DOI: 10.5604/01.3001.0009.5194

WYDAJNOŚĆ PRACY Z BAZAMI DANYCH W APLIKACJACH JEE

Magdalena Grzesińska, Magdalena Waszczyńska, Beata Pańczyk
Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

Streszczenie. *Niniejszy artykuł prezentuje porównanie wydajności bazodanowych aplikacji JEE z wykorzystaniem różnych interfejsów programistycznych (JDBC, Hibernate, JOOQ). Analiza porównawcza została wykonana na podstawie aplikacji testowej i odpowiednio przygotowanych scenariuszy. Porównanie zużycia pamięci oraz czasu realizacji operacji na bazie danych, zaprezentowano w postaci zestawień tabelarycznych i wykresów. We wnioskach wskazano korzyści wynikające ze stosowania omawianych technologii i obszary ich optymalnego stosowania.*

Słowa kluczowe: bazy danych, wydajność, JEE, ORM, Hibernate

JEE DATABASE APPLICATIONS PERFORMANCE

Abstract. *This article presents a comparison of JEE database applications performance using different programming interfaces (JDBC, Hibernate, jOOQ). The comparative analysis was made based on the test application and properly prepared scenarios. Comparison of memory usage and execution time for the database operations, was presented in tables and charts. The conclusions indicate the benefits of these technology and areas of their optimal use.*

Keywords: database, performance, JEE, ORM, Hibernate

Wstęp

Podstawą działania obecnych systemów informatycznych jest nie tylko gromadzenie informacji, ale przede wszystkim oferowanie szybkiego do nich dostępu. Internetowe systemy informatyczne najczęściej korzystają z relacyjnych baz danych. Systemy budowane na platformie Java Enterprise Edition (JEE) zwykle wykorzystują warstwę abstrakcyjną, mapującą tabele z relacyjnej bazy danych na klasy (ang. Object-Relational Mapping – ORM) [7]. Realizacja takiego mapowania jest bardzo wygodna dla programistów, rozwiązuje wiele problemów, ale często powoduje spadek wydajności działania aplikacji w przypadku wykonywania złożonych zapytań do bazy danych. Pojawia się zatem pytanie kiedy warto stosować ORM (w JEE zwykle jest to Hibernate [1]), a kiedy należy ograniczyć się do najprostszego interfejsu JDBC, albo wykorzystać ostatnio promowaną tendencję No ORM (np. Java Object Oriented Quering – jOOQ [2]).

Niniejszy artykuł poświęcony jest porównaniu wybranych technologii dostępu do danych wykorzystywanych w aplikacjach JEE. Dokonanie szczegółowego porównania pozwoli określić zalety i wady wskazanych technologii, co powinno pomóc programistom w wyborze narzędzia, które zapewni najlepszą komunikację aplikacji z bazą danych. Jest to o tyle istotne, że brakuje artykułów na ten temat, ze wskazaniem konkretnych wartości liczbowych [4]. Oczywistym jest, że wybór narzędzia do pracy z bazą danych jest ważnym etapem budowy systemu informatycznego, ponieważ jego konsekwencje mają wpływ na wiele aspektów, takich jak łatwość rozbudowy projektu oraz jego wydajność.

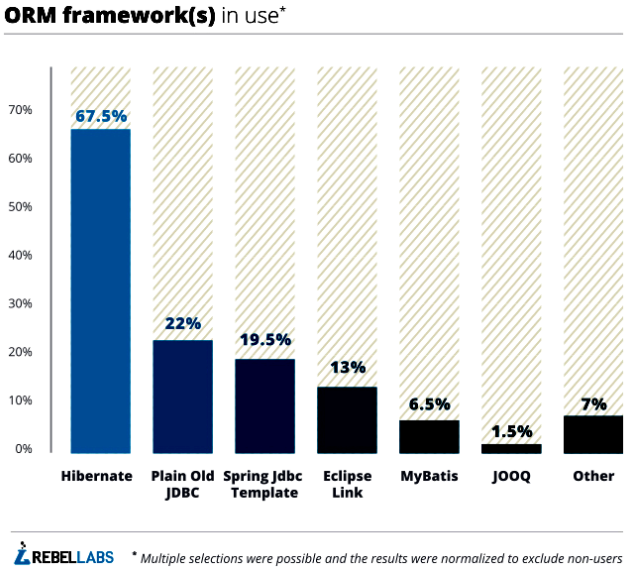
Artykuł koncentruje się na porównaniu podstawowego interfejsu (ang. Java Database Connectivity – JDBC [3]), technologii ORM na przykładzie Hibernate 4.3.1 oraz technologii promującej No ORM na przykładzie jOOQ 3.7.3. Każda z omawianych technologii zostanie scharakteryzowana według kryteriów, które pozwoliły na pokazanie najistotniejszych cech, mających znaczący wpływ na sposób komunikowania się z relacyjną bazą danych.

1. Dostęp do danych w aplikacjach JEE

W procesie programowania aplikacji internetowych wykorzystanie technologii mapowania obiektowo-relacyjnego jest obecnie standardem. Gdy system wymaga pobrania informacji z bazy, wykonywana jest zawsze ta sama sekwencja kroków: nawiązanie połączenia, wysłanie zapytania SQL, odebranie wyniku zapytania oraz zamknięcie połączenia. Może to prowadzić do wielu problemów podczas prac nad systemem. Zmiany w strukturze bazy danych wymuszają zmiany zapytań występujących w wielu miejscach kodu źródłowego. Wykorzystanie innej bazy (np. przejście z MySQL na Oracle) może sprawić, że część zapytań nie będzie prawidłowa.

Mapowanie obiektowo-relacyjne pozwala na automatyzację tej wieloetapowej procedury komunikacji kodu źródłowego tworzonego systemu z relacyjną bazą danych. Zautomatyzowanie tego procesu pozwala zwykle rozwiązać wiele niedogodności. Korzystanie z technologii ORM posiada wiele zalet. Operacje na danych przeprowadzane są z wykorzystaniem programowania zorientowanego obiektowo. Korzystanie z narzędzi ORM nie wymusza na programiście stosowania zapytań SQL. Zapytania te w kodzie źródłowym aplikacji są redukowane do minimum, a w skrajnych przypadkach kod SQL nie jest nigdzie używany. Mapowanie obiektowo-relacyjne nie jest jednak pozbawione wad. Utworzenie warstwy pośredniczącej pomiędzy kodem programu a bazą danych wprowadza opóźnienia w przekazywaniu danych. Za pomocą ORM trudno jest utworzyć skomplikowane, wykorzystujące np. wielokrotnie zagnieżdżone, skorelowane zapytania. Jest to możliwe, jednak w takich przypadkach warto zrezygnować z ORM i zastosować zwykłe zapytanie SQL w dialekcie odpowiednim dla wybranej technologii baz danych.

W ostatnich latach pojawiła się jednak tendencja negująca stosowanie ORM, a promująca powrót do bezpiecznych zapytań SQL z wykorzystaniem dedykowanych interfejsów obiektowych (w przypadku języka Java takim interfejsem jest jOOQ). Na rysunku 1 przedstawiono udział poszczególnych technologii dostępu do relacyjnych baz danych w aplikacjach w języku Java.



Rys. 1. Wykorzystanie technologii dostępu do danych w aplikacjach javy [6]

Niekwestionowanym liderem jest tu Hibernate, uznawany de facto za standard dla aplikacji JEE. Hibernate jest przedstawicielem świata ORM, podczas gdy jOOQ reprezentuje wizję potęgi zapytań SQL. Alternatywą dla Hibernate/JPA/ORM jest jOOQ/JDBC/SQL. JOOQ generuje kod javy na podstawie istniejącej już bazy danych (ang. database first) i pozwala budować bezpieczne (ang. typesafe) zapytania SQL [5].

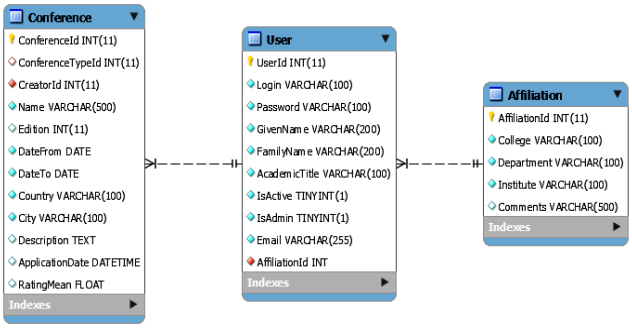
2. Testy aplikacji JEE

Dane platformy systemowej i serwera, na którym wykonano testy zostały zamieszczone w tabeli 1. Testy przeprowadzono na relacyjnej bazie danych MySQL.

Tabela 1. Dane środowiska testowego

Nazwa	Wartość
System operacyjny	Windows 8.1 64-bit
Procesor / pamięć RAM	Intel Core i5-5200U 2.20GHz / 8GB DDR3 L
Serwer GlassFish	4.1.1
MySQL	Version 15.1 Distribution 10.0.17-MariaDB (xampp w wersji 3.2.1)
Java	Java EE 7 Web

Testy wydajności przeprowadzone zostały z wykorzystaniem autorskiej aplikacji internetowej JEE do obsługi konferencji. Fragment diagramu ERD bazy danych przedstawiono na rys. 2.



Rys. 2. Fragment diagramu ERD

W celu praktycznego i obiektywnego porównania omawianych narzędzi ustalono cztery scenariusze testowe. Każdy z nich, przeprowadzony dla wybranych interfejsów pozwolił określić czas wykonania kodu oraz wykorzystanie pamięci. Szczegółowe opisy scenariuszy przedstawia tabela 2.

Tabela 2. Opis scenariuszy testowych

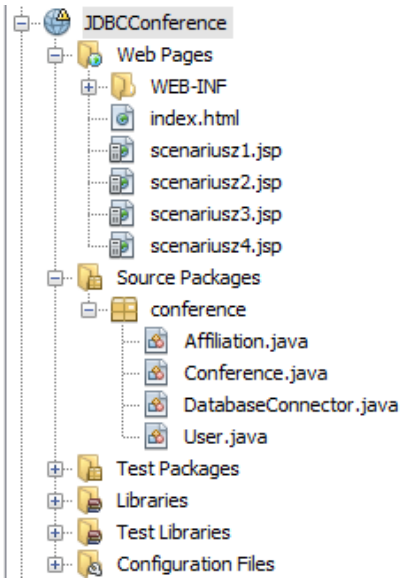
Oznaczenie	Opis
S1	*Zapis pojedynczego rekordu*. Utworzenie obiektu reprezentującego rekord i utrwalenie go w bazie danych.
S2	*Odczyt pojedynczego rekordu*. Odczyt z bazy danych pojedynczego rekordu na podstawie klucza głównego.
S3	*Zapis rekordów powiązanych*. Zapis do bazy danych rekordów w trzech tabelach, połączonych ze sobą relacją.
S4	*Odczyt rekordów powiązanych*. Odczytanie rekordów z trzech tabel powiązanych ze sobą relacją.

Testy zostały zrealizowane w oparciu o trzy projekty ***JDBCConference***, ***HibernateConference*** i ***JOOQConference***, które współpracują z tą samą bazą danych. Dla każdego scenariusza stworzono plik o nazwie odpowiednio *scenariusz1.jsp*, *scenariusz2.jsp*, *scenariusz3.jsp* i *scenariusz4.jsp*.

Cała logika zapisu i odczytu danych z bazy odbywa się w klasach (odpowiednikach tabel z bazy): **Affiliation**, **User** i **Conference** w pakiecie ***conference*** (w przypadku jOOQ w wygenerowanych klasach w dodatkowym pakiecie ***generated.tables***).

Strukturę przykładowego projektu JEE współpracującego z danymi za pośrednictwem podstawowego interfejsu JDBC przedstawia rys.3. W przypadku Hibernate, pakiet ***conference*** zawiera dodatkowo pliki konfiguracyjne *xml* opisujące mapowanie obiektowo relacyjne tabel z bazy na klasy javy (*Conference.hbm.xml*, *Affiliation.hbm.xml* i *User.hbm.xml*) oraz klasę pomocniczą *HibernateUtil.java*, która udostępnia podstawowy obiekt do pracy z Hibernate. Konfiguracja połączenia z bazą dostępna jest w pliku *hibernate.config.xml*.

Dla jOOQ z konsoli wygenerowano (za pomocą bibliotek jooq).dodatkowe pakiety *generated*, *generated.tables* i *generated.tables.records*. Podobnie jak w Hibernate, wykonano reverse engineering. Nie potrzebna była żadna szczególna konfiguracja, wystarczyło dołączyć biblioteki i wygenerować klasy odwzorowujące tabele i rekordy w tabelach.



Rys. 3. Struktura aplikacji testowej JDBCConference

Przykłady 1, 2 i 3 prezentują fragmenty kodu konieczne do realizacji zapisu rekordów powiązanych zgodnie ze scenariuszem S3 odpowiednio dla JDBC, Hibernate i jOOQ.

Przykład 1. Fragmenty kodu z interfejsem JDBC dla S3

```
//Utworzenie obiektów Affiliation, User i Conference:
Affiliation a = new Affiliation("test3", "test3", "test3", "test3");
User u = new User(a, "login", "password", "Jan", "Kowalski", "dr", true,
      false, "email");
 Conference c = new Conference(1, u, "test", null,
      format.parse("25-04-2016"), format.parse("25-04-2016"), "Polska",
       "Lublin", null, format.parse("20-04-2016"), new Float(0));
Connection conn = DatabaseConnector.getConnection();
PreparedStatement ps;
 String sql = "insert into Conference (ConferenceTypeId, CreatorId,
Name,Edition, DateFrom, DateTo, Country, City, Description, Applica-
tionDate, RatingMean) values (?,last_insert_id(),?,?,?,?,?,?,?,?,?)";
try {
          ps = conn.prepareStatement(sql);
          ps.setInt(1, this.conferenceTypeId);
          ps.setInt(2, this.user.getUserId());
          ps.setString(3, this.name);
          ps.setString(4, this.edition);
          ps.setDate(5, new java.sql.Date(this.dateFrom.getTime()));
          ps.setDate(6, new java.sql.Date(this.dateTo.getTime()));
          ps.setString(7, this.country);
          ps.setString(8, this.city);
          ps.setString(9, this.description);
          ps.setDate(10, new java.sql.Date(this.applicationDate.getTime()));
          ps.setFloat(11, ratingMean);
          ps.setString(2, this.name);
          ps.setString(3, this.edition);
          ps.setDate(4, new java.sql.Date(this.dateFrom.getTime()));
          ps.setDate(5, new java.sql.Date(this.dateTo.getTime()));
          ps.setString(6, this.country);
          ps.setString(7, this.city);
          ps.setString(8, this.description);
          ps.setDate(9, new java.sql.Date(this.applicationDate.getTime()));
          ps.setFloat(10, ratingMean);
          ps.executeUpdate();
         conn.close();
      } catch (SQLException ex) {
```

Przykład 2. Fragmenty kodu z Hibernate dla S3

```
//Utworzenie obiektów Affiliation, User i Conference jak dla JDBC
u.addConference(c);
a.addUser(u);
session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = this.session.getTransaction();
tx.begin();
this.session.saveOrUpdate(a);
tx.commit();
session.close();
```

Przykład 3. Fragmenty kodu z jOOQ dla S3

```
DSLContext create = DSL.using(DatabaseConnector.getConnection(),
    SQLDialect.MYSQL);
Record a =create.insertInto(generated.tables.Affiliation.AFFILIATION,
     generated.tables.Affiliation.AFFILIATION.COLLEGE,
    generated.tables.Affiliation.AFFILIATION.DEPARTMENT,
    generated.tables.Affiliation.AFFILIATION.INSTITUTE,
    generated.tables.Affiliation.AFFILIATION.COMMENTS)
    .values("testj","testj","testj","testj").returning(
    generated.tables.Affiliation.AFFILIATION.AFFILIATIONID)
    .fetchOne();
 Record u =create.insertInto(generated.tables.User.USER,
     generated.tables.User.USER.LOGIN,
    generated.tables.User.USER.PASSWORD,
    generated.tables.User.USER.GIVENNAME,
    generated.tables.User.USER.FAMILYNAME,
    generated.tables.User.USER.ACADEMICTITLE,
    generated.tables.User.USER.ISACTIVE,
    generated.tables.User.USER.ISADMIN,
    generated.tables.User.USER.EMAIL,
    generated.tables.User.USER.AFFILIATIONID).values("login",
    "password", "Jan", "Kowalski", "dr", bytetrue, bytefalse, "email",
    a.getValue(generated.tables.Affiliation.AFFILIATION.
    AFFILIATIONID)).returning(generated.tables.User.USER.USERID)
    .fetchOne();
DateFormat format = new SimpleDateFormat("dd-MM-yyyy");
create.insertInto(generated.tables.Conference.CONFERENCE)
    .set(generated.tables.Conference.CONFERENCE.
    APPLICATIONDATE, new java.sql.Date(format.parse("25-04-2016")
    .getTime())).set(generated.tables.Conference.CONFERENCE.CITY,
    "testj").set(generated.tables.Conference.
    CONFERENCE.CONFERENCETYPEID, 1)
    .set(generated.tables.Conference.CONFERENCE.COUNTRY, "testj")
    .set(generated.tables.Conference.CONFERENCE.CREATORID,
    u.getValue(generated.tables.User.USER.USERID))
    .set(generated.tables.Conference.CONFERENCE.DATEFROM,
    new java.sql.Date(format.parse("25-04-2016").getTime()))
    .set(generated.tables.Conference.CONFERENCE.DATETO,
    new java.sql.Date(format.parse("25-04-2016").getTime()))
    .set(generated.tables.Conference.CONFERENCE.DESCRIPTION,
    "testj").set(generated.tables.Conference.CONFERENCE.NAME,
    "testj").set(generated.tables.Conference.CONFERENCE
    .RATINGMEAN, new Double(0)).execute();
```

Podczas kodowania PL/SQL, T-SQL lub innego języka proceduralnego, zapytania SQL są wykonywane natychmiast po ich zakończeniu średnikiem. W przypadku jOOQ wykonanie następuje dopiero w momencie wywołania metody *fetch()* lub *execute()*. W przypadku Hibernate o wykonaniu zapytania decyduje metoda *commit()* wywołana dla obiektu klasy ***Transaction***.

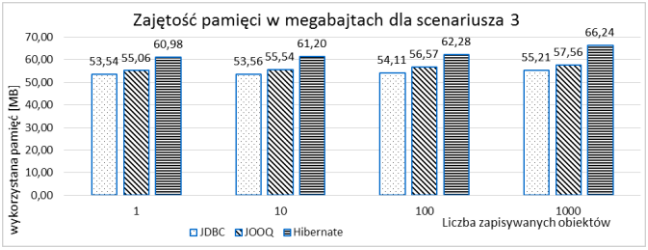
Wszystkie pomiary dla 4 scenariuszy były powtarzane 10 razy, a wartość końcowa jest średnią uzyskanych wyników. Przykładowe rezultaty pomiarów przeprowadzonych według kolejnych scenariuszy prezentują tabele 3-6. Wyniki badań dla scenariuszy S3 i S4 zostały dodatkowo zobrazowane na rysunkach 4-7.

Tabela 3. Czas i zajętość pamięci dla scenariusza S1

Pamięć [MB]			
Liczba obiektów	**JDBC**	**JOOQ**	**Hibernate**
1	53,38	54,44	60,77
10	53,42	54,87	61,33
100	53,92	56,07	62,16
1000	54,58	57,08	65,64
Czas [ms]			
1	43,9	78,7	49,9
10	430,7	501,4	459,3
100	3985,4	4912	4705,2
1000	59333,4	68827	67488,3

Tabela 4. Czas i zajętość pamięci dla scenariusza S2

Pamięć [MB]			
Liczba obiektów	**JDBC**	**JOOQ**	**Hibernate**
1	52,04	54,84	57,50
10	52,35	55,42	57,91
100	52,37	55,95	58,95
1000	52,59	56,78	59,63
10000	54,57	57,56	63,76
Czas [ms]			
1	7,2	9,7	11,4
10	8,2	11,6	27,4
100	8,7	13,4	65,3
1000	13,6	14,7	260,3
10000	44,6	48,2	1676,8



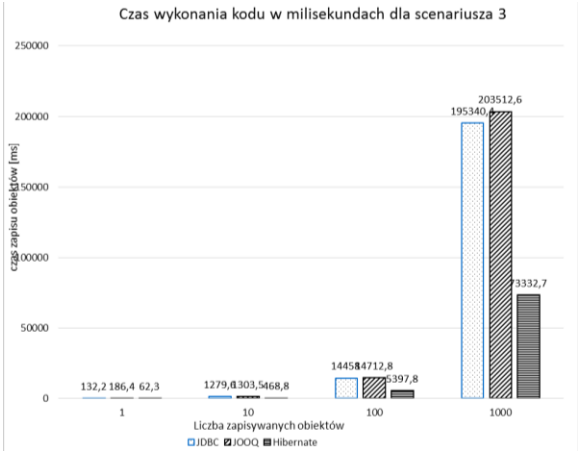
Rys. 4. Zajętość pamięci zgodnie ze scenariuszem S3

Tabela 5. Czas i zajętość pamięci dla scenariusza S3

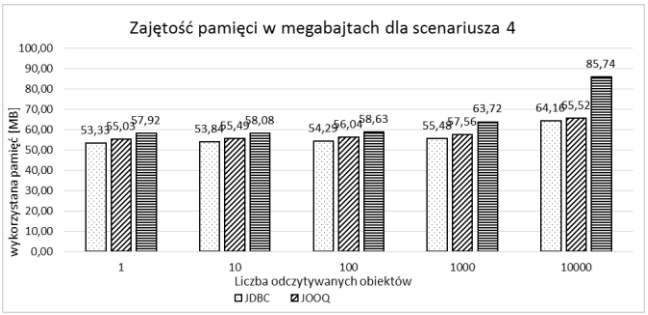
Pamięć [MB]			
Liczba obiektów	**JDBC**	**JOOQ**	**Hibernate**
1	53,54	55,06	60,98
10	53,56	55,54	61,20
100	54,11	56,57	62,28
1000	55,21	57,56	66,24
Czas [ms]			
1	132,2	186,4	62,3
10	1279,6	1303,5	468,8
100	14458	14712,8	5397,8
1000	195340,4	203512,6	73332,7

Tabela 6. Czas i zajętość pamięci dla scenariusza S4

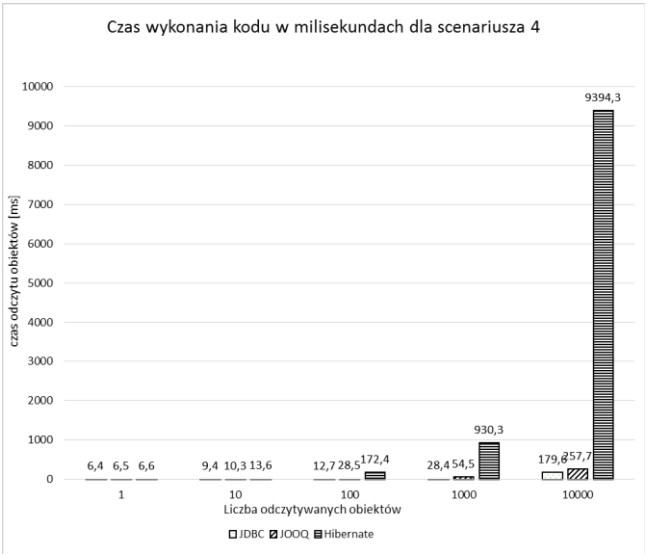
Pamięć [MB]			
Liczba obiektów	**JDBC**	**JOOQ**	**Hibernate**
1	53,33	55,03	57,92
10	53,84	55,49	58,08
100	54,29	56,04	58,63
1000	55,48	57,56	63,72
10000	64,16	65,52	85,74
Czas [MB]			
1	6,4	6,5	6,6
10	9,4	10,3	13,6
100	12,7	28,5	172,4
1000	28,4	54,5	930,3
10000	179,6	257,7	9394,3



Rys. 5.Czas wykonania kodu dla scenariusza S3



Rys. 6. Zajętość pamięci zgodnie ze scenariuszem S4



Rys. 7. Czas wykonania kodu dla scenariusza S4

3. Wnioski

W artykule przedstawiono trzy różne podejścia do pracy z bazami danych w aplikacjach JEE. Dla analizowanej BD, biorąc pod uwagę kryterium czasu dla scenariuszy S1, S2 i S4, najlepszym (ale tylko nieznacznie w stosunku do jOOQ) jest interfejs JDBC. Z kolei w przypadku zapisu powiązanych rekordów (scenariusz S3) najszybszym okazał się Hibernate. Rozważając kryterium zajętości pamięci, technologia Hibernate jest najmniej wydajna, chociaż nie ma tu istotnych różnic.

Z przeprowadzonych testów wynika, ze nie można wskazać jednej, najlepszej metody pracy z danymi, zwłaszcza, że istotna rolę odgrywa też konkretna struktura bazy danych oraz funkcjonalność aplikacji z nią współpracującej.

Rozwiązania oparte na ORM są bardzo wygodne dla programisty. Dane pamiętane w obiektach programu są w takiej samej formie utrwalane w bazie bez dodatkowych operacji zmieniających format z obiektowego na relacyjny (i odwrotnie). Jednak ręczne pisanie zapytania SQL jest szybsze w realizacji niż zapełnianie pamięci obiektami modeli i wykonywanie dynamicznych mapowań. ORM opóźnia transmisję informacji tworząc dodatkową warstwę, przez którą muszą być przesłane dane. Generalnie ORM nie nadaje się do obróbki dużych ilości danych gdzie wydajność jest podstawowym priorytetem, chociaż jak wynika z testów scenariusza S3, w przypadku zapisu dużej liczby rekordów powiązanych ze sobą, Hibernate jest najbardziej efektywny.

Wady ORM przyczyniły się do szukania nowych rozwiązań. W przypadku świata javy takim rozwiązaniem jest biblioteka jOOQ. Pisanie zapytań za pomocą tej biblioteki jest intuicyjne i łatwe. Składnia bardzo przypomina składnię SQL. Na przykład, w przypadku zapytania typu *select* dla wielu połączonych tabel, wszystkie elementy typowego SQl (*from*, j*oin*, *where*, *group by*) mają dokładne odwzorowanie w jOOQ. Według opinii autorek, bardziej skomplikowane zapytania łatwiej jest zbudować w jOOQ niż w Hibernate. W Hibernate bardzo częstym problemem był zgłaszany wyjątek *lazy initialization exception*, który występował przy próbie dostępu do obiektu, zagnieżdżonego w innym (np. dostęp do *Affiliation* z poziomu *User)*. W jOOQ, ten problem nie istnieje - wystarczyło skorzystać ze statycznych stałych w wygenerowanych klasach, które oznaczają nazwę tabeli oraz nazwę kolumny (Przykład 3).

Biblioteka jOOQ została pierwotnie zaprojektowana w celu całkowitego zastąpienia istniejących narzędzi ORM. Tym niemniej społeczność javy zaproponowała nieco inne podejście do jej wykorzystania. Najbardziej rozsądnym i efektywnym wydają się być rozwiązania hybrydowe, wykorzystujące najlepsze możliwości sprawdzonych technologii. Na przykład bardzo efektywne może być podejście promujące stosowanie Hibernate w 70% zapytań (tj klasyczny CRUD) a jOOQ dla pozostałych 30%, gdzie naprawdę potrzebny jest SQL. Inną sugestią jest wykorzystanie jOOQ tylko do budowy SQL a JDBC do jego wykonania. Takie hybrydowe rozwiązania są ciekawą alternatywą w stosunku do stosowania jednej wybranej technologii i mogą stanowić zakres dalszych badań.

Literatura

[1] http://hibernate.org/ [1.11.2016].
[2] http://www.jooq.org/ [1.11.2016].
[3] http://www.oracle.com/technetwork/java/javase/jdbc/index.html [1.11.2016].
[4] http://www.ploug.org.pl/konf_10/materialy/pdf/19.pdf, Boiński Paweł, Analiza porównawcza technologii implementacji warstwy dostępu do danych w aplikacjach ADF, XVI Konferencja PLOUG, Kościelisko 2010 [1.11.2016].
[5] https://blog.jooq.org/2015/03/24/jooq-vs-hibernate-when-to-choose-which/ [1.11.2016].
[6] https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/10/ [1.11.2016].
[7] Pańczyk B., Sławiński A.: Technologie mapowania obiektowo-relacyjnego w aplikacjach PHP. IAPGOŚ, 1(5)/2015, 29–32.

Inż. Magdalena Grzesińska
e-mail: magdagrzesinska@gmail.com

Studentka kierunku Informatyka na Politechnice Lubelskiej (specjalność: Aplikacje Internetowe). Obecnie pracuje na stanowisku programista baz danych w lubelskim oddziale firmy Asseco Business Solutions. W Wydziale Rozwiązań Bazodanowych zajmuje się tworzeniem interfejsów integracyjnych i raportowych na potrzeby aplikacji mobilnej Mobile Touch przy użyciu języka Transact-SQL. Poza bazami danych interesuje się również technologiami webowymi oraz systemami Business Intelligence

Mgr inż. Magdalena Waszczyńska
e-mail: magdalena.waszczynska@gmail.com

Absolwentka Matematyki na Uniwersytecie Marii Skłodowskiej Curie w Lublinie w specjalności matematyka finansowa i ubezpieczeniowa (2014). Obecnie studentka Informatyki na Politechnice Lubelskiej w specjalności aplikacje internetowe. Od kilkunastu miesięcy programista baz danych w firmie Asseco Business Solutions w Wydziale Rozwiązań Bazodanowych. Razem z zespołem rozwija aplikację Mobile Touch - mobilną platformę wsparcia sprzedaży. W obszarze zainteresowań znajdują się również hurtownie danych, systemy BI, analiza danych oraz rynki finansowe.

Dr Beata Pańczyk
e-mail: b.panczyk@pollub.pl

Ukończyła studia matematyczne na UMCS w Lublinie. W latach 1989-2011 pracownik naukowy (asystent, adiunkt) w Instytucie Informatyki Politechniki Lubelskiej. Tytuł doktora uzyskała w roku 1996 na Wydziale Elektrycznym PL. Temat rozprawy doktorskiej: Konstrukcja obrazu rozkładu właściwości fizycznych obiektu metodą Impedancyjnej Tomografii Komputerowej. Od roku 2011 na stanowisku starszego wykładowcy. Obszar zainteresowań dydaktycznych i naukowych to metody numeryczne, języki programowania i technologie tworzenia aplikacji internetowych.

otrzymano/received: 22.06.2016 *przyjęto do druku/accepted:30.10.2016*