fx:Script>
    <views:MasterView masterDataChan-
ge="onMasterDataChange(event)"/>
</s:Application>
  <views:MasterView masterDataChange=
"onMasterDataChange(event)"/>
</s:Application>
```

Rozgłaszanie zdarzeń

Aby stworzyć luźno powiązane komponenty, niezbędne jest zdefiniowanie publicznego API dla komponentu (z zastosowaniem modyfikatorów public) i/lub zdefiniować i rozgłosić stworzone przez siebie zdarzenia jak to pokazano powyżej w przykładowym kodzie MasterView. Metatag [Event] jest stosowany do definiowania tych zdarzeń jako części API komponentu i określenia jaki typ zdarzenia jest przez niego rozgłaszany.

```
<fx:Metadata>
 [Event(name="masterDataChange",type=
"flash.events.Event")]
</fx:Metadata>
```

Kiedy jakieś zdarzenie pojawia się w komponencie (przykładowo dla DropDownList zdarzenie change), komponent ten tworzy instancje zdarzenia o typie określonym w metadanych i rozgłasza go.

```
this.dispatchEvent(new Event(„master-
DataChange"));
```

Kod który instancjonuje stworzony przez nas komponent może w tym samym momencie zarejestrować się do nasłuchiwania zdarzenia naszego komponentu oraz zarejestrować procedurę obsługi zdarzeń (ang. *event handler*).

```
<views:MasterView   masterDataChange-
="onMasterDataChange(event)"/>
```

Luźno powiązane komponenty, jak ten nasz przykładowy, które definiują i rozgłaszają własne zdarzenia są podstawą modułowej konstrukcji aplikacji Flex. W rzeczywistości jest to właśnie sposób w jaki budowane są komponenty w frameworku Flex. Więcej informacji o rozgłaszaniu własnych zdarzeń można znaleźć w video Define events in Flex 4 with Flash Builder 4.

Tworzenie modułów

Domyślnie zostało przyjęte, że cały kod stworzony przez programistę jest kompilowany do jednego pliku SWF. Jeśli wynikowy plik SWF zacznie osiągać zbyt duże rozmiary, albo zawiera on funkcjonalność przeznaczona jedynie dla określonej grupy użytkowników, możliwe jest zastosowanie modułów do podziału aplikacji na większa liczbę plików SWF, które mogą być załadowywane i usuwane dynamicznie przez główną aplikację w czasie jej działania. W celu stworzenia modułu, niezbędne jest stworzenie klasy (ActionScript lub MXML) rozszerzającej klasę Module a następnie skompilowanie jej. Chcąc dynamicznie załadować moduł do aplikacji w czasie jej działania należy użyć tagu <mx:ModuleLoader> lub metod klasy ModuleLoader.

Zastosowanie mikroarchitektury

Przedstawiona wiedza o budowie aplikacji Flex to podstawy tej technologi, którą wraz z rozrostem aplikacji będzie trzeba wzbogacić o metodologie organizacji plików, centralizacji

> Przedstawiona wiedza o budowie aplikacji Flex to podstawy tej technologi

danych i data serwisów oraz zapewnienia komunikacji miedzy wszystkimi komponentami. W tym calu budując aplikacje Flex można zastosować wszystkie wzorce projektowe, które przez lata sprawdzały się przy tworzeniu aplikacji biznesowych. Tak naprawdę wiele konkretnych mikroarchitektur było i nadal jest rozwijanych. Najstarszą i najbardziej rozwiniętą jest Cairngorm, open source'owa mikroarchitektura, która stosuje polecenia (ang. command) i delegacje, wzorzec Front Controller, model i usługę obsługi danych opartą o singleton oraz dyspozytora zdarzeń (ang. event dispatcher). Innymi popularnie stosowanymi frameworkami są Pure MVC, Mate, Parsley, Swiz i Spring ActionScript. Więcej informacji na temat tych i innych frameworków można znaleźć na stronach poświęconych architekturze Flex w Adobe Developer Center.

Gdzie szukać więcej informacji

Ten artykuł przedstawia architekturę aplikacji Flex i Java. Dodatkowe informacje na ten temat można znaleźć na stronach do których prowadzą linki w tekście oraz te poniższe:

- Adobe Flex Developer Center
- Flex and Java on the Adobe Developer Center
- Flex Architecture on the Adobe Developer Center
- BlazeDS documentation
- Adobe LiveCycle Data Services ES2 documentation
- Adobe Flex 4 documentation

O autorze

Jeanette Stallons jest niezależnym trenerem Flexa oraz konsultantem uczącym w wielu firmach, w tym Adobe, Oracle, Boeing, Wachovia, Morgan Stanley, Charles Schwab. Przed rozpoczęciem własnej działalności Jeanette pracowała w Allaire, Macromedia, a następnie w Adobe w dziale szkoleń, projektując i tworząc aplikacje Flash, Flex i inne produkty oraz przeprowadzała szkolenia w ich zakresie. Jej najnowszy projekt to aplikacja flexowa do pomocy w nauce o Flexie (Adobe Flex Learning Paths Flex application), za którą od początku była odpowiedzialna jako programista i ekspert.

Notatki o testowaniu:

WebDriver – łatwe i przyjemne testowanie aplikacji webowych

Bartosz Majsak

Każdy, kto ma choćby minimalne doświadczenie z testami doskonale zdaje sobie sprawę, że nawet najbardziej solidne testy jednostkowe nie dają stuprocentowej pewności poprawnego działania aplikacji. Często musimy spojrzeć na aplikację oczami końcowego użytkownika i zweryfikować, czy wszystkie zaimplementowane i przetestowane jednostkowo funkcjonalności są dla niego dostępne i działają tak jak sobie tego życzy.

W drugiej części serii „Notatki o testowaniu" chciałbym zaprezentować Wam narzędzie *WebDriver*, które pozwala tworzyć automatyczne testy funkcjonalne aplikacji internetowych. W połączeniu z poznanym już *easyb* sprawia, że pisanie testów to czysta przyjemność.

Co oferuje WebDriver?

WebDriver oferuje dwie kluczowe funkcjonalności, które czynią to narzędzie wartym uwagi. Z jednej strony to bardzo wygodne dla programisty API pozwalające na interakcję z przeglądarką, z drugiej zaś to koncepcja sterowników (ang. driver), które tę bezpośrednią komunikację umożliwiają. W tej sekcji przyjrzymy się obu tym aspektom. W drugiej części artykułu wykorzystamy przedstawioną tu wiedzę do stworzenia prostego scenariusza testowego.

WebDriver oferuje obsługę następujących przeglądarek:

- Internet Explorer,
- Firefox,
- Chrome.

W fazie eksperymentalnej dostępna jest także integracja z iPhone i z telefonami z systemem Android.

Dodatkowo otrzymujemy także *HtmlUnitDriver*, czyli implementację korzystającą z *„bezokienkowej"* przeglądarki napisanej w całości w Javie. Zyskujemy zatem przy jej pomocy możliwość uruchamiania testów na wielu platformach. To co odróżnia *WebDrivera* od innego znanego narzędzia jakim jest *Selenium*, to fakt, że ten pierwszy dostarcza natywnego dostępu do przeglądarki, drugi zaś opiera się o warstwę pośredniczącą jaką jest JavaScript. W przypadku *WebDrivera* oznacza to wykorzystanie technologii, które dają największe możliwości interakcji z przeglądarka. Jest to oczywiście dodatkowy wysiłek dla programistów tej biblioteki, ponieważ muszą rozwijać i utrzymywać tyle różnych implementacji ile wspieranych przeglądarek. Na przykład dla przeglądarki Internet Explorer natywny dostęp oznacza wykorzystanie C++, komponentów COM i gimnastyki z API stworzonym przez programistów z Redmond. Dla Firefox napisana została specjalna wtyczka.

Interfejs programistyczny oferowany przez *WebDrivera* jest minimalistyczny i intuicyjny zarazem. Dostarcza podstawowych operacji, które wykonuje na co dzień użytkownik przeglądarki. Zanim jednak przejdziemy do ich omówienia warto wspomnieć w jaki sposób możemy wyszukiwać elementy dostępne na stronie. *WebDriver* oferuje nam następujące metody:

- na podstawie unikalnego identyfikatora (id),
- określonej klasy CSS,
- nazwie elementu HTML (tagu),
- nazwie (atrybutu name),
- selektorów CSS3,
- tekstu bądź jego fragmentu znajdującego się w elemencie.

Dwa główne interfejsy, z których korzystać będziemy najczęściej podczas tworzenia testów korzystających z *WebDrivera*, to `WebElement` oraz `WebDriver`. Pierwszy z nich umożliwia interakcję z elementami dostępnymi na stronie. Tak więc możemy na przykład:

- kliknąć w wybrany element (o ile to ma

> Interfejs programistyczny oferowany przez *WebDrivera* jest minimalistyczny i intuicyjny zarazem.

sens),

- dowiedzieć się z jakim elementem HTML mamy do czynienia, jakie posiada atrybuty itd.,
- jeśli jest polem tekstowym możemy w nim coś napisać bądź wyczyścić jego zawartość,
- wyszukać elementy znajdujące się w poddrzewie DOM, dla którego korzeniem jest dany element,
- wysłać zawartość formularza.

Warto podkreślić tutaj, że API to ma charakter blokujący. Oznacza to, że każda operacja skutkująca przeładowaniem strony spowoduje zatrzymanie wykonywania testu i oczekiwanie na zakończenie tej operacji w przeglądarce. Oczywiście dostępny jest także deklaratywny mechanizm oczekiwania na jakieś zdarzenie, który może być szczególnie pomocny przy testowaniu silnie AJAXowych aplikacji.

`WebDriver`, czyli drugi kluczowy interfejs narzędzia o tej samej nazwie, oferuje nam z kolei operacje typowe dla przeglądarki jako programu. Możemy zatem załadować wybraną stronę wpisując jej URL w pasek adresowy, przełączać się pomiędzy oknami, używać funkcji „Wstecz" i „Dalej", odświeżać stronę a nawet manipulować ciasteczkami.

Bardzo ciekawym pomysłem promowanym przez twórców tego narzędzia jest wzorzec *Page Object*. Najprościej rzecz ujmując jest to klasa enkapsulująca fragment (kontrolkę) strony i dająca możliwość interakcji z nią poprzez przejrzysty zestaw metod. Jest to nie tylko zaleta w postaci czytelnego kodu, ale także możliwość wykorzystywania tak przygotowanych fragmentów w wielu scenariuszach testowych. Przykładowy scenariusz stworzony na potrzeby tego artykułu nie tylko bazuje na tej koncepcji, ale również ilustruje w jaki sposób API *WebDrivera* ją wspiera.

Funkcjonalność, której brakować może wielu programistom, a w szczególności testerom, jest brak odpowiednika SeleniumIDE – narzędzia do nagrywania interakcji użytkownika z przeglądarką, które może ułatwić tworzenie testów i zwiększyć produktywność. Z drugiej strony programista wyposażony w takie dodatki Firefox jak Firebug i XPather (lub XPath Checker) nie powinien mieć wielkich trudności w implementowaniu nawet najbardziej wymyślnych scenariuszy testowych. Twórcy *WebDriver* wyszli jednak naprzeciw temu ograniczeniu dostarczając adapter API *Selenium*. Dzięki temu wciąż możemy używać stworzone wcześniej za pomocą SeleniumIDE testy, ciesząc się przy tym z możliwości jakie oferuje nam *WebDriver*.

Scenariusz testowy

Posiadamy już podstawową wiedzę na temat możliwości jakie daje nam *WebDriver*, możemy zatem bez obaw przejść do praktycznego jej zastosowania. W tym celu stworzymy prosty scenariusz testowy:

- Użytkownik wchodzi na stronę http://dzone.com,
- wyszukuje skatalogowane strony zawierające frazę *„java express"*,
- oczukuje, że na liście rezultatów dostępny jest link do strony magazynu Java exPress.

Korzystając z poznanego już *easyb* stwórzmy zatem wstępną formę scenariusza:

```
scenario "Searching for Java exPress magazine link on dzone.com", {
   given "user is on dzone.com main page"
   when "he searches for 'java express'"
   then "link to the javaexpress.pl website should be present in the results list"
}
```

Dla uproszczenia zweryfikujemy, że wśród wyświetlonych rezultatów znajduje się taki, którego tytuł to *„Free Java exPress newsletter available"*.

> Bardzo ciekawym pomysłem promowanym przez twórców tego narzędzia jest wzorzec *Page Object*.

Rys 1. Strona główna dzone.com. W prawym górnym rogu zaznaczona została wyszukiwarka, którą użyjemy w naszym teście.

Patrząc z perspektywy użytkownika musimy wykonać dwie czynności:

1. Wpisać w polu tekstowym frazę „java express”.
2. Nacisnąć przycisk z lupą (bądź klawisz enter) w celu uruchomienia wyszukiwarki.

Mając dostęp do tych dwóch elementów strony z poziomu API *WebDriver* powyższe czynności możemy zaimplementować w dwóch linijkach kodu:

```
searchBox.sendKeys("\"java
express\"");
searchButton.click();
```

Oba te elementy dostępne są dzięki odpowiednim atrybutom `id`, co potwierdza poniższy listing:

```
<div id="mh_search">
  <form method="get" action="/links/
search.html" id="headerSearch">
    <input type="text" size="20"
name="query" id="mh_searchQuery" />
    <input type="image" id="mh_search-
Submit" alt="Submit"/>
  </form>
</div>
```

Klasa realizująca wzorzec *Page Object*, która umożliwiałaby użycie wyszukiwarki dostępnej na stronie *dzone.com* mogłaby mieć następującą postać:

```
public class DzoneMainPage {

  private static final String
    DZONE_PAGE = "http://dzone.com";

  // Pole typu WebDriver umożliwia
  // interakcję z wyświetlanymi przez
  // przeglądarkę stronami.
  private final WebDriver driver;

  // Element odpowiadający polu
  // tekstowemu zlokalizowany będzie
  // po ID
  @FindBy(how = How.ID,
```

> Funkcjonalność, której brakować może wielu programistom, a w szczególności testerom, jest brak odpowiednika SeleniumIDE

```
  using = "mh_searchQuery")
private WebElement searchBox;

// Element odpowiadający
// przyciskowi uruchamiającemu
// wyszukiwanie zlokalizowany
// będzie po ID
@FindBy(how = How.ID,
  using = "mh_searchSubmit")
private WebElement searchButton;

public DzoneMainPage(
    WebDriver driver) {
  this.driver = driver;
  // Strona http://dzone.com
  // zostanie załadowana przez
  // przeglądarkę
  driver.get(DZONE_PAGE);
}

public DzoneSearchResultPage
    searchFor(String query) {
  searchBox.sendKeys(query);
  searchButton.click();
  // Wyszukiwarka przenosi nas na
  // stronę z rezultatami. Dzięki
  // PageFactory mamy dostęp do
  // instancji klasy
// DzoneSearchResultPage,
    // której pola WebElement zostały
// właściwie zainicjalizowane.
    return PageFactory.
      initElements(driver,
        DzoneSearchResultPage.class);
  }
}
```

Wynik działania wyszukiwarki to strona z listą artykułów pasujących do podanej frazy (rysunek 2).

```
<div class="details">
  ...
  <h3><a href="..." rel="bookmark">
    Free Java exPress
    newsletter available
  </a></h3>
  ...
</div>
```

Kod przedstawiony na powyższym listingu podpowiada nam w jaki sposób możemy zlokalizować na stronie wynikowej wszystkie tytuły artykułów. Wykorzystując XPath możemy

Rys 2. Strona z rezultatami wyszukiwania „java express”. W zielonej ramce wyróżniony został tytuł, którego obecność weryfikujemy w scenariuszu testowym.

> Stworzyliśmy kompletny mechanizm emulujący interakcję użytkownika ze stroną *dzone.com*

zapisać to w następujący sposób:

```
//div[@class='details']/h3/a
```

Powyższe wyrażenie mówi mniej więcej tyle – znajdź wszystkie elementy <a>, których rodzicem jest <h3>, którego z kolei rodzicem jest posiadający klasę details. To wystarczy, aby stworzyć klasę `DzoneSearchResultPage`, którą wykorzystamy w testach.

```
public class DzoneSearchResultPage {

  // Wyrażenie XPath pozwalające na
  // zlokalizowanie wszystkich
  // tytułów na stronie rezultatów.
  private static final String
    SEARCH_RESULT_TITLES_XPATH =
      "//div[@class='details']/h3/a";

  private final WebDriver driver;

  public DzoneSearchResultPage(
      WebDriver driver) {
    super();
    this.driver = driver;
  }

  public List<String>
      getSearchResultTitles() {
    // Elementy pobierać można
    // również korzystając
    // bezpośrednio z drivera
    List<WebElement> elements =
      driver.findElements(By.xpath(
        SEARCH_RESULT_TITLES_XPATH));
    // Wygodny sposób transformacji
    // kolekcji obiektów z jednego
    // typu na drugi, dostępny w
    // bibliotece Google Collections.
    List<String> results = Lists.
      transform(elements,
        new ExtractText());
    return results;
  }

  /**
   * Klasa wykorzystana do
   * transformacji obiektów klasy
   * WebElement na String.
   */
  private final class ExtractText
    implements Function<WebElement,
      String> {
    @Override
    public String apply(
        WebElement from) {
      return from.getText();
    }
  }
}
```

Stworzyliśmy kompletny mechanizm emulujący interakcję użytkownika ze stroną *dzone.com*, możemy zatem uzupełnić nasz scenariusz *easyb*:

```
before "Start WebDriver", {
  // Przed wykonaniem scenariuszy
  // uruchamiamy przeglądarkę.
  // W tym przypadku będzie to
  // Mozilla Firefox.
  driver = new FirefoxDriver()
}

scenario "Searching for Java exPress
magazine link on dzone.com", {

  given "user is on dzone.com main
page", {
    mainPage = PageFactory.
      initElements(driver,
        DzoneMainPage.class)
  }

  when "he searches for 'java ex-
press'", {
    searchResultPage = mainPage.
      searchFor("\"java express\"")
  }

  then "link to the javaexpress.pl
site should be present in the results
list", {
    titles = searchResultPage.
      getSearchResultTitles()
    titles.shouldHave "Free Java ex-
Press newsletter available"
  }
}

after "Close WebDriver", {
  // Po wykonaniu wszystkich
  // scenariuszy zamykamy
  // przeglądarkę.
  driver.close()
}
```

Teraz wystarczy już tylko uruchomić scenariusz z poziomu Anta, Mavena bądź pluginu dla środowiska Eclipse czy IntelliJ IDEA. WebDriver zajmie się za nas całą resztą.

> począwszy od drugiej wersji *Selenium*, *WebDriver* staje się jego integralną częścią.

Podsumowanie

Artykuł ten prezentuje jedynie namiastkę możliwości *WebDrivera*. Biblioteka ta oferuje znacznie więcej – jak chociażby wsparcie dla AJAX oraz innych języków (Python i Ruby), separację pomiędzy przeglądarką a systemem, na którym uruchamiane są testy (moduł RemoteWebDriver), robienie zrzutów ekranu czy wsparcie dla mechanizmu drag&drop. Warto tutaj wspomnieć, że *WebDriver* stosowany był do testów aplikacji Google Wave, a programiści tego projektu aktywnie uczestniczyli w jego rozwoju.

Oczywiście, jak każde dzieło ludzkie, *WebDriver* też nie jest pozbawiony wad. Przede wszystkim największym wyzwaniem jest zapewnienie działania sterowników przeglądarek w jak największej liczbie systemów operacyjnych. Problemy z ewaluacją wyrażeń XPath oraz szybkością działania testów, które z nich korzystają to kolejna zmora, szczególnie jeśli wymogiem stawianym naszej aplikacji jest wsparcie starszych wersji Internet Explorera. Mądrym podejściem wydaje się zatem wykorzystywanie tej techniki tylko w skrajnych przypadkach. Z własnego doświadczenia wiem także, że testowanie w przeglądarce z niebieskim E w logotypie nie zawsze gwarantuje nam powtarzalność testów. Strona często ładuje się zbyt długo, na przykład z bliżej niewyjaśnionych przyczyn oczekując w nieskończoność na obrazek, który tak naprawdę jest już widoczny. Powoduje to wyjątki *WebDrivera* i dominację czerwonego koloru w raportach z testów. Nie jest to jednak najprawdopodobniej ułomność omówionego w tym artykule narzędzia, a raczej problemy samej przeglądarki bądź konfiguracji systemu.

Na koniec pozostaje mi tylko wytłumaczenie, dlaczego w tym artykule brakuje dokładnego porównania *WebDrivera* z chyba najbardziej znanym narzędziem do automatycznych testów funkcjonalnych aplikacji webowych, czyli *Selenium*. Powód jest niezwykle prosty – twórcy obu projektów postanowili połączyć siły i począwszy od drugiej wersji *Selenium* ,*WebDriver* staje się jego integralną częścią. Oba narzędzia już wprowadziły wiele innowacji, aż strach pomyśleć co będzie owocem prac tego tandemu :)

Źródła

http://webdriver.googlecode.com/ - strona projektu WebDriver.

http://code.google.com/p/google-collections/ - strona projektu Google Collections

http://www.youtube.com/watch?v=tGu1ud7hk5I – prezentacja z Google Test Automation Conference.

http://code.google.com/p/bmajsak-javaexpress/ - projekt stworzony na potrzeby tego artykułu

Mistrz Programowania: Refaktoryzacja, cz. IV

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

Końcowe porządki czyli refaktoryzacja: Zmień implementację algorytmu

Być może dla osób, które znają lub posługiwały się wzorcem iteratora, to rozwiązanie nie wygląda najlepiej. Nawet jeśli wzorzec iteratora jest ci obcy, jest kilka rzeczy na które warto zwrócić uwagę:

1. stworzony iterator ma pewien cykl życia — najpierw musi być wykonana metoda `moveToNextWord`, dopiero wtedy ma uzasadnienie `hasNextWord`, a na końcu należy wywołać metodę `dispose`;

2. iterator działa z pewnym przesunięciem — musimy zapamietywać element zwracany przez iterator, żeby uzyskać pożądany efekt (`currentWord = pageIterator.moveToNextWord()`);

3. metoda `hasNextWord` posiada parametr, który należy dostarczyć — oznacza to, że klient częściowo przejmuje odpowiedzialność za pewne elementy stanu iteratora.

Są to efekty uboczne, które nieco zaciemniają kod, poza tym utrudniają jego utrzymanie, gdyż należy znać specyfikę działania klasy `PageIterator`. Mimo że jest to rozwiązanie znacząco czytelniejszego od poprzedniego, cały czas wymaga poprawek. Solą w oku jest w zasadzie metoda `hasNextWord`, która określa, czy istnieją jeszcze słowa, do których jest dostęp przez iterator. Nie dość, że potrzebuje parametru z zewnątrz, to przy obecnej konstrukcji istnieje konieczność wywołania przynajmniej jeden raz metody `moveToNextWord`.

Co zatem można zrobić? Na pewno warto pozbyć się parametru z metody `hasNextWord`, gdyż to iterator powinien wiedzieć, czy są jeszcze dostępne słowa. Ponadto metoda ta powinna być w stanie odczytać wiersze strony HTML w przód, aby stwierdzić, czy istnieje jeszcze jakieś słowo (parametr ten nie powinien pochodzić z zewnątrz). Żeby osiągnąć ten efekt mamy między innymi takie opcje:

- obsłużyć przeglądanie w przód samodzielnie, tworząc pewien mechanizm buforujący;

- skorzystać z metod `mark` i `reset` z klasy `BufferedReader` do podejrzenia tego, co znajduje się w strumieniu.

Wersja wygenerowana dla JAVAExpress

Końcowe porządki czyli refaktoryzacja: Zmień implementację algorytmu

Być może dla osób, które znają lub posługiwały się wzorcem iteratora, to rozwiązanie nie wygląda najlepiej. Nawet jeśli wzorzec iteratora jest ci obcy, jest kilka rzeczy na które warto zwrócić uwagę:

1. stworzony iterator ma pewien cykl życia — najpierw musi być wykonana metoda `moveToNextWord`, dopiero wtedy ma uzasadnienie `hasNextWord`, a na końcu należy wywołać metodę `dispose`;

2. iterator działa z pewnym przesunięciem — musimy zapamietywać element zwracany przez iterator, żeby uzyskać pożądany efekt (`currentWord = pageIterator.moveToNextWord()`);

3. metoda `hasNextWord` posiada parametr, który należy dostarczyć — oznacza to, że klient częściowo przejmuje odpowiedzialność za pewne elementy stanu iteratora.

Są to efekty uboczne, które nieco zaciemniają kod, poza tym utrudniają jego utrzymanie, gdyż należy znać specyfikę działania klasy `PageIterator`. Mimo że jest to rozwiązanie znacząco czytelniejszego od poprzedniego, cały czas wymaga poprawek. Solą w oku jest w zasadzie metoda `hasNextWord`, która określa, czy istnieją jeszcze słowa, do których jest dostęp przez iterator. Nie dość, że potrzebuje parametru z zewnątrz, to przy obecnej konstrukcji istnieje konieczność wywołania przynajmniej jeden raz metody `moveToNextWord`.

Co zatem można zrobić? Na pewno warto pozbyć się parametru z metody `hasNextWord`, gdyż to iterator powinien wiedzieć, czy są jeszcze dostępne słowa. Ponadto metoda ta powinna być w stanie odczytać wiersze strony HTML w przód, aby stwierdzić, czy istnieje jeszcze jakieś słowo (parametr ten nie powinien pochodzić z zewnątrz). Żeby osiągnąć ten efekt mamy między innymi takie opcje:

- obsłużyć przeglądanie w przód samodzielnie, tworząc pewien mechanizm buforujący;

- skorzystać z metod `mark` i `reset` z klasy `BufferedReader` do podejrzenia tego, co znajduje się w strumieniu.

Wersja wygenerowana dla JAVAExpress

Oba rozwiązania wymagają dużego wysiłku implementacyjnego i raczej skomplikowałyby rozwiązanie, a przecież nie o to nam chodzi.

Jest jeszcze jedna opcja, która może znacząco zmienić podejście do rozwiązania. Zauważmy, że zadaniem iteratora jest umożliwienia dostania się do każdego słowa, które znajduje się na stronie HTML wygenerowanej przez witrynę słownika. Można zatem w momencie inicjowania klasy `PageIterator` wczytać całą stronę HTML, zaś wszystkie odnaleziona tłumaczenia wpisać do listy. Mając listę, możemy poprzez delegację udostępnić metody charakterystyczne dla iteratora. Oczywiście klient nie musi w ogóle wiedzieć, że `PageIterator` wewnętrznie używa iteratora listy.

Przystąpmy zatem do działania. Metodę `prepareBufferedReader` zmieniamy w taki sposób, aby przygotowywała listę znalezionych słów na stronie HTML. Nazwijmy ją `prepareWordsList`.

```
private List<String> prepareWordsList(String wordToFind) {

    List<String> result = new ArrayList<String>();
    String urlString = "http://www.dict.pl/dict?word=" + wordToFind
            + "&words=&lang=PL";

    try {
        bufferedReader = new BufferedReader(new InputStreamReader(
                    new URL(urlString).openStream()));

        String word = moveToNextWord();
        while (hasNextWord(word)) {
            result.add(word);
            word = moveToNextWord();
        }

    } catch (MalformedURLException e) {
        throw new WebDictionaryException(e);
    } catch (IOException e) {
        throw new WebDictionaryException(e);
    } finally {
        dispose();
    }

    return result;
}
```

Metody `moveToNextWord` oraz `hasNextWord` uczynimy prywatnymi, gdyż będą potrzebne tylko w procesie inicjacji danych. W klasie `PageIterator` konstruktor będzie miał za zadanie wywołać metodę `prepareWordsList` oraz pobrać iterator z listy słów.

Iterator będzie polem w klasie — na nim będą przeprowadzane poszczególne operacje.

```
public class PageIterator {

    private BufferedReader bufferedReader = null;
    private Iterator<String> wordIterator = null;

    public PageIterator(String wordToFind) {
        List<String> words = prepareWordsList(wordToFind);
        wordIterator = words.iterator();
    }
```

Kluczowymi dla klienta staną się metody delegujące operacje na iteratorze listy słów:

```
    public boolean hasNext() {
        return wordIterator.hasNext();
    }

    public String next() {
        return wordIterator.next();
    }
```

Przy takiej konstrukcji klasy `PageIterator`, klasa klienta `SearchWordService` jest bardzo intuicyjnie napisana i łatwa w zrozumieniu.

```
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);
        int counter = 1;

        while (pageIterator.hasNext()) {
            DictionaryWord dictionaryWord = new DictionaryWord();
            dictionaryWord.setPolishWord( pageIterator.next() );
            dictionaryWord.setEnglishWord( pageIterator.next() );

            result.add( dictionaryWord );

            System.out.println( counter + ") "
                    + dictionaryWord.getPolishWord()
```

```
                    + " => " + dictionaryWord.getEnglishWord());
            counter = counter + 1;
        }

        return result;
    }


}
```

Strategia najlepszych programistów: Małe kroki

Przedstawiony sposób rozwiązywania problemów stanowi przykład kluczowej strategii, która towarzyszy najlepszym programistom. Zauważmy, że realizowany w tej książce przykład ewolucyjnie się rozwija. Rozwój ten następuje w małych krokach, po to by z jednego działającego już kodu przejść jak najprościej do nieco zmienionego rozwiązania. Dzięki temu zwiększamy efektywność, gdyż popełniamy mniej błędów, a jeśli je popełnimy, łatwiej znaleźć powód.

Sposób przedstawiania rozwiązania też nie jest przypadkowy. Otóż często wśród programistów istnieje przekonanie, że dobry algorytm należy wymyślić już na początku. Tymczasem warto rozpocząć od najprostszego rozwiązania, które przychodzi do głowy, a następnie go stopniowo zmieniać, jeśli istnieje taka potrzeba.

Ważne

Wśród programistów istnieje **przekonanie**, że dobry algorytm należy wymyślić już na początku. Tymczasem warto rozpocząć od najprostszego rozwiązania.

Programiści często spędzają godziny szukając *idealnego rozwiązania*. Gdy tymczasem nierzadko takie nie istnieje lub jego implementacja zwyczajnie zajmie zbyt wiele czasu, żeby się nią zajmować.

Wersja wirtualna dla JAVA e

Wersja wygenerowana dla JAVA exPress

Rozdział 4

Tajemnica mistrzów refaktoryzacji

Żył sobie kiedyś suficki filozof, który grał ciągle na jednej strunie. Pewnego dnia ktoś odważył się zapytać, dlaczego gra tylko na jednej strunie, skoro ma tak wiele do dyspozycji.
„Bo znalazłem tę właściwą" — odpowiedział.

Tak naprawdę przykład słownika, który realizujemy jest tylko pretekstem do oswajania się z pewnym sposobem myślenia, z pewnym sposobem działania. Techniki takie jak refaktoryzacja, wzorce projektowe i wzorce architektoniczne mają wspólną oś, pewien rdzeń, który moim zdaniem stanowi sedno sztuki programowania.

Ważne

Celem tego rozdziału jest odnalezienie właściwej struny.

Wersja wygenerowana dla JAVA HelpPress

Refaktoryzacja: Wydzielenie interfejsu

Rzeczywistość lubi zmiany. Użytkownicy swoimi prośbami powodują, że w naszym systemie pojawia się dodatkowe wymaganie:

Podczas wyszukiwania można wybrać rodzaj słownika internetowego

Pojawia się zatem pytanie, jak to zrealizować. Jednym z pomysłów może być napisanie różnych wersji klasy `SearchWordService` dla różnych silników wyszukiwania. Istnieje także inna opcja. Jeśli potrafimy stworzyć różne iteratory dla różnych słowników, wtedy wystarczy, że będziemy podmieniać tylko i wyłącznie iterator. Ponieważ głównym zadaniem iteratora jest zwracanie kolejno słowa polskiego i angielskiego, za-

tem wydaje się, że jest to pomysł możliwy do zrealizowania.

Zrefaktoryzujemy kod w taki sposób, aby przewidywał możliwość obsługi różnych silników słowników internetowych — wydzielimy interfejs. Bazą do tej refaktoryzacji będą metody publiczne z klasy `PageIterator`. Dotychczasowa klasa `PageIterator` w tej sytuacji stanie się jedną z wielu implementacji. Dla klarowności zmieńmy jej nazwę na `DictPageIterator` zaś interfejs nazwijmy `PageIterator`.

Interfejs będzie wyglądał następująco:

```
package pl.bnsit.webdictionary;

public interface PageIterator {

    public abstract boolean hasNext();

    public abstract String next();

}
```

zaś klasa `DictPageIterator` zmieni się nieznacznie:

```
public class DictPageIterator implements PageIterator {
```

Kierunek wprowadzania interfejsów

Często spotykałem się z sytuacją, kiedy w systemie powstawał interfejs (np. `SecurityManager`) oraz jedna implementacja (np. `SecurityManagerImpl`). Później nie powstawały nowe implementacje. Nie jest to zbyt dobra strategia (o ile używana biblioteka lub szkielet aplikacyjny nie wymaga takiej konstrukcji). Powoduje ona nadmierne mnożenie bytów, a w konsekwencji zaciemnia strukturę projektu.

Ważne

Interfejsy warto wyodrębniać dopiero wtedy, kiedy rzeczywiście występuje więcej niż jedna implementacja.

Inny przykład

Poniżej zamieszczam inną implementację interfejsu `PageIterator`, opartą o inny silnik słownika. Proponuję ci Czytelniku, napisanie własnej implementacji dla wprawy. Najważniejsze, aby realizowała założony interfejs w analogiczny sposób jak ma to miejsce w klasie `DictPageIterator`.

Ważne

Przykłady wykorzystania słowników bazują na witrynach internetowych, dlatego może się okazać, iż za jakiś czas przedstawiony kod może nie odpowiadać bieżącemu stanowi. Należy dostosować je samodzielnie lub prosimy o kontakt ebooks@bnsit.pl, a dołożymy wszelkich starań, aby ukazała się zaktualizowana wersja przykładów.

```
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.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class OnetPageIterator implements PageIterator {

    private BufferedReader bufferedReader = null;
    private Iterator<String> wordIterator = null;

    public OnetPageIterator(String wordToFind) {
        List<String> words = prepareWordsList(wordToFind);
        wordIterator = words.iterator();
    }

    @Override
    public boolean hasNext() {
        return wordIterator.hasNext();
    }

    @Override
    public String next() {
```

```
        return wordIterator.next();
    }

    private List<String> prepareWordsList(String wordToFind) {

        List<String> result = new ArrayList<String>();
        String urlString = "http://portalwiedzy.onet.pl/tlumacz.html?qs="
            + wordToFind + "&tr=ang-auto&x=0&y=0";

        try {
            bufferedReader = new BufferedReader(new InputStreamReader(
                        new URL(urlString).openStream()));

            result = extractWords();
        } catch (MalformedURLException e) {
            throw new WebDictionaryException(e);
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        } finally {
            dispose();
        }

        return result;
    }


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

    private List<String> extractWords() {
        List<String> result = new ArrayList<String>();
        try {
            String line = bufferedReader.readLine();
            Pattern pattern = Pattern
                .compile(".*?<div class=a2b style=\"padding: "
                    + "0px 0 1px 0px\">\\s?(<a href=\".*?\">)?"
                    + "(.*?)(</a>)? .*?<BR>(.*?)</div>.*?");

            while (hasNextLine(line)) {

                Matcher matcher = pattern.matcher(line);

                while (matcher.find()) {
                    String englishWord
                        = new String( matcher.group( 2 ).getBytes(),
                                "ISO-8859-2");
                    String polishHTMLFragment
                        = new String (matcher.group( 4 ).getBytes(),
                                "ISO-8859-2");
```

```
                    List<String> words
                        = extractTranslation(
                                englishWord, polishHTMLFragment + "<BR>");
                    result.addAll(words);
                }
                line = bufferedReader.readLine();
            }
        } catch (IOException e) {
            throw new WebDictionaryException(e);
        }

        return result;
    }

    private List<String> extractTranslation(String englishWord,
            String polishHTMLFragment) {
        List<String> result = new ArrayList<String>();

        Pattern pattern = Pattern
            .compile("(<B>\\d+</B>\\s)?(.*?)<BR>");

        Matcher matcher = pattern.matcher(polishHTMLFragment);

        while (matcher.find()) {
            String polishWord = matcher.group(2);
            result.add(polishWord);
            result.add(englishWord);
        }

        return result;
    }

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