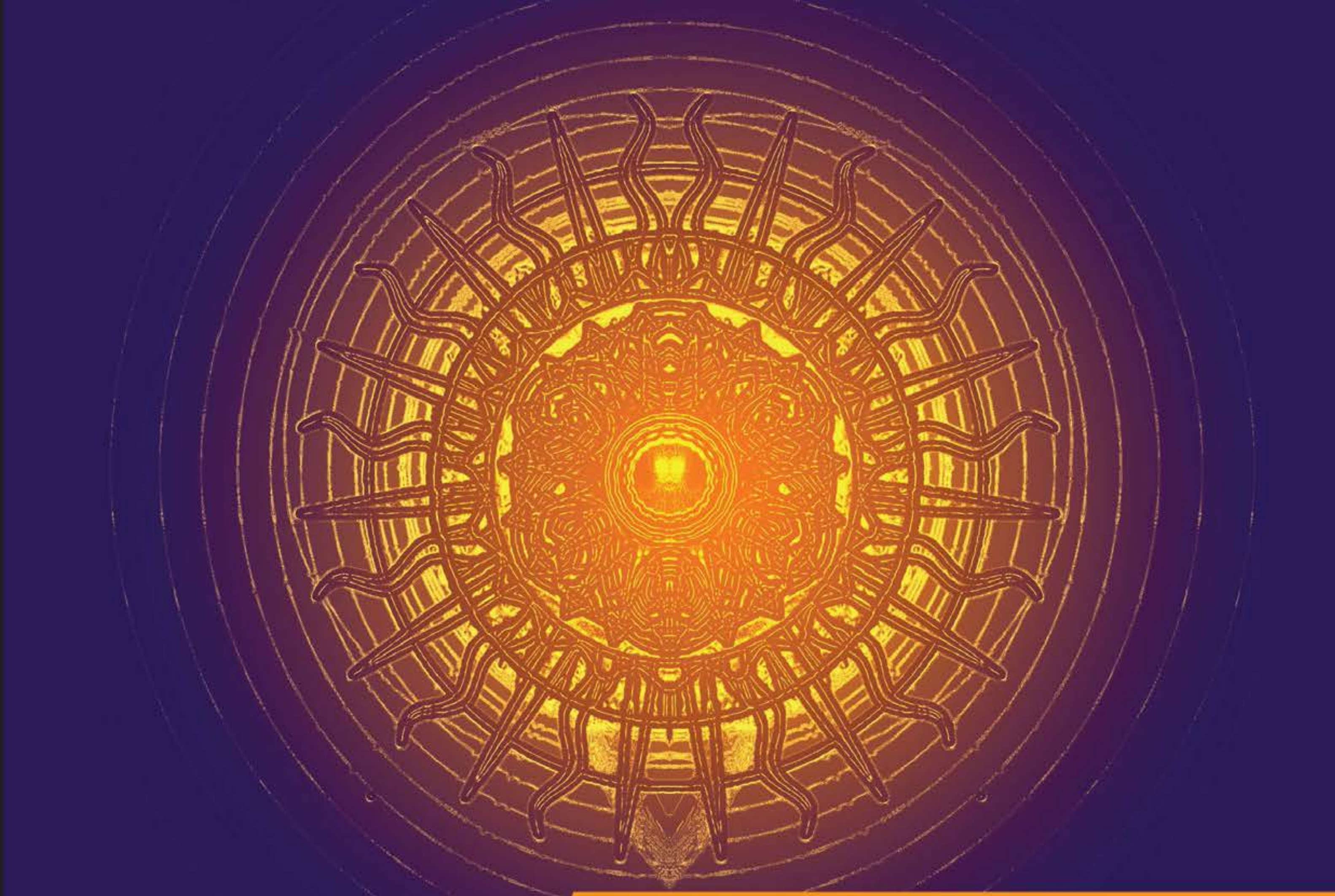


INSTANT

Short | Fast | Focused

Simple Botting with PHP

Enhance your botting skills and create your own web bots with PHP

Shay Michael Anderson

[PACKT]
PUBLISHING

Instant Simple Botting with PHP

Enhance your botting skills and create your own web bots with PHP

Shay Michael Anderson

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Simple Botting with PHP

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1230913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-929-1

www.packtpub.com

Credits

Author

Shay Michael Anderson

Reviewers

Dan Cryer

Abu Ashraf Masnun

Acquisition Editor

Ashwin Nair

Commissioning Editor

Priyanka Shah

Technical Editors

Ruchita Bhansali

Akashdeep Kundu

Project Coordinator

Joel Goveya

Proofreader

Chris Smith

Production Coordinator

Aditi Gajjar

Cover Work

Aditi Gajjar

Cover Image

Valentina D'silva

About the Author

Shay Michael Anderson has been programming and developing software since 1999. He quickly decided on software development as his career and enrolled in a college. He achieved his Bachelor of Science in Software Engineering degree through his studies at Oregon Tech and Colorado Tech. He then received a Master of Science in Software Systems Management from Colorado Tech. While earning his degrees in college, he achieved the undergraduate certificates for Software Engineering Application, Software Engineering Process, Object-Oriented Methods, and Unix Network Administration, and the graduate certificates for Systems Analysis and Integration, Network and Telecommunications, Data Management, and Project Management. Ever since he graduated from college he has been employed as a Web Application Developer, a Software Engineer, and a senior Software Engineer.

He is currently working as a senior Software Engineer for a large e-commerce and retail company. He develops and manages massive software systems, which are backed by a database cluster storing over a billion records. He also publishes open source software on his website, `www.shayanderson.com`.

About the Reviewers

Dan Cryer is a developer and system administrator from Cheshire, UK. With 10 years of commercial experience, he has worked on a wide and varied range of projects from the leading commercial forum software platform to national radio station websites, right through to a 5 TB-search data collection system spanning more than 100 servers, all using PHP and MySQL.

He is now a freelance developer and technical consultant. Working through his company Block 8, he offers in-depth technical advice and mentoring any kind of business, bespoke development and development leadership, and on-going systems support and management services.

Abu Ashraf Masnun is a business graduate from Khulna University, Bangladesh, and has over 6 years of work experience in the local software industry. He crafts web applications using PHP, Python, and JavaScript at his daily work. Besides web development, he also nurtures a keen passion towards Android development and Linux system administration. He's a quick learner, early adopter, and team player. At leisure, he contributes to open source projects and community discussions.

> I would like to thank my friends and family who have always been a great source of encouragement and endless enthusiasm.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Table of Contents

Instant Simple Botting with PHP

Welcome to *Instant Simple Botting with PHP*. This book will explain all the information and code you will need to start simple botting with PHP. Using this book and PHP, you will learn the basics of HTML requests and responses, get started with building your own bot, and learn how to parse and save data that you harvest with your bot.

This document contains the following sections:

So, what is Simple Botting with PHP? lets you discover what simple botting with PHP actually is, what you can do with it, how you can create your own bots, and why it's so great.

Installation teaches you how to create your own command-line PHP applications, how to execute command-line PHP applications, the difference between using cURL and simple socket connections, and how to perform simple HTTP GET and POST requests.

Quick start will teach you how to create your own bot, implement the bot configuration settings, instantiate the bot and execute requests, and save data harvested by the bot.

Top 5 features you need to know about will help you find out how to perform five important botting tasks. By the end of this section, you will be able to parse harvested data, store parsed data in multiple ways, build bot logging, add stealth to your bots, and start creating advanced features for your bots such as link handling.

People and places you should get to know will provide you with various helpful suggestions and links to the project page, as well as articles and tutorials that can further assist you in developing powerful PHP bots. Open source projects are centered on a community of sharing information and tools.

So, what is Simple Botting with PHP?

In this book, I am going to explain how to create your own bots using PHP. You should already be familiar with **PHP** (**Hypertext Preprocessor scripting language**) and common built-in PHP functions. Throughout this book, I will only use common PHP functions that will be available in basic PHP installations. PHP is a good language to use to create your first robot because it is a popular and powerful language that can easily be tested in a web browser.

What is a robot? A robot or bot or web bot or spider (bots that navigate on their own) is a software application that is used to systematically execute requests and handle responses that can be used to the benefit of its developer. These benefits can include activities such as gathering or harvesting data, checking a website for errors or invalid links, checking e-mail, or handling more advanced issues such as crawling and archiving multiple websites.

Why use robots? The benefits listed above are all good reasons to use bots. Furthermore, bots can often be used to complete tasks by saving time through automation. For example, say, the company you work for has a project that requires data entry. A data directory on the local company server stores flat files that must be opened by an end user. Then, the end user must copy the records in the flat file line-by-line and paste the copied strings into various web application form fields. Finally, the end user submits the web application form and the data is saved in a proprietary database.

If there were only twenty flat files on the server with a total of five hundred records, it would probably be logical to have a data entry employee to complete the task. However, say, there were one thousand files with twenty five thousand records. Now, it might be more tactical to develop a bot capable of scanning the files, extracting the records, and submitting the records through the web application using HTTP POST requests. In this book, you will learn the logic that will allow you to create a bot capable of completing these basic tasks; however, you can take that knowledge and—through practice—build advanced bots that can execute a wide variety of tasks.

HTTP request types

A web bot is a bot that can be programmed to carry out commands over the Internet and relies on web resources. Anything you commonly use on the Internet can also be used by a web bot. Obviously, we as end users, use the Internet much differently than a web bot does. Most of the times, when you want to submit a form on a website you simply fill the HTML form and click on a submit button. The website will process the posted information (HTTP request) and maybe redirect you to another page (HTTP response), where the website owner thanks you for completing their form.

When we develop a bot, we must attack the same task using a different workflow. First, we need to program the bot so that it sends an HTTP request with the same data that would be submitted on the website's HTML form. Instead of having the bot click on the submit button, we simply set the response type sent by the bot. By doing this, we can signal the web server that we are sending data that we want the web server to digest. This type of request is called a POST request.

Another type of request is a GET request, which is a more common and simpler request type. A GET request simply asks the server to provide a resource based on a URL. In our bots, we will be using both GET and POST requests. In simple terms, you can think of a GET request exactly like you're telling a web server to get something for you (a getter method type). A POST request, on the other hand, is like telling the server to set something for you (a setter method type).

Simple is smarter

If you are familiar with popular **APIs** (**Application Programming Interfaces**), you'll know that they work much the same as HTTP requests and responses. In fact, our web bots will act as an API to web servers. What do I mean by this? Most APIs work the same; we can request an action that normally triggers a response that can then be consumed and utilized.

In much the same way, we will instruct our bot to request something and after the request has been sent our bot will fetch the response and execute various functions or methods. If a bot is developed correctly, we don't have to think about everything the bot is doing internally. Exactly in the same way we don't have to think about what an API is actually doing when we send the request, rather we will just expect a response.

Code example expectations

In order to build bots that mimic APIs and are simple to use, we need to develop them using PHP classes, which will allow us to use bot objects. If you are unfamiliar with PHP object-oriented programming (OOP) you should research it before we use classes and objects later on.

In this book, I will be demonstrating PHP code using **PHP 5.4 coding standards** and plentiful code comments.

Installation

In this section, I am going to discuss the development environment we will be using to develop bots, using simple PHP command-line applications, summarize PHP error reporting, and execute an HTTP request. In the next section, we will take a look at HTTP GET requests and HTTP responses in detail and begin executing these types of requests.

Step 1 – setting the development environment

While you should already be familiar with basic PHP functions and logic, I am going to outline the basic development environment, which you should use when reading and using the code in this book. As stated earlier, I will be using PHP 5.4 coding standards in all of the code examples in this book. Therefore, you should use a web server equipped with a PHP 5.4 (or higher) basic installation. Also, during the course of this book, I will be using command lines to execute PHP applications using a web server with a Linux operating system (Ubuntu 12 to be exact).

On my Ubuntu server I would install PHP with the following command-line support using:

```
# sudo apt-get install php5 libapache2-mod-php5 php5-cli
```

You can check the PHP version installed on your web server in one of two ways. First, if you have a PHP **CLI** (**Command Line Interface**) **SAPI** (**Server API**) installed on your web server, you can use a command line to get the PHP Version. Here is a command line example on a Linux web server:

```
# php -v
```

This will print something like:

```
PHP 5.4.6-1ubuntu1.1 (cli) (built: Nov 15 2012 01:18:34)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2012 Zend Technologies
```

In the previous example, my web server is prepared with PHP 5.4.6 installed.

The second way to check the PHP Version version on your web server is to set up a PHP script that will display PHP web server information on a web page. To do this carry out the following steps:

1. Create a file called `info.php` on your web server in the `/var/www` directory, and add the following content to the file:

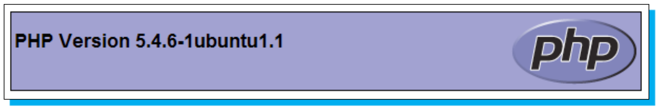
   ```
   <?php
   /**
    * Display PHP version/info
    */

   phpinfo();
   ```

2. Save the `/var/www/info.php` file and open the web page in a web browser. For example, the URL to the web page might look something like:

   `http://localhost/info.php`

3. Once you load the web page in a web browser, you should see the PHP Version version at the top of the page. It will look something like this:



PHP Version 5.4.6-1ubuntu1.1

You can see that—using this method—my web server has PHP Version 5.4.6 installed.

PHP error reporting

If you are not familiar with PHP error reporting (`www.php.net/manual/en/function.error-reporting.php`) and displaying PHP errors (`www.php.net/manual/en/errorfunc.configuration.php#ini.display-errors`), I would suggest reading on these two topics. Any proficient PHP programmer should be accustomed to PHP error reporting and displaying PHP errors.

I would highly recommend that you develop the bots of this book, and all other code of this book, in a development environment, and not a production environment. In your development environment, I would suggest allowing PHP to display errors, warnings, and notices and your `php.ini` file. Without these turned on, you might get lost with some of the examples in this book.

Step 2 – command-line applications

Although you can develop bots and test them using typical web browser-based PHP applications, sometimes it is useful to develop and test bots using PHP command-line applications. I prefer using command-line applications to develop bots, and test them, because command-line applications allow us better memory usage, real-time messages/alerts, and the ability to fire **slave processes** (an advanced topic, outside the scope of this book).

In this book, I will be using typical web browser-based PHP applications because it will be easier for most programmers. However, I will cover using PHP command-line applications here in case you would rather develop and test your bots using command-line applications.

Creating a PHP command-line application is very simple. So, if you have never created one before, don't worry about having to learn a lot of new logic. Let's create a simple PHP command-line application now. In order to create a PHP command-line application and execute it on your web server, you will need a PHP CLI. So, install this before attempting the execution of a PHP command-line application. On my web server (Linux, Ubuntu), I can install a PHP CLI using the following command line:

```
# sudo apt-get install php5-cli
```

Save a file called `01_command_line_app.php` on your web server, in a directory that you will use for all the code in this book (called the `project_directory` throughout this book) and add the following code to the `project_directory/01_command_line_app.php` file:

```
#!/usr/bin/env php
<?php

echo "Hello world\n";
```

Now we can run our simple PHP command-line application using the executable file from the command line:

```
# /var/www/project_directory/01_command_line_app.php
Hello world
```

You may need to allow the file to be executed by the operating system; this can be done using the following command:

```
# sudo chmod +x /var/www/project_directory/01_command_line_app.php
```

> I am using Linux/Unix commands, so if you are using a different operating system please use the appropriate commands.

The first thing you might notice is the difference between the PHP command-line application and a typical web browser-based PHP application, which is in the first line of code `#!/usr/bin/env php`. This line is called a **shebang** and will probably be familiar to most Linux/Unix users. This is a simple way of notifying the operating system what interpreter program should be used to execute the code in the file. In this example, we are telling the operating system to use the PHP interpreter.

You can find the required full path of the PHP interpreter program on your Linux/Unix web server by using the `which` command:

```
# which php
/usr/bin/php
```

We can use a shebang so that we don't have to manually tell the operating system which interpreter program to use, but we don't have to. For example, we could remove the shebang from the `project_directory/01_command_line_app.php` file:

```
<?php

echo "Hello world\n";
```

And now to run the application as a command-line application we will use the following command line:

```
# /usr/bin/php /var/www/project_directory/01_command_line_app.php
Hello world
```

Or, if your web server is set up like mine is, you can simply use:

```
# php /var/www/project_directory/01_command_line_app.php
Hello world
```

You should use whatever method you find easiest for executing your PHP command-line applications.

And that's it!

There are different methods that can be used when making HTTP GET and POST requests in PHP. The two methods that I am going to discuss are the PHP built-in function `file_get_contents()` and the cURL library (`www.php.net/manual/en/book.curl.php`). In this book, I am going to use the PHP function `file_get_contents()` when executing HTTP requests, because I find it the simplest way to learn and teach. However, cURL is a powerful library that you may find useful when you continue to develop bots on your own.

Once you've installed the cURL library with your PHP installation you can use cURL in your code to send HTTP requests. Here is an example:

```
<?php
// ensure PHP cURL library is installed
if(function_exists('curl_init'))
{
  // set cURL resource
  $curl = curl_init('http://www.google.com');

  // set return transfer as string to true
  curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);

  // set variable with response string
  $request_response = htmlentities(curl_exec($curl));

  // close cURL resource
  curl_close($curl);

  // display response string
  echo '<pre>' . print_r($request_response, true) . '</pre>';
}
else // PHP cURL library not installed
{
  echo 'Please install PHP cURL library';
}
```

In the preceding code, we used the cURL library to create a simple HTTP GET request (the default request type for the cURL library is a GET). As you can see, this is a fairly simple way to execute an HTTP GET request in PHP. However, as I stated earlier, in this book, I will be using the `file_get_contents()` PHP function to perform HTTP requests.

Quick start – developing a bot

Now that we have covered the development environment, we can get to the fun material, that is, programming our first bot. To begin with, we will develop an HTTP package, which we can use in our bot applications to handle HTTP requests and responses.

Step 1 – HTTP request classes

Now that we have discussed our development environment, coding standards, and how basic HTTP requests and responses work, let's create an HTTP request class that actually does something useful. Again, if you are not familiar with developing PHP classes, and using PHP objects, you will want to research on these topics before reading this section. While this book is a starter book for developing PHP bots, I want to teach you the correct way to develop bots, which means using the power and reusability of classes and objects, also known as **OOP** (**Object-Oriented Programming**).

When we use well-designed classes, we can use them as objects in our current project, for which we are developing for those classes. Also, other future projects that require the same type of functionality and logic can use these classes as objects. If you desire to be a productive and successful programmer, you will eventually need to accept this point of view, if you haven't already.

In this section, I am going to help you develop a basic HTTP request class, which we will be able to use in *all* of our bots that need this type of tool. Later, we will create an HTTP response class that will easily allow our request class to return objects instead of arrays of information, which will be very useful.

Before you start developing a class, you should always spend some time primarily designing it. This way you won't have to think as much while you are developing the class. So, let's spend a little time thinking about the design of our HTTP request class.

Here are some requirements we will need in our HTTP request class:

- We will need methods for GET and HEAD HTTP requests.
- The request methods should be simple, we only want to pass a URL and a timeout value to the GET and HEAD methods.
- We want the request methods to return HTTP response objects, not arrays or other scalar values.

These are some good requirements for a basic HTTP request class. Obviously, we could go much further with the design of our class and add other advanced options and methods such as debugging mode and logging. However, the functionality for which we are developing this project, will get carried out just fine.

Before we start developing the HTTP request class, let's set up a project directory structure through the following steps. You will be able to use this directory structure for all the projects in this book.

1. Create the following directory structure and files (the files can be empty for now) in a desired location on your web server:

```
'-- project_directory
|-- lib
|      '-- HTTP
|            |--Request.php
|            '--Response.php
|-- 01_command_line_app.php (from command line application
example)
'-- 02_http.php
```

2. Now, place all our class files (library files) in the `project_directory/lib` directory.
3. Open the `Request.php` file.
4. Add the first few lines of code for our `Request` class located at `project_directory/lib/HTTP/`:

```
<?php
namespace HTTP;

/**
 * HTTP Request class - execute HTTP get and head requests
 *
 * @package HTTP
 */
class Request
{
```

First, in our `Request.php` file we set the namespace of `HTTP`. This is a simple way of telling PHP that we want the `Request` class in the `HTTP` container. If you are not familiar with namespaces, you can research the topic further at `www.php.net/manual/en/language.namespaces.php`. However, for now, you can think of a namespace as a container. So, for every class, function, method, or constant in the namespace (or container) `HTTP` should have something to do with HTTP logic. So, for example, if we had the namespace `Database`, everything in that namespace would include logic and methods for database functionality.

Next, we leave some simple, yet helpful, comments that will allow any developer to quickly determine what the class is used for. And finally, we declare our class name as `Request`. One important thing to note here is that this code will not work on any PHP Version prior to PHP 5.3, because PHP didn't include namespace syntax until PHP 5.3.

The first class method we are going to add to our `Request` class is a method that can format a timeout value if a timeout value has been used, or return a default timeout value if no timeout value has been used. This method will make more sense later on when we develop the request methods. Here are the next lines of code in our `Request` class, used for the `formatTimeout` method:

Insert the following snippet of code to our Request class located at project_directory/lib/HTTP/:

```
/**
 * Format timeout in seconds, if no timeout use default timeout
 *
 * @param int|float $timeout (seconds)
 * @return int|float
 */
private static function __formatTimeout($timeout = 0)
{
  $timeout = (float)$timeout; // format timeout value

  if($timeout < 0.1)
  {
    $timeout = 60; // default timeout
  }

  return $timeout;
}
```

As you can see, this is a very simple method that takes an optional parameter called timeout and formats the timeout value so that it can be used properly in our request methods. If no timeout value is passed to the method, it will return the default timeout value (60 seconds).

Next, we need to create a method that can parse the raw HTTP response that we receive from our request methods. Once the raw HTTP response has been parsed, we can use the response parts with our HTTP Response class (which we'll build later) to form a usable response object. Here is the parse response method:

Insert the following snippet of code to our Request class:

```
/**
 * Parse HTTP response
 *
 * @param string $body
 * @param array $header
 * @return \HTTP\Response
 */
private static function __parseResponse($body, $header)
{
    $status_code = 0;
    $content_type = '';

    if(is_array($header) && count($header) > 0)
  {
```

```
                foreach($header as $v)
                {
                    // ex: HTTP/1.x XYZ Message
                    if(substr($v, 0, 4) == 'HTTP'
                        && strpos($v, ' ') !== false)
                    {
                        $status_code = (int)substr($v,
                            strpos($v, ' '), 4); // parse status code
                    }
                    // ex: Content-Type: *; charset=*
                    else if(strncasecmp($v, 'Content-Type:', 13) ===
                      0)
                    {
                        $content_type = $v;
                    }
                }
            }

            return new \HTTP\Response($status_code,
                $content_type, $body, $header);
        }
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

This parse method takes two arguments: the `body` (`string`) and the `header` (`array`). In the method, we first initialize the variables for status code and content type. These variables will be used in the HTTP `Response` object, which is returned by the method. The next part of the method loops through the HTTP header and parses out the status code (HTTP response code) and the content type (MIME type). These variables are also used when creating the HTTP `Response` object. And finally, we instantiate the HTTP `Response` object—which we build in the next section—and return it. This method will make more sense once we have finished creating the entire class, but we needed to include it first so that our request methods can utilize it.

Now, let's build the first request method of our `Request` class—the `get()` request method. The `get()` method will be the most common request method that we will use in our bots. Here is the code for the `get()` method:

Insert the following snippet of code to our Request class:

```
/**
 * Execute HTTP GET request
 *
 * @param string $url
 * @param int|float $timeout (seconds)
 * @return \HTTP\Response
 */
public static function get($url, $timeout = 0)
{
    $context = stream_context_create();
    stream_context_set_option($context, 'http', 'timeout',
      self::__formatTimeout($timeout));

    $http_response_header = NULL; // allow updating

    $res_body = file_get_contents($url, false, $context);

    return self::__parseResponse($res_body,
      $http_response_header);
}
```

The preceding method is very simple. We tell the get() method—through the parameters—which URL to send to the GET request. Then we can pass an optional timeout value. Our get() method then creates the required context for the HTTP GET request, sends the request, and creates and returns an \HTTP\Response object that we can use.

Next, we can build the head() method. An HTTP HEAD request is a very simple GET request that does not include a response message body. This request can be useful for simple requests, such as pinging an HTTP service on a web server.

Here is the code for the HEAD request. Insert the following snippet of code to our Request class located at project_directory/lib/HTTP/:

```
/**
 * Execute HTTP HEAD request
 *
 * @param string $url
 * @param int|float $timeout
 * @return \HTTP\Response
 */
public static function head($url, $timeout = 0)
{
    $context = stream_context_create();
```

```
            stream_context_set_option($context, [
                'http' => [
                    'method' => 'HEAD',
                    'timeout' => self::__formatTimeout($timeout)
                ]
            ]);

            $http_response_header = NULL; // allow updating

            $res_body =     file_get_contents($url, false, $context);

            return self::__parseResponse($res_body,
              $http_response_header);
        }
```

The preceding method is very straightforward. First, we accept the URL and timeout values. Then, we configure a custom `stream_context`, which is used when we fetch the HTTP HEAD request. Then, we use the built-in PHP function `file_get_contents()` to send the HTTP HEAD request. Finally, we return the `\HTTP\Response` object, which is created in our `__parseResponse()` method.

Step 2 – the HTTP response class

Now, we need to create an HTTP `Response` class that can be used to easily transform our HTTP response data into a useable object. This way, once we receive an HTTP response in our `Request` class, we simply pass the response data to our `Response` object and we don't have to think about all the details and methods required to use the HTTP response data in our applications.

Create a file called `Response.php` in the HTTP package directory and add the code available from the book source code download file `project_directory/lib/HTTP/Response.php` at Packt Publishing's website.

As you can see, it takes quite a few lines of code to create our `\HTTP\Response` class, however, the class is very simple.

First, we define HTTP response status codes and status code messages. Status codes are used by web servers to denote the response status. For example, if an HTTP GET request is sent to a web server and the web server returns an HTTP response status code of `200`, we know that the request has been handled successfully. You can see, by looking at the `Response` class status codes and status code messages, that there are a variety of status codes and messages with which the web server can respond.

Next, in the __construct() method of the Response class, we accept the status code, type (content type), body, and header. These are the only parameters that we need to perform operations such as initialization, or instantiation with the Response object in order to make it work properly for us.

Why use objects?

Now that we have our \HTTP\Request and \HTTP\Response objects completed, let's take a look at why we are developing classes and using objects (**Object-Oriented Programming** or **OOP**) instead of using procedural lines of code (**Procedure-Oriented programming** or **POP**). To illustrate this point, I am going to have you execute your first HTTP HEAD request using our HTTP classes.

Create the a file called 02_http.php in your project directory where it will have access to the /project_directory/lib/HTTP directory. Add the following code to the 02_http.php file located at /project_directory/:

```
<?php
/**
 * Example HTTP GET request
 */

// include our classes
require_once './lib/HTTP/Request.php';
require_once './lib/HTTP/Response.php';

// execute example HTTP GET request
$response = \HTTP\Request::head('http://www.google.com');

// print out HTTP response (\HTTP\Response object)
echo '<pre>' . print_r($response, true) . '</pre>';
```

In this code, first we include our Request and Response classes that we have developed. Next, we set the response variable with the response (\HTTP\Response) object that is created by the \HTTP\Request::head() method. Finally, we print the HTTP Response object for illustration, or debugging/testing purposes. If you execute this code in a web browser, you should see something like the following:

```
HTTP\Response Object
(
    [__body:HTTP\Response:private] =>
    [__encoding:HTTP\Response:private] => ISO-8859-1
    [__header:HTTP\Response:private] => Array
        (
            [0] => HTTP/1.0 200 OK
            [1] => Date: (date/time) GMT
            [2] => Expires: -1
            [3] => Cache-Control: private, max-age=0
            [4] => Content-Type: text/html; charset=ISO-8859-1
            [5] => Set-Cookie: ***
            [6] => Set-Cookie: ***
            [7] => P3P: ***
            [8] => Server: gws
            [9] => X-XSS-Protection: 1; mode=block
            [10] => X-Frame-Options: SAMEORIGIN
        )

    [__mime:HTTP\Response:private] => text/html
    [__status:HTTP\Response:private] => 200
    [__status_message:HTTP\Response:private] => OK
    [success] => 1
)
```

Success! We have successfully executed an HTTP HEAD request, received a response, parsed it, created an HTTP `Response` object, and printed the object. Now, we could easily use the object for more useful things; for example, change the `02_http.php` file located at `/project_directory/` as follows:

```
// display response status
if($response->success)
{
    echo 'Successful request <br />';
}
else
{
    echo 'Error: request failed, status code: '
    . $response->getStatusCode() . '<br />'; // prints status code
}
```

If we were using procedural programming (POP) instead of classes and objects (OOP), it would be much more difficult to do this. Also, using classes with namespaces makes it easy for us to use them in other application frameworks and not have class naming conflicts. Also, this approach of programming makes it much easier to determine for what type of logic and purpose a class is designed. For example, it would be easy for another programmer to conclude that the `\HTTP\Request` class is used to generate HTTP requests.

Step 3 – using bootstrap files

Now that we have our `HTTP` package—set by the namespace `HTTP`—completed, we can easily use it for other projects and applications. Sometimes, especially when using large packages or library files, it is hard to remember or find out what exactly needs to take place in order for us to get a package ready for use in our own software applications. A package might require extensive configuration settings, class autoloading, common file loading, external package or library files, and more.

A simple solution to this problem is using a bootstrap file. A **bootstrap file** can be used to initialize everything the package requires to load and initialize properly. Our `HTTP` package doesn't require much file loading or any configuration settings, but for the sake of example, let's create a simple bootstrap file for our `HTTP` package:

1. Create a file called `bootstrap.php` in the `HTTP` package directory.
2. Add the following code to our `bootstrap` class located at `project_directory/lib/HTTP/`:

   ```
   <?php
   namespace HTTP;

   /**
    * Bootstrap file
    *
    * @package HTTP
    */

   // load class files
   require_once './lib/HTTP/Request.php';
   require_once './lib/HTTP/Response.php';
   ```

   The preceding code resembles a very simple bootstrap file. We are simply loading the classes required in our `HTTP` package. However, it will make the use of our `HTTP` package even easier!

3. Now, in our `02_http.php` file, modify the code to use our `bootstrap.php` file instead of loading the class files manually:

   ```
   <?php
   /**
    * Example HTTP GET request
    */

   // load HTTP package with bootstrap file
   require_once './lib/HTTP/bootstrap.php';

   // execute example HTTP GET request
   ```

```
$response = \HTTP\Request::get('http://www.google.com');

// display response status
if($response->success)
{
    echo 'Successful request <br />';
}
else
{
    echo 'Error: request failed, status code: '
    . $response->getStatusCode() . '<br />'; // prints
       // status code
}

// print out HTTP response (\HTTP\Response object)
echo '<pre>' . print_r($response, true) . '</pre>';
```

Although this is an extremely simple example of how a bootstrap file can be used to make the use of a package easier, it is still beneficial to any programmer who uses our code. Also, in later sections of this book we will be using bootstrap files when we develop our bot package.

Step 4 – creating our first bot, WebBot

At this point in the book, you should be aware of and comfortable with HTTP requests and responses, how to develop `HTTP` packages (covered earlier in the book), and why we use bootstrap files.

With the knowledge you have gained, we are now ready to develop our first bot, which will be a simple bot that gathers data (documents) based on a list of URLs and datasets (fields and field values) that we will require.

First, let's start by creating our bot package directory. So, create a directory called `WebBot` so that the files in our `project_directory/lib` directory look like the following:

```
'-- project_directory
    |-- lib
    |    |-- HTTP (our existing HTTP package)
    |    |    '-- (HTTP package files here)
    |    '-- WebBot
    |        |-- bootstrap.php
    |        |-- Document.php
    |        '-- WebBot.php
|-- (our other files)
    '-- 03_webbot.php
```

As you can see, we have a very clean and simple directory and file structure that any programmer should be able to easily follow and understand.

Step 5 – the WebBot class

Next, open the file WebBot.php file and add the code from the book source code download file project_directory/lib/WebBot/WebBot.php at Packt Publishing's website.

In our WebBot class, we first use the __construct() method to pass the array of URLs (or documents) we want to fetch, and the array of document fields that are used to define the datasets and regular expression patterns. Regular expression patterns are used to populate the dataset values (or document field values). If you are unfamiliar with regular expressions, now would be a good time to study them at www.php.net/manual/en/regexp.introduction.php. Then, in the __construct() method, we verify whether there are URLs to fetch or not. If there are not, we set an error message stating this problem.

Next, we use the __formatUrl() method to properly format URLs for which we fetch data. This method will also set the correct protocol: either HTTP or **HTTPS** (**Hypertext Transfer Protocol Secure**). If the protocol is already set for the URL, for example http://www.[dom].com, we ignore setting the protocol. Also, if the class configuration setting conf_force_https is set to true, we force the HTTPS protocol, again unless the protocol is already set for the URL.

We then use the execute() method to fetch data for each URL, set and add the Document objects to the array of documents, and track document statistics. This method also implements fetch-delay logic that will delay each fetch by *x* number of seconds if set in the class configuration settings conf_delay_between_fetches. We also include the logic that only allows distinct URL fetches, meaning that, if we have already fetched data for a URL, we won't fetch it again; this eliminates duplicate URL data fetches. The Document object is used as a container for the URL data, and we can use the Document object to use the URL data, the data fields, and their corresponding data field values.

In the execute() method, you can see that we have performed a \HTTP\Request::get() request using the URL and our default timeout value—which is set with the class configuration settings conf_default_timeout. We then pass the \HTTP\Response object that is returned by the \HTTP\Request::get() method to the Document object. Then, the Document object uses the data from the \HTTP\Response object to build the document data.

Finally, we include the getDocuments() method, which simply returns all the Document objects in an array that we can use for our own purposes as we desire.

Step 6 – the WebBot Document class

Next, we need to create a class called Document that can be used to store document data and field names with their values. To do this we will carry out the following steps:

1. We first pass the data retrieved by our WebBot class to the Document class.
2. Then, we define our document's fields and values using regular expression patterns.
3. Next, add the code from the book source code download file project_directory/lib/WebBot/Document.php at Packt Publishing's website.

Our Document class accepts the \HTTP\Response object that is set in WebBot class's execute() method, and the document fields and document ID.

4. In the Document __construct() method, we set our class properties: the HTTP Response object, the fields (and regular expression patterns), the document ID, and the URL that we use to fetch the HTTP response.
5. We then check if the HTTP response is successful (status code 200), and if it isn't, we set the error with the status code and message.
6. Lastly, we call the __setFields() method.

The __setFields() method parses out and sets the field values from the HTTP response body. For example, if in our fields we have a title field defined as $fields = ['title' => '<title>(.*)<\/title>'];, the __setFields() method will add the title field and pull all values inside the <title>*</title> tags into the HTML response body. So, if there were two title tags in the URL data, the __setField() method would add the field and its values to the document as follows:

```
['title'] => [
    0 => 'title x',
    1 => 'title y'
]
```

If we have the WebBot class configuration variable—conf_include_document_field_raw_values—set to true, the method will also add the raw values (it will include the tags or other strings as defined in the field's regular expression patterns) as a separate element, for example:

```
['title'] => [
    0 => 'title x',
    1 => 'title y',
    'raw' => [
        0 => '<title>title x</title>',
        1 => '<title>title y</title>'
    ]
]
```

The preceding code is very useful when we want to extract specific data (or field values) from URL data.

To conclude the Document class, we have two more methods as follows:

- getFields(): This method simply returns the fields and field values.
- getHttpResponse(): This method can be used to get the \HTTP\Response object that was originally set by the WebBot execute() method.

This will allow us to perform logical requests to internal objects if we wish.

Step 7 – the WebBot bootstrap file

Now we will create a `bootstrap.php` file (at `project_directory/lib/WebBot/`) to load the `HTTP` package and our `WebBot` package classes, and set our `WebBot` class configuration settings:

```
<?php
namespace WebBot;

/**
 * Bootstrap file
 *
 * @package WebBot
 */

// load our HTTP package
require_once './lib/HTTP/bootstrap.php';

// load our WebBot package classes
require_once './lib/WebBot/Document.php';
require_once './lib/WebBot/WebBot.php';

// set unlimited execution time
set_time_limit(0);

// set default timeout to 30 seconds
\WebBot\WebBot::$conf_default_timeout = 30;

// set delay between fetches to 1 seconds
\WebBot\WebBot::$conf_delay_between_fetches = 1;

// do not use HTTPS protocol (we'll use HTTP protocol)
\WebBot\WebBot::$conf_force_https = false;

// do not include document field raw values
\WebBot\WebBot::$conf_include_document_field_raw_values = false;
```

We use our `HTTP` package to handle HTTP requests and responses. You have seen in our `WebBot` class how we use HTTP requests to fetch the data, and then use the HTTP `Response` object to store the fetched data, in the previous two sections. That is why we need to include the bootstrap file to load the `HTTP` package properly.

Then, we load our `WebBot` package files. Because our `WebBot` class uses the `Document` class, we load that class file first.

Next, we use the built-in PHP function `set_time_limit()` to tell the PHP interpreter that we want to allow unlimited execution time for our script. You don't necessarily have to use unlimited execute time. However, for testing reasons, we will use unlimited execution time for this example.

Finally, we set the `WebBot` class configuration settings. These settings are used by the `WebBot` object internally to make our bot work as we desire. We should always make the configuration settings as simple as possible to help other developers understand. This means we should also include detailed comments in our code to ensure easy usage of package configuration settings.

We have set up four configuration settings in our `WebBot` class. These are `static` and `public` variables, meaning that we can set them from anywhere after we have included the `WebBot` class, and once we set them they will remain the same for all `WebBot` objects unless we change the configuration variables. If you do not understand the PHP keyword `static`, now would be a good time to research this subject.

- The first configuration variable is `conf_default_timeout`. This variable is used to globally set the default timeout (in seconds) for all `WebBot` objects we create. The timeout value tells the `\HTTP\Request` class how long it should continue trying to send a request before stopping and deeming it as a bad request, or a timed-out request. By default, this configuration setting value is set to `30` (seconds).
- The second configuration variable—`conf_delay_between_fetches`—is used to set a time delay (in seconds) between fetches (or HTTP requests). This can be very useful when gathering a lot of data from a website or web service. For example, say, you had to fetch one million documents from a website. You wouldn't want to unleash your bot with that type of mission without fetch delays because you could inevitably cause—to that website—problems due to the massive number of requests. By default, this value is set to `0`, or no delay.
- The third `WebBot` class configuration variable—`conf_force_https`—when set to `true`, can be used to force the HTTPS protocol (instead of the default HTTP protocol). As mentioned earlier, this will *not* override any protocol that is already set in the URL. If the `conf_force_https` variable is set to `false`, the HTTP protocol will be used. By default, this value is set to `false`.
- The fourth and final configuration variable—`conf_include_document_field_raw_values`—when set to `true`, will force the `Document` object to include the `raw` values gathered from the fields' regular expression patterns. We've discussed configuration settings in detail in the *The WebBot Document Class* section earlier in this book. By default, this value is set to `false`.

Step 8 – the WebBot execution

Now that we have our WebBot class, WebBot Document class, and WebBot bootstrap file completed, we can start testing our bot. Add the following code to the 03_webbot.php file located at project_directory/:

```
<?php
/**
 * WebBot example
 */

// load WebBot library with bootstrap
require_once './lib/WebBot/bootstrap.php';

// URLs to fetch data from
$urls = [
    'search' => 'www.google.com',
    'chrome' => 'www.google.com/intl/en/chrome/browser/',
    'products' => 'www.google.com/intl/en/about/products/'
];

// document fields [document field ID => document field regex
//  pattern, [...]]
$document_fields = [
    'title' => '<title.*>(.*)\<\/title>',
    'h2' => '<h2[^>]*?>(.*)<\/h2>',
];

// set WebBot object
$webbot = new \WebBot\WebBot($urls, $document_fields);

// execute fetch data from URLs
$webbot->execute();

// display documents summary
echo $webbot->total_documents . ' total documents <br />';
echo $webbot->total_documents_success . ' total documents fetched
  successfully <br />';
echo $webbot->total_documents_failed . ' total documents failed to
  fetch <br /><br />';


// check if fetch(es) successful
if($webbot->success)
{
    // display each document
    foreach($webbot->getDocuments() as /* \WebBot\Document */
      $document)
    {
        if($document->success) // was document data fetched
         // successfully?
        {
            // display document meta data
            echo 'Document: ' . $document->id . '<br />';
            echo 'URL: ' . $document->url . '<br />';

            // display/print document fields and values
            $fields = $document->getFields();
            echo '<pre>' . print_r($fields, true) . '</pre>';
        }
        else // failed to fetch document data, display error
        {
            echo 'Document error: ' . $document->error . '<br />';
        }
    }
}
else // not successful, display error
{
    echo 'Failed, error: ' . $webbot->error;
}
```

Primarily, we load our `WebBot` package and its configuration by including the `WebBot` bootstrap file. Next, we set the variable `urls`, which is an `array` of URLs that we want to fetch, and convert it into `WebBot Document` objects. In this example, I am using `www.google.com` as the URL. This is for example purposes only, and you should use your own URLs. Use URLs that use HTML tags such as `<title>*</title>` and `<h2>*</h2>`.

Subsequently, we set the `document_fields` variable with an `array` of field IDs and field values' regular expression patterns. In the previous example, we are defining the document fields: `title` and `h2`. The `title` field will include all values in the URL's data (or HTTP response body) that are in the `<title>*</title>` HTML tags. Likewise, the `h2` field will include all values in the URL's data that are within the `<h2>*</h2>` HTML tags. Again, if you are not familiar with regular expression patterns, you should read more about the topic.

Regular expressions are a very useful tool to have in your programmer's toolbox.

In the next line of code, we set the `webbot` variable with a `WebBot` object. We pass our `urls` and `document_fields` variables to the object constructor method. These are the only parameters required by our `WebBot` object, which makes it very simple to use and understand.

Following the instantiation of the `WebBot` object, we call the `WebBot` object's `execute()` method. This tells the `WebBot` object to start fetching the URL's data by making HTTP requests, and then build a document array of `WebBot` class `Document` objects.

In the next block of code, we test if the `WebBot` object has successfully executed the fetches by checking the `success` class property. If the `success` property is `true`, this doesn't necessarily mean that every URL fetch was executed successfully; it simply means the object was able to call an HTTP GET request for each URL.

In the next section, we loop through each document—that we get from the `WebBot` method `getDocuments()`—and test if the document—or the HTTP response body—was retrieved properly, using the `Document` class property called `success`. If the `Document` class is ready, or was retrieved properly, we display the document ID, the document URL, and print the document fields and field values. Obviously, in real-world applications we would do something more useful with this data, but for this example we can see the results our bot has generated. If the document wasn't retrieved properly, perhaps the HTTP request encountered a 404 status code (request not found). This is where we will display the document error, which is the HTTP status code and status code message.

Step 9 – the WebBot results

When we execute the `project_directory/03_webbot.php` file in a web browser, we should see something like the following:

```
3 total documents
3 total documents fetched successfully
0 total documents failed to fetch

Document: search
URL: http://www.google.com
Array
(
    [title] => Array
        (
            [0] => Google
        )

    [h2] => Array
        (
            [0] => Account Options
        )
```

```
)

Document: chrome
URL: http://www.google.com/intl/en/chrome/browser/
Array
(
    [title] => Array
        (
            [0] =>
      Chrome Browser

        )

    [h2] => Array
        (
            [0] =>
                Customize your browser

            [1] =>
                Get Chrome for Mobile

            [2] =>
                Up to 15 GB free storage

            [3] =>
            Get a fast, free web browser

        )

)

Document: products
URL: http://www.google.com/intl/en/about/products/
Array
(
    [title] => Array
        (
            [0] => Google  - Products
        )

    [h2] => Array
        (
            [0] => Web
            [1] => Mobile
```

```
                        [2] => Media
                        [3] => Geo
                        [4] => Specialized Search
                        [5] => Home & Office
                        [6] => Social
                        [7] => Innovation
                    )

            )
```

We can see from the results that our bot is operating as expected. First, we can see that we fetched a total of three documents, three documents were fetched successfully, and the bot failed to fetch zero documents. We can then see the title, URL, and fields (and field values) for each document. The bot has successfully parsed out each field and field value. This type of bot would be useful for harvesting various types of documents and document fields from desired websites or web services.

The top 5 features you need to know about

Now that we have a working web bot, we can start looking at other features that can be added to our bot to improve its productivity and allow our bot to keep working successfully. In this section, I am going to cover the top five bot features:

- Bot tracing (debug logging)
- Parsing bot data
- Storing parsed bot data
- Bot stealth
- Other advanced features (includes link handling)

Even though we have a functional bot, it doesn't mean we should stop there. Sure, our bot can go out and grab data using simple parsing techniques, however, in real-world projects we are going to want our bot to easily parse data, store the data, and continue to successfully harvest the data. But many times harvesting data can be problematic, since there are many obstacles for bots when harvesting large amounts of data. So, bot tracing will help us debug any issue that might arise while gathering data. Finally, at the end of this section, I will cover link handling; this is a very important aspect of web bot functionality that should not be overlooked.

In this section, we are going to be modifying the code that we have developed in the `WebBot` package. As we will be making modifications to the previous code, we'll create a copy of our original `WebBot` package and call it `WebBot2`. The directory and file structure should look like the following:

```
'-- project_directory
    |-- lib
    |    |-- HTTP (our existing HTTP package)
    |    |    '-- (HTTP package files here)
    |    |-- WebBot (our existing WebBot package)
    |    |    '-- (WebBot package files here)
    |    '-- WebBot2
    |        |-- bootstrap.php
    |        |-- Document.php
    |        '-- WebBot2.php
'-- (other files)
```

Now, edit the WebBot2 package files by making the following modifications in the bootstrap.php file located at project_directory/lib/WebBot2/, which will reflect the new package name:

1. First, change the namespace and comments in our bootstrap.php file, as follows:

```
namespace WebBot2;

/**
 * Bootstrap file
 *
 * @package WebBot2
 */
```

2. Next, change the name of our include paths in the same file:

```
require_once './lib/WebBot2/Document.php';
require_once './lib/WebBot2/WebBot2.php';
```

3. Change the WebBot class name to WebBot2 in the class configuration settings section:

```
\WebBot2\WebBot2::$conf_default_timeout = 30;
\WebBot2\WebBot2::$conf_delay_between_fetches = 1;
\WebBot2\WebBot2::$conf_force_https = false;
\WebBot2\WebBot2::$conf_include_document_field_raw_values =
false;
```

   That's all we need to edit in the bootstrap.php file.

4. Now, we will edit the Document.php file located at project_directory/lib/WebBot2/, and change its namespace, WebBot class name, and comments as follows:

```
namespace WebBot2;

use WebBot2\WebBot2;

/**
 * WebBot Document class
 *
 * @package WebBot2
 */
```

5. Modify the __setFields() method within the Document class, and change the WebBot class name:

```
private function __setFields()
   {
      // more code here

          // set document field raw values if required
          if(isset($m[0]) &&
          WebBot2::$conf_include_document_field_raw_values)
            {
               $this->__fields[$field_id]['raw'] = $m[0];
            }
         }
      }
   }
```

6. Now modify the `WebBot2.php` file located at `project_directory/lib/WebBot2/`, and change its namespace, class name, and comments as follows:

```
namespace WebBot2;

use HTTP\Request;
use WebBot2\Document;

/**
 * WebBot2 class - fetch document data from Website URLs
 *
 * @package WebBot2
 */
class WebBot2
{
    // more code here

    /**
     * Documents
     *
     * @var array (of \WebBot2\Document)
     */
    private $__documents = [];

    // more code here

    /**
     * Documents getter
     *
     * @return array (of \WebBot2\Document)
     */
    public function getDocuments()
    {
        return $this->__documents;
    }
```

Our `WebBot2` package is now ready for further development.

Bot tracing and debug logging

The first improvement we can add to our bot is **tracing**. Tracing can be used to log bot execution messages, which will be useful for debugging or troubleshooting our bot.

1. To add tracing, we first need to add a trace log property to the WebBot2 class located at project_directory/lib/WebBot2/:

   ```
   /**
    * Trace log
    *
    * @var array
    */
   private $__log = [];
   ```

2. Next, we add a log() method to the WebBot2 class located at the same path:

   ```
   /**
    * Add message to log trace
    *
    * @param string $message
    * @param string $method
    * @return void
    */
   private function __log($message, $method)
   {
       $this->__log[] = $message . ' (' . $method . ')';
   }
   ```

3. Finally, we add a getLog() method to the WebBot2 class, at the same path, for retrieving the trace log:

   ```
   /**
    * Trace log getter
    *
    * @return array
    */
   public function getLog()
   {
       return $this->__log;
   }
   ```

Now we can add trace log messages in our WebBot2 class. First, we add trace logging to the WebBot2 constructor, as follows:

```
public function __construct(array $urls)
{
    $this->__urls = $urls;

    if(count($this->__urls) < 1) // ensure URLs are set
    {
        $this->error = 'Invalid number of URLs (zero URLs)';
        $this->__log($this->error, __METHOD__);
    }
    else
    {
        $this->__log(count($this->__urls) . ' URL(s)
          initialized', __METHOD__);
    }
}
```

In this previous constructor method, we add an error property to the trace log for the time when an error occurs; if this error occurs, we simply add a message to the trace log stating Invalid number of URLs (zero URLs).

We next add tracing to the execute() method in the WebBot2 class located at project_directory/lib/WebBot2/:

```
// more code here

$this->__log('Executing bot URL fetches', __METHOD__);

foreach($this->__urls as $id => $url)
{
    // more code here
    }
    else
    {
        $this->error = 'Invalid URL detected (empty URL
          with ID "' . $id . '")';
        $this->__log($this->error, __METHOD__);
    }

    $i++;
}
```

```
$this->__log($this->total_documents . ' total documents',
  __METHOD__);
$this->__log($this->total_documents_success . ' documents
    fetched successfully', __METHOD__);
$this->__log($this->total_documents_failed . ' documents
    failed to fetch', __METHOD__);

// more code here
```

As you can see, we have added useful messages to the trace log in the `execute()` method. While this is a simple example of tracing in our bot, we can continue using tracing as we develop it, which will make debugging and troubleshooting our bot much easier.

Parsing bot data

In our original `WebBot` package, we allowed our `Document` class to parse and extract field data using regular expression patterns, for example:

```
$document_fields = [
    'title' => '<title.*>(.*)<\/title>',
    'h2' => '<h2[^>]*?>(.*)<\/h2>',
];
```

While this is a simple and effective way to parse data, we can modify our package to make it even easier and more productive when handling bulk amounts of harvested data.

1. Let's modify our `WebBot2` package so that it does not require or use the `document_fields` array of fields and its field patterns. Also, adjust the `WebBot2 __construct()` method to eliminate the `document_fields` parameter:

   ```
   /**
    * Init
    *
    * @param array $urls
    */
   public function __construct(array $urls)
   {
       $this->__urls = $urls;

       if(count($this->__urls) < 1) // ensure URLs are set
       {
       // more code here
   }
   ```

2. You can also remove the `WebBot2` class property `__document_fields` since it is no longer used. Now we'll modify the `execute()` method of our package and remove the `__document_fields` logic:

```
public function execute()
    {
        // more code here

                   $this->__documents[$md5] = new Document(
                        Request::get($this-
>__formatUrl($url),
                        self::$conf_default_timeout), $id
                    );
        // more code here
```

3. We need to modify the `Document` class for these changes to be reflected. To do this, remove the `fields` parameter from the `Document  __construct()` method:

```
/**
 * Init
 *
 * @param \HTTP\Response $response
 * @param mixed $id
 */
public function __construct(\HTTP\Response $response,
  $id)
{
    $this->__response = $response;
    $this->id = $id;
    $this->url = $this->getHttpResponse()->getUrl();

    $this->success = $this->getHttpResponse()->success;

    if(!$this->success) // HTTP Response failed
    {
        $this->error = $this->getHttpResponse()-
          >getStatusCode() . ' '
            . $this->getHttpResponse()-
                >getStatusMessage();
    }
}
```

You can now remove the `Document` methods `__setFields()` and `getFields()`, and properties `__fields` and `__fields_and_patterns` since these are no longer utilized.

4. Now we can add some new logic for our new data parsing techniques. The first method we are going to add to the `Document` class located at `project_directory/lib/WebBot2/` is the `test()` method:

    ```
    /**
     * Test if value/pattern exists in response data
     *
     * @param mixed $value (value or regex pattern, if
         regex pattern do not use
     *        pattern modifiers and use regex delims '/',
                ex: '/pattern/')
     * @param boolean $case_insensitive
     * @return boolean
     */
    public function test($value, $case_insensitive = true)
    {
        if(preg_match('#^\/.*\/$#', $value)) // regex
                                             // pattern
        {
            return preg_match($value . 'Usm' . (
              $case_insensitive ? 'i' : '' ),
                $this->getHttpResponse()->getBody());
        }
        else // no regex, use string position
        {
            return call_user_func(( $case_insensitive ?
              'stripos' : 'strpos' ),
                $this->getHttpResponse()->getBody(),
                  $value) !== false;
        }

        return false; // value/pattern not found
    }
    ```

The `test()` method in our `Document` class defines a way in which we can test our response data to determine if a value or pattern exists in it. How is this useful? Well, our bot is designed to gather data from URLs we define. The data our bot gathers is useless unless we use a specific part, or parts of it. The `test()` method allows us to test the data and ensure whether the part, or parts of the data we require are actually contained in the response data or HTTP response body.

Here is an example of how to use the `test()` method in the `Document` class, in the following code:

```
// code before here sets and executes bot

// display each document
foreach($webbot2->getDocuments() as $document)
{
    if($document->test('xyz'))
    {
        echo 'Found "xyz"';
    }
    else
    {
        echo 'Did not find "xyz"';
    }
}
```

As you can see, we test for the value `xyz` in each document that we loop through. This would be useful for when we want to do something with a document that contains the string `xyz`. We can also use regular expression patterns to test for data in the following way:

```
    if($document->test('/xyz/'))
    {
        echo 'Found "xyz"';
    }
    else
    {
        echo 'Did not find "xyz"';
    }
```

Using / pattern delimiters will cause the `test()` method to use the parameter as a regular expression pattern. This is a very simple example, but you will see how this can be used with more complex regular expression patterns.

The `case_insensitive` parameter in the `test()` method is used simply for instructing the method whether to use case-insensitive search logic (in this case `XYZ` will match `xyz`) or case-sensitive search logic (in this case `XYZ` will not match `xyz`).

Now we can create an even more useful method in the `Document` class—located at `project_directory/lib/WebBot2/`—called the `find()` method:

```
    public function find($value, $read_length_or_str = 0,
      $case_insensitive = true)
    {
        if($this->test($value, $case_insensitive))
```

```
{
    if(preg_match('#^\/.*\/$#', $value)) // regex pattern
    {
        preg_match_all($value . 'Usm' . (
          $case_insensitive ? 'i' : '' ),
            $this->getHttpResponse()->getBody(), $m);

        return $m;
    }
    else // no regex, use string position
    {
        $pos = call_user_func(( $case_insensitive ?
          'stripos' : 'strpos' ),
            $this->getHttpResponse()->getBody(), $value);

        if(is_string($read_length_or_str)) // read to
          string position
        {
            $pos += strlen($value); // move position
                                  // length of value
            $pos_end = call_user_func(( $case_insensitive
              ? 'stripos' : 'strpos' ),
            $this->getHttpResponse()->getBody(),
              $read_length_or_str);

            echo "start: $pos, end: $pos_end<br />";
            if($pos_end !== false && $pos_end > $pos)
            {
                $diff = $pos_end - $pos;
                return substr($this->getHttpResponse()-
                  >getBody(), $pos, $diff);
            }
        }
        else // int read length
        {
            $read_length = (int)$read_length_or_str;

            return $read_length < 1
                ? substr($this->getHttpResponse()-
                  >getBody(), $pos)
                // use read length
                : substr($this->getHttpResponse()-
                  >getBody(), $pos, $read_length);
        }
    }
```

```
            }

            return false; // value/pattern not found
        }
```

The `find()` method in the `Document` class is a bit more complex than the `test()` method we created earlier in the same class. However, it is simple and easy to use it to get the exact data we want to extract from the document data.

As we've seen in the `test()` method, the first parameter in the `find()` method is `value`. Also, the `value` parameter can either be a string value that needs to be matched with another, or a regular expression pattern. Here is an example:

```
// code before here sets and executes bot

// display each document
foreach($webbot2->getDocuments() as $document)
{
    $data = $document->find('<title>'); // get '<title>[data]'

    if($data)
    {
        echo $data;
    }
    else
    {
        echo 'Data not found';
    }
}
```

You can see in the preceding example that we are fetching every piece of data after the `<title>` tag in the document data. Likewise, we could use a regular expression pattern:

```
$data = $document->find('/<h1>(.*)<\/h1>/'); // get
// '<h1>[data]</h1>'
```

In the following example, we will use a regular expression pattern to fetch all the data in any `<h1>*</h1>` tag in the document data. This will give an output like the following:

```
Array
(
    [0] => Array
        (
            [0] => <h1>title x</h1>
            [1] => <h1>title y</h1>
            [2] => <h1>title z</h1>
        )

    [1] => Array
        (
            [0] => title x
            [1] => title y
            [2] => title z
        )

)
```

The next parameter in the `find()` method is `read_length_or_str`. This parameter can be used in two different ways. First, it can be used as an integer to instruct the `find()` method to read a certain length. For example, we can use:

```
$data = $document->find('<title>', 8); // get '<title>[data]{8}'
```

This will fetch the data after the `<title>` tag, and return up to eight characters after the tag. So, for example, if the title tag in the document data was `<title>Test title</title>`, the example above would return `Test ti`. Secondly, the `read_length_or_str` parameter can be used as a string. This causes the method to fetch the data up to a specific string location. Here is an example:

```
$data = $document->find('<title>', '</title>'); //get
// '<title>[data]</title>'
```

As you can see, using the `read_length_or_str` parameter value as a string can be very useful for parsing document data.

As we've seen in the `test()` method, the `find()` method utilizes the `case_insensitive` parameter. These two new `Document` methods—`test()` and `find()`—greatly improved our bot's data-parsing abilities. We can now easily fetch specific data from any document's data, or HTTP response body.

Storing data

Often, when harvesting data with a bot, you will want to store the data fetched by the bot. In the previous section, we discussed parsing data, which allowed us to parse and get specific data from the documents our bot gathered. Let's create a `store()` method in the `WebBot2` class, which will allow us to save this data.

1. Add a new property to the `WebBot2` class located at `project_directory/lib/WebBot2/`:

    ```
    /**
     * Directory for storing data
     *
     * @var string
    ```

```
     */
    public static $conf_store_dir;
```

This configuration setting will set the directory where the `store()` method will save the data.

2. Edit the `WebBot2` bootstrap file located at `project_directory/lib/WebBot2/` and add the storage directory location:

```
// storage directory for storing data
\WebBot2\WebBot2::$conf_store_dir = './data/';
```

We are setting `./data/` as the directory for data storage. Or, you can use the full data directory path:

```
\WebBot2\WebBot2::$conf_store_dir =
'/var/www/project_directory/data/';
```

It's important that when you create the data storage directory you make it writable. If you don't know how to do this using your operating system you should find out how to do this now, otherwise, you will not be able to store data using the following method. On my operating system (Linux) from the command line I can make the storage directory writable using the following command:

```
# sudo chmod 777 /var/www/project_directory/data
```

3. Now we can add our `store()` method to the `WebBot2` class:

```
    /**
     * Store data to storage directory file
     *
     * @param string $filename
     * @param string $data
     * @return boolean
     */
    public function store($filename, $data)
    {
        // check if data directory exists
        if(!is_dir(self::$conf_store_dir))
        {
            $this->error = 'Invalid data storage directory
              "' . self::$conf_store_dir . '"';
            return false;
        }

        // check if data directory is writable
        if(!is_writable(self::$conf_store_dir))
        {
            $this->error = 'Data storage directory "' .
              self::$conf_store_dir . '" is not writable';
```

```
                return false;
            }

            // format data directory and filename
            $file_path = self::$conf_store_dir .
              rtrim($filename, DIRECTORY_SEPARATOR) .
              DIRECTORY_SEPARATOR;

            // flush existing data file
            if(is_file($file_path))
            {
                unlink($file_path);
            }

            // store data in data file
            if(file_put_contents($file_path, $data) === false)
            {
                $this->error = 'Failed to save data to data
                  file "' . $file_path . '"';
                return false;
            }

            return true;
        }
```

The `store()` method takes two parameters: `filename` and `data`. The `filename` parameter is used to tell the method what filename to use when creating the new data file. The `data` parameter is used to simply pass the data we want to save in the file that is taken as an argument by the `store()` method.

In the first part of the `store()` method, we check if the data directory exists, and if it doesn't, we set an error message and return `false`. Next, we check if we can write to the data storage directory and if we can't we set an error and return `false` again. We set the `file_path` variable of the data storage directory with the filename. You'll notice we also removed the trailing directory separator, if it exists, and add a directory separator. This will ensure that there is a directory separation separator between the data directory and the filename.

Then, we perform a check to see if the data file we are attempting to save already exists in that exact location, and if it does, we delete the file so that we can save the new data file. Finally, we attempt to save the new data file. If the file is not saved properly, we set an error message and return `false`, otherwise, we return `true`.

Here is an example of how we can put the `store()` method to use:

```
// code before here sets and executes bot

// display each document
foreach($webbot2->getDocuments() as $document)
```

```
{
    $data = $document->find('<title>', '</title>'); // get
     // '<title>[data]'

    if($data)
    {
        if($webbot2->store(urlencode($document->url) . '.dat',
          $data))
        {
            echo 'Data saved <br />';
        }
        else
        {
            echo 'Failed to save data: ' . $webbot2->error . '<br />';
        }
    }
    else
    {
        echo 'Data not found <br />';
    }
}
```

In the preceding example, we get the desired data using the `Document` class `find()` method. Then, if the data is found, we store it in a data file using our new `WebBot2` class `store()` method. We create a safe file name called `store` using the PHP function `urlencode()` and add the file extension `.dat`. This will give us a filename like `http%3A%2F%2Fwww.domain-name.com%2Fpage.html.dat`.

Of course there are better ways of saving data that our bot fetches from URLs. This is just an example of how storing bot data can be achieved. A more useful system would be to utilize a database to store bot data (outside the scope of this book).

Bot stealth

Bot stealth is an important element of using bots in real-world projects. Without bot stealth, web servers can easily decipher which HTTP requests are generated by humans and which HTTP requests are generated by bots. One easy way for a web server to determine where an HTTP request is generated from is using the user agent. When you view a web page on a website using a web browser, the web browser becomes the user agent. For example, if you are using Microsoft Internet Explorer 9, your user agent text will look something like this to a web server:

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64;
Trident/5.0)
```

Web servers log these user agents and then the logs can be used to create statistics for the web browser types that are used and possible bots that are issuing HTTP requests to the web server. For example, a bot owned by Google that is used for indexing website pages for the web search will look something like:

```
Mozilla/5.0 (compatible; Googlebot/2.1;
+http://www.google.com/bot.html)
```

So, what user agent does our bot use when issuing HTTP requests? Well, in our existing HTTP `Request` class, we are not instructing the HTTP requests to use any user agent. Therefore, the user agent is not used. This can be a problem because if a web server cannot determine the user agent type, it can signal the HTTP request an invalid request, or bot request. Often web servers do *not* want rogue web bots harvesting data from their websites, because these bots can disrupt websites and web servers.

We can add a user agent entry in our HTTP request header. To do this, we modify the HTTP `Request` class.

1. Add the `user_agent` property to the `Request` class located at `project_directory/lib/HTTP/`:

```
    /**
     * User agent
     *
     * @var string
     */
    public static $user_agent = 'Mozilla/5.0 (Windows; U;
Windows NT 5.1; en-US; rv:1.9.0.8) Gecko/2009032609
Firefox/3.0.8';
```

2. First, modify the `get()` method and add the user agent as an HTTP request header entry in both the `Request` class methods `get()` and `head()`:

```
    /**
     * Execute HTTP GET request
     *
     * @param string $url
     * @param int|float $timeout (seconds)
     * @return \HTTP\Response
     */
    public static function get($url, $timeout = 0)
    {
        $context = stream_context_create();

        stream_context_set_option($context, [
            'http' => [
                'timeout' =>
                   self::__formatTimeout($timeout),
                'header' => "User-Agent: " .
                   self::$user_agent .
                   "\r\n"
            ]
        ]);

        $http_response_header = NULL; // allow updating

        $res_body = file_get_contents($url, false,
          $context);

        return self::__parseResponse($res_body,
          $http_response_header, $url);
    }
```

3. Modify the `head()` method in the `Request` class:

```
    /**
     * Execute HTTP HEAD request
     *
     * @param string $url
     * @param int|float $timeout (seconds)
     * @return \HTTP\Response
     */
    public static function head($url, $timeout = 0)
    {
        $context = stream_context_create();

        stream_context_set_option($context, [
            'http' => [
                'method' => 'HEAD',
                'timeout' =>
                   self::__formatTimeout($timeout),
                'header' => "User-Agent: " .
                   self::$user_agent . "\r\n"
            ]
        ]);

        $http_response_header = NULL; // allow updating

        $res_body =    file_get_contents($url, false,
          $context);

        return self::__parseResponse($res_body,
          $http_response_header, $url);
    }
```

As you can see we added a `User-Agent` entry in our HTTP request header. Therefore, when the web server receives our HTTP requests it appears as though we are issuing the requests from a Firefox 3 web browser. This is one of the methods of adding stealth to our bot.

Another way to add stealth to our bots is to use a proxy server, which will hide the actual IP address that our bot is being executed from. If you are unfamiliar with proxy servers, you may want to get some knowledge about what they are and how they operate. In simple terms, a **proxy server** acts as an intermediate communicator between two points, or locations, such as between a client and a server. For example, if we were to use a proxy server with our bot, we would be hiding the IP address where our bot is located, because all requests would go through the proxy server. Once the proxy server has the request sent by the bot, it will forward the request (GET, POST, and so on) to the desired URL (or web server) sent by the bot. Then, the proxy server will send the appropriate HTTP request to the actual URL requested by the bot, and it will receive the HTTP response issued by the URL's web server, and return that HTTP response to the bot.

You can easily add a proxy server to your bot using the `file_get_contents()` function:

```
$context = stream_context_create();

stream_context_set_option($context, [
    'http' => [
        'timeout' => self::__formatTimeout($timeout),
        'header' => "User-Agent: " . self::$user_agent .
          "\r\n",
        'proxy' => 'tcp://0.0.0.0:8080', // proxy IP
        'request_fulluri' => true
    ]
]);

$http_response_header = NULL; // allow updating

$res_body = file_get_contents($url, false, $context);
```

The web server wouldn't have access to the IP address where the bot resides, because the request was sent by the proxy server, and because of that, the web server would only have the IP address of the issuing proxy server. This method of using a proxy server is useful because sometimes when web bots are required to harvest data for a large project (for example, millions of URLs), the IP address could get blocked by the URL's web server because the number of requests is huge. If the bot is using a proxy server, we as the programmer can simply switch the proxy server, and therefore switch the IP address for where the bot requests appear to be generated from from which appear to be coming. This is an effective way to mislead web servers. Again, I encourage everyone to follow the website and web service policies and procedures.

Other advanced features

are One feature that would be useful when using our bot is the ability to **crawl** websites. This would make our bot an effective *spider* which that is smart enough to navigate to a website on its own. This may sound like a daunting task, but it is much simpler than it seems.

If I was assigned the task of turning the `WebBot2` class into a working spider, I would start by building a `store()` method that would save the bot data in a database table. I would then parse each document body (HTTP response body) and gather all links in the body. For example, using the `find()` method of our `Document` class, I would do something like the following:

```
$document->find('<a.*href=["\']+(.*)["\']+'); // match <a
href="[data]"
```

The preceding example pulls all the links from the document body into the data variable. We will then write the logic to separate internal links or relative URLs such as `path/page.htm`, from external links or absolute URLs such as `http://www.[domain name].com/path/page.htm`. Then, we will decide which URLs belong to the website that we are crawling. I would store these URLs and document data in the database table. This would allow the bot to do a quick lookup and see if the bot has already crawled or indexed a web page or not.

Another feature that we can add to our web bot is the ability to perform HTTP POST requests. You can easily create a `post()` request method in the HTTP `Request` class by researching how to set the request method type when using the PHP function `file_get_contents()`. Once you have a working `post()` request method, you can pass fields and field values and post POST data to a web server or web service.

People and places you should get to know

In this section, I am going to list a few websites that may be able to assist you in building more advanced bots and furthering your knowledge of botting with PHP.

Helpful sites

- To read more about PHP regular expression functions for parsing bot data visit:

  `http://www.php.net/manual/en/regexp.introduction.php`

- To read about PHP streams using HTTP requests visit:

  `http://www.php.net/manual/en/intro.stream.php`

- To read about PHP Client URL (cURL) library:

  `http://php.net/manual/en/book.curl.php`

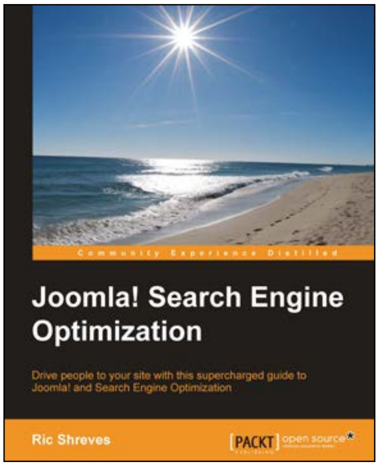
- You can find the PHP Webbot class with a tutorial, harvesting web data, and the HTTP POST method mentioned in detail at:

  `http://www.shayanderson.com/php/php-webbot-class-for-harvesting-web-data.htm`

A warning about using bots

When you are finished reading and understanding this book, you will be able to develop bots that are capable of gathering data from public websites and web services. However, you should be very cautious when gathering data from or sending requests to public websites and web services that you do not own. Always respect public website owners by respecting their website's legal and terms of use policies and procedures.

In the code examples of in this book—available on Packt Publishing's website—sometimes I used example domain names and URLs when I execute bot requests, for example, `www.google.com`. This does *not* mean you should make requests to this domain name or any other domain names or URLs mentioned in this book. These domain names and URLs are used for example purposes only.


Community Experience Distilled
Joomla! Search Engine Optimization
Drive people to your site with this supercharged guide to Joomla! and Search Engine Optimization
Ric Shreves
PACKT open source

Joomla! Search Engine Optimization

ISBN: 978-1-84951-876-5 Paperback: 116 pages

Drive people to your site with this superchanged guide to joomla! and Search Engine Optimization

1. Learn how to create a search engine-optimized Joomla! website.
2. Packed full of tips to help you develop an appropriate SEO strategy.
3. Discover the right configurations and extensions for SEO purposes.

From Technologies to Solutions
PHP 5 e-commerce Development
Create a flexible framework in PHP for a powerful e-commerce solution
Michael Peacock
PACKT

PHP 5 E-commerce Development

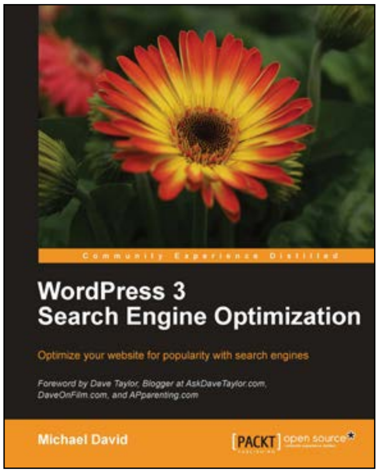
ISBN: 978-1-84719-964-5 Paperback: 356 pages

Create a flexible frmaework in PHP for powerful ecommerce solution

1. Build a flexible e-commerce framework using PHP, which can be extended and modified for the purposes of any e-commerce site
2. Enable customer retention and more business by creating rich user experiences
3. Develop a suitable structure for your framework and create a registry to store core objects

Please check **www.PacktPub.com** for information on our titles

[PACKT] PUBLISHING


Community Experience Distilled
WordPress 3
Search Engine Optimization
Optimize your website for popularity with search engines
Foreword by Dave Taylor, Blogger at AskDaveTaylor.com, DaveOnFilm.com, and APparenting.com
Michael David
[PACKT] open source

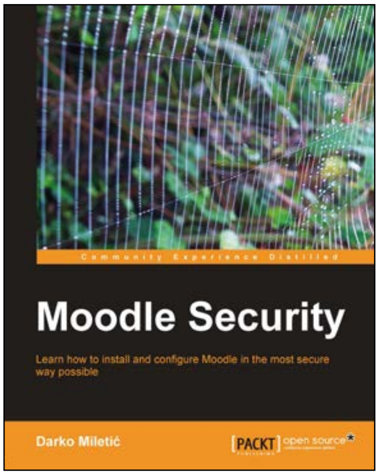
WordPress 3 Search Engine Optimization

ISBN: 978-1-84719-900-3 Paperback: 344 pages

Optimize your website for popularity with search engines

1. Discover everything you need to get your WordPress site to the top of the search engines
2. Learn everything from keyword research and link building to customer conversions in this complete guide
3. Packed with real-word examples to help get your site get noticed by the likes of Google, Yahoo, and Bing

---


Community Experience Distilled
Moodle Security
Learn how to install and configure Moodle in the most secure way possible
Darko Miletić
[PACKT] open source

Moodle Security

ISBN: 978-1-84951-264-0 Paperback: 204 pages

Learn how to install and configure Moodle in the most secure way possible

1. Follow the practical examples to close up any potential security holes, one by one
2. Choose which parts of your site you want to make public and who you are going to allow to access them
3. Protect against web robots that send harmful spam mails and scan your site's information
4. Learn how to monitor site activity and react accordingly

Please check **www.PacktPub.com** for information on our titles