Technologia i rozwiązania

Tworzenie aplikacji z Yii

Receptury

Poznaj najlepsze przepisy dla Yii!

Helion

Alexander Makarov

PACKT PUBLISHING

Tytuł oryginału: Yii Application Development Cookbook, Second Edition

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-246-8596-7

First published in the English language under the title “Yii Application Development Cookbook – Second Edition”

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/twapyi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 8

Rozszerzanie Yii

W tym rozdziale zostaną poruszone następujące zagadnienia:

- Tworzenie zachowań modeli.
- Tworzenie komponentów.
- Tworzenie akcji kontrolerów do wielokrotnego użycia.
- Tworzenie kontrolerów do wielokrotnego użycia.
- Tworzenie widżetu.
- Tworzenie poleceń CLI.
- Tworzenie filtrów.
- Tworzenie modułów.
- Niestandardowe renderowanie widoków.
- Przygotowywanie rozszerzeń do dystrybucji.

Wstęp

W tym rozdziale pokażę nie tylko, jak implementować własne rozszerzenie Yii, ale także jak przygotować rozszerzenie do wielokrotnego użycia i do użytku przez społeczność. Ponadto omówię wiele zagadnień, które należy rozważyć, aby nasze rozszerzenie było możliwie wydajne.

Tworzenie zachowań modeli

W dzisiejszych aplikacjach sieci WWW spotykamy wiele podobnych rozwiązań. Wiodące produkty, takie jak Google Gmail, definiują ciekawe wzorce interfejsu użytkownika. Jednym z nich

jest **nietrwałe usuwanie**. Zamiast ostatecznego usuwania poprzedzonego koniecznością wielu potwierdzeń Gmail pozwala użytkownikowi na natychmiastowe oznaczenie wiadomości jako usuniętych, a następnie na ich łatwe przywrócenie. To samo zachowanie można zastosować do dowolnego obiektu, takiego jak posty bloga, komentarze itd.

Utwórzmy zachowanie, które pozwoli na oznaczanie modeli jako usuniętych, przywracanie modeli i zaznaczanie jeszcze nie usuniętych, usuniętych oraz wszystkich modeli. W tej recepturze do zaplanowania zachowania i przetestowania, czy implementacja jest poprawna, zastosujemy podejście test-driven development.

Przygotowanie

Wykonaj poniższe czynności:

1. Utwórz bazę danych i dodaj do niej tabelę `post`:

```
CREATE TABLE `post` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  `title` varchar(255) default NULL,
  `is_deleted` tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (`id`)
)
```

2. Skonfiguruj Yii, aby korzystał z tej bazy danych w aplikacji głównej (*protected/config/main.php*).
3. Upewnij się, że aplikacja `test` ma takie same ustawienia (*protected/config/test.php*).
4. Usuń komentarz z komponentu `fixture` w ustawieniach aplikacji `test`.
5. Skorzystaj z Gii do wygenerowania modelu `Post`.

Jak to zrobić…

1. Przygotujmy środowisko testowe, zaczynając od zdefiniowania konfiguracji testów dla modelu `Post` w pliku *protected/tests/fixtures/post.php*:

```
<?php
return array(
  array(
    'id' => 1,
    'title' => 'post1',
    'text' => 'post1',
    'is_deleted' => 0,
  ),
  array(
    'id' => 2,
    'title' => 'post2',
    'text' => 'post2',
```

```
    'is_deleted' => 1,
  ),
  array(
    'id' => 3,
    'title' => 'post3',
    'text' => 'post3',
    'is_deleted' => 0,
  ),
  array(
    'id' => 4,
    'title' => 'post4',
    'text' => 'post4',
    'is_deleted' => 1,
  ),
  array(
    'id' => 5,
    'title' => 'post5',
    'text' => 'post5',
    'is_deleted' => 0,
  ),
);
```

2. Musimy utworzyć przypadek testowy. Umieść następujący kod w pliku *protected/tests/unit/soft_delete/SoftDeleteBehaviorTest.php*:

```
<?php
class SoftDeleteBehaviorTest extends CDbTestCase
{
  protected $fixtures = array(
    'post' => 'Post',
  );

  function testRemoved()
  {
    $postCount = Post::model()->removed()->count();
    $this->assertEquals(2, $postCount);
  }

  function testNotRemoved()
  {
    $postCount = Post::model()->notRemoved()->count();
    $this->assertEquals(3, $postCount);
  }

  function testRemove()
  {
    $post = Post::model()->findByPk(1);
    $post->remove()->save();
    $this->assertNull(Post::model()->notRemoved()->findByPk(1));
  }
```

```
  function testRestore()
  {
    $post = Post::model()->findByPk(2);
    $post->restore()->save();
    $this->assertNotNull(Post::model()->notRemoved()->findByPk(2));
  }

  function testIsDeleted()
  {
    $post = Post::model()->findByPk(1);
    $this->assertFalse($post->isRemoved());
    $post = Post::model()->findByPk(2);
    $this->assertTrue($post->isRemoved());
  }
}
```

3. Następnie musimy zaimplementować zachowanie, dołączyć je do modelu i upewnić się, że test zakończy się powodzeniem. Utwórz nowy podfolder *soft_delete* w folderze *protected/extensions*. Umieść w tym podkatalogu nowy plik *SoftDeleteBehavior.php*. Najpierw dołącz zachowanie do modelu `Post`:

```
class Post extends CActiveRecord
{
  // ...

  public function behaviors()
  {
    return array(
      'softDelete' => array(
        'class' => 'ext.soft_delete.SoftDeleteBehavior'
      ),
    );
  }

  // ...
```

4. Teraz zaimplementuj klasę *protected/extensions/soft_delete/SoftDeleteBehavior.php*:

```
<?php
class SoftDeleteBehavior extends CActiveRecordBehavior
{
  public $flagField = 'is_deleted';

  public function remove()
  {
    $this->getOwner()->{$this->flagField} = 1;
    return $this->getOwner();
  }

  public function restore()
  {
```

```
    $this->getOwner()->{$this->flagField} = 0;
    return $this->getOwner();
  }

  public function notRemoved()
  {
    $criteria = $this->getOwner()->getDbCriteria();
    $criteria->compare($this->flagField, 0);
    return $this->getOwner();
  }

  public function removed()
  {
    $criteria = $this->getOwner()->getDbCriteria();
    $criteria->compare($this->flagField, 1);
    return $this->getOwner();
  }

  public function isRemoved()
  {
    return (boolean)$this->getOwner()->{$this->flagField};
  }
}
```

5. Uruchom test i upewnij się, że zakończył się pomyślnie.
6. To wszystko. Utworzyliśmy zachowanie wielokrotnego użycia, które możemy wykorzystać w kolejnych projektach jedynie poprzez dołączenie ich do modelu.

Opis działania

Zacznijmy od przypadku testowego. Ponieważ zamierzamy użyć zestawu modeli, definiujemy konfiguracje testów. Zestaw konfiguracji jest umieszczany w bazie danych za każdym razem, gdy wykonywana jest metoda testowa. Aby skorzystać z konfiguracji, klasa testu powinna być dziedziczona po klasie `CDbTestCase` i mieć zadeklarowaną właściwość prywatną `$fixtures`.

```
protected $fixtures = array(
  'post' => 'Post',
);
```

W powyższej definicji `post` to nazwa pliku zawierającego definicje konfiguracji testów, zaś `Post` to nazwa modelu, do którego zostaną zastosowane konfiguracje testów.

Najpierw testujemy własne zakresy `removed` i `notRemoved`. Powinniśmy ograniczyć rezultat wyszukiwania jedynie do usuniętych elementów, a następnie do elementów nieusuniętych. Ponieważ wiemy, jakie dane uzyskamy z konfiguracji testów, możemy w następujący sposób sprawdzić w testach liczbę elementów usuniętych i nieusuniętych:

```
$postCount = Post::model()->removed()->count();
$this->assertEquals(2, $postCount);
```

Następnie testujemy metody `remove` i `restore`. Metoda testowa `remove` ma następującą postać:

```
$post = Post::model()->findByPk(1);
$post->remove()->save();
$this->assertNull(Post::model()->notRemoved()->findByPk(1));
```

Element definiujemy poprzez identyfikator `id`, usuwamy go, a następnie próbujemy go odzyskać, korzystając z zakresu `notRemoved`. Ponieważ element został usunięty, powinniśmy w wyniku otrzymać `null`.

Na zakończenie testujemy metodę `isRemoved`, która po prostu zwraca wartość odpowiedniej kolumny w postaci boolowskiej wartości logicznej.

Teraz przejdźmy do interesujących szczegółów implementacji. Ponieważ implementujemy zachowanie modelu aktywnego rekordu, musimy skorzystać z rozszerzenia klasy `CActiveRecordBehavior`. W zachowaniu możemy dodać nasze własne metody. Zostaną one włączone do modelu, do którego dołączone będzie zachowanie. W ten sposób dodajemy metody znajdujące się w zakresach `remove/restore/isRemoved` i `removed/notRemoved`:

```
public function remove()
{
  $this->getOwner()->{$this->flagField} = 1;
  return $this->getOwner();
}
public function removed()
{
  $criteria = $this->getOwner()->getDbCriteria();
  $criteria->compare($this->flagField, 1);
  return $this->getOwner();
}
```

W obydwu metodach używamy metody `getOwner` do uzyskania obiektu, do którego dołączono zachowanie. W naszym przypadku jest to model, dlatego możemy pracować z jego danymi lub zmienić jego kryteria wyszukiwania. Zwracamy instancję modelu, aby umożliwić łańcuchowe wywołania metod w następujący sposób:

```
$post->remove()->save();
```

Dodatkowe informacje

Pozostało jeszcze kilka spraw, które należy omówić w niniejszej recepturze.

CActiveRecordBehavior i CModelBehavior

Niekiedy przydałoby się nieco więcej elastyczności w zachowaniu, na przykład aby umożliwić reagowanie na zdarzenia modelu. Zarówno w `CActiveRecordBehavior`, jak i w `CModelBehavior` dostępne są metody odpowiadające zdarzeniom, które można nadpisać, aby obsłużyć zdarzenia modelu. Przykładowo: jeśli chcemy obsłużyć w zachowaniu kaskadowe usuwanie, możemy to zrobić, nadpisując metodę `afterDelete`.

Więcej typów zachowań

Zachowania można dołączyć nie tylko do modelu, ale również do dowolnego komponentu. Każde zachowanie dziedziczy po klasie `CBehavior`, dlatego możemy użyć jego metod w następujący sposób:

- Metoda `getOwner` służy do uzyskania komponentu, do którego dołączono zachowanie.
- Metody `getEnabled` i `setEnabled` służą do sprawdzania, czy zdarzenie jest aktywne, i do ustawiania jego stanu.
- Metody `attach` i `detach` mogą być odpowiednio używane do inicjalizowania zdarzenia i czyszczenia tymczasowych danych, utworzonych podczas korzystania z zachowania.

Warto przeczytać

Dodatkowe informacje o zachowaniach można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/api/CActiveRecordBehavior*
- *http://www.yiiframework.com/doc/api/CModelBehavior*
- *http://www.yiiframework.com/doc/api/CBehavior*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie komponentów

Jeśli pewien kod w naszej aplikacji wygląda jak kod wielokrotnego użytku, ale nie wiemy, czy jest to zachowanie, widżet, czy coś innego, prawdopodobnie jest to komponent. Komponent powinien dziedziczyć po klasie `CComponent` lub `CApplicationComponent`. Po utworzeniu można go dołączyć do aplikacji i skonfigurować w pliku konfiguracyjnym *protected/config/main.php*. Jest to główna korzyść w porównaniu z używaniem czystej klasy PHP. Ponadto uzyskujemy wsparcie dla zachowań, zdarzeń, getterów i setterów.

W naszym przykładzie zaimplementujemy prosty komponent aplikacji `EImageManager`, który będzie mógł skalować obrazy, korzystając z biblioteki GD, dołączyć ją do aplikacji i jej użyć.

Przygotowanie

Aby wypróbować skalowanie obrazów, musimy zainstalować rozszerzenie PHP o nazwie GD.

Jak to zrobić…

1. Utwórz plik *protected/components/EImageManager.php*:

```
<?php
class EImageManager extends CApplicationComponent
{
  protected $image;
  protected $width;
  protected $height;

  protected $newWidth;
  protected $newHeight;

  public function resize($width = false, $height = false){
    if($width!==false) $this->newWidth = $width;
    if($height!==false) $this->newHeight = $height;

    return $this;
  }

  public function load($filePath)
  {
    list($this->width, $this->height, $type) = getimagesize($filePath);

    switch ($type)
    {
      case IMAGETYPE_GIF:
        $this->image = imagecreatefromgif($filePath);
      break;
      case IMAGETYPE_JPEG:
        $this->image = imagecreatefromjpeg($filePath);
      break;
      case IMAGETYPE_PNG:
        $this->image = imagecreatefrompng($filePath);
      break;
      default:
        throw new CException('Niewłaściwy typ obrazu ' . $type);
    }
```

```
    return $this;
  }

  public function save($filePath)
  {
    $ext = pathinfo($filePath, PATHINFO_EXTENSION);
    $newImage = imagecreatetruecolor($this->newWidth, $this->newHeight);
    imagecopyresampled($newImage, $this->image, 0, 0, 0, 0,
      $this->newWidth, $this->newHeight, $this->width,
      $this->height);
    switch($ext)
    {
      case 'jpg':
      case 'jpeg':
        imagejpeg($newImage, $filePath);
      break;
      case 'png':
        imagepng($newImage, $filePath);
      break;
      case 'gif':
        imagegif($newImage, $filePath);
      break;
      default:
        throw new CException("Niewłaściwy typ obrazu ", $ext);
    }

    imagedestroy($newImage);
    if(!is_file($filePath))
      throw new CException("Niepowodzenie zapisu.");
  }

  function __destruct()
  {
    imagedestroy($this->image);
  }
}
```

2. Następnie musimy dołączyć nasz komponent do aplikacji. W pliku *protected/config/main.php* musimy dodać następujący kod:

```
...
// application components
'components'=>array(
  'image' => array(
    'class' => 'EImageManager',
  ),
...
```

3. Teraz możemy użyć komponentu w następujący sposób:

```
Yii::app()->image
  ->load(Yii::getPathOfAlias('webroot').'/src.png')
  ->resize(100,100)
  ->save(Yii::getPathOfAlias('webroot').'/dst.png');
```

Opis działania

Aby można było dodać komponent do aplikacji, musi on dziedziczyć po klasie `CApplicationComponent`. Dołączanie jest tak proste jak dodawanie nowej tablicy do sekcji `component` w konfiguracji. Wartość `class` w tablicy określa klasę komponentu, zaś inne wartości są ustawiane poprzez odpowiadające im właściwości publiczne komponentu i settery.

Sama implementacja jest bardzo prosta. Wywołania GD opakowujemy w wygodne API, zawierające metody `save`, `load` i `resize`. Aby można było wykonywać łańcuchowe wywołania API, `load` i `resize` zwracają sam komponent.

Korzystając z `Yii::app()`, możemy uzyskać dostęp do naszej klasy poprzez nazwę jej komponentu. W naszym przypadku będzie to `Yii::app()->image`.

Dodatkowe informacje

Oprócz tworzenia własnych komponentów możemy dokonać wiele więcej.

Nadpisywanie istniejących komponentów aplikacji

Przez większość czasu nie ma potrzeby tworzenia własnych komponentów aplikacji, ponieważ inne rodzaje rozszerzeń takich jak widżety czy zachowania zawierają niemal wszystkie rodzaje kodu wielokrotnego użytku. Jednakże nadpisywanie komponentów podstawowej platformy jest popularną praktyką i można go użyć do dostosowania zachowania platformy dla własnych potrzeb bez konieczności wprowadzania zmian w kodzie platformy.

Przykładowo: aby można było uzyskać rolę użytkownika z bazy danych z wykorzystaniem `Yii::app()->user->role`, możemy w następujący sposób zaimplementować dziedziczenie po komponencie `CWebUser`:

```
<?php
class WebUser extends CWebUser {
    private $_model = null;
    function getRole() {
        if($user = $this->getModel()){
            return $user->role;
        }
        else return 'guest';
    }
```

```
    private function getModel(){
        if($this->_model === null){
            if($this->id === null) return null;
            $this->_model = User::model()->findByPk($this->id);
        }
        return $this->_model;
    }
}
```

Aby zastąpić standardowy komponent użytkownika, należy dostosować plik konfiguracyjny *main.php*:

```
...
// application components
'components'=>array(
  'user'=>array(
    'class' => 'WebUser',
    // other properties
  ),
...
```

W powyższym kodzie określiliśmy nową klasę `class` dla komponentu `user`.

Warto przeczytać

Dodatkowe informacje o komponentach można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/api/CComponent/*
- *http://www.yiiframework.com/doc/api/CApplicationComponent/*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie akcji kontrolerów do wielokrotnego użycia

Popularne akcje, takie jak usuwanie modelu aktywnego rekordu według klucza prywatnego lub uzyskiwanie danych dla automatycznego uzupełniania z wykorzystaniem AJAX, można przenieść do akcji wielokrotnego użytku i później w razie potrzeby dołączyć ją do kontrolera.

W tej recepturze utworzymy akcję do wielokrotnego użytku `delete`, która będzie usuwać określony model aktywnego rekordu według jego klucza głównego.

Przygotowanie

1. Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.
2. Utwórz i skonfiguruj nową bazę danych.
3. Wykonaj następujące zapytanie SQL:

```
CREATE TABLE `post` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  `title` varchar(255) default NULL,
  PRIMARY KEY  (`id`)
);
CREATE TABLE `comment` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  PRIMARY KEY  (`id`)
);
```

4. Wygeneruj modele dla tabel `post` i `comment`, korzystając z Gii.

Jak to zrobić…

1. Utwórz plik *protected/extensions/actions/EDeleteAction.php* i umieść w nim następujący kod:

```
<?php
class EDeleteAction extends CAction
{
  public $modelName;
  public $redirectTo = array('index');

  /**
   * Uruchamia akcję.
   * Ta metoda jest wywoływana przez kontroler posiadający tę akcję.
   */
  public function run($pk)
  {
    CActiveRecord::model($this->modelName)->deleteByPk($pk);
    if(Yii::app()->getRequest()->getIsAjaxRequest())
    {
      Yii::app()->end(200, true);
    }
    else
    {
      $this->getController()->redirect($this->redirectTo);
    }
  }
}
```

2. Następnie dołączymy akcję do kontrolera *protected/controllers/DeleteController.php*:

```
<?php
class DeleteController extends CController
{
  public function actions()
  {
    return array(
      'deletePost' => array(
        'class' => 'ext.actions.EDeleteAction',
        'modelName' => 'Post',
        'redirectTo' => array('indexPosts'),
      ),
      'deleteComment' => array(
        'class' => 'ext.actions.EDeleteAction',
        'modelName' => 'Comment',
        'redirectTo' => array('indexComments'),
      ),
    );
  }

  public function actionIndexPosts()
  {
    echo "Jestem akcją index dla modelu Post.";
  }

  public function actionIndexComments()
  {
    echo " Jestem akcją index dla modelu Comment.";
  }
}
```

3. To wszystko. Teraz możesz usunąć post, odwiedzając stronę */delete/deletePost/pk/<pk>*, a także usunąć komentarz, odwiedzając */delete/deleteComment/pk/<pk>*. Po usunięciu zostaniemy przekierowani do odpowiedniej akcji *index*.

Opis działania

Aby utworzyć akcję zewnętrznego kontrolera, nasza klasa powinna dziedziczyć po klasie `CAction`. Jedyną metodą, którą należy zaimplementować, jest `run`. W naszym przypadku akceptuje ona parametr o nazwie `pk` ze zmiennej `$_GET`, korzystając z mechanizmu automatycznego dołączania parametru dostępnego w Yii, i próbuje usunąć odpowiedni model.

Aby zapewnić możliwość dostosowywania komponentu, utworzyliśmy dwie publiczne właściwości, które można konfigurować poprzez kontroler. Jest to właściwość `modelName` przechowująca nazwę modelu, z którym pracujemy, oraz `redirectTo` określająca trasę, do której użytkownik zostanie przekierowany.

Sama konfiguracja odbywa się poprzez implementację metody `actions` w kontrolerze. Możemy tam dołączyć akcję raz lub kilka razy i skonfigurować jej właściwości publiczne.

Dodatkowe informacje

W klasie `CAction` zaimplementowane są dwie przydatne metody. Pierwsza to `getController`. Możemy jej użyć do pobrania instancji kontrolera, do którego dołączona jest akcja. Będzie nam to potrzebne do przekierowania do innej akcji lub na przykład do wygenerowania łańcucha URL.

Drugą metodą jest `getId` zwracająca nazwę akcji, określoną w metodzie `actions` kontrolera.

Warto przeczytać

Dodatkowe informacje o zewnętrznej akcji kontrolera można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/api/CAction/*
- *http://www.yiiframework.com/doc/api/CController/#actions-detail*

Zobacz też

- Receptura „Tworzenie kontrolerów wielokrotnego użytku”.
- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie kontrolerów wielokrotnego użytku

W Yii możemy tworzyć kontrolery wielokrotnego użytku. Jeśli rozwijamy wiele aplikacji lub kontrolerów tego samego rodzaju, przeniesienie całego wspólnego kodu do kontrolera wielokrotnego użytku pozwoli nam zaoszczędzić wiele czasu.

W tej recepturze utworzymy prosty kontroler wielokrotnego użytku o nazwie `api`, który zaimplementuje proste `API JSON CRUD` dla modelu. Będzie ono pobierać dane z metod `POST` i `GET` i odpowiadać danymi w formacie `JSON` oraz odpowiednim kodem odpowiedzi HTTP.

Przygotowanie

1. Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.
2. Utwórz nową bazę danych i wykonaj następujące zapytanie SQL:

```
CREATE TABLE `post` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  `title` varchar(255) default NULL,
  PRIMARY KEY (`id`)
);
```

3. Skonfiguruj aplikację, aby korzystała z utworzonej bazy danych, i wygeneruj model, korzystając z Gii. W naszym przykładzie użyjemy modelu `Post`, jednak może to być dowolny model.

Jak to zrobić…

1. Utwórz plik *protected/extensions/json_api/JsonApiController.php* i umieść w nim następujący kod:

```
<?php
class JsonApiController extends CController
{
  const RESPONSE_OK = 'OK';
  const RESPONSE_NO_DATA = 'Brak danych';
  const RESPONSE_NOT_FOUND = 'Nie znaleziono';
  const RESPONSE_VALIDATION_ERRORS = 'Błędy walidacji';

  public $modelName;

  public function init()
  {
    parent::init();
    if(empty($this->modelName))
      throw new CException("Należy określić modelName przed użyciem
          JsonApiController.");
  }

  public function actionCreate()
  {
    if(empty($_POST))
      $this->respond(400, self::RESPONSE_NO_DATA);

    $model = new $this->modelName;
    $model->setAttributes($_POST);

    if($model->save())
      $this->respond(200, self::RESPONSE_OK);
    else
      $this->respond(400, self::RESPONSE_VALIDATION_ERRORS,
          $model->getErrors());
  }
```

```
  public function actionGet($pk)
  {
    $model = CActiveRecord::model($this->modelName)->findByPk($pk);
    if(!$model)
      $this->respond(404, self::RESPONSE_NOT_FOUND);
    $this->respond(200, self::RESPONSE_OK, $model->getAttributes());
  }

  public function actionUpdate($pk)
  {
    if(empty($_POST))
      $this->respond(400, self::RESPONSE_NO_DATA);
    $model = CActiveRecord::model
      ($this->modelName)->findByPk($pk);
    if(!$model)
      $this->respond(404, self::RESPONSE_NOT_FOUND);

    $model->setAttributes($_POST);
    if($model->save())
      $this->respond(200, self::RESPONSE_OK);
    else
      $this->respond(400, self::RESPONSE_VALIDATION_ERRORS,
        $model->getErrors());
  }
  public function actionDelete($pk)
  {
    if(CActiveRecord::model($this->modelName)->deleteByPk($pk))
    {
      $this->respond(200, self::RESPONSE_OK);
    }
    else {
      $this->respond(404, self::RESPONSE_NOT_FOUND);
    }
  }
  protected function respond($httpCode, $status, $data = array())
  {
    $response['status'] = $status;
    $response['data'] = $data;
    echo CJSON::encode($response);
    Yii::app()->end($httpCode, true);
  }
}
```

2. Następnie musimy dołączyć kontroler do naszej aplikacji poprzez plik *protected/config/main.php*. Można to osiągnąć poprzez dodanie konfiguracji kontrolera do właściwości `controllerMap` klasy `CWebApplication`, dlatego musimy umieścić następujący kod na początku tablicy konfiguracji:

```
...
'controllerMap' => array(
```

```
    'api' => array(
      'class' => 'ext.json_api.JsonApiController',
      'modelName' => 'Post',
    ),
  ),
  ...
```

3. To wszystko. Dołączyliśmy kontroler i skonfigurowaliśmy go, aby można było pracować z modelem `Post`.

   Potrzebne są nam formularze do przesłania danych, jednak jeśli mamy już jakieś dane, możemy użyć metod `get` bezpośrednio z URL, na przykład `/api/get/pk/1`. Aplikacje powinny zwracać dane w następującym formacie:

```
{"status":"OK","data":{"id":"1","text":"post1",
  "title":"post1","is_deleted":"0"}}
```

Opis działania

Gdy uruchomiona jest aplikacja i przekazujemy do niej trasę taką jak *api/get*, to zanim uruchomimy `ApiController::actionGet Yii`, musimy sprawdzić, czy zdefiniowany został `controllerMap`. Ponieważ zdefiniowany jest tu kontroler `api`, Yii uruchamia go, zamiast postępować w zwykły sposób.

W samym kontrolerze zdefiniowaliśmy właściwość `modelName`, aby można było połączyć się wielokrotnie z kontrolerem i uzyskać API dla kilku modeli. Ustawiamy ją, gdy dołączamy kontroler.

Sam kontroler nie różni się znacznie od zwykłego kontrolera, z wyjątkiem kilku szczegółów:

- Podczas pracy z aktywnym rekordem nową klasę tworzymy w następujący sposób:

  ```
  $model = new $this->modelName;
  ```

- Model znajdujemy w następujący sposób:

  ```
  $model = CActiveRecord::model($this->modelName);
  ```

- To pozwala nam uniknąć sztywnego przypisania do specyficznego modelu i zamiast tego określić nazwę modelu.
- Używamy `CJSON`, aby zakodować tablice i modele do formatu JSON.
- `Yii::app()->end()` służy do zakończenia wykonywania aplikacji ze zwróceniem określonego kodu odpowiedzi HTTP.

Dodatkowe informacje

Dodatkowe informacje o mapie kontrolera można znaleźć na następującej stronie API:

http://www.yiiframework.com/doc/api/CWebApplication#controllerMap-detail

Zobacz też

- Receptura „Tworzenie akcji kontrolerów wielokrotnego użytku”.
- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie widżetów

Widżet to część widoku do wielokrotnego użytku, która nie tylko renderuje pewne dane, ale również wykonuje to według określonej logiki. Może nawet pobierać dane z modeli i używać własnych widoków, dlatego przypomina zredukowaną używalną wersję modułu.

Utwórzmy widżet, który narysuje wykres kołowy z wykorzystaniem Google API.

Przygotowanie

Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.

Jak to zrobić…

1. Utwórz plik *protected/extensions/chart/EChartWidget.php*:

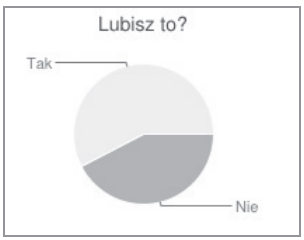
```
<?php
class EChartWidget extends CWidget
{
  public $title;
  public $data=array();
  public $labels=array();
  public function run()
  {
    echo "<img src=\"http://chart.apis.google.com/chart?chtt=".urlencode(
       $this->title)."&cht=pc&chs=300x150&chd=".$this->encodeData(
       $this->data)."&chl=".implode('|', $this->labels)."\">";
  }
  protected function encodeData($data)
  {
    $maxValue=max($data);
    $chars='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
    $chartData="s:";
    for($i=0;$i<count($data);$i++)
    {
      $currentValue=$data[$i];
      if($currentValue>-1)
        $chartData.=substr($chars,61*($currentValue/$maxValue),1);
```

```
        else
          $chartData.='_';
      }
      return $chartData."&chxt=y&chxl=0:|0|".$maxValue;
    }
  }
```

2. Następnie utwórz kontroler *protected/controllers/ChartController.php* i umieść w nim następujący kod:

```
<?php
class ChartController extends CController
{
  public function actionIndex()
  {
    $value = rand(10, 90);
    $this->widget('ext.chart.EChartWidget', array(
      'title' => 'Lubisz to?',
      'data' => array(
        $value, 100-$value
      ),
      'labels' => array(
        'Nie',
        'Tak',
      ),
    ));
  }
}
```

3. Teraz uruchom akcję *index* kontrolera. Powinieneś ujrzeć wykres kołowy podobny do poniższego:



Opis działania

Podobnie jak w każdym innym rodzaju rozszerzenia tworzymy pewne publiczne właściwości, które można skonfigurować podczas wywoływania widżetu metodą `CController::widget`. W tym przypadku konfigurujemy tytuł, zestaw danych oraz etykiety danych.

Główną metodą widżetu jest `run()`. W naszym widżecie generujemy URL wskazujący na API wykresów Google, a następnie wyświetlamy znacznik `<img>`.

Dodatkowe informacje

Dodatkowe informacje o widżetach można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/api/CWidget/*
- *http://www.yiiframework.com/doc/api/CCaptcha/*

Zobacz też

- Receptura „Konfiguracja komponentów" w rozdziale 1. „Dla zaawansowanych".
- Receptura „Konfiguracja domyślnych ustawień widżetów" w rozdziale 1. „Dla zaawansowanych".
- Receptura „Tworzenie własnego widżetu do pobierania danych z wykorzystaniem CInputWidget" w rozdziale 4. „Używanie formularzy".
- Receptura „Przygotowywanie rozszerzeń do dystrybucji".

Tworzenie poleceń CLI

Yii wspiera polecenia wiersza poleceń i pozwala na tworzenie poleceń wielokrotnego użytku. Polecenia konsolowe są szybsze do utworzenia niż graficzny interfejs dla sieci WWW. Jeśli chcemy utworzyć pewien rodzaj użytecznego narzędzia dla aplikacji, która będzie używana przez deweloperów lub administratorów, polecenia konsolowe są najlepszym wyborem.

Aby sprawdzić, jak tworzyć polecenia konsolowe, utworzymy proste polecenie, które wyczyści kilka elementów, takich jak pamięć podręczna, tymczasowe foldery itd.

Przygotowanie

Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.

Jak to zrobić…

1. Utwórz plik *protected/extensions/clean_command/ECleanCommand.php* i umieść w nim następujący kod:

```
<?php
class ECleanCommand extends CConsoleCommand
{
  public $webRoot;
  public function actionCache()
  {
```

```
    $cache=Yii::app()->getComponent('cache');
        if($cache!==null){
            $cache->flush();
            echo "Gotowe.\n";
        }
        else {
            echo "Skonfiguruj komponent cache.\n";
        }
    }

    public function actionAssets()
    {
      if(empty($this->webRoot))
      {
        echo "Podaj ścieżkę do folderu webRoot w parametrze polecenia.\n";
        Yii::app()->end();
      }
      $this->cleanDir($this->webRoot.'/assets');
          echo "Gotowe.\n";
    }

    public function actionRuntime()
    {
      $this->cleanDir(Yii::app()->getRuntimePath());
      echo " Gotowe.\n";
    }

    private function cleanDir($dir)
    {
      $di = new DirectoryIterator ($dir);
          foreach($di as $d)
          {
              if(!$d->isDot())
              {
                  echo "Usunięto ".$d->getPathname()."\n";
                  $this->removeDirRecursive($d->getPathname());
              }
          }
    }

    private function removeDirRecursive($dir)
    {
          $files = glob($dir.DIRECTORY_SEPARATOR.'{,.}*', GLOB_MARK
| GLOB_BRACE);
          foreach ($files as $file)
          {
                          if(basename($file) == '.' ||
basename($file) == '..')
                        continue;               if (substr($file,
```

```
- 1) == DIRECTORY_SEPARATOR)
                        $this->removeDirRecursive($file);
                else
                        unlink($file);
                }
                if (is_dir($dir))
                        rmdir($dir);
  }
}
```

2. Następnie musimy dołączyć polecenie do aplikacji konsolowej. Domyślnie aplikacja konsolowa korzysta z oddzielnego pliku *yiic.php*:

```
<?php
// change the following paths if necessary
$yiic=dirname(__FILE__).'/../../framework/yiic.php';
$config=dirname(__FILE__).'/config/console.php';
require_once($yiic);
```

3. Dlatego konfiguracja mieści się w pliku *protected/config/console.php*. Dodajmy nasze polecenie konsolowe do właściwości `commandMap`:

```
// This is the configuration for yiic console application.
// Any writable CConsoleApplication properties can be configured here.
return array(
  'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
  'name'=>'My Console Application',
  'commandMap' => array(
    'clean' => array(
      'class' => 'ext.clean_command.ECleanCommand',
      'webRoot' => 'sciezka/do/folderu/webroot/aplikacji',
    ),
  ),
);
```

4. Nie zapomnij zmienić elementu `webRoot` na rzeczywistą ścieżkę. To wszystko. Teraz przejdź do folderu *protected* i spróbuj uruchomić poniższe polecenia:

```
yiic clean
yiic clean cache
yiic clean assets
yiic clean runtime
```

Opis działania

Wszystkie polecenia konsolowe powinny dziedziczyć po klasie `CConsoleCommand`. Ponieważ wszystkie polecenia konsolowe są uruchamiane w klasie `CConsoleApplication`, a nie w klasie `CWebApplication`, nie ma sposobu na wyznaczenie głównego folderu aplikacji sieci WWW. W tym celu tworzymy konfigurowalną właściwość publiczną zwaną `webRoot`.

Struktura polecenia konsolowego przypomina typowy kontroler. Definiujemy kilka akcji, które można uruchomić poleceniem `yiic <polecenie konsolowe> <akcja polecenia>`.

Jak widać, nie używamy żadnego widoku, dlatego możemy się skupić na zadaniach programistycznych zamiast na projekcie graficznym, kodzie itd. Nadal jednak musimy zwrócić przydatny wynik, aby użytkownicy wiedzieli, co się dzieje. Wykonywane jest to poprzez zwykłe instrukcje PHP `echo`.

Dodatkowe informacje

Jeśli polecenie jest dość złożone, podobnie jak `message` lub `migrate`, które są wbudowane w Yii, dobrą decyzją jest dostarczenie pewnych dodatkowych informacji na temat dostępnych opcji i akcji. Można tego dokonać poprzez nadpisanie metody `getHelp`.

```
public function getHelp()
{
  $out = "Polecenie clean pozwala wyczyścić różne dane tymczasowe Yii oraz
          wygenerowane przez aplikację.\n\n";
  return $out.parent::getHelp();
}
```

Po uruchomieniu samego polecenia `yiic clean` uzyskamy następujący wynik:

```
Polecenie clean pozwala wyczyˇciŠ r˘ne dane tymczasowe
Yii oraz wygenerowane przez aplikacjŕ.

Usage: yiic clean <action>
Actions:
    cache
    assets
    runtime

C:\xampp\htdocs\example6_7\protected>
```

Warto przeczytać

Dodatkowe informacje o tworzeniu aplikacji i poleceń konsolowych można znaleźć na następujących stronach:

- *http://www.yiiframework.com/doc/guide/1.1/pl/topics.console*
- *http://www.yiiframework.com/doc/api/CConsoleCommand/*
- *http://www.yiiframework.com/doc/api/CConsoleApplication/*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie filtrów

Filtr to klasa, którą można uruchomić przed lub po wykonaniu akcji. Można jej użyć także do zmodyfikowania kontekstu wykonywania lub udekorowania wyniku. W naszym przykładzie zaimplementujemy prosty filtr wyniku, który skompresuje uzyskany kod HTML, usuwając wszystkie opcjonalne elementy formatowania.

> Nie używaj tego filtru w produkcji. GZIP wykona to zadanie lepiej, oszczędzając przepustowość. Ponadto usuwanie formatowania obszaru preformatowanego nie jest bezpieczne.

Przygotowanie

Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.

Jak to zrobić…

1. Utwórz plik *protected/extensions/compress_html/ECompressHtmlFilter.php* i umieść w nim następujący kod:

```
<?php
class ECompressHtmlFilter extends CFilter
{
  protected function preFilter($filterChain)
  {
    ob_start();
    return parent::preFilter($filterChain);
  }

  protected function postFilter($filterChain)
  {
    $out = ob_get_clean();
    echo preg_replace("~>(\s+|\t+|\n+)<~", "><", $out);
    parent::postFilter($filterChain);
  }
}
```

2. To wszystko. Teraz musimy dołączyć filtr do aplikacji. Ponieważ mamy już `SiteController`, dodajmy do niego filtr, nadpisując metodę `filters`:

```
public function filters()
{
  return array(
    array(
```

```
            'ext.compress_html.ECompressHtmlFilter'
        ),
    );
}
```

3. Teraz uruchom aplikację i sprawdź kod źródłowy HTML. Powinien to być pojedynczy wiersz tekstu, taki jak poniżej:

```
<!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" /><meta name="language" content="en" /><!--
blueprint CSS framework -->
…
```

Opis działania

Filtr powinien implementować przynajmniej interfejs `IFilter`, jednak ponieważ chcemy dokonać końcowego i wstępnego filtrowania, możemy bezpośrednio dziedziczyć po klasie `CFilter`. Planujemy uzyskać w pewien sposób zawartość wygenerowaną przez akcję i nieco przetworzyć wynik przed wysłaniem go do przeglądarki. Buforowanie danych wyjściowych PHP świetnie się nadaje do wykonania tego zadania, dlatego nadpisujemy metodę `preFilter`, aby rozpocząć buforowanie poprzez wywołanie w niej metody `ob_start`. Nadpisujemy też metodę `postFilter`, aby uzyskać zawartość bufora przy użyciu `ob_get_clean`, przetworzyć wynik z wykorzystaniem wyrażeń regularnych i go wyświetlić.

Zauważmy, że podczas nadpisywania metod `preFilter` i `postFilter` wywołujemy metody macierzyste. Pozwala to programiście na utworzenie łańcucha filtrów podczas konfigurowania kontrolera.

Dodatkowe informacje

Dodatkowe informacje o filtrach można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/api/CFilter*
- *http://www.yiiframework.com/doc/api/IFilter*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Tworzenie modułów

Jeśli utworzyłeś złożoną część aplikacji i chcesz ją wykorzystać po uprzednim dostosowaniu w następnym projekcie, prawdopodobnie powinieneś utworzyć moduł.

W tej recepturze zobaczysz, jak utworzyć moduł wiki. Dla uproszczenia nie będziemy się skupiać na zarządzaniu użytkownikami i uprawnieniami, lecz pozwolimy wszystkim na nieograniczoną edycję.

Przygotowanie

1. Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.
2. Skonfiguruj bazę danych MySQL i wykonaj następujące zapytanie SQL:

```
CREATE TABLE `wiki` (
  `id` varchar(255) NOT NULL,
  `text` text NOT NULL,
  PRIMARY KEY  (`id`)
)
```

3. Wygeneruj model `Wiki`, korzystając z Gii.
4. Przenieś plik *protected/models/Wiki.php* do folderu *protected/modules/wiki/models/*.
5. Dodaj `wiki` do sekcji `modules` pliku *protected/config/main.php*:

```
'modules'=>array(
  // uncomment the following to enable the Gii tool
  'gii'=>array(
    'class'=>'system.gii.GiiModule',
    'password'=>false,
  ),
  'wiki'
),
```

Jak to zrobić…

Najpierw zaplanujmy działanie:

- **wiki** to zestaw stron, na których można umieszczać łącza do innych stron na podstawie tytułu.
- Zwykle wiki używa prostszej składni, łatwiejszej do odczytu od kodu HTML.
- Jeśli użytkownik przejdzie do strony, która jeszcze nie istnieje, zostanie zachęcony do jej utworzenia.
- Aby usunąć stronę, użytkownik musi zapisać ją wraz z pustą zawartością.

Bazując na powyższych wytycznych, za język znaczników przyjmiemy język markdown. Udostępnimy sposób tworzenia łączy do innych stron na podstawie ich tytułu. Załóżmy, że składnia będzie miała postać *[[nazwa strony]]* lub *[[własny tytuł|nazwa strony]]*, jeśli potrzebne będzie łącze o niestandardowym tekście.

1. Najpierw dodajmy łącza wiki do klasy `CMarkdownParser`. Utwórz plik *protected/modules/wiki/components/WikiMarkdownParser.php*:

```
<?php
class WikiMarkdownParser extends CMarkdownParser
{
  public function transform($text)
  {
    $text = preg_replace_callback('~\[\[(.*?)(?:\|(.*?))?\]\]~', array($this,
       'processWikiLinks'), $text);
    return parent::transform($text);
  }

  protected function processWikiLinks($matches)
  {
    $page = $matches[1];
    $title = isset($matches[2]) ? $matches[2] : $matches[1];
    return CHtml::link(CHtml::encode($title), array('view', 'id' => $page,
    ));
  }
}
```

2. Następnie użyjmy nowej klasy, dodając metodę `getHtml` do modelu *protected/modules/wiki/models/Wiki.php*:

```
public function getHtml()
{
  $parser = new WikiMarkdownParser();
  return $parser->transform($this->text);
}
```

   Przechodzimy do dostosowania kontrolera *protected/modules/wiki/controller/DefaultController.php*. Potrzebne nam będą dwie akcje — `view` i `edit`. Ponadto utworzymy akcję `index`, która po prostu wywoła widok poprzez przypisanie `id = index`. Ponieważ nie potrzebujemy widoku dla akcji `index`, można go bezpiecznie usunąć.

3. Teraz utwórz plik *protected/modules/wiki/controller/DefaultController.php*:

```
class DefaultController extends Controller
{
  public function actionIndex()
  {
    $this->actionView('index');
  }

  public function actionView($id)
```

```
    {
      $model = Wiki::model()->findByPk($id);
      if(!$model)
      {
        $this->actionEdit($id);
        Yii::app()->end();
      }
      $this->render('view', array(
        'model' => $model,
      ));
    }

    public function actionEdit($id)
    {
      $model = Wiki::model()->findByPk($id);
      if(!$model)
      {
        $model = new Wiki();
        $model->id = $id;
      }
      if(!empty($_POST['Wiki']))
      {
        if(!empty($_POST['Wiki']['text']))
        {
          $model->text = $_POST['Wiki']['text'];
          if($model->save())
            $this->redirect(array('view', 'id' => $id));
        }
        else
        {
          Wiki::model()->deleteByPk($id);
        }
      }
      $this->render('edit', array(
        'model' => $model
      ));
    }
}
```

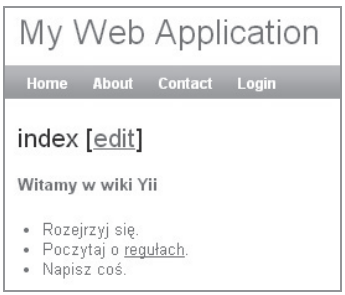
4. Potrzebne nam będą dwa widoki. Utwórz plik *protected/modules/wiki/views/default/view.php*:

```
<h2>
  <?php echo CHtml::encode($model->id)?>
  [<?php echo CHtml::link('edit', array('edit', 'id' => $model->id))?>]
</h2>
<?php echo $model->html ?>
```

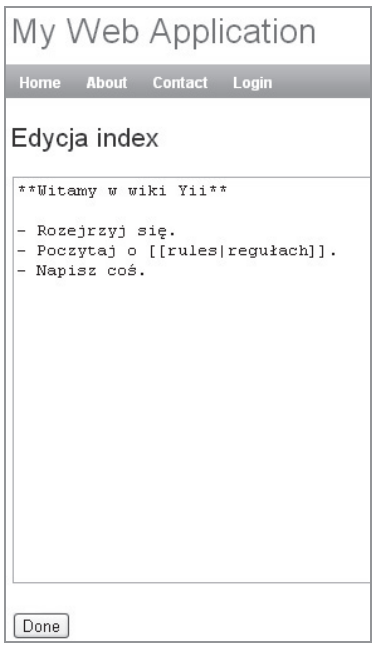
5. Utwórz jeszcze jeden plik *protected/modules/wiki/views/default/edit.php*:

```
<h2>Edycja <?php echo CHtml::encode($model->id)?></h2>
<?php echo CHtml::beginForm()?>
  <?php echo CHtml::activeTextArea($model, 'text',
    array('cols' => 100, 'rows' => 20))?>
  <br /><br />
  <?php echo CHtml::submitButton('Gotowe')?>
<?php echo CHtml::endForm()?>
```

6. To wszystko. Teraz uruchom moduł wiki i sprawdź go, wpisując jakiś tekst. Nie zapomnij dodać wewnętrznego łącza, na przykład *[[rules]]*:



7. Powyższa strona podczas edycji będzie wyglądać następująco:



8. Ponieważ przenieśliśmy model `Wiki` z aplikacji do modułu, model nie posiada żadnych zależności związanych z samą aplikacją. Dlatego możemy go przenieść

do pakietu rozszerzenia, takiego jak *protected/extensions/wiki/*, zmieniając konfigurację modułu *protected/config/main.php* na następującą:

```
'modules'=>array(
  // uncomment the following to enable the Gii tool
  'gii'=>array(
    'class'=>'system.gii.GiiModule',
    'password'=>false,
  ),
  'wiki'=>array(
    'class' => 'ext.wiki.WikiModule'
  ),
),
```

Opis działania

Każdy utworzony moduł zawiera główną klasę modułu, taką jak `WikiModule`, w której można zdefiniować konfigurowalne właściwości czy importowane moduły, zmienić ścieżki, dołączyć kontrolery itd. Domyślnie moduł wygenerowany przez Gii uruchamia akcję `index` kontrolera `default`:

```
public function actionIndex()
{
  $this->actionView('index');
}
```

Na stronie *index* naszego modułu `wiki` wywołujemy akcję `view` poprzez przekazanie do niej `id = index`:

```
$model = Wiki::model()->findByPk($id);
if(!$model)
{
  $this->actionEdit($id);
  Yii::app()->end();
}
$this->render('view', array(
  'model' => $model,
));
```

Jeśli istnieje model o takim identyfikatorze, wyświetlamy go w widoku.

Jeśli nie istnieje żadna strona o takim identyfikatorze, ponownie delegujemy procesowanie do innej akcji. Tym razem jest to akcja `edit`:

```
$model = Wiki::model()->findByPk($id);
if(!$model)
{
  $model = new Wiki();
```

```
    $model->id = $id;
}

if(!empty($_POST['Wiki']))
{
  if(!empty($_POST['Wiki']['text']))
  {
    $model->text = $_POST['Wiki']['text'];
    if($model->save())
      $this->redirect(array('view', 'id' => $id));
  }
  else
  {
    Wiki::model()->deleteByPk($id);
  }
}

$this->render('edit', array(
  'model' => $model
));
```

Jeśli nie istnieje model o przekazanym identyfikatorze, tworzymy nowy; jeśli model istnieje, edytujemy go. Dane formularza edycji pochodzą z metody `POST` i jeśli pole tekstowe formularza jest puste, usuwamy model. Jeśli tekst istnieje, zapisujemy model.

Dodatkowe informacje

W Yii dostępny jest bardzo elastyczny i konfigurowalny moduł wiki o nazwie **Yeeki**. Znajduje się nadal w fazie rozwoju i świetnie nadaje się do nauki tworzenia prawdziwie konfigurowalnych modułów, które można wykorzystać wielokrotnie:

https://github.com/samdark/Yeeki

Dodatkowe informacje o modułach można znaleźć na następujących stronach API:

- *http://www.yiiframework.com/doc/guide/1.1/pl/basics.module*
- *http://www.yiiframework.com/doc/api/CWebModule/*
- *http://www.yiiframework.com/doc/api/CModule/*
- *http://www.yiiframework.com/doc/api/GiiModule/*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji”.

Niestandardowe renderowanie widoków

Na rynku dostępnych jest wiele silników szablonów PHP. Yii wspiera jedynie dwa rodzaje szablonów: **natywne PHP** i szablony **w stylu Prado**. Jeśli chcemy użyć jednego z istniejących silników szablonów lub utworzyć własny, musimy go zaimplementować, o ile nie zostało to jeszcze zrobione przez społeczność Yii.

W tej recepturze zaimplementujemy wsparcie dla szablonów Smarty.

Przygotowanie

1. Utwórz nową aplikację Yii, korzystając z polecenia `yiic webapp`.
2. Pobierz ostatnią wersję Smarty 3 ze strony *http://www.smarty.net/*.
3. Wyodrębnij zawartość folderu *libs* do folderu *protected/vendors/smarty/*.

Jak to zrobić…

1. Utwórz plik *protected/extensions/smarty/ESmartyViewRenderer.php* i umieść w nim następujący kod:

```
<?php
class ESmartyViewRenderer extends CApplicationComponent implements
    IViewRenderer
{
    public $fileExtension='.tpl';
    public $filePermission=0755;

    private $smarty;

    function init()
  {
        Yii::import('application.vendors.smarty.*');

        spl_autoload_unregister(array('YiiBase','autoload'));
        require_once('Smarty.class.php');
        spl_autoload_register(array('YiiBase','autoload'));
        $this->smarty = new Smarty();
        $this->smarty->template_dir = '';
        $compileDir = Yii::app()->getRuntimePath().'/smarty/compiled/';

        if(!file_exists($compileDir)){
            mkdir($compileDir, $this->filePermission, true);
        }
        $this->smarty->compile_dir = $compileDir;
```

```
        $this->smarty->assign('Yii', Yii::app());
    }

    /**
    * Renderuje plik widoku.
    * Ta metoda jest wymagana przez {@link IViewRenderer}.
    * @param CBaseController kontroler lub widżet, który renderuje plik widoku.
    * @param string ścieżka pliku widoku
    * @param mieszany, dane przekazywane do widoku
    * @param boolean, czy należy zwrócić wynik renderowania
    * @return mieszany wynik renderowania lub null, jeśli wynik renderowania jest niepotrzebny.
    */
  public function renderFile($context,$sourceFile,$data,$return) {
        // właściwości bieżącego kontrolera będą dostępne w postaci {this.property}
        $data['this'] = $context;
        if(!is_file($sourceFile) || ($file=realpath($sourceFile))==false)
            throw new CException(Yii::t('ext','Plik widoku "$sourceFile" nie
               istnieje.', array('{file}'=>$sourceFile)));
    $this->smarty->assign($data);
    if($return)
          return $this->smarty->fetch($sourceFile);
    else
      $this->smarty->display($sourceFile);
  }
}
```

2. Następnie musimy dołączyć do aplikacji nową klasę renderującą widok. W tym celu należy nadpisać komponent `viewRenderer` w pliku *protected/config/main.php*:

```
...
// application components
'components'=>array(
  ...
  'viewRenderer'=>array(
    'class'=>'ext.smarty.ESmartyViewRenderer',
  ),
),
...
```

3. Teraz możemy przetestować nasze rozwiązanie. Utwórz plik *protected/controllers/SmartyController.php* i umieść w nim następujący kod:

```
<?php
class SmartyController extends Controller
{
  function actionNative()
  {
    $this->render('native', array(
      'username' => 'Aleksandrze',
    ));
```

```
    }

    function actionSmarty()
    {
      $this->render('smarty', array(
        'username' => 'Aleksandrze',
      ));
    }
}
```

4. Potrzebne są również widoki. Utwórz plik *protected/views/smarty/native.php*:

```
Witaj, <?php echo $username?>!
protected/views/smarty/smarty.tpl:
Witaj, {$username}!
```

5. Spróbujmy teraz uruchomić akcje kontrolera. W obydwu przypadkach powinniśmy uzyskać napis **Witaj, Aleksandrze!**.

Opis działania

Klasa renderująca widok jest potomkiem klasy `CApplicationComponent`, która implementuje interfejs `IViewRenderer`, zawierający tylko jedną metodę o nazwie `renderFile`:

```
/**
    * Renderuje plik widoku.
    * Ta metoda jest wymagana przez {@link IViewRenderer}.
    * @param CBaseController kontroler lub widżet, który renderuje plik widoku.
    * @param string ścieżka pliku widoku
    * @param mieszany dane przekazywane do widoku
    * @param boolean, czy należy zwrócić wynik renderowania
    * @return mieszany wynik renderowania lub null, jeśli wynik renderowania jest niepotrzebny.
    */
public function renderFile($context,$file,$data,$return);
```

W ten sposób uzyskujemy kontekst, ścieżkę szablonu, dane i flagę, która określa, czy musimy wyświetlić wynik przetwarzania natychmiast, czy tylko go zwrócić. W naszym przypadku przetwarzanie odbywa się w silniku szablonów Smarty, dlatego musimy go właściwie zainicjalizować i wywołać jego metody przetwarzania.

Importowanie i inicjalizacja Smarty nie są zbyt łatwe:

```
Yii::import('application.vendors.smarty.*');

spl_autoload_unregister(array('YiiBase','autoload'));
require_once('Smarty.class.php');
spl_autoload_register(array('YiiBase','autoload'));

$this->smarty = new Smarty();
```

```
$this->smarty->template_dir = '';
$compileDir = Yii::app()->getRuntimePath().'/smarty/compiled/';

if(!file_exists($compileDir)){
  mkdir($compileDir, $this->filePermission, true);
}

$this->smarty->compile_dir = $compileDir;
$this->smarty->assign('Yii', Yii::app());
```

Najpierw używamy instrukcji `Yii::import`, aby przekazać Yii, że chcemy użyć klas z folderu *protected/vendors/smarty/*. Ponieważ mechanizm automatycznego ładowania Yii pozostaje w konflikcie z podobnym mechanizmem zawartym w Smarty i ponieważ nie możemy kontrolować tego, co się dzieje w Smarty, musimy wyrejestrować wspomniany mechanizm Yii i dołączyć Smarty. Dopiero wtedy możemy ponownie zarejestrować mechanizm automatycznego ładowania Yii. Ponieważ folder zawierający szablony Yii nie jest stały, zmieniamy domyślną opcję Smarty na pusty łańcuch. Dobrą praktyką jest przechowywanie tymczasowych plików Yii w folderze `appliaction`. To dlatego ustawiamy folder kompilacji, w którym zostaną umieszczone szablony Smarty skompilowane do PHP, na `runtime/smarty/compiled`. Aby nie przeszkadzać zbytnio programistom, tworzymy ten folder automatycznie, o ile jeszcze nie istnieje. Ponadto tworzymy specjalną zmienną szablonu Smarty, zwaną `Yii`, do której przypisujemy `Yii::app()`. Zmienna ta pozwala nam uzyskać właściwości aplikacji wewnątrz szablonu.

Samo renderowanie jest nieco prostsze:

```
// właściwości bieżącego kontrolera będą dostępne w postaci {this.property}
$data['this'] = $context;

if(!is_file($sourceFile) || ($file=realpath($sourceFile))===false)
  throw new CException(Yii::t('ext','View file "$sourceFile" does not
exist.', array('{file}'=>$sourceFile)));
$this->smarty->assign($data);

if($return)
  return $this->smarty->fetch($sourceFile);
else
  $this->smarty->display($sourceFile);
```

Deklarujemy kolejną zmienną Smarty, zwaną `this`, która pozwoli uzyskać właściwości kontrolera, takie jak tytuł strony.

Aby ułatwić debugowanie, sprawdzamy, czy szablon istnieje, a jeśli nie, wyświetlany jest komunikat błędu z nazwą szablonu.

Cały zestaw danych jest przekazywany bez zmian do szablonu Smarty poprzez wywołanie `$this->render`. Następnie renderujemy szablon lub go zwracamy — w zależności od parametru `$return`.

Dodatkowe informacje

Dostępny na stronie *http://www.yiiframework.com/extension/smarty-view-renderer* plugin i wskazówki dotyczące konfiguracji ułatwią korzystanie z renderowania widoków Smarty.

> W aplikacji możemy używać własnych silników szablonów. Jeśli jednak tworzymy rozszerzenie wielokrotnego użytku, spróbujmy unikać silników szablonów innych niż natywne PHP.

Warto przeczytać

Dodatkowe informacje o Smarty i o renderowaniu widoków można znaleźć na następujących stronach:

- *http://www.smarty.net/*
- *http://www.yiiframework.com/doc/api/IViewRenderer/*
- *http://www.yiiframework.com/doc/api/CViewRenderer/*
- *http://www.yiiframework.com/doc/api/CPradoViewRenderer/*

Zobacz też

- Receptura „Przygotowywanie rozszerzeń do dystrybucji".

Przygotowywanie rozszerzeń do dystrybucji

W tym rozdziale nauczyłeś się, jak należy tworzyć różne rodzaje rozszerzeń Yii. Teraz dowiesz się, jak udostępnić swoje wyniki innym osobom i dlaczego to takie ważne.

Przygotowanie

Przygotujmy najpierw listę kontrolną dla dobrego rozszerzenia. Dobry produkt programistyczny powinien spełniać następujące warunki:

- Spójne i łatwe do odczytu i użycia API.
- Dobra dokumentacja.
- Łatwy do znalezienia.
- Rozszerzenie powinno być dostosowane do większości przypadków użycia.
- Rozszerzenie powinno być utrzymywane.

- Dobrze przetestowany kod; w idealnej sytuacji powinien zawierać testy jednostkowe.
- Zapewnione wsparcie dla rozszerzenia.

Oczywiście spełnienie wszystkich tych wymagań wymaga wiele pracy, jednak jest potrzebne do uzyskania dobrego produktu.

Jak to zrobić…

1. Przyjrzyj się dokładnie naszej liście, począwszy od API. API powinno być spójne oraz łatwe do odczytu i użycia. Spójność oznacza, że ogólny styl nie powinien się zmieniać. Dlatego nie powinniśmy wprowadzać różnych konwencji nazewnictwa, na przykład `camelCaasedVariableNames` i `underscored_variable_names`; niespójnych nazw, na przykład `isFlag1()` i `isNotFlag2()`; itd. Wszędzie należy przestrzegać określonych dla kodu reguł. Pozwala to ograniczyć konieczność sprawdzania dokumentacji i skupić się na kodzie.
2. Kod bez żadnej dokumentacji jest niemal nieprzydatny. Wyjątkiem jest dość prosty kod, jednak nawet jeśli kod liczy jedynie kilka wierszy, pozostawienie go bez żadnej wzmianki o instalacji i użyciu nie jest właściwe. Co składa się na dobrą dokumentację? Cele kodu i jego zalety powinny być możliwie jasne i powinny być napisane wyraźnie i przejrzyście. Jeśli na przykład rozszerzenie wysyła wiadomość e-mail, opis powinien być podobny do poniższego:

   EMailer pozwala na przesyłanie wiadomości e-mail w prosty i wygodny sposób. Prawidłowo obsługuje tekst i tytuł w języku angielskim. Ponadto pozwala na użycie widoków Yii jako szablonów wiadomości e-mail.

   Kod jest bezużyteczny, jeśli programiści nie wiedzą, gdzie go umieścić i co powinno się znaleźć w konfiguracji aplikacji. Nie oczekuj, że programiści będą wiedzieć, jak wykonywać zadania specyficzne dla platformy. Wskazówki instalacji powinny być wyczerpujące. Większość programistów preferuje wskazówki krok po kroku. Jeśli kod do swojego działania wymaga schematu SQL, należy go dołączyć.

   Nawet jeśli metody i właściwości naszego API mają odpowiednie nazwy, nadal powinniśmy je udokumentować w komentarzach phpDoc, określając typy argumentów i zwracanych elementów oraz zapewniając krótki opis dla każdej metody. Ponadto należy rozważyć wypisanie w dokumentacji listy metod i właściwości publicznych, aby uzyskać zestaw odsyłaczy.

   Obok dobrze skomentowanego kodu należy udostępnić przykładowe przypadki użycia. Spróbuj uwzględnić większość popularnych sposobów użycia rozszerzeń. W przykładzie postaraj się nie rozwiązywać wielu problemów jednocześnie, ponieważ może to być mylące.
3. Rozszerzenie powinno posiadać numer wersji i rejestr zmian. Pozwoli to społeczności na sprawdzenie, czy posiadana wersja jest aktualna, oraz na zapoznanie się z ostatnimi zmianami przed przystąpieniem do aktualizacji.

Ważne jest, aby kod był elastyczny i aby można go było wykorzystać w wielu przypadkach. Ponieważ jednak nie ma możliwości utworzenia kodu dla każdego możliwego przypadku użycia, spróbuj uwzględnić większość tych popularnych.

4. Ważne jest, aby programiści czuli się wygodnie z rozszerzeniem. Dostarczenie dobrej dokumentacji jest pierwszym krokiem. Drugi to udostępnienie dowodu, że kod działa zgodnie z zamierzeniem i że będzie działać wraz z przyszłymi aktualizacjami. Najlepszym sposobem, aby to osiągnąć, jest dołączenie zestawu testów jednostkowych.
5. Rozszerzenie należy utrzymywać przynajmniej do osiągnięcia stabilności i upewnienia się, że nie ma już żadnych propozycji funkcji i raportów o błędach. Dlatego należy się spodziewać pytań i raportów. Trzeba też zarezerwować w przyszłości nieco czasu na pracę nad kodem. Jeśli nie możemy poświęcić więcej czasu na utrzymanie rozszerzenia, które jest jednak bardzo innowacyjne, i nikt wcześniej nie zaproponował niczego podobnego, nadal warto je udostępnić. Jeśli społeczność doceni rozszerzenie, ktoś z pewnością zaoferuje swoją pomoc.
6. Musimy wreszcie udostępnić rozszerzenie. Popularnym miejscem publikacji rozszerzenia Yii jest strona *http://www.yiiframework.com/extensions/*. Zanim będzie można przesłać własny kod, należy się zarejestrować na forum i przesłać kilka postów. Spróbuj znaleźć dobrą opisową nazwę, napisać opisowe podsumowanie i określić odpowiednie kategorie i znaczniki. Nie używaj archiwów innych niż ZIP lub GZIP, ponieważ są to najpopularniejsze formaty archiwów i możemy być pewni, że każdy będzie mógł je rozpakować.
7. Nawet jeśli rozszerzenie jest dość proste, a dokumentacja dobra, możemy się spodziewać pytań i, przynajmniej na początku, jedynym człowiekiem, który może na nie odpowiedzieć, będziesz Ty. Zwykle pytania są zadawane na oficjalnych forach, dlatego lepiej jest utworzyć nowy wątek, na którym programiści mogą prowadzić dyskusję o kodzie, oraz podać łącze do strony rozszerzenia.

Opis działania

Jeśli chcesz udostępnić rozszerzenie społeczności i upewnić się, że będzie użyteczne i popularne, musisz dostarczyć nie tylko sam kod. Przygotowanie rozszerzeń do dystrybucji wymaga nieco więcej pracy. Może to być nawet więcej niż samo tworzenie rozszerzenia. Dlaczego więc warto udostępnić rozszerzenie społeczności? W porównaniu z kodem używanym we własnych projektach otwarte projekty mają swoje zalety. Możemy wykorzystać do przetestowania gotowych źródeł projektu o wiele więcej osób, niż moglibyśmy znaleźć osobiście. Osoby korzystające z rozszerzenia testują je, dostarczając wartościowych opinii i raportując błędy. Jeśli kod jest popularny, znajdą się programiści pasjonaci, którzy będą chcieli usprawniać kod, aby uczynić go bardziej kompleksowym, stabilnym i zdatnym do wielokrotnego użytku. Ponadto sprawi nam to przyjemność, ponieważ robimy dobry uczynek.

Dodatkowe informacje

Omówiono już większość ważnych zagadnień. Jednak nadal mamy wiele do sprawdzenia. Zanim napiszesz własne rozszerzenia, wypróbuj istniejące. Jeśli rozszerzenie spełnia prawie wszystkie wymagania, spróbuj skontaktować się z autorem rozszerzenia i podzielić się swoimi pomysłami. Przeglądanie istniejącego kodu pomaga sprawdzić przydatne sztuczki i zasady. Przeglądaj też od czasu do czasu artykuły wiki i oficjalne fora; dostępnych jest wiele przydatnych informacji o tworzeniu rozszerzeń i ogólnie programowaniu z wykorzystaniem Yii.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Y

Z

Ż

Tworzenie aplikacji z Yii
Receptury

PHP to jeden z języków programowania najczęściej wybieranych do tworzenia stron internetowych i aplikacji webowych. Yii to platforma MVC napisana w języku PHP, która sprawi, że Twój kod będzie bardziej przejrzysty, a osiągnięcie założonych celów — znacznie prostsze. Jeżeli połączysz potencjał PHP ze wsparciem Yii, otrzymasz kombajn, który pozwoli Ci w okamgnieniu poradzić sobie z dowolnym problemem.

Książka ta należy do cenionej przez programistów serii „Receptury". Znajdziesz w niej najczęściej spotykane problemy wraz z ich najlepszymi rozwiązaniami. W trakcie lektury dowiesz się, jak skonfigurować reguły URL, przekazywać wiadomości pomiędzy żądaniami oraz stosować widoki częściowe. Ponadto poznasz najlepsze przepisy na wykorzystanie technologii AJAX oraz biblioteki jQuery wspólnie ze szkieletem Yii. Jeżeli stoisz przed problemem zabezpieczenia aplikacji przed spamem i chciałbyś wykorzystać mechanizm CAPTCHA, to znajdziesz tu szczegółową instrukcję, jak to zrobić. Co jeszcze zawiera ta książka? Najlepsze porady dotyczące wydajności, współpracy z bazą i bezpieczeństwa. Jest to lektura obowiązkowa dla każdego programisty chcącego wykorzystać możliwości Yii w swojej aplikacji!

Dzięki tej książce:

- zabezpieczysz formularz przy użyciu CAPTCHA
- wyciśniesz siódme poty ze szkieletu Yii
- zwiększysz bezpieczeństwo Twojej aplikacji
- zbudujesz działającą aplikację szybko i bez problemów

Poznaj możliwości szkieletu Yii!

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

ISBN 978-83-246-8596-7

9 788324 685967

Cena: 67,00 zł