

NAJPOPULARNIEJSZY WZORZEC DLA APLIKACJI WWW!

Apress®

Wzorzec MVC w PHP dla profesjonalistów

Chris Pitt

Helion



Tytuł oryginału: Pro PHP MVC

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-7015-4

Original edition copyright © 2012 by Chris Pitt.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Polish edition copyright © 2013 by HELION SA.
All rights reserved.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/wzomvc>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/wzomvc.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

- O autorze 13
- O korektorze merytorycznym 15
- Podziękowania 17
- Wprowadzenie 19
- Rozdział 1. Wprowadzenie do MVC 21
 - Czym jest MVC? 21
 - Zalety wzorca MVC 22
 - Popularne szkielety MVC 23
 - CodeIgniter 23
 - Zend Framework 24
 - CakePHP 24
 - Wzorce projektowe 24
 - Singleton 25
 - Rejestr 25
 - Fabryka 26
 - Obserwator 26
 - Tworzenie własnego szkieletu 27
 - Cele 27
- Rozdział 2. Podstawy 29
 - Cele 29
 - Automatyczne ładowanie 29
 - Przestrzenie nazw 30
 - Leniwe ładowanie 31
 - Wyjątki 32
 - Metody typów 33
 - Metadane 35
 - Pytania 42
 - Odpowiedzi 43
 - Ćwiczenia 43

Rozdział 3. Klasa bazowa45
Cele 45
Metody pobierające i ustawiające 45
Magiczne metody 48
Introspekcja 49
Przezroczyste metody dostępne 52
Pytania 56
Odpowiedzi 56
Ćwiczenia 56

Rozdział 4. Konfiguracja57
Cele 57
Tablice asocjacyjne 57
Pliki INI 58
Pytania 64
Odpowiedzi 64
Ćwiczenia 64

Rozdział 5. Buforowanie65
Cele 65
Wąskie gardła wydajności 65
Kod 66
Pytania 72
Odpowiedzi 72
Ćwiczenia 72

Rozdział 6. Rejestr73
Cele 73
Singleton 73
Rejestr 75
Pytania 77
Odpowiedzi 77
Ćwiczenia 77

Rozdział 7. Trasowanie79
Cele 79
Definiowanie tras 79
Klasy tras 80
Klasa Router 82
Pytania 90
Odpowiedzi 90
Ćwiczenia 90

Rozdział 8. Szablony91
Cele 91
Idea 91
Alternatywne rozwiązania 92
Implementacja 92
Korzyści 107
Pytania 115
Odpowiedzi 115
Ćwiczenia 115

Rozdział 9. Bazy danych	117
Cele	117
Implementacja	117
Konektory	119
Zapytania	123
Pytania	141
Odpowiedzi	141
Ćwiczenia	141
Rozdział 10. Modele	143
Cele	143
Idea	143
Implementacja	144
Budowanie kodu SQL	146
Modyfikowanie rekordów	155
Nie rozdrabniajmy się!	167
Pytania	167
Odpowiedzi	167
Ćwiczenia	168
Rozdział 11. Testowanie	169
Cele	169
Testowanie jednostkowe	169
Klasa testowa	170
Bufor	171
Pokrycie	171
Testy	172
Konfiguracja	174
Pokrycie	174
Testy	175
Baza danych	175
Pokrycie	176
Testy	176
Model	183
Pokrycie	183
Testy	184
Szablon	186
Pokrycie	186
Testy	186
A niech to!	189
Pytania	189
Odpowiedzi	189
Ćwiczenia	189
Rozdział 12. Struktura	191
Cele	191
Baza danych	192
Foldery	192
Pytania	193
Odpowiedzi	193

Rozdział 13. Rozruch aplikacji	195
Cele	195
Kiedy plik nie jest plikiem?	195
Przepisywanie adresów URL	196
Plik index.php	197
Konfiguracja	198
Baza danych	199
Bufor	200
Kontroler	201
Widoki	201
Renderowanie	203
Pytania	208
Odpowiedzi	209
Ćwiczenia	209
Rozdział 14. Rejestracja i logowanie	211
Cele	211
Biblioteki wspólne	211
Model użytkownika	212
Rejestracja	215
Sesje	218
Logowanie	221
Pytania	227
Odpowiedzi	227
Ćwiczenia	228
Rozdział 15. Wyszukiwanie	229
Cele	229
Rozszerzanie implementacji	230
Żądania adresów URL	232
Wyszukiwanie	239
Pytania	245
Odpowiedzi	245
Ćwiczenia	245
Rozdział 16. Ustawienia	247
Cele	247
Sprawdzanie danych	247
Walidacja na zawołanie	252
Ustawienia	253
Pytania	256
Odpowiedzi	256
Ćwiczenia	257
Rozdział 17. Udostępnianie treści	259
Cele	259
Strony błędów	259
Znajomości	263
Udostępnianie treści	268
Pytania	272
Odpowiedzi	272
Ćwiczenia	272

Rozdział 18. Zdjęcia	273
Cele	273
Obsługa wysyłania plików	273
Zdjęcia użytkowników	274
Wyświetlanie zdjęć w profilu	278
Pytania	279
Odpowiedzi	279
Ćwiczenia	279
Rozdział 19. Rozszerzenia	281
Cele	281
Foxy	281
Własne czcionki w CSS	281
Budowa pośrednika	282
Zastosowanie klasy Proxy	286
Imagine	288
Obserwator	290
Synchroniczność	290
Kod	292
Zdarzenia	293
Wtyczki	296
Pytania	299
Odpowiedzi	299
Ćwiczenia	300
Rozdział 20. Administracja	301
Cele	301
Czym jest CMS?	301
Administratorzy	301
Logowanie	302
Użytkownicy	307
Zdjęcia	312
Pytania	314
Odpowiedzi	314
Ćwiczenia	314
Rozdział 21. Testowanie	315
Cele	315
Pytania	318
Odpowiedzi	318
Ćwiczenia	318
Rozdział 22. CodeIgniter: rozruch	319
Cele	319
Dlaczego CodeIgniter?	319
Dlaczego nie CodeIgniter?	320
Przepisywanie adresów URL	320
Trasy	321
Pytania	321
Odpowiedzi	322
Ćwiczenia	322

Rozdział 23. CodeIgniter: MVC	323
Cele	323
Różnice	323
Modele	323
Kontrolery	327
Pytania	335
Odpowiedzi	335
Ćwiczenia	336
Rozdział 24. CodeIgniter: rozszerzanie	337
Cele	337
Wysyłanie plików	337
Biblioteki zewnętrzne	342
Rozszerzanie rdzenia	345
Pytania	346
Odpowiedzi	346
Ćwiczenia	346
Rozdział 25. CodeIgniter: testowanie	347
Cele	347
Narzędzia	347
Inne możliwości	348
Pytania	348
Odpowiedzi	348
Ćwiczenia	348
Rozdział 26. Zend Framework: rozruch aplikacji	349
Cele	349
Dlaczego Zend Framework?	349
Dlaczego nie Zend Framework?	350
Wstępna konfiguracja	350
Trasy	350
Pytania	351
Odpowiedzi	352
Ćwiczenia	352
Rozdział 27. Zend Framework: MVC	353
Cele	353
Różnice	353
Modele	353
Kontrolery	359
Pytania	367
Odpowiedzi	367
Ćwiczenia	367
Rozdział 28. Zend Framework: rozszerzanie	369
Cele	369
Wysyłanie plików	369
Biblioteki zewnętrzne	374
Pytania	376
Odpowiedzi	376
Ćwiczenia	376

Rozdział 29. Zend Framework: testowanie	377
Cele	377
Instalowanie PEAR	377
Windows	377
Unix/Linux/BSD	378
Mac OS X	378
Instalowanie PHPUnit	378
Uruchamianie testów	378
Dodawanie testów	378
Pytania	380
Odpowiedzi	380
Rozdział 30. CakePHP: rozruch	381
Cele	381
Dlaczego CakePHP?	381
Dlaczego nie CakePHP?	382
Rozpoczynanie pracy	382
Trasy	382
Pytania	383
Odpowiedzi	383
Ćwiczenia	383
Rozdział 31. CakePHP: MVC	385
Cele	385
Modele	385
Kontrolery	386
Czynności końcowe	388
Pytania	392
Odpowiedzi	392
Ćwiczenia	392
Rozdział 32. CakePHP: rozszerzanie	393
Cele	393
Wysyłanie plików	393
Zewnętrzne biblioteki	396
Wtyczki	396
Katalog Vendor	397
Pytania	398
Odpowiedzi	398
Ćwiczenia	398
Rozdział 33. CakePHP: testowanie	399
Cele	399
Testowanie	399
Pytania	401
Odpowiedzi	401
Ćwiczenia	401

Dodatek A Konfiguracja serwera sieciowego403

 Cele 403

 Windows 403

 Krok 1. 403

 Krok 2. 406

 Krok 3. 408

 Linux 411

 Krok 1. 411

 Krok 2. 411

 Krok 3. 414

 Krok 4. 414

 Mac OS X 415

 Krok 1. 415

 Krok 2. 416

 Krok 3. 418

 Testowanie 419

Skorowidz.....421



Modele

Do tej pory tworzyliśmy klasy stanowiące podstawę naszego szkieletu, ale niemające wielkiego wpływu na sposób, w jaki będą z niego korzystać przyszłe aplikacje. Wkrótce to się zmieni.

Jak wspominałem wcześniej, moc obliczeniowa jest bezużyteczna, jeśli nie ma danych do przetwarzania. Większość projektów, nad jakimi będziesz kiedykolwiek pracować, ma jakieś wspólne cechy, np. przechowywanie, przekształcanie oraz zwracanie i wyświetlanie danych. Wszystkie te czynności można wykonywać z pomocą bazy danych.

W rozdziale 9. dowiedziałeś się, jak utworzyć rozszerzalną bibliotekę obsługi baz danych, oraz przestudiowałeś przypadek bazy MySQL. Bezpośrednie posługiwanie się bazą danych to jeden ze sposobów na osiągnięcie celu, ale robienie tego w kontrolerze nie jest dobrym pomysłem. Dlatego między innymi tworzy się modele.

Cele

- Zrozumienie, czym są modele.
- Zbudowanie klasy modelu do ułatwienia obsługi ogólnych, powtarzalnych czynności.

Idea

Modele stanowią warstwę izolacyjną dla mechanizmów bezpośredniej komunikacji z bazą danych i zewnętrznymi usługami. Modele, które zbudujemy w tym rozdziale, będą udostępniać prosty interfejs do wykonywania zmian w bazie danych.

■ **Uwaga** Przedstawiony w tym rozdziale sposób definiowania modeli czasami nazywa się odwzorowaniem lub mapowaniem obiektowo-relacyjnym (ang. *Object-Relational Mapping* — ORM). Biblioteka ORM stanowi szczególną warstwę komunikacyjną między dwoma systemami powiązanymi ze sobą ze względu na dane. Więcej na ten temat można przeczytać na stronie http://en.wikipedia.org/wiki/Object-relational_mapping.

Należy podkreślić, że zastosowanie modeli nie ogranicza się do baz danych. Mogą być używane do łączenia się z dowolnymi usługami zewnętrznymi i mogą udostępniać wygodny interfejs dla kontrolerów. My jednak skupimy się na modelu ORM.

Równie dobrze możemy kiedyś otrzymać zadanie napisania modelu łączącego się z serwisem Flickr w celu pobierania zdjęć albo pozwalającego wysyłać lub pobierać pliki przy użyciu usługi chmurowej typu Amazon S3. Modele służą po prostu do przechowywania zasobów niedostępnych bezpośrednio w widoku, czyli wykorzystywanych przez kontrolery do łączenia widoków i modeli.

Implementacja

W bazach danych można przechowywać wiele różnych rodzajów danych. W naszych modelach będziemy potrzebować tylko kilku. Oto one.

- **Automatyczna numeracja**
Pola z automatyczną numeracją generują rosnące wartości liczbowe i najczęściej są używane do identyfikacji.
- **Tekst**
Pola tekstowe zwykle typu `varchar` lub `text`, w zależności od przewidywanej długości ich treści.
- **Liczby całkowite**
Domyślna długość pól naszego typu całkowitoliczbowego będzie wynosić 11.
- **Liczby dziesiętne**
Pola dziesiętne służą do przechowywania wartości zmiennoprzecinkowych.
- **Wartości logiczne**
Pola logiczne to w istocie pola na małe liczby całkowite (`tinyint`). Wartości `true` i `false` są w nich zamieniane na liczby całkowite.
- **Data i godzina**

Wybór tak ograniczonego zestawu typów danych jest podyktowany tym, że chcemy, aby wszystko było jak najprostsze. Ponadto mamy pewność, że typy te są obsługiwane przez zdecydowaną większość baz danych. Jak będziemy ich używać w modelach? Na listingu 10.1 przedstawiam przykład ilustrujący definicje tych typów na potrzeby naszego modelu.

Listing 10.1. Model User

```
class User extends Framework\Model
{
    /**
     * @column
     * @readwrite
     * @primary
     * @type autonumber
     */
    protected $_id;

    /**
     * @column
     * @readwrite
     * @type text
     * @length 100
     */
    protected $_first;

    /**
     * @column
```

```

    * @readwrite
    * @type text
    * @length 100
    */
    protected $_last;

    /**
     * @column
     * @readwrite
     * @type text
     * @length 100
     * @index
     */
    protected $_email;

    /**
     * @column
     * @readwrite
     * @type text
     * @length 100
     * @index
     */
    protected $_password;

    /**
     * @column
     * @readwrite
     * @type text
     */
    protected $_notes;

    /**
     * @column
     * @readwrite
     * @type boolean
     * @index
     */
    protected $_live;

    /**
     * @column
     * @readwrite
     * @type boolean
     * @index
     */
    protected $_deleted;

    /**
     * @column
     * @readwrite
     * @type datetime
     */
    protected $_created;

    /**
     * @column
     * @readwrite
     * @type datetime
     */
    protected $_modified;
}

```

Pierwszą rzeczą, jaka rzuca się w oczy w tym modelu, jest to, że zawiera on tylko kilka własności i jest bardzo prosty. Każda z tych chronionych zmiennych ma flagę `@readonly`, którą znamy, ponieważ używaliśmy już jej wcześniej. Z jej pomocą będziemy mogli wywoływać metody dostępne typu `setFirst()` czy `getCreated()` bez ich uprzedniego definiowania w klasie `Model`.

Jednak najbardziej interesują nas nowe własności. Własności klasy, które przekładają się na kolumny w bazie danych, mają flagę `@column`. Nasz kod inicjacji modelu będzie ignorował wszystkie własności pozbawione tej flagi. Własność `$id` ma flagę `@primary` oznaczającą, że jest to kolumna klucza głównego. Własności `$email`, `$password`, `$live` oraz `$deleted` mają flagę `@index` oznaczającą, że powinny być indeksowane w tabeli bazy danych.

Pozostały jeszcze flagi `@type` i `@length`. Określają typ danych własności i długości pól (a to będzie potrzebne tylko w przypadku pól tekstowych do wybierania między typami `varchar` i `text`).

Przedstawiona struktura, jeśli ma się do czegoś przydać, musi dać się przełożyć na tabelę bazy danych. Warstwa baza danych-model musi zamienić utworzone przez nas kolumny na kod SQL, który następnie będzie można wykonać w bazie danych. Po zakończeniu powinniśmy otrzymać takie polecenie SQL, jakie możesz zobaczyć na listingu 10.2.

Listing 10.2. Reprezentacja modelu *User* w postaci tabeli bazy danych MySQL

```
CREATE TABLE 'user' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'first' varchar(100) DEFAULT NULL,
  'last' varchar(100) DEFAULT NULL,
  'email' varchar(100) DEFAULT NULL,
  'password' varchar(100) DEFAULT NULL,
  'notes' text,
  'live' tinyint(4) DEFAULT NULL,
  'deleted' tinyint(4) DEFAULT NULL,
  'created' datetime DEFAULT NULL,
  'modified' datetime DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'email' ('email'),
  KEY 'password' ('password'),
  KEY 'live' ('live'),
  KEY 'deleted' ('deleted')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

-
- **Uwaga** Hasła nigdy nie należy przechowywać w postaci zwykłego tekstu, ale tutaj dla uproszczenia złamiemy tę zasadę. Pamiętaj jednak, że hasła należy szyfrować, aby nikt ich nie wykraść i nie wykorzystał do niecnych celów.
-

Budowanie kodu SQL

W kodzie SQL pokazanym na listingu 10.2 przedstawiam, jak powinien nasz model wyglądać w bazie danych. Kod ilustruje konwersję typów danych wykonywaną przez klasę `Model` między prostymi typami (tekst, liczby całkowite, wartości logiczne) a rzeczywistymi typami pól w bazie danych. Na listingu 10.3 znajduje się początek klasy `Model`.

Listing 10.3. Klasa *Model*

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;
```

```

class Model extends Base
{
    /**
     * @readwrite
     */
    protected $_table;

    /**
     * @readwrite
     */
    protected $_connector;

    /**
     * @read
     */
    protected $_types = array(
        "autonumber",
        "text",
        "integer",
        "decimal",
        "boolean",
        "datetime"
    );

    protected $_columns;
    protected $_primary;

    public function _getExceptionForImplementation($method)
    {
        return new Exception\Implementation("Metoda {$method} nie jest zaimplementowana");
    }
}

```

Początek tej klasy jest bardzo prosty. Zawiera definicje własności, dla których powinny zostać utworzone metody dostępne i metody przesłaniające generowanie wyjątków. Własność `$_types` zawiera proste typy danych rozpoznawane przez nasze modele i jest używana zarówno wewnątrz, jak i na zewnątrz do walidacji. Zanim będziemy mogli utworzyć kod SQL dla naszej tabeli modelu, musimy napisać kilka metod pomocniczych w klasie `StringMethods` (listing 10.4).

Listing 10.4. Metody odmiany klasy *StringMethods*

```

namespace Framework
{
    class StringMethods
    {
        private static $_singular = array(
            "(matr)ices$" => "\\1ix",
            "(vert|ind)ices$" => "\\1ex",
            "^ (ox)en$" => "\\1",
            "(alias)es$" => "\\1",
            "([octop|vir])i$" => "\\1us",
            "(cris|ax|test)es$" => "\\1is",
            "(shoe)s$" => "\\1",
            "(o)es$" => "\\1",
            "(bus|campus)es$" => "\\1",
            "([m|l])ices$" => "\\1ouse",
            "(x|ch|ss|sh)es$" => "\\1",
            "(m)ovies$" => "\\1\\2ovie",
            "(s)eries$" => "\\1\\2eries",
            "([^\aeiouy]|qu)ies$" => "\\1y",
            "([lr])ves$" => "\\1f",
            "(tive)s$" => "\\1",

```

```

        "(hive)s$" => "\\1",
        "([f])ves$" => "\\1fe",
        "(^analy)ses$" => "\\1sis",
        "((a)naly|(b)a|(d)iagno|(p)arenthe|(p)rogno|(s)ynop|(t)he)ses$" => "\\1\\2sis",
        "([ti])a$" => "\\1um",
        "(p)eople$" => "\\1\\2erson",
        "(m)en$" => "\\1an",
        "(s)tatuses$" => "\\1\\2tatus",
        "(c)hildren$" => "\\1\\2hild",
        "(n)ews$" => "\\1\\2ews",
        "([u])s$" => "\\1"
    );

    private static $_plural = array(
        "(^ox)$" => "\\1\\2en",
        "([m|l])ouse$" => "\\1lice",
        "(matr|vert|ind)ix|ex$" => "\\1lices",
        "(x|ch|ss|sh)$" => "\\1les",
        "([aeiouy]|qu)y$" => "\\1ies",
        "(hive)$" => "\\1s",
        "(?:([f])fe|([r])f)$" => "\\1\\2ves",
        "sis$" => "ses",
        "([ti])um$" => "\\1a",
        "(p)erson$" => "\\1eople",
        "(m)an$" => "\\1en",
        "(c)hild$" => "\\1hildren",
        "(buffal|tomat)o$" => "\\1\\2oes",
        "(bu|campu)s$" => "\\1\\2ses",
        "(alias|status|virus)" => "\\1les",
        "(octop)us$" => "\\1i",
        "(ax|cris|test)is$" => "\\1les",
        "s$" => "s",
        "$" => "s"
    );

    public static function singular($string)
    {
        $result = $string;

        foreach (self::$_singular as $rule => $replacement)
        {
            $rule = self::$_normalize($rule);

            if (preg_match($rule, $string))
            {
                $result = preg_replace($rule, $replacement, $string);
                break;
            }
        }

        return $result;
    }

    function plural($string)
    {
        $result = $string;

        foreach (self::$_plural as $rule => $replacement)
        {
            $rule = self::$_normalize($rule);

            if (preg_match($rule, $string))
            {
                $result = preg_replace($rule, $replacement, $string);
                break;
            }
        }
    }

```



```

        }
        return $result;
    }
}

```

Musimy też dodać kilka metod do klasy `Model` (listing 10.5).

Listing 10.5. *Przesłonięcia metod pobierających*

```

namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function getTable()
        {
            if (empty($this->_table))
            {
                $this->_table = strtolower(StringMethods::singular(get_class($this)));
            }

            return $this->_table;
        }

        public function getConnector()
        {
            if (empty($this->_connector))
            {
                $database = Registry::get("database");

                if (!$database)
                {
                    throw new Exception\Connector("Brak dostępnego konektora");
                }

                $this->_connector = $database->initialize();
            }

            return $this->_connector;
        }
    }
}

```

Do klasy `StringMethods` dodaliśmy dwie nowe metody (zwane metodami **odmiany**). Ich działanie polega na zamianie przy użyciu wyrażeń regularnych łańcuchów na formy w liczbie pojedynczej lub mnogiej. Następnie przesłoniliśmy metody pobierające własności `$_table` i `$_connector` w klasie `Model`. Chcemy, aby metoda `getTable()` zwracała zdefiniowaną przez użytkownika nazwę tabeli lub domyślnie pojedynczą formę nazwy klasy bieżącego modelu (przy użyciu metody PHP `get_class()` i jednej z nowych metod odmiany dodanych w klasie `StringMethods`).

Metodę `getConnector()` przesłoniliśmy po to, aby można było zwracać zawartość własności `$_connector` lub egzemplarz konektora zapisany w klasie `Registry` albo zgłaszać wyjątek `Model\Exception\Connector`. Jest to nasze pierwsze pobranie czegoś z klasy `Registry`. Robimy to w tym miejscu, ponieważ jest bardzo możliwe, że do tej pory połączenie z bazą danych będzie już nawiązane.

Potrzebujemy też metody zwracającej w uporządkowanej tablicy kolumny, zgodne z definicją w metadanych, aby zbudować zapytanie SQL. Jej kod źródłowy jest pokazany na listingu 10.6.

Listing 10.6. Metoda *getColumns()*

```

namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function getColumns()
        {
            if (empty($_columns))
            {
                $primaries = 0;
                $columns = array();
                $class = get_class($this);
                $types = $this->types;

                $inspector = new Inspector($this);
                $properties = $inspector->getClassProperties();

                $first = function($array, $key)
                {
                    if (!empty($array[$key]) && sizeof($array[$key]) == 1)
                    {
                        return $array[$key][0];
                    }
                    return null;
                };

                foreach ($properties as $property)
                {
                    $propertyMeta = $inspector->getPropertyMeta($property);

                    if (!empty($propertyMeta["@column"]))
                    {
                        $name = preg_replace("#^#", "", $property);
                        $primary = !empty($propertyMeta["@primary"]);
                        $type = $first($propertyMeta, "@type");
                        $length = $first($propertyMeta, "@length");
                        $index = !empty($propertyMeta["@index"]);
                        $readwrite = !empty($propertyMeta["@readwrite"]);
                        $read = !empty($propertyMeta["@read"]) || $readwrite;
                        $write = !empty($propertyMeta["@write"]) || $readwrite;

                        $validate = !empty($propertyMeta["@validate"]) ? $propertyMeta["@validate"]
                        ↪: false;
                        $label = $first($propertyMeta, "@label");

                        if (!in_array($type, $types))
                        {
                            throw new Exception\Type("Typ {$type} jest nieprawidłowy");
                        }

                        if ($primary)
                        {
                            $primaries++;
                        }

                        $columns[$name] = array(
                            "raw" => $property,
                            "name" => $name,

```

```

        "primary" => $primary,
        "type" => $type,
        "length" => $length,
        "index" => $index,
        "read" => $read,
        "write" => $write,

        "validate" => $validate,
        "label" => $label
    );
}

if ($primaries != 1)
{
    throw new Exception\Primary("Klasa {$class} musi mieć dokładnie jedną kolumnę
    ↳@primary");
}

$this->_columns = $columns;
}

return $this->_columns;
}
}
}

```

Metoda `getColumns()` może wyglądać tak, jakby pochodziła z biblioteki `Template`, którą zbudowaliśmy w poprzednim rozdziale, ale w istocie jest naprawdę bardzo prosta. Tworzy egzemplarz klasy `Inspector` i funkcję pomocniczą (`$first`) zwracającą pierwszy element z tablicy metadanych. Następnie przegląda za pomocą pętli wszystkie własności modelu i odsiewa wszystkie te, które mają flagę `@column`. Wszystkie inne własności są w tym procesie ignorowane.

Następnie sprawdzana jest flaga `@type` kolumny, aby upewnić się, że jest poprawna — jeśli nie jest, następuje zgłoszenie wyjątku `Model\Exception\Type`. Jeżeli natomiast typ kolumny jest prawidłowy, zostaje ona dodana do własności `_columns`. Każda poprawna kolumna `$primary` powoduje zwiększenie zmiennej `$primaries`, która jest weryfikowana na końcu metody, aby sprawdzić, czy została zdefiniowana tylko jedna kolumna z kluczem głównym. Mówiąc krótko, metoda ta pobiera definicję modelu użytkownika i zwraca tablicę asocjacyjną danych kolumn. Utworzymy też dwie metody pomocnicze zwracające indywidualne kolumny z tej tablicy (listing 10.7).

Listing 10.7. Metody pobierające kolumny

```

namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function getColumn($name)
        {
            if (!empty($this->_columns[$name]))
            {
                return $this->_columns[$name];
            }
            return null;
        }

        public function getPrimaryColumn()
        {

```

```

        if (!isset($this->_primary))
        {
            $primary;

            foreach ($this->columns as $column)
            {
                if ($column["primary"])
                {
                    $primary = $column;
                    break;
                }
            }

            $this->_primary = $primary;
        }

        return $this->_primary;
    }
}

```

Metoda `getColumn()` zwraca kolumny wg nazw. Można zauważyć, że przyjęto założenie, iż kolumny jako własności klas mają nazwy zaczynające się od znaku podkreślenia. Założenie to jest obecne także w metodzie `getColumn()`, która szuka kolumny bez znaku podkreślenia. Kolumny zadeklarowane jako własności kolumnowe będą podobne do `_firstName`, natomiast w publicznych metodach dostępowych będziemy odwoływać się do nich za pomocą wyrażenia `setFirstName/firstName`. Metoda `getPrimaryColumn()` przegląda kolumny za pomocą pętli i zwraca kolumnę oznaczoną jako `primary`.

Kiedy już dysponujemy techniką opisu i mamy możliwość iteracji przez własności naszego modelu (a więc też i tabele bazy danych), możemy rozpocząć tworzenie kodu SQL, za pomocą którego zostanie w końcu rzeczywiście zbudowana tabela. Jako że tabele są tworzone przy użyciu składni SQL i składnia ta jest zazwyczaj zależna od konkretnej bazy danych, napiszemy metodę dla naszych klas konektorów, przy użyciu której będziemy mogli przenieść struktury tych tabel do bazy danych. Metoda ta będzie zamieniać tablicę kolumn na odpowiedni kod SQL, co pokazano na listingu 10.8.

Listing 10.8. Metoda `sync()` klasy `Database\Connector\Mysql`

```

namespace Framework\Database\Connector
{
    use Framework\Database as Database;
    use Framework\Database\Exception as Exception;

    class Mysql extends Database\Connector
    {
        public function sync($model)
        {
            $lines = array();
            $indices = array();
            $columns = $model->columns;
            $template = "CREATE TABLE `{$s}` (\n%s,\n%s\n) ENGINE=%s DEFAULT CHARSET=%s;";

            foreach ($columns as $column)
            {
                $raw = $column["raw"];
                $name = $column["name"];
                $type = $column["type"];
                $length = $column["length"];

                if ($column["primary"])
                {
                    $indices[] = "PRIMARY KEY (`{$name}`)";
                }
                if ($column["index"])
                {

```

```

    $indices[] = "KEY `{$name}` (`{$name}`)";
}

switch ($type)
{
    case "autonumber":
    {
        $lines[] = "`{$name}` int(11) NOT NULL AUTO_INCREMENT";
        break;
    }
    case "text":
    {
        if ($length !== null && $length <= 255)
        {
            $lines[] = "`{$name}` varchar({$length}) DEFAULT NULL";
        }
        else
        {
            $lines[] = "`{$name}` text";
        }
        break;
    }
    case "integer":
    {
        $lines[] = "`{$name}` int(11) DEFAULT NULL";
        break;
    }
    case "decimal":
    {
        $lines[] = "`{$name}` float DEFAULT NULL";
        break;
    }
    case "boolean":
    {
        $lines[] = "`{$name}` tinyint(4) DEFAULT NULL";
        break;
    }
    case "datetime":
    {
        $lines[] = "`{$name}` datetime DEFAULT NULL";
        break;
    }
}

}

$table = $model->table;
$sql = sprintf(
    $template,
    $table,
    join(",\n", $lines),
    join(",\n", $indices),
    $this->_engine,
    $this->_charset
);

$result = $this->execute("DROP TABLE IF EXISTS {$table};");
if ($result === false)
{
    $error = $this->lastError;
    throw new Exception\Sql("Wystąpił błąd w zapytaniu: {$error}");
}

$result = $this->execute($sql);
if ($result === false)
{

```

```

        $error = $this->lastError;
        throw new Exception\Sql("Wystąpił błąd w zapytaniu: {$error}");
    }

    return $this;
}
}
}

```

Metoda `sync()` konwertuje własności na zapytania SQL i ostatecznie na fizyczne tabele w bazie danych. Robi to poprzez pobranie najpierw listy kolumn, a następnie wywołanie metody `getColumns()` modelu. Podczas przeglądania kolumn za pomocą pętli tworzy tablice indeksów i łańcuchów pól.

-
- **Uwaga** Polecenie SQL `CREATE TABLE` jest tworzone z myślą o bazie danych MySQL i można je znaleźć w klasie `Database\Connector`. Jeśli używasz innej bazy danych i występują problemy z wykonaniem otrzymanego zapytania SQL, może być konieczne przesłonięcie metody `sync()` w podklasie `Database\Connector`, aby zdefiniować poprawną składnię dla swojej bazy.
-

Wszystkie łańcuchy pól po utworzeniu zostają połączone (razem z indeksami) i zastosowane do łańcucha `CREATE TABLE $template`. Tworzone i wykonywane jest też polecenie `DROP TABLE`, aby oczyścić miejsce dla nowej tabeli. Następnie zostaje wykonane polecenie SQL tworzące tabelę. Wszelkie błędy SQL powodują zgłoszenie wyjątku `Database\Exception\Sql`. Przykład użycia metody `sync()` pokazuję na listingu 10.9.

Listing 10.9. Przykład użycia metody `sync()`

```

$database = new Framework\Database(array(
    "type" => "mysql",
    "options" => array(
        "host" => "localhost",
        "username" => "prophpmvc",
        "password" => "prophpmvc",
        "schema" => "prophpmvc"
    )
));
$database = $database->initialize()->connect();

$user = new User(array(
    "connector" => $database
));
$database->sync($user);

```

W kodzie tym najpierw tworzone jest połączenie z bazą danych, które następnie zostaje przypisane jako konektor do egzemplarza modelu `User`. Na koniec wywoływana jest metoda `sync()` na egzemplarzu modelu `User` i zostaje utworzona tabela w bazie danych. Ponieważ w klasie `Registry` sprawdzaliśmy istnienie klucza `database`, ten sam efekt możemy osiągnąć przy użyciu kodu pokazanego na listingu 10.10.

Listing 10.10. Alternatywny sposób użycia konektora

```

$database = new Database(array(
    "type" => "mysql",
    "options" => array(
        "host" => "localhost",
        "username" => "prophpmvc",
        "password" => "prophpmvc",
        "schema" => "prophpmvc"
    )
));
Registry::set("database", $database->initialize()->connect());
$database->sync(new User());

```

Modyfikowanie rekordów

Mamy gotową tabelę bazy danych. Musimy teraz rozszerzyć nasz model i dodać mechanizm bezpośredniej współpracy z bazą danych oraz prosty interfejs. Najpierw sprawimy, że nasz model będzie łąadował rekord, kiedy zostanie podana wartość kolumny głównej. Zrobimy to, modyfikując konstruktor i dodając metodę `load()`, co pokazują na listingu 10.11.

Listing 10.11. Metody `__construct()` i `load()` modelu

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function __construct($options = array())
        {
            parent::__construct($options);
            $this->load();
        }

        public function load()
        {
            $primary = $this->primaryColumn;

            $raw = $primary["raw"];
            $name = $primary["name"];

            if (!empty($this->$raw))
            {
                $previous = $this->connector
                    ->query()
                    ->from($this->table)
                    ->where("{ $name} = ?", $this->$raw)
                    ->first();

                if ($previous == null)
                {
                    throw new Exception\Primary("Nieprawidłowy klucz główny");
                }

                foreach ($previous as $key => $value)
                {
                    $prop = "_{$key}";
                    if (!empty($previous->$key) && !isset($this->$prop))
                    {
                        $this->$key = $previous->$key;
                    }
                }
            }
        }
    }
}
```

Jeśli wrócimy do klasy `Base`, przypomnimy sobie, że tablica asocjacyjna, którą przekazujemy do konstruktora, jest stosowana do obecnych metod pobierających i ustawiających. Oznacza to, że możemy załadować istniejący rekord, kiedy dostarczymy do funkcji konstruktora odpowiednią parę klucz-wartość pasującą do nazwy kolumny głównej i wartość korespondującą z istniejącym rekordem.

Metoda `load()` znacznie upraszcza proces pobierania rekordów. Określa kolumnę główną modelu i sprawdza, czy nie jest pusta. W ten sposób dowiadujemy się, czy klucz główny został dostarczony, co z kolei pozwala znaleźć szukany rekord. Jeśli własność klasy klucza głównego jest pusta, przyjmujemy, że ten egzemplarz modelu służy do utworzenia nowego rekordu, i niczego dalej nie robimy.

Aby załadować rekord z bazy danych, pobieramy konektor bieżącego modelu. Jeśli nie uda się znaleźć żadnego konektora, wykonywanie zostaje zatrzymane. Następnie tworzymy zapytanie do bazy danych o ten rekord, bazując na wartości własności kolumny klucza głównego. Jeśli nie zostanie znaleziony żaden rekord, następuje zgłoszenie wyjątku `Model\Exception\Primary`. Dzieje się tak, gdy wartość kolumny klucza głównego jest podana, ale nie reprezentuje poprawnego identyfikatora rekordu w tabeli bazy danych.

Na koniec za pomocą pętli przeglądamy dane załadowanego rekordu i ustawiamy tylko te wartości własności, które nie zostały ustawione w metodzie `__construct()`. Dzięki temu żadne dane nie zostaną utracone po zainicjowaniu modelu. Kolejną metodą przydatną w naszym modelu jest metoda, która pozwala tworzyć i modyfikować rekordy. Jej kod przedstawiam na listingu 10.12.

Listing 10.12. *Metoda `save()`*

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function save()
        {
            $primary = $this->primaryColumn;

            $raw = $primary["raw"];
            $name = $primary["name"];

            $query = $this->connector
                ->query()
                ->from($this->table);

            if (!empty($this->$raw))
            {
                $query->where("{ $name } = ?", $this->$raw);
            }

            $data = array();
            foreach ($this->columns as $key => $column)
            {
                if (!$column["read"])
                {
                    $prop = $column["raw"];
                    $data[$key] = $this->$prop;
                    continue;
                }

                if ($column != $this->primaryColumn && $column)
                {
                    $method = "get".ucfirst($key);
                    $data[$key] = $this->$method();
                    continue;
                }
            }

            $result = $query->save($data);

            if ($result > 0)
```



```

        {
            $this->$raw = $result;
        }

        return $result;
    }
}
}

```

Metoda `save()` tworzy egzemplarz zapytania i dotyczy tabeli związanej z klasą `Model`. Stosuje klauzulę `WHERE`, jeśli wartość własności klucza głównego nie jest pusta, oraz buduje tablicę danych z kolumn zwróconych przez metodę `getColumns()`. Na koniec wywołuje metodę `save()` egzemplarza zapytania, aby zapisać dane w bazie danych. Jako że klasa `Database\Connector` wykonuje polecenia `INSERT` lub `UPDATE`, posługując się kryteriami w klauzuli `WHERE`, metoda ta wstawi nowy rekord albo zaktualizuje istniejący w zależności od tego, czy własność klucza głównego ma wartość, czy nie. Ostatnie dwie metody modyfikujące, które będą potrzebne, noszą nazwy `delete()` i `deleteAll()`; usuwają one dane z bazy danych. Ich kod źródłowy znajduje się na listingu 10.13.

Listing 10.13. Metody `delete()` i `deleteAll()`

```

namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public function delete()
        {
            $primary = $this->primaryColumn;

            $raw = $primary["raw"];
            $name = $primary["name"];

            if (!empty($this->$raw))
            {
                return $this->connector
                    ->query()
                    ->from($this->table)
                    ->where("{ $name } = ?", $this->$raw)
                    ->delete();
            }
        }

        public static function deleteAll($where = array())
        {
            $instance = new static();

            $query = $instance->connector
                ->query()
                ->from($instance->table);

            foreach ($where as $clause => $value)
            {
                $query->where($clause, $value);
            }

            return $query->delete();
        }
    }
}

```

Metoda `delete()` jest najprostsza z wszystkich metod modyfikujących naszego modelu. Tworzy obiekt zapytania tylko wtedy, kiedy wartość własności klucza głównego nie jest pusta, i wykonuje metodę `delete()` tego zapytania. Metoda `deleteAll()` działa bardzo podobnie, tylko jest wywoływana statycznie. Do tej pory była mowa tylko o działaniach na pojedynczych rekordach, ale przydałaby się też możliwość operowania na wielu rekordach, podobnie jak to robią metody `all()`, `first()` i `count()` naszej bazy danych. Zaczniemy od napisania metody `all()`, której kod źródłowy jest pokazany na listingu 10.14.

Listing 10.14. Metoda `all()`

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public static function all($where = array(), $fields = array(""), $order = null,
            ↪$direction = null, $limit = null, $page = null)
        {
            $model = new static();
            return $model->_all($where, $fields, $order, $direction, $limit, $page);
        }

        protected function _all($where = array(), $fields = array(""), $order = null,
            ↪$direction = null, $limit = null, $page = null)
        {
            $query = $this
                ->connector
                ->query()
                ->from($this->table, $fields);

            foreach ($where as $clause => $value)
            {
                $query->where($clause, $value);
            }

            if ($order != null)
            {
                $query->order($order, $direction);
            }

            if ($limit != null)
            {
                $query->limit($limit, $page);
            }

            $rows = array();
            $class = get_class($this);

            foreach ($query->all() as $row)
            {
                $rows[] = new $class(
                    $row
                );
            }

            return $rows;
        }
    }
}
```

Metoda `all()` to proste statyczne opakowanie dla chronionej metody `_all()`. Metoda `_all()` tworzy zapytanie, uwzględniając różne filtry i flagi zwracające wszystkie pasujące rekordy.

Powodem, dla którego zadaliśmy sobie trud opakowywania metody egzemplarzowej w statycznej, jest to, że utworzyliśmy kontekst, w którym egzemplarz modelu jest równy rekordowi tabeli. W takiej sytuacji będzie bardziej sensownie, kiedy operacje na wielu rekordach będą wykonywane przy użyciu metod klasowych.

Metoda `first()` jest podobna do metody `all()` pod tym względem, że również jest prostym statycznym opakowaniem chronionej metody egzemplarzowej. Metoda `_first()` zwraca pierwszy dopasowany rekord, co możesz zobaczyć na listingu 10.15.

Listing 10.15. Metoda `first()`

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public static function first($where = array(), $fields = array("*"), $order = null,
            ↪$direction = null)
        {
            $model = new static();
            return $model->_first($where, $fields, $order, $direction);
        }

        protected function _first($where = array(), $fields = array("*"), $order = null,
            ↪$direction = null)
        {
            $query = $this
                ->connector
                ->query()
                ->from($this->table, $fields);

            foreach ($where as $clause => $value)
            {
                $query->where($clause, $value);
            }

            if ($order != null)
            {
                $query->order($order, $direction);
            }

            $first = $query->first();
            $class = get_class($this);

            if ($first)
            {
                return new $class(
                    $query->first()
                );
            }

            return null;
        }
    }
}
```

Metoda `count()` jest podobna do swoich dwóch statycznych poprzedniczek. Metoda `_count()` zwraca liczbę dopasowanych rekordów. Na listingu 10.16 przedstawiam kod źródłowy metody `count()`.

Listing 10.16. Metoda `count()`

```
namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        public static function count($where = array())
        {
            $model = new static();
            return $model->_count($where);
        }

        protected function _count($where = array())
        {
            $query = $this
                ->connector
                ->query()
                ->from($this->table);

            foreach ($where as $clause => $value)
            {
                $query->where($clause, $value);
            }

            return $query->count();
        }
    }
}
```

Kiedy mamy cały kod, możemy zacząć wykorzystywanie naszego modelu, co pokazuję na listingu 10.17.

Listing 10.17. Przykład użycia modelu

```
$database = new Database(array(
    "type" => "mysql",
    "options" => array(
        "host" => "localhost",
        "username" => "prophpmvc",
        "password" => "prophpmvc",
        "schema" => "prophpmvc"
    )
));
$database = $database->initialize();

$user = new User(array(
    "connector" => $database
));
$database->sync($user);

$elijah = new User(array(
    "connector" => $database,
    "first" => "Chris",
    "last" => "Pitt",
    "email" => "chris@example.com",
```

```

        "password" => "password",
        "live" => true,
        "deleted" => false,
        "created" => date("Y-m-d H:i:s"),
        "modified" => date("Y-m-d H:i:s")
    ));
    $elijah->save();
    $all = User::all(array(
        "last = ?" => "Pitt"
    ));
    $elijah->delete();

```

Jest to z pewnością łatwiejsze, niż gdybyśmy musieli bezpośrednio korzystać z klas Database\Connector lub Database\Query. Mamy możliwość zachowania zgodności naszych tabel bazy danych z ORM. Możemy odpytywać i modyfikować rekordy przy użyciu prostego kodu obiektowego. Na listingu 10.18 przedstawiam kompletny kod klasy Model.

Listing 10.18. Klasa Model

```

namespace Framework
{
    use Framework\Base as Base;
    use Framework\Registry as Registry;
    use Framework\Inspector as Inspector;
    use Framework\StringMethods as StringMethods;
    use Framework\Model\Exception as Exception;

    class Model extends Base
    {
        /**
         * @readwrite
         */
        protected $_table;

        /**
         * @readwrite
         */
        protected $_connector;

        /**
         * @read
         */
        protected $_types = array(
            "autonumber",
            "text",
            "integer",
            "decimal",
            "boolean",
            "datetime"
        );

        protected $_columns;
        protected $_primary;

        public function _getExceptionForImplementation($method)
        {
            return new Exception\Implementation("Metoda {$method} nie jest zaimplementowana");
        }

        public function __construct($options = array())
        {
            parent::__construct($options);
            $this->load();
        }
    }

```

```

    }

    public function load()
    {
        $primary = $this->primaryColumn;

        $raw = $primary["raw"];
        $name = $primary["name"];

        if (!empty($this->$raw))
        {
            $previous = $this->connector
                ->query()
                ->from($this->table)
                ->where("{ $name } = ?", $this->$raw)
                ->first();

            if ($previous == null)
            {
                throw new Exception\Primary("Nieprawidłowy klucz główny");
            }

            foreach ($previous as $key => $value)
            {
                $prop = " { $key }";
                if (!empty($previous->$key) && !isset($this->$prop))
                {
                    $this->$key = $previous->$key;
                }
            }
        }
    }

    public function delete()
    {
        $primary = $this->primaryColumn;

        $raw = $primary["raw"];
        $name = $primary["name"];

        if (!empty($this->$raw))
        {
            return $this->connector
                ->query()
                ->from($this->table)
                ->where("{ $name } = ?", $this->$raw)
                ->delete();
        }
    }

    public static function deleteAll($where = array())
    {
        $instance = new static();

        $query = $instance->connector
            ->query()
            ->from($instance->table);

        foreach ($where as $clause => $value)
        {
            $query->where($clause, $value);
        }

        return $query->delete();
    }

```

```

    }

    public function save()
    {
        $primary = $this->primaryColumn;

        $raw = $primary["raw"];
        $name = $primary["name"];

        $query = $this->connector
            ->query()
            ->from($this->table);

        if (!empty($this->$raw))
        {
            $query->where("{ $name } = ?", $this->$raw);
        }

        $data = array();
        foreach ($this->columns as $key => $column)
        {
            if (!$column["read"])
            {
                $prop = $column["raw"];
                $data[$key] = $this->$prop;
                continue;
            }

            if ($column != $this->primaryColumn && $column)
            {
                $method = "get".ucfirst($key);
                $data[$key] = $this->$method();
                continue;
            }
        }

        $result = $query->save($data);

        if ($result > 0)
        {
            $this->$raw = $result;
        }

        return $result;
    }

    public function getTable()
    {
        if (empty($this->_table))
        {
            $this->_table = strtolower(StringMethods::singular(get_class($this)));
        }

        return $this->_table;
    }

    public function getConnector()
    {
        if (empty($this->_connector))
        {
            $database = Registry::get("database");

            if (!$database)
            {
                throw new Exception\Connector("Brak dostępnego konektora");
            }
        }
    }

```

```

    }

    $this->_connector = $database->initialize();
}

return $this->_connector;
}

public function getColumns()
{
    if (empty($_columns))
    {
        $primaries = 0;
        $columns = array();
        $class = get_class($this);
        $types = $this->types;

        $inspector = new Inspector($this);
        $properties = $inspector->getClassProperties();

        $first = function($array, $key)
        {
            if (!empty($array[$key]) && sizeof($array[$key]) == 1)
            {
                return $array[$key][0];
            }
            return null;
        };

        foreach ($properties as $property)
        {
            $propertyMeta = $inspector->getPropertyMeta($property);

            if (!empty($propertyMeta["@column"]))
            {
                $name = preg_replace("#^#", "", $property);
                $primary = !empty($propertyMeta["@primary"]);
                $type = $first($propertyMeta, "@type");
                $length = $first($propertyMeta, "@length");
                $index = !empty($propertyMeta["@index"]);
                $readwrite = !empty($propertyMeta["@readwrite"]);
                $read = !empty($propertyMeta["@read"]) || $readwrite;
                $write = !empty($propertyMeta["@write"]) || $readwrite;

                $validate = !empty($propertyMeta["@validate"]) ? $propertyMeta["@validate"]
                ↪: false;
                $label = $first($propertyMeta, "@label");

                if (!in_array($type, $types))
                {
                    throw new Exception\Type("Typ {$type} jest nieprawidłowy");
                }

                if ($primary)
                {
                    $primaries++;
                }

                $columns[$name] = array(
                    "raw" => $property,
                    "name" => $name,
                    "primary" => $primary,
                    "type" => $type,
                    "length" => $length,
                    "index" => $index,

```



```

        "read" => $read,
        "write" => $write,

        "validate" => $validate,
        "label" => $label
    );
    }
}

if ($primaries != 1)
{
    throw new Exception\Primary("Klasa { $class } musi mieć dokładnie jedną kolumnę
    ↳@primary");
}

$this->_columns = $columns;
}

return $this->_columns;
}

public function getColumn($name)
{
    if (!empty($this->_columns[$name]))
    {
        return $this->_columns[$name];
    }
    return null;
}

public function getPrimaryColumn()
{
    if (!isset($this->_primary))
    {
        $primary;

        foreach ($this->columns as $column)
        {
            if ($column["primary"])
            {
                $primary = $column;
                break;
            }
        }

        $this->_primary = $primary;
    }

    return $this->_primary;
}

public static function first($where = array(), $fields = array("*"), $order = null,
    ↳$direction = null)
{
    $model = new static();
    return $model->_first($where, $fields, $order, $direction);
}

protected function _first($where = array(), $fields = array("*"), $order = null,
    ↳$direction = null)
{
    $query = $this
        ->connector
        ->query()

```

```

        ->from($this->table, $fields);

    foreach ($where as $clause => $value)
    {
        $query->where($clause, $value);
    }

    if ($order != null)
    {
        $query->order($order, $direction);
    }

    $first = $query->first();
    $class = get_class($this);

    if ($first)
    {
        return new $class(
            $query->first()
        );
    }

    return null;
}

public static function all($where = array(), $fields = array(""), $order = null,
    ↪$direction = null, $limit = null, $page = null)
{
    $model = new static();
    return $model->_all($where, $fields, $order, $direction, $limit, $page);
}

protected function _all($where = array(), $fields = array(""), $order = null,
    ↪$direction = null, $limit = null, $page = null)
{
    $query = $this
        ->connector
        ->query()
        ->from($this->table, $fields);

    foreach ($where as $clause => $value)
    {
        $query->where($clause, $value);
    }

    if ($order != null)
    {
        $query->order($order, $direction);
    }

    if ($limit != null)
    {
        $query->limit($limit, $page);
    }

    $rows = array();
    $class = get_class($this);

    foreach ($query->all() as $row)
    {
        $rows[] = new $class(
            $row
        );
    }
}

```

```

        return $rows;
    }

    public static function count($where = array())
    {
        $model = new static();
        return $model->_count($where);
    }

    protected function _count($where = array())
    {
        $query = $this
            ->connector
            ->query()
            ->from($this->table);

        foreach ($where as $clause => $value)
        {
            $query->where($clause, $value);
        }

        return $query->count();
    }
}

```

Nie rozdrabniajmy się!

W tym miejscu można by mocno zagłębić się w szczegóły. Dobrze byłoby, gdyby np. nasze modele rozpoznawały połączone tabele i złączenia oraz mogły pobierać nie tylko pojedyncze rekordy, lecz również wszystkie powiązane rekordy w innych tabelach. Bez wątpienia wszystko to dałoby się zrobić, ale nie jest to niezbędne do zrozumienia koncepcji ORM ani objaśnienia, jak pasuje ona do naszego szkieletu.

W wielu bibliotekach ORM taki kod jest zaimplementowany, dzięki czemu ich użytkownicy mają kontrolę nad powiązanymi danymi baz danych. W przyszłości możemy rozszerzyć naszą bibliotekę albo nawet użyć większej, bardziej rozbudowanej biblioteki ORM, ale to, co przedstawiłem w tym rozdziale, wystarczy na nasze bieżące potrzeby.

Pytania

1. Czy modele służą tylko do pracy z bazami danych?
2. Utworzona przez nas klasa `Model` zawiera mieszankę metod egzemplarzowych i klasowych do pracy z wierszami bazy danych. Dlaczego wszystkie operacje na wierszach nie są wykonywane przy użyciu jednego rodzaju metod?

Odpowiedzi

1. Nie! Modeli można używać do wielu innych celów, np. łączenia się z zewnętrznymi API albo modyfikowania informacji systemu plików. Model to nie to samo, co ORM.
2. Czasami wygodniej pracować z wieloma wierszami jednocześnie, np. przy pobieraniu albo usuwaniu wierszy z bazy danych. W innych czynnościach natomiast udział biorą pojedyncze wiersze, np. w zapytaniach `INSERT` i `UPDATE`.

Ćwiczenia

1. W klasie `Model` znajduje się kilka dobrych metod ułatwiających pracę z bazą danych. Można dodać jeszcze kilka innych tego typu metod, np. metodę zwracającą liczbę stron dla określonego limitu i klauzuli `WHERE`. Napisz taką metodę.
2. Za pomocą podklas klasy `Model` możemy z łatwością wstawiać i aktualizować wiersze bazy danych, ale nie mamy możliwości weryfikacji danych. Dodaj kilka metod, przy użyciu których można sprawdzić poprawność danych przed wykonaniem poleceń `INSERT` i `UPDATE`.

Skorowidz

A

akcja

- `delete()`, 309, 312
- `edit()`, 307
- `fonts()`, 286, 288
- `friend()`, 264
- `index()`, 270
- `login()`, 222, 223, 252, 302, 330
- `logout()`, 256, 331
- `Messages::add()`, 270
- `profile()`, 223, 343
 - CakePHP, 397
- `register()`, 252, 276, 327, 342
 - CakePHP, 387
- `search()`, 241, 335
- `settings()`, 253, 277, 303, 335
- `thumbnails()`, 288
- `undelele()`, 309, 312
- `unfriend()`, 264
- `view()`, 307, 312, 313

asynchroniczność, 291

B

bazy danych, 117

- ekspresywne generowanie zapytań, 118
- implementacja, 117
- konektory, 119
- konfiguracja rozruchowa, 199
- modele, 143
 - rodzaje danych, 144
- modyfikowanie rekordów, 155

MySQL, 117

PostgreSQL, 119

przykład użycia, 133

sieci społecznościowe, 192

SQL Server, 119

testowanie, 175

zapytania, 123

- metody dostępne, 131

- metody pomocnicze, 125, 130

biblioteki

CSS, 281

`form_validation`, 329

Foxy, 281

Imagine, 288

PHPUnit, 348, 378

bootstrapping, *Patrz* rozruch aplikacji

bufor, 66

- konfiguracja rozruchowa, 200

- testowanie, 171

buforowanie, 65

Memcached, 66

C

CakePHP, 24, 381

- akcje, 388

- dokumentacja, 381

- kontroler, 386

- modele, 385

- szkielety, 24, 381

- testowanie, 399

- odpowiedzi na żądania formularzy, 400

- testy jednostkowe, 399

CakePHP

- trasowanie, 382
- uruchomienie, 382
- widoki, 390
- wtyczki, 396
- wysyłanie plików, 393
- zewnętrzne biblioteki, 396

CMS, 23, 301

- administrator, 301
- logowanie, 302
 - usuwanie kontrolera, 305
- użytkownicy, 307
 - edycja, 311
 - lista, 310
 - widok, 309
- zdjęcia, 312
 - edycja, 312
 - widok, 313

CodeIgniter, 23, 319

- .htaccess, 320
- biblioteki, 342
- dokumentacja, 319
- kontroler, 327
 - Users, 327
- logowanie, 329
- modele, 323
 - metody, 325
 - struktura tabeli, 325
 - User, 323
- PHPUnit, 348
- przepisywanie adresów URL, 320
- rejestracja, 339
- rozszerzanie rdzennych klas, 345
- testowanie, 347
 - testy jednostkowe, 347
 - zewnętrzne biblioteki, 348
- trasowanie, 321
- widok profilu, 330
- wysyłanie plików, 337

Content Management System, *Patrz* CMS

CRUD, 307

CSS, 281

czcionki, 282

- dodanie formatów, 282
- dodanie katalogu, 282
- trasa, 286

D

dokumentacja

- CakePHP, 381
- CodeInteger, 319
- Zend Framework, 349

F

fabryka, 26

- tworzenie obiektu konfiguracyjnego, 60
- zasada działania, 26

form_validation, 329

Foxy, 281

funkcje

- autoload(), 32
- get_include_path(), 32
- haki, 86
- include(), 31
- parse_ini_file(), 58
- parsePhp(), 58
- preg_match(), 34
- preg_split(), 34
- require(), 31
- spl_autoload_register(), 31
- unset(), 305

H

haki, 86

- @after, 86
- @before, 86
- admin(), 306

Homebrew, 417

I

identyfikacja przeglądarek, 284

Imagine, 288

- automatyczne ładowanie klas, 288

INI, 58

K

klasy, 24

- Application_Model_DbTable_File, 371
- Application_Model_DbTable_User, 355
- Application_Model_User, 355, 357
- ArrayMethods, 35

- Base, 53
 - metody, 49
- bazowa, 45
- Cache, 66, 71
 - konfiguracja automatyczna, 200
 - testowanie, 172
- Cache\Driver, 67
- Cache\Driver\Memcached, 67, 70
 - metody klasy, 68, 69
- Configuration, 59, 198
 - testowanie, 174
- Configuration\Driver, 60
- Configuration\Driver\Ini, 62
- Controller, 88, 293
 - biblioteki, 211
 - metody renderujące, 204
 - modyfikacja, 203
- Core, 198
- Curl, 235
- Database, 118
 - konfiguracja automatyczna, 199
 - testowanie, 176
- Database\Connector, 120
- Database\Connector\Mysql, 120
- Database\Query, 123, 134
- DirectoryIterator, 297
- Events, 292
 - przykład użycia, 293
- Exception, 32
- fabryka, 26
- fasadowa, 342
- File_Transfer, 369
- Files, 312
- Inspector, 40
 - metody klasy, 37
 - wewnętrzne własności, 37
- konfiguracyjna, 59
- Logger, 297
- Model, 146, 161, 249
 - metody pobierające, 149
 - testowanie, 183
- Proxy, 286
- Query, 123, 134
 - metody dostępowe, 131
 - metody pomocnicze, 125, 130
 - testowanie, 181
- Regex, 80
- Registry, 76
 - przykład użycia, 76
- rejestr, 25
- Request, 232
 - testowanie, 315
- RequestMethods, 216
- Request\Response, 235
- Router, 79, 82, 86
- Router\Route, 80
- Router\Route\Regex, 80
- Router\Route\Simple, 81
- Session, 219
- Session\Driver\Server, 220
- Shared\Controller, 294
 - ustawianie sesji, 311
 - zdarzenia, 304
- Shared\Model, 226
- singleton, 25
 - Ford, 74
 - przykład użycia, 75
- statyczne, 76
- StringMethods, 34, 147
 - metody klasy, 92
- Template, 98, 110
 - metody parsujące, 99
 - metody publiczne, 104
- Template\Implementation, 94
- Template\Implementation\Extended, 230
- Template\Implementation\Standard, 107
- Test, 170
- Thumbnail, 342
- tras, 80
- User
 - konto użytkownika, 212
- View, 201
- konektory, 119
 - testowanie, 178
- konfiguracja
 - klasa, 174
 - rozruchowa
 - aplikacja, 198
 - baza danych, 199
 - bufor, 200
 - kontroler, 201
- serwer sieciowy, 403
 - Linux, 411
 - Mac OS X, 415
 - Windows, 403
- szkieletu, 57
- testowanie, 174
- Zend Framework, 350

kontroler, 21

CakePHP, 386

CodeIgniter, 327

logowanie, 329

Users, 327

Files, 286

Home, 269

konfiguracja rozruchowa, 201

Messages, 270

Users, 224

akcje, 307

CodeIgniter, 327, 340

metody kontrolera, 275

usuwanie, 305

Zend Framework, 359

L

LAMP, 411

dbconfig-common, 413

instalowanie narzędzia tasksel, 411

Memcached, 414

MySQL, 413

phpMyAdmin, 411

ustawianie katalogu głównego, 415

wybór pakietów, 412

logowanie

CMS, 302

CodeInteger, 329

sieci społecznościowe, 221

Zend Framework, 362

M

MAMP, 415

instalacja Homebrew, 417

instalacja wget, 417

Memcached, 416, 418

ustawianie portów serwera, 416

mapa gramatyki, 96

mapowanie obiektowo-relacyjne, *Patrz* ORM

Memcached, 66

buforowanie, 66

instalacja, 406, 408, 414, 416, 418

pobieranie, 418

testowanie, 172

uruchamianie, 408

metadane, 35

komentarze, 36

metody

add(), 171, 293

addFont(), 283

addRoute(), 83

all(), 133, 158

append(), 238

array(), 101

ArrayMethods::flatten(), 81

ArrayMethods::toObject(), 60

authenticate(), 37, 89

buildDelete(), 130

buildInsert(), 130

buildSelect(), 127

buildUpdate(), 130

call(), 48, 51

clean(), 35

clone(), 74

Configuration\Driver\Ini parse(), 61

connect(), 68, 69, 178

construct(), 50, 59, 68, 74, 155, 205, 212, 240,
295, 304

count(), 133, 160

delete(), 131, 157

deleteAll(), 157

deleteFont(), 283

destruct(), 305

detectSupport(), 285

disconnect(), 69, 178

dostępowe, 47, 49

przezroczyste, 52

tworzenie, 50

each(), 98

echo(), 98

egzemplarzowe, 48

elif(), 98

else(), 98

erase(), 69, 76, 306

escape(), 178

fire(), 293

first(), 133, 159

from(), 127

get(), 52, 69, 76

get_class(), 149

getColor(), 48

getColumns(), 150, 152

getConnector(), 149

getFile(), 278

getKey(), 238

getModel(), 48

getPrimaryColumn(), 152
 getRoutes(), 83
 getTable(), 149
 getUser(), 327
 getValue(), 238
 handler(), 95
 home(), 89
 if(), 98
 include(), 61, 231
 indexOf(), 93
 init(), 89
 instance(), 75
 isFriend(), 267
 isValidService(), 68
 join(), 127
 limit(), 127
 literal(), 98
 load(), 155
 loginAction(), 364
 logoutAction(), 364
 loop(), 98
 macro(), 98
 magiczne, 48, 53
 match(), 34, 95
 matches(), 81
 normalize(), 235
 notify(), 89
 odmiany, 149
 order(), 127
 pair(), 62
 parse(), 38, 61, 105
 wynik działania, 63
 parse_ini_string(), 61
 partial(), 237
 pass(), 84
 pobierające, 46
 przesłonięcia, 149
 przezroczyste, 55
 kolumny, 151
 prepend(), 238
 proces(), 105
 profileAction(), 364, 375
 publiczne, 38
 query(), 46, 123
 quote(), 124
 register(), 218
 registerAction(), 360, 372
 remove(), 221, 293

removeRoute(), 83
 render(), 205, 240
 renderujące, 204
 request(), 233
 run(), 171
 sanitize(), 93
 save(), 131, 156
 przesłonięcie, 218
 script(), 98, 103, 106
 searchAction(), 366
 secure(), 267
 serve(), 286
 set(), 52, 69, 76, 221, 238
 setColor(), 48
 setModel(), 48
 setOption(), 235
 setRequestHeaders(), 235
 setRequestOptions(), 235
 settingsAction(), 366
 setUp(), 380
 setValue(), 238
 sizeof(), 58
 sniff(), 284
 split(), 34
 strpos(), 93
 sync(), 152
 przykład użycia, 154
 tag(), 101
 tree(), 102
 trim(), 35
 unique(), 93
 upload(), 275, 342
 CakePHP, 395
 ustawiające, 46
 przezroczyste, 55
 żądań, 234
 validate(), 250
 validateRequired(), 249
 walidacji, 249
 where(), 127
 yield(), 239
 zarządzania trasami, 82
 model, 21, 143
 CakePHP, 385
 CodeIgniter, 323
 metody, 325
 struktura tabeli, 325
 User, 323

model

- File, 274
 - CodeIgniter, 337
 - Zend Framework, 371
- Friend, 263
- implementacja, 144
- kod SQL, 146
 - metody pobierające kolumny, 151
 - metody pomocnicze, 147
- Message, 268
 - metody modelu, 271
- modyfikowanie rekordów, 155
- Photo, 394
- rodzaje danych, 144
- testowanie, 183
- User, 144, 226
 - administrator, 301
 - CakePHP, 394
 - CodeIgniter, 323
 - konto użytkownika, 213
 - własności klasy, 146
 - Zend Framework, 355, 369
- użytkownika, 212
- walidacja, 247
- zastosowanie, 144
- Zend Framework, 353

Model-View-Controller, *Patrz* MVC

MVC, 21

- kontroler, 21
- model, 21
- szkielety, 23
 - CakePHP, 24, 381
 - CodeIgniter, 23, 319
 - Zend Framework, 24, 349
- widok, 21
- zalety wzorca, 22
- zasada działania, 22

MySQL, 117

- reprezentacja modelu User, 146
- SQL CREATE TABLE, 154
- testowanie, 177

O

- obserwator, 26, 290
 - nasłuchiwanie nowych stanów, 292
 - zasada działania, 26
 - zdarzenia, 293
- odzworowanie, *Patrz* ORM
- ORM, 143

P

parser

- plików INI, 64
- pliku konfiguracyjnego, 35
- szablonów, 91
 - funkcje obsługi instrukcji, 96
 - główne publiczne metody, 104
- implementacja, 92, 114
- metody parsujące, 99
- plan gramatyki, 95
- przykład działania, 105
- testowanie, 186

PEAR, 377

- instalacja, 377

PHP

- automatyczne ładowanie klas, 29
 - leniwe ładowanie, 31
- deserializacja, 223
- dołączanie zewnętrznych skryptów, 29
- metody
 - dostępowe, 47
 - łańcuchowe, 33
 - magiczne, 48
 - pobierające, 45
 - ustawiające, 45
- MyAdmin, 141
- MySQL, 117
- PEAR, 377
- przestrzenie nazw, 30
- serializacja, 223
- synchroniczność, 290
- Unit, 378
- wyjątki, 32
 - przechwytywanie, 33
 - try-catch, 32

pliki INI, 58

- parser, 64

PostgreSQL, 119

przepisywanie adresów URL, 195

- Apache2, 196
- CodeIgniter, 320

przestrzeń nazw, 30

- Zend Framework, 375

R

- rejestr, 25, 73, 75
 - zasady działania, 25

rewriting, *Patrz* przepisywanie adresów URL
 routing, *Patrz* trasowanie
 rozruch aplikacji, 195
 index.php, 197
 konfiguracja, 198
 baza danych, 199
 bufor, 200
 kontroler, 201
 plik rozruchowy, 197
 przepisywanie adresów URL, 196

S

serwer sieciowy, 419
 sieci społecznościowe, 191
 bazy danych, 192
 biblioteki, 211
 fani, 263
 konta użytkowników, 212
 kontakty, 263
 logowanie, 221
 formularz logowania, 221
 strona profilu, 222
 rejestracja, 215
 formularz rejestracyjny, 215
 relacje, 259
 sesje, 218
 strony błędów, 259
 struktura, 191
 danych, 192
 folderów, 192
 udostępnianie treści, 268
 przechowywanie wiadomości, 268
 publikowanie własnych wiadomości, 270
 wyświetlanie strumienia wiadomości, 269
 ustawienia, 247
 widok, 253
 weryfikacja danych, 247
 mapa walidacji, 249
 metody walidacji, 249
 wysyłanie plików, 274
 zdjęcia użytkowników, 274
 wyszukiwanie, 239
 szablon, 243
 widok, 241
 zdjęcia
 wysyłanie, 274
 wyświetlanie w profilu, 278
 znajomości, 273
 znajomości, 263
 model, 263
 sprawdzanie znajomych, 267
 trasowanie, 265
 zawieranie, 263
 zdjęcia, 273
 zrywanie, 263
 singleton, 25, 73
 klasa, 25
 zasada działania, 25
 SQL Server, 119
 synchroniczność, 290
 szablony, 91
 dialekt szablonowy, 91
 korzyści, 107
 gramatyka szablonowa, 229
 implementacja, 92
 metody, 92
 rozszerzanie, 230
 mapa gramatyki, 96
 parser szablonów, 91
 pobieranie podszałonów, 232
 metody pomocnicze, 237
 testowanie, 186
 tokeny, 92
 wyszukiwanie, 243
 znaczniki, 96
 druku, 96
 instrukcji, 96
 skryptów, 96

T

tablice asocjacyjne, 57
 testowanie, 169, 315
 bazy danych, 175
 bufor, 171
 CakePHP, 399
 odpowiedzi na żądania formularzy, 400
 testy jednostkowe, 399
 CodeIgniter, 347
 testy jednostkowe, 347
 zewnętrzne biblioteki, 348
 formularze wejściowe, 315
 pola formularzy, 316
 wyniki interakcji, 317
 wysyłanie formularzy, 317
 jednostkowe, 169

testowanie

klasy

- Cache, 172
- Configuration, 174
- Database, 176
- Model, 183
- Query, 181
- Request, 315
- konektor, 178
- konfiguracja, 174
- Memcached, 172
- modele, 183
- MySQL, 177
- pola formularzy, 316
- serwer sieciowy, 419
- szablony, 186
- Zend Framework, 377
 - dodawanie testów, 378
 - uruchamianie testów, 378
- znaczniki szablonowe, 186
- zpytania, 178

trasowanie, 22, 79

- CakePHP, 382
- CodeIgniter, 321
- definiowanie tras, 79
- klasy tras, 80
- metody zarządzania trasami, 82
- przetwarzanie istniejących tras, 84
- przetwarzanie zdefiniowanych tras, 83
- trasa czcionek, 286
- Zend Framework, 350
- znajomości, 265

tworzenie szkieletu, 29

- automatyczne ładowanie klas, 29
- bazy danych, 117
- buforowanie, 65
- klasa bazowa, 45
- kod szkieletu, 29
- konfiguracja, 57
- modele, 143
- rozruch aplikacji, 195
- rozszerzenia, 281
- struktura, 191
- szablony, 91
- testowanie, 169
- trasowanie, 79
- zdarzenia, 293
 - lista dodanych zdarzeń, 295

W

WAMP, 403

- Apache, 406
- dodaj użytkownika, 410
- lista użytkowników, 409
- Memcached, 406
- okno instalatora, 404
- PHP, 406
- Skype, 405
- tworzenie bazy danych, 409
- uprawnienia, 410
- wybór domyślnej przeglądarki, 404

wąskie gardło, 290

wget, 417

- instalacja, 417

widok, 21, 201

- CakePHP, 390
- metody renderujące, 204
- profil, 330
- rejestracja, 254, 339
 - CakePHP, 394
- renderowanie, 203
- użytkownicy, 309
 - edycja, 311
 - lista, 310
- ustawienia, 253
 - CakePHP, 394
- wyszukiwania, 241

wtyczki, 296

- CakePHP, 396
- loader, 296
- Logger, 297

wysyłanie plików

- CakePHP, 393
- CodeInteger, 337
- sieci społecznościowe, 274
- Zend Framework, 369

wzorce projektowe, 24

- fabryka, 26
- MVC, 21
- obserwator, 26, 290
- rejestr, 25, 73, 75
- singleton, 25, 73

Z

Zend Framework, 24, 349
biblioteki zewnętrzne, 374
dokumentacja, 349
konfiguracja, 350
kontroler, 359
 Users, 359
logowanie, 362
mechanizm automatycznego ładowania, 374
metody inicjacyjne, 356
modele, 353
modyfikowanie wierszy, 354
przestrzeń nazw, 375
testowanie, 377
 dodawanie testów, 378
 uruchamianie testów, 378
trasowanie, 350
wybieranie wierszy, 354
wysyłanie plików, 369

znaczniki

 druku, 96
 instrukcji, 96
 skryptów, 96
 szablonowe, 186

Ż

żądania adresów URL, 232
 metody ustawiające żądań, 234

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

Wzorzec MVC w PHP dla profesjonalistów

W dobie aplikacji internetowych wzorzec MVC jest jednym z najpowszechniej używanych. Pozwala on utrzymać aplikację w ryzach i sprawia, że jej konserwacja nie nastrocza zbyt wielu problemów. U podstaw MVC leży podział aplikacji na trzy warstwy — modelu, widoku i kontrolera (ang. Model View Controller). Wzorzec ten jest obecnie wykorzystywany praktycznie w każdym języku programowania.

Dzięki tej książce zrozumiesz, jak z niego korzystać w aplikacji pisanej przy użyciu języka PHP. W trakcie lektury poznasz jego zalety oraz zobaczysz, jak zacząć tworzenie aplikacji opartej na MVC. W kolejnych rozdziałach zagłębisz się w szczegóły konfigurowania, trasowania, tworzenia modeli oraz wykorzystywania baz danych. Ponadto zdobędziesz dogłębną wiedzę na temat testowania aplikacji — to klucz do tworzenia niezawodnych systemów. W tej książce znajdziesz również opis popularnych szkieletów wspierających MVC: Zend Framework i CakePHP to tylko niektóre z nich. Książka ta jest obowiązkową lekturą dla każdego programisty aplikacji internetowych korzystającego z języka PHP.

Dzięki tej książce:

- poznasz zalety wzorca MVC
- skutecznie przetestujesz Twoją aplikację
- zaznajomisz się z dostępnymi szkieletami aplikacji wspierającymi MVC

Apress®

Nr katalogowy: 14214

Księgarnia internetowa:
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900

helion.pl
księgarnia
internetowa

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>



Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

cena: 79,00 zł

ISBN 978-83-246-7015-4



9 788324 670154

Informatyka w najlepszym wydaniu