MOJOLICIOUS: REVOLUTIONS

Joe McMahon
pemungkah@me.com

EASY TO GET STARTED

```
$ curl get.mojolicio.us | sh
```

- Perl 5.10 or better
  - There are no other prerequisites
  - `cpan Mojolicious` if you're allergic to executing random scripts
- Ultra-simple web development platform

HOW SIMPLE IS IT?

```
use Mojolicious::Lite;

get '/hi' => {text => 'Hello World!'};
get '/bye' => {text => 'Goodbye World!'};

app->start;
```

NOW RUN IT

```
$ morbo hello.pl
Server available at http://127.0.0.1:3000.

$ curl http://127.0.0.1:3000/hi
Hello World!
```

STUPID EASY

- Translate your URLs into routes
- Write subs for each route
- Run it
- Great for one-offs

MORBO AND HYPNOTOAD

- Morbo
  - handy low-traffic web server
    - non-blocking I/O
    - HTTP and Websocket support
- Hypnotoad
  - production-ready server, adds:
    - preforking
    - hot deployment

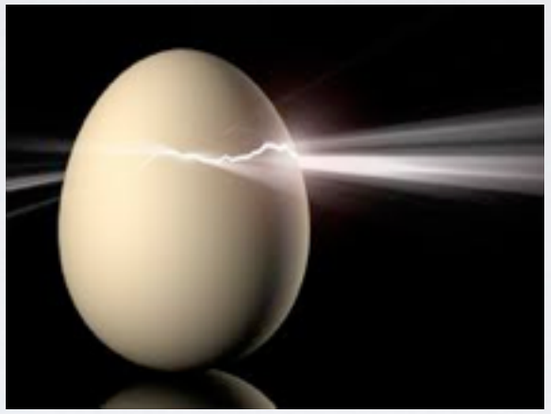
CASE STUDY: HTTP CLIENT VALIDATION

- Custom HTTP client/server pair
- Client makes requests, server sends responses

CHICKEN AND EGG

- Not sure if the client works right
- Not sure if the server works right
- How do we get started testing this?

BREAKING THE CIRCLE



- Arbitrarily pick side A to be source of truth
- Side B will trust it
- Once side B is tested and known-good, we can test the real side A with it

WHAT DO WE NEED?

- A webserver, obviously
- Accept requests and respond to them in a predictable way
- Throw errors that the client should be able to handle: 400's, 500's, timeouts, bad output...
- Actually succeed on a request too

BASIC CLIENT/SERVER PATTERN

- The client: sends requests, receives responses, may take some action
- The request: a specific URL (GET, DELETE), or URL plus data (POST, PUT, file upload)
- The server: receives a request, performs some arbitrary processing, sends a response
- The response: a specific set of headers and data in a known, well-defined format

WHAT DO WE KNOW?

- The request: some combination of headers, URL, and possibly data
- The response: headers plus a known set of text
- A (possibly broken) client implementation ready to test

SERVER CAN BE DUMB

- All we care is that the response is right
- It can be a fixed string, or a string with a few interpolations
- We don't have to implement the entire server, just send back the right text for a limited set of interactions
- The dumber it is, the less likely we are to have bugs

BUT NOT TOO DUMB

- Traditional web servers only know how to speak HTTP, which is not smart enough
- Making them respond arbitrarily to input, even trivially, requires a significant amount of code
- Mojolicious's routes and built-in server let us get around this

OUR MADE-UP SERVICE

- Custom header required in responses: "X-My: YES"
- Returned data should be in the expected format (XML)
- Client should handle
  - timeouts
  - failed and error responses
- Should be able to stop the server (makes tests easier)
- All in one test file would be great

OUR PRIMITIVE CLIENT

```
package Client;
use strict;
use warnings;

use Exporter;
@Client::EXPORT = qw(client server_url);

sub server_url { 'http://127.0.0.1:3000' }
sub client {
  my $path = shift;

  my $mech = WWW::Mechanize->new(timeout=>3);  # autocheck on!
  $mech->add_header('Accept' => 'application/json');
  $mech->get("$server_url/my/api/get/$path");

  my $x = $mech->response->header('X-My') || 'NO';
  die "The response from the server did not include an 'X-My' header\n"
    unless $x eq 'YES';

  die "Response was not JSON\n"
    unless $mech->response->header('Content-Type') eq 'application/json';

  return $mech->content;
}
```

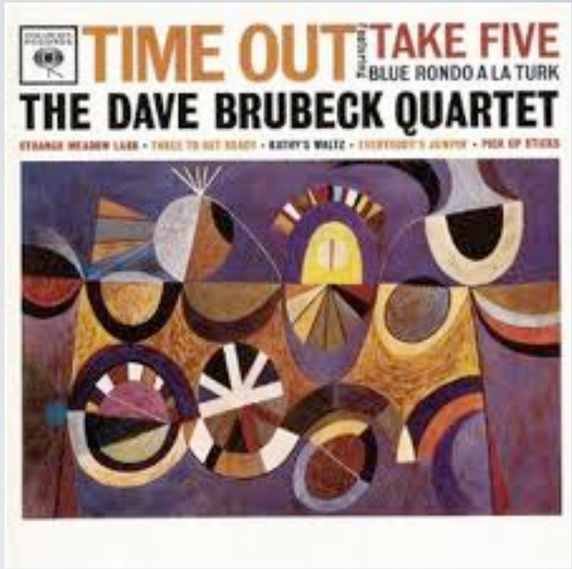
BASIC TEST SCAFFOLDING

```
use strict;
use warnings;

use Mojolicious::Lite;
use WWW::Mechanize;
use Test::More;
use Test::Exception;

use Client;
```

TEST CASE 1: “NOT ME”

```
# Case 1: Pretend that the server isn’t a proper
# target for this client by not returning the
# expected header, but returning valid HTML.

get ‘/my/api/get/timeOfDay’ => sub {
  my $self = shift;
  $self->render(text =>
‘This isn’t the server you’re expecting, move along’);
}
```

TEST CASE 2: TIMEOUT




```
# Case 2: server times out
get '/my/api/get/currentUsers' => sub {
  my $self = shift;

  # This is longer than the client's timeout threshold;
  # client will need to wait an additional 3 seconds (plus 1 for
  # good measure)
  # before sending another request.
  sleep 6;

  # Now return a valid response so that the server continues to
  # run properly.
  # The client will ignore this since it will be sent after the
  # client timeout expires.
  $self->tx->res->headers->header('X-My', 'YES');
  $self->render(text => 'Sorry, I was busy');
}
```

TEST CASE 3: INTERNAL SERVER ERROR

```
# Case 3: internal server error without processing
# request.
get '/my/api/get/nextBackupTime' => sub {
  my $self = shift;
  $self->tx->res->headers->header('X-My', 'YES');
  $self->tx->res->code(500);
  $self->render(text => 'Internal failure, sorry');
}
```

TEST CASE 4: BAD REQUEST

```
# Case 4: server processes request but application
# logic detects a problem.
get '/my/api/get/nextBackupDate' => sub {
  my $self = shift;
  $self->tx->res->headers->header('X-My', 'YES');
  $self->tx->res->code(400);
  $self->render(text => 'No backups scheduled');
}
```

TEST CASE 5: BAD RESPONSE DATA

```
# Case 5: server processes the request, but returns
# a (non-XML) response.
get '/my/api/get/databaseConsistent' => sub {
  my $self = shift;
  $self->tx->res->headers->header('X-My', 'YES');
  $self->render(text => 'database OK');
}
```

TEST 6: IT WORKS!

```
# Case 6: server processes request and returns
# a good response.
get '/my/api/lastUser/:username' => sub {
  my $self = shift;
  my $username = $self->stash('username');
  $self->tx->res->headers->header('X-My', 'YES');
  $self->render( json => { user => $username } );
};
```

CASE 7: SHUT IT DOWN


```
# Case 7: Shut the server down.
get '/my/api/get/shutdown' => sub {
  exit 0;
}
```

FORKING THE SERVER

```
my $pid;
if ($pid = fork) {
  diag “Waiting for the server to start”;
  sleep 5;
}
else {
  local @ARGV = qw(daemon);
  app->log->level(‘error’);
  app->start;
}
```

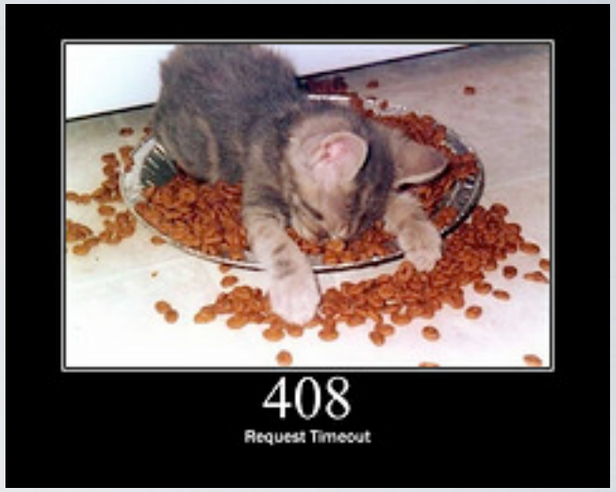
WRITING THE TESTS

- Now we can use standard testing modules to verify the client
- `Test::More` for response testing
- `Test::Exception` for successes and failures

CASE 1: NOT THE RIGHT SERVER

```
# Case 1: wrong server
dies_ok
  { $result = client('timeOfDay') }
  'Wrong server detected';

like $@,
  qr/The response from the server did not include an 'X-My' header/,
  'correct diagnosis of wrong server';
```

CASE 2: REQUEST TIMEOUT




```
# Case 2: timeout
dies_ok
  { $result = client('currentUsers') }
  'timeout detected';

like $@,
  qr[Error GETing .*?currentUsers: read timeout],
    'correct diagnosis of timeout';
print STDERR "# Resyncing";
my $wait = 5;
while ($wait) { sleep 1;  print STDERR ".";  $wait-- }# resync

print STDERR "done\n";
like $@,
  qr[Error GETing .*?currentUsers: read timeout],
  'correct diagnosis of timeout';
```

CASE 3: SERVER ERROR


```
# Case 3: internal error
dies_ok
  { $result = client('nextBackupTime') }
  'internal server error caught';

like $@,
  qr[Error GETing .*?nextBackupTime: Internal Server Error],
  'correct diagnosis of internal server error';
```

CASE 4: REQUEST ERROR


```
# Case 4: app error
dies_ok
  { $result = client('nextBackupDate') }
  'caught application error';

like $@,
  qr[Error GETing .*?nextBackupDate: Bad Request],
  'correct diagnosis of application failure';
```

CASE 5: BAD RESPONSE FORMAT


```
# Case 5: bad response format
dies_ok
  { $result = client('databaseConsistent') }
  'caught application error';

like $@,
  qr/Response was not XML/,
  'correct diagnosis of bad response format';
```

FINALLY, CASE 6: IT WORKS

```
# Case 6: OK
lives_ok
  { $result = client('lastUser/foo')
}
  'call worked';

is $result, qq<{"user":"foo"}>,
  'got expected data in JSON';

ok ! $@, 'success';
```

TWIN TEST: SHUTDOWN AND "NO DATA RETURNED"

```
# Case 7: shut down the server
dies_ok
  { $result = client('shutdown') }
  'shutdown occurred';
like $@,
  qr[Error GETing .*?shutdown: Server closed connection without
sending any data back ],
  'detected closed connection';

# Clean up server process and we're done
waitpid($pid,0);
exit 0;
```

DEMO

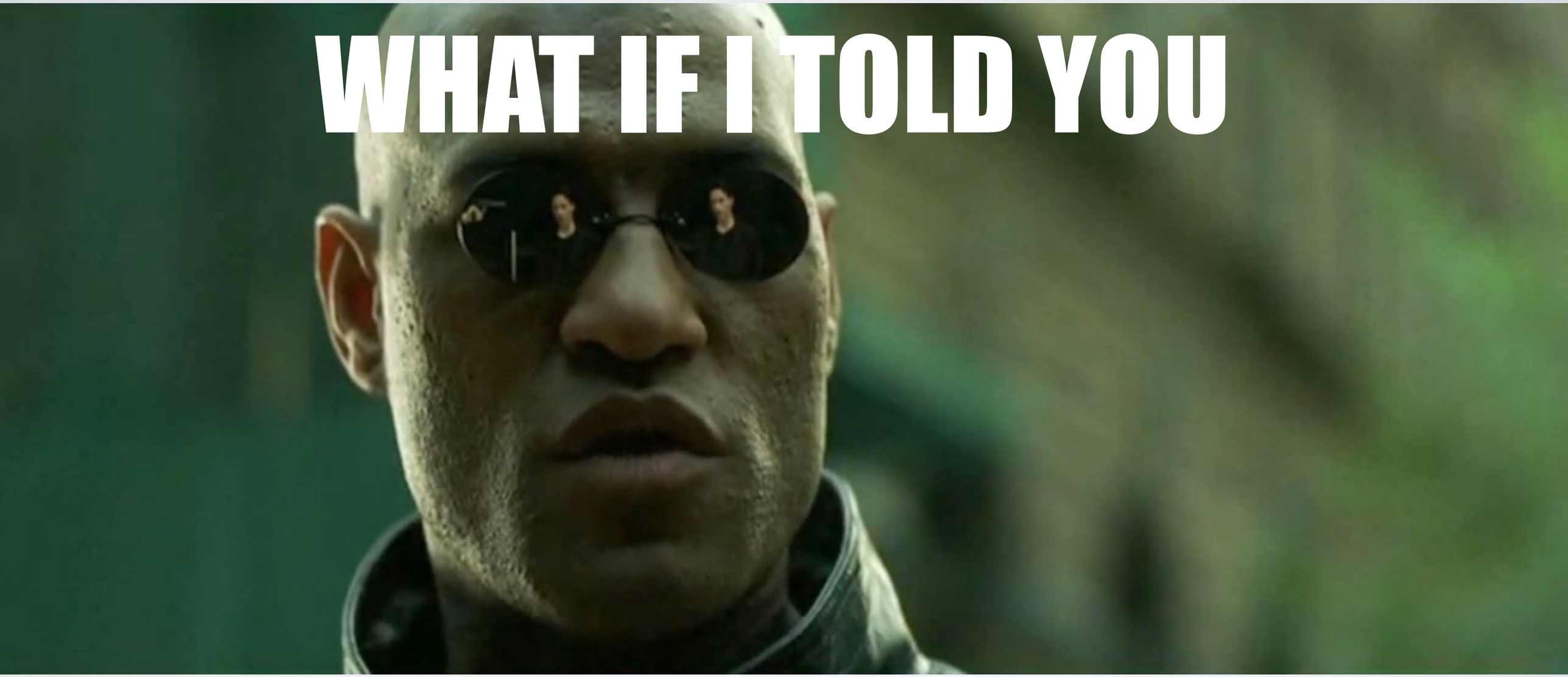
MOJOLICIOUS SERVERS

- Simple, easy to set up
- Route map to actions
- Templating for responses
- Great for building quick web tools

SO MOJOLICIOUS IS A SERVER DEVELOPMENT TOOL?

NOT SO FAST, SUNSHINE

NOT SO FAST, SUNSHINE



NOT SO FAST, SUNSHINE

WEB FETCH REFRESHER

```
require LWP::UserAgent;

my $ua = LWP::UserAgent->new;
$ua->timeout(10);
$ua->env_proxy;

my $response = $ua->get('http://search.cpan.org/');

if ($response->is_success) {
    print $response->decoded_content;  # or whatever
}
else {
    die $response->status_line;
}
```

MOJOLICIOUS VERSION

```
use 5.010;
use Mojo::UserAgent;

my $ua = Mojo::UserAgent->new;

say $ua->get('http://search.cpan.org/')->res->body;
```

LET'S FETCH A PAGE'S TITLE

GET THE CONTENT

```
use 5.010;
use LWP::UserAgent;
use HTML::Parser;

my $ua = LWP::UserAgent->new;
$ua->timeout(10);
$ua->env_proxy;

my $content = $ua->get(‘http://search.cpan.org/')->decoded_content;
```

PREPARE TO PARSE

```
my($title_text, $capture_now);

my $parser = HTML::Parser->new(
    start_h => [\&open_tag, "tagname"],
    text_h  => [\&conditional_capture, "dtext"],
);

sub open_tag {
    my($tagname) = @_;
    $capture_now ++ if $tagname eq "title";
}

sub conditional_capture {
    if ($capture_now) {
        $title_text = $_[0];
        undef $capture_now;
    }
}
```

EXTRACT AND PRINT

```
# Actually extract the <title> tag’s text.
$parser->parse($content);

say $title_text;
```

MOJOLICIOUS VERSION

```
use 5.010;
use Mojo::UserAgent;

my $ua = Mojo::UserAgent->new;

say $ua->get(‘http://search.cpan.org/')->res
    ->dom->html->head->title->text;
```

BUILT-IN DOM!

COMPARISON

- LWP version
  - 31 lines, 20 minutes, 3 debugger sessions
- Mojolicious version
  - less than a minute, 6 lines

AS A ONE-LINER

```
perl -Mojo -E 'say g("http://search.cpan.org")->dom->html->head->title->text'
```

OR...

```
bash-3.2$ mojo get http://search.cpan.org 'head > title'
<title>The CPAN Search Site - search.cpan.org</title>
```

HEAD > TITLE? LOOKS LIKE...JQUERY!

GETTING FANCIER

- Let's extract all the titles from the blogs.perl.org feed
- We'll need to access the feed, then loop over the items
- Not even going to bother writing the LWP version

ATOM FEED ACCESS, 12 LINES

```
use 5.010;

use Mojo::UserAgent;
my $ua = Mojo::UserAgent->new;

$ua->get('http://blogs.perl.org/atom.xml')->res
    ->dom( 'entry > title' )
        ->each( sub {
                        state $n = 0;
                        say ++$n, ": ", $_->text
                    }
            );
```

```
bash-3.2$ perl mojo-feedread.pl
1: blogs.perl.org users being phished
2: Keep Calm, Roll on.
3: My Favourite Test::* Modules
4: Finding my computer
5: Stupid Lucene Tricks: Storing Non-Documents
6: Rule Role's or Role's Rule?
7: Const::Exporter Released
8: New module Path::Iterator::Rule::RT
9: A Class Without Class
10: Moose Dying With "Invalid version format (version required)"
11: Significant newlines? Or semicolons?
12: GitPrep 1.5 is released - Improve HTTP repository access, and
display README in subdirectory
13: Seeking code to find a free TCP/IP port
14: A Race to the Finish
15: Bugs in the compiler
16: No Grot is Good Grot
17: Announcing Test::mongod
18: Easier Database Fixtures
19: Lessons to learn from Apple
20: Towards Type::Tiny 1.000000
21: DC-Baltimore Perl Workshop 2014 - Call For Speakers
22: Frustrated Moose
…
```

PHP CEO
@PHP_CEO

Following

IT HAS COME TO MY ATTENTION THAT SOMEONE CALLED JASON HAS BEEN ENCODING AND DECODING DATA IN OUR APP. PLEASE CHANGE YOUR PASSWORDS

DIRECT JSON QUERY AND PARSE

```
use 5.010;
use Mojo::UserAgent;
use Mojo::URL;

my $ua = Mojo::UserAgent->new;

my $url = Mojo::URL->new('http://api.metacpan.org/v0/release/_search');
$url->query({q => 'MCMAHON', sort => 'date:desc'});
for my $hit ( @{ $ua->get($url)->res->json->{hits}{hits} } ) {
  say “$hit->{_source}{name}\t($hit->{_source}{date})";
}
```

CPAN PUSH HISTORY, TEN LINES

```
bash-3.2$ perl json-query.mojo
Test-XML-Simple-1.04     2013-01-26T21:48:33
Test-XML-Simple-1.03     2013-01-26T21:40:34
Date-PeriodParser-0.17   2013-01-26T09:46:51
GraphViz-Data-Structure-0.19    2013-01-26T03:27:36
Test-WWW-Simple-0.34     2013-01-26T03:22:18
Test-XML-Simple-1.02     2013-01-26T02:15:34
URI-ImpliedBase-0.08     2012-07-08T18:29:34
Test-XML-Simple-1.01     2012-06-05T20:32:11
Date-PeriodParser-0.14   2011-07-13T04:49:13
Test-Tail-Multi-0.06     2011-07-13T04:42:35
bash-3.2$
```

SUPPORTS FULL REST

- Supports all HTTP verbs
  - GET, PUT, POST, DELETE, PATCH
- Couldn’t get a useful REST server running locally to demo this...sorry

OTHER PARTS

- PSGI (Plack) support
- cpanify - upload files to CPAN
- generate {app | lite-app | makefile | plugin}

DRAWBACKS

- There always are some
- It's all or nothing

QUESTIONS?

THANKS