Web Scraping with PHP

scrapingbee.com/blog/web-scraping-php

Jérôme Gamez | 22 September 2020 | 14 min read

You might have seen one of our other tutorials on how to scrape websites, for example with Ruby, JavaScript or Python, and wondered: what about the most widely used server-side programming language for websites, which, at the same time, is the most dreaded? Wonder no more - today it's time for **PHP** 🥳!

Believe it or not, but PHP and web scraping have much in common: just like PHP, Web Scraping can be used in a quick and dirty way, more elaborately, and enhanced with the help of additional tools and services.

In this article, we'll look at some ways to scrape the web with PHP. Please keep in mind that there is no general "the best way" - each approach has its use-case depending on what you need, how you like to do things, and what you want to achieve.

As an example, we will try to get a list of people that share the same birthday, as you can see, for instance, on famousbirthdays.com. If you want to code along, please ensure that you have installed a current version of PHP and Composer.

Create a new directory and in it, run:

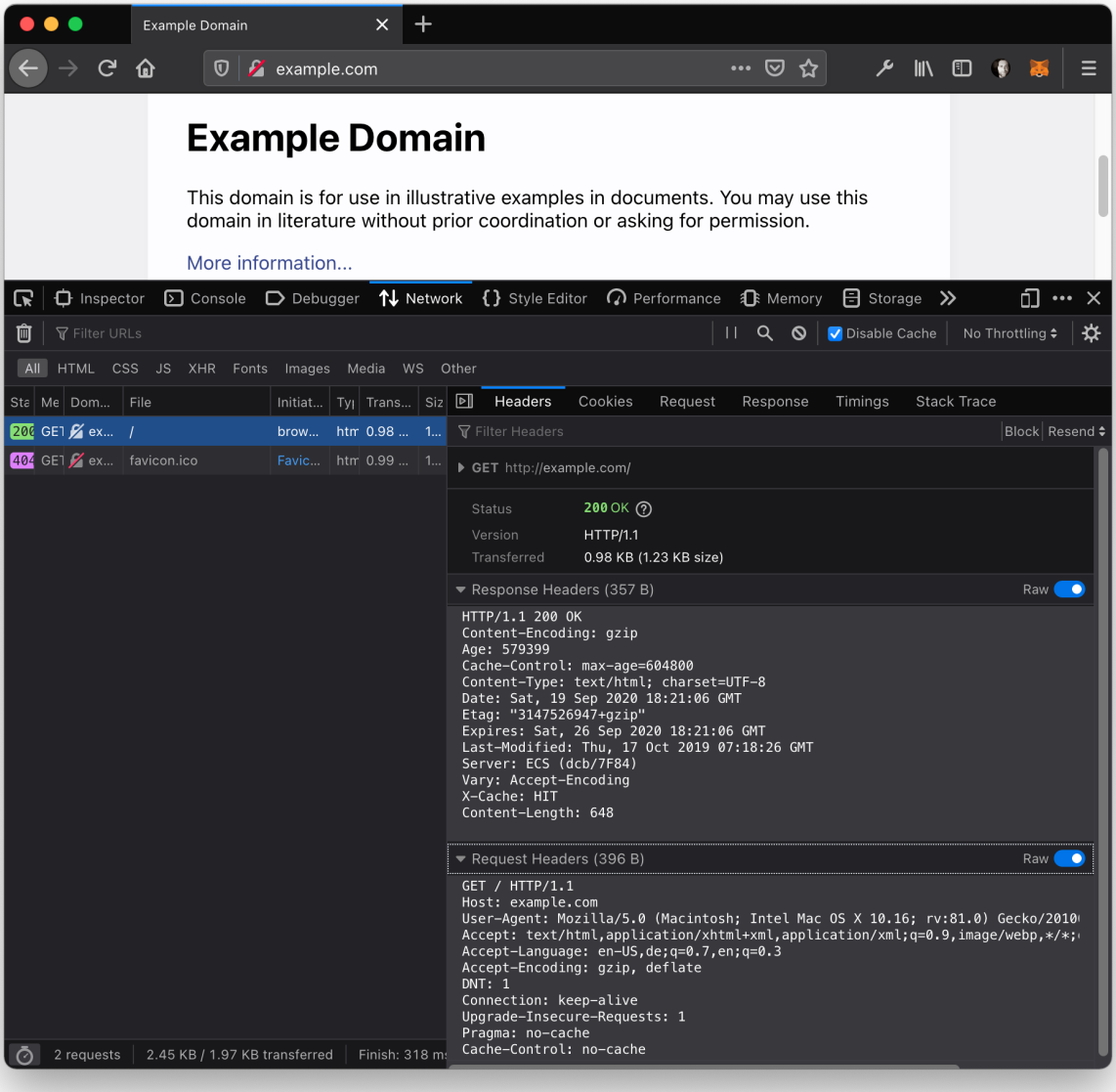
```
$ composer init --require="php >=7.4" --no-interaction
$ composer update
```

We're ready!

1. HTTP Requests

When it comes to browsing the web, the most commonly used communication protocol is *HTTP*, the Hypertext Transport Protocol. It specifies how participants on the World Wide Web can communicate with each other. There are *servers* hosting resources and *clients* requesting resources from them.

Your browser is such a client - when we enable the developer console, select the "Network" tab and open the famous example.com, we can see the full request sent to the server, as well as the full response:



Network tab of your browser developer console

That's quite some request- and response headers, but in its most basic form, a request looks like this:

```
GET / HTTP/1.1
Host: www.example.com
```

Let's try to recreate what the browser just did for us!

`fsockopen()`

We usually don't see this lower-deck communication, but just for the sake of it, let's create this request with the most basic tool PHP has to offer: `fsockopen()`:

```
<?php
# fsockopen.php

// In HTTP, lines have to be terminated with "\r\n" because of
// backward compatibility reasons
$request = "GET / HTTP/1.1\r\n";
$request .= "Host: www.example.com\r\n";
$request .= "\r\n"; // We need to add a last new line after the last header

// We open a connection to www.example.com on the port 80
$connection = fsockopen('www.example.com', 80);

// The information stream can flow, and we can write and read from it
fwrite($connection, $request);

// As long as the server returns something to us...
while(!feof($connection)) {
    // ... print what the server sent us
    echo fgets($connection);
}

// Finally, close the connection
fclose($connection);
```

And indeed, if you put this code snippet into a file `fsockopen.php` and run it with `php fsockopen.php`, you will see the same HTML that you get when you open http://example.com in your browser.

Next step: performing an HTTP request with Assembler... just kidding! But in all seriousness: `fsockopen()` is usually not used to make HTTP requests with PHP; I just wanted to show you that it's feasible, using the easiest possible example. While it *is* possible to make all HTTP (and non-HTTP) interactions work with it, it's not fun and requires a lot of boilerplate code that we don't *need* to do - performing HTTP requests is a solved problem, and in PHP (and many other languages) it's solved by...

cURL

Enter cURL (a **c**lient for **URL**s)! Let's jump right into the code, it's quite straight forward:

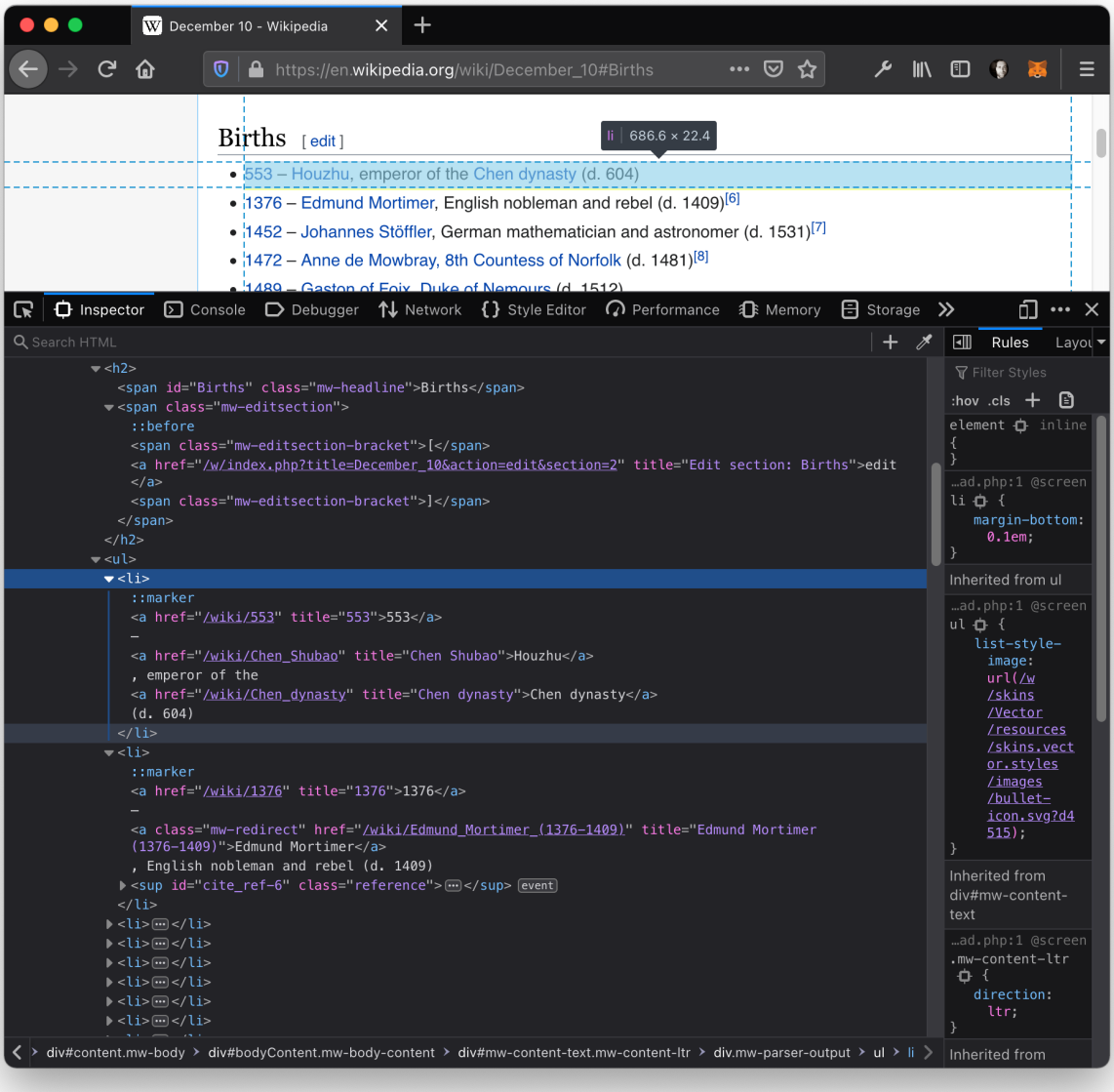
```php
<?php
# curl.php

// Initialize a connection with cURL (ch = cURL handle, or "channel")
$ch = curl_init();

// Set the URL
curl_setopt($ch, CURLOPT_URL, 'http://www.example.com');

// Set the HTTP method
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'GET');

// Return the response instead of printing it out
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

// Send the request and store the result in $response
$response = curl_exec($ch);

echo 'HTTP Status Code: ' . curl_getinfo($ch, CURLINFO_HTTP_CODE) . PHP_EOL;
echo 'Response Body: ' . $response . PHP_EOL;

// Close cURL resource to free up system resources
curl_close($ch);
```

Now, this does look a lot more controlled than our previous example, doesn't it? No need to create a connection to a specific port of a particular server, to manually separate the headers from the actual response, or to close a connection. To follow a website redirect, all we need is a `curl_setopt($ch, CURLOPT_FOLLOWLOCATION, true);`, and there are many more options available to accommodate further needs.

Great! Now let's get to actual scraping!

2. Strings, regular expressions, and Wikipedia

Let's look at Wikipedia as our first data provider. Each day of the year has its own page for historical events, including birthdays! When we open, for example, the page for December 10th (which happens to be *my* birthday), we can inspect the HTML in the developer console and see how the "Births" section is structured:



Wikipedia's HTML structure

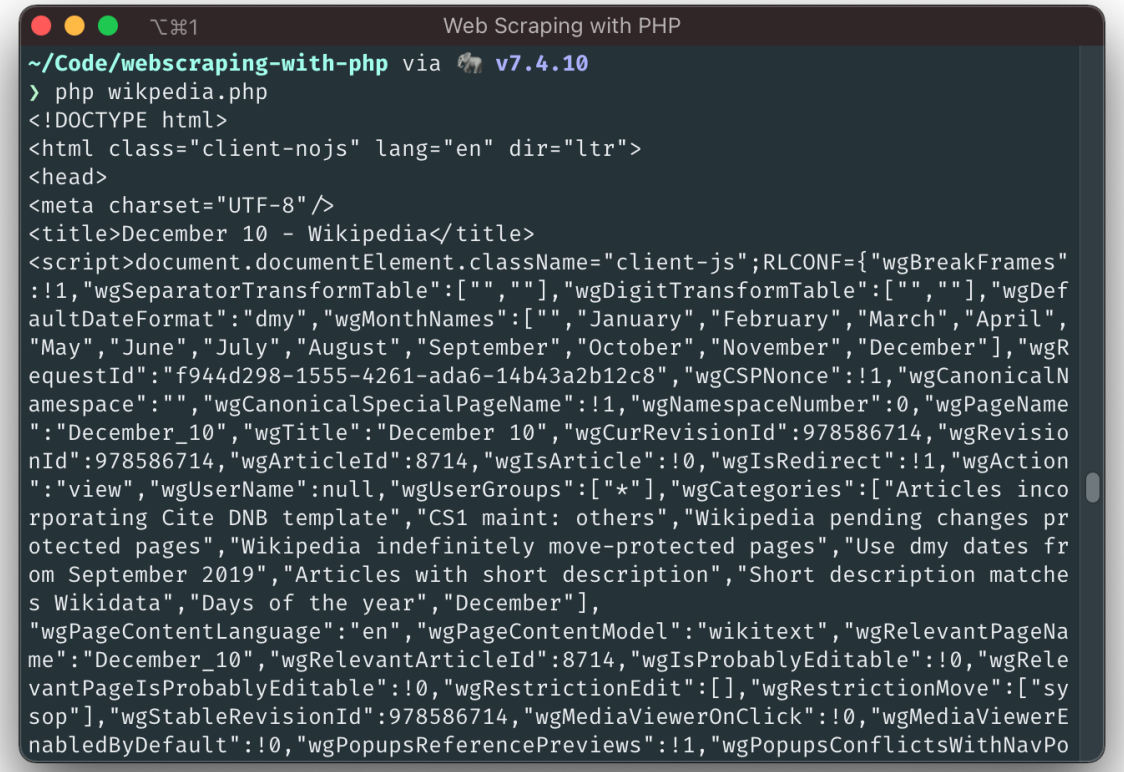
This looks nice and organized! We can see that:

- There's an `<h2>` header element containing `<span id="Births" ...>Births</span>` (only one element on the whole page should™ have an ID named "Births").
- The header is immediately followed by an unordered list (`<ul>`).
- Each list item (`<li>...</li>`) contains a year, a dash, a name, a comma, and a teaser of what the given person is known for.

This is something we can work with, let's go!

```
<?php
# wikipedia.php

$html = file_get_contents('https://en.wikipedia.org/wiki/December_10');

echo $html;
```

Wait what? Surprise! `file_get_contents()` is probably the easiest way to perform uncomplicated GET requests - it's not really *meant* for this use case, but PHP is a language allowing many things that you shouldn't do 🙊. (But it's fine for this example and for one-off scripts when you know what you're requesting).

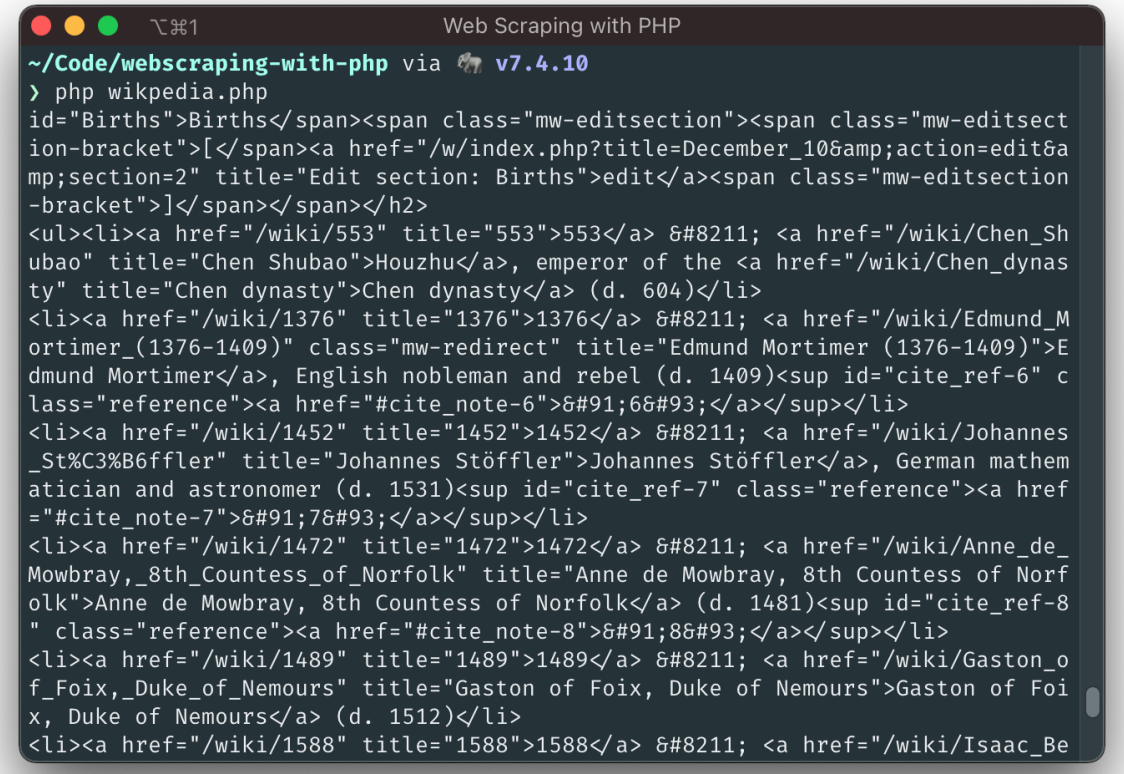


Script output

Have you read all the HTML that the script has printed out? I hope not, because it's a lot! The important thing is that we know where we should start looking: we're only interested in the part starting with `id="Births"` and ending after the closing `</ul>` of the list right after that:

```php
<?php
# wikipedia.php

$html = file_get_contents('https://en.wikipedia.org/wiki/December_10');

$start = stripos($html, 'id="Births"');

$end = stripos($html, '</ul>', $offset = $start);

$length = $end - $start;

$htmlSection = substr($html, $start, $length);

echo $htmlSection;
```

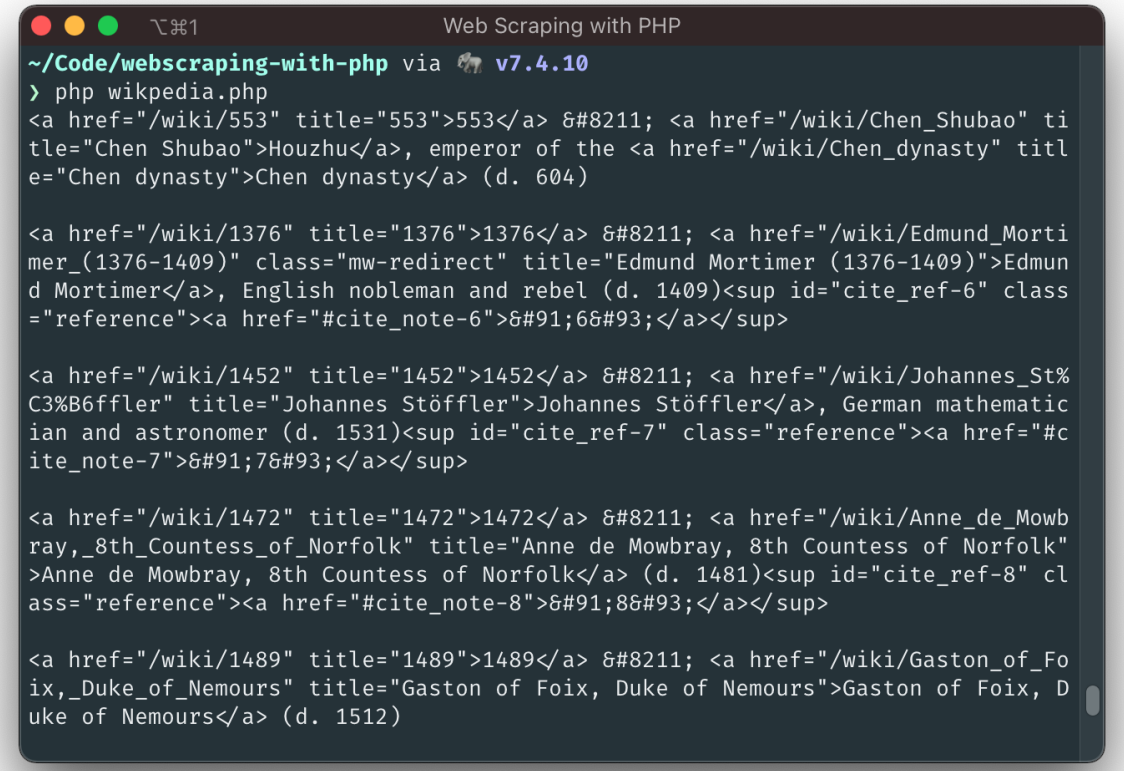
We're getting closer!



Cleaner results

This is not valid HTML anymore, but at least we can see what we're working with! Let's use a regular expression to load all list items into an array so that we can handle each item one by one:

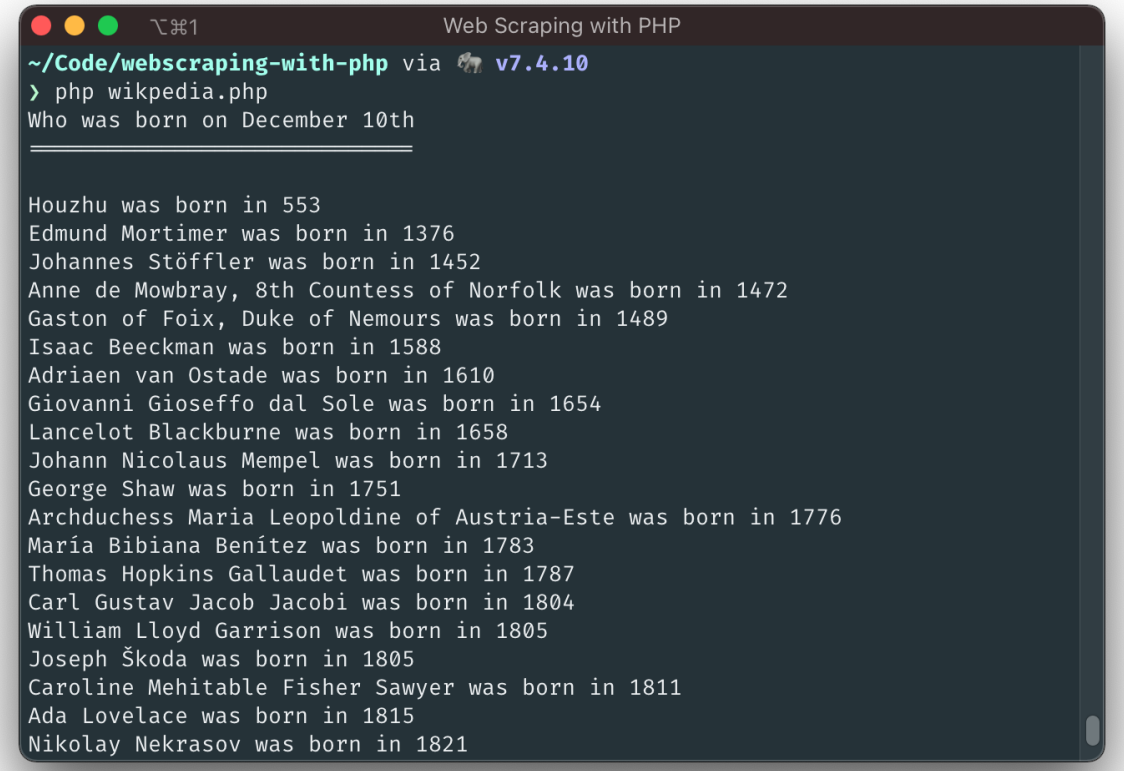
```
preg_match_all('@<li>(.+)</li>@', $htmlSection, $matches);
$listItems = $matches[1];

foreach ($listItems as $item) {
    echo "{$item}\n\n";
}
```



Cleaner results (bis)

For the years and names... We can see from the output that the first number is the birth year. It's followed by an HTML-Entity `&#8211;` (a dash). Finally, the name is located within the following `<a>` element. Let's grab 'em all, and we're done 👌.

```
<?php
# wikipedia.php

$html = file_get_contents('https://en.wikipedia.org/wiki/December_10');

$start = stripos($html, 'id="Births"');

$end = stripos($html, '</ul>', $offset = $start);

$length = $end - $start;

$htmlSection = substr($html, $start, $length);

preg_match_all('@<li>(.+)</li>@', $htmlSection, $matches);
$listItems = $matches[1];

echo "Who was born on December 10th\n";
echo "============================\n\n";

foreach ($listItems as $item) {
    preg_match('@(\d+)@', $item, $yearMatch);
    $year = (int) $yearMatch[0];

    preg_match('@;\s<a\b[^>]*>(.*?)</a>@i', $item, $nameMatch);
    $name = $nameMatch[1];

    echo "{$name} was born in {$year}\n";
}
```



Final results

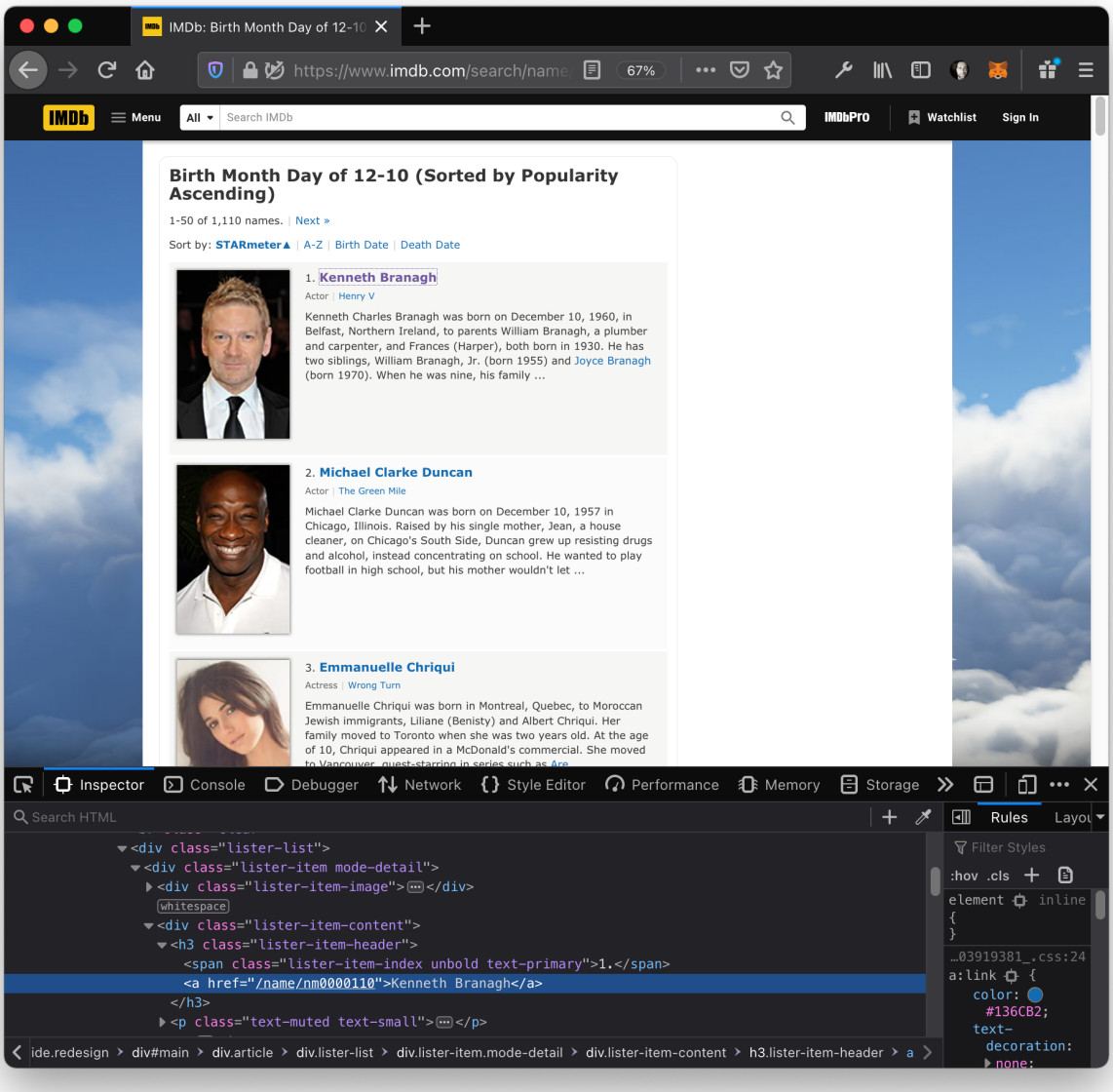
I don't know about you, but I feel a bit dirty now. We achieved our goal, but instead of elegantly navigating the HTML DOM Tree, we destroyed it beyond recognition and ripped out pieces of information with commands that are not easy to understand. And worst of all, this script will show an error with items where the year is *not* wrapped in a link (I didn't show you because the screenshot looks nicer without it 😅).

We can do better! When? Now!

3. Guzzle, XML, XPath, and IMDb

Guzzle is a popular HTTP Client for PHP that makes it easy and enjoyable to send HTTP requests. It provides you with an intuitive API, extensive error handling, and even the possibility of extending its functionality with middleware. This makes Guzzle a powerful tool that you don't want to miss. You can install Guzzle from your terminal with `composer require guzzlehttp/guzzle`.

Let's cut to the chase and have a look at the HTML of https://www.imdb.com/search/name/?birth_monthday=12-10 (Wikipedia's URLs were definitely nicer)



IMDB HTML structure

We can see straight away that we'll need a better tool than string functions and regular expressions here. Instead of a list with list items, we see nested `<div>` s. There's no `id="..."` that we can use to jump to the relevant content. But worst of all: the birth year is either buried in the biography excerpt or not visible at all! 😱

We'll try to find a solution for the year-situation later, but for now, let's at least get the names of our jubilees with XPath, a query language to select nodes from a DOM Document.

In our new script, we'll first fetch the page with Guzzle, convert the returned HTML string into a `DOMDocument` object and initialize an XPath parser with it:

```
<?php
# imdb.php

require 'vendor/autoload.php';

$httpClient = new \GuzzleHttp\Client();

$response = $httpClient->get('https://www.imdb.com/search/name/?birth_monthday=12-10');

$htmlString = (string) $response->getBody();

// HTML is often wonky, this suppresses a lot of warnings
libxml_use_internal_errors(true);

$doc = new DOMDocument();
$doc->loadHTML($htmlString);

$xpath = new DOMXPath($doc);
```

Let's have a closer look at the HTML in the window above:

- The list is contained in a `<div class="lister-list">` element
- Each direct child of this container is a `<div>` with a `lister-item mode-detail` class attribute
- Finally, the name can be found within an `<a>` within a `<h3>` within a `<div>` with a `lister-item-content`

If we look closer, we can make it even simpler and skip the child divs and class names: there is only one `<h3>` in a list item, so let's target that directly:

```
$links = $xpath->evaluate('//div[@class="lister-list"][1]//h3/a');

foreach ($links as $link) {
    echo $link->textContent.PHP_EOL;
}
```

- `//div[@class="lister-list"][1]` returns the first (`[1]`) `div` with an attribute named `class` that has the exact value `lister-list`
- within that div, from all `<h3>` elements (`//h3`) return all anchors (`<a>`)
- We then iterate through the result and print the text content of the anchor elements

I hope I explained it well enough for this use case, but in any case, our article "Practical XPath for Web Scraping" here on this blog explains XPath far better and goes much deeper than I ever could, so definitely check it out (but finish reading this one first! 💪)

> 💡 We released a new feature that makes this whole process way simpler. You can now extract data from HTML with one simple API call. Feel free to check the documentation here.

4. Goutte and IMDB

Guzzle is one HTTP client, but many others are equally excellent - it just happens to be one of the most mature and most downloaded. PHP has a vast, active community; whatever you need, there's a good chance someone else has written a library or framework for it, and web scraping is no exception.

Goutte is an HTTP client *made* for web scraping. It was created by Fabien Potencier, the creator of the Symfony Framework, and combines several Symfony components to make web scraping very comfortable:

- The BrowserKit component simulates the behavior of a web browser that you can use programmatically
- Think of the DomCrawler component as DOMDocument and XPath on steroids - except that steroids are bad, and the DomCrawler is good!
- The CssSelector component translates CSS queries into XPath queries.
- The Symfony HTTP Client is a relatively new component (it was released in 2019) - being developed and maintained by the Symfony team, it has gained in popularity very quickly.

Let's install Goutte with `composer require fabpot/goutte` and recreate the previous XPath with it:

```
<?php
# goutte_xpath.php

require 'vendor/autoload.php';

$client = new \Goutte\Client();

$crawler = $client->request('GET', 'https://www.imdb.com/search/name/?
birth_monthday=12-10');

$links = $crawler->evaluate('//div[@class="lister-list"][1]//h3/a');

foreach ($links as $link) {
    echo $link->textContent.PHP_EOL;
}
```

This alone is already pretty good - we saved the step where we had to explicitly disable XML warnings and didn't need to instantiate an `XPath` object ourselves. Now, let's replace the XPath expression with a CSS query (thanks to the CSSSelector component integrated into Goutte):

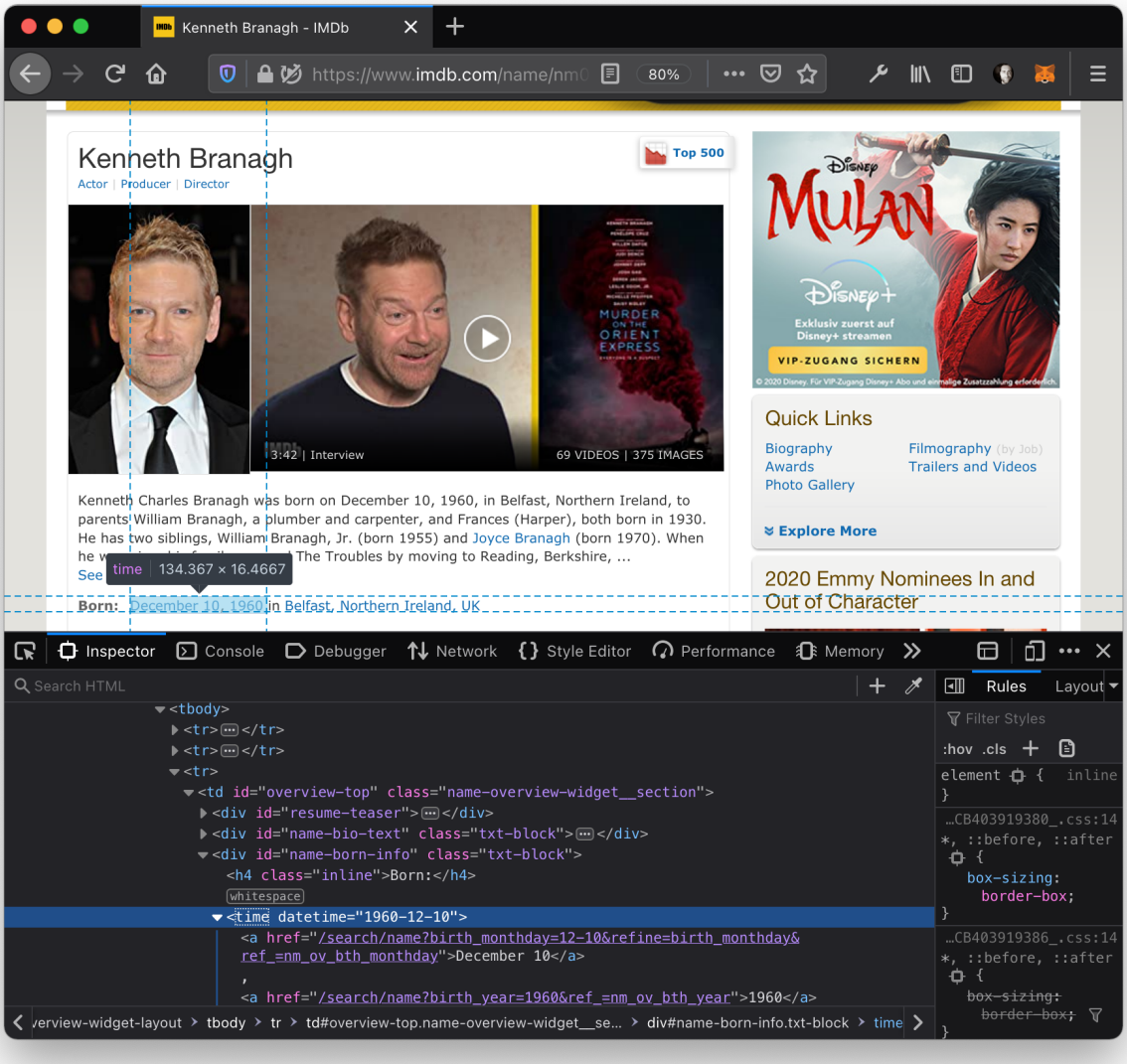
```
<?php
# goutte_css.php

require 'vendor/autoload.php';

$client = new \Goutte\Client();

$crawler = $client->request('GET', 'https://www.imdb.com/search/name/?
birth_monthday=12-10');

$crawler->filter('.lister-list h3 a')->each(function ($node) {
    echo $node->text().PHP_EOL;
});
```

I like where this is going; our script is more and more looking like a conversation that even a non-programmer can understand, not just *code* 🥰. However, now is the time to find out if you're coding along or not 🤨: does this script return results when running it? Because for me, it didn't at first - I spent an hour debugging why and finally discovered a solution:

```
composer require masterminds/html5
```

As it turns out, the reason why Goutte (more precisely: the DOMCrawler) doesn't report XML warnings is that it just throws away the parts it cannot parse. The additional library helps with HTML5 specifically, and after installing it, the script runs as expected.

We will talk more about this later, but for now, let's remember that we're still missing the birth years of our jubilees. This is where a web scraping library like Goutte really shines: we can click on links! And indeed: if we click one of the names in the birthday list to go to a person's profile, we can see a "Born: " line, and in the HTML a `<time datetime="YYYY-MM-DD">` element within a div with the id `name-born-info`:



IMDB HTML stucture

This time, I will not explain the single steps that we're going to perform beforehand, but just present you the final script; I believe that it can speak for itself:

```php
<?php
# imdb_final.php

require 'vendor/autoload.php';

$client = new \Goutte\Client();

$client
    ->request('GET', 'https://www.imdb.com/search/name/?birth_monthday=12-10')
    ->filter('.lister-list h3 a')
    ->each(function ($node) use ($client) {
        $name = $node->text();

        $birthday = $client
            ->click($node->link())
            ->filter('#name-born-info > time')->first()
            ->attr('datetime');

        $year = (new DateTimeImmutable($birthday))->format('Y');

        echo "{$name} was born in {$year}\n";
});
```

Look at this clean output

As there are 50 people on the page, 50 additional GET requests have to be made, so the run of this script takes some time - but it works and gives us some opportunities to improve it even further:

- Guzzle supports concurrent requests; perhaps we could leverage them to improve the processing speed.
- The IMDb page we scraped included 50 people out of 1,110 - we could certainly grab the "Next" link at the bottom of the page to get more birthdays.
- With all the knowledge that we've built up so far, it shouldn't be too hard to download our celebrities' profile pictures.

5. Headless Browsers

Here's a thing: when we looked at the HTML DOM Tree in the Developer Console, we didn't see the actual HTML code that has been sent from the server to the browser, but the final result of the browser's_interpretation_ of the DOM Tree. In the static HTML case, the output might not differ, but the more JavaScript is embedded in the HTML source, the more likely it will be that the resulting DOM tree is very different. When a website uses AJAX to dynamically load content, or when even the complete HTML is generated dynamically with JavaScript, we cannot access it with just downloading the original HTML document from the server. Tools like Goutte can simulate a browser's behavior to make it easier for us, but only full-blown browsers can fully handle modern websites.

This is where so called headless browsers come into play. A headless browser is a browser engine without a graphical user interface and can be controlled programmatically in a similar way as we did before with the simulated browser.

Symfony Panther is a standalone library that provides the same APIs as Goutte - this means you could use it as a drop-in replacement in our previous Goutte scripts. A nice feature is that it can use an already existing installation of Chrome or Firefox on your computer so that you don't need to install additional software.

Since we have already achieved our goal of getting the birthdays from IMDB, let's conclude our journey with getting a screenshot from the page that we so diligently parsed.

After installing Panther with `composer require symfony/panther` we could write our script for example like this:

```
<?php
# screenshot.php

require 'vendor/autoload.php';

$client = \Symfony\Component\Panther\Client::createFirefoxClient();
// or
// $client = \Symfony\Component\Panther\Client::createChromeClient();

$client
    ->get('https://www.imdb.com/search/name/?birth_monthday=12-10')
    ->takeScreenshot($saveAs = 'screenshot.jpg');
```

Conclusion

We've learned about several ways to scrape the web with PHP today. Still, there are a few topics that we haven't spoken about - for example, website providers like their sites to be seen in a browser and often frown upon being accessed programmatically.

- When we used Goutte to load 50 pages in quick succession, IMDb could have interpreted this as unusual and could have blocked our IP address from further accessing their website.
- Many websites have rate limiting in place to prevent Denial-of-Service attacks.
- Depending on in which country you live and where a server is located, some sites might not be available from your computer.
- Managing headless browsers for different use cases can take a toll on you and your computer (mine sounded like a jet engine at times).

That's where services like ScrapingBee can help: you can use the Scraping Bee API to delegate thousands of requests per second without the fear of getting limited or even blocked so that you can focus on what matters: the content 🚀.

If you'd rather use something free, we have also benchmarked thoroughly the most used free proxy provider.

If you want to read more about web scraping without being blocked, we have written a complete guide, but we still would be delighted if you decided to give Scraping Bee a try, the first 1,000 requests are on us!

Jérôme Gamez

Jérôme is an experienced PHP developer very active in the Open-Source community, if you use PHP and Firebase, you should check-out his SDK on Github (1.4k stars).