mojolicious

web clients

brian d foy

{Perl School}

Preface

Just Shut Up And Take My Money!

I've wanted to write this sort of book for 20 years. I'm quick to use web client automation to get things done with onerous web sites or to test web applications. I've spent a lot of time working in the guts of the HTTP protocol to watch things work, and my first presentation at my first Perl conference was about a web sniffer tool, which I hope that no one ever finds because its embarrassing now.

First there was Randal Schwartz's Perl 4-era `chat2.pl`. At the time it was amazing. When I required that library I could easily talk to other things. But that was a simpler time.

Then, Graham Barr's `LWP` (also known as libwww-perl) was the tool everyone used. And it was amazing. I liked it and it solved many problems for me.

Then Mojolicious reinvented the web framework, and it's amazing. It reinvented the web client with some advanced features that I had wished `LWP` had. Someday I may look back on it with the reverence I have for its predecessors, but something many times better would have to exist to displace it.

I had thoroughly enjoyed Lincoln Stein's *Network Programming with Perl*, but it was a comprehensive book about all sorts of network programming. I dreamed of something smaller and more focused on web clients and less about sockets.

But, there was always another book that had my attention. After I finished *Learning Perl 6* (the language now named *Raku*) I wanted to work on something lighter that I could deliver much faster. Instead of 450 pages I wanted to do something in 50 pages (I'm guessing I went over that). I can finally get this web client book off my to-do list.

These sort of books are fun. My *Learning Perl*, *Intermediate Perl*, and *Learning Perl 6* books are tutorial books where I'm limited to using only the features I've previously explained. Not so with this book. This isn't a tutorial. You should try the examples but I won't have exercises. I'll reach for the Perl features that I tend to use when I actually program. If something's a bit tricky I'll probably explain it quickly and move on. Often I won't quickly move on because I can't help myself; don't worry though—I employ no black magic or special cleverness.

I set the goal of releasing a preview version of this book at the 2018 Nordic Perl Workshop in September. That was about three months to pull it all together. How much could I get done? Since I'm not committing it to paper distribution, I could easily fix problems later. As an eBook, I should be able to release this often. The end of this preface has version and change information.

This book is an experiment for this type of publishing. I'll see how it turns out.

What You Should Already Know

This is a book about the web user-agent that comes with the Mojolicious web framework. You don't need to know that much about the web and how it works; you should realize that the web exists, it has a particular protocol, and clients can fetch resources using that protocol.

If you are new to Perl this isn't a good first book. You should know some basic Perl—the level of *Intermediate Perl*. You need to be comfortable using modules and working with objects, but not necessarily creating classes or modules on your own.

Web design, CSS, and other things don't matter that much here—I'm the wrong person to give you any advice on that. This book is strictly about exchanging content and extracting data. It's helpful if you know a little HTML or CSS, but you don't need to know how to make it do anything fancy.

Having said that, I should also point out that I completely ignore the existence of JavaScript. Perl web agents don't handle that for you. It's a big cut-out in the topics you might expect, and I'm not going to include any workarounds for that.

Some eBook Notes

This is an eBook first and any other sort of book after that. Any design decision starts with the eBook presentation even if other forms suffer. Particularly, a PDF version of this book is not part of that design even though it's the form I prefer when I can't have paper.

Will the eBook work? Can I produce more content more frequently for less money? Are people more interested when I can update it frequently (maybe twice a year)? Does that make up for the other trade-offs?

eBooks and their readers weren't designed around technical books with significant code examples. The resolution is quite low which is a problem for normal code organization when you're accustomed to an 80 character line. I suggest that you use the largest page size that you can, but some readers make odd choices about how much of the screen they should give a single page. The reader I like magically switches to a two page layout when the page size gets large enough.

Some lines shouldn't wrap, such as the one-liners in chapter 11, but they do and I preëmpt that. I use the ⮰ character to note that the line is continued on the next line in the book but really belongs together:

```
% perl -Mojo -E "p('http://example.com' =>⏎
    form => { robot => 'Bender'})"
```

In other places I've made code style concessions to fit in the format and the behavior of the eBook readers that I've tested. Your favorite reader might handle long lines just fine, but some other readers have problems. With this experiment I can figure out what works the best; let's hope that's at least good.

Let me know what works for you and send me screenshots of the places where your eBook reader doesn't do what you think I meant.

Installing Mojolicious

Mojolicious 9 moved to Perl v5.16 as the minimum version and will soon move to requiring v5.20 so the code can use subroutine signatures. Those versions are already a decade old, so now's a good time to upgrade.

Mojolicious is a mostly self-contained system. Start by installing the basic framework with your favorite CPAN client:

```
% cpan Mojolicious
% cpanm Mojolicious
```

Use the Perl Package Manager (**ppm**) if you are using ActivePerl:

```
% ppm install Mojolicious
```

Check the version you installed. I originally based this book on Mojolicious 8 and have updated it for version 9; the examples should work with both in most cases, or you should see an example for each. If you have an earlier version, upgrade! If you have a later version some things might have changed but are probably a small adjustment:

```
% mojo version
```

```
CORE
  Perl        (v5.32.1, darwin)
  Mojolicious (9.16, Waffle)

OPTIONAL
  Cpanel::JSON::XS 4.09+   (4.25)
  EV 4.32+                 (n/a)
  IO::Socket::Socks 0.64+  (n/a)
  IO::Socket::SSL 2.009+   (2.070)
  Net::DNS::Native 0.15+   (n/a)
  Role::Tiny 2.000001+     (2.002004)
  Future::AsyncAwait 0.36+ (n/a)

This version is up to date, have fun!
```

Notice that it also lists optional modules that Mojolicious can use if you have them installed. Remember this command in case you have to report an issue; you should include this output in your report.

Check your version of OpenSSL and install the latest `IO::Socket::SSL`. These ensure that you'll be able to access servers using SSL and all of its fancy features:

```
% openssl version
LibreSSL 2.2.7
% cpan IO::Socket::SSL
```

Your particular system may have specialized ways to package and manage modules, libraries, and tools. You'll have to figure out that end on your own.

Once you've installed Mojolicious, try it out. There's a `ojo` module designed to load from the command line; with `-M` it looks like "Mojo":

```
% perl -Mojo -E 'say g(shift)->body'⏎
    https://www.mojolicious.org
```

That `g()` does a GET request and returns the response. The `body` method extracts the content. To look at only the headers (a common task in working out automation tasks) use `h()` to do a HEAD request:

```
% perl -Mojo -E 'say h(shift)->headers⏎
    ->to_string' https://www.mojolicious.org
```

If those worked out you're ready for everything in this book.

Third-party tools

Web client programming isn't just Perl. You'll use interactive browsers (so, Safari, Chrome, Firefox, Opera) to figure out what's happening so you can implement that in code. These are more important than they used to be; the world is moving toward ubiquitous HTTPS so the secure networking precludes external sniffer tools. I won't explain these tools here but they shouldn't be too hard to figure out:

- In Safari, turn on the developer tools in the Advanced tab of preferences. You should see a Develop menu item after you do that.
- For Chrome and Firefox, look for extensions like "HTTP Headers". There are many of varying quality. Try them until you find one that you like.
- The Postman tool on macOS has been helpful to me for exploring APIs.
- httpbin.org has a simple server that can show you what's happening in HTTP transactions. There's even a dockerized version of it. I'll use that in many examples.

Getting Help

I don't have special techniques for debugging Mojo code. I still use my favorite debugger that's available in almost any programming language: `print` (or `puts` or `println` or …).

Running my programs with the environment variable `MOJO_CLIENT_DEBUG` set to some true value shows me what's happening underneath all of my client code. That might elucidate my problem. You'll see an example on that in chapter 1.

Pare down your situation to a small program that demonstrates the problem. Remove as many distractions as possible; often that gives me my answer. Not only do I solve the problem but I learn about how things work and how I probably misunderstood things. If I can't figure it out I have all the makings of a good question.

There are many people waiting to help. Develop a good question and include the versions you are using, what you expected to happen, and how that is different than what you got.

- Mojolicious website
- Mojolicious Google Group
- #mojo on IRC
- Stackoverflow
- The GitHub project

Acknowledgments

The Mojolicious core developers have been quite kind in their support of the book. Sebastian Riedel even designed the cover after he saw my kludgey attempt. Joel Berger reviewed early drafts and offered great advice. Marcus Ramberg was quite helpful and supportive as well. The Oslo Perl mongers invited and sponsored me for the 2018 Nordic Perl Workshop where I presented some of the book and worked on it in the hackathon.

David Cross, the publisher of Perl School, gave me the impetus to try an eBook. He's publishing low-cost, easy-to-update eBooks on a variety of niche subjects that traditional publishers won't invest in. He walked me through

the publishing process he developed after trying many different ways his idea could work.

Several people have provided helpful edits that led to updates, including Mark Anderson, Tigran Khachikyan, Tim Potapov, and Gabor Szabo.

Perl School

The Perl School brand has its roots in a series of low-cost Perl training courses that Dave Cross ran in 2012. By running low-cost training at the weekend, he hoped to encourage more programmers to keep their Perl knowledge up to date. These courses were run regularly for about a year before the idea was put on hold for a while.

Dave always knew that he would want to return to the Perl School brand at some point and late in 2017 he realised what the obvious next step was low-cost Perl books. He had already developed a pipeline for creating e-books from Markdown files so it was a short step to republishing some of his training materials as books.

The first Perl School book, Perl Taster was published at the end of 2017 (just in time for the London Perl Workshop). The second was Selenium and Perl and this is the third. Dave has plans to publish more over the coming months.

If you are interested in writing a book for the Perl School range, then please get in touch. We are @perl_school on Twitter.

Changes

January 2024

- More small fixes from Brigham Johnson

December 2023

- Various small fixes from Brigham Johnson

June 2023

- Various small fixes from Hussam Qasem

May 2021

- Various small fixes, mostly from Tim Ka.

May 2021

- Update chapter 5 for `:matches` renamed to `:is` (Mojo 8.42)
- Update chapter 5 for `:has` and `:scope` (Mojo 8.54)
- Note that Mojolicious 9 exists and requires v5.16
- Update chapter 5 for `:text` (Mojo 9.14)
- Minor text fixes from Jan Jona Javoršek

October 2020

- `Mojo::DOM` example for building a macOS plist from scratch (chapter 4)
- `Mojo::DOM` examples for finding next and previous sibling nodes (chapter 5)

May 2020

- Tigran Khachikyan provided edits to minor technical mistakes in examples for `Mojo::Collection` and counting elements with postfix dereferences.

April 2020

- Added an example for processing a response in chunks (chapter 8)
- Added a few more one-liner examples (chapter 11)

February 2020

- Minor fixes

January 2020

- Minor fixes

December 2019

- First release

Chapter 1

Introduction

My story is a lot like yours, only more interesting ‘cause it involves robots.

This chapter looks a little bit at Mojolicious and its philosophy. I’ll show some simple programs to review basic HTTP principles while I introduce the framework itself.

The Mojo Philosophy

The Mojolicious framework is self-contained. It needs only core Perl; changes in CPAN modules have minimal impact on the basic operations. It handles just about anything that you need to do in a web application or client. That includes a couple of web servers to run your application, but that’s the subject of a different book.

The client, however, isn’t everything you’d need for a full-fledged interactive web browser (for instance, it’s not going to handle a graphic interface or run JavaScript) but it has most things you’d need to automate web tasks.

Mojo reinvented some wheels but those wheels came out even better. Since almost everything is under the control of the Mojolicious team, they can fix, extend, and control almost every aspect of your experience. This gives you a consistent experience across the entire framework.

However, the developers move as quickly as they want to go even if they might break things. And, they do sometimes break backward compatibility, but they tend to give fair warning and wait for a major version change. There is a three-month deprecation period, so parts of this book might be outdated by the time you read it. When that happens I'll try to update the book.

Mojo only guarantees that it works with supported *perl*s. That's the current and previous stable versions. This saves time testing and working around older versions. As I write this, that's v5.28 and v5.30 (see official perl support policy). Mojo may work on early versions of *perl* but don't rely on it. If you are worried about that, don't update things. And, even if you aren't worried about that, try things before updating. Come up with some system to rollback changes to a known good system. That's just good advice no matter what you are using.

Futurama

Many of the examples and sample data in the Mojo documentation draw from the animated series Futurama. I'll continue that here. The pithy statements and references probably come from that show. You don't need to know anything about the show to understand the structure and purpose of the code, but it can't hurt. I'll leave it to you and your web browser to discover who Bender, Fry, Leela, and the others are.

Fluent programming

The Mojo interface is an example of "fluent programming"—a type of interface from Eric Evans and Martin Fowler that relies on chained method calls where most methods return another object (which you call more methods on):

```
use Mojo::UserAgent;

# shift() outside a subroutine works on @ARGV
say Mojo::UserAgent->new->get( shift )->result->body;
```

That's a soup of text where it's hard to pick out what's going on. I like to rewrite these things to highlight major operations on their own line (and it works out better for eBooks):

```
say Mojo::UserAgent->new
    ->get( shift )
    ->result
    ->body;
```

The `new` returns a `Mojo::UserAgent` object. The `get` returns the transaction object, which includes the request and the response. The `result` extracts the response object, and the `body` gets its content. At first you'll struggle with the documentation to keep all this straight, but you'll get used to it.

If something goes wrong, that code will `croak`. Most often you'll make the transaction then check that it works. This is generally a good idea with anything that interacts with the outside world. Things fail for all sorts of reasons, many outside of your control:

```
use Mojo::UserAgent;

my $tx = Mojo::UserAgent->new->get( shift );
unless( $tx->result->success ) {
    die "There was a problem\n";
    }

say $tx->result->body;
```

Often I'll define a method in `UNIVERSAL` (a particular no-no I don't recommend for production code). Every object will inherit these methods; I can call the `NAME` method on any object to see the class name for an object:

```
package UNIVERSAL {
    sub NAME { ref $_[0] }
    }

use Mojo::UserAgent;

my $tx = Mojo::UserAgent->new->get( shift );
say $tx->NAME;          # Mojo::Transaction::HTTP
unless( $tx->result->is_success ) {
    die "There was a problem\n";
    }

say $tx->result->NAME;  # Mojo::Message::Response
```

In some code examples in Mojolicious, you'll see a code reference used as a method. It's valid Perl. The argument has the invocant as the first element so you can use that to do more localized debugging:

```
my $namer = sub { ref $_[0] };

...

say $tx->result->$namer();  # Mojo::Message::Response
```

Some future version of Perl might have reflection features that make this sort of trickery obsolete.

Be Nice to Servers

This is a book essentially about how to annoy servers with your own programs. Sometimes these programs are called "bots" because they operate on their own without supervision. Sometimes they can go awry and wreak havoc with too many requests, repetitive requests, and so on. These are typically different from the programs that connect to a couple of websites to grab particular resources then shut down.

Many of the things that you'll likely want to create will be a force for good: automating repetitive tasks, collating information, and other things. However, not everyone sees it that way. It's their server so it's their rules even if I don't like them.

Some sites have Terms of Service. If you want to interact with these sites with your own program, read that agreement. Some sites might outright disallow it, but others have instructions on how to do it within their frameworks.

Don't send several hundred requests to a server right away. Spread them out a bit by pausing slightly between requests. A couple requests a second is a rule that I like to use (and the rate that many rate-limiting APIs seem to like). You don't want to draw too much attention to your program lest the server stops responding to you (or worse, complaining to your hosting provider).

Remember that your actions can cost the server side real money in bandwidth costs. You'd be surprised how many things that look like sophisticated big businesses are really some guy at his kitchen table.

/robots.txt

Servers might have a `/robots.txt` file that specifies some of their rules according to the Standard for Robot Exclusion. This was a more popular thing a couple decades ago and seems to have fallen out of favor. I'm guessing most bots didn't care and servers developed much better methods to respond to unfriendly bots).

It specifies rules by the user-agent string. You should compare your user-agent identifier to the rules to see what the servers wants you to exclude. This one wants Mojo programs to ignore everything:

```
User-agent: Mojolicious (Perl)
Disallow: /
```

Perhaps it tells you that parts of the website are inappropriate for any bots by using a * as the user-agent string:

```
User-agent: *
Disallow: /cgi-bin/gnarly-db-query/
```

Maybe it tells you that everything is disallowed:

```
User-agent: *
Disallow: /
```

Although many servers don't provide these, some of the big web crawlers (such as Google) look for and respect them. Remember that when you start creating your own Mojolicious servers (but not in this book!).

I assume that you have permission to do what you are doing. If it's something worthwhile you're likely to want to keep doing it and not be shut out from the server. Play nice so you can keep playing.

How HTTP Works

HTTP (and HTTPS, the secure version) work on a request-response protocol. The client sends a request and the server sends a response. On top of that there are various tricks to make things more efficient but you don't need to worry about that for this book. Or, when that's not sufficient you'll read more about that topic.

Most of this is defined in RFC 7230 and RFC 7231. If you want to do high-powered web client programming you should study these documents. Mojo handles the mechanics but you need to know what to tell the server.

HTTP messages

Each request and response is an HTTP "message". There are three parts—the "start line", the "header", and the "message body". Here's an example:

```
POST / HTTP/1.1
```

```
Host: www.example.com
Content-Length: 10

name=value
```

The first line is the start line of the request. It gives the HTTP verb, the path, and the HTTP version. The headers immediately follow and are line-oriented collection of names and values. The message body is set apart from the headers by a blank line (technically, double carriage return / newline pair).

The request triggers a response. Its start line includes the protocol version, the HTTP status number that describes what happened, and a string version of that status. It has headers and possibly a message body:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 6

Bender
```

The headers

The header contains various metadata, instructions, and other tracking things. The message body is the content. That might be HTML, JSON, or almost anything else. A request can send a body to the server and the server can send a body back. Or either can have an empty body.

The headers are line-oriented. There's a header name, a colon, and the header value. Here's a short program to print the request header then the response header. After you make the request you get a transaction object in `$tx` (the customary name). That transaction has both the request (`req`) and response (`result`). The `headers` extracts that part of the message. The `to_string` formats it nicely for you:

```
use Mojo::UserAgent;
```

```
my $tx = Mojo::UserAgent->new->get( shift );
unless( $tx->result->is_success ) {
    die 'Could not make request';
    }

say "Request\n", '-' x 50, "\n",
    $tx->req->to_string;
say "Response\n", '-' x 50, "\n",
    $tx->result->headers->to_string;
```

The output shows each side:

```
% ./show_transaction https://www.mojolicious.org/
Request
--------------------------------------------------
Content-Length: 0
Accept-Encoding: gzip
User-Agent: Mojolicious (Perl)
Host: www.mojolicious.org

Response
--------------------------------------------------
Content-Type: text/html;charset=-UTF8
Connection: keep-alive
Server: cloudflare
Date: Wed, 20 Jun 2018 01:13:45 GMT
Content-Length: 13420
Expect-CT: max-age=604800, report-uri="..."
CF-RAY: 42da5fe7cd5421d4-EWR
Set-Cookie: __cfduid=...
```

The request headers output in the previous section excludes the start line of the transaction, which specifies the request verb, the resource path, and the protocol version:

```
GET / HTTP/1.1
```

You can reconstruct this yourself from the request object since it has all the information stored in various internal objects:

```
say join ' ',
    $tx->req->method,
    $tx->req->url->path,
    'HTTP/' . $tx->req->version
    ;
```

You'll often want to use an interactive browser to watch this happen so you can figure out what you need to tell Mojo to do. The "Show Developer Tools" in Safari's Advanced preferences gives you the Develop menu. The HTTP Headers extension for Chrome or the HTTP Headers Live extension for Firefox are useful. There are various other developer tools that show you the HTTP transactions.

Request verbs

Each request specifies a "verb" and each of these has a particular intent and meaning. In this book you'll see them in all uppercase to distinguish them from normal text. A "resource" is a fancy word for data the server or client provides.

These are the HTTP methods:

- GET - Return something but don't change the server state (idempotency)
- HEAD - Return the header for a resource
- POST - Here are some data; do something with it that possibly changes the server state
- PUT - Create (or replace) this resource on the server
- PATCH - Change this resource on the server
- DELETE - Remove this resource from the server

- OPTIONS - Return information about what you can do to the resource

Some of these are *idempotent*, meaning that you can request that resource as often as you like without changing the state of the server. You make the same request over and over and the server stays the same. Think about viewing your bank's balance. Check as often as you like without that changing the number you see. The GET and HEAD requests are supposed to be like that.

Other requests may change the state of the server (but not necessarily). Your request might add or update a record in the database. If you repeated your request you might create additional database records or change the server in some other way. You don't want to repeat those or make duplicate records. All of the other request methods besides GET and HEAD are like these.

There's nothing particularly special about these methods or their names. There are others out there. You can make up your own if you like; you just need a server that understands it.

Each of the HTTP methods have a convenience method. To make a HEAD request to get just the headers and metadata for a resource, use the `head` method:

```
use Mojo::UserAgent;
my $ua = Mojo::UserAgent->new;
my $tx = $ua->head( 'https://www.mojolicious.org/' );
```

Response codes

The first line of the response also has a start line. It shows the protocol version, the HTTP status code (a number), and the status description:

```
HTTP/1.1 200 OK
```

The stringification of the request headers didn't show this line but you can reconstruct it:

```
say join ' ',
    'HTTP/' . $tx->result->version,
    $tx->result->code,
    $tx->result->message,
    ;
```

Each group of hundreds is a type of status and they are the code that Mojo uses to determine if the request succeeded or failed:

- 100 - Informational
- 200 - Successful request
- 300 - Redirection
- 400 - An error in the request
- 500 - An error in the server

Expand the earlier program to be more particular about the status of the response. These methods come from the response object (so look in `Mojo::Message::Response`):

```
use Mojo::UserAgent;

my $ua = Mojo::UserAgent->new;
my $tx = $ua->get( shift );

if( $tx->result->is_success or
    $tx->result->is_redirect ) {
    say "Request\n", '-' x 50, "\n",
        join( ' ',
            $tx->req->method,
            $tx->req->url->path,
            'HTTP/' . $tx->req->version
            ), "\n",
        $tx->req->headers->to_string;

    say "\nResponse\n", '-' x 50, "\n",
        join( ' ',
            'HTTP/' . $tx->result->version,
            $tx->result->code,
            $tx->result->message
            ), "\n",
        $tx->result->headers->to_string;
    }
elsif( $tx->result->is_error ) {
    if( $tx->is_server_error ) {
        say "Oops! Server Error!";
        }
    else {
        say "A 4xx error";
        }
    }
```

Try this with a working address and you get mostly the same thing. But try it with HTTP instead of HTTPS:

```
% ./show_transaction http://www.mojolicious.org/
Request
--------------------------------------------------
GET / HTTP/1.1
Host: www.mojolicious.org
Accept-Encoding: gzip
User-Agent: Mojolicious (Perl)
Content-Length: 0

Response
--------------------------------------------------
HTTP/1.1 301 Moved Permanently
Server: cloudflare
Location: https://www.mojolicious.org/
Content-Length: 0
Expires: Wed, 20 Jun 2018 03:38:53 GMT
Connection: keep-alive
Date: Wed, 20 Jun 2018 02:38:53 GMT
Cache-Control: max-age=3600
CF-RAY: 42dadc9f367391a6-EWR
```

Now there's a different sort of response. The status is 301. This domain immediately redirects to the HTTPS site. The status message says "Moved Permanently" which normally means your client should remember the new location and use that preferentially next time.

By default Mojo doesn't follow that redirection for you. Set the `max_redirects` to some positive value and Mojo will follow these redirections up to that many times. Three is a good number (because you can get into infinite loops):

```
my $ua = Mojo::UserAgent->new;
$ua->max_redirects(3);
```

This works as one method chain too because the methods to configure the user-agent object return that same object:

```
my $ua = Mojo::UserAgent->new->max_redirects(3);
```

The same request ends up with a different response now. Mojo saw the redirection, looked in its `Location` header, and made another request for that URL. You didn't have to look at each step of the process:

```
% ./show_transaction http://www.mojolicious.org/
Request
--------------------------------------------------
GET / HTTP/1.1
User-Agent: Mojolicious (Perl)
Host: www.mojolicious.org
Accept-Encoding: gzip
Content-Length: 0

Response
--------------------------------------------------
HTTP/1.1 200 OK
Connection: keep-alive
Set-Cookie: __cfduid=...
Expect-CT: max-age=604800, ...
Content-Type: text/html;charset=-UTF8
CF-RAY: 42daf614cc229260-EWR
Content-Length: 13420
```

```
Server: cloudflare
Date: Wed, 20 Jun 2018 02:56:16 GMT
```

It gets even better. The $tx knows about the previous HTTP requests and responses that led to the final one. If there were a redirect and a new transaction, you can see the previous transaction. Wrap that output in a loop to show all of the transactions leading to the final response:

```
use Mojo::UserAgent;

my $ua = Mojo::UserAgent->new->max_redirects(3);
my( @txs ) = $ua->get( shift );

while( my $previous_tx = $txs[0]->previous ) {
    unshift @txs, $previous_tx;
    }

foreach my $tx ( @txs ) { ... }
```

For me, this interface for the transaction object and everything that happened is a big benefit over LWP. Everything is wrapped into a tidy transaction package.

To see everything that happened, set the MOJO_CLIENT_DEBUG environment variable to a true value. Exporting that variable sets it for the rest of the session. This uses *bash* syntax to set the value for the entire session:

```
% export MOJO_CLIENT_DEBUG=1
% ./show_transaction http://www.mojolicious.org/
```

Setting it before the command does it for only that one invocation:

```
% MOJO_CLIENT_DEBUG=1 ./show_transaction http://www.mojolicious.org/
```

Either way you'll see the initial request, the redirection response, and the new request with its response. You wouldn't need to output the transactions yourself.

I've written a tiny tool called 3xx to show the entire redirection chain. It has a Mojo example, but also a Ruby and Python example.

Add to the Request

You can affect your request in almost any way that you like. You'll see more of this later but here's a simple one. Instead of naming your client "Mojolicious (Perl)" as you see in the `User-Agent` field in the previous section, change it to something that's not quite as suspicious.

Modify the previous example to include a new line:

```
my $ua = Mojo::UserAgent->new->max_redirects(3);
$ua->transactor->name( 'Planet Express' );
```

Now the request is slightly different:

```
% ./show_transaction https://www.mojolicious.org/
Request
--------------------------------------------------
GET / HTTP/1.1
Content-Length: 0
Accept-Encoding: gzip
User-Agent: Planet Express
Host: www.mojolicious.org
```

Sometimes you'll fake being another sort of user-agent because some servers are content to look only at the `User-Agent` header to decide what to do, such as returning user-agent specific content (*e.g.* "Update to a supported browser"). Grab one that you like from www.useragentstring.com.

httpbin

httpbin is a webserver suitable for testing user-agent code. It has many different endpoints to show you different things. Here's your IP address:

```
% curl http://httpbin.org/ip
{
  "origin": "107.152.104.229, 107.152.104.229"
}
```

It shows the headers from your request:

```
% curl http://httpbin.org/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.54.0"
  }
}
```

Or anything in the request:

```
% curl http://httpbin.org/anything?robot=Bender
{
  "args": {
    "robot": "Bender"
  },
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.54.0"
  },
  "json": null,
  "method": "GET",
  "origin": "89.46.102.12, 89.46.102.12",
  "url": "https://httpbin.org/anything?robot=Bender"
}
```

Or some XML (that I've truncated in this example):

```
% curl http://httpbin.org/xml
<?xml version='1.0' encoding='us-ascii'?>

...
```

Run httpbin in docker

httpbin is handy for isolated checks about what you are doing, but you don't want to overwhelm a free service. If you really want to bang on it, you can run your **httpbin** server locally by using it through **docker** (Getting Started with Docker):

```
% docker run -p 80:80 kennethreitz/httpbin
% curl localhost:80/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "localhost",
    "User-Agent": "curl/7.54.0"
  }
}
```

Add an entry for *httpbin.org* to your */etc/hosts* and you don't have to change the **httpbin** addresses you see in this book:

```
# /etc/hosts
127.0.0.1    httpbin.org
```

Summary

You have the basic ideas behind Mojo, HTTP, and fluent programming. The rest of the book builds on those concepts, but also assumes you'll read more about the HTTP specification on your own.

By now, you should have installed Mojo and played around with some simple examples to see that it works. Before you dive into more Mojo features, I'm going to use the next couple of chapters to catch up on some Perl features you'll see throughout the book.

Chapter 4

The Document Object Model

I've never seen anything so mind-blowing. Ooh, a reception table with muffins!

The Document Object Model, or simply DOM, represents structured data such as HTML or XML so that you can easily inspect and modify it. The `Mojo::DOM` module handles both of those formats. This chapter only introduces some of the common things that you can do with it, but you'll see more in chapter 5.

Walking Through HTML or XML

What are you going to do with that HTML once the server sends it back to you? Like every other programmer who has touched a keyboard you've probably pulled out a regex to get out the parts that want. There's a better way to do that.

Mojo represents the HTML in a Document Object Model (DOM). This makes way for some powerful tools. Parsing some text with `Mojo::DOM` defaults to parsing HTML:

```
use Mojo::DOM;

my $html = <<~'HTML';
    <ul>
    <li><a href="http://www.example.com/bender">Bender</a></li>
    <li><a href="http://www.example.com/fry">Fry</a></li>
    <li><a href="http://www.example.com/hermes">Hermes</a></li>
    </ul>
    HTML

my $dom = Mojo::DOM->new( $html );

my $links = $dom->find( 'a' );  # a collection

say $links->join( "\n" );
```

The `find` takes a CSS selector (coming up in the next chapter, chapter 5) and walks the DOM looking for matching pieces. It returns a collection of what it finds. The example extracts all of the anchor tags (complete with opening tag, contents, and closing tag):

```
<a href="http://www.example.com/bender">Bender</a>
<a href="http://www.example.com/fry">Fry</a>
<a href="http://www.example.com/hermes">Hermes</a>
```

It gets better. The elements in that collection are still `Mojo::DOM::HTML` objects. Use the `attr` method to get the attribute values. Use that as the first argument to `map` and give it `href` as its argument:

```
my $links = $dom->find( 'a' )->map( attr => 'href' );
say $links->join( "\n" );
```

Now you get the URL in every anchor:

```
http://www.example.com/bender
http://www.example.com/fry
http://www.example.com/hermes
```

Use `text` to get the stuff inside the tags:

```
my $links = $dom->find( 'a' )->map( 'text' );
say $links->join( "\n" );
```

This time the output is stuff between the opening and closing tags:

```
Bender
Fry
Hermes
```

This only includes text not inside of nested HTML tags. Try this by parsing this paragraph with some styled text in it. The `new` can work with snippets of HTML:

```
say Mojo::DOM->new( '<p>Bender <i>Fry</i> Leela</p>' )
    ->find( 'p' )
    ->map( 'text' )
    ->join( "\n" );
```

The output finds the text outside the `<i>` tag. There are two spaces between the words because there are two spaces around the interior tag:

```
Bender  Leela
```

Use `all_text` instead to strip out interior HTML:

```
say Mojo::DOM->new( '<p>Bender <i>Fry</i> Leela</p>' )
    ->find( 'p' )
    ->map( 'all_text' )
    ->join( "\n" );
```

Now the output has the text from the interior tags too:

```
Bender Fry Leela
```

In this example you know that you have exactly one `P` tag so you could have also written it without the `map`:

```
say Mojo::DOM->new( '<p>Bender <i>Fry</i> Leela</p>' )
    ->at( 'p' )
    ->all_text;
```

You can extract parts of the DOM using much more sophisticated situations with fancier selectors. Those get their own chapter in chapter 5.

Parsing XML

You might be tempted to use `XML::Simple` to turn an XML string into a Perl data structure; many people are. Do not be seduced to the dark side! That module is only suitable for the simplest problems or until you can arrange to use something better—perhaps `XML::Hash::XS` or `XML::Fast`. Or, for a really nasty problem, `XML::Twig`.

However, if you only need to grab a few values out of the XML data you can extract them from the DOM directly. `Mojo::DOM` switches to its XML mode if it sees the XML declaration at the beginning of the string. Almost nothing else in your program needs to change:

```
use Mojo::DOM;

say Mojo::DOM->new(
        '<?xml version="1.0"?><p>Bender <i>Fry</i> Leela</p>'
        )
    ->find( 'p' )
    ->map( 'all_text' )
    ->join( "\n" );
```

How does `Mojo::DOM` know that it's XML? There is the DOCTYPE declaration there! But, XML and XHTML (and clean HTML) play by the same rules so the situation is not that different.

Modifying the DOM

There are many methods to modify the DOM but I'm going to ignore those other than this tiny example that builds a new HTML document from an empty string:

```
use Mojo::DOM;
my $dom = Mojo::DOM->new( '' );
$dom->append_content( '<html>' );
$dom
    ->at('html')
    ->append_content( '<h1>Planet Express</h1>' );
say $dom->to_string;
```

Each modification returns a new DOM object but you can assign that back to the same variable:

```
<html><h1>Planet Express</h1></html>
```

That `append_content` takes a DOM too, and will append all of that. The `new_tag` uses the tag name and content I supply:

```
use Mojo::DOM;

my $dom = Mojo::DOM->new( '' );
$dom->append_content( '<html>' );
$dom
    ->at('html')
    ->append_content(
        $dom->new_tag( 'h1' => 'Planet Express' )
        );

say $dom->to_string;
```

Creating a PropertyList

I've used `Mojo::DOM` to construct macOS PropertyList files (although I wrote the `Mac::PropertyList` module). I start with the header, which includes a couple of declarations:

```
use v5.26;
use Mojo::DOM;

my $dom = Mojo::DOM->new( <<~"HERE" );
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
      "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    HERE

say $dom;
```

That's easy enough. The next part creates the root element, `plist`, and a `string` element that it will contain. Once I have both, I can select the `plist` element from that sub-DOM and append content to it:

```
use v5.26;
use Mojo::DOM;

my $dom = Mojo::DOM->new( <<~"HERE" );
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
      "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    HERE

my $plist  = $dom->new_tag( 'plist', version => '1.0' );
my $string = $dom->new_tag( 'string', 'https://www.example.com' );
$plist->at( 'plist' )->append_content( $string );

$dom->append_content( $plist );

say $dom;
```

I end up with the PropertyList I need:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.
    apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0"><string>https://www.example.com</string></plist>
```

Summary

This is just the start of your DOM processing. The mechanism is easy, and in the next chapter you'll see the myriad ways to find exactly the parts of HTML you want to extract.

Chapter 11

Command-Line Programs

Oh wait, you're serious. Let me laugh even harder.

Several of the methods from `Mojo::UserAgent` and other utilities have shortcuts to make them easy for you to use directly from the command line. These one-liners are handy for short programs you probably won't need again or even those that you'd want to make into shell aliases.

The mojo Command

The *mojo* command is mostly for creating, running, and testing Mojo web applications—stuff that I don't cover in this book. It does have one interesting command: `get`, which makes a web request. It's there to make requests against your own application but it's useful for any request to any server.

Without any arguments, you get a help message that lists several things that you can do. Most of these aren't interesting for this chapter:

```
% mojo
Usage: APPLICATION COMMAND [OPTIONS]

mojo version
mojo generate lite_app
./myapp.pl daemon -m production -l http://*:8080
./myapp.pl get /foo
./myapp.pl routes -v
...
```

Try the `get` command without further arguments; you get another help message that's specific to that command.

```
% mojo get
Usage: APPLICATION get [OPTIONS] URL⏎
  [SELECTOR|JSON-POINTER] [COMMANDS]

./myapp.pl get /
...
```

Fetch a resource by supplying the URL as the argument to get. IPify returns your IP address as text:

```
% mojo get api.ipify.org
89.187.178.38
```

Dump it to the screen or redirect it to a file:

```
% mojo get https://mojolicious.org
% mojo get https://mojolicious.org > mojo.html
```

Use `-v` to see the request and response headers. If you request just `mojolicious.org`, it looks like nothing happens. The headers tell a different story. You get a response but it's an HTTP redirect. The *mojo* command didn't automatically handle that for you:

```
% mojo get -v www.mojolicious.org
GET / HTTP/1.1
```

```
User-Agent: Mojolicious (Perl)
Accept-Encoding: gzip
Host: www.mojolicious.org
Content-Length: 0

HTTP/1.1 301 Moved Permanently
Location: https://www.mojolicious.org/
Date: Mon, 23 Jul 2018 18:08:54 GMT
Transfer-Encoding: chunked
CF-RAY: 43f01855c650923c-EWR
Connection: keep-alive
Server: cloudflare
Cache-Control: max-age=3600
Expires: Mon, 23 Jul 2018 19:08:54 GMT
```

Follow redirects (up to 10 of them) with the `-r` switch. Now you see some content:

```
% mojo get -r www.mojolicious.org
<!doctype html><html>
  <head>
    <link rel="search" type="..."
      href="/opensearch.xml" title="Mojolicious" />
    <title>
      Mojolicious - Perl real-time web framework
    </title>
...
```

Add a header with `-H`. Separate the header name and value with a colon:

```
% mojo get -v -H 'X-Bender: Bite me!' www.mojolicious.org
GET / HTTP/1.1
Accept-Encoding: gzip
Host: www.mojolicious.org
Content-Length: 0
User-Agent: Mojolicious (Perl)
X-Bender: Bite me!

...
```

Add some form data with `-f`. Specify each name and value pair with a separate `-f`:

```
% mojo get -v -f 'a=b' -f 'a=c' www.httpbin.org/get
GET /get?a=b&a=c HTTP/1.1
Host: www.httpbin.org
User-Agent: Mojolicious (Perl)
Accept-Encoding: gzip
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: application/json
Content-Encoding: gzip
Server: nginx
Date: Sat, 13 Apr 2019 01:26:51 GMT
Connection: keep-alive
Content-Length: 202
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

{
  "args": {
    "a": [
      "b",
      "c"
    ]
  },
  "headers": {
    "Accept-Encoding": "gzip",
    "Host": "www.httpbin.org",
    "User-Agent": "Mojolicious (Perl)"
  },
  "origin": "89.46.103.172, 89.46.103.172",
  "url": "https://www.httpbin.org/get?a=b&a=c"
}
```

Try it with something that needs URL encoding, such as a space in the value for `a`:

```
% mojo get -v -f 'a=b' -f 'a=c d' www.httpbin.org/get
```

```
GET /get?a=b&a=c+d HTTP/1.1
Host: www.httpbin.org
Accept-Encoding: gzip
User-Agent: Mojolicious (Perl)
Content-Length: 0

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json
Content-Length: 205
Connection: keep-alive
Date: Sat, 13 Apr 2019 01:28:34 GMT
Access-Control-Allow-Credentials: true
Server: nginx
Content-Encoding: gzip

{
  "args": {
    "a": [
      "b",
      "c d"
    ]
  },
  "headers": {
    "Accept-Encoding": "gzip",
    "Host": "www.httpbin.org",
    "User-Agent": "Mojolicious (Perl)"
  },
  "origin": "89.46.103.172, 89.46.103.172",
  "url": "https://www.httpbin.org/get?a=b&a=c+d"
}
```

Although the command name is `get` you can change the HTTP verb with `-M`. Now the form data are in the message body so you don't see those in the request headers:

```
% $ mojo get -v -M POST -f 'a=b' -f 'a=c d'⏎
    www.httpbin.org/post
POST /post HTTP/1.1
```

```
User-Agent: Mojolicious (Perl)
Content-Length: 9
Host: www.httpbin.org
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip

...
```

Processing the response

Immediately process the result of a request by providing a *selector*. For JSON responses, the selector starts with `/` and follows the rules from `Mojo::JSON::Pointer` (RFC 6901. XPath users should be comfortable with this:

```
% mojo get -f 'a=b' -f 'a=c d' www.httpbin.org/get /url
https://www.httpbin.org/get?a=b&a=c+d
% mojo get -f 'a=b' -f 'a=c d' www.httpbin.org/get /args
{"a":["b","c d"]}
```

Earlier, in chapter 3, you saw jq doing the same sort of thing. Use it to extract part of the response:

```
% mojo get -f 'a=b' -f 'a=c d' www.httpbin.org/get /args⏎
    | jq '.a[1]'
"c d"
```

If the response is HTML, use CSS selectors instead. This extracts the `title` part of the DOM and includes the HTML:

```
% mojo get -r www.mojolicious.org title
<title>
      Mojolicious - Perl real-time web framework
    </title>
```

After the selector, supply a command and its arguments. Select only the text in the `title` tag:

```
% mojo get -r www.mojolicious.org title text
      Mojolicious - Perl real-time web framework
```

In this example, `attr` selects values from attributes of the selected items; the `href` is the argument to `attr`:

```
% mojo5.28.0 get -r www.mojolicious.org 'a[href]' attr href
https://metacpan.org/release/Mojolicious
https://mojolicious.org
https://mojolicious.org/perldoc
...
```

Here's another example in which you extract the text inside the H1, H2, and H3 tags:

```
% mojo get -r www.mojolicious.org 'h1, h2, h3' text
  A next generation web framework for the Perl programming language.

Features
Installation
Getting Started
Duct tape for the HTML5 web
Want to know more?
```

Perl One-Liners

A "one-liner" is a program that you write and execute on the command line. It's typically for small bits of code that you may not intend to reuse. Specify that program as the argument to the `-e` switch:

```
% perl -e 'print "Hello World\n"'
```

On Windows, you need to use double quotes for the argument, which is a problem inside your program. Use the `qq` generalized quoting instead of double quotes in your program:

```
C:\perl -e "print qq(Hello World\n)"
```

In those two examples you need the double quotes inside the program to interpolate the `\n` into a newline. You can get rid of the double quote problem by using the `-l` switch; that automatically adds a newline to a `print`. You don't need the interior double quotes (or equivalent) now. The Unix shell and Windows command lines can look the same:

```
% perl -l -e "print 'Hello World'"
C:\perl -l -e "print 'Hello World'"
```

Since the `-l` takes no argument, you can "bundle" the switches for a little less typing:

```
% perl -le "print 'Hello World'"
C:\perl -le "print 'Hello World'"
```

The `-E` is like `-e` but also enables the new features added since v5.8. One of those features is `say`—it adds a newline to the end of its output. This is effectively the same as the `-l` and `-e` together:

```
% perl -E "say 'Hello World'"
C:\perl -E "say 'Hello World'"
```

Use `-M` to load a module from the command line. The `ojo` module is cleverly named to spell out "Mojo" when you load it with `-M`:

```
% perl -Mojo -E "say Mojolicious->VERSION"
```

The `-I` adds directories to the module search path. This example will find the *./lib/yModule.pm* file:

```
% perl -Ilib -MyModule "..."
```

Remember that v5.26 removed the dot from the default `@INC`. Reädd that with `-I.` if you need it:

```
% perl -I. -MyModule "..."
```

The PERL5OPT environment variable can make these one-liners a bit shorter and are often useful in a session (but maybe not across sessions). These command-line options are automatically added to your call to *perl*. Along with that, the PERL5LIB environment variable adds directories to @INC. These reduce the amount of repeated typing a bit:

```
% export PERL5OPT='-Mojo'
% export PERL5LIB='lib'
% perl -E "say Mojolicious->VERSION"
```

In your shell setting file you should be able to define aliases for one-liners. Here's the *bash* alias for a Perl one-liner that converts a decimal number to its hexadecimal representation:

```
# .bash_profile
alias d2h="perl -e 'printf qq|%X\n|, int( shift )'"
```

You can do a similar thing with DOS or PowerShell but it's a bit more complicated than I want to show here; I'll punt to a [Aliases in Windows command prompt](https://stackoverflow.com/q/20530996/2766176) on StackOverflow.

Fetching a Resource

All of the HTTP verbs have a shortcut function. Instead of returning a transaction object they return the response object; that cuts out one step in getting to the result. Instead of get there is g and you can call body right away:

```
% perl -Mojo -E "say g('mojolicious.org')->body"
```

Notice the lack of a scheme in that example. It's just the host name and the shortcut figures it out. The g takes the same arguments as get. This one includes some form data in the request:

```
% perl -Mojo -E "say g('http://httpbin.org/get'⏎
    => form => { robot => 'Bender'})->body"
```

Save the content so you can play with it later:

```
% perl -Mojo -E "g(shift)->save_to('test.html')"⏎
    mojolicious.org
```

The `p` does a POST request:

```
% perl -Mojo -E "say p('http://httpbin.org/post' =>⏎
    form => { robot => 'Bender'})->body"
```

The `h` shortcut stands in for the `head` method. Sometimes you don't care about the content (which can be quite long). This one-liner shows all of the headers:

```
% perl -Mojo -E "say h('mojolicious.org')⏎
    ->headers->to_string"
Server: cloudflare
Content-Encoding: gzip
Connection: keep-alive
Set-Cookie: ...
CF-RAY: 43eeea976c1021c2-EWR
Content-Type: text/html;charset=UTF-8
Expect-CT: max-age=604800, report-uri="..."
Date: Mon, 23 Jul 2018 14:42:55 GMT
```

Or look at a particular header:

```
% perl -Mojo -E 'say h("mojolicious.org")->headers⏎
    ->set_cookie'
__cfduid=d1727b77aa1044975e86fb61badd3f1ad1532357521;⏎
expires=Tue, 23-Jul-19 14:52:01 GMT; path=/; domain=⏎
.mojolicious.org; HttpOnly; Secure
```

Some one-liners

This section includes some one-liners. Some of them can use both the *mojo* command and the shortcuts.

Show the HTTP status code, which I like to check if the right thing is happening:

```
% perl -Mojo -e "say g(shift)->code" www.perl.com
200
```

Extract the unique links in a resource:

```
% mojo get https://www.mojolicious.org a attr href
% perl -Mojo -E 'g("mojolicio.us")->dom("a")⏎
    ->map( attr => "href" )->uniq->join("\n")->say'
```

Extract all the images:

```
% mojo get https://www.mojolicious.org img attr src
% perl -Mojo -E 'g("mojolicio.us")->dom("img")⏎
    ->map( attr => "src" )->uniq->join("\n")->say'
```

Get the titles from the latest articles in /r/perl on Reddit:

```
% mojo https://www.reddit.com/r/perl/⏎
    'p.title > a.title' text
```

List all the HTML headings (not HTTP headers) in a document:

```
% mojo get https://www.mojolicious.org 'h1,h2,h3' text
```

Grab the value of a particular header. Most of the time you only need a `HEAD` request:

```
% perl -Mojo -E "say h(shift)->headers⏎
    ->header(shift) // ''"⏎
    https://www.mojolicious.org Server
cloudflare
% perl -Mojo -E "say h(shift)->headers⏎
    ->header(shift) // ''"⏎
    https://www.perl.com/ ETag
"5b5a1636-8248"
```

Download some JSON and extract something from it. The `j` turns a JSON string into a Perl data structure. This one gets the author record from MetaCPAN by the ID and extracts the full name:

```
% perl -Mojo -e "say⏎
    j(g('https://fastapi.metacpan.org/v1/author/'⏎
    .shift)->body)->{name}" BDFOY
brian d foy
```

And here's one with the IPify response:

```
% perl -Mojo -E "say j(g(shift)->body)->{ip}"⏎
    'api.ipify.org?format=json'
89.187.178.38
```

In this case, it's easier to go for the JSON payload directly:

```
% perl -Mojo -E "say g(shift)->json->{ip}"⏎
    'api.ipify.org?format=json'
```

Playing with the DOM

The `f` shortcut creates a `Mojo::File` object. Call `slurp` on that and you've reïnvented the *cat* (or *type*) command:

```
% perl -Mojo -e "print f('test.md')->slurp"
```

Instead of specifying the filename inside the program, use `shift` to get it from the command-line arguments. Outside of a subroutine definition the `shift` works on `@ARGV`. This give the same result:

```
% perl -Mojo -e "print f(shift)->slurp" test.md
```

The `x` shortcut creates a `Mojo::DOM` object from its argument.

```
% perl -Mojo -E "say x('<p>Bender</p>')⏎
    ->at('p')->text"
Bender
```

Combine that with `f` to load a local file into a DOM

```
% perl -Mojo -E "say x(f(shift)->slurp)⏎
    ->at('p')->text"
```

The `n` runs a block of code the number of times that you specify. Here's one that slurps a file then finds all the DIV tags 10 times. It outputs a report of the benchmarks:

```
% perl -Mojo -E 'my $dom=x(f(shift)->slurp);⏎
    n { $dom->find("div") } 10' spec.html
5.09754 wallclock secs ( 5.06 usr +  0.01 sys =  5.07 CPU)
```

Or grab it from the network, which is a little bit slower:

```
% perl -Mojo -E 'my $dom=g(shift)->dom;⏎
    n { $dom->find("div") } 10' https://html.spec.whatwg.org
5.31821 wallclock secs ( 5.28 usr +  0.01 sys =  5.29 CPU)
```

Extending ojo

Create your own `ojo`-like module to provide other shortcuts you'd like. Here's an extension that adds an `S` to slurp a file and `cookie` to extract the `Set-Cookie` header. It loads `ojo` first and reëxports those functions:

```
use Mojo::Base -strict, -signatures;
use ojo;

use Exporter qw(import);
our @EXPORT = (
    # new shortcuts
    qw( s cookie ),
    # from ojo
    qw( a b c d f g h j n o p r t u x )
    );

sub S ( $file ) { f( $file )->slurp }
sub cookie ( $url ) { g( $url )->headers->set_cookie }
```

Now loading a file is a bit shorter (something important for one-liners):

```
% perl -Ilib -MyOjo -e "print S(shift)" test.html
```

Create a DOM from the slurped file right away:

```
% perl -Ilib -MyOjo -e "x(S(shift))->at('p')->text" test.html
```

Check the cookie value for a resource:

```
% perl -Ilib -MyOjo -E "say cookie(shift)"⏎
    mojolicious.org
__cfduid=dc889efdd23c2fca4379087e969ae66901532359564;⏎
expires=Tue, 23-Jul-19 15:26:04 GMT; path=/; domain=⏎
.mojolicious.org; HttpOnly; Secure
```

You might also consider not inheriting from `ojo`. Take a look at the source; it's very simple and you can easily create your standalone module doing the same sort of thing with your specialized subroutines.

Summary

There are several shortcuts that make Mojo one-liners easy. Make your request and process it right from the command line. If you want to reüse these one-liners you can create shell aliases. If you need more you can create your own shortcut module to handle it for you.