helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

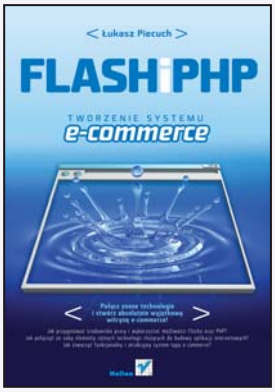
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Flash i PHP. Tworzenie systemu e-commerce



Autor: Łukasz Piecuch
ISBN: 83-246-1740-X
Format: 158×235, stron: 272

Połącz znane technologie i stwórz absolutnie wyjątkową witrynę e-commerce!

- Jak przygotować środowisko pracy i wykorzystać możliwości Flasha oraz PHP?
- Jak połączyć z sobą elementy różnych technologii, służących do budowy aplikacji internetowych?
- Jak stworzyć funkcjonalny i atrakcyjny system typu e-commerce?

Tworzenie statycznych lub naszpikowanych migającymi obrazkami stron internetowych jest już dzisiaj passé. Nikt nie lubi ani nudnych stron, tworzonych w HTML-u, ani rozpraszających, ruchomych obrazków, powtarzających się w nieskończoność. Dobrze zaprojektowana, funkcjonalna i estetyczna witryna musi być to taka która jest przejrzysta i interaktywna. Powinna także działać szybko i niezawodnie, a wyniki wyszukiwania czy innej aktywności użytkownika prezentować w atrakcyjnej i zrozumiałej formie. Taki efekt może zapewnić połączenie kilku znanych technologii – HTML-a, Flasha oraz PHP. Pozwala ono tworzyć zupełnie wyjątkowe aplikacje internetowe: z kodem wykonywanym po stronie serwera, z doskonałą strukturą i fantastyczną nawigacją, zapewniającą internautom pełny komfort poruszania się po serwisie.

Książka „Flash i PHP. Tworzenie systemu e-commerce” zawiera kompletny opis tworzenia takiej doskonałej witryny, łączącej w sobie elementy różnych języków i technik programowania. Jej autor krok po kroku przeprowadzi Cię przez cały proces projektowy, od momentu powstania pomysłu na e-biznes, do chwili ukończenia programowania. Pokaże Ci, jak działają różne systemy e-commerce, jakie są dostępne rozwiązania i jakie wzorce projektowe można wykorzystać. Wyjaśni, co to jest streaming, pomoże zbudować interfejs, bazę danych i katalog produktów, będzie Ci także towarzyszyć podczas procesu instalacji i obsługi strony. Z jego pomocą poradzisz sobie z najtrudniejszymi zadaniami – tym bardziej, że na starcie oczekuje on od Ciebie jedynie podstawowej wiedzy o wykorzystywanych językach programowania.

- Systemy e-commerce – dostępne rozwiązania, Flash, PHP, MySQL
- środowisko pracy – Apache, PHP i MySQL, Adobe Flash CS4 Professional, Adobe Photoshop CS4
- Flash – ładowanie zmiennych z zewnątrz, techniki połączenia, streaming audio i video
- Wzorzec projektowy MVC i założenia projektu
- Przygotowanie do pracy bazy danych i Frameworka
- Programowanie Front-end
- Tworzenie interfejsu we Flashu i panelu administracyjnego
- Instalacja i obsługa systemu

Stwórz estetyczny i komfortowy sklep internetowy!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2010

Spis treści

O autorze **5**

Wstęp **7**

Rozdział 1. Systemy e-commerce **11**
Co decyduje o popularności systemów e-commerce 11
Dostępne rozwiązania 12
Dlaczego Flash? 13
Dlaczego PHP? 14
Dlaczego MySQL? 14

Rozdział 2. Przygotowujemy środowisko pracy **15**
Instalujemy Apache, PHP i MySQL 15
Konfiguracja 17
Wybieramy edytor kodu 27
Instalujemy Adobe Flash CS4 Professional 29
Przygotowujemy program do pracy 33
Instalujemy Adobe Photoshop CS4 34
Przygotowujemy program do pracy 36

Rozdział 3. Jak Flash łączy się ze światem **39**
Ładowanie zmiennych z zewnątrz 39
Query String, czyli zmienne w adresie 39
Ładowanie zmiennych zapytania HTTP 43
Zmienne w pliku tekstowym 47
Dostępne techniki połączenia 52
Połączenia Flash – Flash 52
Połączenia Flash – JavaScript 56
Połączenia Flash – PHP – MySQL 59
Zagnieżdżanie zewnętrznych obrazów we Flashu 64
Zagnieżdżanie zewnętrznych plików swf we Flashu 74
Flash i XML 75
Streaming 88
Streaming audio 88
Streaming wideo 91

Rozdział 4. Projekt ... 97
Wzorzec projektowy MVC ... 97
Budowa frameworka ... 98
Działanie frameworka ... 100
Funkcje frameworka ... 107
Założenia projektu ... 110
Ogólne założenia ... 110
Projekt bazy danych ... 111
Projekt kontrolerów, modeli oraz widoków ... 114
Bezpieczeństwo ... 118
Projekt interfejsu ... 119

Rozdział 5. Tworzenie systemu ... 125
Przygotowanie bazy danych ... 125
Wypełnienie przykładowymi danymi ... 128
Przygotowanie frameworka do pracy ... 129
Konfiguracja połączenia z bazą danych ... 129
Programujemy front-end ... 130
Katalog produktów ... 130
Koszyk ... 140
Składanie zamówień ... 145
Tworzymy interfejs we Flashu ... 150
Katalog produktów ... 174
Koszyk ... 194
Składanie zamówień ... 202
Ostatnie szlify ... 210
Tworzymy panel administracyjny ... 215
Tworzymy menu panelu administracyjnego ... 218
Obsługa asortymentu ... 218
Obsługa zamówień ... 250

Rozdział 6. Instalacja i obsługa ... 259

Skorowidz ... 263

Rozdział 4.
Projekt

W poprzednich rozdziałach przyjrzeliśmy się programistycznym aspektom łączenia różnych technologii z ActionScript. Nadszedł czas na wykorzystanie tej wiedzy w praktyce. W kolejnych rozdziałach postaramy się wykorzystać zdobytą wiedzę przy tworzeniu prostego, lecz w pełni funkcjonalnego sklepu internetowego, czyli systemu e-commerce. Naturalnie przed przystąpieniem do programowania musimy całość zaprojektować i omówić. Faza projektowania jest pierwszą kluczową fazą w tworzeniu jakiegokolwiek oprogramowania, dlatego też warto poświęcić kilka chwil, chwycić długopis w dłoń i wypisać założenia systemu oraz rozrysować szczegółowo budowę. Przed przystąpieniem do projektowania systemu warto przeczytać omówienie przygotowanego przeze mnie do tego celu frameworka. Jest to szkielet kontrolera *MVC*, czyli *Model-View-Controller*, oparty na języku PHP. Pozwoli on nam na stworzenie logicznej, hierarchicznej organizacji oprogramowania. Kolejnym krokiem będzie omówienie modułów, z jakich składał się będzie system. W tym kroku zdecydujemy, jakie zadanie będzie realizował PHP, a jakie ActionScript. Następnie zaprojektujemy bazę danych, która będzie przechowywać dla nas informacje o produktach i zamówieniach. Jeśli o niczym nie zapomniałem, to pod koniec tego rozdziału powinieneś mieć w głowie klarowny obraz tego, w jaki sposób program będzie zbudowany i jak będzie funkcjonował. Wytłumaczę też zależności pomiędzy poszczególnymi funkcjami i modułami systemu. Zatem do dzieła.

Wzorzec projektowy MVC

Czym jest wzorzec projektowy? Można powiedzieć, że są to pewne uniwersalne, wypracowane przez „pokolenia", sprawdzone w praktyce rozwiązania problemów projektowych. Nie można powiedzieć, że są to konkretne funkcjonalne rozwiązania jakichś zadań czy problemów, lecz raczej jest to podwalina i szkielet, na którym takowe będą się opierać. Jak donosi moja ulubiona encyklopedia (Wiki oczywiście), wzorzec *MVC* (ang. *Model-View-Controller* — Model-Widok-Kontroler) to „architektoniczny" wzorzec projektowy w informatyce, którego głównym założeniem jest wyodrębnienie trzech podstawowych komponentów aplikacji:

- modelu danych,
- interfejsu użytkownika,
- logiki sterowania[1].

Definicja może nie mówi Ci wiele, ale już spieszę z wyjaśnieniami. Wzorzec MVC stanowi niejako szkielet tworzonych aplikacji. Wyznacza logikę organizacji kodu. Zmusza programistów do czytelnej organizacji modułów oprogramowania. Oddziela od siebie interfejs użytkownika, logikę sterowania oraz dane. Rozważmy to na prozaicznym przykładzie sklepu. Użytkownik końcowy wybiera produkt, operując w warstwie interfejsu. Przyciskając przyciski, wywołuje funkcje **kontrolera**, czyli logiki sterowania. **Kontroler** decyduje, jakie operacje na danych przeprowadzić w zależności od działań użytkownika. Następnie przekazuje zlecenia wykonania operacji na danych do odpowiedniego **modelu**. Ten zwraca nieobrobiony wynik swoich operacji. **Kontroler** opracowuje wyniki działań modelu i decyduje o kolejnym kroku, czyli np. wyświetla kolejną stronę bądź odpowiedni komunikat. Tak w skrócie wygląda sposób działania wzorca MVC. Na podstawie wzorców MVC powstało i wciąż powstaje wiele frameworków, czyli szkieletów wyposażonych w biblioteki i rozwiązania ułatwiające i przyśpieszające programowanie. Przykładem niech będą: CakePHP, Zend Framework czy Symfony. Można na ten temat dyskutować, jednak nie jestem fanem korzystania z ogólnodostępnych rozwiązań i o wiele bardziej cenię sobie te własne, opracowane w pocie czoła, ulepszane przy tworzeniu kolejnych projektów. Dlatego właśnie specjalnie na potrzeby naszego projektu przygotowałem prosty framework, na którym będziemy tworzyć nasz system. Na początek przyjrzymy się jego budowie. Następnie prześledzimy sposób działania oraz poznamy jego funkcje.

Budowa frameworka

Framework zbudowany jest z dwóch zasadniczych części: jądra oraz samej aplikacji. Jądro frameworka znajduje się w katalogu *core* i jest niezmienne, chyba że wprowadzamy niezbędne poprawki i unowocześnienia. Sama zaś aplikacji znajduje się w katalogu *app*.

- Katalog *core*:
  - *addons* — przechowuje wszelkie dodatki, biblioteki;
  - *classes* — przechowuje kluczowe dla działania frameworka klasy, konstruktory i funkcje;
  - *includes* — zawiera plik *startup.php*.
- Katalog *app*:
  - *controllers* — kontrolery aplikacji;
  - *models* — modele aplikacji;
  - *wiews* — interfejs aplikacji;
  - *webroot* — skrypty *js*, grafika, style *css*, pliki *Flash*.

[1] *http://pl.wikipedia.org/wiki/MVC*

Istotny jest również plik *index.php*, który zawiera podstawową konfigurację — ustawienia dostępu do bazy danych. Jądro frameworka można z powodzeniem rozszerzać o nowe funkcje poprzez jego edycję bądź dodawanie nowych klas do katalogu *addons*. Natomiast wszelkie pliki programu będziemy umieszczać w katalogu *app*. Pliki zawierające logikę aplikacji znajdą się zatem w katalogu *controllers*, a klasy modeli realizujące dostęp do danych — w katalogu *models*. Pliki interfejsu, czyli dokumenty HTML (lecz zapisywane z rozszerzeniem php) zapisywać będziemy w katalogu *views*. Warto przyjrzeć się temu katalogowi bliżej, gdyż został on podzielony na kilka logicznych części:

- *elements*,
- *layouts*,
- *pages*,
- *texts*.

W katalogu *elements* umieszczać będziemy małe, często występujące i powtarzające się w witrynie elementy typu koszyk, menu, boksy z newsami itp. Jest to doskonałe miejsce na przechowywanie elementów ładowanych przez Ajax.

Katalog *layouts* zawiera natomiast szablony witryny, czyli jej powtarzalny szkielet — szablon. Szablonem takim będzie dokument HTML z nagłówkami, stopką, stałymi elementami typu menu.

W katalogu *pages* umieszczać będziemy poszczególne strony witryny. Jako że front-end naszej aplikacji wykonany będzie w technologii Flash, katalog ten posłuży nam głównie do przechowywania stron panelu administracyjnego.

Katalog *texts* przechowuje pliki ze specyficznymi tekstami, głównie komunikatów witryny. Każda wersja językowa witryny posiada osobny plik z listą komunikatów. Zatem gdy użytkownik zmieni język witryny, system załaduje odpowiedni plik z tłumaczeniem i witryna zacznie komunikować się z nim w rodzimym języku.

Jeśli już jesteśmy przy wersjach językowych witryny, muszę nadmienić, iż komunikaty językowe to tylko mały procent tekstów znajdujących się w witrynie. Dlatego też w każdym katalogu z *views* znajduje się osobny folder oznaczony skrótem językowym w postaci „pl”, „eng”, „de” itp. Dlatego po zmianie języka witryny system wczyta adekwatne elementy, szablony oraz strony.

Zobaczmy jeszcze, co kryje się w folderze *webroot*:

- *css*,
- *files*,
- *img*,
- *js*,
- *upload*,
- *xml*.

Przeznaczenie poszczególnych katalogów mówi samo za siebie. Jedynie znaczenie katalogu *upload* może nie być do końca jasne. Służy on do przechowywania wszelkiej zawartości przesyłanej przez użytkowników, np. obrazków do newsów ładowanych przez redaktorów, avatarów rejestrujących się użytkowników itp.

Działanie frameworka

Po uruchomieniu witryny wczytywany jest plik *index.php*. Jednak warto nadmienić, iż przed nim ma miejsce interpretacja pliku *.htaccess*, w którym to zawarto procedurę niezbędną do realizacji zamysłu „przyjaznych linków". Otóż wszystkie używane w systemie linki są w postaci: *adreswitryny.com/kontroller/akcja/parametry* w odróżnieniu od paskudnych *adreswitryny.com?controller=xxx&action=yyy&arg1=zzz&arg2=qqq*.

Jak już wcześniej wspomniałem, w pliku *index.php* znajduje się kilka linii kodu odpowiedzialnych za konfigurację dostępu do baz danych:

```
$dsn = array(
    'phptype'  => 'mysql',
    'username' => '',
    'password' => '',
    'hostspec' => 'localhost',
    'database' => '',
);
```

Przed uruchomieniem frameworka należy te dane bezwzględnie ustawić. Zmienna `username` oznacza nazwę użytkownika, `password` — hasło do bazy danych, `hostspec` — adres serwera baz danych, `database` — nazwę bazy. Jest to praktycznie jedyne ustawienie niezbędne do działania frameworka. Opcjonalnymi ustawieniami jest ustawienie domyślnego języka witryny oraz adresów e-mail systemu oraz administracji. Adres e-mail systemu to adres, który będzie pojawiał się w polu nadawcy w e-mailach wysyłanych przez system, natomiast adres administracji to adres, na który system wysyła e-maile związane z działaniem, np. dane backupu.

W pliku *index.php* tworzone są kolejno instancje niezbędnych klas: `Registry`, `Session`, `Database`, `Template`, `Router`. Gdy w wywołaniu podamy nazwę kontrolera, router go dla nas załaduje, jeśli nie — załaduje domyślną klasę kontrolera `home`. `Router` znajdzie też dla nas zadeklarowane w kontrolerze modele i je załaduje, to samo dotyczy dodatków `addons`. Jeśli wywołanie zawiera nazwę akcji kontrolera, zostanie ona znaleziona i wywołana, jeśli nie — router wykona funkcję `index` kontrolera. Przed każdą akcją zawsze i bezwzględnie wywoływana jest funkcja `BeforeFilter`. Jest to doskonałe miejsce do umieszczenia procedur sprawdzających poprawność logowania w kontrolerze administracji czy też prozaicznego ustawiania tytułu strony. Procedury zawarte w tej funkcji wykonywane są przed każdą wywoływaną akcją, dlatego gdy istnieje konieczność wywoływania pewnych partii kodu przy każdej z nich, jest to najlepsze miejsce. W funkcji kontrolera użyjemy modeli oraz załadujemy odpowiednie widoki, ale o tym za chwilę.

Zróbmy małe ćwiczenie. Wyświetlmy typowe *Hello World* i rozpocznijmy naszą przygodę z tym jakże prostym i ułatwiającym życie frameworkiem. Skopiuj katalog z frameworkiem do swojego katalogu serwera, czyli *htdocs*, jeśli używasz *xampp*.

Czysty framework znajduje się na dołączonej płycie CD, w katalogu *Framework*.

Otwórz katalog *app/controllers*, a następnie otwórz plik *home.php*. Jak widzisz, w pierwszych dwóch liniach deklarujemy nazwę bądź nazwy używanych w kontrolerze modeli oraz nazwę szablonu, z jakiego korzystają domyślnie strony wyświetlane przez kontroler (można to zmieniać dla poszczególnych akcji w wywołaniu strony: `$this->tem ↪plate->show('home/index', 'nazwa_szablonu');`. Załóżmy, że chcemy wysłać do interfejsu zmienną z tekstem *Hello World*. Aby to zrobić, skorzystamy z metody `set` klasy `template`. Dodajmy zatem do funkcji `index` linię:

```
$this->template->set('zmienna','Hello world');
```

Kod ten należy dodać przed wywołaniem strony — jest to logiczne, gdyż jaki sens miałoby wysyłanie do szablonu zmiennych po jego wyświetleniu. Funkcja będzie wyglądać następująco:

```
function index($args=null,$post=null){
         $this->template->set('zmienna','Hello world');
         $this->template->show('home/index', 'index');
}
```

Kolejnym krokiem jest wyświetlenie zmiennej w widoku. Przechodzimy zatem do katalogu *views/pages/pl/home* i otwieramy plik *index.php*. Wpiszmy w nim następujący kod:

```
Tutaj pojawi się zmienna ustawiona w kontrolerze:<br />
<?php echo $zmienna; ?>
```

Po uruchomieniu programu naszym oczom powinien ukazać się powyższy tekst oraz ustawiona przez nas zmienna. Aby sprawdzić działanie systemu, w przeglądarce należy wpisać adres serwera oraz nazwę katalogu, w którym znajduje się framework, np.: *http://localhost/test1*. Powinieneś zobaczyć napis:

Tutaj pojawi się zmienna ustawiona w kontrolerze:

Hello World

Kiedy otworzysz źródło, zauważysz, że poza wpisaną przez Ciebie treścią w pliku *views/pages/pl/home/index.php* znajduje się również reszta struktury dokumentu HTML. Jest to zawartość pliku *views/layouts/pl/index.php*. Struktura źródła została przedstawiona na rysunku 4.1.

Warto zauważyć, iż akcja `index` kontrolera `home` wywoływana jest jako domyślna. Możemy ją również wywołać ręcznie, wpisując w polu adresu przeglądarki: *http://localhost/test1/home/index*.

Utwórzmy teraz kolejną akcję kontrolera. Pod deklaracją funkcji `index` w pliku *home.php* utwórzmy kolejną funkcję o nazwie np. `proba`:

```
function proba($args=null,$post=null){

}
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl" lang="pl">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta name="keywords" content="" />
    <meta name="description" content="" />
    <meta name="language" content="pl" />
    <meta name="Pragma" content="no-cache" />
    <meta name="Cache-Control" content="no-store, no-cache, must-revalidate" />
    <meta name="revisit-after" content="2 days" />
    <meta name="Robots" content="index,follow,all" />
    <meta name="googlebot" content="index,follow,all" />
    <meta name="msnbot" content="index,follow,all" />
    <title>Tytuł strony</title>
    <base href="http://localhost/Ksiazka/Framework_cwiczenia/test1/app/webroot/" />

    <link href="css/style.css" rel="stylesheet" type="text/css" />
    <link rel="stylesheet" href="css/jquery.lightbox.css" type="text/css" media="screen" />

    <style type="text/css">
                img(
                      border:0px;
        behavior: url('http://localhost/Ksiazka/Framework_cwiczenia/test1/app/webroot/css/pngbehaviour.htc');
                )
    </style>

    <script type="text/javascript" src="js/jquery-1.3.2.min.js"></script>
    <script type="text/javascript" src="js/jquery.lightbox.js"></script>
    <script type="text/javascript" src="js/AC_RunActiveContent.js"></script>
</head>
<body>

Tutaj pojawi się zmienna ustawiona w kontrolerze:<br />
Hello world <- Zmienna $zmienna przesłana przez kontroler home.

</body>
</html>
```

Zawartość pliku views/layouts/pl/index.php

Zawartość pliku views/pages/pl/home/index.php

Zawartość pliku views/layouts/pl/index.php

Rysunek 4.1. *Struktura dokumentu HTML wygenerowanego przez framework*

Nic nie stoi na przeszkodzie, abyśmy użyli tego samego widoku. Jednak przećwiczymy teraz ładowanie obrazków oraz skorzystamy z funkcji modelu. Utwórz więc w katalogu *views/pages/pl/home* plik o nazwie *proba.php*. Będzie to nasz plik HTML dla akcji `proba` kontrolera `home`. Zadeklaruj w funkcji akcji nasz plik widoku:

```
function proba($args=null,$post=null){
          $this->template->show('home/proba', 'index');
 }
```

Otwórz plik *proba.php* i umieść w nim dowolną zawartość. Aby wyświetlić na stronie obrazek, wystarczy takowy skopiować do katalogu *webroot/img/home/*, a w kodzie strony HTML umieścić:

```
<img src="img/home/1.jpg" alt="" />
```

System sam odnajdzie odpowiednią ścieżkę do obrazka. Akcję wywołamy, wpisując *http://localhost/test1/home/proba* w przeglądarce.

Korzystamy z modelu

Wspominałem wcześniej o tym, iż model służy do operacji odczytu i zapisu danych. Tworząc różnego rodzaju aplikacje, staram się przeprowadzać w modelu operacje możliwie niskiego poziomu, lecz o kluczowym znaczeniu, a kontrolerowi pozostawiać rolę decyzyjną w oparciu o dane wynikowe dostarczone przez model. W modelach będziemy zatem pobierać dane z baz danych oraz je zapisywać. Będziemy kopiować pliki i przeprowadzać na nich operacje typu kompresja i skalowanie.

Otwórzmy zatem plik *models/home.php*. Znajduje się w nim tylko konstruktor i dwie funkcje odpowiedzialne za zmianę języka witryny. Utwórzmy więc własną funkcję, która zwróci do kontrolera pewne dane, tak abyśmy mieli pogląd na sposób działania modelu. Przyjmijmy, że nazwiemy ją `Get_users`:

```
function get_users(){

 }
```

W funkcji utwórzmy tablicę z fikcyjnymi imionami użytkowników:

```
function get_users(){
          $users = array('Marek','Agnieszka','Karolina','Łucja','Franek');
          return $users;
 }
```

To wszystko, jeśli chodzi o model. Wywołanie naszej funkcji w kontrolerze będzie wyglądać następująco:

```
$users = $this->model->home->get_users();
```

Pozostaje nam jeszcze przesłać zdobyte dane do widoku:

```
$this->template->set('uzytkownicy',$users);
```

Zatem akcja `proba` będzie wyglądać w tej chwili następująco:

```
function proba($args=null,$post=null){
          $users = $this->model->home->get_users();

          $this->template->set('uzytkownicy',$users);
          $this->template->show('home/proba', 'index');
 }
```

Wywołanie strony w przeglądarce niewiele zmieni, gdyż przesłaną zmienną `uzytkownicy` trzeba jeszcze wykorzystać w pliku HTML widoku. Zrobimy to na przykład tak:

```
<?php
  if($uzytkownicy){
    $i=0;
    foreach($uzytkownicy as $user){
      $i++;
      echo '<b>'.$i.'.</b>'.$user.'<br />';
    }
  }
?>
```

Kod pozwoli na wyświetlenie listy użytkowników wraz z numerem porządkowym.

Zrealizowaną akcję możemy teraz wyświetlić w przeglądarce, wpisując: *http://localhost/test1/home/proba*.

W zależności od tego, co wpisałeś w kodzie widoku, Twoja strona może wyglądać mniej więcej tak, jak na rysunku 4.2.

Gotowy program znajdziesz na płycie CD pod nazwą *test1* w katalogu *Framework_cwiczenia*.

Rysunek 4.2.
Wynik działania aplikacji

Tworzenie nowego kontrolera, modelu i widoku

Nic nie stoi na przeszkodzie, aby utworzyć nowe kontrolery, które będą odpowiedzialne za różne moduły realizowanego projektu. Taka budowa oprogramowania pozwala w prosty sposób adaptować pewne moduły i rozwiązania w innych projektach. Równie dobrze możemy tworzyć całkiem nowe modele i wykorzystywać je zamiennie w kontrolerach. Każdy kontroler może korzystać z różnych modeli.

Przeprowadźmy zatem ćwiczenie, w którym utworzysz, Drogi Czytelniku, kontroler, model i odpowiednie pliki widoków.

Aby utworzyć nowy kontroler, należy:

1. W katalogu *app/controllers/* utworzyć plik *nazwa_kontrolera.php* — pamiętając, aby w miejsce *nazwa_kontrolera* wpisać wymyśloną przez siebie nazwę, poprzez którą będziemy odwoływać się do metod kontrolera. Nie należy używać polskich znaków, spacji oraz dużych liter.
2. Otworzyć utworzony plik kontrolera i umieścić w nim następujący kod:

```
<?php
$models=array('home');
$layout='index';

Class Controller_Nazwa_Kontrolera Extends Controller_Base {

        function index($args=null,$post=null){

        }

      ////////////////////////////////////////////////////////////////////////

       function BeforeFilter($action){
          $this->template->set('action',$action);
          $this->template->set('lang', $_SESSION['languageID']);
          $this->template->set('pageTitle', 'Tytuł strony');
        }

        function deadend($args=null,$post=null){
```

```
            $this->template->show('home/deadend', 'index');
        }

    }
    ?>
```

W tablicy `$models` deklarujemy wszystkie modele, z których chcemy korzystać, zaś zmienna `$layout` służy do określenia domyślnego szablonu wczytywanego przy wyświetlaniu stron HTML. Należy zwrócić uwagę na linię `Class Controller_`*`Nazwa_Kontrolera`* `Extends Controller_Base`, gdyż należy w niej wpisać nazwę kontrolera w postaci `Controller_PrzykladowyKontroler`. To wszystko. Nowy kontroler jest utworzony i gotów do pracy. Wystarczy tylko wypełnić go funkcjami, czyli tak zwanymi akcjami kontrolera.

Dążąc do utworzenia nowego modelu, przygotujmy odpowiednio plik kontrolera. Doskonale wiesz, jak to zrobić. Wystarczy w zmiennej `$models` dodać nazwę utworzonego modelu. Oczywiście kontroler może równolegle korzystać z wielu modeli. Wyobraź sobie taką sytuację: tworzysz oprogramowanie do obsługi firmy hostingowej. Takie oprogramowanie musi zawierać kilka kluczowych modułów. Jednym z nich będzie moduł do zarządzania klientami, drugim modułem będzie obsługa pakietów hostingowych, trzecim modułem może być helpdesk, czyli czaty do komunikacji z pomocą techniczną. Każdy z tych modułów powinien posiadać oddzielny plik modelu w celu zachowania logiki działania, modułowości, ułatwienia obsługi i zachowania pewnej kultury programowania. Zatem utworzymy w takim wypadku modele o nazwach `clients`, `products` i `helpdesk`. W modelu `clients` znajdą się wszystkie funkcje odpowiedzialne za tworzenie nowych klientów, składanie przez nich zamówień na produkty z modułu `products`, obsługę płatności (gdyby ten moduł był rozbudowany i w grę wchodziłaby obsługa kilku czy kilkunastu różnych kanałów płatności, nic nie stoi na przeszkodzie, aby posiadał własny model), bilingowanie. W modelu `products` znajdą się funkcje odpowiedzialne za zarządzanie produktami, czyli pakietami hostingowymi. Model `helpdesk` będzie odpowiedzialny za przeprowadzanie komunikacji pomiędzy użytkownikiem a pracownikiem obsługi za pomocą czatów wykonanych w dowolnej technologii (czemu by nie Flash?) czy też internal messagingu.

Zmodyfikujmy zatem odpowiednio nowo stworzony kontroler o nazwie `shopping` (będzie odpowiedzialny za wyświetlanie produktów i dokonywanie zakupu), tak aby przystosować go do korzystania z trzech modeli: `home`, `products`, `clients`. Zapewne będzie to dla Ciebie pikuś, ale gwoli ścisłości poniżej znajdziesz pełny kod kontrolera (listing 4.1).

Listing 4.1. *Kod kontrolera products*

```
<?php
$models=array('home','products','clients');
$layout='index';

Class Controller_Shopping Extends Controller_Base {

        function index($args=null,$post=null){

        }

//////////////////////////////////////////////////////////
```

```
        function BeforeFilter($action){
           $this->template->set('action',$action);
           $this->template->set('lang', $_SESSION['languageID']);
           $this->template->set('pageTitle', 'Tytuł strony');
         }

         function deadend($args=null,$post=null){
           $this->template->show('home/deadend', 'index');
         }
}

?>
```

Po zapisaniu zmian spróbuj uruchomić w przeglądarce system, wywołując kontroler `shopping`. Powinieneś zobaczyć następujący komunikat o błędzie:

404 — Model File Not Found

Oznacza to, iż nasze modele jeszcze nie istnieją. Musimy je więc utworzyć.

Aby utworzyć nowy model, należy:

1. W katalogu *app/model/* utworzyć plik o nazwie modelu, np. *products.php*.
2. Otworzyć plik i umieścić w nim kod:

```
<?php

Class Model_Products{

        function __construct($registry) {
                $this->registry = $registry;
                $this->db=$registry->db;
                $this->error=$registry->error;
                $this->text=$registry->text;
                $this->addon=$registry->addon;
        }

}

?>
```

Tak jak i w wypadku kontrolera należy zwrócić uwagę na linię: `Class Model_Pro ➥ducts` — należy w niej podać odpowiednią nazwę modelu, w naszym wypadku będzie to `Products`. Dla ścisłości podam kod modelu `clients` (listing 4.2).

Listing 4.2. *Kod modelu clients*

```
<?php

Class Model_Clients{
        function __construct($registry) {
                $this->registry = $registry;
                $this->db=$registry->db;
                $this->error=$registry->error;
```

```
                $this->text=$registry->text;
                $this->addon=$registry->addon;
        }
}

?>
```

Po utworzeniu tych dwóch plików można odświeżyć stronę kontrolera w przeglądarce. Okazuje się, iż błąd znika. Oczywiście nic się nie pojawia, gdyż nie umieściliśmy w funkcji `index` żadnego kodu.

Pozostaje nam jeszcze przećwiczyć tworzenie widoków. Aby utworzyć nowy widok, należy:

1. W katalogu: *app/views/pages/WYBRANY_JĘZYK/NAZWA_KONTROLERA/* utworzyć plik php o dowolnej nazwie, nie zapominając o zakazie używania polskich znaków diakrytycznych oraz spacji.

Jako że chcemy utworzyć nowy widok dla funkcji `index` kontrolera `shopping` w języku polskim, wejdźmy do katalogu *app/views/pages/pl/* i utwórzmy folder o nazwie *shopping*. W katalogu *shopping* utwórzmy nowy plik o nazwie *index.php*, zaś w jego zawartości wpiszmy dowolny kod HTML. Aby użyć nowo utworzonego widoku, w kodzie akcji `index` kontrolera `shopping` należy umieścić znany Ci już kod: `$this->template->show ➥('shopping/index');`. Zatem funkcja `index` będzie wyglądać następująco:

```
function index($args=null,$post=null){
        $this->template->show('shopping/index');
}
```

Proste? Nie! Banalne. Mam nadzieję, iż tych kilka prostych ćwiczeń wprowadziło Cię w arkana tworzenia oprogramowania przy użyciu struktur MVC. Jako że zaproponowany przeze mnie kontroler MVC posiada kilka funkcji ułatwiających pracę z widokami, bazami danych czy pluginami, w kolejnym rozdziale przedstawię te funkcje, lecz bez szczegółowego zagłębiania się w sposoby ich wykorzystywania. Będzie to swego rodzaju ściągawka, z której będziesz mógł korzystać przy tworzeniu naszego sklepu. Zapewne nie użyjemy wszystkich funkcji frameworka, dlatego może stanowić on dla Ciebie doskonały szkielet, który możesz dowolnie rozbudowywać i wykorzystywać we własnych projektach. W mojej *Agencji Interaktywnej — Blazing Bright* stosujemy opracowany przez nas framework, który pracuje według dokładnie tych samych zasad i jest zbudowany według tego samego wzorca. Użycie go w dziesiątkach projektów potwierdza skuteczność rozwiązania.

Funkcje frameworka

W poniższym rozdziale zapoznasz się z funkcjami dostępnymi we frameworku. Są wśród nich procedury odczytywania i zapisu danych w bazie danych, funkcje odpowiedzialne za zarządzanie widokami czy też wczytujące dodatki.

Metody template

```
$this->template->set('nazwazmiennej', wartośćzmiennej);
```

Przesyła do widoku zmienną. Używane w kontrolerze.

nazwazmiennej: String;

wartośćzmiennej: String, Integer, Array;

```
$this->template->show('sciezka/do/pliku/widoku','plik_layoutu');
```

Wyświetla widok przy użyciu wskazanego layoutu. Jeśli layout nie został zdefiniowany, użyty zostanie layout defaultowy zadeklarowany w kontrolerze. Używane w kontrolerze.

sciezka/do/pliku/widoku: String — ścieżka do pliku widoku [php], bez rozszerzenia;

plik_layoutu: String — ścieżka do pliku layoutu [php], bez rozszerzenia.

```
$this->template->popup('typ', $affected, immediate);
```

Wyświetla komunikat odpowiedniego typu o zadanej treści. Może być wyświetlony od razu bądź po przeładowaniu strony. Używany w kontrolerze admin.

typ: String ['error','success','prompt'] — typ komunikatu.

$affected: String, Array — zawartość komunikatu.

immediate: Boolean [true,false] — określa, czy komunikat wyświetla się od razu, czy po przeładowaniu strony.

```
$this->template->element('sciezka/do/elementu', zmienne);
```

Zwraca sparsowany element HTML. Używany w kontrolerze — przesyłany do widoku np. jako efekt wywołań Ajax bądź używany w widoku do wczytywania elementów strony, np. menu.

sciezka/do/elementu: String — ścieżka do pliku elementu [php], bez rozszerzenia;

zmienne: Array — tablica zmiennych przesyłanych do elementu.

```
$this->template->url('url');
```

Metoda używana w widoku do generowania linków prowadzących do akcji kontrolerów. Wszystkie linki w systemie powinny być tworzone za pomocą tej metody.

url: String — ścieżka do akcji kontrolera zbudowana z nazwy kontrolera oraz nazwy akcji w postaci: *nazwa_kontrolera/akcja*. W parametrach można przesyłać wartości zmiennych GET w postaci: *nazwa_kontrolera/akcja/zmienna1/zmienna2/zmienna3*. Zmienne odczytujemy w kontrolerze: $args[0] ('*zmienna1*') , $args[1] ('*zmienna2*') itd.

Metody db

```
$this->db->query("zapytanie SQL");
```

Wykonuje zapytanie SQL. Używane w modelu.

```
$this->db->query_and_fetchrow("zapytanie SQL");
```

Zwraca jeden wiersz wynikowy zapytania SQL. Używane w modelu.

```
$this->db->query_and_fetchall("zapytanie SQL");
```

Zwraca wszystkie wiersze wynikowe zapytania SQL. Używane w modelu.

```
$this->db->insert("zapytanie SQL", $types, $data);
```

Zapisuje wiersz w bazie danych. Używane w modelu.

> *Zapytanie SQL*: `String` — zapytanie SQL w postaci: `UPDATE gallery SET title=:title, description=:description WHERE id=:id`, gdzie *:nazwazmiennej*.
>
> `$types`: `Array` — tablica zawierająca listę przyporządkowującą zmiennym do zapisu w bazie konkretny typ danych `String` bądź `Integer`.

Przykład:

```
$types = array('title'=>'str', 'description' => 'str', 'id' => 'int');
```

> `$data`: `Array` — tablica danych do zapisu.

```
$this->db->check_values($data, $types);
```

Sprawdza, czy dane `$data` są odpowiedniego typu `$types`. Zwraca tabelę z nazwami zmiennych i wychwyconymi błędami walidacji: 1 — jeśli zmienna jest pusta, 0 — jeśli zmienna jest złego typu. Używane w modelu.

> `$data:Array` — tabela danych.
>
> `$types:Array` — tabela typów danych w postaci: *'nazwa zmiennej'=>'typ zmiennej'*. Może przyjąć następujące typy zmiennych: `str`, `int`, `email`.

Przykład:

```
$this->db->check_values(array('title'=>'Tytuł newsa', 'email'=>'xyz@aaa'),
array('title'=>'str', 'email'=>'email'));
```

Powyższy przykład zwróci błąd dla zmiennej e-mail, gdyż nie jest ona poprawnym adresem poczty e-mail. Zwrócona tablica będzie wyglądała następująco: `array('email'=>0);`.

```
$this->db->is_unique($table, $fields);
```

Sprawdza unikalność wystąpień wartości w tabeli. Zwraca `true`, jeśli wartość jest unikalna, bądź `false`, jeśli wartość już występuje w tabeli. Używane w modelu.

`$table`: `String` — nazwa tabeli.

`$fields`: `Array` — lista sprawdzanych wartości w postaci *`'pole'=>'wartość'`*.

Metody routera

```
$this->router->redirect('url');
```

Służy do przekierowywania na akcje kontrolera. Używane w kontrolerze.

`url`: `String` — ścieżka do akcji kontrolera zbudowana z nazwy kontrolera oraz nazwy akcji w postaci: *`nazwa_kontrolera/akcja`*. W parametrach można przesyłać wartości zmiennych `GET` w postaci: *`nazwa_kontrolera/akcja/zmienna1/zmienna2/zmienna3`*.

Metody addons

Aby zadeklarować dodatki używane w kontrolerze, należy je zadeklarować w sekcji, w której deklarujemy również modele:

```
$models=array('home','products','clients');
$addons = array('token','hmac_md5');
$this->addon->nazwa_klasy_dodatku->funkcja_dodatku(parametry);
```

Służy do wywołania funkcji dodatku deklarowanych w pliku *core/addons/nazwa_dodatku.php*. Używane w kontrolerze.

Założenia projektu

Naszym celem jest utworzenie funkcjonującego sklepu internetowego wraz z panelem administracyjnym. System składał się będzie z niezbędnych modułów, takich jak katalog produktów, koszyk, formularz zamówienia, moduł zarządzania produktami czy moduł zarządzania zamówieniami. Są to elementy niezbędne do działania sklepu, jednak stanowią minimum i w ramach ćwiczeń polecam Ci, Drogi Czytelniku, rozbudować system o dodatkowe moduły, takie jak panel klienta wraz z listą zamówień i statusem zamówienia czy formularze kontraktowe bądź czat ze sprzedawcami. Taki sklep będzie swoistym RIA, czyli *Rich Internet Application*. Sam sklep, czyli front-end, wykonamy we Flashu, natomiast panel administracyjny będzie wykonany w HTML. Rozważmy zatem ogólne założenia projektu.

Ogólne założenia

Sklep

1. Wykonany w aplikacji Flash.
2. Katalog produktów wyświetlający produkty z podziałem na kategorie.

3. Indywidualna strona produktu z galerią zdjęć, opisem oraz ceną.
4. Koszyk zamówień z możliwością dodawania oraz usuwania produktów.
5. Formularz zamówienia do wprowadzania danych i finalizacja zamówienia.

Panel administracyjny

1. Moduł zarządzania produktami z możliwością dodawania i usuwania produktów oraz edycją produktów i ich kategorii.
2. Przegląd zamówień z możliwością zmiany ich statusu.

Projekt bazy danych

Nie pozostaje nam nic innego jak przystąpienie do prac projektowych. Zacznijmy od bazy danych. Zakładając, iż przechowywać w niej będziemy informacje o produktach oraz zamówieniach, na pewno składać się będzie z dwóch tabel. Po głębszym zastanowieniu można przyjąć, iż jedna tabela, o nazwie *products*, przechowywać będzie takie informacje jak: niepowtarzalny identyfikator produktu, jego nazwa, opis, cena oraz ilość dostępnych sztuk. Założyliśmy jednak, iż każdy produkt przyporządkowany będzie do pewnej kategorii produktów, dlatego będziemy musieli zapisać również identyfikator kategorii, który odnosił się będzie do tabeli kategorii. Zatem tabela przechowująca informacje o produktach prezentować będzie się następująco (rysunek 4.3):

Rysunek 4.3.
Tabela products

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
category	int(11)			Nie	*Brak*	
name	varchar(255)	utf8_general_ci		Nie	*Brak*	
description	text	utf8_general_ci		Nie	*Brak*	
price	decimal(15,2)			Nie	*Brak*	
quantity	int(11)			Nie	*Brak*	

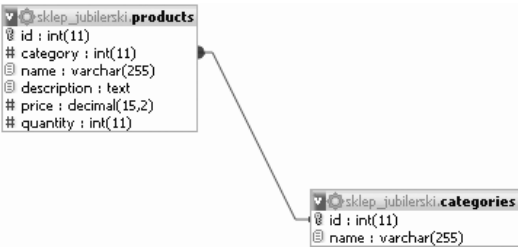
Wspominałem o tabeli kategorii. Zastanówmy się zatem, jak powinna ona wyglądać. Na pewno będzie posiadać unikalny identyfikator, który używany jest w tabeli produktów. Powinna też posiadać pole nazwy. Nie będziemy zgłębiać zagadnienia zagnieżdżania kategorii, jednak podpowiem, iż można to rozwiązać, dodając pole *parent*, które przechowywać będzie *id* nadrzędnej kategorii. Następnie budowanie drzewka kategorii wymagać będzie zastosowania rekurencyjnych wywołań. Na szczęście ActionScript obsługuje rekurencję, więc w ramach ćwiczeń po ukończeniu programu proponuję Ci modyfikację kodu, tak aby obsługiwał podkategorie. Tabela *categories* będzie więc wyglądać następująco (rysunek 4.4):

Rysunek 4.4.
Tabela categories

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
name	varchar(255)	utf8_general_ci		Nie	*Brak*	

Relacje pomiędzy tabelami prezentować będą się następująco (rysunek 4.5):

Rysunek 4.5.
Relacje pomiędzy tabelami products i categories



Pozostaje nam jeszcze rozważyć budowę tabeli przechowującej zamówienia. W takiej tabeli na pewno trzeba będzie zapisać dane klienta, datę zamówienia, sposób płatności (udostępnimy wyłącznie płatność przy odbiorze bądź przelew bankowy) oraz listę zamówionych produktów wraz z ilością. W jednym polu nie zapiszemy listy produktów (chyba że oddzielimy je przecinkiem, ale to śmieszne rozwiązanie). Dlatego też będziemy musieli utworzyć dwie tabele — jedna przechowywać będzie informacje o kliencie oraz szczegółach zamówienia bez listy zamówionych produktów, natomiast w drugiej zapisywać będziemy zamawiane produkty wraz z aktualną ceną jednostkową, ilością oraz identyfikatorem zamówienia, tak aby jednoznacznie przyporządkować je do konkretnego zamówienia. Tabela zamówień wyglądać będzie jak na rysunku 4.6.

Rysunek 4.6.
Tabela orders

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
name	varchar(255)	utf8_general_ci		Nie	*Brak*	
address	varchar(255)	utf8_general_ci		Nie	*Brak*	
email	varchar(255)	utf8_general_ci		Nie	*Brak*	
phone	varchar(32)	utf8_general_ci		Nie	*Brak*	
date	int(11)			Nie	*Brak*	
payment_type	tinyint(4)			Nie	*Brak*	
status	tinyint(4)			Nie	*Brak*	

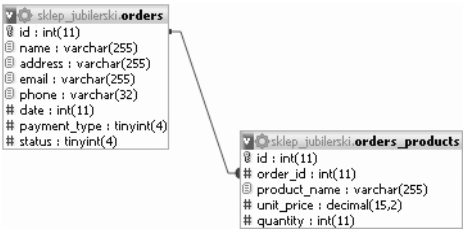
Natomiast tabela z listą zamówionych produktów wyglądać będzie jak na rysunku 4.7.

Rysunek 4.7.
Tabela orders_products

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
order_id	int(11)			Nie	*Brak*	
product_name	varchar(255)	utf8_general_ci		Nie	*Brak*	
unit_price	decimal(15,2)			Nie	*Brak*	
quantity	int(11)			Nie	*Brak*	

Relacje pomiędzy tymi dwoma tabelami będą prezentować się następująco (rysunek 4.8):

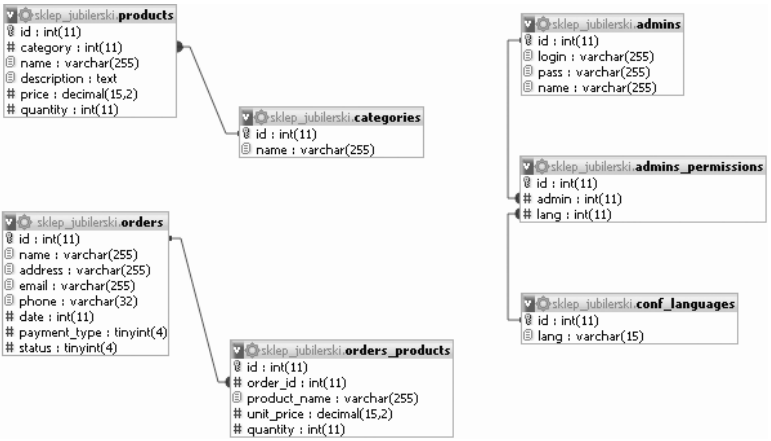
Rysunek 4.8.
Relacje pomiędzy tabelami orders i orders_products



Łatwo się domyślić, do czego służy pole *product_name* w tabeli *orders_products*. Przechowywać będzie ono nazwy produktów. Nie będziemy kojarzyć tabeli *orders_products*

z tabelą *products*, aby uniknąć niebezpieczeństwa utraty integralności w wypadku usunięcia produktu z bazy. Nie będziemy również stosować osobnych tabel do przechowywania usuniętych produktów na potrzeby zamówień, aczkolwiek takie rozwiązanie byłoby bardziej prawidłowe. Diagram relacji tabel możesz zobaczyć na rysunku 4.9.

Rysunek 4.9.
Relacje wszystkich tabel



Są to wszystkie tabele, jakich będziemy potrzebować do obsługi sklepu. Jednakże biorąc pod uwagę fakt, iż będziemy również budować panel administracyjny, na rysunku zamieszczono też tabele *admins*, *admins_permissions* oraz tabelę, w której framework zapisuje dostępne wersje językowe: *conf_languages*. Musimy gdzieś przechowywać login i hasło administratorów. Dlatego też tworzymy tabelę składającą się z identyfikatora, loginu, odcisku palca stworzonego na podstawie hasła, czyli *hasha md5*, oraz nazwy administratora (rysunek 4.10).

Rysunek 4.10.
Tabela admins

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
login	varchar(255)	utf8_general_ci		Nie	*Brak*	
pass	varchar(255)	utf8_general_ci		Nie	*Brak*	
name	varchar(255)	utf8_general_ci		Nie	*Brak*	

Tworzymy też tabelę, która przechowa sygnatury dostępnych wersji językowych (rysunek 4.11). Mimo że tworzymy stronę tylko w jednej wersji, to system potrzebuje tej tabeli do poprawnego działania, nawet jeśli będzie w niej tylko jeden wpis.

Rysunek 4.11.
Tabela conf_languages

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
lang	varchar(15)	utf8_general_ci		Nie	*Brak*	

Trzecia tabela, *admins_permissions*, zawiera informacje określające, do edycji których wersji językowych mają dostęp poszczególni administratorzy (rysunek 4.12).

Rysunek 4.12.
Tabela admins_permissions

Pole	Typ	Metoda porównywania napisów	Atrybuty	Null	Domyślnie	Dodatkowo
id	int(11)			Nie	*Brak*	auto_increment
admin	int(11)			Nie	*Brak*	
lang	int(11)			Nie	*Brak*	

Na koniec rzuć okiem na wygląd naszej bazy danych (rysunek 4.13).

Rysunek 4.13. *Wszystkie tabele projektu*

Tabela	Działanie	Rekordy[1]	Typ	Metoda porównywania napisów
admins		0	MyISAM	utf8_general_ci
admins_permissions		0	MyISAM	utf8_general_ci
categories		0	MyISAM	utf8_general_ci
conf_languages		0	MyISAM	utf8_general_ci
orders		0	MyISAM	utf8_general_ci
orders_products		0	MyISAM	utf8_general_ci
products		0	MyISAM	utf8_general_ci

Projekt kontrolerów, modeli oraz widoków

Pójdźmy zatem o krok dalej. Zastanówmy się, jakie usługi będziemy świadczyć użytkownikowi, jakich operacji na danych będziemy dokonywać. Postarajmy się zaprojektować hierarchię kontrolerów oraz modeli. Jeśli chodzi o widoki, będą to raczej poszczególne pliki *swf*, gdyż na nich właśnie będzie opierał się nasz interfejs, przynajmniej ten front-endowy. Podstawową usługą będzie katalog produktów z podziałem na kategorie. Będziemy zatem potrzebować kontrolera o nazwie `catalog`. Przyjmijmy, iż podstawową akcją kontrolera *catalog* będzie `index`. Po jej uruchomieniu naszym oczom powinna ukazać się lista kategorii oraz kilka losowo wybranych produktów (nie wybraliśmy przecież jeszcze żadnej kategorii). Zatem funkcja `index` powinna mieć możliwość wczytywania listy kategorii oraz kilku losowych produktów. Po wybraniu dowolnej kategorii użytkownik powinien zobaczyć listę produktów. Powinniśmy zatem przygotować funkcję o nazwie `category`, która w swych parametrach przyjmować będzie identyfikator kategorii i zwróci listę produktów w niej zawartych. Po kliknięciu wybranego produktu powinien wyświetlić się kompletny opis produktu, jednak jak zobaczysz później, obejdzie się przy tym bez kolejnych zapytań do bazy danych. Zatem reasumując, kontroler `catalog` powinien wyglądać następująco:

Kontroler `Catalog`

- `index` — nie przyjmuje parametrów, wyświetla stronę główną sklepu;
- `categories` — zwraca listę kategorii;
- `randomProducts` — zwraca kilka losowo wybranych produktów;
- `category` — jako parametr przyjmuje identyfikator kategorii, zwraca listę produktów z danej kategorii.

Model o tej samej nazwie powinien posiadać następujące funkcje:

Model `Catalog`

- `get_all_categories` — wczytuje listę wszystkich kategorii;
- `get_random_products` — wczytuje listę kilku przykładowych produktów;
- `get_products_by_category` — wczytuje produkty z danej kategorii.

Podstawową opcją katalogu będzie dodawanie produktów do koszyka, dlatego też musimy utworzyć kontroler odpowiedzialny za jego obsługę. Nazwijmy go `shop`. Oprócz koszyka kontroler ten będzie obsługiwał proces składania zamówienia. Kontroler powi-

nien mieć możliwość wyświetlenia zawartości koszyka, dodawania, jak też i usuwania produktów, obsługi formularza zamówienia oraz przesyłania zamówienia do realizacji. Przyjmijmy, iż kontroler będzie składał się z następujących funkcji:

Kontroler `Shop`

- `cartContent` — zwraca zawartość koszyka;
- `addToCart` — dodaje określoną liczbę sztuk produktu do koszyka;
- `removeFromCart` — usuwa określony przedmiot z koszyka;
- `clearCart` — usuwa z koszyka wszystkie produkty;
- `sendOrder` — wysyła zamówienie, czyli zapisuje je w bazie danych.

Zastanówmy się, gdzie będziemy przechowywać informacje związane z zawartością koszyka. Proponuję użyć do tego sesji. Będziemy w nich zapisywać, jakie produkty oraz w jakiej ilości zostały do koszyka dodane. Zapis oraz odczyt sesji będzie realizowany w funkcjach modelu `shop`. Postarajmy się określić, jakie funkcje będą nam potrzebne:

Model `Shop`

- `get_cart_content` — odczytuje zapisaną w sesji zawartość koszyka;
- `add_to_cart` — zapisuje do sesji kolejno dodawane do koszyka produkty;
- `remove_from_cart` — usuwa z sesji informacje o produkcie;
- `clear_cart` — usuwa dane sesji koszyka;
- `save_checkout_data` — zapisuje w bazie danych szczegóły zamówienia.

Powinny to być wszystkie funkcje, jakich będziemy potrzebować do obsługi front-endu sklepu. Teraz trzeba się zastanowić, co będzie niezbędne do obsługi sklepu przez administrację.

Zakładamy, iż będziemy mieli możliwość wyświetlania listy produktów w sprzedaży, ich dodawania, edycji oraz usuwania. Musimy też udostępnić funkcje obsługi kategorii. Oraz oczywiście najistotniejszą część panelu administracji, czyli przegląd zamówień. Potrzebna będzie lista zamówień z możliwością wyświetlania danych klienta oraz zmiany statusu zamówienia. Utworzymy zatem kontroler o nazwie `admin`, natomiast do operacji na danych użyjemy modeli `catalog` oraz `shop`. Jakie funkcje utworzymy w kontrolerze `admin`?

Kontroler `admin`

- `login` — realizacja logowania;
- `logout` — wylogowanie;
- `settings` — zmiana hasła dostępu;
- `products` — wyświetlanie listy produktów;
- `products_add` — dodawanie nowego produktu;

- `products_edit` — edycja produktu;
- `products_remove` — usuwanie produktu;
- `products_categories` — zarządzanie kategoriami;
- `products_categories_add_new` — dodawanie nowej kategorii produktów;
- `products_categories_edit` — edycja kategorii;
- `products_categories_remove` — usuwanie kategorii;
- `orders` — zarządzanie zamówieniami;
- `orders_change_status` — zmiana statusu zamówienia;
- `orders_remove` — usuwanie zamówienia.

Utworzymy również model `admin`, który będzie realizował wyłącznie funkcje związane z logowaniem oraz zmianą hasła.

Model `admin`

- `login` — funkcja realizująca logowanie;
- `change_password` — funkcja realizująca zmianę hasła.

Jak już wspomniałem wcześniej, do operacji na danych użyjemy wcześniej utworzonych modeli `catalog` oraz `shop`. Dodamy do nich odpowiednie funkcje, które będą odpowiadać za zapisywanie do bazy danych nowych produktów czy kategorii.

Model `catalog`

- `ADMIN_get_all_products` — wczytuje wszystkie produkty;
- `ADMIN_save_new_product` — zapisuje nowy produkt w bazie danych;
- `ADMIN_edit_product` — wczytuje dane produktu do edycji;
- `ADMIN_update_product` — zapisuje zmienione dane produktu do bazy danych;
- `ADMIN_remove_product` — usuwa produkt;
- `ADMIN_upload_image` — obsługuje wgrywanie zdjęć produktów;
- `ADMIN_save_new_category` — zapisuje nową kategorię;
- `ADMIN_update_category` — zapisuje zmienioną kategorię;
- `ADMIN_remove_category` — usuwa kategorię.

Model `shop`

- `ADMIN_get_all_orders` — wczytuje wszystkie zamówienia;
- `ADMIN_change_order_status` — zmienia status zamówienia oraz powiadamia o tym klienta poprzez wysłanie e-maila;
- `ADMIN_remove_order` — usuwa zamówienie.

Określiliśmy zatem kluczowe dla działania systemu funkcje. Stanowią one niejako szkielet i są niezbędne do sprawnego działania całości. Oczywiście nie jesteśmy w stanie przewidzieć wszystkich możliwości i scenariuszy — jest to umiejętność budowana na doświadczeniu. Jednak jak doskonale wiesz, chociażby na przykładzie gigantów w zakresie oprogramowania, doświadczenie nie zawsze idzie w parze z umiejętnościami przewidywania, co użytkownik jest w stanie zrobić z programem oraz jakie błędy mogą wystąpić. Dlatego też sporządzony przez nas projekt należy traktować elastycznie i sukcesywnie modyfikować go w procesie deweloperskim, co też będziemy czynić.

W kwestii widoków nasza sytuacja jest nieco inna niż w wypadku tworzenia systemu wyłącznie w HTML. Do obsługi interfejsu front-endu będziemy używać aplikacji Flash. Zakładamy, iż będzie miał on budowę modułową i składał się będzie z co najmniej czterech modułów — jednego do wyświetlania katalogu produktów, drugiego do wyświetlania szczegółów produktu, trzeciego do obsługi koszyka oraz czwartego do obsługi formularzy zamówienia. Dobrze będzie też zbudować program nadrzędny, czyli ten, który ładuje się jako pierwszy. Będzie on stanowił swoisty layout dla reszty modułów.

Widoki interfejsu — front-end

- Layout — kontener
  - Moduł katalogu
  - Moduł szczegółów produktu
  - Moduł koszyka
  - Moduł składania zamówienia

Jako że Flash nie potrzebuje do działania przeładowań stron, nie będziemy musieli tworzyć dziesiątek pojedynczych stron HTML.

Zgoła inaczej wyglądać to będzie w panelu administracyjnym. Tam główną rolę odgrywać będzie HTML. Dlatego będziemy tworzyć strony z tabelami produktów, formularze do dodawania oraz edycji produktów oraz kategorii itp. Z góry możemy przewidzieć, iż stworzymy kilka podstron:

Widoki interfejsu panelu administracyjnego

- `login` — strona logowania,
- `index` — strona powitalna panelu,
- `logout` — strona pożegnalna panelu,
- `settings` — strona zmiany hasła,
- `orders` — strona z listą zamówień,
- `products` — strona z listą produktów,
- `add_product` — formularz dodawania nowego produktu,

- `edit_product` — formularz edycji produktu,
- `categories` — strona z listą kategorii (na niej będzie przeprowadzane również dodawanie, edycja i usuwanie kategorii.

To wszystko, jeśli chodzi o szczegóły budowy szkieletu systemu. Mamy już poglądowy obraz tego, jak będzie wyglądało i funkcjonowało to oprogramowanie.

Bezpieczeństwo

Już w fazie projektu warto zwrócić uwagę na bezpieczeństwo tworzonego oprogramowania. Obowiązkiem programisty jest przewidzieć i odpowiednio zabezpieczyć oprogramowanie przed skutkami niepożądanych działań i wypadków. W przypadku tworzenia sklepu internetowego, gdzie w bazie danych przechowywane są dane osobowe jego klientów, należy zachować należytą staranność i za główny cel przyjąć ochronę tych danych. Dlatego też bardzo istotną kwestią jest dostęp do baz danych. Mimo że główna konfiguracja serwera baz danych leży po stronie administracji serwera, to już na poziomie użytkownika domeny jesteśmy w stanie powziąć pewne kroki zmierzające ku lepszemu zabezpieczeniu danych naszych klientów. Dlatego bardzo istotne jest, aby zwrócić uwagę na takie prozaiczne ustawienia PHP, jak `register_globals`. Dzięki niemu wszystkie zmienne przesyłane do skryptu tworzone są jako globalne, co w przypadku nienależytej staranności przy tworzeniu skryptów umożliwi crackerom wykonywanie ataków typu SQLInjection, czyli manipulację zapytaniami SQL, oszukiwanie systemów logowania do panelu administracyjnego itp. Dlatego bardzo istotne jest, aby parametr ten ustawiony był na 0. Uczynimy to chociażby w pliku konfiguracyjnym serwera *httpd.conf*:

```
php_admin_value register_globals 0
```

Bądź też w pliku *.htaccess*:

```
php_flag register_globals 0
```

Drugą kwestią jest filtrowanie wszystkich danych wprowadzanych przez użytkownika. Nie można zakładać, iż użytkownik wprowadzi do formularzy oczekiwane przez nas dane. Zresztą formularze najczęściej poddawane są walidacji, natomiast nie można spodziewać się oczekiwanych wartości w zmiennych, które z reguły powinny być generowane przez system, chociażby parametrów przesyłanych w adresie metodą `GET`. Stare greckie porzekadło mówi, że gdy spodziewasz się w parametrze otrzymać identyfikator w postaci liczbowej, najpewniej otrzymasz 256 losowych znaków w cyrylicy (oczywiście żart, lecz prawdziwy). Dlatego istotne jest przeprowadzanie filtracji wszystkiego, co wychodzi od użytkownika, oraz wszystkiego, na co teoretycznie nie powinien mieć wpływu. Pamiętaj więc o używaniu funkcji: `htmlspecialchars` bądź `strip_tags`, `trim` do usuwania znaków niedrukowalnych, `addslashes`, `mysql_real_escape_string`. Oprogramowanie powinno być „idiotoodporne”, dlatego warto przeprowadzać niezbędne testy przy pomocy zaprzyjaźnionych internautów :-). Testy takie pozwolą wykryć więcej luk, niż mógłbyś sobie wyobrazić, dlatego są tak istotnym punktem procesu deweloperskiego.

Jeśli chodzi o bezpieczeństwo skryptów ActionScript, to jesteśmy o tyle w komfortowej sytuacji, iż nasze skrypty uruchamiane są w obrębie jednej domeny, dlatego nie musimy przejmować się dyrektywami `Security.allowDomain()`. Jednak i w tym wypadku warto przewidzieć niepożądane zachowania użytkownika, takie jak wielokrotne klikanie linków z uporem maniaka. Bardzo wartościową lekturą traktującą o zabezpieczeniach w programie Flash Player jest rozdział manuala wydanego przez Adobe, znajdujący się pod adresem: *http://help.adobe.com/pl_PL/ActionScript/3.0_Programming AS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7d23.html*.