Creating components

If you have some code that looks like it can be reused but you don’t know if it’s a behavior, widget, or  
something else, most probably it’s a component. A component should be inherited from the  
yii\base\component class. Later on, the component can be attached to the application and configured  
using the components section of the configuration file. That’s the main advantage compared with using just  
a plain PHP class. Additionally, we are getting behavior, event, getter, and setter support.

For our example, we’ll implement a simple Exchange application component that will be able to get  
currency rates from the <http://fixer.io> site, attach it to the application, and use it.

Подготовка

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>.

How to do it...

For getting the currency rates, our component should send a HTTP GET query to a service URL such as  
<http://api.fixer.io/2016-05-14?base=USD>.

The service must return all supported rates on the nearest working day:

{

"base":"USD",

"date":"2016-05-13",

"rates": {

"AUD":1.3728,

"BGN":1.7235,

"ZAR":15.168,

"EUR":0.88121

}

}

The component should extract needle currency from the response in JSON format and return a target  
rate:

1. Create the components directory in your application structure.
2. Create the component class example with the following interface:

<?php

namespace app\components;

use yii\base\Component;

class Exchange extends Component  
{

public function getRate($source, $destination, $date = null)

{

}

}

3. Implement the component functional:

<?php

namespace app\components;

use yii\base\Component;

use yii\base\InvalidConfigException;

use yii\base\InvalidParamException;

use yii\caching\Cache;

use yii\di\Instance;

use yii\helpers\Json;

class Exchange extends Component  
{

/\*\*

* @var string remote host  
  \*/

public $host = 'http://api.fixer.io';

/\*\*

* @var bool cache results or not  
  \*/

public $enableCaching = false;

/\*\*

* @var string|Cache component ID

\*/

public $cache = 'cache';

public function init()

{

if (empty($this->host)) {

throw new InvalidConfigException('Host must be set.');

}

if ($this->enableCaching) {

$this->cache = Instance::ensure($this->cache, Cache::className());

}

parent::init();

}

public function getRate($source, $destination, $date = null)

{

$this->validateCurrency($source);

$this->validateCurrency($destination);

$date = $this->validateDate($date);

$cacheKey = $this->generateCacheKey($source, $destination, $date);  
if (!$this->enableCaching || ($result = $this->cache->get($cacheKey)) ===

false) {

$result = $this->getRemoteRate($source, $destination, $date);  
if ($this->enableCaching) {

$this->cache->set($cacheKey, $result);

}

}

return $result;

}

private function getRemoteRate($source, $destination, $date)

{

$url = $this->host . '/' . $date . '?base=' . $source;

$response = Json::decode(file\_get\_contents($url));  
if (!isset($response['rates'][$destination])) {

throw new \RuntimeException('Rate not found.');

}

return $response['rates'][$destination];

}

private function validateCurrency($source)

{

if (!preg\_match('#A[A-Z]{3}$#s', $source)) {

throw new InvalidParamException('Invalid currency format.');

}

}

private function validateDate($date)

{

if (!empty($date) && !preg\_match('#\d{4}\-\d{2}-\d{2}#s', $date)) {  
throw new InvalidParamException('Invalid date format.');

}

if (empty($date)) {

$date = date('Y-m-d');

}

return $date;

}

private function generateCacheKey($source, $destination, $date)

{

return [ CLASS , $source, $destination, $date];

}

}

1. Attach the component to your config/console .php or config/web. php configuration files:

'components' => [

'cache' => [

'class' => 'yii\caching\FileCache',

],

'exchange' => [

'class' => 'app\components\Exchange',

'enableCaching' => true,

],

// ...

db' => $db,

],

1. Right now, we can use a new component directly or via the get method:  
   echo \Yii::$app->exchange->getRate('USD', 'EUR');

echo \Yii::$app->get('exchange')->getRate('USD', 'EUR', '2014-04-12');

1. Create a demonstration console controller:

<?php

namespace app\commands;

use yii\console\Controller;

class ExchangeController extends Controller  
{

public function actionTest($currency, $date = null)

{

echo \Yii::$app->exchange->getRate('USD', $currency, $date) . PHP\_EOL;

}

}

1. Now try to run any command:

$ ./yii exchange/test EUR

* 0.90196

$ ./yii exchange/test EUR 2015-11-24

* 0.93888

$ ./yii exchange/test OTHER

* Exception 'yii\base\InvalidParamException' with message 'Invalid currency  
  format.'

$ ./yii exchange/test EUR 2015/24/11

Exception 'yii\base\InvalidParamException' with message 'Invalid date format.'

$ ./yii exchange/test ASD

* Exception 'RuntimeException' with message 'Rate not found.'

As a result, you must see the rate values in the success cases or specific exceptions in the error ones.  
Besides creating your own components, you can do more.

Overriding existing application components

Most of the time, there will be no need to create your own application components since other types of  
extension such as widgets or behaviors, cover almost all types of reusable codes. However, overriding  
core framework components is a common practice and can be used to customize the framework’s  
behavior for your specific needs without hacking into the core.

For example, to be able to format numbers using the Yii: : app()->formatter->asNumber($value)  
method instead of our NumberHelper: : format method from the Creating helpers recipe, you can follow  
the next steps:

1. Extend the yii\i18n\Formatter component as follows:

<?php

namespace app\components;

class Formatter extends \yii\i18n\Formatter  
{

public function asNumber($value, $decimal = 2)

{

return number\_format($value, $decimal, '.', ',');

}

}

1. Override the class of the built-in formatter component:

'components' => [

// ...

formatter => [

'class' => 'app\components\Formatter,

],

// ...

],

1. Right now, we can use this method directly:

echo Yii::app()->formatter->asNumber(1534635.2, 3);

Alternatively, it can be used as a new format for the GridView and DetailView widgets:

<?= \yii\grid\GridView::widget([

'dataProvider' => $dataProvider,

'columns' => [

'id',

'created\_at:datetime',

'title',

'value:number',

],

]) ?>

4. Also, you can extend every existing component without overwriting its source code.

How it works...

To be able to attach a component to an application, it can be extended from the yii\base\component  
class. Attaching is as simple as adding a new array to the components section of configuration. There, a  
class value specifies the component’s class, and all other values are set to a component through the  
corresponding component’s public properties and setter methods.

Implementation itself is very straightforward; we are wrapping the <http://api.fixer.io>calls into a  
comfortable API with validators and caching. We can access our class by its component name using  
Yii : : $app. In our case, it will be Yii: :$app->exchange.

See also

[For official information about components, refer to http://www.yiiframework.com/doc-2.0/guide-concept-](http://www.yiiframework.com/doc-2.0/guide-concept-components.html)  
components.html.

For the NumberHelper class sources, refer to the Creating helpers recipe.