DOI: 10.5604/20830157.1148044

TECHNOLOGIE MAPOWANIA OBIEKTOWO-RELACYJNEGO W APLIKACJACH PHP

Beata Pańczyk[1], Arkadiusz Sławiński[2]
[1]Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki, [2]Imaginalis, ul. Dobrzańskiego 1, 20-262 Lublin

Streszczenie. *Niniejszy artykuł prezentuje porównanie dwóch najczęściej wykorzystywanych w programowaniu aplikacji internetowych PHP technologii mapowania obiektowo-relacyjnego: Propel i Doctrine. Analiza porównawcza została wykonana na podstawie aplikacji testowej i odpowiednio opracowanych scenariuszy. Wyniki zaprezentowano w postaci zestawień tabelarycznych i wykresów. We wnioskach wskazano korzyści wynikające ze stosowania obu technologii w odniesieniu do czystego kodu PHP.*

Słowa kluczowe: aplikacje internetowe, PHP, ORM, Doctrine, Propel

OBJECT-RELATIONAL MAPPING TECHNOLOGIES IN PHP APPLICATIONS

Abstract. *This paper presents a comparison of the two most commonly used for PHP applications object-relational mapping technologies: Propel and Doctrine. The comparative analysis was made on the basis of the test application and test cases. The results are presented in tables and figures. The conclusions indicate the benefits of applying both technologies in relation to the pure PHP code.*

Keywords: web applications, PHP, ORM, Doctrine, Propel

Wstęp

Gromadzenie informacji jest podstawą działania systemów informatycznych. Internetowe systemy informatyczne – poprzez proste strony firmowe, rozbudowane aplikacje internetowe a także zaawansowane systemy zarządzania przedsiębiorstwem, korzystają z relacyjnych baz danych. Zdecydowana większość systemów tworzona jest w sposób obiektowy. Przy zestawieniu relacyjnej bazy danych z obiektowym programowaniem można wyodrębnić abstrakcyjną warstwę aplikacji, która pozwoli na odwzorowanie tabel bazodanowych na klasy i obiekty w kodzie źródłowym systemu. Taka technika nazywa się mapowaniem obiektowo-relacyjnym (ORM – ang. object-relational mapping) [11]. Niniejszy artykuł poświęcony jest porównaniu niezależnych technologii ORM (Doctrine i Propel) wykorzystywanych w aplikacjach PHP na przykładzie popularnego na całym świecie frameworka Symfony 2 [13, 14]. Dokonanie porównania pozwoli określić zalety i wady obydwu technologii, co powinno pomóc programistom w dokonaniu wyboru narzędzia ORM, który zapewni lepszą komunikację aplikacji z bazą danych. Wybór ten jest ważnym etapem budowy systemu informatycznego, ponieważ jego konsekwencje mają wpływ na wiele aspektów, takich jak łatwość rozbudowy projektu i jego wydajność.

Artykuł obejmuje porównanie Propel w wersji 1.7 oraz Doctrine w wersji 2.1. Każda z omawianych technologii została scharakteryzowana według kryteriów, które pozwoliły na pokazanie ich najistotniejszych cech, mających znaczący wpływ na sposób komunikowania się z relacyjną bazą danych przy użyciu języka PHP. Dobór kryteriów porównawczych w części teoretycznej pozwolił dokonać analizy zarówno na poziomie technicznej struktury tych narzędzi (aspekt inżynierii oprogramowania) jak i na poziomie praktycznego użycia podczas procesu programowania. Porównania dokonano wykorzystując testową aplikację [8].

1. Technologie mapowania obiektowo-relacyjnego

W procesie programowania aplikacji internetowych wykorzystanie technologii mapowania obiektowo-relacyjnego jest obecnie standardem. Za każdym razem, gdy system wymaga pobrania informacji z bazy, wykonywana jest sekwencja kroków: nawiązanie połączenia, wysłanie zapytania SQL, odebranie wyniku zapytania oraz zamknięcie połączenia. Może to prowadzić do wielu problemów podczas prac nad projektem: zmiany w strukturze bazy danych wymuszają zmiany zapytań występujących w wielu miejscach kodu źródłowego. Wykorzystanie innej technologii bazy danych (np. przejście z MySQL na Oracle) może sprawić, że część zapytań nie będzie prawidłowa. ORM pozwala na automatyzację tej wieloetapowej procedury komunikacji kodu źródłowego tworzonego systemu z relacyjną bazą danych. Zautomatyzowanie tego procesu pozwala na rozwiązanie wymienionych niedogodności [8]. Korzystanie z technologii mapowania obiektowo-relacyjnego posiada wiele zalet. Przede wszystkim operacje na danych przeprowadzane są z wykorzystaniem programowania zorientowanego obiektowo, co oferuje programiście chociażby możliwość wykorzystania mechanizmu dziedziczenia (co w relacyjnej bazie danych jest dość trudne do zrealizowania). Korzystanie z narzędzi ORM nie wymusza korzystania z 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 pewne opóźnienia w przekazywaniu danych. Innym problemem związanym z technologią ORM jest fakt, iż trudno za jej pomocą utworzyć skomplikowane, wykorzystujące np. wielokrotnie zagnieżdżone, skorelowane zapytania. Jest to oczywiście możliwe, jednak w takich przypadkach warto zastosować zwykłe zapytanie napisane w języku SQL odpowiednim dla wybranej technologii baz danych [10, 11].

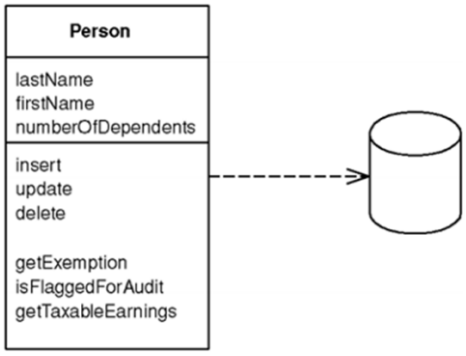
Narzędzia mapowania obiektowo-relacyjnego są dostępne jako samodzielne biblioteki, które mogą być używane w każdym projekcie, jak również mogą być ściśle zintegrowane z frameworkami opartymi na wzorcu projektowym MVC. ORM jest obecnie podstawowym elementem każdego frameworka dla języka PHP. Yii, Kohana czy CakePHP używają własnych ORM. Symfony i Zend Framework wykorzystują niezależnie rozwijające się technologie takie jak Doctrine czy Propel.

2. Propel

Propel to projekt oparty na licencji MIT, przeznaczony do pracy w języku PHP od wersji 5.4. Opisywana technologia mapowania obiektowo-relacyjnego używa rozszerzenia PDO (PHP Data Objects) jako warstwy abstrakcji pomiędzy językiem PHP a bazą danych. Oprócz zapewnienia prostej obsługi zapisywania i pobierania danych z relacyjnej bazy danych, narzędzie to posiada inne mechanizmy często wykorzystywane w programowaniu aplikacji, np. migracje schematów baz danych czy inżynierię wsteczną [13].

Propel oparty jest na wzorcu **Active Record** (AR). Jego koncepcja jest dość prosta: należy utworzyć klasę o strukturze jak najbardziej zbliżonej do tabeli lub widoku bazy danych, która będzie odpowiadała jednemu rekordowi w tej bazie. Tak utworzona klasa jest odpowiedzialna za zapis i odczyt danych, a bardzo często także za operacje związane z logiką biznesową operującą na nich [3]. Schematyczny diagram tego wzorca został przedstawiony na rysunku 1. Klasa *Person* posiada

właściwości odpowiadające strukturze rekordu, metody służące do tworzenia, modyfikacji i usunięcia rekordu oraz metody operujące na danych tego rekordu.



Rys. 1. Schemat UML wzorca Active Record [3]

Ważne jest, aby struktura ***Rekordu Aktywnego*** dokładnie odpowiadała schematowi bazy danych: jedno pole w klasie identyfikuje jedną kolumnę w tabeli. Klasy ***Active Record*** posiadają głównie metody odpowiedzialne za [1, 3, 7]:

- utworzenie instancji AR zawierających zestaw wierszy pobranych w zapytaniu;
- utworzenie instancji dla późniejszego zapisu nowo tworzonego rekordu;
- wyszukiwanie danych w tabeli na podstawie często używanych kryteriów (wartość klucza głównego, wartość wybranego atrybutu lub atrybutów itp.); metody te są statyczne, zwracają zestaw obiektów AR;
- ustawianie i pobieranie wartości pól (potocznie znane jako metody typu *get/set*). Funkcje te, zwane akcesorami, oprócz ich podstawowych zadań mogą zawierać mechanizmy przetworzenia danych pobranych z bazy, ich konwersję z typów danych specyficznych dla bazy na typy charakterystyczne dla języka programowania i odwrotnie. W sytuacji, gdy pole rekordu zawiera w sobie identyfikator wiersza z innej tabeli funkcja *get*, na podstawie zdefiniowanej w schemacie relacji pomiędzy tabelami, umożliwia pobranie ***Rekordu Aktywnego*** połączonego z wierszem.
- implementację logiki biznesowej.

Active Record jest dobrym wyborem w sytuacji, gdy model domenowy i jego logika nie są zbyt skomplikowane, tak jak chociażby w przypadku używania prostych operacji CRUD (ang. Create, Read, Update and Delete) czy walidacji danych. Należy podkreślić, że wzorzec został stworzony do pracy tylko z jednym rekordem na raz [5]. Dlatego dobrze nadaje się do prostych operacji jak np. edycja wpisu na blogu czy komentarza [9]. Zaletą tego wzorca jest prostota i przejrzystość - klasy rekordu aktywnego buduje się łatwo i według jasno określonego schematu [3]. Problem pojawia się w przypadku, gdy twórcy oprogramowania mają do czynienia z bardziej rozbudowaną bazą danych. Wynika to z faktu, że struktura obiektu jest ściśle związana z budową rekordu tabeli, co może prowadzić do konieczności częstej refaktoryzacji kodu klas AR w sytuacji wprowadzenia zmian w strukturze tabeli [3].

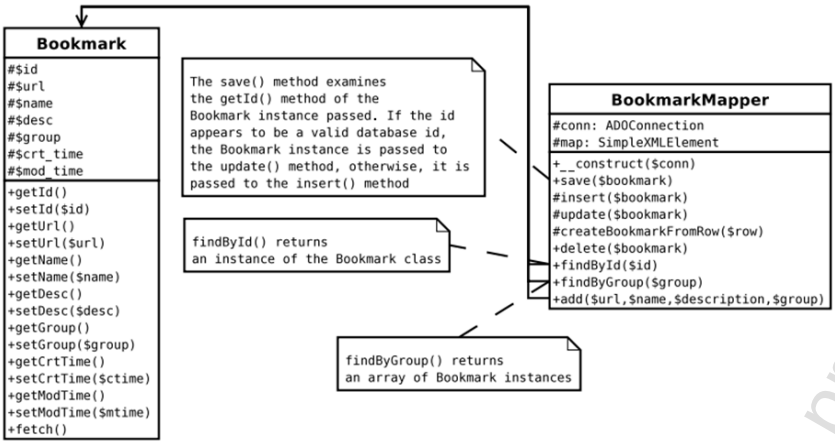
Omawiany wzorzec projektowy jest bardzo podobny do innego bazodanowego wzorca - ***Row Data Gateway***. W największym skrócie polega on na tym, że za pomocą obiektu istnieje dostęp do zasobu. Jednak w ***Rekordzie Aktywnym*** oprócz zapewnienia dostępu do bazy danych dodano także możliwość zaimplementowania funkcji związanych z logiką biznesową [3].

Przy pracy z Propel 1.7 pierwszym krokiem jest zawsze proces definiowania struktury elementów składających się na model biznesowy aplikacji. Wynikiem tego działania jest plik schematu zapisany w formacie XML. Na podstawie tego pliku Propel tworzy klasy PHP, nazywane klasami modelu, zawierające kod mapujący obiekty na tabele w relacyjnej bazie danych. Klasy modelu stanowią interfejs do wyszukiwania i modyfikacji danych w bazie [13]. Propel oferuje prosty mechanizm służący do działań CRUD i automatycznie generuje na podstawie pliku schematu odpowiednie metody dostępowe.

Propel oferuje obsługę wszystkich typów relacji: jeden do wielu, jeden do jednego oraz wiele do wielu. Zdefiniowanie kluczy obcych dla tabel w pliku schematu umożliwia wygenerowanie metod pozwalających na efektywną pracę z obiektami reprezentującymi tabele połączone ze sobą relacją [13].

3. Doctrine

Doctrine 2 jest narzędziem do mapowania obiektowo-relacyjnego stworzonym dla języka PHP w wersji 5.3.3 i wyższych. Zapewnia przejrzystą obsługę bazy danych w sposób obiektowy. Jest domyślnie używanym ORM w Symfony 2. Framework ten oparty jest na wzorcu projektowym **Data Mapper** (w wersji 1.* stosowany był wzorzec **Active Record** [2]). Charakterystyczną cechą Doctrine jest możliwość komunikowania się z bazą danych poprzez zapytania w sposób zorientowany obiektowo z wykorzystaniem języka Doctrine Query Language (DQL), inspirowanym na HQL znanym z technologii Hibernate. Ponieważ struktura bazy danych podczas procesu programowania często się zmienia, konieczna jest refaktoryzacji kodu źródłowego aplikacji odpowiedzialnego za operacje na bazie danych [3]. Obiekty są zorganizowane w inny sposób niż relacyjne bazy danych. Tabele zbudowane są z rekordów, będących w relacji z innymi tabelami na podstawie kluczy obcych. Z kolei powiązania między obiektami są realizowane w zupełnie inny sposób: jeden obiekt może zawierać drugi, używany jest też zwykle mechanizm dziedziczenia. Rozwiązaniem tego problemu jest wzorzec projektowy **Data Mapper**. Jego ideą jest utworzenie klasy, która mapuje atrybuty (i ewentualnie metody) modelu domenowego na pola w tabelach bazy danych i odwrotnie. Mapująca klasa powinna dostarczać funkcjonalności umożliwiające utworzenie obiektu na podstawie danych pochodzących z bazy oraz pozwalające na zapis i modyfikację rekordu na podstawie danych obiektu. Informacje służące do mapowania pomiędzy obiektowo zorientowanym kodem źródłowym a tabelami bazodanowymi mogą być zapisane na kilka sposobów: wewnątrz klasy *Data Mapper*, jako tablica PHP czy w zewnętrznych plikach, np. INI lub XML. To, w jaki sposób zapisane są te dane zależy od konkretnej implementacji omawianego wzorca projektowego. Diagram UML Data Mapper został przedstawiony na rysunku 2. W tym przypadku obiekt klasy *Bookmark* jest obiektem z modelu domenowego, a *BookmarkMapper* stanowi implementację wzorca **Data Mapper**. Klasa odpowiadająca części modelu domenowego powinna zawierać logikę biznesową, *Mapper* natomiast powinien zapewniać komunikację z bazą danych [9]. Wzorzec odseparowuje obiekty znajdujące się w pamięci aplikacji od bazy danych. Jest odpowiedzialny za przepływ danych pomiędzy obiektami i tabelami; izoluje warstwę aplikacji od warstwy bazodanowej. W najprostszej wersji, *Data Mapper* mapuje tabelę relacyjnej bazy danych na odpowiadającą jej klasę. Jednak takie sytuacje zdarzają się rzadko (i w takich przypadkach wygodniejsze jest użycie wzorca Active Record), więc *mapper* również zapewnia obsługę bardziej skomplikowanych problemów, takich jak np. klasa składająca się z wielu tabel czy dziedziczenie w kodzie źródłowym [12]. W aplikacji można używać tylko jednej klasy mapującej lub kilku klas opartych o omawiany wzorzec. Ciągła rozbudowa projektu sprawia, że kod '*mappera'* staje się coraz większy i trudniejszy do zarządzania. Dobrym przykładem są funkcje służące do wyszukiwania rekordów. W systemie z rozbudowaną bazą danych składającą się z wielu tabel funkcji tych jest bardzo dużo (ze względu na wyszukiwanie według wybranych kryteriów, pobieranie zestawów rekordów z tabel połączonych relacjami), więc uporządkowanie metod poprzez umieszczenie ich w kilku różnych klasach wydaje się być trafnym rozwiązaniem, ułatwiającym zarządzanie kodem źródłowym [9].



Rys. 2. Diagram UML wzorca projektowego Data Mapper [9]

Użycie wzorca projektowego **Data Mapper** zaleca się w sytuacji, gdy programista chce rozbudowywać model domenowy i modyfikować schemat bazy danych niezależnie od siebie. Zaletą takiego rozwiązania jest możliwość pracy nad modelem domenowym nie zwracając uwagi na strukturę źródła danych, zarówno podczas budowania jak i testowania systemu [9].

Model w Doctrine oparty jest na mechanizmie encji. Encja jest klasą opisującą wybrany element warstwy biznesowej tworzonej aplikacji. Głównym elementem frameworka Doctrine 2 jest klasa *EntityManager* (menadżer encji) stanowiąca centralny punkt dostępu do funkcjonalności mapowania obiektowo-relacyjnego. Interfejs tej klasy pozwala na utrwalenie obiektu w nośniku danych oraz pobranie obiektu z bazy [15].

Sama klasa napisana w języku PHP z wyróżnionymi polami i metodami nie stanowi jeszcze podstawy do mapowania jej na tabelę w relacyjnej bazie danych. Aby było to możliwe, konieczna jest specyfikacja tzw. metadanych, na podstawie których dokonane będzie mapowanie obiektowo-relacyjne. Doctrine 2 pozwala na umieszczenie metadanych na kilka sposobów: za pomocą adnotacji, w pliku XML oraz w pliku YAML. Sposób zapisu tych danych nie ma znaczenia dla szybkości przetwarzania skryptu. Dzieje się tak, ponieważ w chwili wczytania ze źródła informacji pozwalających na mapowanie, zostają one przekazane do instancji klasy *ClassMetadata*, która jest następnie przechowywana w pamięci podręcznej metadanych. Istotną informacją jest też konieczność używania w pakiecie Symfony tylko jednego sposobu definicji metadanych [15]. Na potrzeby testów posłużono się zapisem tych danych w klasie przy pomocy adnotacji. Jest to rodzaj specjalnych komentarzy, poprzedzonych znakiem @, które umożliwiają osadzenie w sekcji dokumentacyjnej zmiennej czy klasy metadanych, wykorzystywanych później przez odpowiednie narzędzia. Doctrine definiuje własny zestaw adnotacji opisujących informacje dotyczące mapowania obiektów.

Do wykonania operacji wstawienia, usunięcia, pobrania i zmodyfikowania encji (operacje typu CRUD) odpowiadającej rekordowi w bazie danych konieczne jest połączenie go z menadżerem encji. Oprócz klasy *EntityManager* używana jest inna ważna klasa - *UnitOfWork*. Najprościej *Unit of Work* można opisać jako transakcję na poziomie obiektowym.

Relacje pomiędzy tabelami bazy danych są przenoszone na środowisko obiektowe przy pomocy tzw. asocjacji. Asocjacje pomiędzy klasami *Entity* są reprezentowane tak jak połączenia między obiektami w kodzie PHP - na podstawie referencji do innych obiektów lub kolekcji obiektów. W Doctrine 2 dostępne są dwa typy relacji: jednokierunkowe (ang. unidirectional), w których obsługa powiązania znajduje się tylko w jednej klasie uczestniczącej w relacji oraz dwukierunkowe (ang. bidirectional), mające zaimplementowaną obsługę asocjacji w dwóch klasach uczestniczących. Wprowadzenie dwukierunkowości powoduje, że każdą z dwóch klas uczestniczących w relacji można opisać jako właściciela (ang. owning side) lub jako klasę odwrotną relacji (ang. inverse side) [4].

4. Porównanie frameworków ORM dla PHP

Platforma testowa, na której uruchamiano aplikację działającą zarówno z Propelem jak i Doctrine to XAMPP w wersji 1.8.1 [15]. Z punktu widzenia tworzonej aplikacji w skład tego pakietu wchodzą serwer bazy danych MySQL oraz kompletny interpreter języka PHP. Dane serwera, na którym wykonano testy zostały zamieszczone w tabeli 1. Wszystkie testy zostały przeprowadzone na maszynie z zainstalowanym systemem operacyjnym Windows 7 w wersji 64-bitowej [15].

Testy te zostały przeprowadzone na najczęściej używanej z językiem PHP relacyjnej bazie danych MySQL [12]. Wybór tego silnika baz danych zdeterminowany był przez dwa czynniki. Po pierwsze - MySQL jest niezwykle popularny, o czym może świadczyć fakt, że wielkie korporacje takie jak Facebook, Google czy Adobe w wielu swoich rozwiązaniach używają właśnie tego silnika [16]. Drugą kwestią jest ścisła integracja MySQL z pakietem XAMPP. Więcej informacji na temat środowiska testowego i aplikacji testowej można znaleźć w [8].

Tabela 1. Dane środowiska testowego

Nazwa	Wartość
System operacyjny	Windows 7 64-bit
Procesor / pamięć RAM	Intel Core i3-3110M / 8GB DDR3
Serwer Apache	2.4.3
MySQL	5.5.27
PHP	5.4.7

W testowanej bazie danych występują relacje jeden do jednego oraz wiele do wielu, co pozwoliło zbadać wydajność działań dwóch technologii ORM stosowanych w Symfony 2.

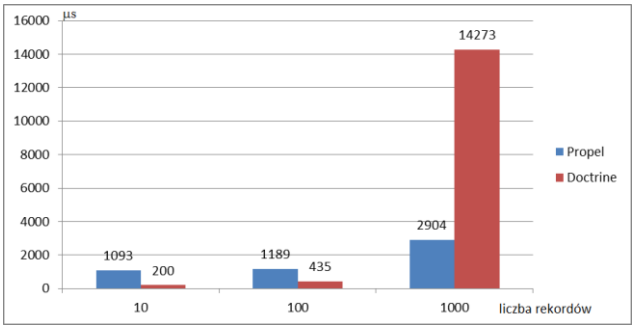
W celu praktycznego i obiektywnego porównania omawianych narzędzi ustalono scenariusze testowe. Każdy z nich, przeprowadzony dla Propela i Doctrine pozwolił określić czas wykonania kodu PHP realizującego wybrany scenariusz oraz wykorzystanie pamięci. Szczegółowe opisy scenariuszy zostały przedstawione w tabeli 2.

Tabela 2. Opis scenariuszy testowych

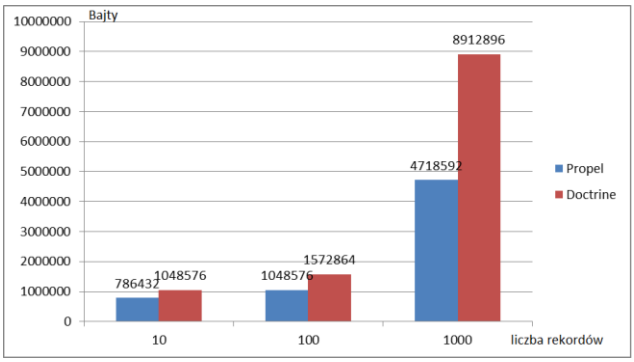
Oznaczenie	Opis
S1	*Zapis pojedynczego rekordu*. Utworzenie obiektu reprezentującego rekord i utrwalenie go w bazie danych.
S2	*Odczyt pojedynczego rekordu*. Odczyt z bazy danych pojedynczego rekordu na podstawie klucza głównego.
S3	*Zapis rekordów powiązanych*. Zapis do bazy danych rekordów w trzech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

Scenariusz testowy S1 badał zapisanie pojedynczego rekordu. W sytuacji gdy tworzonych było 10 i 100 rekordów, krótszy czas wykonania kodu realizującego pętlę zapisującą zanotowano w przypadku Doctrine. Natomiast w przypadku zapisu 1000 rekordów do bazy danych, czas wykonania kodu Doctrine okazał się blisko 5 razy dłuższy, niż podczas użycia do tego samego celu technologii Propel (rys. 3). Zestawienie użycia pamięci na rysunku 4 pokazuje, że w każdym przypadku Doctrine potrzebował więcej pamięci niż Propel.

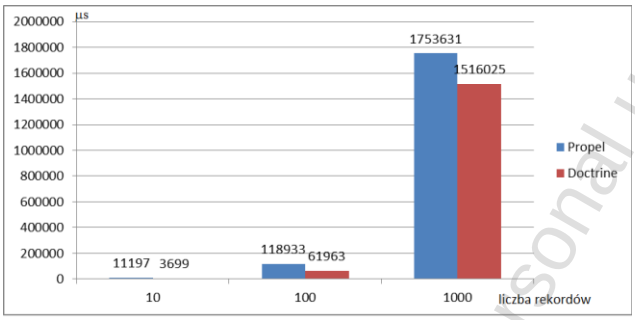
Odczyt pojedynczego rekordu z bazy został zrealizowany na podstawie scenariusza S2. Każda próba odczytu pokazała, że Propel 1.7 potrzebował dużo więcej czasu niż Doctrine 2.1 na pobranie danych i przypisanie rezultatów zapytań do obiektów. Patrząc na operację odczytu pojedynczego rekordu (S2) pod kątem użycia pamięci przez skrypt PHP do przechowania danych pobranych ze źródła, Propel potrzebuje więcej pamięci od Doctrine.



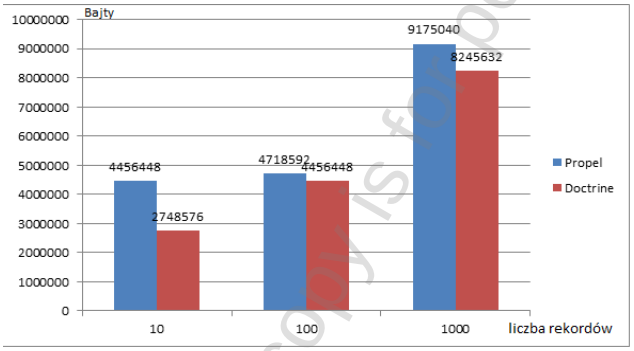
Rys. 3. Czas wykonania kodu PHP (w mikrosekundach) dla scenariusza S1



Rys. 4. Zużycie pamięci (w bajtach) dla scenariusza S1



Rys. 5. Czas wykonania kodu PHP (w mikrosekundach) dla scenariusza S4



Rys. 6. Użycie pamięci (w bajtach) dla scenariusza S4

Tabela 3. Wyniki dla scenariuszy S2 i S3

Scenariusz	Czas wykonania (µs) dla rekordów:			Zużycie pamięci (B) dla rekordów:		
	10	100	1000	10	100	1000
S2 - Doctrine	47	85	658	1048576	1572864	5242880
S2 - Propel	1048	1108	1776	1572864	1835008	5242880
S3 - Doctrine	109	1041	36095	2359296	3407872	35651584
S3 - Propel	1113	1585	7061	1048576	2621440	15728640

Scenariusz S3 dotyczył zapisu rekordów w tabelach połączonych relacjami. W ostatnim scenariuszu S4 testowano pobranie z bazy danych uprzednio zapisanych rekordów połączonych relacją. Wyniki testów dla scenariusza S4 przedstawia rysunek 5 i rysunek 6. Wyniki testów dla scenariuszy S2 i S3 przedstawione są w tabeli 3.

5. Wnioski

W przypadku odczytu rekordów znajdujących się w jednej tabeli (S2) lub też w wielu tabelach bazy danych połączonych relacją (S4), zarówno w kwestii czasu wykonania kodu PHP realizującego te operacje jak i kwestii użycia pamięci serwera, w każdej z trzech prób lepiej spisała się technologia Doctrine 2.1. Testy dotyczące zapisu danych do bazy (scenariusz S1 i S3) nie dały już tak jednoznacznych wyników. Mniejsze (a więc korzystniejsze z punktu widzenia wydajności) użycie pamięci w przypadku zapisu 10, 100 oraz 1000 rekordów oferuje Propel. Rezultat badania dotyczącego czasu wykonania kodu dla dwóch omawianych scenariuszy zapisu pokazuje natomiast, że w przypadku utrwalania 10 i 100 rekordów kod PHP wykonuje się szybciej z użyciem Doctrine. Podczas testu zapisywania obiektów połączonych ze sobą relacjami, zdecydowanie szybszy okazał się jednak Propel. Na podstawie badań nie można jednoznacznie stwierdzić, która technologia ORM jest wydajniejsza, ale można podać obszary w których odpowiednio Propel lub Doctrine są efektywniejsze.

Literatura

[1] Barnes J. M.: Object-Relational Mapping as a Persistence for Object-Oriented Applications, Macalester College Honor Projects, 2007.
[2] Czarnecki J.: ORM w PHP z wykorzystaniem wzorca Active Record. Programista 3/2014 (22), Dom Wydawniczy Anna Adamczyk, 2014.
[3] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R.: Patterns of Enterprise Application Architecture, Addison Wesley, 2002.
[4] Gajda W.: Symfony 2 od podstaw, Helion, 2012.
[5] Hayder H.: Object-Oriented Programming with PHP5, Packt Publishing, 2007.
[6] Laplante Philllip A.: What every engineer should know about software engineering, CRC Press, 2007.
[7] McArthur C.: Pro PHP: Patterns, Frameworks, Testing and More, Apress, 2008.
[8] Sławiński A.: Porównanie technologii ORM wykorzystywanych w Symfony, praca magisterska, Politechnika Lubelska 2014.
[9] Sweat J.E.: Architect's Guide to PHP Design Patterns, Marco Tabini & Associates Inc., 2005.
[10] Porębski B., Przystalski K., Nowak L.: Building PHP Applications with Symfony, CakePHP and ZendFramework, Wiley Publishing, 2011.
[11] http://blog.blueage-software.com/post/Zarys-technologii-ORM.aspx
[12] http://db-engines.com/en/ranking
[13] http://propelorm.org/Propel/documentation
[14] http://symfony.com/doc/current/book/index.html
[15] https://www.apachefriends.org/pl/index.html
[16] http://www.mysql.com/why-mysql/

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 i języki programowania.

Mgr inż. Arkadiusz Slawiński
e-mail: slawinski.arkadiusz@gmail.com

Absolwent kierunku Informatyka Politechniki Lubelskiej (specjalność: Technologie Wytwarzania Oprogramowania). Obecnie pracuje na stanowisku programisty PHP w lubelskiej firmie Imaginalis, gdzie uczestniczy w tworzeniu zaawansowanych systemów internetowych bazujących na Yii Framework oraz bazach danych MySQL. Oprócz technologii działających po stronie serwera, interesuje się narzędziami służącymi do tworzenia front-endu aplikacji. Od kilku miesięcy poznaje framework PHP Symfony 2.

otrzymano/received: 28.11.2014 *przyjęto do druku/accepted: 04.12.2014*