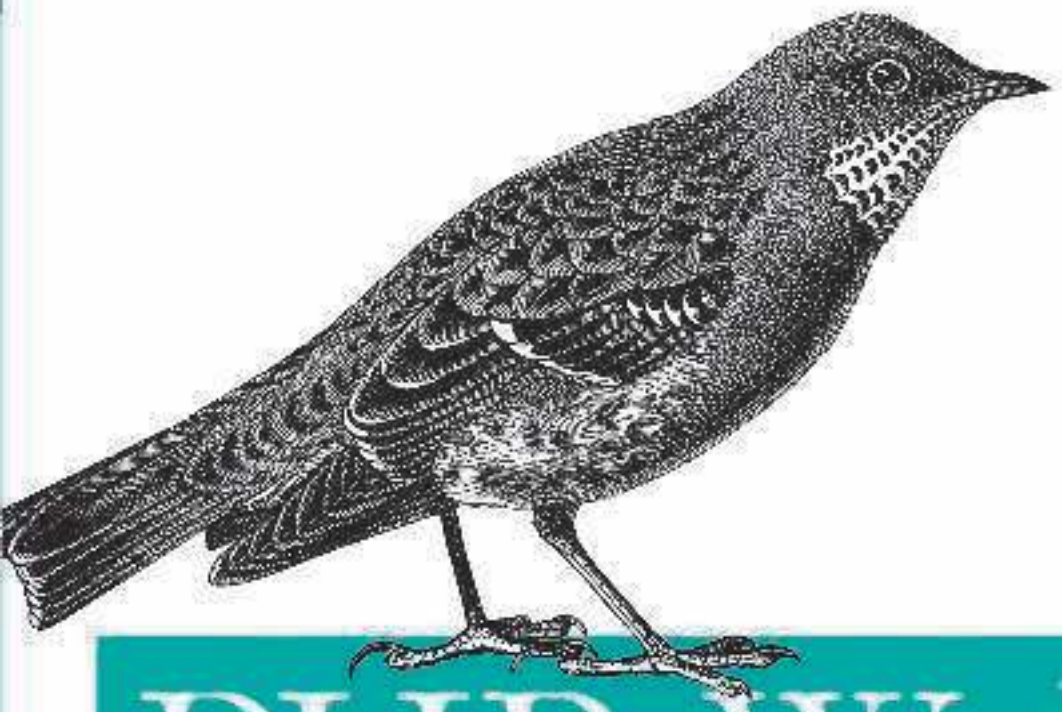
APIs for the Modern Web



PHP Web Services

O'REILLY®

Lorna Jane Mitchell

PHP Web Services

Whether you're sharing data between two internal systems or building an API so users can access their data, this practical book provides everything you need to build web service APIs with PHP. Author Lorna Jane Mitchell uses code samples, real-world examples, and advice based on her extensive experience to guide you through the process—from the underlying theory to methods for making your service robust.

PHP is ideally suited for both consuming and creating web services. You'll learn how to use this language with JSON, XML, and other web service technologies.

- Explore HTTP, from the request/response cycle to its verbs, headers, and cookies
- Determine whether JSON or XML is the best data format for your application
- Get practical advice for working with RPC, SOAP, and RESTful services
- Use a variety of tools and techniques for debugging HTTP web services
- Choose the service that works best for your application, and learn how to make it robust
- Learn how to document your API ... and how to design it to handle errors

US $14.99 CAN $15.99
ISBN: 978-1-449-35656-9

twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

PHP Web Services

Lorna Jane Mitchell

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

PHP Web Services

by Lorna Jane Mitchell

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Maria Gulick and Rachel Roumeliotis
Production Editor: Marisa LaFleur
Proofreader: Marisa LaFleur
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

April 2013: First Edition

Revision History for the First Edition:

2013-04-19: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449356569* for release details.

ISBN: 978-1-449-35656-9

[LSI]

Table of Contents

Preface

In this age, when it can sometimes seem like every system is connected to every other system, dealing with data has become a major ingredient in building the Web. Whether you will be delivering services or consuming them, web service is a key part of all modern, public-facing applications, and this book is here to help you navigate your way along the road ahead. We will cover the different styles of service—from RPC, to SOAP, to REST—and you will see how to devise great solutions using these existing approaches, as well as examples of APIs in the wild. Whether you're sharing data between two internal systems, using a service backend for a mobile application, or just plain building an API so that users can access their data, this book has you covered, from the technical sections on HTTP, JSON, and XML to the "big picture" areas such as creating a robust service.

Why did we pick PHP for this book? Well, PHP has always taken on the mission to "solve the web problem." Web services are very much part of that "problem" and PHP is ideally equipped to make your life easy, both when consuming external services and when creating your own. As a language, it runs on many platforms and is the technology behind more than half of the Web, so you can be sure that it will be widely available, wherever you are. This book does not adopt any particular frameworks; instead, it aims to give you the tools you will need to understand the topic as a whole and apply that knowledge to whichever frameworks, libraries, or other wrappers you choose to use.

The book walks you through everything you need to know in three broad sections. We begin by covering HTTP and the theory that goes with it, including detailed chapters on the request/response cycle, HTTP verbs and headers, and cookies. There are also chapters on JSON and XML: when to choose each data format, and how to handle them from within PHP. The second section aims to give very practical advice on working with RPC and SOAP services, with RESTful services, and on how to debug almost anything that works over HTTP, using a variety of tools and techniques. In the final section, we look at some of the wider issues surrounding the design of top-quality services, choosing what kind of service will work for your application, and determining how to make it robust. Another chapter is dedicated to handling errors and giving advice on why and

how to document your API. Whether you dip into the book as a reference for a specific project, or read it in order to find out more about this area of technology, there's something here to help you and your project to be successful. Enjoy!

Conventions Used in This Book

The following typographical conventions are used in this book:

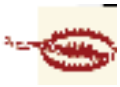
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*PHP Web Services* by Lorna Jane Mitchell (O'Reilly). Copyright 2013 Lorna Jane Mitchell, 978-1-449-35656-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/php-web-services*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

While this is quite a small book on the scale of things, a great many people gave their input to make it happen and they deserve to be acknowledged for the contributions they made.

Several people reviewed early drafts of the book from a technical standpoint and asked many difficult questions at a stage when there was scope for answering them. Thanks to Sean Coates, Jon Phillips, Michele Davis, and Chris Willcock for all their input.

My editors Maria Gulick and Rachel Roumeliotis have been patient and supportive throughout, something I'm sure gets tiring with such a large number of titles coming past at high speed. Their advice and support were invaluable, and I thank them for their gracious help. The rest of the O'Reilly staff have been rockstars also, in particular Josette Garcia, who always makes me believe, and the team that supports the tools I broke so regularly.

My wider "geek support network" has been at once encouraging and providers of practical help. Many people rescued me from my own code samples, gave advice where my own experience fell short, and pointed me to further reading on a variety of topics that made it into this book (and many others that did not). This was very much a hive effort and I consider myself lucky to be part of a community from which help can be requested and given so readily.

Finally, thanks are due to my mystified, but fantastically supportive, family and friends. Chief among these, of course, is my husband, Kevin, who served as cheerleader, proofreader, and head technical support consultant throughout this project and so many others.

CHAPTER 1
HTTP

HTTP stands for HyperText Transfer Protocol, and is the basis upon which the Web is built. Each HTTP transaction consists of a *request* and a *response*. The HTTP protocol itself is made up of many pieces: the URL at which the request was directed, the verb that was used, other headers and status codes, and of course, the body of the responses, which is what we usually see when we browse the Web in a browser.

When surfing the Web, ideally we experience a smooth journey between all the various places that we'd like to visit. However, this is in stark contrast to what is happening behind the scenes as we make that journey. As we go along, clicking on links or causing the browser to make requests for us, a series of little "steps" is taking place behind the scenes. Each step is made up of a request/response pair; the client (usually your browser or phone if you're surfing the Web) makes a request to the server, and the server processes the request and sends the response back. At every step along the way, the client makes a request and the server sends the response.

As an example, point a browser to *http://oreilly.com/* and you'll see a page that looks something like Figure 1-1; either the information desired can be found on the page, or the hyperlinks on that page direct us to journey onward for it.

Figure 1-1. O'Reilly home page

The web page arrives in the body of the HTTP response, but it tells only half of the story. The rest is elsewhere in the HTTP traffic. Consider the following examples.

Request header:

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.8 (KHTML, like Gecko)
Chrome/23.0.1246.0 Safari/537.8
Host: oreilly.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
```

Response header:

```
HTTP/1.1 200 OK
Date: Thu, 15 Nov 2012 09:36:05 GMT
Server: Apache
Last-Modified: Thu, 15 Nov 2012 08:35:04 GMT
Accept-Ranges: bytes
Content-Length: 80554
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Thu, 15 Nov 2012 13:36:05 GMT
Vary: Accept-Encoding
```

As you can see, there are plenty of other useful pieces of information being exchanged over HTTP that are not usually seen when using a browser. Understanding this separation between client and server, and the steps taken by the request and response pairs, is key to understanding HTTP and working with web services. Here's an example of what happens when we head to Google in search of kittens:

1. We make a request to *http://www.google.com/* and the response contains a Location header and a 301 status code sending us to a regional search page; for me that's *http://www.google.co.uk/*.
2. The browser follows the redirect instruction (without confirmation from the user, browsers follow redirects by default) and makes a request to *http://www.google.co.uk/* and recceives the page with the search box (for fun, view the source of this page. There's a lot going on!). We fill in the box and hit search.
3. We make a request to *https://www.google.co.uk/search?q=kittens* (plus a few other parameters) and get a response showing our search results.

In the story shown here, all the requests were made from the browser in response to a user's actions, although some occur behind the scenes, such as following redirects or requesting additional assets. All the assets for a page, such as images, stylesheets, and so on are all fetched using separate requests that are handled by a server. Any content that is loaded asynchronously (by JavaScript, for example) also creates more requests. When we work with APIs, we get closer to the requests and make them in a more deliberate manner, but the mechanisms are the same as those we use to make very basic web pages. If you're already making websites, then you already know all you need to make web services!

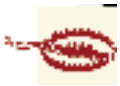
Clients and Servers

Earlier in this chapter we talked about a request and response between a client and a server. When we make websites with PHP, the PHP part is always the server. When using APIs, we build the server in PHP, but we can consume APIs from PHP as well. This is the point where things can get confusing. We can create either a client or a server in PHP, and requests and responses can be either incoming or outgoing—or both!

When we build a server, we follow patterns similar to the way that we build web pages. A request arrives, and we use PHP to figure out what was requested and craft the correct response. For example, if we built an API for customers so they could get updates on their orders programmatically, we would be building a server.

Using PHP to consume APIs means we are building a client. Our PHP application makes requests to external services over HTTP, and then uses the responses for its own purposes. An example of a client would be a page that fetches your most recent tweets and displays them.

It isn't unusual for an application to be *both* a client and a server, as shown in Figure 1-2. An application that accepts a request, and then calls out to other services to gather the information it needs to produce the response, is acting as both a client and a server.



When working on applications like this, take care with how you name variables involving the word "request" to avoid confusion!

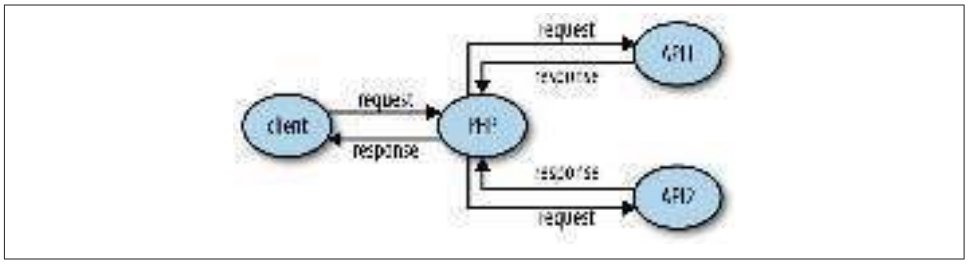


Figure 1-2. Web application acting as a server to the user, but also as a client to access other APIs

Making HTTP Requests

There are a few different ways to communicate over HTTP. In this section, three of them will be covered: Curl, tools in your browser, and PHP itself. The tool you choose depends entirely on your experience and on what it is that you're trying to achieve. We'll also look at tools for inspecting and debugging HTTP in Chapter 9.

The examples here use a site that is set up to log requests made to it, which is perfect for exploring how different API requests are seen by a server. To use it, visit the site and create a new "request bin." You will see the URL needed to make requests to and be redirected to a page showing the history of requests made to the bin. Another excellent way to try making different kinds of requests is to use the reserved endpoints (*http://example.com*, *http://example.net*, and *http://example.org*) established by the Internet Assigned Numbers Authority.

Curl

Curl is a command-line tool available on all platforms. It allows us to make any web request imaginable in any form, repeat those requests, and observe in detail exactly what information is exchanged between client and server. In fact, Curl produced the example output at the beginning of this chapter. It is a brilliant, quick tool for inspecting what's

going on with a web request, particularly when dealing with those outside the usual scope of a browser.

In its most basic form, a Curl request can be made like this (replace the URLs with your own):

```
curl http://requestb.in/example
```

We can control every aspect of the request to send; some of the most commonly used features are outlined here and used throughout this book to illustrate and test the various APIs shown.

If you've built websites before, you'll already know the difference between `GET` and `POST` requests from creating web forms. Changing between `GET`, `POST`, and other HTTP verbs using Curl is done with the `-X` switch, so a POST request can be specifically made by using the following:

```
curl -X POST http://requestb.in/example
```

To get more information from Curl than just the body response, there are a couple of useful switches. Try the `-v` switch since this will show everything: request headers, response headers, and response body in full! It splits the response up, though, sending the header information to STDERR and the body to STDOUT.

When the response is fairly large, it can be hard to find a particular piece of information while using Curl. To help with this, it is possible to combine Curl with other tools such as `less` or `grep`; however, Curl shows a progress output bar in normal operation, which is confusing to these other tools. To silence the progress bar, use the `-s` switch (but beware that it also silences Curl's errors). It can be helpful to use `-s` in combination with `-v` to create output that you can send to a pager such as `less` in order to examine it in detail, using a command like this:

```
curl -s -v http://requestb.in/example 2>&1 | less
```

The extra `2>&1` is there to send the STDERR output to STDOUT so that you'll see both headers and body; by default, only STDOUT would be visible to `less`.

Working with the Web in general, and APIs in particular, means working with data. Curl lets us do that in a few different ways. The simplest way is to send data along with a request in key/value pairs—exactly as when a form is submitted on the Web—which uses the `-d` switch. The switch is used as many times as there are fields to include:

```
curl -X POST http://requestb.in/example -d name="Lorna" -d email="lorna@exam-
ple.com" -d message="this HTTP stuff is rather excellent"
```

APIs accept their data in different formats; sometimes the data cannot be `POST`ed as a form, but must be created in JSON or XML format, for example. In such instances, the entire body of a request can be assembled in a file and passed to Curl. Inspect the previous request, and you will see that the body of it is sent as:

```
name=Lorna&email=lorna@example.com&message=this HTTP stuff is excellent
```

Instead of sending the data as key/value pairs on the command line, it can be placed into a file called *data.txt* (for example). This file can then be supplied each time the request is made. This technique is especially useful for avoiding very long command lines when working with lots of fields, and when sending non-form data, such as JSON or XML. To use the contents of a file as the body of a request, we give the filename prepended with an `@` symbol as a single `-d` switch to Curl:

```
curl -X POST http://requestb.in/example -d @data.txt
```

Working with the extended features of HTTP requires the ability to work with various headers. Curl allows sending of any desired header (this is why, from a security standpoint, the header can never be trusted!) by using the `-H` switch, followed by the full header to send. The command to set the Accept header to ask for an HTML response becomes:

```
curl -H "Accept: text/html" http://requestb.in/example
```

Before moving on from Curl to some other tools, let's take a look at one more feature: how to handle cookies. Cookies will be covered in more detail in a later chapter, but for now it is just important to know that cookies are stored by the client and sent with requests, and that new cookies may be received with each response. Browsers send cookies with requests as default behavior, but in Curl we need to do this manually by asking Curl to store the cookies in a response and then use them on the next request. The file that stores the cookies is called the "cookie jar"; clearly, even HTTP geeks have a sense of humor.

To receive and store cookies from one request:

```
curl -c cookiejar.txt http://requestb.in/example
```

At this point, *cookiejar.txt* can be amended in any way you see fit (again, never trust information that came from outside the application!), and then sent to the server with the next request you make. To do this, use the `-b` switch and specify the file to find the cookies in:

```
curl -b cookiejar.txt http://requestb.in/example
```

To capture cookies and resend them with each request, use both `-b` and `-c` switches, referring to the same *cookiejar* file. This way, all incoming cookies are captured and sent to a file, and then sent back to the server on any subsequent request, behaving just as they do in a browser.

Browser Tools

All the newest versions of the modern browsers (Chrome, Firefox, Opera, Safari, Internet Explorer) have built-in tools or available plug-ins for helping to inspect the HTTP that's being transferred, and for simple services you may find that your browser's tools

are an approachable way to work with an API. These tools vary between browsers and are constantly updating, but here are a few favorites to give you an idea.

In Firefox, this functionality is provided by the Developer Toolbar and various plug-ins. Many web developers are familiar with FireBug, which does have some helpful tools, but there is another tool that is built specifically to show you all the headers for all the requests made by your browser: LiveHTTPHeaders. Using this, we can observe full details of each request, as seen in Figure 1-3.

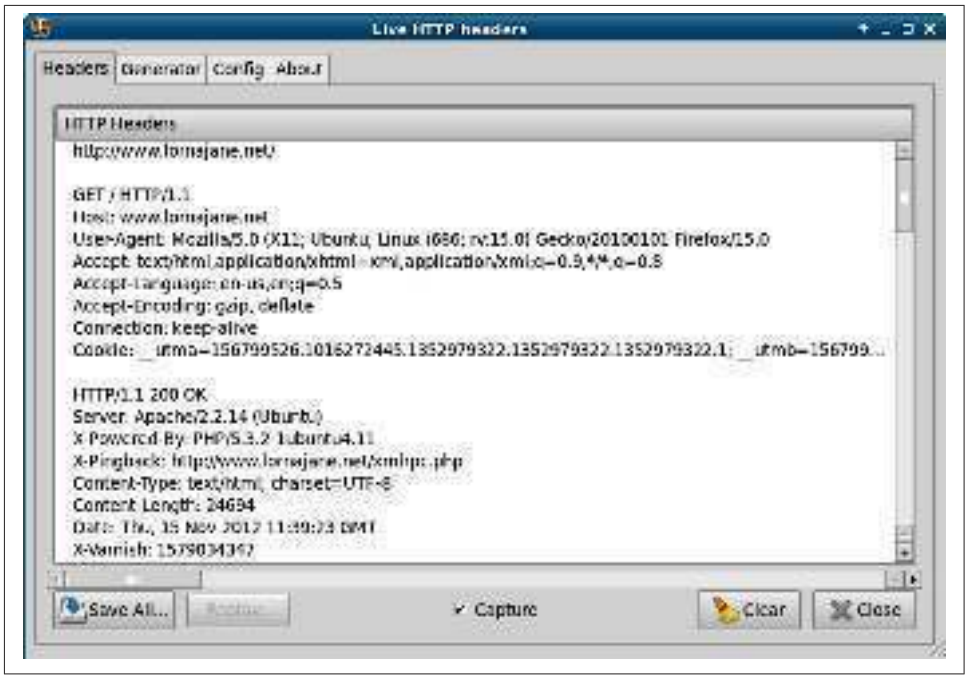


Figure 1-3. LiveHTTPHeaders showing HTTP details

All browsers offer some way to inspect and change the cookies being used for requests to a particular site. In Chrome, for example, this functionality is offered by an extension called "Edit This Cookie," and other similar extentions. This shows existing cookies and lets you edit and delete them—and also allows you to add new cookies. Take a look at the tools in your favorite browser and see the cookies sent by the sites you visit the most.

Sometimes, additional headers need to be added to a request, such as when sending authentication headers, or specific headers to indicate to the service that we want some extra debugging. Often, Curl is the right tool for this job, but it's also possible to add the headers into your browser. Different browsers have different tools, but for Chrome try an extension called ModHeader, seen in Figure 1-4.

Figure 1-4. The ModHeader plug-in in Chrome

PHP

Unsurprisingly, there is more than one way to handle HTTP requests using PHP, and each of the frameworks will also offer their own additions. This section focuses on plain PHP and looks at three different ways to work with APIs: using the built-in Curl extension for PHP, using the `pecl_http` extension, and making HTTP calls using PHP's stream handling.

Earlier in this chapter, we discussed a command-line tool called Curl (see "Curl" on page 4). PHP has its own wrappers for Curl, so we can use the same tool from within PHP. A simple `GET` request looks like this:

```
<?php

$url = "http://oreilly.com";
$ch = curl_init($url);

curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

The previous example is the simplest form, setting the URL, making a request to its location (by default this is a `GET` request), and capturing the output. Notice the use of `curl_setopt()`; this function is used to set many different options on Curl handles and it has excellent and comprehensive documentation on *http://php.net*. In this example, it is used to set the `CURLOPT_RETURNTRANSFER` option to *true*, which causes Curl to *return* the results of the HTTP request rather than *output* them. In most cases, this option should be used to capture the response rather than letting PHP echo it as it happens.

We can use this extension to make all kinds of HTTP requests, including sending custom headers, sending body data, and using different verbs to make our request. Take a look

at this example, which sends some form fields and a `Content-Type` header with the `POST` request:

```
<?php

$url = "http://requestb.in/example";
$data = array("name" => "Lorna", "email" => "lorna@example.com");

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));

curl_setopt($ch, CURLOPT_HTTPHEADER,
    array('Content-Type: application/json')
);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

Again, `curl_setopt()` is used to control the various aspects of the request we send. Here, a `POST` request is made by setting the `CURLOPT_POST` option to 1, and passing the data we want to send as an array to the `CURLOPT_POSTFIELDS` option. We also set a `Content-Type` header, which indicates to the server what format the body data is in; the various headers are covered in more detail in Chapter 3.

The PHP Curl extension isn't the easiest interface to use, although it does have the advantage of being reliably available. A great alternative if you control your own platforms is to add the `pecl_http` extension from PECL. This offers a much more intuitive way of working and has both function and object-oriented interfaces. For example, here's the previous example, this time using `pecl_http`:

```
<?php

$url = "http://requestb.in/example";

$data = array("name" => "Lorna", "email" => "lorna@example.com");

$request = new HTTPRequest($url, HTTP_METH_POST);
$request->setPostFields($data);
$request->setHeaders(array("Content-Type" => "application/json"));

$request->send();
$result = $request->getResponseBody();
```

This extension works more elegantly by creating an `HTTPRequest` object, and then working with the properties on that object, before calling its `send()` method. Once the request has been sent, the body of the response is fetched by calling the `getResponseBody()` method.

Finally, let's look at one more way of making HTTP requests from PHP: using PHP's stream-handling abilities with the file functions. In its simplest form, this means that, if `allow_url_fopen` is enabled (see the PHP manual), it is possible to make a `GET` request using `file_get_contents()`:

```
<?php

$result = file_get_contents("http://oreilly.com");
```

We can take advantage of the fact that PHP can handle a variety of different protocols (HTTP, FTP, SSL, and more) and files using streams. The simple `GET` requests are easy, but what about something more complicated? Here is an example that makes the same `POST` request with headers, illustrating how to use various aspects of the streams functionality:

```
<?php

$url = "http://requestb.in/example";
$data = array("name" => "Lorna", "email" => "lorna@example.com");

$context = stream_context_create(array(
    'http' => array(
        'method' => 'POST',
        'header' => array('Accept: application/json',
            'Content-Type: application/x-www-form-urlencoded'),
        'content' => http_build_query($data)
    )
));

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

Options are set as part of the *context* that we create to dictate how the request should work. Then, when PHP opens the stream, it uses the information supplied to determine how to handle the stream correctly—including sending the given data and setting the correct headers.

As you can see, there are a few different options for dealing with HTTP, both from PHP and the command line, and you'll see all of them used throughout this book. These approaches are all aimed at "vanilla" PHP, but if you're working with a framework, it will likely offer some functionality along the same lines; all the frameworks will be wrapping one of these methods so it will be useful to have a good grasp of what is happening underneath the wrappings. After trying out the various examples, it's common to pick one that you will work with more than the others; they can all do the job, so the one you pick is a result of both personal preference and which tools are available (or can be made available) on your platform.

CHAPTER 2

HTTP Verbs

HTTP verbs such as `GET` and `POST` let us send our intention along with the URL so we can instruct the server what to do with it. Web requests are more than just a series of addresses, and verbs contribute to the rich fabric of the journey.

I mentioned `GET` and `POST` because it's very likely you're already familiar with those. There are many verbs that *can* be used with HTTP—in fact, we can even invent our own—but we'll get to that later in the chapter (see "Using Other HTTP Verbs" on page 15). First, let's revisit `GET` and `POST` in some detail, looking at when to use each one and what the differences are between them.

Making GET Requests

URLs used with `GET` can be bookmarked, they can be called as many times as needed, and the request should not affect change to the data it accesses. A great example of using a `GET` request when filling in a web form is when using a search form, which should always use `GET`. Searches can be repeated safely, and the URLs can be shared.

Consider the simple web form in Figure 2-1, which allows users to state which category of results they'd like and how many results to show. The code for displaying the form and the (placeholder) search results on the page could be something like this:

```
<?php

if(empty($_GET)) {

?>

<form name="search" method="get">
    Category:
    <select name="category">
        <option value="entertainment">Entertainment</option>
        <option value="sport">Sport</option>
```

```
            <option value="technology">Technology</option>
        </select> <br />

        Rows per page: <select name="rows">
            <option value="10">10</option>
            <option value="20">20</option>
            <option value="50">50</option>
        </select> <br />

        <input type="submit" value="Search" />
    </form>

    <?php
    } else {
        echo "Wonderfully filtered search results";
    }
```

Figure 2-1. An example search form

You can see that PHP simply checks if it has been given some search criteria (or indeed any data in the `$_GET` superglobal) and if not, it displays the empty form. If there was data, then it would process it (although probably in a more interesting way than this trivial example does). The data gets submitted on the URL when the form is filled in (`GET` requests typically have no body data), resulting in a URL like this:

```
http://localhost/book/get-form-page.php?category=technology&rows=20
```

The previous example showed how PHP responds to a `GET` request, but how does it make one? Well, as discussed in Chapter 1, there are many ways to approach this. For a very quick solution, and a useful approach to use when working with `GET` requests in particular, use PHP's stream handling to create the complete request to send:

```
<?php

$url = 'http://localhost/book/get-form-page.php';
$data = array("category" => "technology", "rows" => 20);

$get_addr = $url . '?' . http_build_query($data);
```