helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

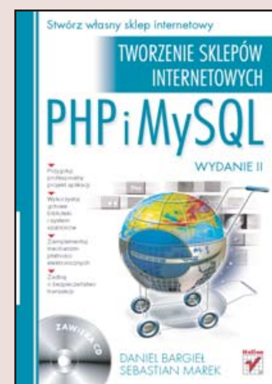
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

PHP i MySQL. Tworzenie sklepów internetowych. Wydanie II



Autorzy: Daniel Bargieł, Sebastian Marek
ISBN: 83-7361-939-9
Format: B5, stron: 512

Liczba firm oferujących towary w internecie stale wzrasta. Taki sposób przedstawienia oferty umożliwia dotarcie do szerszego grona klientów i pozwala na znaczną redukcję kosztów prowadzenia działalności handlowej. Rozwój handlu elektronicznego spowodował zwiększenie zainteresowania usługami związanymi z tworzeniem sklepów internetowych. Programiści, którzy otrzymują takie zlecenie, zazwyczaj korzystają z dwóch bezpłatnych aplikacji: PHP i MySQL. Ciągle rozwijany i rozbudowywany PHP jest jednym z najpopularniejszych języków skryptowych interpretowanych po stronie serwera. Jego najnowsza wersja, oznaczona numerem 5, to w pełni obiektowe środowisko stosowane przez twórców dynamicznych aplikacji WWW odwołujących się do baz danych. Funkcję zaplecza bazodanowego spełnia baza MySQL — prosta i wydajna.

Zbudowanie funkcjonalnego i bezpiecznego sklepu internetowego to ciekawe wyzwanie dla programisty. Jeśli chcesz się z nim zmierzyć, książka „PHP i MySQL. Tworzenie sklepów internetowych. Wydanie II" jest dla Ciebie idealną lekturą. Znajdziesz w niej wszystkie informacje, jakich potrzebujesz, by zaprojektować i stworzyć sklep internetowy, korzystając z języka PHP 5, bazy danych MySQL i dodatkowych mechanizmów opisanych w kolejnych rozdziałach książki.

- Opracowanie koncepcji sklepu internetowego
- Nowe możliwości PHP 5
- Oddzielenie kodu PHP od HTML z zastosowaniem szablonów Smarty
- Wykorzystanie funkcji z biblioteki PEAR
- Mechanizmy obsługi sesji i plików cookie
- Zabezpieczanie aplikacji
- Przygotowanie projektu sklepu
- Katalog produktów i koszyk
- Moduł zarządzania klientami i zamówieniami
- Obsługa płatności elektronicznych

Przyczyń się do rozwoju e-biznesu — stwórz własny sklep internetowy

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Podziękowania ... 7
Wstęp ... 9

Rozdział 1. Koncepcja sklepu internetowego ... 13
Część publiczna ... 13
Część administracyjna ... 14

Rozdział 2. Migracja z PHP4 do PHP5 ... 17
Nowy model obiektowy ... 17
Konstruktory oraz destruktory klas ... 18
Dostęp do metod oraz właściwości obiektów ... 21
Klasy i metody abstrakcyjne ... 25
Dziedziczenie po interfejsach ... 27
Obsługa XML poprzez rozszerzenie DOM ... 28
Ładowanie i zapisywanie obiektów XML ... 28
Klasy obiektów rozszerzenia DOM ... 32
Wyrażenia XPath ... 37
Wyjątki ... 41
Klasa Exception ... 42
Zgłaszanie i przechwytywanie wyjątków ... 43
Klasy potomne klasy Exception ... 47
Rozwiązania przyjęte w aplikacji sklepu internetowego ... 49
Model obiektowy ... 49
Korzystanie z dokumentów XML ... 54
Wyjątki w aplikacji ... 54

Rozdział 3. Oddzielenie kodu PHP od kodu HTML ... 57
System szablonów Smarty ... 58
Instalacja systemu szablonów Smarty ... 60
Struktura katalogowa ... 63
Parametry konfiguracyjne ... 64
Praca ze Smarty ... 70
Metody obiektów klasy Smarty ... 75
Zasady tworzenia szablonów TPL ... 78
Transformacje XSL ... 93
Czym jest XSL ... 95
Rozwiązania przyjęte w aplikacji sklepu internetowego ... 104

Rozdział 4. PEAR ... 109
Czym jest, a czym nie jest PEAR? ... 109
Biblioteka skryptów PHP ... 110
Zarządzanie pakietami i rozpowszechnianie ich ... 110
PHP Foundation Classes ... 110
Pakiet DB — komunikacja z bazą danych ... 111
Konfiguracja połączenia ... 113
Łączenie i rozłączanie się z bazą danych ... 114
Wysyłanie zapytań do bazy danych ... 115
Pobieranie zwróconych danych ... 117
Szybkie pobieranie danych ... 121
Pobieranie dodatkowych informacji ... 127
Pakiet Log — obsługa dziennika zdarzeń ... 129
Tworzenie obiektów dziennika zdarzeń ... 130
Pakiet Mail — wysyłanie wiadomości e-mail ... 132
Pakiet QuickForm — obsługa formularzy ... 136
Pierwszy formularz ... 136
Formularz z życia wzięty ... 138
Filtry i reguły walidacji ... 140
Wartości domyślne i stałe ... 142
Przetwarzanie danych ... 143
Elementy formularza ... 145
Zmiana wyglądu formularza ... 161
Integracja z systemem szablonów Smarty ... 166
Pakiet QuickForm_Controller — formularze zaawansowane ... 176
Typowy formularz ... 177
Formularz w formie wizarda ... 184
Formularz z zakładkami ... 192
Pakiet Benchmark — pomiar wydajności skryptów ... 198
Benchmark_Timer ... 198
Benchmark_Profiler ... 201
Benchmark_Iterate ... 204

Rozdział 5. Mechanizmy autoryzacji i sesje ... 207
Sposoby autoryzacji użytkownika w aplikacji ... 207
Identyfikacja użytkownika ... 207
Zabezpieczenie sesji ... 220
Standardowy mechanizm obsługi sesji ... 220
Własny mechanizm obsługi sesji ... 220
Wykorzystanie cookies ... 226
Do czego można wykorzystać cookies ... 227
Bezpieczeństwo cookies ... 228
Sposób użycia ... 228
Wykorzystanie nagłówków HTTP ... 230

Rozdział 6. Bezpieczeństwo ... 237
Bezpieczeństwo systemu operacyjnego oraz serwera WWW ... 237
Cel instalacji serwera ... 238
Tylko potrzebne usługi ... 238
Bezpieczna konfiguracja serwera WWW ... 239
Bezpieczeństwo wykorzystywanego oprogramowania ... 239
Instalacja PHP jako pliku wykonywalnego CGI ... 239
Instalacja PHP jako modułu Apache'a ... 242
Opcja register_globals ... 242

Raportowanie o błędach 244
Ukrywanie PHP 245
Aktualizacje 245
Bezpieczeństwo własnej aplikacji 246
Brak walidacji danych 246
Nieskuteczne mechanizmy kontroli dostępu i autoryzacji 247
Nieprawidłowe zarządzanie kontami oraz sesjami użytkowników 249
Ataki typu Cross-Site Scripting (XSS) 250
Wstrzykiwanie kodu 251
Przechowywanie niezabezpieczonych danych 253
Bezpieczeństwo bazy danych 253
Zarządzanie hasłami 254

Rozdział 7. Projekt aplikacji 255
Wykorzystywane narzędzia 256
DBDesigner 4 — projektowanie bazy danych 257
Aqua Data Studio 270
Eclipse 272
Założenia projektowe 281
Obsługa słowników 281
Wersje językowe 286
Interfejs użytkownika 299
Nagłówek strony 300
Menu główne sklepu 300
Część centralna sklepu 301
Stopka strony 302
Struktura bazy danych 303
Użytkownicy i klienci sklepu 304
Produkty 306
Kategorie 308
Producenci 309
Zamówienia 310
Słowniki 313
Biblioteka zdjęć 314
Wersje językowe 316
Budowa modułowa i struktura aplikacji 317
Struktura katalogowa 318
Konfiguracja serwisu 320
Przetwarzanie żądań 320
FCKeditor 322

Rozdział 8. Katalog produktów 327
Asortyment i produkty 327
Asortyment sklepu — klasa Item 327
Produkty dostępne w ofercie sklepu — klasa Produkt 346
Produkty w promocji 364
Obsługa promocji — klasa Special 365
Zarządzanie promocjami 367
Kategorie produktów 367
Struktura katalogowa — klasa Catalog 368
Wyświetlanie struktury katalogowej 369

Rozdział 9. Koszyk 371
Sesja jako podstawowy mechanizm realizacji koncepcji koszyka 372
Koszyk — klasa Basket 373
Operacje na produktach w koszyku 374
Operacje na sumarycznych wartościach cen produktów w koszyku 375
Mechanizm obsługi koszyka 375
Kontroler BasketPage 376
Kontroler ChangeCountPage 378
Kontroler ClearBasketPage 379
Akcje kontrolera — obsługa zdarzeń 380

Rozdział 10. Rejestracja klientów i zarządzanie nimi 383
Koncepcja użytkowników aplikacji 384
Klasy użytkownika — User oraz CustomUser 384
Złożone operacje na obiekcie użytkownika 386
Mechanizm rejestracji nowego użytkownika 387
Pierwszy etap rejestracji — formularze rejestracyjne 387
Drugi etap rejestracji — aktywacja konta użytkownika 406

Rozdział 11. Obsługa zamówień 409
Realizacja koncepcji zamówień 409
Warunki złożenia zamówienia 409
Moduł zamówienia — klasa Order 411
Zarządzanie zamówieniami 412
Realizacja płatności przez bank Inteligo 413
Wykonywanie płatności z bankiem Inteligo 414
Uruchomienie płatności „Płacę z Inteligo" 415
Realizacja płatności przez bank mBank 415
Uruchomienie płatności typu mTransfer 415
Wykonywanie przelewów przez mTransfer 419
Płatności elektroniczne — karty płatnicze 422
Usługa Bezpieczne z@kupy firmy Polcard 423
Usługa ePłatności firmy eCard 427

Dodatek A Przygotowanie środowiska pracy 439

Dodatek B Instalacja i konfiguracja sklepu internetowego 479

Skorowidz 489

Rozdział 5.
Mechanizmy autoryzacji i sesje

Sposoby autoryzacji użytkownika w aplikacji

Identyfikacja użytkownika

W przypadku bardziej złożonych aplikacji, a za taką z pewnością można uznać sklep internetowy, kwestia kontroli poczynań użytkowników staje się bardzo ważna. Pewne elementy aplikacji muszą zostać zabezpieczone przed dostępem osób nieupoważnionych, a inne będziemy chcieli udostępnić tylko wybranym użytkownikom. Musimy zatem w jakiś sposób sprawdzić, kto korzysta z naszej aplikacji i co mu tak naprawdę wolno w niej zrobić. Jedynym sposobem na to jest implementacja skutecznego, bezpiecznego i szybkiego mechanizmu identyfikacji użytkownika. Nie bez znaczenia są również koszty wdrożenia takiego systemu.

Spróbujemy przedstawić różne sposoby implementacji mechanizmów autoryzacji oraz ich podstawowe wady i zalety.

Prosta kontrola dostępu

Zabezpieczenie dostępu do pojedynczych stron nie stanowi większego problemu. W tym celu wystarczy stworzyć prosty formularz oraz skrypt PHP, który sprawdzi dane wpisane do niego. Przykładowy skrypt może wyglądać tak jak na listingu 5.1.

Listing 5.1. *Skrypt zabezpieczający dostęp do strony WWW przy użyciu mechanizmu prostej kontroli dostępu*

```
<?php
if (!isSet($_POST['username']) && !isSet($_POST['password'])) {
//Wyświetl formularz autoryzacyjny
?>
  <h1>Proszę się zalogować!</h1>
  <form method="post">
    <table border="1">
      <tr><td>Użytkownik:</td><td><input type="text" name="username"></td></tr>
      <tr><td>Hasło:</td><td><input type="password" name="password"></td></tr>
      <tr><td colspan="2" align="center"><input type="submit" value="Zaloguj się">
      </td></tr>
    </table>
  </form>
<?php
}elseif ($_POST['username']=="sebastian" && $_POST['password']=="tajnehaslo") {
  //Autoryzacja przebiegła pomyślnie
?>
  <h1>Witamy Cię serdecznie!</h1>
  Autoryzacja przebiegła pomyślnie i uzyskałeś dostęp do strony chronionej.
<?php
} else {
  //Błędna nazwa użytkownika lub hasło
?>
  <h1>Przepraszamy!</h1>
  Nie masz dostępu do tej strony.
<?php
}
?>
```

Zasada działania takiego skryptu jest następująca:

- jeżeli użytkownik nie podał nazwy użytkownika oraz hasła, na stronie wyświetlony zostanie formularz autoryzacyjny,
- jeżeli użytkownik podał złą kombinację danych użytkownik-hasło, wyświetlona zostanie informacja o wystąpieniu błędu w trakcie autoryzacji,
- jeżeli podana nazwa użytkownika oraz hasło będą prawidłowe, wyświetlona zostanie zabezpieczona treść strony.

Zalety:

- Prostota i szybkość implementacji.

Wady:

- Zabezpieczona zostaje w ten sposób tylko jedna strona. Jeżeli chcielibyśmy zabezpieczyć większą liczbę stron przed ciekawskimi oczami internautów, taki mechanizm trzeba by umieścić na każdej z nich.

- Nazwa użytkownika i jego hasło są „na stałe” umieszczone w kodzie skryptu. Stwarza to kilka problemów oraz niebezpieczeństw. W przypadku gdy użytkownik chciałby zmienić hasło, musiałby się zwrócić do autora skryptu lub osoby mającej dostęp do jego źródeł. Pomijając kwestię sposobu przekazywania nowego hasła między tymi dwoma osobami, sytuacja jest bardzo niezręczna i kłopotliwa. Napisanie osobnego skryptu (albo „samomodyfikowalnego” skryptu) będzie już przerostem formy nad treścią. Największym problemem jest ograniczenie liczby użytkowników, którzy mają dostęp do strony. Można oczywiście dopisać w skrypcie następnych użytkowników oraz ich hasła, jednak dodatkowo skomplikuje to sposób autoryzacji.
- Hasło jest zapisane oraz przesyłane do serwera w oryginalnej postaci. Oznacza to, że w bardzo prosty sposób można je przechwycić i wykorzystać w celu nieautoryzowanego dostępu do strony.

Zaawansowana kontrola dostępu

Aby uprościć mechanizmy zarządzania użytkownikami mającymi dostęp do zabezpieczonych stron, należy pomyśleć o przeniesieniu informacji na ich temat do jakiegoś zewnętrznego źródła. Modyfikacja danych w skrypcie, w którym jednocześnie umieszczone są mechanizmy autoryzacyjne, nie jest dobrym pomysłem. Prościej i skuteczniej zrealizujemy taki mechanizm, gdy nazwy użytkowników oraz ich hasła będziemy przechowywać w osobnym pliku (listing 5.2), a do tego stworzymy osobny skrypt oraz stronę WWW, za pomocą których będziemy mieli możliwość dodawania i usuwania użytkowników oraz zmiany ich haseł. Informacje takie możemy przechowywać w pliku tekstowym lub bazie danych.

Listing 5.2. *Przykładowa struktura pliku tekstowego z nazwami użytkowników oraz ich hasłami*

```
#Nazwa użytkownika oraz hasło oddzielone są od siebie znakiem dwukropka
sebastian:hasło1
daniel:hasło2
magda:hasło3
beata:hasło4
```

Mechanizm przeszukiwania oraz edycji pliku tekstowego jest dosyć złożony. Należy pamiętać, aby zabezpieczyć się przed jednoczesnym dostępem wielu użytkowników do tego pliku, w przeciwnym razie mogą oni zniszczyć oryginalną strukturę pliku.

Przechowywanie nazw użytkowników oraz ich haseł w pliku tekstowym sprawdza się w przypadku, gdy nie przekroczy on 100 wierszy (czyli będzie przechowywał informację o maksymalnie 100 użytkownikach). W przypadku większej liczby użytkowników szybkość działania skryptu znacząco spadnie. W takiej sytuacji należy zdecydować się na przechowywanie tych informacji w bazie danych.

Sam skrypt autoryzacyjny nie będzie o wiele bardziej skomplikowany niż ten, w którym hasła są przechowywane w pliku tekstowym, natomiast w zamian uzyskamy dużo szybszy mechanizm autoryzacji. W przypadku sklepu internetowego taki mechanizm

przechowywania danych jest najlepszy i najbezpieczniejszy, a jednocześnie zapewnia największą wydajność. Dodatkowo otrzymujemy większą kontrolę nad tym, kto, kiedy i w jakich warunkach dokonał autoryzacji oraz do jakich informacji uzyskał dostęp.

Przykład 5.1

Tworząc prostą aplikację autoryzującą:

1. W pierwszej kolejności musimy utworzyć odpowiednią strukturę bazy danych. Odpowiedni skrypt SQL znajdziemy na listingu 5.3.

Listing 5.3. *Skrypt tworzący strukturę tabeli Users na serwerze baz danych MySQL*

```
CREATE DATABASE baza_danych;
USE baza_danych;
CREATE TABLE Users
(
   USR_Id                      varchar(32)                  not null,
   UINFO_Email                 varchar(96),
   USR_Password                varchar(32)                  not null,
   primary key (USR_Id)
) type = InnoDB;

GRANT SELECT, INSERT, UPDATE, DELETE ON user.* TO 'user'@'localhost' IDENTIFIED BY
'tajnehaslo';
```

Struktura utworzonej tabeli została pokazana na rysunku 5.1.

Rysunek 5.1. *Struktura tabeli z danymi użytkowników*

users
USR_Id: VARCHAR(32)
UINFO_Email: VARCHAR(96)
USR_Password: VARCHAR(32)

W tabeli będziemy przechowywać następujące dane:

- ♦ `USR_Id` — unikalna nazwa użytkownika (ciąg 32 znaków, nie może być pusty),
- ♦ `UINFO_Email` — adres e-mail użytkownika (ciąg 96 znaków, nie może być pusty),
- ♦ `USR_Password` — hasło użytkownika (ciąg 32 znaków, nie może być pusty).

2. Aplikację przygotujemy na podstawie struktury aplikacji, którą poznaliśmy w rozdziale 4., rozbudowanej o obsługę bazy danych. Struktura ta znajduje się na CD-ROM-ie dołączonym do książki w katalogu *materialy/rozdzial05/struktura_aplikacji*. Skopiujmy ją na serwer WWW, tak aby katalog *www* był katalogiem *rootwww* dla serwera.

   Nowością jest tutaj plik *config/dbmanager.inc.php*, w którym znajduje się definicja klasy `DBManager` obsługującej połączenia z bazą danych MySQL. Obsługa odbywa się przy wykorzystaniu pakietu PEAR::DB (patrz rozdział 4. „PEAR").

Parametry połączenia z bazą danych stanowią stałe `DB_HOST` (host bazy danych), `DB_NAME` (nazwa bazy danych), `DB_USERNAME` (nazwa użytkownika bazy danych), `DB_PASSWD` (hasło dostępu do bazy danych). Wszystkie te stałe zdefiniowane są w pliku *config/define.inc.php*.

Inicjalizację połączenia z bazą danych przeprowadzimy w głównym pliku aplikacji *www/index.php*. Wykorzystamy do tego metodę `init()`, którą wywołamy statycznie:

```
DBManager::init()
```

3. Przeprowadzenie całego procesu autoryzacji wymagać będzie dwóch funkcji: jednej obsługującej transakcje zapewniające zachowanie integralności wprowadzanych informacji do bazy danych (listing 5.4) oraz drugiej przeprowadzającej sam proces autoryzacji. W tym celu stworzymy klasę użytkownika, dzięki której cały proces stanie się przejrzysty i łatwy w zarządzaniu. Pierwszą z tych funkcji uczynimy metodą klasy `DBManager`, natomiast drugą częścią klasy `User`.

Listing 5.4. *Klasa DBManager wraz z metodą obsługującą transakcje*

```
<?php
class DBManager extends PEAR {
   private static $dataBaseConnection = null;
   private function __construct() {}

    public function init() {
     if (DB::isError(self::$dataBaseConnection = DB::connect(self::DSNInit())))
     throw new DBError(self::$dataBaseConnection->getMessage(),
     self::$dataBaseConnection->getCode());
   }
   public function Query($queryString) {
       if (self::$dataBaseConnection) {
           if (DB::isError($result = self::$dataBaseConnection->Query($queryString)))
               throw new DBError($result->getMessage(), $result->getCode());
           return $result;
       }
   }
   public function DSNInit() {
       //Dołączenie obsługi klasy PEAR-DB
       require_once('DB.php');

           return 'mysql://' . DB_USERNAME . ':' . DB_PASSWD . '@tcp(' . DB_HOST . ')
           /' . DB_NAME;
   }
   /**
    * Obsługa transakcji bazodanowych
    *
    * @access public
    * @param string $type Rodzaj transakcji:
    *                      - BEGIN    - rozpoczęcie transakcji
    *                      - COMMIT   - zatwierdzenie transakcji
    *                      - ROLLBACK - zwinięcie transakcji
    * @return mixed TRUE              - jeśli funkcja wykonała się prawidłowo
    *              zgłoszenie wyjątku - jeśli w trakcie wykonywania funkcji
    *                                   wystąpił błąd
    */
```

```
    public function Transaction($type) {
        //Dołączenie obsługi klasy PEAR-DB
        require_once('DB.php');

       $sqlquery = $type;
        if (DB::isError($result = self::$dataBaseConnection->Query($sqlquery))) {
            throw new DBError('Przesłanie zapytania SQL nie powiodło się!');
        } else {
            if (self::$dataBaseConnection->affectedRows() < 0) {
                throw new DBError('Odebranie wyników wykonania zapytania SQL nie
                powiodło się!');
            }
        }
        return TRUE;
    }
}
?>
```

Do obsługi transakcji MySQL wymaga stworzenia tabel typu *InnoDB*. Proces instalacji oraz konfiguracji MySQL-a pod kątem obsługi tego typu tabel został omówiony w dodatku A w podrozdziale „Środowisko pracy — Windows i Linux".

Argumentem metody `Transaction()` jest tak naprawdę jedno z trzech poleceń systemu transakcji: `BEGIN`, `COMMIT` oraz `ROLLBACK`, które kolejno rozpoczynają transakcję, potwierdzają ją oraz cofają stan bazy danych do stanu sprzed rozpoczęcia transakcji. W przypadku wystąpienia błędu w trakcie wysyłania zapytania SQL do bazy danych funkcja zgłosi wyjątek `DBError` z komunikatem `Przesłanie zapytania SQL nie powiodło się!`, w przypadku błędu wykonania zapytania SQL funkcja zgłosi wyjątek `DBError` z komunikatem `Odebranie wyników wykonania zapytania SQL nie powiodło się!`, a jeżeli cały proces przebiegnie prawidłowo, funkcja zwróci wartość `TRUE`.

O wyjątkach i sposobie obsługi błędów pisaliśmy w podrozdziale „Wyjątki" rozdziału 2. „Migracja z PHP 4 do PHP 5".

Dobrze jest zadeklarować swoją klasę wyjątku i przechwytywać tylko własne wyjątki, a na przechwycenie wyjątków globalnych zezwolić jądru aplikacji. Dlatego też założyliśmy, iż wyjątki globalne będą nosić nazwę `Error` i dziedziczyć będą wszystkie właściwości i metody po klasie `Exception`, wyjątki generowane przez klasę `DBManager` będą obiektami klasy `DBError` (dziedziczącej po klasie `Error`), a wyjątki generowane w trakcie procesu autoryzacji użytkownika (opisanego poniżej) będą obiektami klasy `AuthError` (również dziedziczącej po klasie `Error`).

4. Przejdźmy teraz do stworzenia klasy `User`. Jej definicja znajduje się w pliku *config/user.inc.php*. Będzie ona mieć podobne właściwości jak pola w tabeli `Users` oraz metody pozwalające na szybkie ustawianie oraz zwracanie ich wartości (listing 5.5).

Listing 5.5. *Klasa User*

```
<?php
class User {
  private $_Id;
  private $_DSN;
  private $_Password;
  private $_Email;

//Konstruktor klasy
public function __construct($userdata = NULL){}
//Metody ustawiające wartości właściwości klasy
public function setId($id){}
public function setDSN($dsn){}
public function setPassword($password){}
public function setEmail($email){}

//Metody zwracające wartości właściwości klasy
public function getId(){}
public function getDSN(){}
public function getPassword(){}
public function getEmail(){}

//Metoda autoryzująca użytkownika
public function autorize ($userLogin, $userPassword) {}
//Metoda logująca użytkownika
public function Logon($login, $password) {}
}
?>
```

Parametrem konstruktora klasy jest tablica asocjacyjna, w której klucze tablicy przyjmują wartości pól z tabeli `Users` naszej bazy danych (listing 5.6). W trakcie tworzenia nowego obiektu tej klasy ustawimy na podstawie zawartości tej tablicy wartości właściwości klasy.

Listing 5.6. *Konstruktor klasy User*

```
public function __construct($userdata = NULL) {
  foreach ($userdata as $key => $value) {
    switch (strtolower($key)) {
      case 'usr_id':
        $this->setId($value);
        break;
      case 'uinfo_email':
        $this->setEmail($value);
        break;
    }
  }
  $this->setDSN(DSNInit());
}
```

Typowe metody ustawiające oraz zwracające wartość danej właściwości klasy będą wyglądać tak jak na listingu 5.7.

Listing 5.7. *Metoda ustawiająca oraz zwracająca wartość właściwość Id klasy*

```
//Metody ustawiające wartości właściwości klasy
public function setId($id) {
  if ($id <> NULL) {
    $this->_Id = $id;
    return TRUE;
  } else { return FALSE; }
}
//Metody zwracające wartości właściwości klasy
public function getId() { return $this->_Id; }
```

5. Sercem klasy jest tak naprawdę metoda `autorize()` (listing 5.8). Jej argumentami są kolejno nazwa użytkownika (`$userLogin`) oraz hasło (`$userPassword`). Cały proces autoryzacji przeprowadzamy w ramach jednej transakcji, zatem pierwszą czynnością, jaką należy wykonać, będzie jej rozpoczęcie:

```
DBManager::Transaction('BEGIN')
```

Listing 5.8. *Metoda autorize() — rozpoczęcie transakcji*

```
<?php
public function autorize ($userLogin, $userPassword)
{
    //Rozpoczęcie transakcji
    DBManager::Transaction('BEGIN');
    //Sprawdź, czy istnieje konto użytkownika
    $sqlquery = "SELECT * FROM Users WHERE Users.USR_Id = '$userLogin'";
    $result = DBManager::Query($sqlquery);
    //Pobranie wyników zapytania
    if ($user = &$result->fetchRow(DB_FETCHMODE_ASSOC)) {
      if ($user['USR_Password'] == md5($userPassword)) {
        DBManager::Transaction('COMMIT');
        return TRUE;
      }
    }
    //Odwołanie transakcji
    DBManager::Transaction('ROLLBACK');
    throw new AuthError('Logowanie nieudane - Użytkownik lub/i hasło niepoprawne!');
}
?>
```

Aby niepotrzebnie nie pobierać z bazy wszystkich danych, w kolejnym kroku sprawdzimy, czy użytkownik, który próbuje dokonać autoryzacji, w ogóle istnieje:

```
$sqlquery = "SELECT * FROM Users WHERE Users.USR_Id = '$userLogin'";
$result = DBManager::Query($sqlquery);
```

W przypadku wystąpienia jakiegokolwiek błędu w trakcie procesu autoryzacji należy pamiętać o „zwinięciu” transakcji. Bez względu na to, czy w trakcie transakcji wystąpił jakiś błąd, czy też nie, zawsze należy ją zakończyć — albo pomyślnie instrukcją `COMMIT`, która dopiero zapisze wszystkie zmiany w bazie danych, albo niepomyślnie instrukcją `ROLLBACK`, która przywróci bazę danych do stanu sprzed rozpoczęcia transakcji.

W przypadku gdy transakcja nie zostanie jawnie zakończona instrukcją `COMMIT` lub `ROLLBACK`, serwer bazy danych po czasie określonym w pliku konfiguracyjnym sam zakończy transakcję, przywracając bazę danych do stanu sprzed jej rozpoczęcia.

Pusty wynik przesłanego zapytania oznacza, że takiego użytkownika nie ma w bazie danych. Generujemy zatem wyjątek klasy `AuthError` z komunikatem `Logowanie nieudane - Użytkownik lub/i hasło niepoprawne!` i kończymy proces autoryzacyjny niepowodzeniem:

```
//Odwołanie transakcji
DBManager::Transaction('ROLLBACK');
throw new AuthError('Logowanie nieudane - Użytkownik lub/i hasło niepoprawne!');
```

W przypadku gdy użytkownik znajduje się w bazie danych, sprawdzamy zgodność wprowadzonego przez niego hasła. Jeżeli hasło jest zapisane w bazie danych w oryginalnej postaci (czego nie powinno się robić), wystarczy zwykłe porównanie ciągów, jednak w przypadku gdy hasło znajduje się w bazie danych w postaci zaszyfrowanej, musimy skorzystać z odpowiedniego algorytmu, który pozwoli zaszyfrować ciąg znaków podany przez logującego się użytkownika.

Zalecamy stosowanie funkcji szyfrujących przy przechowywaniu haseł użytkowników. Szczególnie skuteczne są funkcje `md5()` oraz `crypt()`, których odszyfrowanie jest praktycznie niemożliwe. Jeśli użytkownik zapomni hasła do swojego konta, jedynym rozwiązaniem jest wygenerowanie nowego. MySQL, zapisując hasło użytkownika do bazy danych, już domyślnie korzysta z algorytmu MD5.

Gdy nazwa użytkownika oraz jego hasło są poprawne, sytuacja jest prosta — kończymy proces autoryzacji sukcesem, zwracając wartość `TRUE`, i zakańczamy transakcję instrukcją `COMMIT`. Tę część metody `autorize()` przedstawia listing 5.9.

Listing 5.9. *Sposób obsługi pomyślnej autoryzacji użytkownika*

```
if ($user['USR_Password'] == md5($userPassword)) {
  DBManager::Transaction('COMMIT');
  return TRUE;
}
```

6. Pozostaje zatem wykorzystać w praktyce dopiero co stworzoną metodę. Zaimplementujemy ją w metodzie `Logon()` tej samej klasy (listing 5.10).

Listing 5.10. *Metoda Logon() klasy User*

```
//Metoda logująca użytkownika
  public function Logon($login, $password) {
    self::autorize($login, $password);
    //Pobranie danych zautoryzowanego użytkownika
    $sqlquery = "SELECT * FROM users WHERE usr_id='$login'";
    $result = DBManager::Query($sqlquery);
```

```
    if ($userdata = $result->fetchRow(DB_FETCHMODE_ASSOC)) {
      //Utworzenie obiektu User i zalogowanie użytkownika
      $currentUser = new User($userdata);
      return $currentUser;
    } else {
      throw new Error('Pobranie danych nie powiodło się!');
    }
  }
```

Jeżeli wykonanie metody przebiegnie bez problemów, na podstawie informacji pobranych z bazy danych tworzymy nowy obiekt użytkownika i zwracamy go. W przeciwnym razie generujemy wyjątek klasy `Error` (listing 5.10).

7. Ostatnim etapem naszego przykładu będzie stworzenie strony WWW umożliwiającej przeprowadzenie procesu autoryzacji użytkownika. Skrypt z listingu 5.11 znajduje się również w pliku *www/logon.php*, a szablony potrzebne do wyświetlenia zawartości strony w plikach *smartydirs/templates/logon.tpl* oraz *smartydirs/templates/logon_finished.tpl*.

Listing 5.11. *Skrypt zabezpieczający dostęp do strony WWW przy użyciu mechanizmu autoryzacji opartej na bazie danych MySQL*

```
<?php
require_once('PEAR.php');
require_once('DB.php');
require('../config/user.inc.php');

class AuthError extends Error {}

if (!isSet($_POST['username']) && !isSet($_POST['password'])) {
  //Wyświetl formularz autoryzacyjny
  $smarty->display('logon.tpl');
} elseif (isSet($_POST['username']) && isSet($_POST['password'])) {
  try {
    //Próba nawiązania połączenia z bazą danych
    if (PEAR::isError(DB::connect(DBManager::DSNInit()))) {
          throw new Error('dbconnectionerror');
    }
    User::Logon($_POST['username'], $_POST['password']);
    //Autoryzacja przebiegła pomyślnie
    $smarty->assign('title', 'Witamy Cię serdecznie!');
    $smarty->assign('message', 'Autoryzacja przebiegła pomyślnie i uzyskałeś dostęp
    do strony chronionej.');
  } catch (AuthError $err) {
    $smarty->assign('title', 'Przepraszamy!');
    $smarty->assign('message', 'Dostęp do tej strony został zablokowany.<br/>' .
    $err->getMessage());
  }
  $smarty->display('logon_finished.tpl');
}
?>
```

Zalety:

- nazwa użytkownika i hasło przechowywane są poza skryptem (albo w osobnym pliku tekstowym, albo w bazie danych),
- możliwość zdefiniowania nieograniczonej ilości użytkowników,
- łatwy sposób implementacji zarządzania użytkownikami,
- możliwość zapisu zaszyfrowanego hasła,
- możliwość gromadzenia większej ilość informacji o użytkownikach.

Wady:

- w ten sposób ciągle zostaje zabezpieczona pojedyncza strona,
- aby zabezpieczyć większą liczbę stron, należy znacząco rozbudować przedstawiony mechanizm,
- w przypadku użycia pliku tekstowego znaczące spowolnienie wykonywania skryptu przy większej liczbie użytkowników.

Podstawowa kontrola dostępu po stronie serwera

Istnieją również mechanizmy oparte na protokole HTTP pozwalające zaimplementować podstawową kontrolę dostępu po stronie serwera (ang. *basic authentication*). Implementację takiego mechanizmu można wykonać na dwa sposoby:

- poprzez odpowiednio napisany skrypt PHP,
- poprzez umieszczenie pliku o nazwie *.htaccess* w katalogu, w którym umieszczone będą chronione pliki (listing 5.12).

Listing 5.12. *Przykładowa zawartość pliku .htaccess wymagana do obsługi mechanizmu podstawowej autoryzacji po stronie serwera*

```
AuthType Basic
AuthName "Domena testowa"
AuthUserFile G:\Praca\ProjektyCVS\test\www\.htpasswd
require valid-user
```

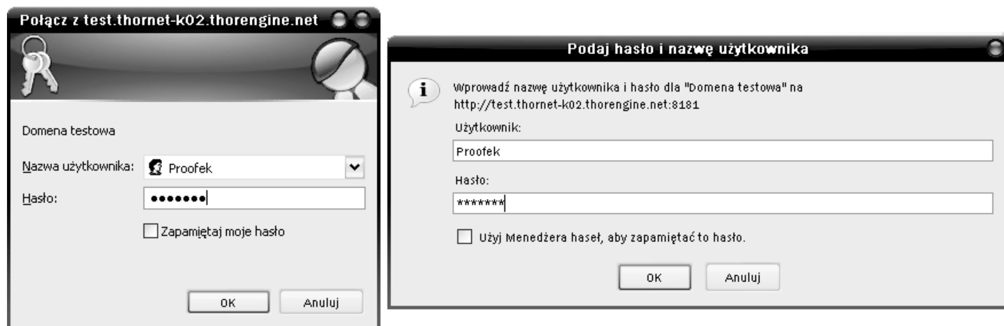
Aby było możliwe korzystanie z plików *.htaccess*, należy wcześniej odpowiednio zmodyfikować plik konfiguracyjny serwera WWW. W przypadku serwera Apache wystarczy umieścić w dyrektywie `<Directory>` opcję `AllowOverride AuthConfig` lub odpowiednio zmodyfikować już istniejący wpis.

Z podstawowej kontroli dostępu po stronie serwera Apache można korzystać po zainstalowaniu modułu `mod_auth`.

Działanie tego mechanizmu wygląda następująco:

- użytkownik prosi o wyświetlenie strony zabezpieczonej mechanizmem,
- serwer zwraca do przeglądarki żądanie przedstawienia się użytkownika,

- przeglądarka wyświetla okno dialogowe (rysunek 5.2), w którym użytkownik podaje nazwę użytkownika oraz hasło,



Rysunek 5.2. *Okno autoryzacyjne wyświetlane przez przeglądarkę Internet Explorer oraz Mozilla Firefox przy mechanizmie podstawowej kontroli dostępu*

- dane przesyłane są do serwera i tam weryfikowane,
- efektem pomyślnej autoryzacji jest uzyskanie dostępu do chronionych stron.

Aby wygenerować plik z hasłami dla użytkowników, należy skorzystać z programu *htpasswd* (w systemie Windows znajduje się on w podkatalogu *bin* katalogu, w którym został zainstalowany serwer Apache). Przykładowe wykorzystanie programu wygląda następująco:

```
htpasswd -c .htpasswd Proofek
```

Pierwszy parametr — `-c` — informuje program o tym, że ma zostać stworzony nowy plik, do którego zostaną zapisane informacje o użytkowniku i haśle, drugi parametr określa nazwę pliku, a trzeci parametr nazwę dodawanego użytkownika. Po uruchomieniu program poprosi jeszcze o wpisanie hasła dla podanego użytkownika.

Zalety:

- prostota i szybkość implementacji,
- możliwość zabezpieczenia całej struktury katalogów,
- w przypadku wykorzystania pliku *.htaccess* przerzucenie na serwer WWW całego mechanizmu autoryzacji oraz sposobu przechowywania danych z informacjami o użytkownikach i ich hasłach.

Wady:

- hasło przesyłane jest w postaci oryginalnej (niezaszyfrowanej),
- brak możliwości wybiórczego chronienia pojedynczych stron,
- w przypadku wykorzystania pliku *.htaccess* wymagane jest użycie osobnego narzędzia do zarządzania użytkownikami lub napisanie własnego skryptu wspomagającego ten proces.

Chroniona kontrola dostępu po stronie serwera

Chroniona kontrola dostępu po stronie serwera (ang. *digest authentication*) jest dużo bezpieczniejszym mechanizmem. Cały mechanizm autoryzacji przebiega dokładnie w taki sam sposób jak w przypadku podstawowej kontroli dostępu po stronie serwera. Różnica polega na tym, że hasła przesyłane są do serwera w postaci zakodowanej algorytmem MD5 (rysunek 5.3). Zawartość pliku *.htaccess* przedstawia listing 5.13.

Rysunek 5.3. *Okno autoryzacyjne wyświetlane przez przeglądarkę Internet Explorer oraz Mozilla Firefox przy mechanizmie chronionej kontroli dostępu*

Listing 5.13. *Przykładowa zawartość pliku .htaccess wymagana do obsługi mechanizmu chronionej autoryzacji po stronie serwera*

```
AuthType Digest
AuthName "Test"
AuthDigestFile /www/.htdigest
require valid-user
```

Z chronionej kontroli dostępu po stronie serwera Apache można korzystać po zainstalowaniu modułu *mod_auth_digest*.

Aby wygenerować plik z hasłami dla użytkowników, należy skorzystać programu *htdigest* (w systemie Windows znajduje się on w podkatalogu *bin* katalogu, w którym został zainstalowany serwer Apache). Przykładowe wykorzystanie programu wygląda następująco:

```
htdigest -c .htdigest "Domena testowa" Proofek
```

Pierwszy parametr — `-c` — informuje program o tym, że ma zostać stworzony nowy plik, do którego zostaną zapisane informacje o użytkowniku, nazwie domeny i haśle, drugi parametr określa nazwę pliku, trzeci parametr nazwę domeny, a czwarty nazwę dodawanego użytkownika. Po uruchomieniu program poprosi jeszcze o wpisanie hasła dla podanego użytkownika.

Zabezpieczenie sesji

Protokół HTTP jako protokół „bezstanowy” nie pozwala na przekazywanie informacji pomiędzy poszczególnymi stronami internetowymi. Z tego właśnie względu w PHP (począwszy od wersji 4.0) zaimplementowano obsługę tzw. sesji. Sam mechanizm obsługi sesji opiera się przede wszystkim na *unikalnym identyfikatorze sesji*, tzw. *SID*. W momencie gdy użytkownik po raz pierwszy kontaktuje się z serwerem, tworzy on unikalny identyfikator sesji i przesyła go razem z żądaną stroną z powrotem do użytkownika. Identyfikator ten jest zazwyczaj zapisywany w pliku cookie na lokalnym dysku twardym komputera użytkownika. W trakcie trwania sesji istnieje możliwość przechowywania na serwerze informacji, do których można odwoływać się podczas danej sesji. Kiedy użytkownik wysyła każde następne żądanie do serwera WWW, przekazuje do niego jednocześnie swój identyfikator sesji. Na jego podstawie aplikacja analizuje wszystkie dane powiązane z tym identyfikatorem i odsyła je wraz z żądaną stroną do użytkownika. Sytuacja taka powtarza się do momentu zakończenia sesji. Po zakończeniu sesji wszystkie informacje w z nią powiązane są kasowane.

Standardowy mechanizm obsługi sesji

W większości przypadków mechanizm obsługi sesji wbudowany w PHP jest w zupełności wystarczający. Dane dotyczące sesji przechowywane są w plikach tekstowych zapisywanych w miejscu określonym w pliku *php.ini* (opcja `session.save_path`). Domyślnie jest to ogólnie dostępny katalog tymczasowy (np. */tmp*). Nie jest to jednak miejsce bezpieczne, gdyż każdy może uzyskać dostęp do danych przechowywanych w tych plikach.

Własny mechanizm obsługi sesji

Napisanie własnego mechanizmu obsługi sesji i zapisywanie danych sesyjnych w bazie danych może znacząco zwiększyć bezpieczeństwo całego systemu. Dodatkowo uzyskujemy możliwość implementacji własnych funkcji w tym mechanizmie, tj. np. śledzenie liczby użytkowników aktualnie odwiedzających sklep; czasu, jaki każdy użytkownik spędza w sklepie; stron, które użytkownik w danym momencie ogląda, itp. Wykorzystanie bazy danych do przechowywania danych sesji rozwiązuje również problem, jaki występuje w momencie, gdy serwis przechowywany jest na kilku serwerach jednocześnie z zastosowaniem równoważenia obciążenia (ang. *load balancing*). W takiej sytuacji standardowy mechanizm obsługi sesji nie nadaje się do użycia.

Aby zaimplementować własny mechanizm obsługi sesji, musimy:

- zaprojektować oraz stworzyć odpowiednią strukturę tabel w bazie danych (listing 5.14),

Listing 5.14. *Skrypt tworzący tabelę active_session*

```
USE Sklep;
CREATE TABLE active_session
(
   SESS_Id                      varchar(32)                  not null,
   SESS_Value                   text,
   SESS_LastUpdate              timestamp,
   primary key (SESS_Id)
) type = InnoDB;
```

- określić mechanizm obsługi sesji w pliku *php.ini*,
- napisać zestaw funkcji obsługujących mechanizmy sesji,
- zarejestrować ten zestaw funkcji jako obsługujący mechanizmy sesji,
- dołączyć definicję klasy do strony, na której chcemy zaimplementować własny mechanizm obsługi sesji.

Tabela przechowująca dane sesji musi przynajmniej przechowywać informacje na temat identyfikatora sesji, danych w niej zarejestrowanych oraz czasu jej ostatniej aktualizacji.

Opcja konfiguracyjna odpowiedzialna za uaktywnienie danego mechanizmu obsługi sesji nosi nazwę `session.save_handler`. Może ona przyjąć wartości:

- `files` (wartość domyślna) — sesje obsługiwane przez wewnętrzny mechanizm PHP, dane sesyjne zapisywane są w plikach tekstowych,
- `user` — zewnętrzny mechanizm obsługi sesji,
- `mm` — sesje obsługiwane przez wewnętrzny mechanizm PHP, dane sesyjne zapisywane są w pamięci współdzielonej (ang. *shared memory*).

Wartość tej opcji można ustawić bezpośrednio w pliku *php.ini* lub też przy wykorzystaniu funkcji `ini_set()`:

```
ini_set('session.save_handler','user');
```

Pozostaje tylko napisać zestaw funkcji, w których zaimplementujemy mechanizmy obsługi sesji. W tym celu stworzymy odpowiednią klasę (listing 5.15) z metodami:

- `_open()` — metoda, która zostanie wywołana w momencie rozpoczęcia nowej sesji lub wznowienia już rozpoczętej. Musi mieć dwa argumenty: ścieżkę dostępu do miejsca, gdzie dane sesji będą przechowywane, oraz nazwę pliku sesji. Obie wartości ustawia się przy wykorzystaniu opcji `session.save_path` oraz `session.name`. Dla nas jednak nie będą miały one żadnego znaczenia, ponieważ dane sesji będziemy przechowywać w bazie danych. Funkcja ta powinna zwracać wartość logiczną `TRUE` lub `FALSE`. W przypadku gdy wcześniej nie nawiązywaliśmy połączenia z naszą bazą danych, w tej właśnie metodzie powinniśmy to zrobić.

- `_close()` — metoda, która zostanie wywołana w momencie zakończenia sesji. Nie ma ona żadnych argumentów oraz podobnie jak metoda `_open()` powinna zwrócić wartość logiczną `TRUE` lub `FALSE`. Użyjemy tej metody do kontrolowanego wywołania metody `_gc()` odpowiedzialnej za usuwanie danych nieaktywnych sesji (ang. *garbage collection*).
- `_read()` — metoda wywoływana przy odczycie danych sesji. Jej jedynym argumentem powinien być poprawny identyfikator sesji (SID), której dane chcemy odczytać. Powinna ona zwrócić dane sesji lub pusty ciąg znaków.

> Zdarza się, że w przypadku gdy ta metoda nie zwróci żadnej wartości, PHP generuje tzw. błąd segmentacji, co w praktyce oznacza zakończenie jego pracy. Mimo że w najnowszych wersjach PHP nie powinno się to już zdarzać, zadbajmy, aby ta metoda zawsze zwracała jakąś wartość.

- `_write()` — metoda wywoływana przy zapisie danych sesji w miejscu określonym w opcjach `session.save_path` oraz `session.name` lub, tak jak w naszym przypadku, w bazie danych. Argumentami tej funkcji powinny być: poprawny identyfikator sesji (SID) oraz zmienna z danymi, które chcemy zapisać. Metoda powinna zwrócić wartość logiczną `TRUE` lub `FALSE`.
- `_destroy()` — metoda wywoływana przy niszczeniu danych sesji. Jej argumentem powinien być poprawny identyfikator sesji (SID) i musi ona zwrócić wartość logiczną `TRUE` lub `FALSE`.
- `_gc()` — metoda służąca do kasowania danych nieaktywnych sesji. Jest wywoływana losowo z prawdopodobieństwem określonym w opcji `session.gc_probability`. Będziemy ją wywoływać z poziomu metody `_close()`. Jedynym argumentem tej metody jest zmienna liczbowa stałoprzecinkowa określająca maksymalny czas życia sesji. Zignorujemy tę wartość i będziemy usuwać dane sesji zgodnie z czasem, który określimy wewnątrz tej metody. Metoda powinna zwrócić wartość logiczną `TRUE` lub `FALSE` oraz właściwość `$_sessionTable` przechowującą nazwę tabeli w bazie danych.

Listing 5.15. *Klasa CustomSession implementująca metodę własnej obsługi mechanizmów sesji*

```
<?php
class CustomSession {
  private $_sessionTable;

  public function __construct() {
    $this->setSessionTable('active_session');
    //Zarejestrowanie własnej obsługi sesji
    session_set_save_handler(
                            array(&$this, "_open"),
                            array(&$this, "_close"),
                            array(&$this, "_read"),
                            array(&$this, "_write"),
                            array(&$this, "_destroy"),
                            array(&$this, "_gc"));
  }
```

```
 public function setSessionTable($tablename) {
   if ($tablename <> NULL) {
     $this->_sessionTable = $tablename;
     return TRUE;
   } else { return new Error('cannotsetsessiontable'); }
}

 public function getSessionTable(){ return $this->_sessionTable; }

 public function _open($save_path, $session_name) {
try {
  DBManager::init();
          return TRUE;
} catch (DBError $err) {
        exit('Error: '.$err->getMessage().'<br/>');
   }
 }

 public function _close() {
   $this->_gc(0);
   return TRUE;
 }

 public function _read($session_id) {
   //Pobranie danych sesji
   $sqlquery = "SELECT SESS_Value FROM " . $this->getSessionTable() . " WHERE SESS_Id
   = '$session_id'";

   try {
     $result = DBManager::Query($sqlquery);

     if ($result->numRows() > 0) {
       if ($sessiondata = $result->fetchRow(DB_FETCHMODE_ASSOC)) {
         return current($sessiondata);
       }
     } else {
       return '';
     }
   } catch (DBError $err) {
        exit('Error: '.$err->getMessage().'<br/>');
   }
 }

 public function _write($session_id, $session_data) {
try {
     //Sprawdzenie, czy sesja już istnieje
     $sqlquery = "SELECT count(*) as 'ilosc' FROM " . $this->getSessionTable() . "
     WHERE SESS_Id = '$session_id'";
     $result = DBManager::Query($sqlquery);
     if ($result->numRows() > 0) {
       //Uaktualnienie danych sesji
       $sqlquery = "UPDATE " . $this->getSessionTable() . " SET SESS_LastUpdate
       = CURRENT_TIMESTAMP(0), SESS_Value = '$session_data' WHERE SESS_Id
       = '$session_id'";
            DBManager::Query($sqlquery);
```

```
        if (DBManager::AffectedRows() > 0) {
          return TRUE;
        } else { return FALSE; }
      } else {
        //Zapisanie nowych danych sesji
        $sqlquery = "INSERT INTO " . $this->getSessionTable() . " (SESS_Id,
        SESS_LastUpdate, SESS_Value) VALUES ('$session_id', CURRENT_TIMESTAMP(0),
        '$session_data')";
        DBManager::Query($sqlquery);
        if (DBManager::AffectedRows() > 0) {
          return TRUE;
        } else { return FALSE;}
      }
  } catch (DBError $err) {
         exit('Error: '.$err->getMessage().'<br/>');
    }
   }

   public function _destroy($session_id) {
     try {
       //Usunięcie danych sesji
       $sqlquery = "DELETE FROM " . $this->getSessionTable() . " WHERE SESS_Id
       = '$session_id'";
       DBManager::Query($sqlquery);
       return TRUE;
     } catch (DBError $err) {
         exit('Error: '.$err->getMessage().'<br/>');
     }
   }

   public function _gc($maxlifetime) {
     $ses_life = strftime('%Y-%m-%d %H:%M', strtotime("-5 minutes"));
  try {
       //Usunięcie przedawnionych danych sesji
       $sqlquery = "DELETE FROM " . $this->getSessionTable() . " WHERE SESS_LastUpdate
       < '$ses_life'";
    DBManager::Query($sqlquery);
    return DBManager::AffectedRows();
  } catch (DBError $err) {
         exit('Error: '.$err->getMessage().'<br/>');
    }
   }
 }
 ?>
```

Dodatkowego wyjaśnienia w zasadzie wymagają jedynie trzy metody: konstruktor tej klasy, metoda `_write()` oraz `_gc()`.

W konstruktorze klasy, oprócz określenia nazwy tabeli w bazie danych, w której zapisywane będą dane sesji, musimy zarejestrować metody naszej klasy jako funkcje obsługujące mechanizmy sesji. Do tego celu wykorzystamy funkcję `session_set_save_handler()`.

```
bool session_set_save_handler (string open, string close, string read, string write,
string destroy, string gc)
```

Argumentami tej funkcji powinny być kolejno nazwy funkcji odpowiedzialne za: otwarcie sesji, zamknięcie sesji, odczytanie danych sesji, zapisanie danych sesji, zniszczenie danych sesji oraz niszczenie danych starych sesji. Ze względu na to, że korzystamy z metod klasy, parametrami tej funkcji muszą być tablice zawierające kolejno nazwę obiektu obsługującego mechanizmy sesji oraz nazwę konkretnej metody.

W metodzie `_write()` należy zwrócić uwagę na kolejność przeprowadzanych operacji. Najpierw musimy sprawdzić, czy operacja zapisu danych nie dotyczy sesji, która już istnieje. W takiej sytuacji uaktualniamy dane istniejącej sesji i kończymy wykonanie metody. W przypadku gdy zapisywane dane dotyczą nowej sesji, musimy dodać nowy wiersz do tabeli `active_session`.

W przypadku metody `_gc()` — ze względu na to, że ignorujemy wartość jej argumentu — musimy na początku określić, które sesje mamy traktować jako przedawnione. Ustaliliśmy maksymalny czas życia sesji na `5 minut`.

```
$ses_life = strftime('%Y-%m-%d %H:%M', strtotime("-5 minutes"));
```

Można oczywiście przyjąć dowolną wartość tego znacznika czasu, jednak uważamy, że pięć minut bezczynności po stronie użytkownika jest wystarczająco długim okresem pozwalającym stwierdzić, iż przestał on korzystać ze sklepu.

> Jeżeli serwer WWW oraz serwer bazy danych zostały zainstalowane na osobnych maszynach, należy upewnić się, iż ustawione na nich daty systemowe są takie same. Najlepiej skonfigurować je tak, aby synchronizowały czas z tym samym serwerem czasu. Jeżeli między tymi maszynami będzie istnieć różnica czasowa, obsługa sesji może być nieprawidłowa.

Przykład 5.2

Aby zaimplementować w aplikacji własny mechanizm obsługi sesji oparty na bazie danych, należy:

1. Utworzyć odpowiednią strukturę bazy danych. Skrypt SQL tworzący tę strukturę znajdziemy na listingu 5.16.

Listing 5.16. *Testowa instrukcja switch sprawdzająca poprawność mechanizmu obsługi sesji*

```
if (isSet($_GET['action']))
   switch ($_GET['action']) {
         case 'new':
           $_SESSION['test'] = 'Nowa zmienna sesyjna';
           break;
         case 'change':
           $_SESSION['test'] = 'Zmieniona zmienna sesyjna';
           break;
         case 'clear':
           $_SESSION['test'] = NULL;
           break;
   }
```

2. Aplikację przygotujemy na podstawie struktury aplikacji, którą poznaliśmy w rozdziale 4., rozbudowanej o obsługę bazy danych. Struktura ta znajduje się na CD-ROM-ie dołączonym do książki w katalogu *materialy/rozdzial05/struktura_aplikacji*. Skopiujmy ją na serwer WWW, tak aby katalog *www* był katalogiem *rootwww* dla serwera.

   W pliku *www/index.php* dołączymy do skryptu za pomocą funkcji `require_once()` plik *config/custom_session.inc.php* oraz rozpoczniemy sesję za pomocą funkcji `session_start()`:

   ```
   require_once('../config/custom_session.inc.php');
   session_start();
   ```

3. Prosta instrukcja `switch` pozwoli przetestować poprawność obsługi sesji (listing 5.16). Dzięki trzem różnym akcjom możemy stworzyć nową zmienną sesyjną (akcja `new`), zmienić jej wartość (akcja `change`) lub wyczyścić jej zawartość (akcja `clear`). Kod z listingu 5.16 należy dopisać do pliku *www/index.php*. Opcja przeładowania strony pozwoli sprawdzić, czy informacje rzeczywiście trzymane są w sesji.

4. Na koniec zmodyfikujmy jeszcze szablon *index.tpl*, dopisując do niego następującą treść:

   ```
   Zmienna sesyjna: {$smarty.session.test|default:"Brak wartości"}<br/>
   <a href="index.php?action=new">Utwórz zmienną sesyjną</a><br/>
   <a href="index.php?action=change">Zmień wartość zmiennej sesyjnej</a><br/>
   <a href="index.php?action=clear">Wyczyść zawartość zmiennej sesyjnej</a><br/><br/>
   <a href="index.php">Przeładuj stronę</a><br/>
   ```

Ze względu na to, że w konfiguracji Smarty domyślnie włączyliśmy tryb obsługi pamięci podręcznej (stała `SMARTY_CACHING` w pliku konfiguracyjnym *define.inc.php*), zmiany na stronie mogą nie pojawiać się szybko — pamiętajmy, że wtedy Smarty nie przekompiluje szablonu, tylko pobierze go z pamięci podręcznej. Na czas testów dobrze ustawić wartość stałej `SMARTY_CACHING` na `FALSE`.