Using different log routes

Logging is the key to understanding what your application actually does when you have no chance to  
debug it. Believe it or not, even if you are 100% sure that the application will behave as expected, in  
production, it can do many things you were not aware of. This is OK, as no one can be aware of  
everything. Therefore, if we are expecting unusual behavior, we need to know about it as soon as possible  
and have enough details to reproduce it. This is where logging comes in handy.

Yii allows a developer not only to log messages but also to handle them differently depending on the  
message level and category. You can, for example, write a message to the database, send an e-mail, or  
just show it in the browser.

In this recipe, we will handle log messages in a wise manner: the most important message will be sent  
through an e-mail, less important messages will be saved in files A and B, and the profiling will be routed  
to Firebug. Additionally, in a development mode, all messages and profiling information will be displayed on  
the screen.

Getting ready

Create a new yii2 -app-basic application by using the Composer package manager, as described in the  
official guide at <http://www.yiiframework.com/doc-2.0/guide-start-installation.html>.

How to do it...

Carry out the following steps:

1. Configure logging using config/web .php:

'components' => [

'log' => [

'traceLevel' => 0,

'targets' => [

[

'class' => 'yii\log\EmailTarget',

'categories' => ['example'],

'levels' => ['error'],

'message' => [

'from' => ['log@example.com'],

'to' => ['developer1@example.com', 'developer2@example.com'],  
'subject' => 'Log message',

],

],

[

'class' => 'yii\log\FileTarget',

'levels' => ['error'],

'logFile' => '@runtime/logs/error.log',

],

[

'class' => 'yii\log\FileTarget',

'levels' => ['warning'],

'logFile' => '@runtime/logs/warning.log',

],

[

'class' => 'yii\log\FileTarget',

'levels' => ['info'],

'logFile' => '@runtime/logs/info.log',

],

],

],

'db' => require( DIR . '/db.php'),

],

1. Now, we win produce a few log messages in protected/controllers/LogController . php as  
   follows:

<?php

namespace app\controllers;

use yii\web\Controller;  
use Yii;

class LogController extends Controller  
{

public function actionIndex()

{

Yii::trace('example trace message', 'example');

Yii::info('info', 'example');

Yii::error('error', 'example');

Yii::trace('trace', 'example');

Yii::warning('warning','example');

Yii::beginProfile('preg\_replace', 'example');

for($i=0;$i<10000;$i++){

preg\_replace('~A[ a-z]+~', '', 'test it');

}

Yii:: endProfile('preg\_replace', 'example');

return $this->render('index');

}

}

and view views/log/index.php:

<div class="log-index">

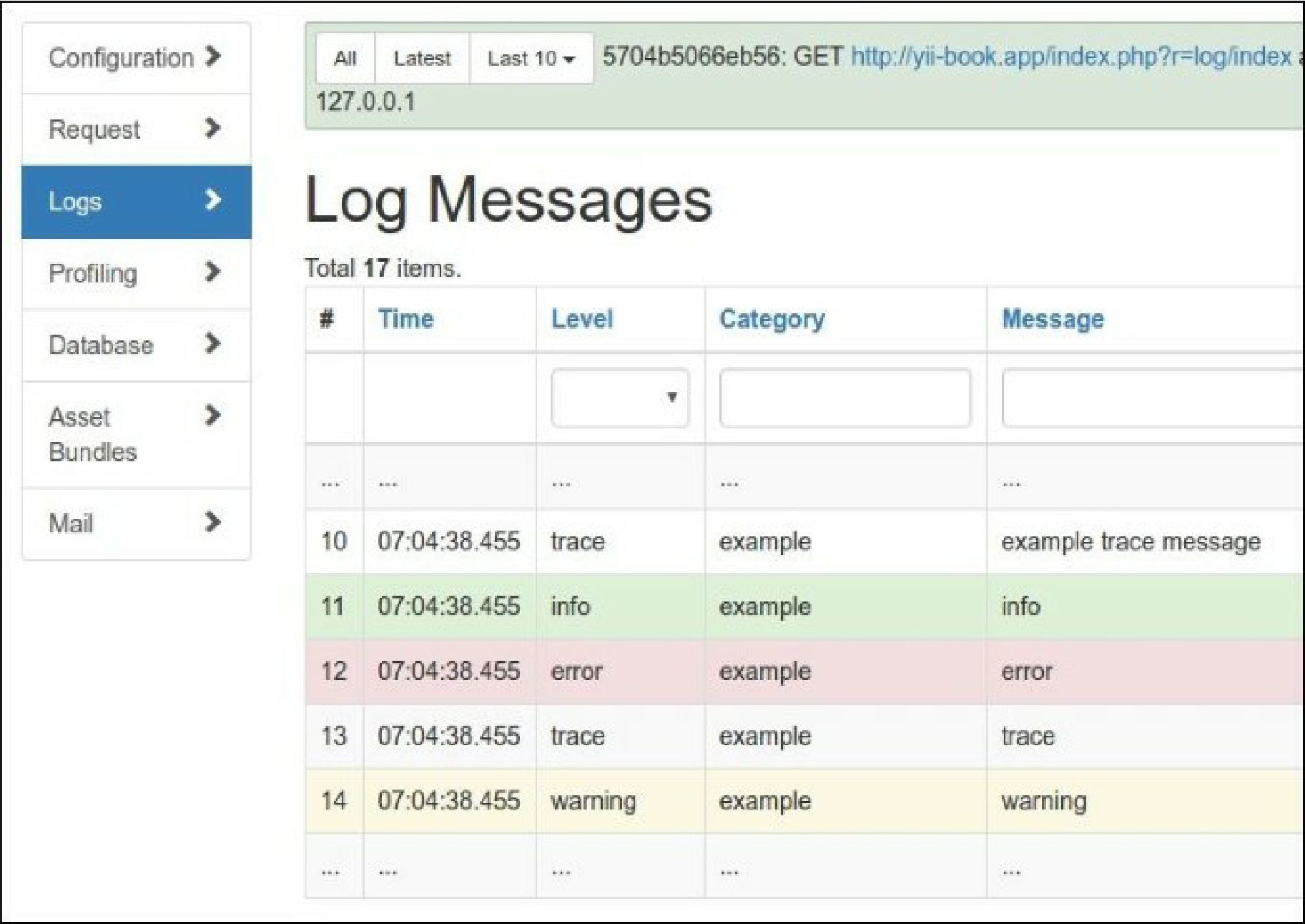
<h1>Log</h1>

</div>

1. Now run the preceding action multiple times. On the screen, you should see the Log heading and a  
   debug panel with the log messages number:



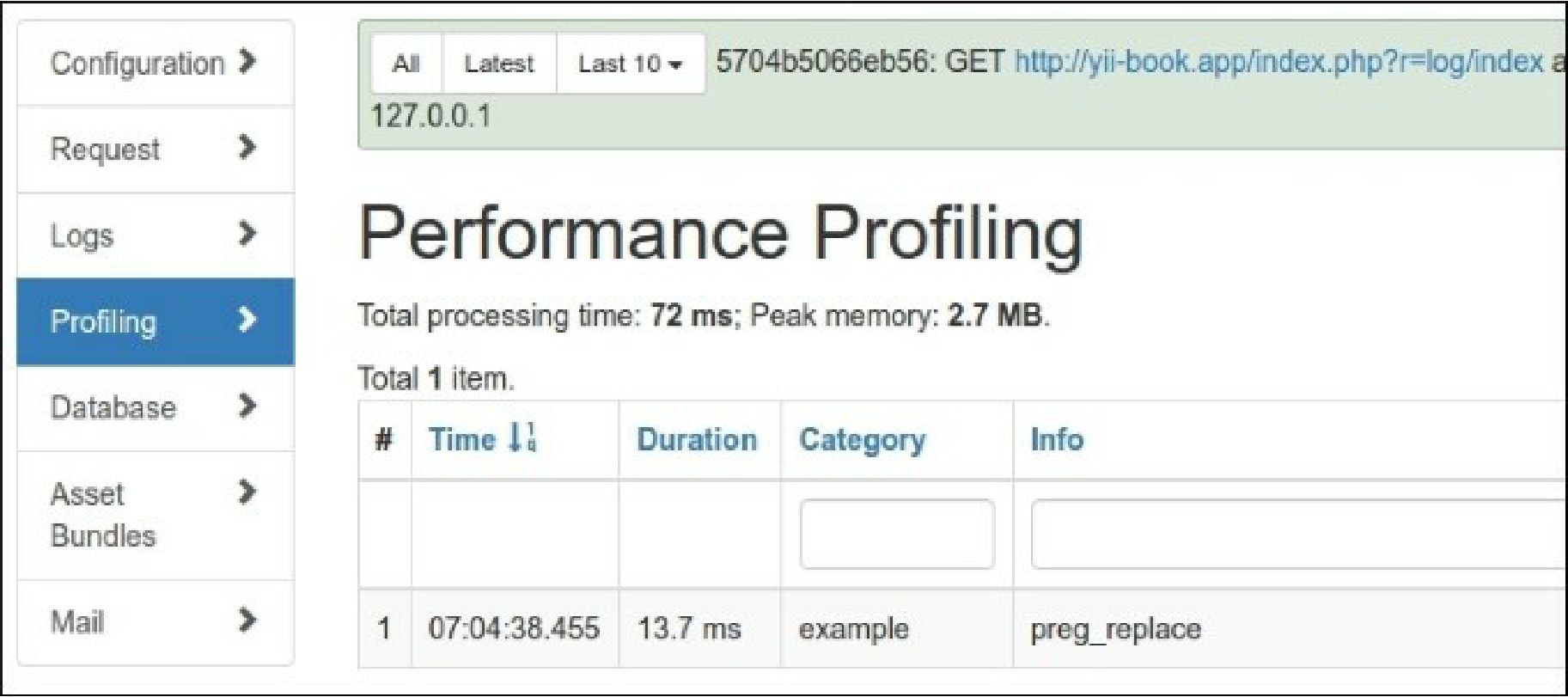
4. If you click on 17, you will see a web log similar to the one shown in the following screenshot:



5. A log contains all the messages we have logged along with stack traces, timestamps, levels, and

categories.

6. Now open the Profiling page. You should see profiler messages, as shown in the following  
screenshot:



Profiling info displays the total executing duration of own code block.

1. As we just changed the log file names and not the paths, you should look in runtime/logs to find  
   log files named error . log, warning . log, and info . log.
2. Inside, you will find the following messages:

2016-03-06 07:28:35 [127.0.0.1][-][-][error][example] error

2016-03-06 07:28:35 [127.0.0.1][-][-][warning][example] warning

2016-03-06 07:28:35 [127.0.0.1][-][-][info][example] inf  
o

How it works...

When one logs a message using Yii: :erorr, Yii: :warning, Yii: : info, or Yii: :trace, Yii passes it to  
the log router.

Depending on how it is configured, it passes messages to one or many targets, for example, e-mailing  
errors, writing debug information in file A, and writing warning information in file B.

The object of the yii\log\Dispatcher class is typically attached to an application component named log.  
Therefore, in order to configure it, we should set its properties in the configuration file components section.  
The only configurable property there is targets that contains an array of log routes and their configurations.

We have defined four log routes. Let’s review them as follows:

[

'class' => 'yii\log\EmailTarget',

'categories' => ['example'],

'levels' => ['error'],

// 'mailer' => 'mailer',

'message' => [

'from' => ['log@example.com'],

'to' => ['developer1@example.com', 'developer2@example.com'],

'subject' => 'Log error,

],

],

EmailTarget sends log messages through an e-mail via the Yii: :$app->mailer component by default.  
We limit category to example and level to error. An e-mail will be sent from log@example. com to two  
developers and the subject will be Log error:

[

'class' => 'yii\log\FileTarget',

'levels' => [warning],

'logFile' => '@runtime/logs/warning.log',

],

FileTarget appends error messages to a specified file. We limit the message level to warning and use a  
file named warning. log. We do the same for info-level messages by using a file named Info. log.

Also, we can use yii\log\SyslogTarget to write messages into the Unix /var/log/syslog system file  
and yii\log\DbTarget to write logs into the database. For the second one, you must apply their  
migrations:

./yii migrate --migrationPath=@yii/log/migrations/

There’s more...

There are more interesting things about Yii logging, which are covered in the following subsections.

Yii::trace versus Yii::getLogger()->log

Yii: :trace is a simple wrapper around Yii: :log:

public static function trace($message, $category = 'application')

{

if (YII\_DEBUG) {

static::getLogger()->log($message, Logger::LEVEL\_TRACE, $category);

}

}

Therefore, Yii: :trace logs a message with a trace level, if Yii is in the debug mode.

Yii::beginProfile and Yii::endProfile

These methods are used to measure the execution time of some part of the application’s code. In our  
LogController, we measured 10,000 executions of preg\_replace as follows:

Yii::beginProfile('preg\_replace', ' example');

for($i=0;$i<10000;$i++){

preg\_replace('~A[ a-z]+~', '', 'test it');

}

Yii::endProfile('preg\_replace', ' example');

Yii: : beginProfile marks the beginning of a code block for profiling. We must set a unique token for  
every code block and optionally specify a category:

public static function beginProfile($token, $category = 'application') { ... }

Yii: :endProfile has to be matched with a previous call to beginProfile with the same category name:  
public static function endProfile($token, $category = 'application') { ... }

The begin - and end - calls must also be properly nested.

Log messages immediately

By default, Yii keeps all log messages in memory until the application is terminated. That’s done for  
performance reasons and generally works fine.

However, if there is a console application with long running duration, log messages will not be written  
immediately. To make sure your messages will be logged at any moment, you can flush them explicitly  
using Yii:: $app->getLogger ()>flush(true) or change flushInterval and exportInterval for your  
console application configuration:

'components' => ['log' => ['flushInterval' => 1,'targets' => [['class' =>  
'yii\log\FileTarget','exportInterval' => 1,],], ],

],

See also

* [In order to learn more about logging, refer to http://www.yiiframework.com/doc-2.0/guide-runtime-](http://www.yiiframework.com/doc-2.0/guide-runtime-logging.html)  
  logging.html
* The Logging and using the context information recipe