Creating model behaviors

There are many similar solutions in today’s web applications. Leading products such as Google’s Gmail are  
defining nice UI patterns. One of these is soft delete. Instead of a permanent deletion with tons of  
confirmations, Gmail allows us to immediately mark messages as deleted and then easily undo it. The  
same behavior can be applied to any object such as blog posts, comments, and so on.

Let’s create a behavior that will allow marking models as deleted, restoring models, selecting not yet  
deleted models, deleted models, and all models. In this recipe, we’ll follow a test-driven development  
approach to plan the behavior and test if the implementation is correct.

Подготовка

1. Создайте новое приложение с помощью диспетчера пакетов Composer, как описано в официальном руководстве по адресу  
   <http://www.yiiframework.com/doc-2.0/guide-start-installation.html>.   
   По русски <http://yiiframework.domain-na.me/doc/guide/2.0/ru/start-installation>.
2. Create two databases for working and for tests.
3. Configure Yii to use the first database in your primary application in config/db. php. Make sure the  
   test application uses the second database in tests/codeception/config/config. php.
4. Create a new migration:

<?php

use yii\db\Migration;

class m160427\_103115\_create\_post\_table extends Migration  
{

public function up()

{

$this->createTable('{{%post}}', [

'id' => $this->primaryKey(),

'title' => $this->string()->notNull(),

'content\_markdown' => $this->text(),

'content\_html' => $this->text(),

]);

}

public function down()

{

$this->dropTable('{{%post}}');

}

}

1. Apply the migration to both the working and test databases:

./yii migrate

tests/codeception/bin/yii migrate

1. Create the Post model:

<?php

namespace app\models;

use app\behaviors\MarkdownBehavior;  
use yii\db\ActiveRecord;

/\*\*

* @property integer $id
* @property string $title
* @property string $content\_markdown
* @property string $content\_html  
  \*/

class Post extends ActiveRecord  
{

public static function tableName()

{

return '{{%post}}';

}

public function rules()

{

return [

[['title'], 'required'],

[['content\_markdown'], 'string'],

[['title'], 'string', 'max' => 255],

];

}

}

How to do it...

Let’s prepare a test environment first starting with defining fixtures for the Post model. Create the  
tests/codeception/unit/fixtures/PostFixture .php file:

<?php

namespace app\tests\codeception\unit\fixtures;

use yii\test\ActiveFixture;

class PostFixture extends ActiveFixture  
{

public $modelClass = 'app\models\Post';

public $dataFile = '@tests/codeception/unit/fixtures/data/post.php';

}

1. Add a fixture data file to tests/codeception/unit/fixtures/data/post. php:

<?php  
return [

[

'id' => 1,

'title' => 'Post 1',

'content\_markdown' => 'Stored \*markdown\* text 1',

'content\_html' => "<p>Stored <em>markdown</em> text 1</p>\n",

],

];

1. Then, we need to create a test case, tests/codeception/unit/MarkdownBehaviorTest. php:  
   <?php

namespace app\tests\codeception\unit;  
use app\models\Post;

use app\tests\codeception\unit\fixtures\PostFixture;  
use yii\codeception\DbTestCase;

class MarkdownBehaviorTest extends DbTestCase  
{

public function testNewModelSave()

{

$post = new Post();

$post->title = 'Title';

$post->content\_markdown = 'New \*markdown\* text';  
$this->assertTrue($post->save());

$this->assertEquals("<p>New <em>markdown</em> text</p>\n", $post-  
>content\_html);

}

public function testExistingModelSave()

{

$post = Post::findOne(1);

$post->content\_markdown = 'Other \*markdown\* text';  
$this->assertTrue($post->save());

$this->assertEquals("<p>Other <em>markdown</em> text</p>\n", $post-  
>content\_html);

}

public function fixtures()

{

return [

'posts' => [

'class' => PostFixture::className(),

]

];

}

}

1. Run the unit tests:

codecept run unit MarkdownBehaviorTest  
Ensure that tests has not passed:

Codeception PHP Testing Framework v2.0.9

Powered by PHPUnit 4.8.27 by Sebastian Bergmann and contributors.

Unit Tests (2)

Trying to test... MarkdownBehaviorTest::testNewModelSave Error

Trying to test. MarkdownBehaviorTest::testExistingModelSave Error

Time: 289 ms, Memory: 16.75MB

1. Now we need to implement behavior, attach it to the model, and make sure the test passes. Create  
   a new directory, behaviors. Under this directory, create a MarkdownBehavior class:

<?php

namespace app\behaviors;

use yii\base\Behavior;  
use yii\base\Event;

use yii\base\InvalidConfigException;  
use yii\db\ActiveRecord;  
use yii\helpers\Markdown;

class MarkdownBehavior extends Behavior  
{

public $sourceAttribute;  
public $targetAttribute;

public function init()

{

if (empty($this->sourceAttribute) || empty($this->targetAttribute)) {  
throw new InvalidConfigException('Source and target must be set.');

}

parent::init();

}

public function events()

{

return [

ActiveRecord::EVENT\_BEFORE\_INSERT => 'onBeforeSave',

ActiveRecord::EVENT\_BEFORE\_UPDATE => 'onBeforeSave',

];

}

public function onBeforeSave(Event $event)

{

if ($this->owner->isAttributeChanged($this->sourceAttribute)) {  
$this->processContent();

}

}

private function processContent()

{

$model = $this->owner;

$source = $model->{$this->sourceAttribute};

$model->{$this->targetAttribute} = Markdown::process($source);

}

}

1. Let’s attach the behavior to the Post model:

class Post extends ActiveRecord  
{

public function behaviors()

{

return [

'markdown' => [

'class' => MarkdownBehavior::className(),

'sourceAttribute' => 'content\_markdown',

'targetAttribute' => 'content\_html',

],

];

}

}

1. Run the test and make sure it passes:

Codeception PHP Testing Framework v2.0.9

Powered by PHPUnit 4.8.27 by Sebastian Bergmann and contributors.

Unit Tests (2)

Trying to test... MarkdownBehaviorTest::testNewModelSave Ok

Trying to test. MarkdownBehaviorTest::testExistingModelSave Ok

Time: 329 ms, Memory: 17.00MB

1. That’s it. We’ve created a reusable behavior and can use it for all future projects by just connecting  
   it to a model.

How it works...

Let’s start with the test case. Since we want to use a set of models, we are defining fixtures. A fixture set  
is put into the “database” each time the test method is executed.

We prepare unit tests for specifying how the behavior must work:

* First, we are testing a processing of a new model content. The behavior must convert the  
  Markdown text from the source attribute to HTML and store the second one to the target attribute.
* Second, we are testing to update the content of the existing model. After changing the Markdown  
  content and saving the model, we must get the updated HTML content.

Now let’s move to the interesting implementation details. In behavior, we can add our own methods, which  
will be mixed into the model that the behavior is attached to. Also, we can subscribe to the owner  
component events. We are using it to add an own listener:

public function events()

{

return [

ActiveRecord::EVENT\_BEFORE\_INSERT => 'onBeforeSave',

ActiveRecord::EVENT\_BEFORE\_UPDATE => 'onBeforeSave',

];

}

Now we can implement this listener:

public function onBeforeSave(Event $event)

{

if ($this->owner->isAttributeChanged($this->sourceAttribute))

{

$this->processContent();

}

}

In all the methods, we can use the owner property to get the object the behavior is attached to. In general,  
we can attach any behavior to our models, controllers, applications, and other components that extend the  
yii\base\Component class. Also, we can attach one behavior repeatedly to the model for processing  
different attributes:

class Post extends ActiveRecord  
{

public function behaviors()

{

return [

[

'class' => MarkdownBehavior::className(),

'sourceAttribute' => 'description\_markdown',

'targetAttribute' => 'description\_html',

],

[

'class' => MarkdownBehavior::className(),

'sourceAttribute' => 'content\_markdown',

'targetAttribute' => 'content\_html',

],

];

}

}

Besides, we can extend the yii\base\AttributeBehavior class like  
yii\behaviors\TimestampBehavior for updating specified attributes for any events.

See also

To learn more about behaviors and events, refer to the following pages:

* <http://www.yiiframework.com/doc-2.0/guide-concept-behaviors.html>
* <http://www.yiiframework.com/doc-2.0/guide-concept-events.html>

For more information about the Markdown syntax, refer to [http://daringfireball net/proje cts/ma rkdown/](http://daringfireball.net/projects/markdown/).  
Also, refer to the Making extensions distribution-ready recipe of this chapter.