O'REILLY®

2nd Edition

PHP Web Services

APIs FOR THE MODERN WEB

Lorna Jane Mitchell

www.itbook.store/books/9781491933091

O'REILLY®

PHP Web Services

Whether you're sharing data between two internal systems or building an API so that users can access their data, this practical guide has everything you need to build APIs with PHP. Author Lorna Jane Mitchell provides lots of hands-on 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.

You'll learn how to use this language to work with JSON, XML, and other web service technologies. This updated second edition includes new tools and features that reflect PHP updates and changes on the Web.

- Explore HTTP, from the request/response cycle to its verbs, headers, and cookies
- Work with and publish webhooks—user-defined HTTP callbacks
- Determine whether JSON or XML is the best data format for your application
- Get advice for working with RPC, SOAP, and RESTful services
- Use several tools and techniques for debugging HTTP web services
- Choose the service that works best for your application, and learn how to make it robust
- Document your API—and learn how to design it to handle errors

Lorna Jane Mitchell is an independent web development consultant, specializing in PHP and APIs. With over 10 years of PHP development experience across a wide variety of industries, she also teaches public courses and offers training to private clients around the world. You can contact her via her blog at *http://lornajane.net.*

"*PHP Web Services* is my go-to reference for writing a web service with PHP. I love that this book contains the theory so I can understand the concept, and also includes practical information on how to implement it. Recommended."

—Rob Allen
Consultant, 19FT

WEB DEVELOPMENT

US $29.99 CAN $34.99
ISBN: 978-1-491-93309-1

Twitter: @oreillymedia
facebook.com/oreilly

www.itbook.store/books/9781491933091

PHP Web Services
by Lorna Jane Mitchell

Copyright © 2016 Lorna Mitchell. All rights reserved.

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Allyson MacDonald
Production Editor: Colleen Lobner
Copyeditor: Charles Roumeliotis
Proofreader: James Fraleigh
Indexer: WordCo Indexing Services
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

April 2013: First Edition
January 2016: Second Edition

Revision History for the Second Edition
2016-01-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491933091* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *PHP Web Services*, the cover image of an alpine accentor, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-93309-1

[LSI]

www.itbook.store/books/9781491933091

Table of Contents

Preface vii

1. HTTP 1
Clients and Servers 4
Making HTTP Requests 5
  Command-Line HTTP 6
  Browser Tools 10
  Doing HTTP with PHP 12

2. HTTP Verbs 17
Serving GET Requests 17
Making GET Requests 19
Handling POST Requests 20
Making POST Requests 22
Using Other HTTP Verbs 23

3. Headers 27
Request and Response Headers 28
Identify Clients with User-Agent 29
Headers for Content Negotiation 30
  Parsing an Accept Header 31
  Demonstrating Accept Headers with cURL 33
Securing Requests with the Authorization Header 34
  HTTP Basic Authentication 35
  HTTP Digest Authentication 36
  OAuth 36
Caching Headers 37
Custom Headers 38

www.itbook.store/books/9781491933091

4. Cookies. 41
Cookie Mechanics 41
Reading and Writing Cookies 43
Making Requests with Cookies 44
Cookies and APIs 45

5. JSON. 47
When to Choose JSON 48
Handling JSON with PHP 49
The JSONSerializable Interface 50
Consuming JSON APIs 51

6. XML. 53
XML in PHP 55
Creating XML 55
Consuming XML APIs 58
Parsing XML 58
Flickr's XML API 58

7. RPC and SOAP Services. 63
RPC 63
SOAP 66
WSDL 66
PHP SOAP Client 67
PHP SOAP Server 68
Generating a WSDL File from PHP 70
PHP Client and Server with WSDL 72

8. REST. 75
RESTful URLs 76
Resource Structure and Hypermedia 76
Build the Basic RESTful Server 79
Example Project: The Wishlist 79
Create Resources with POST 82
Fetch a Resource or Collection with GET 84
Update a Resource with PUT 87
DELETE a Resource 88
RESTful Versus Useful 90

9. Webhooks. 91
GitHub's Webhooks 92
Publishing Your Own Webhooks 94

www.itbook.store/books/9781491933091

10. HTTP Tools. 97
Easy Command-Line JSON 98
Graphical cURL Alternatives 100
Inspect HTTP Traffic with Wireshark 101
Tunnel Local Traffic Remotely with ngrok 105
Inspect, Edit, Repeat, and Share Requests 107
Proxying PHP Applications 111
  Proxy Settings for Guzzle 111
  Proxy Settings for HTTP Stream Handling 112
Finding the Tool for the Job 112

11. Maintainable Web Services. 113
Sample API Application 113
Consistent Output Formats 115
Debug Output as a Tool 117
Effective Logging Techniques 120
  Error Logging in PHP Applications with Monolog 122
Error Handling with PHP Exceptions 123

12. Making Service Design Decisions. 127
Service Type Decisions 128
How to Present API Data 129
  Hypermedia for Easy API Navigation 130
  Nested Data or Many Round Trips 130
  Data Formats and Media Types 131
Customizable Experiences 134
Pick Your Defaults 137

13. Building a Robust Service. 139
Consistency Is Key 139
  Consistent and Meaningful Naming 140
  Common Validation Rules 140
  Predictable Structures 141
Error Handling in APIs 142
Meaningful Error Messages 142
What to Do When You See Errors 143
Making Design Decisions for Robustness 144

14. Publishing Your API. 145
Documentation Is Key 145
  Overview Documentation 145
  Generated API Documentation 146

www.itbook.store/books/9781491933091

Interactive Documentation 148
API Description Languages 150
Automated Testing Tools 151
Tutorials and the Wider Ecosystem 154

A. A Guide to Common Status Codes. 157

B. Common HTTP Headers. 161

Index. 163

www.itbook.store/books/9781491933091

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. We'll see more detailed examples later in the book, but this idea of requests and responses consisting of headers as well as body data is a key concept.

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, either on your laptop or your phone) 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://lornajane.net* 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.

The web page arrives in the body of the HTTP response, but it tells only half of the story. There is so much more going on in the request and response as they happen; let's inspect that request to *http://lornajane.net* (a pretty average WordPress blog) in more detail.

www.itbook.store/books/9781491933091

Request headers:

```
GET / HTTP/1.1
Host: www.lornajane.net
Connection: keep-alive
Cache-Control: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/
*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) ...
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
```

Request body: *(no body needed for a GET request)*

Response headers:

```
HTTP/1.1 200 OK
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.6
X-Pingback: http://www.lornajane.net/xmlrpc.php
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Content-Encoding: gzip
Content-Type: text/html; charset=UTF-8
Content-Length: 8806
Date: Tue, 15 Sep 2015 08:43:54 GMT
X-Varnish: 612483212
Age: 0
Via: 1.1 varnish
```

Response body (truncated):

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width" />
<meta name="bitly-verification" content="ff69fb2e45ef"/>
<title>Home - LornaJaneLornaJane | Lorna Jane Mitchell&#039;s Website</title>
<link rel="shortcut icon" href="http://www.lornajane.net/wp-content/themes/lj/
images/favicon.ico">

... (truncated)
```

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. The browser understands how to work with request and response headers, and handles that so the user doesn't need to.

www.itbook.store/books/9781491933091

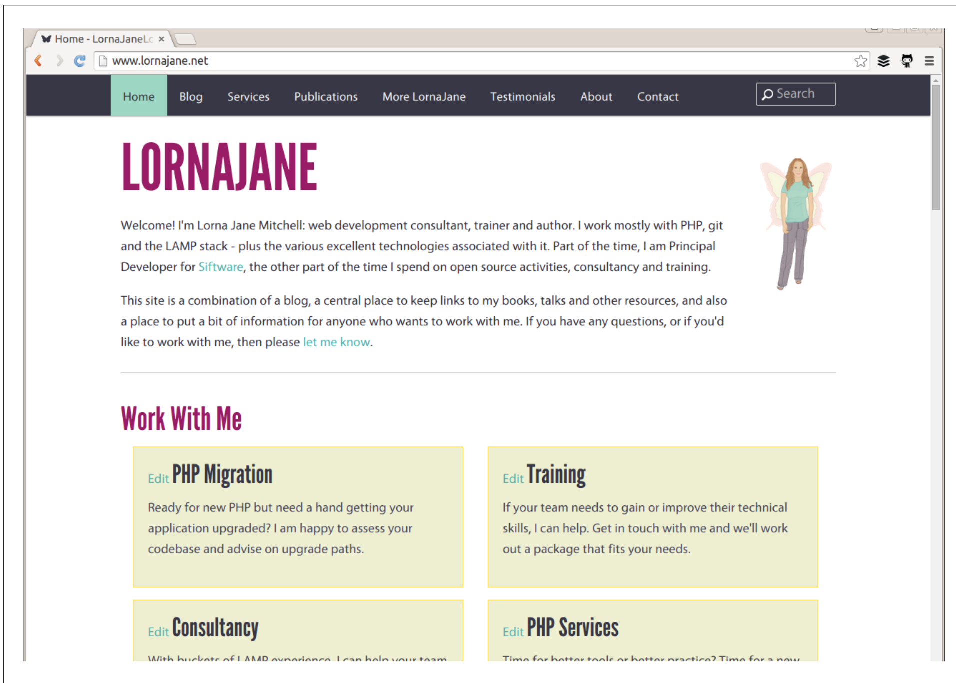

Figure 1-1. Front page of lornajane.net

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), makes a request to *http://www.google.co.uk*, and receives 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.

www.itbook.store/books/9781491933091

The part of the URL after the ? is the "query string" and it's one way of passing additional data to a particular URL or endpoint.

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 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 those we use to 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 script 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 that are APIs or consume APIs, take care with how you name variables involving the word "request" to avoid confusion!

www.itbook.store/books/9781491933091

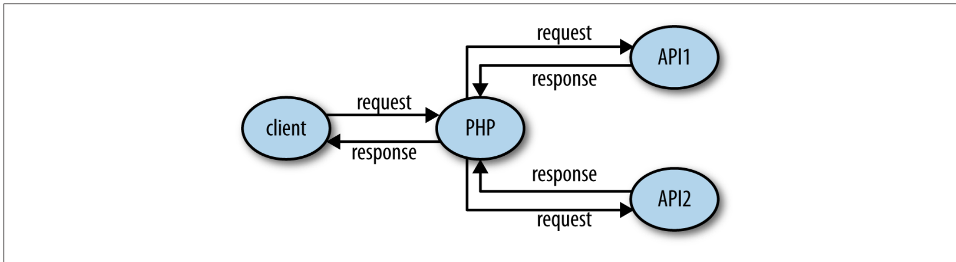

Figure 1-2. Web application acting as a server to the user, but also as a client to access other APIs

Making HTTP Requests

To be able to work with web services, it is important to have a very good understanding of how to work with HTTP from various angles. In this section we'll cover three common ways of working with HTTP:

- Using command-line tools
- Using browser tools
- Using PHP itself

We'll also look at tools specifically designed for inspecting and debugging HTTP in Chapter 10.

The examples here use a site that logs the requests it receives, 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 be given a URL to make requests to and be redirected to a page showing the history of requests made to the bin. This is my own favorite tool, not just for teaching HTTP but also when actually building and testing API clients.

There are a few other tools that are similar and could be useful to you when testing. Try out some of these:

- The reserved endpoints (*http://example.com*, *http://example.net*, and *http://example.org*) established by the Internet Assigned Numbers Authority.
- HTTPResponder is a similar tool and is on GitHub so you could host/adapt it yourself.
- A selection of endpoints with specific behaviors at httpbin.org.

Register your own endpoint at *http://requestb.in* and use it in place of *http://requestb.in/example* in the examples that follow.

www.itbook.store/books/9781491933091

Command-Line HTTP

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 something that isn't in a browser or where you need to be more specific about how the request is made. There's also a cURL extension in PHP; we'll cover that shortly in "Doing HTTP with PHP" on page 12, but this section is about the command-line tool.

In its most basic form, a cURL request can be made like this:

```
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
```

There are also specific switches for `GET`, `POST`, and so on, but once you start working with a wider selection of verbs, it's easier to use `-X` for everything.

To get more information than just the body response, try the `-v` switch since this will show everything: request headers, response headers, and the response body in full! It splits the response up, though, sending the header information to STDERR and the body to STDOUT:

```
$ curl -v -X POST http://requestb.in/example -d name="Lorna" -d
email="lorna@example.com" -d message="this HTTP stuff is rather excellent"
* Hostname was NOT found in DNS cache
*   Trying 54.197.228.184...
* Connected to requestb.in (54.197.228.184) port 80 (#0)
> POST /example HTTP/1.1
> User-Agent: curl/7.38.0
> Host: requestb.in
> Accept: */*
> Content-Length: 78
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 78 out of 78 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
* Server gunicorn/19.3.0 is not blacklisted
```


www.itbook.store/books/9781491933091

```
< Server: gunicorn/19.3.0
< Date: Tue, 07 Jul 2015 14:49:57 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 2
< Sponsored-By: https://www.runscope.com
< Via: 1.1 vegur
<
* Connection #0 to host requestb.in left intact
```

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 if it realizes it isn't outputting to a terminal, which is confusing to these other tools (and to humans). 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`. With the preceding command, you can see the full details of the headers and body, request and response, all available in a pager that allows you to search and page up/down through the output.

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. To make a `POST` request as if I had filled in a web form, I can use a `curl` command like this:

```
curl -X POST http://requestb.in/example -d name="Lorna"
-d email="lorna@example.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. There are dedicated chapters in this book for working with those formats, but in either case we would assemble the data in the correct format and then send it with cURL. We can either send it on the command line by passing a string rather than a key/value pair to a single `-d` switch, or we can put it into a file and ask cURL to use that file rather than a string (this is a very handy approach for repeat requests where the command line can become very long). If you run the previous request and inspect it, you will see that the body of it is sent as:

```
name=Lorna&email=lorna@example.com
```


www.itbook.store/books/9781491933091

We can use this body data as an example of using the contents of a file as the body of a request. Store the data in a file and then 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 the 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 Chapter 4, but for now it is 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* contains the cookies that were returned in the response. The file is a plain-text file, and the way that a browser would store this information is pretty similar; the data is just text. Feel free to open this file in your favorite text editor; it can be amended in any way you see fit (which is another good reminder of why trusting outside information is a bad idea; it may well have been changed), and then sent to the server with the next request you make. To send the cookie jar, amended or otherwise, 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 the `-b` and `-c` switches, referring to the same *cookiejar* file with each switch. 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. This approach is useful if you want to test something from cURL that requires, for example, logging in.

Another command-line tool well worth a mention here is HTTPie, which claims to be a cURL-like tool for humans. It has many nice touches that you may find useful, such as syntax highlighting. Let's see some examples of the same kinds of requests that we did with cURL.

www.itbook.store/books/9781491933091

The first thing you will probably notice (for example, in Figure 1-3) is that HTTPie gives more output.

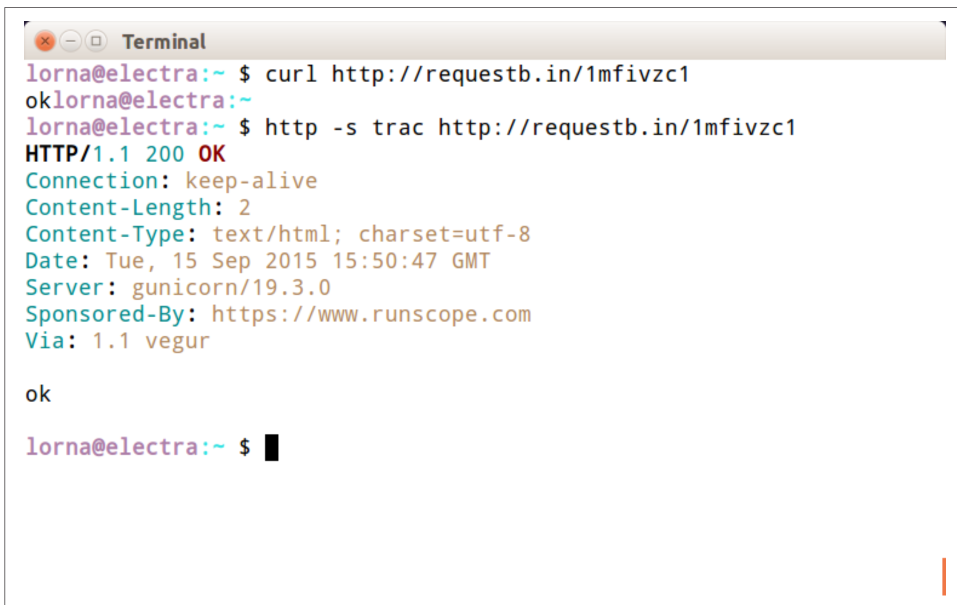

Figure 1-3. A simple GET request with both cURL and HTTPie

You can control what HTTPie outputs with the `--print` or `-p` switch, and pass `H` to see the request header, `B` to see the request body, `h` to see the response header, or `b` to see the response body. These can be combined in any way you like and the default is `hb`. To get the same output as cURL gives by default, use the `b` switch:

```
http -p b http://requestb.in/example
```

HTTPie will attempt to guess whether each additional item after the URL is a form field, a header, or something else. This can be confusing, but once you've become used to it, it's very quick to work with. Here's an example with POSTing data as if submitting a form:

```
$ http -p bhBH -f http://requestb.in/example name=Lorna email=lorna@example.com
message="This HTTP stuff is rather excellent"

POST /example HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 80
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Host: requestb.in
User-Agent: HTTPie/0.8.0
```


www.itbook.store/books/9781491933091

```
name=Lorna&email=lorna%40example.com&message=This+HTTP+stuff+is+rather+excellent

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 2
Content-Type: text/html; charset=utf-8
Date: Tue, 07 Jul 2015 14:46:28 GMT
Server: gunicorn/19.3.0
Sponsored-By: https://www.runscope.com
Via: 1.1 vegur

ok
```

To add a header, the approach is similar; HTTPie sees the `:` in the argument and uses it as a header. For example, to send an `Accept` header:

```
$ http -p H -f http://requestb.in/example Accept:text/html

GET /149njzd1 HTTP/1.1
Accept: text/html
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Host: requestb.in
User-Agent: HTTPie/0.8.0
```

Whether you choose cURL or HTTPie is a matter of taste; they are both worth a try and are useful tools to have in your arsenal when working with HTTP.

Browser Tools

All the newest versions of the modern browsers (Chrome, Firefox, Opera, Safari, Internet Explorer) have built-in tools or available plug-ins to help 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 the full details of each request, as seen in Figure 1-4.

www.itbook.store/books/9781491933091

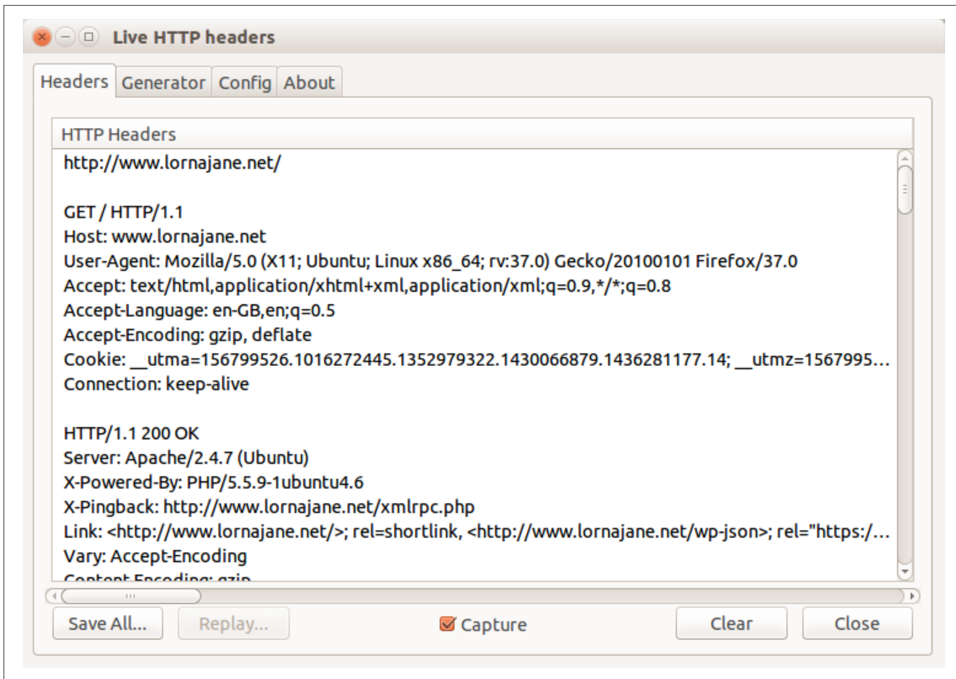

Figure 1-4. LiveHTTPHeaders showing HTTP details

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 1-5.

www.itbook.store/books/9781491933091

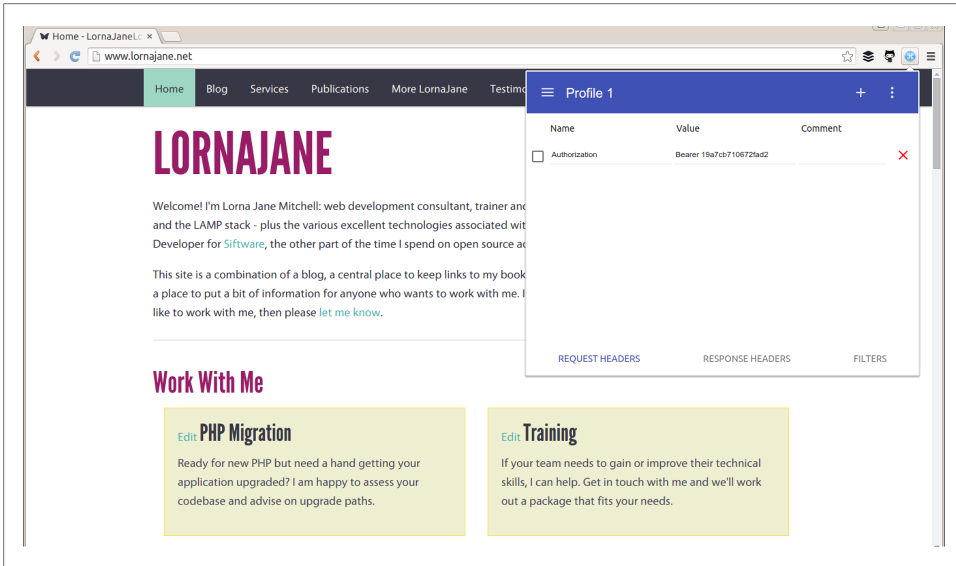

Figure 1-5. The ModHeader plug-in in Chrome

Doing HTTP with PHP

You won't be surprised to hear that 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:

- PHP's cURL extension (usually available in PHP, sometimes via an additional package)
- PHP's built-in stream-handling functionaltiy
- Guzzle (a PHP library)

Earlier in this chapter, we discussed a command-line tool called cURL (see "Command-Line HTTP" on page 6). 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://www.lornajane.net/";
$ch = curl_init($url);

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


www.itbook.store/books/9781491933091

The previous example is the simplest form; it sets the URL, makes a request to its location (by default this is a `GET` request), and capture 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. There aren't many use cases where you'd want to output the response so this flag is very useful.

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 JSON data and a `Content-Type` header with the `POST` request:

```
<?php

$url = "http://requestb.in/example";
$data = ["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($ch, CURLOPT_HTTPHEADER,
    ['Content-Type: application/json']
);
curl_setopt($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. Another great way of making HTTP requests that is always available in PHP is to use 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 requests using `file_get_contents()`. The simplest example is making a `GET` request and reading the response body in as if it were a local file:

```
<?php

$result = file_get_contents("http://www.lornajane.net/");
```


www.itbook.store/books/9781491933091

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 as our earlier example with JSON data and headers, illustrating how to use various aspects of the streams functionality:

```
<?php

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

$context = stream_context_create([
    'http' => [
        'method' => 'POST',
        'header' => ['Accept: application/json',
            'Content-Type: application/json'),
        'content' => json_encode($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.

The third way that I'll cover here for working with PHP and HTTP is Guzzle, a PHP library that you can include in your own projects with excellent HTTP-handling functionality. It's installable via Composer, or you can download the code from GitHub and include it in your own project manually if you're not using Composer yet (the examples here are for version 6 of Guzzle).

For completeness, let's include an example of making the same POST request as before, but this time using Guzzle:

```
<?php

require "vendor/autoload.php";

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

$client = new \GuzzleHttp\Client();

$result = $client->post($url, ["json" => $data]);
echo $result->getBody();
```

The Guzzle library is object-oriented and it has excellent documentation, so do feel free to take these examples and build on them using the documentation for reference.

www.itbook.store/books/9781491933091

The preceding example first includes the Composer autoloader since that's how I installed Guzzle. Then it initializes both the URL that the request will go to and the data that will be sent. Before making a request in Guzzle, a client is initialized, and at this point you can set all kinds of configuration on either the client to apply to all requests, or on individual requests before sending them. Here we're simply sending a `POST` request and using the `json` config shortcut so that Guzzle will encode the JSON and set the correct headers for us. You can see this in action by running this example and then visiting your *http://requestb.in* page to inspect how the request looked when it arrived.

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. Most of my own projects make use of streams unless I need to do something nontrivial, in which case I use Guzzle as it's so configurable that it's easy to build up all the various pieces of the request and still understand what the code does when you come back to it later.

www.itbook.store/books/9781491933091