www.phparch.com

August 2019
Volume 18 - Issue 8

php[architect]

# Renovating Applications with Symfony

## Symfony 4: A New Way to Develop Applications

## How to Deal With Legacy Code

## Can You Migrate Any Legacy Code Under One Month?

## Cultivating a Community:
Five Things I've Learned Running a PHP User Group

### ALSO INSIDE

**Education Station:**
Writing DRY, SOLID FOSS OOP CRUD Code

**Community Corner:**
Why Soft Skills are Hard Skills

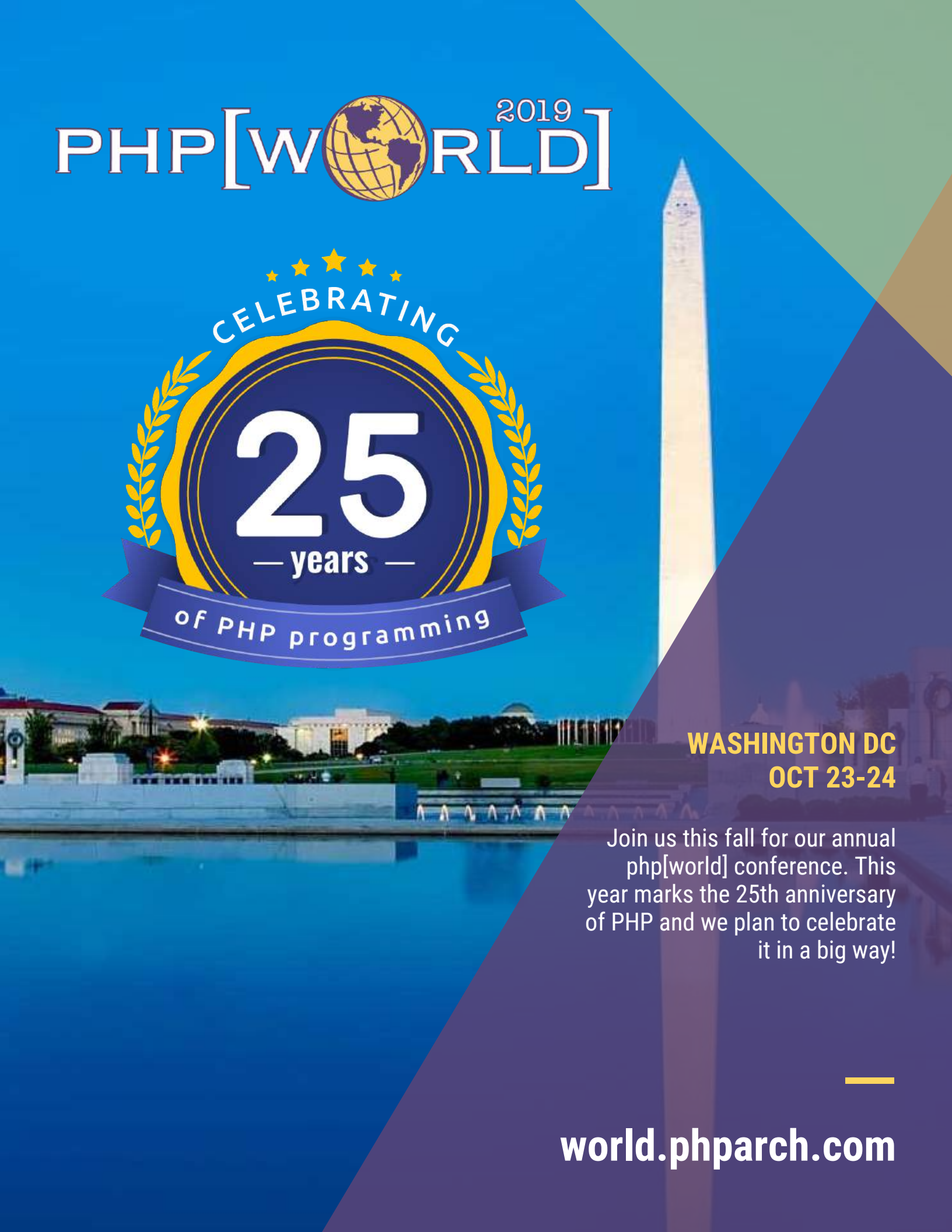
**Internal Apparatus:**
Generated Singletons

**Security Corner:**
System Enumeration

**The Workshop:**
Introduction to PDF Generation

**finally{}:**
25 Years of PHP

# PHP[WORLD] 2019

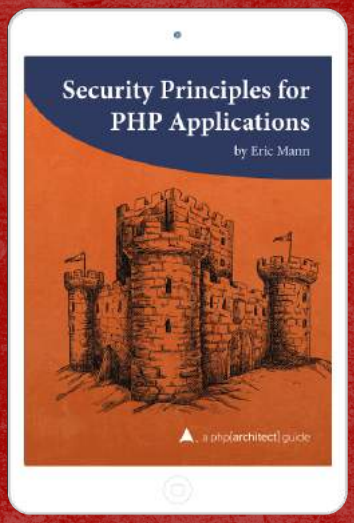


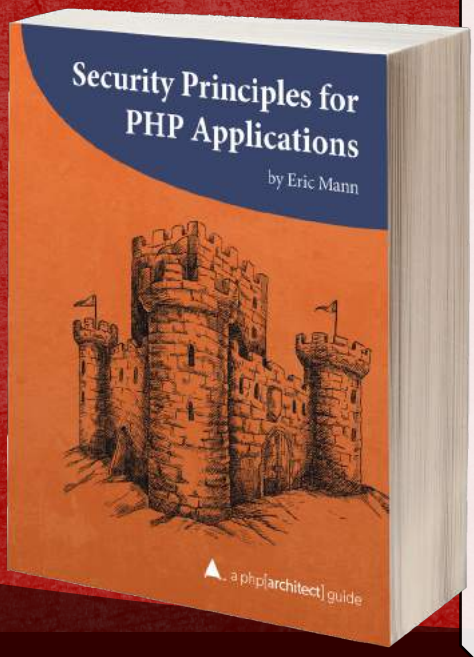

**WASHINGTON DC**
**OCT 23-24**

Join us this fall for our annual php[world] conference. This year marks the 25th anniversary of PHP and we plan to celebrate it in a big way!

**world.phparch.com**

a php[architect] guide




## Discover how to secure your applications against many of the vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

*Security Principles for PHP Applications* is a comprehensive guide. This book contains examples of vulnerable code side-by-side with solutions to harden it. Organized around the 2017 OWASP Top Ten list, topics cover include:

- Injection Attacks
- Authentication and Session Management
- Sensitive Data Exposure
- Access Control and Password Handling
- PHP Security Settings
- Cross-Site Scripting
- Logging and Monitoring
- API Protection
- Cross-Site Request Forgery
- ...and more.

Read a Sample Online

Written by PHP professional Eric Mann, this book builds on his experience in building secure, web applications with PHP.

# Order Your Copy

**http://phpa.me/security-principles**

# Introduction to PDF Generation

*Joe Ferguson*

**Despite the promise of a "paperless" office, we still need to create documents that print and render nearly-identically across devices and operating systems. PDFs have filled this niche nicely for end-users, but if you need to generate PDFs with PHP programmatically, the options are overwhelming. How do you choose? In this series, we'll investigate the solutions at our disposal and the pros and cons of each.**

In early July 2019, I asked Reddit *What questions do you have about generating PDFs w/ PHP?*[1]. I was pleasantly surprised at the genuine answers and feedback I received (as opposed to the normal nonstop trolling Reddit is usually known for). One comment listed 19 different questions! Many of the comments mention different libraries and services, and I believe even one of the commercial products had a representative chime in about their solution. While I was excited to get so much interest from the Reddit community, I was now a bit overwhelmed with options. This task was suddenly bigger than Oscar Merida saying, "I can't get anyone to write about this. Why don't you do it?" Now equipped with more than 70 comments from internet strangers about PDF generation, we're going to explore some of the more basic options and step up in complexity with the same goal designed for each library. We'll put the library through its paces to give a short overview so you can pick a library to try out for yourself.

## PDF Files

The Portable Document Format (PDF) was created in the early 1990s by Adobe Systems. PDFs were added to the desktop publishing workflow as a way to share documents without having to worry about what operating system or platform other users were on. Way before the browser wars, we had the word processing wars. Microsoft Word couldn't open Corel WordPerfect documents and vice versa. These were the days when "Microsoft Office" experience on a job posting was *very* serious, as serious as we take our programming languages. It also led to "WordPerfect" based publishing shops and "Microsoft Word" shops to denote the tooling used. PDFs to the desktop publishing world was essentially a mutually agreed upon specification on how to present documents across platforms independent of the authoring tool. Some 26 years later, the use case is still a big deal to just about every computer or technology user: being able to create a document that looks and feels the exact same way no matter how we consume the document, whether via browser, desktop application, mobile email client, or any other way you can think to display a document to a user.

Adobe held tight control over the PDF specifications until 2007 when they announced the release of the full Portable Document Format 1.7 specification to the American National Standards Institute (ANSI). Adobe now publishes PDF extensions; these extensions are *not* part of the PDF standards. A year after Adobe turned over the 1.7 specification, the ISO Technical Committee 171 published ISO 32000-1:2008 named "Document management—Portable document format – Part 1: PDF 1.7." You can view the full spec in PDF format[2]. In July 2017, the ISO committee published ISO 32000-2 (PDF 2.0), the first version of the PDF specification to be entirely developed by the ISO Committee process. The PDF 2.0 specification allowed the deprecation of aging parts of the original specification, as well as the standardization of PDF subset standards. This allows for the extension of the PDF without having to add to the existing specification. These subsets focus on the specific use case of PDF publishing such as PDF/A for long term archiving and PDF/E for Engineering (Building, Manufacturing, and Geospatial). These subsets allowed entire industries to adopt the PDF for their document publishing and sharing purposes.

The reason we're still talking about PDFs in 2019 is that the specification is *really good* when it comes to making *portable* documents. The open specifications allowed software vendors to adopt PDF support into their programs easily and quickly just about everyone in the software world supported the format due to the straightforward ability to create and share these documents without having to reformat them, or suffer through inadequate conversion tools that would mangle the output. Chances are you're reading this article via a PDF. It allows php[architect] to format the magazine once, and know it'll be displayed the same way on every platform (*Editors note: except Microsoft Edge's PDF renderer, for some reason*).

Think of PDFs as a container running your content. The application runs the same way on any server platform, much like a magazine displays the same way on any viewing platform. While consuming PDFs has become quite trivial and ordinary, creating PDFs can be challenging.

1 *What questions do you have about generating PDFs w/ PHP?:* https://phpa.me/reddit-php-pdfgen

2 *PDF format:* https://phpa.me/pdf-1-7-spec

Most PHP developers who work in a business servicing customers in some way have had to write code that turns text and graphics into a PDF file. The most common use cases in my development career have been generating customer receipts and invoices either to match the paper copy versions to be filed away or maybe as the only copy of a receipt for a company that has gone entirely digital. Nearly all ebook publishes offer PDF as a standard option to consume the materials. The healthcare industry also uses PDFs extensively to store records and forms for patients to fill out. PHP has basic PDF generation abilities via the free PDFlib[3] library which has been unmaintained since 2010. Most developers reach for libraries to create PDF files. There are also commercial software-as-a-service options for developers to use so they completely offload this task. Depending on your needs, you may fit into this category. Always look around to see who's already solved a problem before you take on writing (and maintaining!) custom code. We don't want to reinvent the wheel, so we're going to outline a somewhat basic scenario of a PDF we need to build and investigate different options to get us as close to our final product as we can.

The PHP documentation mentions two libraries specifically to get you started generating PDF files. FPDF[4], and TCPDF[5]. We're going to take a basic look at FPDF, and over the next two installments of this series, we'll get more and more complex and demonstrate other libraries.

> *I'm going to be using Laravel because it bootstraps all the nice frontend things for me. I don't have to worry about writing that code, and I can focus on PDF generating code. If you want to look over the code, you can find it on GitHub[6].*

We can easily add FPDF to our application via Composer thanks to fpdf/fpdf—Packagist[7], which is a wrapper around the primary class as shown in Output 1.

We're going to create a new controller in our application. If you're following along with the repository, this code is located in `app/Http/Controllers/FpdfCreate.php`. We begin by creating a new `FPDF` instance and passing in the basics: orientation (portrait), unit of measurements (in for inches), and the paper size (letter). For our European friends, you could also use `mm` for measurements and `A4` for size. We continue our set up by specifying the font, style, and size we want. We'll use `Arial` as our font, leave style blank (or use `B` for bold), and then set our size to 14.

Before we get too ahead of ourselves, we need to call `AddPage()` to create a page in our PDF file. Wouldn't it be silly to add content before adding a page? Now that we have described how the text should appear, we need to call the `Cell()` method to place something on our page. The `Cell()` method takes several parameters. Think of it as bootstrapping your document at a very low level; we specify height, width, and other parameters to describe how to place our content on the page. In the example below, we use `4` for width, which becomes four inches because we used `in` as our unit when we called `FPDF()`. We'll use `.5`, half an inch, for our height. We'll pass a string `We made a PDF!` as our text to place on the page.

The next parameter is the border. To get a feel for how cells work, we'll use `LTRB` which means left, top, right, bottom border lines. The second to last parameter we need to specify is the `ln` which is an indication where the current position should go after the call to `Cell()` happens. The options are `0` to go to the right, `1` to go to the beginning of a new line, or `2` which is below. We'll use `0` as we may want to add more content later that picks up right where we left off. The last option we specify is the alignment; we specify `C`, so our text is centered in our 4-by-.5 inch cell. You can view our full method in Listing 1.

**Output 1**

```
$ composer require fpdf/fpdf
Using version ^1.81 for fpdf/fpdf
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing fpdf/fpdf (1.81.2): Downloading (100%)
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: beyondcode/laravel-dump-server
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
```

**Listing 1**

```
public function createPdf() {
    $pdf = new FPDF('P', 'in', 'Letter');
    $pdf->AddPage();
    $pdf->SetFont('Arial', '', 14);
    $pdf->Cell(
        4, // width
        .5, //height
        'We made a PDF!', // text
        'LTRB', // border
        0, // where the current position goes after the call
        'C'
    );

    return Response::make($pdf->Output(), 200, [
        'Content-Type' => 'application/pdf',
    ]);
}
```

3 *PDFlib: https://php.net/intro.pdf*

4 *FPDF: http://www.fpdf.org*

5 *TCPDF: https://tcpdf.org*

6 *GitHub: https://github.com/svpernova09/pdf-creation*

7 *fpdf/fpdf—Packagist: https://packagist.org/packages/fpdf/fpdf*

> *Since I'm using Laravel I have access to the* `Response` *helper class which generates a response for me. You'll want to ensure whatever you're using correctly sets the* `Content-Type` *header to* `application/pdf` *so that your browser correctly renders the PDF.*

We'll see Figure 1 if we load up our URL in the browser: `/fpdf/create`.

All of the parameters we passed to the `Cell()` method made sense, but `ln`, which means "line break." It dictates where the cursor goes after the method call, which can be confusing. We intentionally used `0` in our example to be able to continue right where we left off. Let's add another cell and see how things line up:

```
$pdf->Cell(
   4, // width
   .5, //height
   'This Cell should be on the right', // text
   'LTRB', // border
   0, // where the current position should go after the call
   'C'
);
```

This added call ends up placing our new cell right next to the first cell we created as we would expect (see Figure 2).

What if we wanted to move the cell on the right to below our first cell? We can easily adjust the code as you can see in Listing 2.

If we refresh our browser, we'll see the cell which was on the right is now below our first cell as in Figure 3.

We could also add another page and move one of our boxes off the first page by calling `AddPage()` again. See Listing 3.

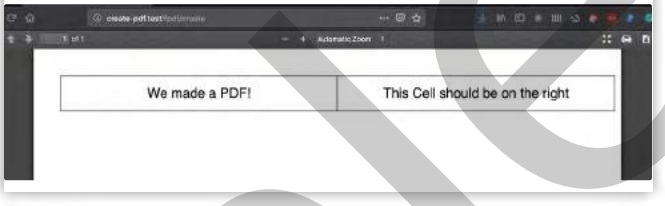
Now, when we refresh our browser, we can scroll down to page two to see our second cell, shown in Figure 4.

**Listing 3**

```
1. <?php
2. $pdf->Cell(
3.     4, // width
4.     .5, //height
5.     'We made a PDF!', // text
6.     'LTRB', // border
7.     2, // where the current position goes after the call
8.     'C'
9. );
10.
11. $pdf->AddPage();
12. $pdf->Cell(
13.     4, // width
14.     .5, //height
15.     'Welcome to the Second Page!', // text
16.     'LTRB', // border
17.     0, // where the current position goes after the call
18.     'C'
19. );
```

**Figure 1**


**Figure 2**




**Listing 2**

```
1. $pdf->Cell(
2.     4, // width
3.     .5, //height
4.     'We made a PDF!', // text
5.     'LTRB', // border
6.     1, // where the current position goes after the call
7.     'C'
8. );
9. $pdf->Cell(
10.     4, // width
11.     .5, //height
12.     'This Cell should be below', // text
13.     'LTRB', // border
14.     0, // where the current position goes after the call
15.     'C'
16. );
```

**Figure 3**

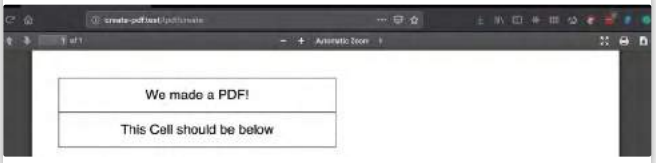



**Figure 4**

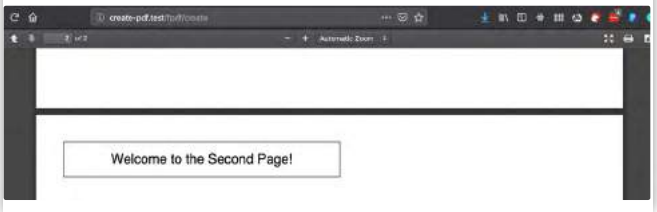


Figure 5


The last thing we're going to cover this month is adding images. We can add an image to our PDF by placing the image in an accessible folder within our application (such as the application root, or where you store your images). Since I'm using Laravel, I'm going to use the `storage_path` helper to determine the path to my files easily. The `Image()` method takes several parameters with the first being the path to the image file itself. We're going to skip over the next two, `x` and `y`, which are coordinates of where the image should be placed. By leaving these `null`, the image is placed where the cursor currently exists, at the start of the document. We'll then pass `6` in as the fourth parameter to set the image width to 6 inches.

```
$pdf->Image(
    storage_path() . '/app/public/world2019.png',
    null,
    null,
    '6'
);
```

Refreshing our browser, we can see in Figure 5 the php[world] logo[8]. (Make sure you have your tickets!):

We've covered the basics of getting started and building simple PDFs with PHP and FPDF. FPDF was easy to install just like any other package via Composer, which is a big plus. Another plus is how low level we're working to build our PDFs. Choosing how low level (you doing more lifting), as opposed to a higher level (where the library does more work), is going to be a tradeoff based on individual use cases. In many cases, having to do low-level operations can be painful, and you may want to skip over FPDF in favor of a higher-level library. We're not quite ready to take this PDF into production yet. Next month, we'll cover a practical scenario of PDF generation by creating receipts for customers who bought an item from our fictitious online store.

Happy coding!

*Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson*

8 *logo:* https://world.phparch.com


# Borrowed this magazine?

## Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

**Digital and Print+Digital Subscriptions Starting at $49/Year**

**http://phpa.me/mag_subscribe**

php[**architect**]