UNIWERSYTET IM. ADAMA MICKIEWICZA
Wydział Fizyki

Zend Framework

Podstawowe zastosowanie w tworzeniu prostej aplikacji opartej o bazę danych

Prowadzący: **dr Wojciech Czart**

Autor: **Grzegorz Michalak**

Spis Treści:

1.1. Czym jest framework?

Jak podaje Wikipedia, framework to szkielet działania aplikacji, który zapewnia podstawowe mechanizmy i może być wypełniany właściwą treścią programu. Kiedy siadasz do nowego projektu, zaczynasz wszystko od nowa - tworzysz strukturę katalogów, konfiguracje, piszesz kod, który przetwarza żądania. Często kopiujesz fragmenty kodu, albo całe moduły, bo przecież ile razy można pisać to samo? I dlatego powstały frameworki!

Jest dostępnych wiele frameworków dla różnych języków programowania. Dla PHP należy wspomnieć o oficjalnym - Zend Framework oraz o innych znanych - CakePHP, Code Igniter, Symfony. Warto zauważyć, że pomimo dostępności różnorodnych frameworków, programiści często decydują się na napisanie własnego.

Frameworki udostępniają pełny mechanizm rozruchowy dla aplikacji. Dzięki temu programista może skupić się tylko na właściwym kodzie modułu. Oprócz tego frameworki posiadają biblioteki do obsługi sesji, baz danych, kontrolę dostępu i szereg innych narzędzi do łatwego i szybkiego tworzenie oprogramowania.

Zdecydowana większość frameworków opiera się na budowie MVC (model-view-controller). Wyjątkiem jest tu Prado, który bazuje na zdarzeniach. Dobre zrozumienie MVC jest podstawą do dalszej nauki.

1.2. Architektura MVC

Model-View-Controller (model-widok-kontroler) jest wzorcem projektowym stosowanym nie tylko w aplikacjach webowych. Mówiąc najprościej, wzorzec to przepis na zaprojektowanie aplikacji w pewien sprawdzony sposób.

W MVC aplikacja podzielona jest na 3 części: model, widok i kontroler. Poszczególne części aplikacji są obiektami współpracującymi ze sobą.

Model reprezentuje źródło danych. Najczęściej jest to baza danych, ale równie dobrze mogą to być pliki tekstowe, XML, czy nawet zasoby położone na innym serwerze. Ważne jest, aby model w MVC ukrywał przed aplikacją na czym pracuje, udostępniając jej jedynie interfejs do zarządzania danymi.

Widok jest odpowiedzialny za prezentację. W aplikacjach webowych najczęściej jest to XHTML. Widok przypomina trochę system szablonów (i często jest z nim mylony). Widok otrzymuje dane, ubiera je w szatę graficzną (jeżeli to XHTML) i wyświetla.

Kontroler zarządza procesem wymiany danych między widokiem a modelem. Jest to właściwy kod, jaki tworzy programista. W praktyce kontroler przyjmuje dane żądania, pobiera dane od modelu i przekazuje je widokowi.

Kontroler łączy pozostałe warstwy w spójną całość. W zasadzie jest to jedyne miejsce, gdzie powinien nastąpić kontakt widoku z modelem.

Zend Framework wykorzystuje architekturę Model-View-Controller (MVC), by oddzielić różne części Twojej aplikacji na poszczególne segmenty, dzięki czemu pisanie i późniejsze zarządzanie (rozbudowywanie) aplikacji staje się o wiele prostsze.

2. Zend Framework

2.1. Krótka historia

Po ponad roku prac, wersjach alfa, beta, dwóch Release Candidate, pojawiła się pierwsza stabilna wersja, firmowanego i tworzonego przez Zend Technologies, w pełni obiektowego Frameworka dla PHP5 – Zend Framework.

2.2. Funkcjonalność

Środowisko zostało oparte na wzorcu projektowym MVC (Model-View-Controller) i powala swoją funkcjonalnością, w której znajdziemy takie moduły jak:

- autentykacja i autoryzacja
- listy kontroli dostępów
- obsługa baz danych
- obsługa cache
- filtrowanie i walidacja wprowadzonych danych
- obsługa kanałów RSS i Atom
- obsługa daty i lokalizacji
- obsługa PDF, protokołu HTTP i SMTP, Web Services
- wsparcie dla L10n oraz I18n
- przyjazne URL
- obsługa usług takich serwisów jak: Yahoo, Google, Delicious, Flickr, Amazon, Akismet
- i wiele wiele innych…

Po efektach widać, że programiści Zenda przyłożyli się do pracy. Opracowano przejrzystą strukturę katalogów, umożliwiających programistom tworzenie własnych klas i pluginów nie ingerując tym sam w serce biblioteki.

Niewątpliwym atutem Zend Framework, jest dość obszerna dokumentacja, dostępna także w języku polskim (na razie tylko niektóre moduły).

2.3. Główne moduły

- **Zend_Controller oraz Zend_Viev**
  Komponenty zapewniające solidny fundament dla prostego MVC, można ich używać razem z szablonami. Pozwalają zaoszczędzić sporo czasu i zmniejszyć ilość kodu potrzebnego do kontroli strony.
- **Zend_db**
  Dostęp do bazy danych używający PDO. Zend_Dp umożliwia napisanie prostych obiektów obsługujących wiersze.

- **Zend_Feed**
  Pozwala na przetwarzanie kanałów RSS oraz Atom.
- **Zend_InputFilter**
  Komponent odpowiedzialny za filtrowanie danych od użytkownika.
- **Zend_Json**
  Konwertowanie struktur PHP na Json, po to aby następnie użyć ich w Ajaxie.
- **Zend_Log**
  Wspaniałe narzędzie w połączeniu z Cronem, a także do zapisywania błędów.
- **Zend_Mail i Zend_Mime**
  Umożliwia tworzenie i wysyłania wiadomości email. Obsługuje załączniki.
- **Zend_Pdf**
  Pozwala na tworzenie dokumentów pdf "w locie".
- **Zend_Search_Lucene**
- **Zend_Service: Amazon, Flickr, and Yahoo!**
- **Zend_XmlRpc**

2.4. Wymagania

Wymagania Zend Framework są następujące:

- PHP 5.1.4 (lub nowszy)
- Serwer www wspierający funkcionalnośc “mod_rewrite” (w założeniu Apache)

2.5. Pobieranie Zend Framework

Zend Framework może być pobrany z strony domowej projektu:
http://framework.zend.com/download w formacie .zip lub .tar.gz

2.6. Struktura Katalogów

Mimo tego, że Zend Framework nie narzuca struktury katalogów dla aplikacji, dokumentacja Framework’a sugeruje nam korzystanie z uwspólnionego modelu struktury katalogów. Typ ten zakłada, że posiadamy pełną kontrolę nad serwerem Apache, jednakże na potrzeby tego opisu, użyjemy pewnej modyfikacji, by ułatwić sobie pracę.

Rozpoczynamy poprzez stworzenie katalogu w głównym katalogu serwera (root directory) i nazywamy go ‘zf-tutorial’. Dzięki temu adres URL do naszej aplikacji będzie następujący: http://localhost/zf-tutorial.

Tworzymy następującą strukturę katalogów, która będzie przetrzymywała pliki naszej aplikacji:

```
zf-tutorial/
  /application
    /controllers
    /models
    /views
      /filters
      /helpers
      /scripts
  /library
```

```
/public
  /images
  /scripts
  /styles
```

Rozdzieliliśmy dla naszej aplikacji katalogi dla Modelu, Widoku i Kontrolera. Pliki dodatkowe tj. Obrazki, skrypty JavaScript oraz arkusze stylów CSS są przechowywane w całkiem oddzielnej od aplikacji gałęzi katalogów: public. Ściągnięte pliki bibliotek Zend Framework będą umieszczone w katalogu bibliotek 'library'. Jeśli będziemy potrzebowali użyć jakichkolwiek dodatkowych bibliotek, powinny one również znaleźć się w tym katalogu.

Rozpakowujemy ściągnięte archiwum, ZendFramework-1.0.3.zip w moim przypadku, do tymczasowego katalogu. Powinniśmy otrzymać podkatalog ZendFramework-1.0.3, który w sobie zawiera kolejną strukturę podkatalogów i plików. Następnie kopiujemy zawartość podkatalogu library/Zend/ do katalogu zf-tutorial/library/. Po tej operacji katalog zf-tutorial/library/ powinien zawierać pod-katalog nazwany Zend.

3. Nasza aplikacja

3.1.1. Bootstrapper

Główny kontroler Zend Framework'a, Zend_Controller jest zaprojektowany by obsługiwać strony internetowe w oparciu o "czyste URL'e". Aby to osiągnąć, wszystkie żądania HTTP (wywołania stron) muszą przechodzić przez jeden plik : index.php, zwany w tym kontekście jako "bootstrapper". Takie działanie zapewnia nam centralny punkt startu dla wszystkich naszych generowanych stron i upewnia nas, że środowisko aplikacji jest zawsze skonfigurowane w ten sam sposób. Osiągamy ten efekt poprzez stworzenie pliku .htaccess w głównym katalogu naszej aplikacji (application root directory), czyli zf-tutorial/.

plik zf-tutorial/.htaccess

```
RewriteEngine on
RewriteRule .* index.php

php_flag magic_quotes_gpc off
php_flag register_globals off
```

Ta reguła RewriteRule jest bardzo prosta i może być interpretowana jako: "dla każdego wywołania URL użyj pliku index.php w zamian".

Poprzez zdrowy rozsądek i dla bezpieczeństwa naszej aplikacji, ustawiliśmy także kilka ustawień PHP ini. Prawdopodobnie już są one prawidłowo ustawione w głównym pliku ustawień PHP (php.ini), jednak chcemy mieć pewność, że tak jest.

Jednakże musimy nanieść poprawkę na nasze reguły przekierowań, ponieważ żądania obrazków, skryptów Java Script lub plików CSS nie powinny być nakierowywane na naszego "bootstrapper'a". Dzięki temu, że trzymamy wszystkie tego typu pliki w publicznym katalogu, możemy "powiedzieć" serwerowi Apache, by traktował te pliki w specjalny sposób, poprzez stworzenie kolejnego pliku .htaccess tworząc go w katalogu zf-tutorial/public/:

plik zf-tutorial/public/.htaccess

```
RewriteEngine off
```

Mimo że nie jest to całkowicie potrzebne dla tego przykładu, możemy dodać jeszcze kolejne pliki .htaccess aby uchronić pliki naszej aplikacji oraz biblioteki przed niepowołanym dostępem z zewnątrz. Stwórzmy więc następujące dwa pliki:

plik zf-tutorial/application/.htaccess

```
deny from all
```

plik zf-tutorial/library/.htaccess

```
deny from all
```

Aby poprawnie używać w/w plików .htaccess, konfiguracja serwera Apache (plik httpd.conf) musi mieć ustawioną poprawnie dyrektywę “AllowOverride” na wartość “All“.

3.1.2. Plik index.php

Skoro zf-tutorial/index.php jest naszym centralnym plikiem “bootstrapper’a”, rozpoczniemy od następującego kodu:

zf-tutorial/index.php

```
<?php
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/London');
set_include_path('.' . PATH_SEPARATOR . './library'
   . PATH_SEPARATOR . './application/models/'
   . PATH_SEPARATOR . get_include_path());

include "Zend/Loader.php";
Zend_Loader::loadClass('Zend_Controller_Front');

// setup controller
$frontController = Zend_Controller_Front::getInstance();
$frontController->throwExceptions(true);
$frontController->setControllerDirectory('./application/controllers');

// run!
$frontController->dispatch();
```

Zauważ, że nie zamykamy sekcji kodu PHP poprzez znacznik ?> na końcu pliku. Jest to działanie celowe, ponieważ nie jest on wymagany, a nie umieszczenie go może być dla nas pożyteczne w przypadku trudnych do znalezienia błędów przekierowań. Błędy te powstają podczas przekierowywania stron (redirecting) funkcją header(), gdy gdzieś w kodzie pojawia się dodatkowa spacja (white space) za znacznikiem ?>

Prześledźmy teraz powyższy plik.

```
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/London');
```

Te linie kodu upewniają nas, że wszystkie powstałe podczas uruchamiania programu błędy będą wyświetlane na ekran (przy czym zakładam, że w konfiguracji PHP'a dyrektywa "display_errors" jest ustawiona na "On"). Ustawiamy także strefę czasową, co jest wymogiem specyfikacji PHP v.5.1+ względem każdej aplikacji.

```
set_include_path('.' . PATH_SEPARATOR . './library'
  . PATH_SEPARATOR . './application/models/'
  . PATH_SEPARATOR . get_include_path());

include "Zend/Loader.php";
```

Zend Framework jest zaprojektowany tak, że ścieżka do jego bibliotek musi się znajdować w "include path", aby biblioteki te były poprawnie odnajdywane przez PHP. W ten sam sposób podajemy także ścieżkę do naszych plików Modelu, byśmy mogli później je łatwo wywoływać. Aby wystartować, musimy także zainkludować plik "ładowarki bibliotek" Zend_Loader, która pozwoli nam ładować później wszystkie inne klasy Zend Framework wedle naszych potrzeb.

```
  Zend_Loader::loadClass('Zend_Controller_Front');
```

Zend_Loader::loadClass ładuje klasę z odpowiedniego pliku, według podanej jako parametr nazwy klasy. Dzieje się to poprzez konwersję znaków podkreślenia (underscore) na separatory ścieżki (np: "/") i dodanie przyrostka ".php" na końcu. W takim razie Zend_Controller_Front będzie odczytany z lokalizacji: Zend/Controller/Front.php. Jeśli wykorzystamy tą samą metodę do nazewnictwa własnych bibliotek/klas, będziemy mogli także wykorzystać Zend_Loader::loadClass() by je ładować. Powyższa klasa to pierwsza jaką potrzebujemy – kontroler frontowy.

Kontroler frontowy wykorzystuje klasę Routera by żądaniom URL przypisywać odpowiednie funkcje PHP, które z kolei będą wykorzystane do generowania właściwej strony. Aby Router mógł wykonywać swoją pracę, musi on dostać informację, która część żądania URL jest wskazaniem na nasz plik index.php, tak by resztę adresu mógł odpowiednio wykorzystać. Tym zadaniem zajmuje się obiekt żądania (Request object). Wykonuje on świetną pracę w automatycznym określaniu bazowego adresu ULR (base URL), lecz jeśli nie uda mu się to, np. ze względu na nietypową konfigurację naszego środowiska, możemy mu w tym pomóc poprzez manualne nadpisanie wartości adresu bazowego funkcją: $frontController->setBaseUrl().

Musimy także skonfigurować kontroler frontowy, tak by wiedział gdzie szukać naszych kontrolerów:

```
$frontController = Zend_Controller_Front::getInstance();
$frontController->setControllerDirectory('./application/controllers');
$frontController->throwExceptions(true);
```

Jako że jest to aplikacja, która będzie działała na środowisku testowym, zdecydowałem ustawić kontroler frontowy by "wyrzucał" wszystkie "wyjątki" (throw exceptions) jakie

ewentualnie wystąpią. Domyślnie jest tak, że kontroler frontowy “wyłapuje” wszystkie obiekty wyjątków i przetrzymuje je w własności “_exceptions” obiektu “Response“. Obiekt odpowiedzi (Response object) przechowuje wszystkie informacje na temat odpowiedzi na zadane żądanie URL. Zawierają się w nim nagłówki strony (Headers), treść strony (Body content) i wspomniane obiekty wyjątków. Kontroler frontowy zajmuje się także automatycznym wysłaniem nagłówków i wyświetleniem treści strony tuż przed zakończeniem jego pracy.

Wszystko to może wydawać się zawiłe dla osoby po raz pierwszy korzystającej z Zend Framework. Z tego też względu “wyrzucamy” na ekran wszystkie wyjątki, by można było obserwować ewentualne błędy. Oczywiście na serwerze produkcyjnym ,takie rozwiązania nie powinny mieć miejsca.

Ostatecznie dochodzimy do sedna sprawy i uruchamiamy naszą aplikację:

```
// run!
$frontController->dispatch();
```

Jeśli w tym momencie w przeglądarce uruchomimy następujący adres:
http://localhost/zf-tutorial/ dla testu, powinieneś zobaczyć następujący błąd:

```
Fatal error: Uncaught exception 'Zend_Controller_Dispatcher_Exception'
   with message 'Invalid controller specified (index)' in...
```

Oznacza to dla nas, że nie stworzyliśmy jeszcze naszej aplikacji. Zanim jednak się tym zajmiemy, omówmy sobie co tak na prawdę chcemy zbudować.

3.2. Strona internetowa

Naszym celem będzie zbudowanie prostego programu do inwentarzu płyt CD. Strona główna będzie wyświetlać listę naszej kolekcji i pozwalać nam korzystać z funkcji dodawania, edycji i usuwania płyt. Jako miejsce przechowywania danych o naszej kolekcji wybierzemy bazę danych MySql o następującej strukturze:

Fieldname	Type	Null?	Notes
id	Integer	No	Primary key, Autoincrement
artist	Varchar(100)	No	
title	Varchar(100)	No	

3.2.1. Wymagane strony

Następujące strony muszą być więc przygotowane:

- Stronę główną (home page) – będzie wyświetlać listę albumów i linki do edycji i ich usuwania. Również dostępny powinien być link do dodawania nowego albumu.
- Stronę dodawania albumu – będzie wyświetlać formularz dodawania nowego albumu.
- Strona edycji albumu - będzie wyświetlać formularz edycji istniejącego albumu.

- Strona usuwania albumu – będzie wyświetlać stronę potwierdzania usunięcia albumu.

3.2.2. Organizacja stron

Zanim zaczniemy tworzyć nasze pliki, ważne jest by zrozumieć jakiej organizacji plików oczekuje od nas Zend Framework. Każda strona naszej aplikacji będzie rozumiana jako "akcja" (action), a akcje są grupowane w "kontrolery" (controllers). Dla przykładu dla ULR'a formatu
http://localhost/zftutorial/news/view, kontrolerem jest "news", a akcją jest "view". Rozwiązanie to istnieje po to by grupować pokrewne akcje. Więc dla przykładu kontrolera "news", akcjami pokrewnymi mogły by również być "current" (bieżące wiadomości), "archived" (archiwum wiadomości) i "search" (wyniki wyszukiwania). Struktura MVC zaimplementowana w Zend Framework przygotowana jest również do obsługi modułów, które służą do grupowania kontrolerów. Jednak nasza aplikacja nie jest wystarczająco wielka by z nich korzystać.

Kontrolery Zend Framework'a mają zarezerwowaną specjalną nazwę dla jednej z akcji. Akcja ta nazywa się "index" i jest przeznaczona dla adresu URL takiego jak np: http://localhost/zf-tutorial/news/. Jako że w podanym przykładzie nie ma podanej części URL odpowiadającej akcji (jest tylko "news" odpowiadająca nazwie kontrolera), domyślna nazwa akcji "index" będzie użyta. W przypadku gdyby w URL'u nie było podanej nazwy kontrolera, Zend Framework rezerwuje także nazwę domyślnego kontrolera dla takiego przypadku. Nie powinno być to już zaskoczeniem, że nazwą tą jest "index". Reasumując, dla adresu URL: http://localhost/zf-tutorial/ wywołana zostania akcja "index" w kontrolerze "index".

Jako że nasza aplikacja ma być prosta, nie będziemy również zajmować się komplikowaniem "rzeczy" poprzez np. mechanizm do logowania użytkownika.

Jako że mamy zaplanowane wykonanie czterech stron, które wszystkie dotyczą "albumów", zgrupujemy je w jeden kontroler. Tak więc naszymi akcjami w kontrolerze będą:

Strona	Kontroler	Akcja
Główna strona	Index	Index
Dodawanie albumu	Index	Add
Edycja albumu	Index	Edit
Usuwanie album	Index	Delete

3.2.3. Przygotowanie kontrolera

Jesteśmy już gotowi by przygotować nasz kontroler. W Zend Framework kontroler jest klasą, której nazwa składa się z {Nazwa kontrolera}Controller. Trzeba pamiętać, by {Nazwa kontrolera} była napisana małymi literami, a rozpoczynała się z dużej litery. Klasa ta powinna się znajdować w pliku o nazwie {Nazwa kontrolera}Controller.php, a plik ten będzie stworzony w katalogu kontrolerów (controllers). Ponownie, dla pliku, {Nazwa kontrolera} powinna być napisana małymi literami i zaczynać się z dużej litery. Wewnątrz klasy

kontrolera, każda funkcja akcji musi być publiczna. Nazwa akcji składa się z: {nazwa akcji}Acrion i tylko w tym przypadku nazwa ta powinna zaczynać się z małej litery.

Tak więc, jak już wcześniej ustaliliśmy, nasza jedyna klasa kontrolera nazywać się będzie IndexController i zdefiniowana będzie w pliku:

zf-tutorial/application/controllers/IndexController.php :

```
<?php
class IndexController extends Zend_Controller_Action {
  function indexAction() {
    echo "<p>in IndexController::indexAction()</p>";
  }

  function addAction() {
    echo "<p>in IndexController::addAction()</p>";
  }

  function editAction() {
    echo "<p>in IndexController::editAction()</p>";
  }

  function deleteAction() {
    echo "<p>in IndexController::deleteAction()</p>";
  }
}
```

Na dobry początek, ustawiliśmy każdą akcję tak, by tylko wyświetlała swoją nazwę. Dla testu możesz teraz przejść do poniższych adresów URL i zobaczyć rezultat.

URL	Wyświetlony tekst
http://localhost/zf-tutorial	in IndexController::indexAction()
http://localhost/zf-tutorial/index/add	in IndexController::addAction()
http://localhost/zf-tutorial/index/edit	in IndexController::editAction()
http://localhost/zf-tutorial/index/delete	in IndexController::deleteAction()

W tym momencie mamy już poprawnie działający router, oraz dające się wywołać akcje dla każdej strony naszej aplikacji. Jeśli z jakiegoś powodu coś nie zadziałało, poszukaj pomocy w dziale “3.5. Rozwiązywanie problemów”.

Nadszedł czas na zbudowanie widoku (View)

3.2.4. Przygotowanie widoku

W Zend Framework komponent odpowiedzialny nazywa się bardzo odkrywczo: Zend_View. Komponent ten pozwala nam oddzielić kod odpowiedzialny za wyświetlanie strony, od kodu funkcji kontrolerów.

Podstawowym użyciem Zend_View jest :

```
$view = new Zend_View();
$view->setScriptPath('/sciezka/do/pliku_widoku');
echo $view->render('plik_widoku.php');
```

Łatwo zauważyć, że jeśli umieścilibyśmy ten szablon kodu bezpośrednio w każdej akcji, powtórzylibyśmy się - co nie jest wskazane. Zresztą kod ten nie tyczy się tego, czym akcje powinny się zajmować. Wolelibyśmy raczej by obiekt widoku został zainicjowany w innym miejscu, byśmy mogli odwoływać się z naszych akcji bezpośrednio do już zainicjalizowanego obiektu widoku.

Projektanci Zend Framework'a przewidzieli tego typu problem, a rozwiązanie umieścili w "klasie akcji pomocniczych " (action helper). Zend_Controller_Action_Helper_ViewRenderer zajmuje się zainicjowaniem obiektu widoku i umieszczeniem go w własności "widok" w kontrolerze ($this->view). Dzięki temu możemy go wygodnie używać jak i renderować skrypt widoku. Dla renderingu, obiekt Zend_View wyszuka w views/scripts/{nazwa kontrolera}/ plików, których nazwa jest zgodna z nazwą kontrolera, a rozszerzeniem nazwy pliku (przynajmniej domyślnie) jest .phtml. Tak więc plikiem skryptu do renderowania będzie: views/scripts/{nazwa kontrolera}/{nazwa akcji}.phtml. Zrenderowany plik zostaje dołączony do obiektu odpowiedzi (Response object). Obiekt odpowiedzi, dzięki strukturze MVC prócz wygenerowania ciała strony, zgromadzi również nagłówki HTTP oraz ewentualne błędy. Dalej, kontroler frontowy wyśle automatycznie ciało strony (body) poprzedzone nagłówkami w końcowej fazie przetwarzania. Zgrabnie i sprawnie.

Aby zintegrować widok z naszą aplikacją, musimy stworzyć kilka skryptów widoku z testowym kodem.

Żadnych poważnych zmian w kontrolerze nie musimy wykonywać. Dodamy jedynie nowe linie (pogrubiony tekst):

zf-tutorial/application/controllers/IndexController.php

```
<?php
class IndexController extends Zend_Controller_Action {
  function indexAction() {
    $this->view->title = "My Albums";
  }

  function addAction() {
    $this->view->title = "Add New Album";
  }

  function editAction() {
    $this->view->title = "Edit Album";
  }

  function deleteAction() {
    $this->view->title = "Delete Album";
  }
}
```

W każdej funkcji ustawiamy w obiekcie widoku pojedynczą własność. Na razie tylko tyle. Można zauważyć, że żadne wyświetlanie treści nie odbywa się w tym momencie! Stanie się to automatycznie pod koniec procesu przetwarzania kontrolera frontowego.

Teraz musimy dodać cztery skrypty widoku do naszej aplikacji. Pliki te są znane jako "szablony" (templates), a metoda renderująca render() oczekuje, że nazwa pliku szablonu będzie zbudowana z nazwy akcji i rozszerzenia .phtml. Rozszerzenie to ma symbolizować plik szablonu. Pliki szablonu muszą się znajdować w katalogu, którego nazwa jest nazwą kontrolera, z którego te akcje są wywoływane.
Tak więc dla nas są nimi:

zf-tutorial/application/views/scripts/index/index.phtml

```
<html>
  <head>
    <title><?php echo $this->escape($this->title); ?></title>
  </head>

  <body>
    <h1><?php echo $this->escape($this->title); ?></h1>
  </body>
</html>
```

zf-tutorial/application/views/scripts/index/add.phtml

```
<html>
  <head>
    <title><?php echo $this->escape($this->title); ?></title>
  </head>

  <body>
    <h1><?php echo $this->escape($this->title); ?></h1>
  </body>
</html>
```

zf-tutorial/application/views/scripts/index/edit.phtml

```
<html>
  <head>
    <title><?php echo $this->escape($this->title); ?></title>
  </head>

  <body>
    <h1><?php echo $this->escape($this->title); ?></h1>
  </body>
</html>
```

zf-tutorial/application/views/scripts/index/delete.phtml

```
<html>
  <head>
    <title><?php echo $this->escape($this->title); ?></title>
  </head>

  <body>
    <h1><?php echo $this->escape($this->title); ?></h1>
```

```
  </body>
</html>
```

Przetestowanie aplikacji w tym momencie, powinno dać rezultat w postaci wyświetlania tytułów i pogrubionego tekstu.

3.2.5. Wspólny kod HTML

Bardzo szybko staje się oczywistym, że mamy sporo wspólnego kodu HTML w naszych widokach. Zorganizujemy teraz nasz kod w taki sposób by mieć dodatkowe dwa pliki : header.phtml oraz footer.phtml w katalogu z skryptami szablonów. Pliki te wykorzystamy do przechowywania wspólnego dla wszystkich naszych szablonów kodu HTML. Jedyne co więcej musimy zrobić to z naszych czterech szablonów odnieść się do dwóch nowych: nagłówka i stopki.

Nowymi plikami są:

zf-tutorial/application/views/scripts/header.phtml

```
<!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="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <title><?php echo $this->escape($this->title); ?></title>
  </head>

  <body>
    <div id="content">
```

zf-tutorial/application/views/scripts/footer.phtml

```
    </div>
  </body>
</html>
```

Raz jeszcze nasze cztery skrypty widoku wymagają zmian:

zf-tutorial/application/views/scripts/index/index.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/add.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/edit.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
```

```
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/delete.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

3.2.6. Stylizacja

By naszej aplikacji dodać choć trochę prezencji i czytelności będziemy potrzebowali arkusza stylów CSS,. Nie jest to jednak takie proste, ponieważ jako że nie wiemy na jak rzeczywisty główny katalog aplikacji wskazuje adres URL, nie możemy z dowolnego miejsca w szablonie odnieść się do bezpośredniej ścieżki pliku CSS. Aby poradzić sobie z tym, użyjemy funkcji getBaseUrl(), która zawiera się w obiekcie żądania (Request) i jest przekazywana do widoku. Udostępni nam to tę część ścieżki URL, której nie możemy sami określić.

Aby tego dokonać, operację tę wykonamy w funkcji inicjalizacyjnej init(), która jest wywoływana przez konstruktor (__construct()) każdego kontrolera (specyfika Zend Framework). Dzięki temu ustawiona przez nas własność, będzie dostępna z poziomu każdej akcji.

zf-tutorial/application/controllers/IndexController.php

```
...
class IndexController extends Zend_Controller_Action {

  function init() {
    $this->view->baseUrl = $this->_request->getBaseUrl();
  }

  function indexAction() {
…
```

Musimy dodać jeszcze odnośnik do pliku CSS w sekcji <HEAD> pliku header.phtml:

zf-tutorial/application/views/scripts/header.phtml

```
...
<head>
  <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  <title><?php echo $this->escape($this->title); ?></title>
  <link rel="stylesheet" type="text/css" media="screen"
    href="<?php echo $this->baseUrl;?>/public/styles/site.css" />
</head>
…
```

Ostatecznie, powinniśmy stworzyć ów plik CSS:

zf-tutorial/public/styles/site.css

```
body,html {
  font-size:100%;
  margin: 0;
```

```
  font-family: Verdana,Arial,Helvetica,sans-serif;
  color: #000;
  background-color: #fff;
}

h1 {
  font-size:1.4em;
  color: #800000;
  background-color: transparent;
}

#content {
  width: 770px;
  margin: 0 auto;
}

label {
  width: 100px;
  display: block;
  float: left;
}

#formbutton {
  margin-left: 100px;
}

a{
  color: #800000;
}
```

Ta zmiana powinna zauważalnie poprawić wygląd naszej aplikacji.

3.3. Baza danych

Teraz, gdy mamy już rozdzieloną kontrolę aplikacji od jej widoku, nadszedł czas by przyjrzeć się części Modelu aplikacji. Przypomnijmy sobie, iż model jest tą częścią aplikacji, która zajmuje się jej "rdzeniem", obsługą danych wejścia/wyjścia (często nazywany także jako "model biznesowy"). W naszym przypadku jest to obsługa bazy danych. Oczywiście Zend Framework ma przygotowane narzędzia, by wspierać pracę z bazą danych. Wykorzystamy klasę Zend_Db_Table, która zajmie się zapisywaniem, edycją i usuwaniem rekordów w bazie danych.

3.3.1. Konfiguracja

Aby móc użyć Zend_Db_Table, musimy podać informacje takie jak typ bazy danych, jej nazwa, użytkownik i jego hasło. Jako że nie preferujemy "zakodowania" na stałe danych konfiguracyjnych w aplikacji ("hard coding"), użyjemy do tego celu osobnego pliku konfiguracji.

Po raz kolejny, Zend Framework udostępnia nam gotowe rozwiązanie. Zend_Config jest elastycznym obiektem, który pozwala odczytywać dane konfiguracyjne z plików INI lub XML. My wykorzystamy typ INI.

zf-tutorial/application/config.ini

```
[general]
  db.adapter = PDO_MYSQL
  db.config.host = localhost
  db.config.username = greg
  db.config.password = 123456
  db.config.dbname = zftest
```

Oczywiście każdy powinien użyć własnych wartości dla nazwy użytkownika (username), hasła (password) i nazwy bazy danych (dbname), nie moich - przykładowych.

Używanie Zend_Config jest bardzo poste!:

```
$config = new Zend_Config_Ini('config.ini', 'general');
```

W tym przypadku Zend_Config_Ini ładuje tylko jedną sekcję konfiguracyjną (”general”) z pliku config.ini, a nie każdą (owszem, mogło by tak być gdyby nie podać nazwy sekcji). Dodatkowo Zend_Config_Ini, traktuje parametr “kropki” do ustawiania hierarchii ustawień i dodatkowego grupowania ich. W naszym pliku konfiguracji, host, nazwa użytkownika, hasło i nazwa bazy są zgrupowane w : $config->db->config, dzięki zapisowi db.config.{własność}.

Nasz plik konfiguracji załadujemy w “bootstrapperze” :

zf-tutorial/index.php

```
...
Zend_Loader::loadClass('Zend_Controller_Front');
Zend_Loader::loadClass('Zend_Config_Ini');
Zend_Loader::loadClass('Zend_Registry');

// load configuration
$config = new Zend_Config_Ini('./application/config.ini', 'general');
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);

// setup controller
…
```

Nowe linie to te pogrubionym tekstem. Najpierw inkludujemy dwie potrzebne nam klasy Zend_Config_Ini oraz Zend_Registry. W kolejnym kroku tworzymy obiekt konfiguracji, który z pliku application/config.ini ładuje sekcję konfiguracyjną pod nazwą “general”. Ostatecznie referencję do obiektu Zend_Config_Ini zapisujemy w “rejestrze” (korzystając z tego, dostarczonego przez Zend Framework: Zend_Registry), by można było w dowolnym miejscu w aplikacji uzyskać dostęp do tej referencji i skorzystać z zapamiętanego obiektu konfiguracji.

3.3.2. Przygotowanie Zend_Db_Table

Aby korzystać z Zend_Db_Table, musimy przesłać do niego ustawienia konfiguracyjne bazy danych, które właśnie załadowaliśmy z pliku konfiguracyjnego. Aby to uczynić, stworzymy instancję obiektu Zend_Db i zarejestrujemy w tym obiekcie adapter obsługi tabel funkcją

statyczną: Zend_Db_Table:: setDefaultAdapter(). Po raz kolejny nowe linie dodamy do "bootstrappera":

zf-tutorial/index.php

```
...
Zend_Loader::loadClass('Zend_Controller_Front');
Zend_Loader::loadClass('Zend_Config_Ini');
Zend_Loader::loadClass('Zend_Registry');
Zend_Loader::loadClass('Zend_Db');
Zend_Loader::loadClass('Zend_Db_Table');

// load configuration
$config = new Zend_Config_Ini('./application/config.ini', 'general');
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);

// setup database
$db = Zend_Db::factory(          $config->db->adapter,
$config->db->config->toArray() );
Zend_Db_Table::setDefaultAdapter($db);

// setup controller
...
```

3.3.3. Tworzenie tabeli

Jako że używam bazy danych MySQL, polecenie SQL tworzące tabelę jest następujące:

```
CREATE TABLE album (
  id int(11) NOT NULL auto_increment,
  artist varchar(100) NOT NULL,
  title varchar(100) NOT NULL,
  PRIMARY KEY (id)  );
```

Uruchommy powyższe polecenie w kliencie bazy MySQL, takim jak phpMyAdmin lub standardowa linia poleceń w konsoli klienta.

3.3.4. Dodawanie testowych albumów

Dodamy także kilka nowych rekordów do naszej tabeli, dzięki czemu będziemy mogli zaobserwować czy nasza "home page" zachowuje się prawidłowo. Dodajmy więc dwa albumy.

```
INSERT INTO album
  (artist, title)
VALUES
  ('James Morrison', 'Undiscovered'),
  ('Snow Patrol', 'Eyes Open');
```

3.3.5. Przygotowanie modelu

Zend_Db_Table jest klasą abstrakcyjną, więc musimy ją uzupełnić przez dziedziczenie nadając jej specyfikę naszej tabeli albumów. Nie ma znaczenia jak nazwiemy naszą klasę,

jednak ma sens ze względu na przejrzystość, by nazywać ją nazwą tabeli bazy danych, na której będzie operowała (choć nie jest to żadnym wyznacznikiem). Tak więc skoro nasza tabela nazywa się "album", nasza klasa będzie nazywać się będzie 'Album'. Aby Zend_Db_Table wiedział jak się nazywa tabela, którą ma obsługiwać, musimy tę nazwę podać ustawiając chronioną własność tego obiektu $_name nazwą tabeli w bazie danych. Co ważne, Zend_Db_Table zakłada, że obsługiwana przez niego tabela to auto-inkrementacyjny klucz główny o nazwie "id". Nazwa tego pola także może być zmieniona, jeśli to wymagane.

Naszą klasę 'Album', zapiszemy w katalogu modelów:

zf-tutorial/application/models/Album.php

```
<?php
class Album extends Zend_Db_Table {
  protected $_name = 'album';
}
```

Na nasze podstawowe potrzeby, wszelką funkcjonalność przez nas wymaganą dostarczy nam obiekt Zend_Db_Table sam w sobie. Jeśli jednak pisalibyśmy aplikację, gdzie potrzebowalibyśmy więcej funkcjonalności niż ta podstawowa, ten obiekt byłby właściwym miejscem na ich implementację. Dla przykładu mogły by to być funkcje "wyszukiwania", które uwzględniając szczególne warunki i filtry, zwracały by kolekcję specjalnie przygotowanych danych.
Sam obiekt Zend_Db_Table daje także większe możliwości jego wykorzystania, poprzez podawanie mu informacji nt. relacyjnych tabel.

3.4. Budowa kolejnych stron

3.4.1. Wyświetlanie listy albumów

Teraz, gdy mamy już ustawioną konfigurację i informacje nt. bazy danych, możemy przejść do rzeczy: wyświetlenia listy albumów. Stanie się to przez klasę IndexController.

Dla jasności, każda akcja w IndexController będzie manipulowała danymi w tabeli bazy danych "album" używając do tego celu klasy 'Album'. Dlatego też ma uzasadnienie, by załadować klasę albumu wraz z inicjalizacją IndexController (czyli w metodzie init())

zf-tutorial/application/controllers/IndexController.php

```
...
function init() {
  $this->view->baseUrl = $this->_request->getBaseUrl();
  Zend_Loader::loadClass('Album');
}
…
```

Powyżej widzimy przykład użycia Zend_Loader::loadClass() do ładowania plików klas naszego własnego autorstwa. Oczywiście powyższy przykład zadziała poprawnie, ponieważ umieściliśmy ścieżkę plików modelów w ścieżce plików inkludowanych PHP (php include path) w naszym bootstrapperze.

Teraz wylistujemy wszystkie albumy w akcji indexAction(), tak jak mamy to zaplanowane.

zf-tutorial/application/controllers/IndexController.php

```
...
function indexAction() {
  $this->view->title = "My Albums";

  $album = new Album();
  $this->view->albums = $album->fetchAll();
}
…
```

Metoda Zend_Db_Table::fetchAll() zwraca tablicę obiektów Zend_Db_Table_Rowset, która pozwoli nam poprzez iterację uzyskać dostęp do kolejnych rekordów pobranych z bazy danych. Uczynimy to w pliku szablonu:

zf-tutorial/application/views/scripts/index/index.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>

<p><a href="<?php echo $this->baseUrl; ?>/index/add">Add new album</a></p>
<table>
  <tr>
    <th>Title</th>
    <th>Artist</th>
    <th> </th>
  </tr>

  <?php foreach($this->albums as $album) : ?>
  <tr>
    <td><?php echo $this->escape($album->title);?></td>
    <td><?php echo $this->escape($album->artist);?></td>
    <td>
      <a href="<?php echo $this->baseUrl; ?>/index/edit/id/
        <?php echo $album->id;?>">Edit</a>
      <a href="<?php echo $this->baseUrl; ?>/index/delete/id/
        <?php echo $album->id;?>">Delete</a>
    </td>
  </tr>
  <?php endforeach; ?>

</table>
<?php echo $this->render('footer.phtml'); ?>
```

Uruchomienie adresu http://localhost/zf-tutorial/ w przeglądarce powinno wyświetlić śliczną listę (dwóch) albumów.

3.4.2. Dodawanie nowego albumu

Możemy teraz zaprogramować funkcjonalność dodawania albumów. Składają się na to dwa kroki:

- wyświetlenie użytkownikowi formularza do wprowadzenia szczegółów,
- przetworzenie wprowadzonych danych i zapisanie ich w bazie danych

Dokonamy tego w addAction():

zf-tutorial/application/controllers/IndexController.php

```
...
function addAction() {
  $this->view->title = "Add New Album";

  if ($this->_request->isPost()) {
    Zend_Loader::loadClass('Zend_Filter_StripTags');
    $filter = new Zend_Filter_StripTags();

    $artist = $filter->filter($this->_request->getPost('artist'));
    $artist = trim($artist);
    $title = trim($filter->filter(
    $this->_request->getPost('title')));

    if ($artist != '' && $title != '') {
      $data = array(
        'artist' => $artist,
        'title' => $title,
      );
      $album = new Album();
      $album->insert($data);
      $this->_redirect('/');
      return;
    }
  }
  // set up an "empty" album
  $this->view->album = new stdClass();
  $this->view->album->id = null;
  $this->view->album->artist = '';
  $this->view->album->title = '';
  // additional view fields required by form
  $this->view->action = 'add';
  $this->view->buttonText = 'Add';
}
…
```

Zwróćmy uwagę na to jak sprawdzamy wartość zmiennej $_SERVER['REQUEST_METHOD'], by dowiedzieć się czy formularz został wysłany. Jeśli tak, odczytujemy nazwę artysty oraz albumu z tablicy POST i poprzez klasę Zend_Filter_StripTags upewniamy się, że nazwy te nie zawierają w sobie niedopuszczalnych tagów HTML. Wtedy, zakładając że nazwy te były podane poprawnie, wykorzystujemy naszą klasę modelu 'Album', by zapisać informację o nowym albumie do bazy danych.

Po dodaniu nowego albumu, przekierowujemy aplikację do głównej strony metodą kontrolera: _redirect().

Ostatecznie, przygotowujemy w tej akcji kontrolera także pustą klasę standardową, odpowiadającą swoimi własnościami temu, czego oczekuje od nas plik widoku dla tej akcji. Wypełniamy także własności, które nadają nazwę wyświetlaną i nazwę pola "Add" ("dodaj").

Wychodząc krok naprzód, jako że formularz dla akcji dodawania albumu będzie identyczny z formularzem jego edycji, przygotujemy sobie wspólny plik skryptu szablonu _form.phtml, który zaimplementujemy do obu akcji dodawania (add.phtml) oraz edycji (edit.phtml).

Pliki szablonu potrzebne dla akcji dodawania to:

zf-tutorial/application/views/scripts/index/add.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('index/_form.phtml'); ?>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/_form.phtml

```
<form action="<?php echo $this->baseUrl ?>/index/<?php
    echo $this->action; ?>" method="post">
  <div>
    <label for="artist">Artist</label>
    <input type="text" name="artist" value="<?php
      echo $this->escape(trim($this->album->artist));?>"/>
  </div>
  <div>
    <label for="title">Title</label>
    <input type="text" name="title" value="<?php
      echo $this->escape($this->album->title);?>"/>
  </div>
  <div id="formbutton">
    <input type="hidden" name="id" value="<?php
      echo $this->album->id; ?>" />
    <input type="submit" name="add" value="<?php
      echo $this->escape($this->buttonText); ?>" />
  </div>
</form>
```

Jest to dość prosty kod. Jako, że chcemy wykorzystać skrypt _form.phtml także w akcji edycji, użyliśmy zmiennej $this->action zamiast wpisywania "na sztywno" ("hard coding") tej wartości w pliku. W taki sposób również postępujemy z wyświetlaniem nazwy guzika zatwierdzania.

3.4.3. Edycja Albumu

Edycja albumu jest niemal identyczna z dodawaniem go, więc i kod jest bardzo podobny:

zf-tutorial/application/controllers/IndexController.php

```
...
function editAction() {
  $this->view->title = "Edit Album";
  $album = new Album();

  if ($this->_request->isPost()) {
    Zend_Loader::loadClass('Zend_Filter_StripTags');
    $filter = new Zend_Filter_StripTags();

    $id = (int)$this->_request->getPost('id');
    $artist = $filter->filter($this->_request->getPost('artist'));
    $artist = trim($artist);
    $title = trim($filter->filter(
    $this->_request->getPost('title')));
```

```
    if ($id !== false) {
      if ($artist != '' && $title != '') {
        $data = array(
          'artist'      => $artist,
          'title'       => $title,
        );
        $where = 'id = ' . $id;
        $album->update($data, $where);

        $this->_redirect('/');
        return;
      } else {
        $this->view->album = $album->fetchRow('id='.$id);
      }
    }
  } else {
    // album id should be $params['id']
    $id = (int)$this->_request->getParam('id', 0);
    if ($id > 0) {
      $this->view->album = $album->fetchRow('id='.$id);
    }
  }
  // additional view fields required by form
  $this->view->action = 'edit';
  $this->view->buttonText = 'Update';
}
…
```

Tym razem sprawdzamy czy nie było przesyłania danych w trybie POST, staramy się odczytać parametr "id" z tablicy REQUEST (przyp: która to z kolei jest sumą tablic GET i POST. Więc skoro nie było trybu POST podczas pierwszego sprawdzenia, oznacza to, że tak naprawdę w tablicy REQUEST poszykujemy zmiennej typu GET). Do tego celu korzystamy z metody getParam().

Plikiem szablonu dla edycji jest teraz:

zf-tutorial/application/views/scripts/index/edit.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('index/_form.phtml'); ?>
<?php echo $this->render('footer.phtml'); ?>
```

3.4.4. Usuwanie Albumu

Aby dopełnić wszystkich założonych kroków, musimy stworzyć jeszcze akcję usuwania. Mamy już link usuwania wyświetlany przy każdym albumie z osobna, więc z rozpędu, moglibyśmy usuwać album w momencie kliknięcia na link. Jednak to nie jest dobry sposób. Przywołując ku pamięci specyfikację HTTP, nie powinno się wykonywać nieodwracalnych operacji z użyciem parametrów typu GET, lecz POST! Program "Google Web Accelerator" uświadomił tę potrzebę wielu ludziom!

Więc powinniśmy wyświetlić ekran potwierdzający tę operację z przyciskami "tak" lub "nie". Kod będzie wyglądał w pewien sposób podobnie do akcji dodawania i edycji:

zf-tutorial/application/controllers/IndexController.php

```
...
function deleteAction() {
  $this->view->title = "Delete Album";
  $album = new Album();

  if ($this->_request->isPost()) {
    Zend_Loader::loadClass('Zend_Filter_Alpha');
    $filter = new Zend_Filter_Alpha();

    $id = (int)$this->_request->getPost('id');
    $del = $filter->filter($this->_request->getPost('del'));
    if ($del == 'Yes' && $id > 0) {
      $where = 'id = ' . $id;
      $rows_affected = $album->delete($where);
    }
  } else {
    $id = (int)$this->_request->getParam('id');
    if ($id > 0) {
      // only render if we have an id and can find the album.
      $this->view->album = $album->fetchRow('id='.$id);
      if ($this->view->album->id > 0) {
        // render template automatically
        return;
      }
    }
  }
  // redirect back to the album list unless we have rendered the view
  $this->_redirect('/');
}
…
```

Po raz kolejny, wykorzystujemy trik ze sprawdzaniem typu zapytania (GET, POST) by dowiedzieć się, czy mamy wykonać operację wyświetlenia ekranu potwierdzającego usuwanie, czy po prostu usunąć album z bazy danych za pomocą klasy 'Album'. Tak samo jak dodawanie rekordu (insert) i jego edycja (update) w tabeli bazy danych, wykonuje się usuwanie – przez wywołanie Zend_Db_Table::delete().

Zwróćmy uwagę, że wychodzimy z funkcji (return;) zaraz po ustawieniu treści odpowiedzi. Jest to zamierzone działanie by wyświetlić informację o usuwanym albumie, bez odwoływania się do kolejnych funkcji w tej akcji (w szczególności by uniknąć _redirect()).Także, jeśli choć jeden z zapisanych warunków zawiedzie, zostajemy przeniesieni na stronę domową, a konstrukcja tych warunków pozwala posiadać tylko jedną metodę przekierowania (_redirect()) na końcu całej akcji.

Szablon jest prostym formularzem:

zf-tutorial/application/views/scripts/index/delete.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>

<?php if ($this->album) :?>
  <form action="<?php echo $this->baseUrl ?>/index/delete" method="post">
    <p>Are you sure that you want to delete '
```

```
      <?php echo $this->escape($this->album->title); ?>' by '
      <?php echo $this->escape($this->album->artist); ?>'? </p>
    <div>
      <input type="hidden" name="id" value="<?php
        echo $this->album->id; ?>" />
      <input type="submit" name="del" value="Yes" />
      <input type="submit" name="del" value="No" />
    </div>
  </form>

<?php else: ?>
  <p>Cannot find album.</p>
<?php endif;?>

<?php echo $this->render('footer.phtml'); ?>
```

3.5. Rozwiązywanie problemów

Jeśli pojawiły się problemy z działaniem którejkolwiek akcji innej niż index/index, to najprawdopodobniej router nie był w sanie określić bazowego adresu katalogu dla aplikacji. Najczęstszą przyczyną takiego zachowania jest przypadek, gdy URL aplikacji nie odpowiada ścieżce do głównego katalogu (root folder) aplikacji.

Jeśli więc prezentowany kod nie zadziałał poprawnie, można spróbować ustawić $baseURL na poprawną wartość manualnie:

zf-tutorial/index.php

```
...
// setup controller
$frontController = Zend_Controller_Front::getInstance();
$frontController->throwExceptions(true);
$frontController->setBaseUrl('/mysubdir/zf-tutorial');
$frontController->setControllerDirectory('./application/controllers');
…
```

Należy zamienić wyrażenie '/mysubdir/zf-tutorial/' na poprawną ścieżkę wskazującą na katalog w którym znajduje się index.php. Dla przykładu, jeśli URL wskazujący na plik index.php to jest http://localhost/~misiu/zf-tutorial/index.php, wtem poprawny adres bazowy to: '/~misiu/zf-tutorial/'.

4. Podsumowanie

W tym momencie możemy już podziwiać naszą prostą, aczkolwiek w pełni funkcjonalną, opartą o strukturę MVC aplikację zbudowaną na środowisku Zend Framework.

Powyższy opis wykorzystania 'Zend Framework' do stworzenia prostej aplikacji daje jedynie ogólny pogląd na podstawowe możliwości tego frameworka. Przedstawione wyżej biblioteki to jedynie mała część całości, którą można wykorzystać do wielu celów. Najlepszym kolejnym krokiem po zapoznaniu się z przedstawionymi tutaj podstawami, jest rozpoczęcie czytania "instrukcji użytkownika" (manual) na oficialnej stronie frameworka (http://framework.zend.com/manual), a także z wiki (http://framework.zend.com/wiki).