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

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.

Spis treści

ROZDZIAŁ 10

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 szczelną 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 `@readwrite`, którą znamy, ponieważ używaliśmy już jej wcześniej. Z jej pomocą będziemy mogli wywoływać metody dostępowe 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** Haseł 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 wykradł 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 dostępowe i metody przesłaniające generowanie wyjątków. Własność `$_types` zawiera proste typy danych rozpoznawane przez nasze modele i jest używana zarówno wewnętrznie, 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$" => "\\1",
            "([m|l])ice$" => "\\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$" => "\\1ice",
        "(matr|vert|ind)ix|ex$" => "\\1ices",
        "(x|ch|ss|sh)$" => "\\1es",
        "([^aeiouy]|qu)y$" => "\\1ies",
        "(hive)$" => "\\1s",
        "(?:([^f])fe|([lr])f)$" => "\\1\\2ves",
        "sis$" => "ses",
        "([ti])um$" => "\\1a",
        "(p)erson$" => "\\1eople",
        "(m)an$" => "\\1en",
        "(c)hild$" => "\\1hildren",
        "(buffal|tomat)o$" => "\\1\\2oes",
        "(bu|campu)s$" => "\\1\\2ses",
        "(alias|status|virus)" => "\\1es",
        "(octop)us$" => "\\1i",
        "(ax|cris|test)is$" => "\\1es",
        "s$" => "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 ładował rekord, kiedy zostanie podana wartość kolumny głównej. Zrobimy to, modyfikując konstruktor i dodając metodę `load()`, co pokazuję 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

D

F

H

I

K

L

M

O

P

R

S

T

W

Z

Ż

TWÓRZ ELASTYCZNE I NIEZAWODNE APLIKACJE INTERNETOWE!

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 nastręcza 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/nowosci

Helion

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

cena: 79,00 zł

ISBN 978-83-246-7015-4

9 788324 670154

Informatyka w najlepszym wydaniu