www.phparch.com

July 2023
Volume 22 - Issue 7

php[architect]

The Magazine For PHP Professionals

Be Barrier Free

Exploring PHP 8's JIT Compiler

Symfony Image Uploads With Cloud Static Object Storage

ALSO INSIDE

CONTENTS

JULY 2023
Volume 22 - Issue 07

php[architect]

Edited in a state of denial

php[**architect**] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Be Barrier Free

I am sure that I am not alone in thinking, "My site is so small; I don't have to worry about it". "I have a focused user base; everything is fine." "Nobody has complained, so it must not be a problem."

I am referring to the fact that most websites create barriers to usability. Most of us do not have barriers to our vision, so we don't think about it. We often think that only the monster sized sites, like Google and Facebook, need to worry about accessibility for the masses. I am guilty of knowing it's important and then not thinking about it when writing code.

I am proud that we here at PHP Architect are adding the Barrier-Free Bytes column to our magazine. I hope this will inspire all of our readers to do a better job at making our applications less of a barrier to as many people as possible.

This system of having barriers on our website is akin to the Prisoner's Dilemma outlined in this month's Security Corner. As developers, we often code for the "happy path" just to get our job done and a project for our client completed. However, we should take more into consideration and make thoughtful decisions based on all the data at our disposal.

This month brings our first feature article, *Symfony Image Uploads With Cloud Static Object Storage*, by Sherri Wheeler. Almost gone are the days of storing files locally to the web server. As we make our websites more scalable and deployments more automated, the uploaded files need to be available to all of your servers. Take a few minutes to learn about the AWS PHP SDK today.

Next up, *Exploring PHP 8's JIT Compiler: Performance Boosts and Limitations*, by Rahul Kumar, can help you boost your application's speed. Learn about what currently does and does not work with the PHP 8 Just In Time (JIT) compiler. How incredible is it that we are on PHP version 8, and we are still finding ways to improve the speed and performance of our code with very little effort on our part as a developer. This means that we can get some benefit today and when we do find time to finally optimize our own code, the performance will be incredible.

As I mentioned above, we want to help make your website Barrier Free to as many people as possible, so we will begin to learn about Site Navigation in this month's column from Maxwell Ivey. He is sharing with us how he has to navigate web applications and the struggles that he faces in doing so.

Over in the Workshop, Joe Ferguson is continuing his series on databases, bringing us some information on *PostgreSQL*. I have talked to many DBAs over the years, and many of them tout the virtues of PostgreSQL, and I need to take Joe's article and really start to learn it.

Artisan Way by Matt Lantz discusses *MPA vs. SPA vs. Transitional* websites application trends. There are pros and cons in everything we do, so learn to make the right decision for you and your application.

Chris Tankersly brings us *Serverless PHP with Bref* in this month's Education Station. Let's use PHP as an AWS Lambda function.

Over in Security Corner, we have the *Prisoner's Dilemma* from Eric Mann. Have you ever faced an ethical dilemma while writing code?

Our rats are getting more clever over in PHP Puzzles with Oscar Merida's, *Maze Directions*. This article will wrap up the maze rats series.

Frank Wallen looks at four PSRs that are currently in DRAFT status with the PHP-FIG in this month's PSR Pickup, *PSRs and PERs: What's Next?*.

Edward Bernard continues his DDD series with *Create Observability, Part 2: Capture "Tribal Knowledge"*. As a holder of Tribal Knowledge for one of our bigger clients at PHP Architect, I know the struggle of getting developers up to speed when they onboard onto the project. Let's create some observability and capture that knowledge.

And finally{}, Beth Tucker Long asks if you are *Living on the Edge*.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us.

- Subscribe to our list: https://phpa.me/sub-to-updates
- Mastodon: @editor@phparch.social
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

https://phpa.me/July2023_code

Symfony Image Uploads With Cloud Static Object Storage

Sherri Wheeler

When we can't store files on the web server's filesystem, we can turn to static object storage. Services like AWS S3 are cheap, scalable, and reliable, so now is the time to learn about the AWS SDK for PHP.

I've been working on a new Symfony project recently, and the client informed me they would be hosting it on AWS. Unfdocortunately, I'd already built the functionality to attach profile pictures to user accounts by storing those images on the web server's filesystem. However, with S3, I can't store any data on the server's disk because the server can be regenerated at any time. The client will provide me with an AWS S3 bucket[1] I can use instead. Static object storage is cheap, scalable, and highly available, so it's time to learn a bit about the AWS SDK for PHP[2]. I've modified my application to store image uploads on S3 instead of the local disk and accomplished the same task using Digital Ocean's S3-compatible Spaces[3] object storage. Let's have a look.

Using the Aws Php Sdk

Before we jump into using the SDK with Symfony, let's first find out what this PHP library provides out of the box, so we know we can use it with any PHP application, irrespective of framework. Amazon has PHP SDK documentation[4] for this library with some getting-started information. Following along, we install the SDK with Composer.

```
composer require aws/aws-sdk-php
```

An S3 bucket should be set up, and you need to retrieve your Access Key and Secret Access Key for the bucket from your AWS panel. The documentation prefers that you have these values available as environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, respectively. In that case, we can create an S3 client object by passing in the version and region configured for the bucket, and it will use the environment variables automatically. (See Listing 1)

You can pass credentials to the constructor if you don't have your keys in environment variables. *Just make sure that your keys are not committed to your source code repository!*

Listing 1.

```
1. <?php
2. require 'vendor/autoload.php';
3.
4. use Aws\S3\S3Client;
5. use Aws\Exception\AwsException;
6.
7. // Create an S3Client.
8. $s3Client = new Aws\S3\S3Client([
9.     'version' => 'latest',
10.    'region'  => 'us-east-2'
11. ]);
```

```
use Aws\Credentials\Credentials
// Load key and secret from your application's
// secure values config.
$credentials = new Credentials('key', 'secret');

$s3Client = new Aws\S3\S3Client([
    'version'     => 'latest',
    'region'      => 'us-west-2',
    'credentials' => $credentials
]);
```

Once we have an S3 client, we can save a file to the bucket in a single command.

```
try {
    $s3Client->putObject([
        'Bucket' => 'acme-bucket',
        'Key'    => 'profile.jpg',
        'Body'   => fopen('/path/to/profile.jpg', 'r'),
    ]);
} catch (Aws\S3\Exception\S3Exception $e) {
  // Handle failed upload.
  echo 'Error: ' . $e->getAwsErrorMessage() . PHP_EOL;
}
```

The key for the object is used as a path and file name. You'll use this to manipulate and serve the object. If this doesn't work, you may need to check your bucket's ACL settings.

There are many more useful methods for the client, but immediately useful are the methods to delete an object and list a bucket's contents. (See Listings 2 & 3 on the next page)

Now that we know how the AWS SDK will work under the hood, let's find a way to abstract some of this functionality

1 https://aws.amazon.com/s3/

2 https://aws.amazon.com/sdk-for-php/

3 https://www.digitalocean.com/products/spaces

4 https://phpa.me/amazon

Listing 2.

```
try
{
  $result = $s3Client->deleteObject([
      'Bucket' => 'acme-bucket',
      'Key'    => 'profile.jpg'
  ]);
} catch (Aws\S3\Exception\S3Exception $e) {
  // Handle failed deletion.
  echo 'Error: ' . $e->getAwsErrorMessage() . PHP_EOL;
}
```

Listing 3.

```
try {
    $contents = $s3Client->listObjects([
        'Bucket' => 'acme-bucket',
        "Prefix" => "path/to/", // optional
    ]);
    echo "Bucket contents: \n";
    foreach ($contents['Contents'] as $content) {
        echo $content['Key'] . "\n";
    }
} catch (Exception $e) {
    // Handle failed listing of objects.
    echo('Error: ' . $e->getMessage() . PHP_EOL);
}
```

away and better integrate it with our plan to store file uploads coming from a web form in Symfony.

> *An Important Note: Cloud storage is usually billed via a combination of volume of data stored and the upload/download bandwidth used to store and retrieve these files. Keep an eye on your billing plan for these services to avoid surprises on your invoice.*

Using Flysystem

Helpfully suggested to me by several people on Mastodon is to use the PHP library Flysytem[5]. This library provides an abstraction layer for file storage and allows you to swap easily between storing on the filesystem, in memory, or using several cloud storage providers. It's by The PHP League, so I instantly get a good feeling about its quality and that it will remain well maintained. We'll be using version 3 in a Symfony 6.2 application.

Setup

First, we have to install several libraries, including the AWS SDK, Flysystem, and its AWS adapter. We will also be using the Vich file uploader bundle[6] to handle file uploads in our forms.

5 *Flysytem: https://phpa.me/github*

6 *Vich file uploader bundle: https://phpa.me/github-dustin*

```
composer require league/flysystem-bundle
composer require league/flysystem-aws-s3-v3
composer require aws/aws-sdk-php
composer require vich/uploader-bundle
```

Next, we have some configuration to set up, including ENV variables in our environment-specific .env files, which we DO NOT check into git. Throughout this article, we'll use 'acme' to stand in for our company or app-specific tag/namespace.

```
# .env.dev.local
AWS_ACCESS_KEY="ABC123"
AWS_SECRET_KEY="abcxyz123"
AWS_REGION="us-east-1"
AWS_STORAGE_BUCKET="acme-bucket"
FLYSYSTEM_ADAPTER_STORAGE="acme.storage.aws"
```

The `FLYSYSTEM_ADAPTER_STORAGE` variable will toggle between two different storage options we will configure so we can easily switch between them: the local filesystem and an AWS S3 bucket. We have already configured our bucket to allow ACLs and ensure uploaded files are publicly accessible.

Next, we set up the configuration for our AWS client adapter in services.yaml. We disable shared config files in order to prevent warnings in our logs since this file doesn't exist. (See Listing 4)

Listing 4.

```
# config/services.yaml
services:
    acme.s3_client:
        class: Aws\S3\S3Client
        arguments:
            -
                version: '2006-03-01' # or 'latest'
                region: "%env(AWS_REGION)%"
                use_aws_shared_config_files: false
                credentials:
                    key: "%env(AWS_ACCESS_KEY)%"
                    secret: "%env(AWS_SECRET_KEY)%"
```

Then we configure our two storage adapters (acme.storage.aws and acme.storage.local) and one lazy adapter (acme.storage) to which everything else will be referring. Still, it refers back to one of the two previously defined adapters. This enables us to toggle between the 2 options with only a change to an environment variable. The naming of these doesn't matter; I named mine to keep it obvious that they are related. (See Listing 5)

And finally we configure our Vich Uploader package and tell it to use Flysystem for the storage and define mappings for the users entities by telling it which flysystem storage adapter to use and how to name the uploaded files. To make sure to avoid naming collisions, I use the unique id namer. (See Listing 6)

Listing 5.

```
# config/packages/flysystem.yaml
flysystem:
  storages:
    acme.storage.local:
      adapter: 'local'
      options:
        directory: '%kernel.project_dir%/.../storage/'
    acme.storage.aws:
      adapter: 'aws'
        # Make file publicly accessible in S3
      visibility: public
      options:
          # The service ID of the Aws\\S3\\S3Client
          # instance defined in services.yaml.
          client: 'acme.s3_client'
          bucket: '%env(AWS_STORAGE_BUCKET)%'
          # Optional path prefix
          #   you can set empty string
          prefix: '%env(APP_ENV)%/uploads/storage/'
          streamReads: true
    acme.storage:
      adapter: 'lazy'
      options:
          source: '%env(FLYSYSTEM_ADAPTER_STORAGE)%'
```

Image Uploads

In our case, the file uploads will be profile pictures for the user account. So, in our Symfony Form builder type for the users, we add a field for the image uploads. (See Listing 7)

There is good documentation for the Vich Uploader[7] found at the GitHub repo for the project. Essentially, the magic happens in our controller, which processes the form submission. (See Listing 8)

When setImageFile is called on the user entity, the Vich Uploader package puts our file where it belongs. This happens because the $imageFile entity property in our User entity has the Vich attribute.

```
// NOTE: This is not a mapped field of entity metadata,
// just a simple property.
#[Vich\UploadableField(
  mapping: 'users',
  fileNameProperty: 'imageName',
  size: 'imageSize'
)]
private ?File $imageFile = null;
```

The `mapping: 'users'` field refers to the mapping we defined in vich_uploader.yaml, which in turn refers to flysystem.

When a new image is uploaded to the entity to replace the existing one, Flysystem deletes the original from the storage. And because we have `'allow_delete'` set to true in our Form field definition, a checkbox appears below the file browser input, which can be used to set the image to null and deletes it from the file storage.

7 https://phpa.me/github-dustin

Listing 6.

```
# config/packages/vich_uploader.yaml
vich_uploader:
  db_driver: orm
  storage: flysystem
  mappings:
    users:
      # Defined in flysystem.yaml
      upload_destination: acme.storage
      namer: Vich\UploaderBundle\Naming\UniqidNamer
```

Listing 7.

```
$msg = 'Please upload a valid image (jpg, png, gif).';
$builder
      // ...
      ->add('imageFile', VichFileType::class, [
          'label' => 'Profile Picture',
          'label_attr' => ['class' => 'form-label'],
          'required' => false,
          'allow_delete' => true,
          'download_uri' => false,
          'constraints' => [
              new File([
                  'maxSize' => '500k',
                  'mimeTypes' => [
                      'image/jpeg',
                      'image/gif',
                      'image/png',
                  ],
                  'mimeTypesMessage' => $msg,
              ])
          ],
      ]);
```

Listing 8.

```
// src/Controller/UserController.php
public function editUser(?int $id, Request $request)
{
  $user = $this->userRepository->findOneBy([
    'id' => $id
  ]);
  if (is_null($user)) {
    $user = new User();
  }

  $form = $this->createForm(
    UserFormType::class,
    $user
  );
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    $imageFile = $form->get('imageFile')->getData();
    $user->setImageFile($imageFile);

    // ... save the entity.
  }
  // ... Render the template with the form.
}
```

Listing 9.

```php
<?php
// src/Twig/AppExtension.php

/**
 * Provides a new Twig filter (flysystem_asset) which
 * uses the flysystem adapter to generate public URLs.
 */

namespace App\Twig;

use League\Flysystem\FilesystemOperator;
use League\Flysystem\UnableToGeneratePublicUrl;
use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class AppExtension extends AbstractExtension
{
  private FilesystemOperator $acmeStorage;

  public function __construct(
    FilesystemOperator $acmeStorage
  )
  {
    $this->acmeStorage = $acmeStorage;
  }

  public function getFilters(): array
  {
    return [
      new TwigFilter('flysystem_asset', [
        $this,
        'flysystemAsset',
      ]),
    ];
  }

  // Use the storage adapter to generate
  //   the public url.
  public function flysystemAsset(
    string $imgName
  ): string
  {
    if ( ! is_null($imgName) && $imgName) {
      try {
        return $this->acmeStorage->publicUrl($imgName);
      } catch (UnableToGeneratePublicUrl $e) {
        // Filesystem adapter doesn't generate
        //    public urls. Do it manually.
        return '/uploads/storage/' . $imgName;
      }
    }

    return '';
  }
}
```

If this feels a bit like magic to you, you aren't alone. The key is to know that these tools are calling each other automatically. Our User entity is configured to use VichUploader when an image is set on the entity. VichUploader then uses Flysystem as a filesystem abstraction, which uses the AWS SDK for storing the file if the `acme.storage.aws` storage is enabled, or the local filesystem otherwise. As a result, it all just happens for us when `setImage(...)` is called. Awesome!

Displaying Images in Twig

The final problem we have to solve is that the value stored in the database in our Users table is just a file name like 345345j643k324k.jpg. How do we turn that into a public URL, and how can it be an AWS URL if we use the AWS adapter or a local URL if we use local storage? The solution I came up with was to create a custom Twig filter that will generate the url based on which adapter is used. (See Listing 9)

Unfortunately, the publicUrl method currently does not exist for a local filesystem storage adapter. So, we have to catch the exception that is thrown, then manually create the url path. In reality, we can move this hard-coded path to a config variable in the container.

Then we can display our image with our new twig filter like so:

```
<img src="{{ user.imageName | flysystem_asset }}"
        alt="Profile Picture" />
```

Expanding to Multiple Storages

Since user profile pictures might not be the only type of file uploads we need to store, my loosely defined plan is to build a FlysystemService class that can store a collection of flysystem adapters so we can refer to them by a key. This class is my first draft. It also provides some helpful methods to list a storage's contents and to delete something from the storage directly. (See Listing 10)

Digital Ocean Spaces

Digital Ocean has a comparable service called Spaces that is compatible with the AWS S3 API. This means that if we are already comfortable with using Digital Ocean products like I am, we can use this instead. The process is similar; it just needs configuration with the Digital Ocean settings:

```
# .env.dev.local
DO_ACCESS_KEY="ABC123"
DO_SECRET_KEY="abc123xyz"
DO_REGION="nyc3"
DO_STORAGE_BUCKET="mybucket"
```

The configuration of the DO client is similar; however, we must configure the url (used for generating the public URL of the asset) and the endpoint, which is used for the API calls. (See Listing 11)

Listing 10.

```php
<?php
/**
 * Wrapper for the FlySystem adapters.
 * <https://flysystem.thephpleague.com/docs/>
 */

namespace App\Service\Utility;

use League\Flysystem\FilesystemOperator;
use League\Flysystem\StorageAttributes;

class FlysystemService
{
  private array $storages;

  public function __construct(
    FilesystemOperator $acmeStorage
  ) {
    $this->storages['user_profile_pictures'] =
                  $acmeStorage;
  }

  public function getStorage(string $key)
  {
    return $this->storages[$key];
  }

  public function listContents(string $key)
  {
    $allPaths = $this->storages[$key]
      ->listContents('/')
      ->filter(fn(StorageAttributes $attributes) =>
              $attributes->isFile())
      ->map(fn(StorageAttributes $attributes) =>
              $attributes->path())
      ->toArray();
    return $allPaths;
  }

  public function delete(string $key, string $filename)
  {
    $this->storages[$key]->delete($filename);
  }
}
```

The flysystem configuration simply adds another storage adapter and the type is still 'aws'. (See Listing 12)

Conclusion

During my research, there was *very* little documentation on how to do all of this. I pieced most of what I did from the library documentation, some StackOverflow answers, and some SymfonyCasts video transcripts that used the Oneup Flysystem Bundle. It's unclear to me how the Oneup bundle differs from what I did or what it adds to the situation, but it's what SymfonyCasts used, so it may be worth a deeper look. Ultimately I opted to use Flysystem and Vich Uploader directly.

Listing 11.

```yaml
# config/services.yaml
services:
  acme.do_client:
    class: Aws\\S3\\S3Client
    arguments:
      -
        version: 'latest'
        region: "%env(DO_REGION)%"
        credentials:
          key: "%env(DO_ACCESS_KEY)%"
          secret: "%env(DO_SECRET_KEY)%"
        url: "https://%env(DO_STORAGE_BUCKET)%. \
           %env(DO_REGION)%.digitaloceanspaces.com"
        endpoint: "https://%env(DO_REGION)%. \
           digitaloceanspaces.com"
```

Listing 12.

```yaml
# config/packages/flysystem.yaml
acme.storage.digitalocean:
      adapter: 'aws'
      # Make the uploaded file publicly accessible
      visibility: public
      options:
          # The service ID of the Aws\\S3\\S3Client
          # instance defined in services.yaml.
          client: 'acme.do_client'
          bucket: '%env(DO_STUDENTS_BUCKET)%'
          # Optional path prefix,
          #      you can set empty string
          prefix: '%env(APP_ENV)%/uploads/storage/'
          streamReads: true
```

It took a bit of a journey to get here, but I've successfully stored files in an S3 bucket (and in a Digital Ocean space) for the first time. Yay! 🎉 Using Symfony and the excellent libraries available made this simpler than I expected. With the Vich Uploader and Flysystem working together, there was very little in my own application code that had to deal with AWS or even Flysystem. Hence, the storage method used is transparent and able to be swapped. If you've been hesitating to make the leap to cloud object storage, jump in—this article will get you started in Symfony.

Sherri is a web software developer and business professional with over 15 years of experience working with PHP. She currently operates her company Avinus Web Services, and for fun she creates games and open source software. When not at the computer you'll find her camping, swimming and making forts with her husband and three sons in Ontario, Canada. Connect with her on Twitter. @SyntaxSeed

php[architect] [consulting]

php[architect]

Get customized solutions for your business needs

Leverage the expertise of experienced PHP developers

Create a dedicated team or augment your existing team

Improve the performance and scalability of your web applications

Building cutting-edge solutions using today's development patterns and best practices

consulting@phparch.com

Exploring PHP 8's JIT Compiler: Performance Boosts and Limitations

Rahul Kumar

PHP 8.0 ushers in an exciting new development for PHP developers: Just-In-Time (JIT) compilation. This means that your code can now be compiled directly into machine code—providing a huge performance boost to your applications.

JIT compilation allows you to compile your code into optimized machine code that runs natively on the CPU. This increases performance by eliminating the overhead of interpretation and allowing for more efficient memory management.

With JIT, the framework will compile parts of the code when they are needed and cache them so that subsequent requests don't need to be recompiled.

Not all of your application needs to be compiled, so the cost of JIT is quite low—and it can yield impressive results.

What is a Jit Compiler in Php8?

So what exactly is a JIT (Just-in-Time) Compiler? In its simplest form, it's a technique that enables you to compile a program into machine code immediately before it is executed.

For PHP 8, this means that instead of the traditional approach of converting source code into an intermediary language, then pre-compiling it into machine code, the JIT Compiler in PHP 8 will convert the intermediary language to machine code on-the-fly. This has the potential to speed up performance dramatically.

For example, with JIT Compilation, functions are compiled into binary instructions just once and stored in memory; subsequent calls to the same function will be much faster due to their ready availability in memory.

Additionally, JIT Compilation can optimize loops by eliminating redundant calculations and detecting patterns so execution time is reduced significantly.

In short, what we have here with JIT Compilation in PHP 8 is an incredibly efficient way of optimizing **script execution** and **providing huge performance boosts**—which may even reach up to double the speed in certain cases!

How Does Php 8's Jit Compiler Work?

OK, so you know that the JIT compiler is an awesome way to get performance boosts in code execution, but how does it actually work? First, it's helpful to understand that JIT compilation takes place at runtime–so while the code is being executed, the JIT compiler is optimizing and compiling the code quickly.

To get specific, there are three different optimization techniques that PHP 8 implements with its JIT compiler:

- **Inline caching** – this process borrows from a data structure called a "call" cache, which records recent parameters and returns values of a function call. This helps determine whether or not it can pull from the cache or if executing the code again is necessary.
- **Guarded optimizations** – this entails skipping redundant operations within a single function call that have already been done previously. This helps speed up code execution by not repeating operations and ensuring routines run faster.
- **Optimistic type speculation** – this process makes guesses about a given expression and then confirms them after the fact through type analysis and guards that fail when type assumptions are wrong.

By utilizing these different optimization techniques, PHP 8's JIT compiler speeds up code execution significantly–so much so that it can provide performance improvements between 3x-10x faster than traditional methods!

Pretty impressive stuff when you consider the power behind it all!

Let's see what happens under the hood.

When PHP 8's JIT compiler is enabled, it analyzes the PHP code being executed and identifies sections of code that are executed repeatedly. These sections are then compiled into machine code, which can be executed much faster than the original PHP code.

The compiled machine code is then stored in memory so that it can be reused the next time that section of code is executed. This means subsequent executions of the same code section can be performed much faster than the initial execution.

In addition to the three optimization techniques I mentioned above, PHP's JIT compiler also uses **profiling** to identify which sections of code are executed most frequently. This helps the JIT compiler prioritize which sections of code to optimize first, which can further improve performance.

Note that not all PHP code will benefit equally from the JIT compiler. For example, code heavily dependent on **I/O operations** may not see as much of a performance boost.

However, for CPU-bound tasks like image processing or mathematical operations, the JIT compiler can provide significant performance improvements.

Which Php Functions and Features Work with the Jit?

So now that you know how the JIT compiler works under the hood in PHP 8, you're probably wondering which parts of the language actually benefit from the performance boost. The good news is a large portion of PHP's functionality is JIT-compatible and will run faster in PHP 8.

Built-in functions

Many of PHP's built-in functions have been optimized to work with the JIT compiler. This includes string functions like strlen(), str_replace(), and substr(), numeric functions such as abs() and round(), and array functions like count(), sort(), and array_merge().

Loops and conditionals

The JIT also speeds up the execution of loops and conditional logic in your code. For loops, while loops, if/else statements, and switch/case statements will all receive a performance boost. This means any repetitive logic or complex decision-making in your code will benefit.

Object-oriented code

PHP's object-oriented features are JIT-compatible, including classes, inheritance, traits, interfaces, and magic methods. Method calls, instantiation, and property access will be faster. The JIT can also optimize away unused properties and methods, giving OOP code an additional speedup.

Exceptions

Throwing and catching exceptions in your code will run faster thanks to the JIT compiler. The engine can remove unused exception handlers and perform additional optimizations on the stack unwinding process that occurs during an exception.

What's not compatible?

Unfortunately, not all PHP functionality is JIT-compatible at this time. This includes:

- Variable variables ($$var)
- Extract()
- Compact()
- Serialization
- Dynamic calls (call_user_func())
- Reflection
- And more…

The PHP team is continuously working to expand JIT coverage to more areas of the language. With each minor release of PHP 8, check the changelog to see what new JIT optimizations and compatibilities have been added.

While the JIT compiler provides huge performance gains for much of PHP's functionality, it's important to know its current limitations. When writing high-performance code, you'll want to avoid incompatible features when possible and keep an eye on updates to see when they become JIT-ready.

When the Jit Compiler Does (and Doesn't) Help

PHP 8 introduced a Just-In-Time (JIT) compiler to boost performance for CPU-intensive tasks. The JIT compiles PHP code into machine code, allowing it to run faster. When enabled, the JIT can provide significant speed improvements for some types of applications.

The JIT excels at optimizing mathematical, looping, and recursion-heavy code. It stores the compiled machine code in memory, so it can skip the compilation step the next time that part of the code is executed. This results in major performance gains for algorithms, data processing tasks, and other non-trivial logic.

However, the JIT may not benefit non-blocking (asynchronous) applications as much. The JIT focuses on optimizing synchronous PHP logic. Asynchronous code that spends a lot of time waiting on external services won't see huge improvements. The time spent waiting outweighs any gains from faster PHP execution.

The performance boost you get from the JIT also depends a lot on your code and infrastructure. Well-optimized code that already runs efficiently may only see minor improvements.

The JIT can also be limited by factors like:

- Available server memory: The JIT requires more memory to store compiled machine code.
- Opcache settings: More aggressive opcache settings reduce the

need for re-compilation, limiting JIT benefits.

- CPU limitations: Underpowered servers won't be able to take full advantage of the JIT's optimizations.

While enabling the JIT can provide up to a 90% performance increase for some code, the real-world gains for most applications are more modest.

For complex mathematical and algorithmic code, you may see speedups of 30-50% or more. For basic CMSs, e-commerce sites, and API backends, the improvement is often 10-30%.

The JIT is a welcome addition to PHP 8, but it really shines when used for the types of workloads it's designed to optimize.

The JIT will provide a performance boost for most standard web applications but likely won't revolutionize your infrastructure needs. The improvements are most dramatic for specialized use cases.

How Php 8's Jit Compares to Other Languages

So how does PHP 8's JIT compiler stack up against other languages? Let's compare it to some major players.

Python has had a JIT compiler for a while now called PyPy. PyPy can provide 3-10 times performance improvements over the standard CPython interpreter. While PHP 8's JIT is still new, early benchmarks show it provides around 2-4 times better performance than the standard PHP interpreter.

So PHP is gaining ground but still has a way to go to match PyPy's speeds.

JavaScript's JIT compilers in browsers like Chrome's V8 and Firefox's SpiderMonkey have optimized JS performance for years. They can make JS run an order of magnitude faster than without a JIT.

PHP's JIT is still quite new, so while it provides solid performance boosts, JS JITs currently have a significant performance edge.

Java has had JIT compilation since its inception with the HotSpot JVM. Java's JIT can provide over ten times better performance than interpreting bytecode. Again, while PHP 8's JIT is promising, Java still has a major performance advantage thanks to over 25 years of JIT optimization.

Other languages like C# (.NET), Ruby (JRuby), and Lua (LuaJIT) also have mature JIT compilers that generally outperform PHP's new JIT. However, PHP's JIT is a big step forward and an exciting new addition to the language.

Over time, as more optimizations are added, the performance gap between PHP's JIT and other languages will narrow. But PHP likely won't match the performance of lower-level languages like C/C++ that compile to native machine code. PHP is a dynamically typed scripting language with certain performance limitations.

That said, the performance boost from PHP 8's JIT will be more than enough for many web applications and APIs. And combining it with other PHP 8 features like typed properties, match expressions, and constructor property promotion, PHP 8 becomes a very compelling upgrade for any PHP project.

The future is bright for PHP performance. Even though PHP's JIT isn't quite on par with other languages yet, the fact it now exists at all is a big win for the PHP community. Over the next few years, we'll see it continue to improve and further speed up our PHP applications.

Limitations of Php 8's Jit Compiler

The power of the JIT Compiler in PHP 8 should not be underestimated—it promises some of the most significant performance gains ever seen when optimizing code. However, even with this newfound power, there are still some limitations that you should consider when it comes to using the JIT compiler.

First and foremost, code complexity plays a significant role—the more complex the code is, the less likely it is to benefit from JIT compilation. Additionally, certain kinds of computations—such as those involving randomness, compression algorithms, or shell commands—may also not benefit from JIT optimization either.

Another limitation of the JIT compiler is that it can only be used on PHP scripts that are run multiple times, not on one-off scripts. This means that if your code is only run once, then JIT compilation won't do anything for you.

It's also worth noting that the JIT compiler doesn't optimize all code equally. Some code can be optimized more than others, and some optimizations may be less effective than others. As a result, it's important to thoroughly test your code before deploying it to ensure that it's efficient and performs as expected.

Finally, consider that if you choose to optimize your code for the JIT compiler, you may have to sacrifice other important components, such as readability or even maintainability of the code, to achieve performance optimization.

When optimizing for PHP 8's JIT compiler, it's important to weigh these considerations and make sure that they are consistent with your overall project goals and objectives.

Best Practices for Using Php 8's Jit Compiler

Now that you understand the basics of JIT Compiler, you're probably wondering what are the best practices for making use of it.

Here's a quick list of recommendations for optimizing performance with PHP 8's JIT:

Enable OPCache

The first step is to make sure OPCache is enabled on your server, as this allows JIT to work in the first place. You can enable OPCache by adding some lines of code to your php.ini file.

1. Locate your php.ini file: Its location can vary depending on your server configuration, but it's usually located in /etc/php or /usr/local/lib/php. You can use the command "php –ini" to find

the location of your php.ini file. After this, open the php.ini file.

2. Find the OPCache settings: In your php.ini file, you'll need to find the settings for OPCache. These will be in the [opcache] section. If you can't find the [opcache] section, you can add it to the bottom of your php.ini file.
3. Enable OPCache: To enable OPCache, you'll need to set the following settings:

```
opcache.enable=1
opcache.jit_buffer_size=100M
opcache.jit=1235
```

4. The `opcache.enable` setting turns on OPCache, while the `opcache.jit_buffer_size` setting sets the size of the JIT buffer (in megabytes). The opcache.jit in the setting tells OPCache to use the Zend Optimizer Plus JIT engine, which is currently the only JIT engine supported by OPCache.
5. Note: the opcache.jit value (1235) could change as updates to PHP are made.
6. Save your php.ini file
7. Restart your web server: To see the changes, you would have to restart your web server. You can usually do this with the command sudo service apache2 restart (or whatever command is appropriate for your web server).

That's it! OPCache should now be enabled on your server, allowing JIT to work properly.

Using Strict Typing

Using PHP 8, you can specify the data types for function arguments and return values. This will help JIT to optimize the code more effectively, resulting in better performance. You can use the "declare(strict_types=1);" statement to enable strict typing.

```
declare(strict_types=1);

function add(int $x, int $y): int {
    return $x + $y;
}

echo add(10, 20);
```

Here we declared that the "add" function takes two integer arguments and returns an integer value. This helps JIT to optimize the code better, resulting in faster execution.

Checking Memory Usage and Benchmarking

With JIT also comes an increase in memory usage, so make sure that your server has enough memory available to accommodate it. Additionally, check your memory usage regularly and monitor how much of it is being used by the JIT compiler.

In general, the best way to determine whether or not PHP 8's JIT compiler will benefit you is by testing it out with real-world scenarios and benchmarks — that way, you'll get a clear understanding of how your application will perform before implementing it in production.

Also, You can use tools like Xdebug or Blackfire to profile your code and identify performance bottlenecks.

Listing 1 shows an example of profiling a PHP script using Xdebug.

Listing 1.

```
1. <?php
2.
3. // enable profiling
4. xdebug_start_profiling();
5.
6. // your PHP code here
7. // stop profiling
8. xdebug_stop_profiling();
9.
10. // output profiling results
11. echo '<pre>';
12. echo file_get_contents(xdebug_get_profiler_filename());
13. echo '</pre>';
```

Here we used the "xdebug_start_profiling" and "xdebug_stop_profiling" functions to enable and disable profiling, respectively. We then output the profiling results using the "xdebug_get_profiler_filename" function.

This allows us to identify which parts of our code are slow and can benefit from JIT optimization.

Avoid Excessive String Concatenation

String concatenation can also be expensive in PHP.It can add up quickly if you're doing a lot of string concatenation in a loop.

```
// Instead of this
$result = '';
foreach ($myArray as $item) {
    $result .= $item;
}

// You can do this
$result = implode('', $myArray);
```

Optimize the Critical Path

The critical path refers to the parts of your code that take the longest to execute. By optimizing the critical path, you can achieve the most significant performance gains. Therefore, it's important to identify your codebase's critical path and optimize it using JIT.

Reduce Array Allocations

In PHP, creating new arrays can be a relatively expensive operation. If you're creating arrays frequently in your code, consider ways to reduce the number of allocations.

Listing 2 on the next page shows you how to create a new array in a more optimized way, reducing the overhead of array allocations.

Listing 2.

```
1. //Instead of this
2.
3. $result = array();
4.
5. foreach ($myArray as $item) {
6.     $result[] = $item * 2;
7. }
8.
9. //do this
10.
11. $result = array_map(function($item) {
12.     return $item * 2;
13. }, $myArray);
```

Future Developments and Community Feedback

The JIT compiler translates PHP code into machine code, allowing it to run faster. While the JIT compiler shows a lot of promise, it also has some limitations and areas for improvement. The future of the JIT compiler depends on overcoming these challenges and getting feedback from the PHP Community.

Optimizing the JIT Compiler

The JIT compiler still has room for optimization to maximize performance gains. Some possible optimizations include:

- Improving functions inlining by analyzing call graphs to determine which functions should be inlined. This can avoid the overhead of function calls.
- Implementing escape analysis to allocate objects on the stack instead of the heap when possible. This reduces memory usage and improves cache locality.
- Adding support for tracing JITs which can optimize across function boundaries. A tracing JIT records hot traces (frequently executed paths) and optimizes them.

Expanding JIT Coverage

The JIT compiler currently only supports a subset of PHP functionality. To realize the full potential of the JIT, its coverage needs to expand to include:

- Full support for object-oriented features like classes, interfaces, traits, etc.
- Integration with popular PHP extensions like MySQLi, GD, Curl, etc.
- Support for eval(), include/require, and variable variables (${$$var})
- Compatibility with a larger range of PHP web frameworks and content management systems

Community Feedback and Contributions

The success of the JIT compiler depends on the PHP community. The community can provide feedback on performance, report any issues or limitations, and even contribute code to help improve the JIT. Some ways the community can help include:

- Reporting performance issues or regressions on real-world apps and frameworks. This helps the developers optimize the JIT for common use cases.
- Filing bug reports on any limitations, crashes, or incorrect behavior. The more issues that are identified, the more robust the JIT can become.
- Contributing code to expand functionality, add optimizations, or fix issues. Community contributions will be key to improving the JIT.
- Providing general feedback on the JIT and its impact. The developers need to know if the JIT is achieving its goal of substantially improving PHP performance.

The JIT compiler is an exciting new feature in PHP 8 that shows a lot of promise for boosting performance. But continued optimization, expanded coverage, and community support will be needed to realize its potential fully. The future of the JIT depends on overcoming its current limitations with the help of the PHP community. Overall, the JIT is a big step forward for PHP performance.

Wrapping up

In conclusion, PHP 8's JIT compiler offers exciting possibilities for professional developers looking to boost the performance of their applications. You can achieve significant performance gains by understanding how the JIT compiler works and adopting best practices for coding with the JIT compiler in mind.

While the JIT compiler is not a one-size-fits-all solution, and there are potential limitations and trade-offs to consider, it is still an important tool in the developer's toolkit.

With examples of JIT compiler optimizations, tips for debugging and troubleshooting, and insights into the future direction of the JIT compiler, developers can continue exploring the potential of PHP 8's JIT compiler in their own projects.

Rahul is an 18-year-old technical writer and developer. He has a deep passion for writing, coding, and everything in between. Over the past few years, he has written over 500 articles for developers, sharing his knowledge and expertise with others. Currently, he is working on two exciting projects. The first is Fueler.io[1], a platform that allows users to quickly and easily find the best gas prices in their area. The second is Learnn.cc[2], an online education platform that helps people learn new skills and advance their careers. @rahuldotbiz

1 *Fueler.io: http://fueler.io*

2 *Learnn.cc: http://learnn.cc*

Serverless PHP with Bref

Chris Tankersley

Most PHP applications should run inside a serverless application without any problems, and by using Bref, we get access to the most common extensions. If you need something special, you can compile your own additional layers for AWS to access.

Serverless computing is becoming all the rage for developers. As discussed in the previous article, many developers have moved on to a deployment system where the "where" of serving an application is being handled—they handed almost all that decision-making off to their hosting platform. They bundle their code up, upload it to their provider, and the provider takes it from there.

This is the heart of Serverless computing. The provider takes your code and decides how it is deployed, the available environments, and even how to scale your application. As a developer, you do not worry about this and pay a hosting bill at the end of the month, which is usually determined by how long your application has run.

This was an odd concept for PHP developers for a long time. We were already used to just uploading code to a server and having it run, and this could be accomplished for around $5 a month on hundreds of providers. This was also an odd concept for Serverless hosts, as even to this day almost no one supports PHP as a serverless environment.

All of this is also not to be confused with "Platform as a Service," which is generally a containerized system that provides a traditional hosting environment with some sort of web server and an installation of PHP. Platforms as a service, while similar to serverless computing, offer more options for how the environment for your code is configured compared to serverless computing.

PHP developers have few options for serverless computing compared to many other languages, and each provider offers PHP support at various levels. At the time of this article, your options are:

1. Amazon Web Services through their "Custom Runtime API"[1] (simplified via Bref[2]
2. Digital Ocean[3] through their "Functions"[4] product
3. Apache OpenWhisk[5], which powers Digital Ocean Functions.
4. Appwrite[6] through their "Functions"[7] product
5. Microsoft Azure through custom handlers[8] for their "Azure Functions"[9] service
6. Google Cloud Functions[10]

AWS and Google are the easiest and most full-featured platforms, as they can be configured to full applications and not just "functions."

What is a "function?"

A few times, I have used the term "Function," especially regarding products. As you climb down the rabbit hole of serverless computing, you will come across this term multiple times, and each provider may have a different definition.

The most basic definition of a "function" is code you upload to a provider to be executed on demand. Where this gets confusing is how different providers expect your code to work. To help make things clear, we will use two terms throughout this article, "Function" and "Application," to differentiate between how code is structured and executed.

A "Function" will be a PHP script comprising a single function that is either defined or returned. This function will generally take in a request and either return a response object or echo out a response directly, but the idea is that you write a single function. Your function can call other code, including code installed via Composer, but you execute a function: (See Listing 1)

Listing 1.

```
1. // Sample "Function" serverless code
2.
3. <?php
4.
5. return function($request) {
6.
7.     return [
8.         'status' => 200,
9.         'body' => [
10.            'message' => 'Hello World',
11.        ]
12.    ];
13. }
```

1 https://phpa.me/compute-serverless-lamp
2 https://bref.sh/
3 https://www.digitalocean.com
4 https://phpa.me/docs-digitalocean
5 https://openwhisk.apache.org
6 https://appwrite.io/
7 https://appwrite.io/docs/client/functions
8 https://phpa.me/digitalocean
9 https://phpa.me/learn-microsoft
10 https://phpa.me/azure-microsoft

An "Application" is structured like a traditional PHP application. Your function will execute a script, like `index.php`, which contains the standard bootstrapping code that also sets up non-serverless applications. If you use a framework like Laravel or Symfony, your code will run without modification.

Which code you can execute is up to your provider. OpenWhisk, Digital Ocean, and Appwrite use the "Function" approach, and AWS, Azure, and Google will support both. Neither approach is necessarily better than the other except for how you want to write your code.

To make things easier, we will be using Bref and the Serverless Framework[11] to deploy an application to AWS's Lamba service.

Why Deploy to Serverless Platforms?

The most significant draw for deploying to a serverless platform is the cost. Most providers charge you only for the runtime of your application serving requests, not an arbitrary "uptime" like a monthly cost. For example, deploying to Digital Ocean requires you to purchase a Virtual Private Server, which is billed at $0.00595 per hour. Since most VPS are left on all month, you pay for the entire month (roughly $4.00) even if you did not have any traffic.

Serverless platforms instead charge for execution time and the number of requests. Digital Ocean charges $0.0000185 per gibibit-second (basically, how long did you keep the CPU occupied for a request times the amount of memory used). That same $4 you spent on monthly hosting handles roughly 216,000 requests (assuming small memory usage and around 100ms response times). Most platforms also have a free tier, and in the case of Digital Ocean, you would not actually pay for any hosting as they offer the first 90,000 Gib-seconds for free (roughly 1 million requests).

This can be a great way to save money for many startups and new websites. As execution ramps up, however, watch out for skyrocketing bills. If you have a CPU or memory-intensive call, these can also drastically affect your billing. Serverless is meant for short, quick execution times.

Other than cost, Serverless has the benefit of handing off all the infrastructure problems to the provider. All the developer has to do is write and deploy the code; the provider handles everything else. Most PHP applications should run inside a serverless application without any problems, and by using Bref, we get access to the most common extensions[12]. If you need something special, you can compile your own additional layers for AWS to access.

Aws, the Serverless Framework, and Bref

Deploying to a serverless platform like AWS requires a few tools to make things easier. For PHP, we can use a combination of Bref, a set of extensions for AWS Lambda that enable PHP support, and the Serverless Framework, a tool that makes it easier to interface with a variety of serverless platforms. Bref itself is AWS-only. We will deploy our sample application to AWS.

If you do not have an account with AWS, you can sign up for one at https://aws.amazon.com/[13]. They offer a free tier which, at the time of writing, gives one million free requests and 40,000 GB-seconds (the amount of time your code runs, in seconds, times the amount of memory you use).

We then need to install the Serverless Framework CLI. It is supplied as an NPM package, so you will need Node.js[14] and NPM installed. We can install the serverless package for the CLI tool:

```
npm install -g serverless
```

You can make sure everything is installed by running `serverless --version` to see what version was installed. If you get an error like the command is not found, you may need to check your Node.js and NPM installation to ensure they are in your path, which is above and beyond this tutorial.

Now we get to venture into the deep and confusing world of AWS configuration. Since AWS is about running virtual infrastructure, they go to great lengths to ensure that only the proper users you expect to perform an action can perform that action. We will need to create a Policy, a Group, and a User that can upload code to AWS Lambda.

Create an Aws Policy

A Policy in AWS is a list of permissions that a user or a group has. Users, and members of groups, can only perform the actions that their associated Policies allow. This means you can have specific users for doing things like deploying code but restricting what servers they have access to or who can access various storage systems like S3 buckets.

We will create a new policy to ensure that only specific users can upload code. This policy will use the permissions suggested by the Serverless Framework and the Bref, and we will attach this policy to a group. While all of this can be done via the AWS command line tool, we will use the web UI to set everything up.

In the AWS UI, go to the IAM service and:

- Go to "Policies"
- Click on "Create Policy"
- Under "Specify Permissions," click "JSON."

Serverless Framework provides a gist that contains the minimum permissions for the Serverless CLI[15]. Open up that gist and copy the JSON that is there. We will paste that into the editor but do not click "Next" immediately. We need to add the following permission to the list: "logs:TagResource".

Once you add that additional permission, you can click "Next."

- Name the policy "serverless-cli"
- Scroll down and click "Create policy"

11 https://www.serverless.com

12 https://bref.sh/docs/environment/php.html

13 https://aws.amazon.com

14 https://nodejs.org/en

15 https://phpa.me/gist-github

After a few moments, AWS should list your new policy as available. We can now create a group that will abide by this policy.

Set Up an Aws Group and User

Now we need a group to attach the policy to. While still under the IAM service:

- Go to "User Groups"
- Click "Create Group"
- Name the group "serverless-cli"
- Scroll past "Attach Users" to "Attach permission policies - Optional"
- Select the "serverless-cli" policy we just created
- Click "Create group"

AWS will churn for a moment and then tell you that the group has been created. Now we can create a user that is part of the group. While still under the IAM service:

- Go to "Users"
- Click on "Add users"
- Give the new user the name "serverless-cli-user" and click "Next"- Leave "Provide user access to the AWS Management Console" as the user will not need to log into the AWS website
- Select "Add user to group," select the "serverless-cli" group, and click "Next"
- Click "Create User" to create the user

Once the user is created, you will be dropped back to the list of users on your account. The user has access to AWS Lambda and can push up code changes, but we need credentials for the Serverless Framework CLI to log into AWS. While we are still on the Users page:

- Find the new "serverless-cli-user"
- Click on the "Security Credentials" tab below the "Summary" section
- Scroll down to "Access Keys" and click on "Create Access Keys"
- In the options for best practices, select "Command Line Interface"
- Down at the bottom, check "I understand the above recommendations"
- Click "Next"
- Skip adding a tag, and click "Create Access Key"
- Copy down the "Access Key" and "Secret access key" values

Please keep track of these credentials, as we will need them to configure Serverless Framework, but do not lose them! You cannot retrieve the secret value after leaving this page, so if you lose either the key or secret, you will need to remove and regenerate the access keys.

"I can't get past the 'Best Practices' page for credentials!" If you try and select the CLI option and then select the "I understand" box," sometimes when you click "Next," it will deselect the "I understand" confirmation box. This seems to be due to something in various ad blockers, so you may need to temporarily disable any ad blockers in your browser.

Set Up the Serverless Cli

We are in the home stretch. We can configure the Serverless Framework CLI tool using the credentials we just made. Use the following command, substituting your access key and access secret:

```
serverless config credentials --provider aws \
  --key {key} --secret {secret}
```

If everything works, you should see the following:

```
✓ Profile "default" has been configured
```

With our easy-to-follow 29-step process, we are ready to deploy serverless code! While I am being a bit sarcastic, most of this is due to AWS's security policies, and at worst, in the future, you need to create new users. This policy and group setup allows you to be quicker in the future by just dropping users into the serverless-cli group and granting permission to things like continuous integration pipelines or other users who need to test our serverless functions or applications.

Creating a Serverless Application

With the serverless command now configured, we can use Bref to set up an application that can be deployed to AWS. Despite Bref being a Composer package, it is actually a JavaScript application that talks to AWS Lambda to work with their custom runtime API. Our application never needs to know that Bref is there (unless you are using Symfony[16] or Laravel[17], there are bridges to help it work with the request/response system).

We do need to have it installed as a dependency, however. Let's create a new application and check out the demo it provides:

```
composer require bref/bref

bref init
```

When prompted, enter "0" to select "Web Application." Since web applications are designed to work like standard PHP applications, we will focus on that for this article. After a moment, you should have a shiny new index.php and a serverless.yaml file.

If we open up the index.php file, there is nothing special here. In fact, there is not even any PHP code. You could add some PHP code, but for now, we will leave it as the HTML code for the demo page.

16 https://bref.sh/docs/frameworks/symfony.html

17 https://bref.sh/docs/frameworks/laravel.html

Listing 2.

```
# serverless.yaml

service: app
provider:
name: aws
region: us-east-1
plugins:
    - ./vendor/bref/bref
functions:
api:
handler: index.php
description: ''
runtime: php-82-fpm
timeout: 28
events:
    - httpApi: '*'

# Exclude files from deployment

package:
patterns:
    - '!node_modules/**'
    - '!tests/**'
```

serverless.yaml is where the configuration for deployment lives. Bref creates the configuration file the serverless command will use to deploy our code. (See Listing 2)

This file denotes that we are deploying an application service and that we are deploying it to AWS. We also specify that we are using an additional plugin, Bref, and information about our application. We want our handler to be `index.php` and use the php-82-fpm runtime to execute our code. We also specify that it will trigger on an httpApi event (basically, put this application behind an HTTP call).

While Bref itself only supports AWS, the Serverless Framework supports a variety of hosts. We are just limited to AWS by the fact we are running PHP. In theory, as more hosts support PHP, Bref will expand this to support more services. If you are starting with an existing project, all you should need to do is adjust your PHP runtime and handler.

For now, we can use the existing setup. We can deploy it to AWS with a single command: "serverless deploy".

After a few moments, you should see the following:

```
Deploying app to stage dev (us-east-1)

✓ Service deployed to stack app-dev (60s)

endpoint: ANY--- https://....amazonaws.com
functions:
api: app-dev-api (255 kB)

Need a faster logging experience than CloudWatch? Try our
Dev Mode in Console: run "serverless dev"
```

If you do, congratulations! Your code should now be accessible from the URL that was generated. You should be able to now copy and paste the URL into your browser, and you should see the following default demo page show up:

Listing 3.

```
// index.php

<?php

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Slim\Factory\AppFactory;

require_once __DIR__ . '/vendor/autoload.php';

$app = AppFactory::create();

$app->get('/', function(
    requestInterface $request,
    ResponseInterface $response,
) {
    $response->getBody()->write(
        "This is the main route of a Slim application"
    );
    return $response;
});

$app->get('/hello', function(
RequestInterface $request,
ResponseInterface $response,
) {
    parse_str($request->getUri()->getQuery(), $args);
    $name = $args['name'] ?? 'world';
    $response->getBody()->write("Hello {$name}");
    return $response;
});

$app->run();
```

Running Php Code

This sample app works, but it could be more indicative of a real-world application. Let's look at a small application that uses a framework and some PSR components. I have set up a small application that uses the Slim framework[18] over at https://github.com/dragonmantank/bref-example[19] that you can check out that is all pre-configured. Clone that repository to your computer, and open it in a code editor. (See Listing 3)

This script has nothing Bref or serverless specific about it. It pulls in the AppFactory from Slim, creates a new application, registers two routes, and runs the application. This code will run the same on a bare-metal server, a container, or inside the AWS Lambda serverless platform.

Assuming you do not want to change the deployment region, you can install the dependencies with Composer install and then run serverless deploy to deploy this application to AWS

18 https://www.slimframework.com

19 https://github.com/dragonmantank/bref-example

Lambda. Once it's deployed, you can visit the URL that is returned. If you visit the base URL, you should see:

This is the main route of a Slim application.

You can also visit /hello and pass a query parameter of name={something} to get a "Hello World" response. For example, visiting

```
https://{id}.execute-api.{region}.amazonaws.com/hello?name=Chris
```

should return:

```
Hello Chris
```

As I noted before, if you are using Laravel or Symfony, you will need additional packages to help Bref make them aware of the incoming requests. This has more to do with their use of the Symfony HttpKernel and its lack of PSR-7 support out of the box, and the changes required are minimal. Just know that if you are using either of those frameworks, you will need to check the Bref documentation, and if you are not using a PSR-7-backed framework, you may need to make custom modifications to your code.

Get Rid of Your Servers

Besides the headache of setting up AWS, deploying your code using Bref is very close to deploying to a traditional platform. If your project is low traffic enough, serverless computing is a great way to save some cash. Since Bref handles cleanly wrapping PHP, almost any existing PHP application can be moved to serverless.

AWS and Bref are not the only contenders in town. Digital Ocean and Appwrite also natively support PHP for writing backend functions which work great with sites designed around Single Page Applications or ones that just need some dynamic server-side code to work with client-side JavaScript. If you want to create a more micro-service type application, these types of hosting work great to separate out the execution.

Take a look at serverless computing and see if you think it will work well for your application. While there is nothing wrong with traditional hosting options, seeing what else is available never hurts.

Related Reading

- *The New LAMP Stack is Serverless* by Benjamin Smith, March 2021. http://phpa.me/serverless-lamp-stack
- *Community Corner: A Bref of Fresh Air* by Eric Van Johnson, April 2021. https://phpa.me/community-apr-21

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank

Prisoner's Dilemma

Eric Mann

Every application must be designed, and the ethical consideration of that tool's use (or misuse) must be key to the technical design.

It was simultaneously the most depressing meeting we had and the most worthwhile use of team time we could have come up with. Rather than code, plan projects, or listen to presentations, we sat in a conference room filled with whiteboards. We took turns coming up with the worst possible use of our tool—a fully encrypted, anonymous data storage platform.

Would people use the tool to store their medical data? Sensitive personal legal documents? Who might leverage the strong encryption and anonymity to protect pornography? Would we be empowering terrorists to hide their plans from law enforcement?

It was a remarkably heavy discussion.

At the core, we all believed very deeply in the tool we were trying to build. Privacy. Security. User control of their data. These principles were paramount in what we hoped to accomplish. But the sticky problem of a truly secure, anonymous system was that we **would not know** what kind of data was being stored or who was storing it.

This meeting came on the heels of the UK Prime Minister publicly attacking end-to-end encryption tools[1] and alleging that open-source frameworks were aiding and empowering terrorist activity.

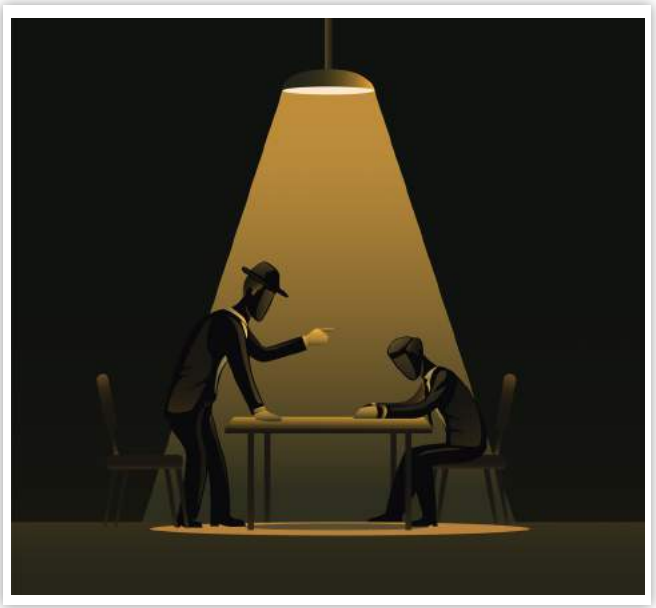
Could we build this tool and live with the chance that some unknown bad actor might use it for evil?

Ethics in Software

Software developers are primarily occupied with the "happy path" of development. Think through a customer use case and build the application to service that use case efficiently, effectively, and with smooth user interaction. It's when unintended or unexpected uses arise that we run into problems.

A user loads a website built for high-resolution monitors on a grayscale feature phone. Someone completes a web form with an unexpectedly-long, hyphenated surname. A PDF submitted for processing is 250 pages rather than the expected max of 12.

Each of these edge cases needs to be considered by the development team. It's important to ensure that any unexpected use of the application is well handled—failure to do so helps would-be attackers find ways to break or otherwise misuse the tool.



But it's not just edge cases about how the application works that are important. It's edge cases around how the application is `used`.

In our case, we were concerned about bad actors leveraging an encryption solution to obscure or protect illegal or dangerous activity. In your case, it might be concern over illicit sales through an e-commerce platform. Or perhaps malicious content hosted on a peer-to-peer sharing platform[2].

With the advent and growing popularity of generative AI like ChatGPT and Stable Diffusion, it's more critical than ever that every software engineer think carefully about how their tools will be used. Many of the most popular generative AI tools are already used in political misinformation campaigns. Others are merely causing waves due to a lack of user understanding, like the NY lawyer who leveraged ChatGPT for legal research[3] and submitted a brief citing previous cases invented entirely by AI.

Every application must be designed, and the ethical consideration of that tool's use (or misuse) must be key to the technical design.

It's not just a matter of how your tool will be misused; it's a question of the impact of that misuse. Would someone break a webpage heading? Transfer money into the wrong account? Lose a legal case? Die?

1 https://phpa.me/techcrunch

2 https://phpa.me/techcrunch-mozilla

3 https://phpa.me/bbc-news

Security in software development doesn't end at correcting the code within an application—it always extends to the ethics of its development and eventual use.

A Difficult Calculus

Our meeting years ago ended with a solid decision by the team. Yes, we could build the tool, and yes, we could live with the ramifications.

The decision was a difficult one to come by. We knew our tool would be abused and leveraged by the kinds of people and groups we loathed. But we also recognized two crucial truths:

1. Even if we never built or shipped our tool, someone else would. That someone else might be the bad actors themselves. Regardless, we would abdicate any control over or say in how the tool worked and the community it aimed to serve once we elected `not` to build it.
2. Our target customers had no other options and desperately needed this tool.

We knew who we were building the application for. We knew why they needed it. And their use case trumped the potential risk that someone other than our intended customer base might use the tool for nefarious purposes. Even if we chose `not` to build it, someone else would have inevitably done so.

Ethical conversations can be nuanced and lead to heated disagreements about approach. But they're critical to your product and systems' secure development and operation. Take time to threat model your application[4], but also take time to threat model its potential use in the world as well.

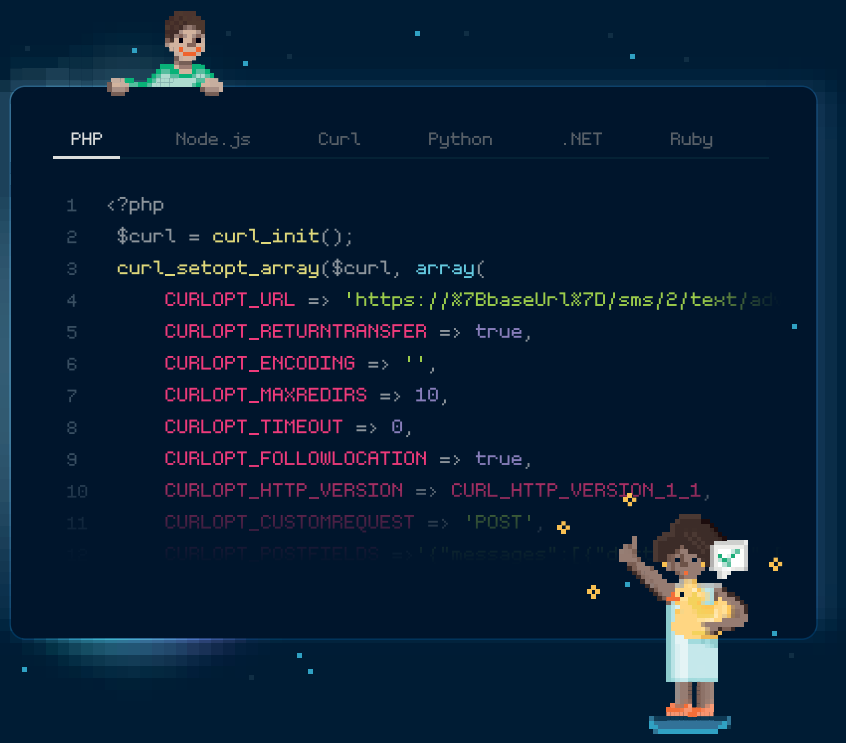
Related Reading

- *Security Corner: Types of Tokens* by Eric Mann, June 2023. https://phpa.me/security-jun-23
- *Security Corner: Tabletop: Planning for Disaster* by Eric Mann, May 2023. https://phpa.me/security-may-23

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

4 *threat model your application:* https://phpa.me/phparch

infobip



Infobip CPaaS speaks your language

> Integrate feature-rich CPaaS solutions with your tech stack

Integrate any communication channel or module by using a flexible and programmable API stack that supports PHP. Tap into the open-source SDKs designed to get you up and running fast—with just a few lines of code.

Sign-up and get started with Infobip APIs.

[Try for free]

PostgreSQL

Joe Ferguson

Last month we covered Databases as a Service and glossed over the fact that MySQL and PostgreSQL are the commonly found relational database management systems (DBMS) across providers. While MySQL is often the default for most PHP developers, this month, we will dive into PostgreSQL and get a PHP application up and running.

PostgreSQL or Postgres is an ACID[1]-compliant database management system designed to handle single-machine small request workloads to data warehouses scaled across systems and web services with high-traffic workloads. Atomicity, Consistency, Isolation, and Durability (ACID) are database transaction properties that guarantee data even during hardware failures or service outages. Postgres emerged from the University of California, Berkley, in the 1980s. After many iterations and changes, the project was renamed from POSTGRES to PostgreSQL in 1996 to denote it was fully compliant with the Structured Query Language(SQL)[2] .

Atomicity

Atomicity refers to the property of a database transaction that ensures that all of the transaction's operations are completed or none are. In other words, a transaction is an indivisible and irreducible series of database operations that must be treated as a single unit of work. If a transaction fails at any point, all of its changes must be rolled back, and the database must be restored to its previous state before the transaction began. This guarantees that the database is always in a consistent state, regardless of any errors that may occur during a transaction.

A practical example of atomicity in SQL can be illustrated using a simple SQL query. Consider the following:

```
BEGIN;
UPDATE accounts
    SET balance = balance - 100 WHERE id = 123;
INSERT INTO transactions (account_id, amount)
    VALUES (123, -100);
COMMIT;
```

This query begins a transaction by executing the `BEGIN` statement, which marks the start of a transaction. It then updates the balance of account 123 by subtracting 100 from the current balance using the `UPDATE` statement. The query then inserts a new transaction record into the `transactions` table using the `INSERT` statement, which records the details of the transaction. Finally, it commits the transaction by executing the `COMMIT` statement, which marks the end of the transaction.

If any part of this transaction fails due to an error, such as a hardware failure or a service outage, the entire transaction will be rolled back, and the database will be restored to its previous state before the transaction began. This ensures that the database is always in a known state, regardless of any errors that may occur during a transaction.

Another example would be a scenario where a user's password is being updated. If the transaction fails, we still want the user to be able to login with their old password, which will be reset when the transaction rolls back if something fails.

```
BEGIN;
UPDATE users SET
password = 'fOurty42Two!p455word!' WHERE user_id = 42;
COMMIT;
```

In summary, atomicity is a crucial property of transactions in SQL, which requires that either all of the operations are completed or none of them are. This ensures that the database is always in a consistent known-good state, regardless of any errors that may occur during a transaction, and helps to ensure the integrity of the data stored in the database. If any of these errors occur, they should surface in your application logs for further debugging.

Consistency

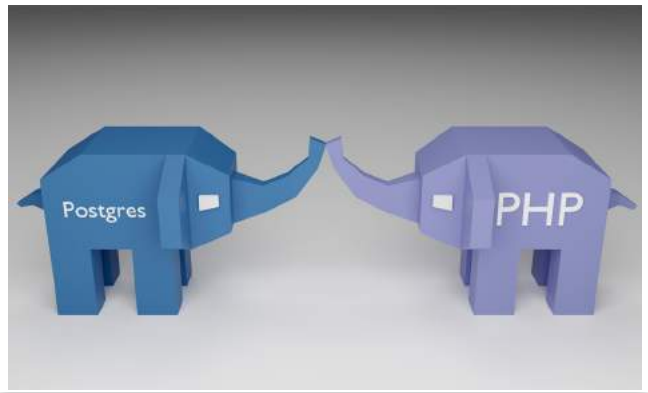
In PostgreSQL, ACID consistency means that the database ensures that any changes made by a transaction to the data conform to all defined rules, constraints, and relationships. This requires that the database is always in a known state, regardless of any errors that may occur during a transaction. If a transaction fails at any point, all of its changes must be rolled back, and the database must be restored to its previous state before the transaction began. This ensures the database stays consistent even during hardware failures or service outages.

A practical example of consistency in PostgreSQL can be illustrated using a simple SQL query. Consider the following SQL query:

```
BEGIN;
INSERT INTO employees (id, name, salary)
    VALUES (1, 'John Smith', 50000);
UPDATE departments
    SET budget = budget + 5000 WHERE name = 'IT';
COMMIT;
```

1 *ACID: https://en.wikipedia.org/wiki/ACID*

2 *SQL: https://en.wikipedia.org/wiki/SQL*



In this query, we begin a transaction by executing the `BEGIN` statement, which marks the start of a transaction. Next, we insert a new employee record into the `employees` table using the `INSERT` statement. Finally, we update the `budget` column of the `departments` table using the `UPDATE` statement, increasing it by 5000 for the department named 'IT'.

This query ensures consistency by conforming to the rules and constraints defined for the `employees` and `departments` tables. For example, the `id` column of the `employees` table may be defined as a primary key, which ensures that each employee record has a unique identifier. The `budget` column of the `departments` table may be defined as a non-negative value, which ensures that the department budget is always a positive number. If the transaction violates any of these rules or constraints, the entire transaction will be rolled back, and the database will be restored to its previous state before the transaction began. This ensures the database stays consistent even during hardware failures or service outages.

Isolation

In PostgreSQL, ACID isolation means that each transaction is executed as if it were the only transaction running on the database, and its changes are not visible to other transactions until it is committed. This ensures that concurrent transactions do not interfere with one another and that the database remains in a known or safe state even when multiple transactions are being executed simultaneously.

A practical example of isolation in PostgreSQL can be illustrated using a simple SQL query. Consider the following SQL query:

```
BEGIN;
UPDATE accounts
    SET balance = balance - 100.00 WHERE id = 123;
SELECT balance FROM accounts WHERE id = 123;
COMMIT;
```

This query begins a transaction by executing the `BEGIN` statement, which marks the start of a transaction. It then updates the balance of account 123 by subtracting 100.00 from the current balance using the `UPDATE` statement. The query then selects the updated balance from the `accounts` table using the `SELECT` statement. Finally, it commits the transaction by executing the `COMMIT` statement, which marks the end of the transaction.

If another transaction executes concurrently and updates the balance of account 123, PostgreSQL will ensure that the two transactions do not interfere. Each transaction will be executed in isolation from the other, and its changes will not be visible to the other transaction until it is committed. This ensures that the database remains in a safe state, even when multiple transactions are being executed simultaneously.

Durability

In the context of ACID, durability refers to the property of a database transaction that ensures that once a transaction has been committed, its changes will persist even in the event of a system failure. This guarantees that data is not lost due to hardware or software failures, power outages, or other system failures.

In PostgreSQL, ACID durability means that once a transaction has been committed, its changes will persist even in the event of a system failure. This is achieved through the use of write-ahead logging (WAL), which records all changes made to the database in a separate log file before they are written to the database itself. In the event of a system failure, the database can be restored to a uniform state by replaying the log file.

In summary, durability is an important property of a transaction in PostgreSQL, which ensures that once a transaction has been committed, its changes will persist even in the event of a system failure. This is achieved through the use of write-ahead logging, which records all changes made to the database in a separate log file before they are written to the database itself.

In PostgreSQL, a database can contain one or more schemas. A schema is a named collection of database objects, including tables, views, indexes, sequences, and functions. Schemas provide a way to organize database objects into logical groups and can be used to control access to those objects.

When a database is created in PostgreSQL, a default schema named `public` is also created. This schema is where most database objects are typically created unless a different schema is specified. However, additional schemas can be created within the database using the `CREATE SCHEMA` command, which allows for greater organization and management of database objects.

Each schema in PostgreSQL is essentially a namespace that contains a set of related database objects. Schemas can be used to group related tables, views, and functions together, making managing and maintaining a large database easier. Additionally, schemas can be used to control access to database objects by granting privileges on a per-schema basis.

In summary, in PostgreSQL, a schema is a named collection of database objects within a database. Schemas provide a way to organize database objects into logical groups and can be used to control access to those objects. A database can contain one or more schemas; a default schema named `public` is created when a database is created.

Installation On Macos for Local Dev

To install PostgreSQL on macOS using Homebrew, follow these steps:

1. Install Homebrew by running the following command in your Terminal if you haven't already.

```
/bin/bash -c \
      "$(curl -fsSL https://phpa.me/homebrew-install)"
```

2. Once Homebrew is installed, run the followingcommands to update it, install and start PostgreSQL.

```
brew update
brew install postgresql
brew services start postgresql
```

3. To connect to the PostgreSQL server, you can use the `psql` command-line tool. To do this, run the following command:

```
psql postgres
```

This will connect you to the default PostgreSQL database named `postgres`.

That's it! You now have PostgreSQL installed on your macOS system. You may want to create your own super user account, which can be done via SQL. Once we have created our role, we can use `\\du` to show our current roles. (See Listing 1)

Listing 1.

```
1. psql postgres
2. psql (14.8 (Homebrew))
3. Type "help" for help.
4.
5. postgres=# CREATE ROLE my_super SUPERUSER LOGIN \
6.       PASSWORD 'secret';
7. CREATE ROLE
8. postgres=# \\du
9.                          List of roles
10.  Role name |                Attributes                | Member of
11. -----------+------------------------------------------+-----------
12.  halo      | Superuser, Create role,                  | {}
13.            |     Create, DB, Replication              |
14.  my_super  | Superuser                                | {}
15.
16. postgres=#
```

Installation On Windows for Local Development

To install PostgreSQL on Windows 11, you can follow these steps:

1. Download the PostgreSQL installer from the official website[3], making sure to select the appropriate version for your system.
2. Run the installer and follow the prompts to install PostgreSQL on your system. During the installation process, you will be asked to select the components to install and to choose a location for the installation.
3. Once the installation is complete, you can launch the PostgreSQL command-line tool by searching for "psql" in the Start menu. Alternatively, you can use a graphical user interface tool such as pgAdmin to interact with PostgreSQL.
4. To create a new database, you can use the `createdb` command in the command-line tool:

```
createdb phparch
```

Replace "phparch" with the name you want to use for your new database.

That's it! With these steps, you should have PostgreSQL up and running on your Windows 11 system.

Installation On Ubuntu 22.04

Install PostgreSQL by running the following commands. We'll check that PostgreSQL is running by running. By default, PostgreSQL is configured to use the "postgres" user with no password. To create a new user, you can use the `createuser` command

```
sudo apt-get install postgresql
systemctl status postgresql.service
sudo -u postgres createuser --interactive
```

Follow the prompts to create a new user with the desired permissions. To create a new database, you can use the `createdb` command:

```
createdb phparch
```

Replace "mydatabase" with the name you want to use for your new database.

That's it! With these steps, you should have PostgreSQL installed and running on your Ubuntu 22.04 system.

Postgresql Users and Roles

In PostgreSQL, a user is an account that can connect to a PostgreSQL database. Users can be assigned various privileges such as read, write, or execute access on specific database objects. A user is created using the `CREATE USER` command followed by the name and password of the user. For example:

```
CREATE USER john WITH PASSWORD 'phparch2023';
```

3 *official website:* https://www.postgresql.org/download/windows/

A PostgreSQL role is a named group of users and other roles that define a set of privileges. Roles are used to control access to database objects and to manage permissions for users. A role can be created using the `CREATE ROLE` command followed by the name and password of the role. For example:

```
CREATE ROLE editor WITH LOGIN PASSWORD 'hunter42';
```

Roles are preferred over users in PostgreSQL because they provide greater flexibility and control over permissions. For example, a role can be granted privileges to perform specific actions on behalf of multiple users, which can simplify the management of permissions. Additionally, roles can be granted to other roles, allowing for hierarchical permission structures.

By default, a new user or role in PostgreSQL has no permissions or privileges. To grant permissions to a user or role, use the `GRANT` command followed by the desired privileges and the name of the user or role. For example:

```
GRANT SELECT, INSERT, UPDATE, DELETE
    ON users TO editor;
GRANT SELECT, INSERT, UPDATE, DELETE
    ON users TO john;
```

Managing Roles and Permissions

In PostgreSQL, roles can be granted to other roles, allowing for hierarchical permission structures. To grant a role to another role, use the `GRANT` command followed by the name of the role and the name of the target role. For example:

```
GRANT editor TO super_user;
```

To revoke a role from another role, use the `REVOKE` command followed by the name of the role and the name of the target role. For example:

```
REVOKE super_user FROM editor;
```

To revoke permissions from a user or role, use the `REVOKE` command followed by the name of the privilege and the name of the user or role. For example:

```
REVOKE SELECT, INSERT, UPDATE, DELETE
      ON users FROM editor;
```

PostgreSQL provides powerful features for managing users and roles and controlling access to database objects. By using roles instead of users, administrators can create flexible and hierarchical permission structures that simplify the management of permissions and improve security.

Now that we've covered getting PostgreSQL installed and a database created, we should test drive a new application with our fresh database server.

Configuring Our Database Connection

To connect our PostgreSQL database with our fresh Laravel application, we'll update the `.env` with the following credentials and port number. The default server port will be `5432`, but if you end up with multiple versions of PostgreSQL running simultaneously, they will be running on different port numbers. This doesn't hurt anything but can get confusing.

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=phparch
DB_USERNAME=my_super
DB_PASSWORD=secret
```

Running Migrations

Applying our database migrations with `php artisan migrate`, we'll see the typical output showing our tables have been created. (See Listing 2)

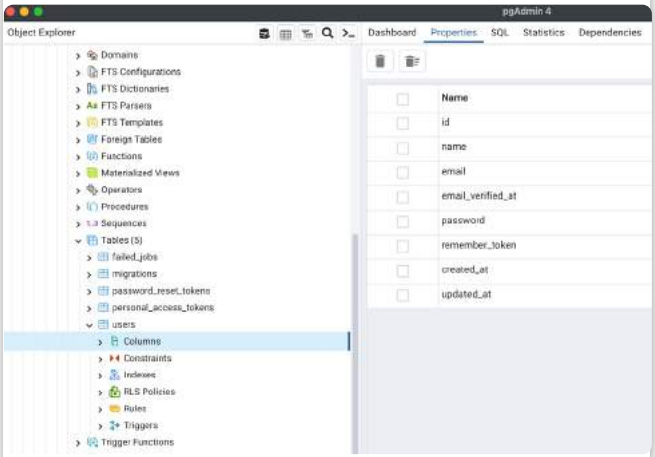
Listing 2.

```
1. php artisan migrate
2.
3.   INFO  Preparing database.
4.
5.  Creating migration table ..................... 22ms DONE
6.
7.   INFO  Running migrations.
8.
9.  2014_10_12_000000_create_users_table .......8ms DONE
10. 2014_10_12_100000_create_password_reset_to  3ms DONE
11. 2019_08_19_000000_create_failed_jobs_table .7ms DONE
12. 2019_12_14_000001_create_personal_access_t  7ms DONE
```

Listing 3.

```
1. phparch=# \\dt
2.                List of relations
3.  Schema |          Name           | Type  | Owner
4. --------+-------------------------+-------+-------
5.  public | failed_jobs             | table | halo
6.  public | migrations              | table | halo
7.  public | password_reset_tokens   | table | halo
8.  public | personal_access_tokens  | table | halo
9.  public | users                   | table | halo
10. (5 rows)
11.
12. phparch=# select * from users;
13.  id | name | email | email_verified_at | password |
14.  remember_token | created_at | updated_at
15. ----+------+-------+-------------------+----------+
16. ----------------+------------+------------
17. (0 rows)
18.
19. phparch=#
```

Figure 1.

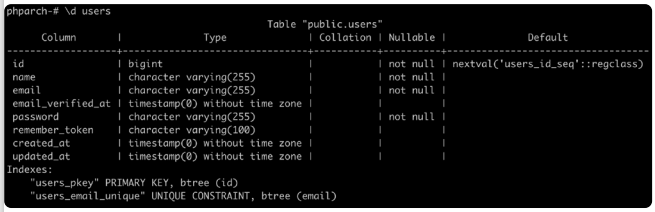


Listing 4.

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('users',
            function (Blueprint $table) {
                $table->id();
                $table->string('name');
                $table->string('email')->unique();
                $table->timestamp('email_verified_at')
                      ->nullable();
                $table->string('password');
                $table->rememberToken();
                $table->timestamps();
        });
    }
    public function down(): void
    {
        Schema::dropIfExists('users');
    }
};
```

Figure 2.



```
phparch=# \d users
                                        Table "public.users"
      Column       |            Type             | Collation | Nullable |              Default
-------------------+-----------------------------+-----------+----------+-----------------------------------
 id                | bigint                      |           | not null | nextval('users_id_seq'::regclass)
 name              | character varying(255)      |           | not null |
 email             | character varying(255)      |           | not null |
 email_verified_at | timestamp(0) without time zone |        |          |
 password          | character varying(255)      |           | not null |
 remember_token    | character varying(100)      |           |          |
 created_at        | timestamp(0) without time zone |        |          |
 updated_at        | timestamp(0) without time zone |        |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_unique" UNIQUE CONSTRAINT, btree (email)
```

Listing 5.

```
phparch=# SELECT table_name, column_name, data_type
FROM information_schema.columns
WHERE table_name = 'users';
 table_name |     column_name     |     data_type
------------+---------------------+---------------------
 users      | id                  | bigint
 users      | email_verified_at   | timestamp without tz
 users      | created_at          | timestamp without tz
 users      | updated_at          | timestamp without tz
 users      | password            | character varying
 users      | name                | character varying
 users      | email               | character varying
 users      | remember_token      | character varying
(8 rows)
```

Inspecting our database via CLI tool psql[4], we can use `\\dt` to show our tables and also run raw SQL queries to see that our `users` table is empty. (See Listing 3 on previous page)

If you would prefer a GUI tool, such as pgAdmin[5] is a great option to inspect our tables (See Figure 1)

pgAdmin showing our phparch database and users table Looking at the default `users` table migration for Laravel, we can see the following (See Listing 4)

Using `psql`, we can use the command `\\d users` to `DESCRIBE` our table. (See Figure 2)

We can also use SQL to query the `information_schema.columns` catalog to see our `users` table. (See Listing 5)

Just as we would see with MySQL, our `id` field is a `bigint` fiend, and our `string` fields end up being character varying[6], similar to MySQL's `VARCHAR` type.

I hope this has been enough of an initial dive into PostgreSQL to get you up and running, testing out your own applications, and seeing what PostgreSQL can do for you.

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

4 *psql: https://phpa.me/postgresql*

5 *pgAdmin: https://www.pgadmin.org*

6 *character varying: https://phpa.me/postgresql-datatype*

Site Navigation

Maxwell Ivey

Hello, as a kid, I was one of those kids who rarely asked why I had to do something. I just assumed my parents knew what was best for me and wouldn't make up rules just for their entertainment.

I feel that one of the problems with achieving a more inclusive internet is the fact that we, as disabled people, don't share enough about how we navigate the world or explain why we need a particular accommodation.

So, one of the best things I could do with this column is try to bring you into my world. I'm hoping that you will decide to be even more inclusive in your design after learning more about the why behind certain accessibility requests.

First, let's talk about the time factor. When I use my screen reader, I listen to so much information on the way to finding what I came to the site for in the first place. For friends who use screen magnification, they are often slowly scrolling through, also seeking info. Either way, it takes a lot longer to navigate to a website, find the link you want, and visit the related page, hoping you will find what you were promised to find there.

When seeking information using a search engine, the amount of time grows exponentially because so many websites don't deliver as the search engine suggested they would.

So, please create an orderly page structure with as few images, headings, links, buttons, etc., per page.

Ask yourself if this item *really* needs to be on the home page or start screen.

Next, you need to know that screen readers aren't very smart. They can only read things that the developer labels.

This is especially true with images. I imagine that many of you have at least heard a little about the need for alternative text tags? They are especially important, as the image doesn't exist without them. Nor does the link or button they are connected to. I will write more expansively on that topic in a future post.

Next, let's talk about how I navigate what information you determine needs to be on each page.

Using the tab key, I can navigate from the top left of a screen to the bottom left. The same works in reverse with shift plus tab.

There are similar keys to navigate by headings on a page. When reading headings, I will be moved through them by heading levels if there is more than one level.

So, you want to ensure that headings are consistent and in the correct order within headings.

Headings allow me to navigate a site so much faster than the other methods, so additional headings are especially helpful.

In fact, I would suggest adding headings for anything you want a screen reader user to find quickly, even if you wouldn't ordinarily add one.

Next, I can navigate the screen using the arrow keys in combination with other keys to move by character, word, line, sentence, paragraph, or page. However, these methods work best when I can anticipate where information will occur on a website.

You can help me and others like me by having the organization of your sites and the information on them appear in an orderly manner. And keeping your site consistent is also very important to users of adaptive tech.

Finally, I want to mention one way I can't access a site or an app.

Due to how screen readers interact with a computer, the developers have to disconnect the mouse keys. This means I can't perform actions that require a mouse click.

I can sometimes simulate a click, but it isn't reliable.

For this reason you should always offer a keyboard equivalent to any mouse-based actions you desire your users to have access to.

Finally, I want to encourage you not to make radical changes to a site or app once you have created the pattern unless you absolutely have to.

Because every time someone updates their site, there is the possibility that I may have to start all over and re-learn how to navigate sites or apps I use often.

Even a small change can greatly affect the life of an adaptive technology user visiting your site. And I wouldn't want that to come between you and regular disabled users of your website, app, or content.

I'm hopeful that this look into how I navigate the net will help you avoid some of the most common accessibility mistakes. I also hope it will help us to have a better, more fruitful conversation as we go forward together.

If you have any questions, please feel free to reach out. I plan to answer all messages. And your question could end up being the basis for a future article.

Maxwell Ivey is known around the world as The Blind Blogger. He has gone from failed carnival owner to respected amusement equipment broker to award-winning author, motivational speaker, online media publicist, and host of the What's Your Excuse podcast. @maxwellivey


CREATING A PHP LIBRARY WITH PACKAGIST
GETTING OUT OF PHP DEPENDENCY HELL WITH COMPOSER
php[podcast] Episode 2023.1.1
QUEUES WITH PHP'S SPLQUEUE
THE PHP DATETIME CLASSES
WHAT'S CHANGED IN PHP 8.2
WRITING CLEANER MORE RESILIENT CODE WITH VALUE OBJECTS
DEPLOYING OUR CODE WITH DEPLOYER 7.0
php[architect]
STAY UP-TO-DATE
WITH THE LATEST PHP TRENDS
AND TECHNOLOGIES
Parallelize
Decrypting Cryptography
About PHP's Git Server
PHP[WORLD]
LIVE STREAM
THE FUTURE IS COMMUNITY
PHP[TEK]
SUBSCRIBE
www.youtube.com/@Phparch

Maze Directions

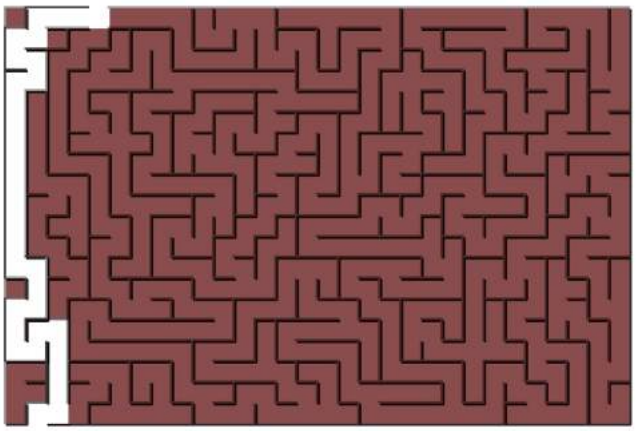
Oscar Merida

We know how to plot a course through a maze. Next, we're tasked with translating the maze solution into a list of directions to escape the labyrinth. We'll study the problem, come up with a solution, and throw in some unit testing for good measure.

Recap & Planning

Recall from previous months that we can generate a maze, draw a map of it, and superimpose the path to escape (Figure 1).

Figure 1.



> *"How am I supposed to follow this map if I'm stuck in the maze with treklins chasing me?" The wizard Archlin exclaimed as you handed him a scroll with the exit path marked.*
>
> *"This simply won't do, apprentice.", he mutters with a wagging finger pointed at you. "I want the directions in text. From the entrance, tell me if I should go forward, left, or right. Don't make me think. I need to get out quickly!"*

Let's look at our path overlay again. What do we notice? Our path doesn't loop since our maze doesn't repeat itself. Even though our maze is a two-dimensional array of cells, we can write the path to the exit as a sequence of cells we need to walk through. That is something like the notation below. Remember, the first coordinate is the row we're in, and the second number is our column.

```
(0,4), (0,3), (0,2), (0,1), (1,1), (1,0), (2,0),
(2,1), (3,1), (3,0), (4,0), (5,0)...(19,2), (19,1)
```

Once we have a list of the coordinate pairs, we can turn them into directions like:

```
Turn left, go forward
Go forward
Go forward
Turn right, go forward
```

"flattening" the Path

How can we "extract" the cells we need to visit while maintaining the order in which to traverse them? We need to know where the entrance and exit are, for one. Last month, I found the path out by "filling in" all the dead ends in the maze. This left cells on the path untouched, and each cell is only connected to two other cells or one other cell, either the entrance or exit. So, once we know where the entrance is, we should be able to extract a list of the cells on the path.

The Solver class from last month can return all our cells. Since we're not interested in filled-in cells, we can remove them right off the bat. Then we can find our entrance, which is always on the north wall or row `0`. The entrance cell will NOT have a wall in the north, which we've represented using the bit for 8. Likewise, the exit is always in the last row and doesn't have a southern wall.

I added a `flattenPath()` method to the Solver class I used last month. Once all dead ends are found, it returns an array of row-column coordinates to visit to get from the entrance to the exit. See Listing 1. It's a brute-force approach of looping through the cells and figuring out which cell to go to next by tracking which ones are already in our visited path. We have to account for each cell's walls since they define the open connections between two cells.

Making Step-by-step Directions

Our desired solution is a list of directions telling someone to turn left, right, or forward. This is trickier than just telling them to go north, south, east, or west because we have to know which way we're facing to give the relative moving instructions. Someone in the maze can be facing any one of four directions and need to move to an orthogonal cell. That is, in one of four directions. This gives us 4x4 = 16 possible ways to go from one cell to the next. However, we never backtrack

Listing 1.

```
1. public function flattenPath(): array
2. {
3.   $cells = $this->getCells();
4.
5.   // get rid of closed cells
6.   $cells = array_map(function ($row) {
7.     return array_filter($row,
8.             fn($cell) => $cell !== self::CLOSED);
9.   }, $cells);
10.
11.  // find our entrance cell
12.  $entrance = array_filter($cells[0],
13.    function (int $cell) {
14.      return ($cell & self::NORTH) === 0;
15.  });
16.
17.  $path = [];
18.  foreach ($entrance as $col => $value) {
19.    $path[] = [0, $col];
20.  }
21.  $current = $path[0];
22.
23.  // now, we visit each every cell
24.  while ( ! empty($cells)) {
25.    // where can we go from this cell?
26.    [$currentRow, $currentCol] = $current;
27.
28.    $walls = $cells[$currentRow][$currentCol];
29.    // build a list of potential destinations, in most
30.    // cases this will be two and we should then
31.    // ignore the cell we we're coming from
32.    $maybe = [];
33.    if (($walls & self::NORTH) === 0 &&
34.        ($currentRow - 1 >= 0)) {
35.      // could go north
36.      $maybe[] = [$currentRow - 1, $currentCol];
37.    }
38.    if (($walls & self::SOUTH) === 0) {
39.      $maybe[] = [$currentRow + 1, $currentCol];
40.    }
41.    if (($walls & self::EAST) === 0) {
42.      $maybe[] = [$currentRow, $currentCol + 1];
43.    }
44.    if (($walls & self::WEST) === 0 &&
45.        ($currentCol - 1 >= 0)) {
46.      $maybe[] = [$currentRow, $currentCol - 1];
47.    }
48.
49.    // don't include cells we've already visited
50.    $maybe = array_filter($maybe,
51.      function ($c) use ($path) {
52.        // keep if we don't find it
53.        return ! in_array($c, $path, true);
54.    });
55.
56.    // we should ony have one cell to go to next
57.    $next = array_shift($maybe);
58.
59.    // add it to our path
60.    $path[] = $next;
61.    // remove from map so this loop ends eventually
62.    unset($cells[$currentRow][$currentCol]);
63.    // remove empty rows to end the looping
64.    $cells = array_filter($cells);
65.    // now move to the next cell and repeat
66.    $current = $next;
67.  }
68.
69.  return $path;
70. }
```

(if you're facing north, you're not going to go south), which removes four movement possibilities.

Facing	Moving to	Instructions
North	North	Go Forward
North	East	Turn right, Go Forward
North	West	Turn left, Go Forward
South	South	Go Forward
South	East	Turn left, Go Forward
South	West	Turn right, Go Forward
East	North	Turn left, Go Forward
East	South	Turn right, Go Forward
East	East	Go Forward
West	North	Turn right, Go Forward
West	South	Turn left, Go Forward
West	West	Go Forward

That table is the heart of our solution. We'll need to track which direction we're facing—which will always be the direction we last moved—and need to know which direction we're going from one cell to the next. To practice my unit testing skills, I think I'll use phpunit to design a `DirectionFinder` class.

Unit Testing Practice

I've installed phpunit with Composer and set it up to autoload my classes. As the first step, let's write a test to describe how our class might work. Before writing what the `DirectionFinder` class does, this gives us a chance to think about how we want someone to use it. There's no need for it to maintain state—I only want this unit of code to know how to go from one cell to the next. And yes, this could be a static method, but … why? Static methods are a great way to incur technical debt you'll regret later.

Since the `DirectionFinder` class doesn't do anything, the tests fail.

```
ERRORS!
Tests: 1, Assertions: 0, Errors: 1.
```

Listing 1 cont.

Listing 2.

```php
<?php

namespace OMerida\Maze;

class DirectionFinder
{
    public function get($from, $to, $facing)
    {
        return [Dir::TURN_RIGHT, Dir::GO_FORWARD];
    }
}
```

Listing 3.

```php
<?php

namespace OMerida\Maze;

class MoveStep
{
    /**
     * @param string[] $instructions
     */
    public function __construct(
        public int $direction,
        public array $instructions,
    ) {}
}
```

The next step is to update the class (See Listing 2) so the test passes. We can do so with a naive method to make phpunit happy.

```
$ ./vendor/bin/phpunit tests/DirectionFinderTests.php
PHPUnit 9.6.9 by Sebastian Bergmann and contributors.
.                                                   1 / 1 (100%)
Time: 00:00.005, Memory: 4.00 MB
OK (1 test, 1 assertion)
```

Next, I'd add another test case, maybe named `testMoveEastWhenFacingSouth`, and update the class to pass. Then another, and another, and iterate to refine our solution. However, I realized that the direction finder shouldn't just tell us how to get from one cell to the next, but also what direction we are moving in. And we have 12 test cases we want to verify work—we can use data providers to automate our test.

Listing 3 shows the value object I used to describe moving from one cell to another.

Skipping to the chase, I used a phpunit data provider to test the 12 cases we identified earlier. See Listing 4.

Finally, Listing 5 is the class and method for figuring out how to move from one cell to the next.

Writing the Directions

Now that we have our path out and can turn moving from one cell to another into local directions, we can put the two together. By looping through the path, we can look at the cell we're in, see where we want to go, and show the directions to our user. See Listing 6.

Running it given a solved path gives the output shown in listing 7. That's sufficient for now, but we could improve it. For one, the "Move forward." instruction repeats, our output might be more useful if we said "Move forward 3 times" instead.

> *Due to the number and size of listings, most of them will appear on the pages following this article.*

Bubble Sorting Time

Mazes have been fun to explore. Let's turn our attention to a staple of computer algorithms—sorting. Although PHP has a built-in `sort()` method which you should use, let's expand our understanding of how sorting can work.

> *For the next issue, generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small. Write a function that sorts your array from lowest to highest by comparing two adjacent elements at a time—a bubble sort.*
>
> *Bonus points—generate many such arrays, collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.*
>
> *Some Guidelines And Tips*
>
> - *The puzzles can be solved with pure PHP. No frameworks or libraries are required.*
> - *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
> - *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
> - *Refactoring is encouraged.*
> - *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
> - *Go ahead and try many solutions if you like.*
> - *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH[1] can be helpful when working on your solution.*
> - *To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.*

Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. @omerida

1 *PsySH: https://psysh.org*

Listing 4.

```php
<?php

namespace OMeridaTest;

use PHPUnit\Framework\TestCase;
use OMerida\Maze\DirectionFinder;
use OMerida\Maze\Dir as Dirs;

class DirectionFinderTests extends TestCase
{
    public function testMoveWestWhenFacingSouth()
    {
        $df = new DirectionFinder();
        $steps = $df->get(
            from: [0, 5],
            to:   [0, 4],
            facing: Dirs::SOUTH
        );

        $this->assertEquals(
          $steps, [Dirs::TURN_RIGHT, Dirs::GO_FORWARD]
        );
    }
}
```

Listing 5.

```php
namespace OMerida\Maze;
use Directions as Dir;

class DirectionFinder {
  // first index is the way we're facing,
  // second is which way we want to move
  private $stepMap = [
    Dir::NORTH => [
      Dir::NORTH => [Dir::GO_FORWARD],
      Dir::EAST => [Dir::TURN_RIGHT, Dir::GO_FORWARD],
      Dir::WEST => [Dir::TURN_LEFT, Dir::GO_FORWARD],
    ],
    Dir::SOUTH => [
      Dir::SOUTH => [Dir::GO_FORWARD],
      Dir::EAST => [Dir::TURN_LEFT, Dir::GO_FORWARD],
      Dir::WEST => [Dir::TURN_RIGHT, Dir::GO_FORWARD],
    ],;;
    Dir::EAST => [
      Dir::NORTH => [Dir::TURN_LEFT, Dir::GO_FORWARD],
      Dir::SOUTH => [Dir::TURN_RIGHT, Dir::GO_FORWARD],
      Dir::EAST => [Dir::GO_FORWARD],
    ],
    Dir::WEST => [
      Dir::NORTH => [Dir::TURN_RIGHT, Dir::GO_FORWARD],
      Dir::SOUTH => [Dir::TURN_LEFT, Dir::GO_FORWARD],
      Dir::WEST => [Dir::GO_FORWARD],
    ],
  ];
```

Listing 6.

```php
// Solver
$solver = new \OMerida\Maze\Solver($maze->getCells());
$solver->findPath();
$path = $solver->flattenPath();

// the last cell will be next to the exit
$cells = count($path) - 1;
// we enter form the north
$facing = \OMerida\Maze\Directions::SOUTH;

$finder = new \OMerida\Maze\DirectionFinder();
for ($i = 0; $i < $cells; $i++ ) {
  $go = $finder->get($path[$i], $path[$i+1], $facing);
  $facing = $go->direction;
  echo ($i+1) . ': ' . implode(' ', $go->instructions);
  echo PHP_EOL;
}
```

Listing 5 cont.

```php

  public function get(
    array $from, array $to, int $facing
  ): MoveStep {
    $direction = $this->getDirection($from, $to);
    return new MoveStep($direction,
      $this->stepMap[$facing][$direction]
    );
  }

  private function getDirection($from, $to): int {
    $rowDelta = $to[0] - $from[0];
    $colDelta = $to[1] - $from[1];
    // since we're moving orthogonally,
    // we don't consider diagonals
    switch (true) {
      case ($rowDelta > 0):
        return Dir::SOUTH;
      case ($rowDelta < 0):
        return Dir::NORTH;
      case ($colDelta < 0):
        return Dir::WEST;
      case ($colDelta > 0):
        return Dir::EAST;
      default:
        return 0;
    }
  }
}
```

Listing 7.

```
1: Turn right. Move forward.
2: Move forward.
3: Move forward.
4: Turn left. Move forward.
5: Turn right. Move forward.
6: Turn left. Move forward.
7: Turn left. Move forward.
8: Turn right. Move forward.
9: Turn right. Move forward.
10: Turn left. Move forward.
11: Move forward.
12: Move forward.
13: Move forward.
14: Move forward.
15: Move forward.
16: Move forward.
17: Move forward.
18: Move forward.
19: Turn left. Move forward.
20: Turn right. Move forward.
21: Move forward.
22: Turn right. Move forward.
23: Turn left. Move forward.
24: Move forward.
25: Turn left. Move forward.
26: Turn left. Move forward.
27: Turn right. Move forward.
28: Turn right. Move forward.
29: Move forward.
30: Move forward.
31: Move forward.
32: Turn right. Move forward.
33: Turn left. Move forward.
```

Listing 9.

```php
<?php

namespace OMerida\Maze;

class Directions
{
    // constants for the walls of a cell
    public const WEST = 0x1;
    public const EAST = 0x2;
    public const SOUTH = 0x4;
    public const NORTH = 0x8;

    public const TURN_LEFT = 'Turn left.';
    public const TURN_RIGHT = 'Turn right.';
    public const GO_FORWARD = 'Move forward.';
}
```

Listing 8.

```php
namespace OMeridaTest;

use PHPUnit\Framework\TestCase;
use OMerida\Maze\DirectionFinder;
use OMerida\Maze\Dir as Dirs;
use OMerida\Maze\MoveStep;

class DirectionFinderTests extends TestCase
{
  /**
   * @dataProvider moveProvider
   */
  public function testMoveWhenFacing(
    array $from, array $to, int $facing,
    int $newFacing, array $newInstructions,
  ): void
  {
    $df = new DirectionFinder();
    $steps = $df->get($from, $to, $facing);

    $this->assertInstanceOf(MoveStep::class, $steps);
    $this->assertEquals($newFacing, $steps->direction);
    $this->assertEquals($newInstructions,
          $steps->instructions);
  }

  public function moveProvider(): array
  {
    return [
      [[1, 0], [0, 0], Dirs::NORTH, Dirs::NORTH,
        [Dirs::GO_FORWARD]],
      [[0, 0], [0, 1], Dirs::NORTH, Dirs::EAST,
        [Dirs::TURN_RIGHT, Dirs::GO_FORWARD]],
      [[0, 1], [0, 0], Dirs::NORTH, Dirs::WEST,
        [Dirs::TURN_LEFT, Dirs::GO_FORWARD]],

      [[0, 1], [0, 0], Dirs::SOUTH, Dirs::WEST,
        [Dirs::TURN_RIGHT, Dirs::GO_FORWARD]],
      [[0, 0], [0, 1], Dirs::SOUTH, Dirs::EAST,
        [Dirs::TURN_LEFT, Dirs::GO_FORWARD]],
      [[0, 0], [1, 0], Dirs::SOUTH, Dirs::SOUTH,
        [Dirs::GO_FORWARD]],

      [[0, 0], [0, 1], Dirs::EAST, Dirs::EAST,
        [Dirs::GO_FORWARD]],
      [[1, 0], [0, 0], Dirs::EAST, Dirs::NORTH,
        [Dirs::TURN_LEFT, Dirs::GO_FORWARD]],
      [[0, 0], [1, 0], Dirs::EAST, Dirs::SOUTH,
        [Dirs::TURN_RIGHT, Dirs::GO_FORWARD]],

      [[0, 1], [0, 0], Dirs::WEST, Dirs::WEST,
        [Dirs::GO_FORWARD]],
      [[1, 0], [0, 0], Dirs::WEST, Dirs::NORTH,
        [Dirs::TURN_RIGHT, Dirs::GO_FORWARD]],
      [[0, 0], [1, 0], Dirs::WEST, Dirs::SOUTH,
        [Dirs::TURN_LEFT, Dirs::GO_FORWARD]],
    ];
  }
}
```

Listing 10.

```
1. public function flattenPath(): array
2. {
3.   $cells = $this->getCells();
4.
5.   // get rid of closed cells
6.   $cells = array_map(function ($row) {
7.     return array_filter($row,
8.             fn($cell) => $cell !== self::CLOSED);
9.   }, $cells);
10.
11.  // find our entrance cell
12.  $entrance = array_filter($cells[0],
13.    function (int $cell) {
14.      return ($cell & self::NORTH) === 0;
15.  });
16.
17.  $path = [];
18.  foreach ($entrance as $col => $value) {
19.    $path[] = [0, $col];
20.  }
21.  $current = $path[0];
22.
23.  // now, we visit each every cell
24.  while ( ! empty($cells)) {
25.    // where can we go from this cell?
26.    [$currentRow, $currentCol] = $current;
27.
28.    $walls = $cells[$currentRow][$currentCol];
29.    // build a list of potential destinations, in most
30.    // cases this will be two and we should then
31.    // ignore the cell we we're coming from
32.    $maybe = [];
33.    if (($walls & self::NORTH) === 0 &&
34.        ($currentRow - 1 >= 0)) {
35.      // could go north
```

Listing 10 cont.

```
36.      $maybe[] = [$currentRow - 1, $currentCol];
37.    }
38.    if (($walls & self::SOUTH) === 0) {
39.      $maybe[] = [$currentRow + 1, $currentCol];
40.    }
41.    if (($walls & self::EAST) === 0) {
42.      $maybe[] = [$currentRow, $currentCol + 1];
43.    }
44.    if (($walls & self::WEST) === 0 &&
45.        ($currentCol - 1 >= 0)) {
46.      $maybe[] = [$currentRow, $currentCol - 1];
47.    }
48.
49.    // don't include cells we've already visited
50.    $maybe = array_filter($maybe,
51.      function ($c) use ($path) {
52.        // keep if we don't find it
53.        return ! in_array($c, $path, true);
54.    });
55.
56.    // we should ony have one cell to go to next
57.    $next = array_shift($maybe);
58.
59.    // add it to our path
60.    $path[] = $next;
61.    // remove from map so this loop ends eventually
62.    unset($cells[$currentRow][$currentCol]);
63.    // remove empty rows to end the looping
64.    $cells = array_filter($cells);
65.    // now move to the next cell and repeat
66.    $current = $next;
67.  }
68.
69.  return $path;
70. }
```

Listing 11.

```
1. <?php
2.
3. namespace OMerida\Maze;
4.
5. class DirectionFinder
6. {
7.     public function get($from, $to, $facing)
8.     {
9.         return [Dir::TURN_RIGHT, Dir::GO_FORWARD];
10.    }
11. }
```

Create Observability, Part 2: Capture "Tribal Knowledge"

Edward Barnard

"Tribal knowledge" is information that should be written down but isn't. If it were easy to write down, it probably already would be! Another common problem is with documentation quickly becoming obsolete.

This month we'll see a specific technique for helping a specific area of "tribal knowledge". Our business processes run in phases according to the calendar. We'll use the fact that we have a spring season followed by tournaments as our example. Our tournament software assumes the regular season has finished. That makes sense for the production software but can be a problem when developing a feature during the off-season. The software configuration may not be correct for that feature to work in that developer's environment. What setup is needed to make feature development possible? That's something we don't have written down! Here's a solution.

Draw the Business Process

There are many reasons why you might want to draw out your business process. A whiteboard drawing might be sufficient; however, sometimes, something less casual will be needed.

How do you know when something less casual is needed? It depends! Here is the question I will be asking over and over again: "What problem are we trying to solve?"

At the USA Clay Target League, one of our current objectives is to get the "tribal knowledge" written down. Tribal knowledge[1] is:

> *A type of institutional memory that lacks adequate documentation, such that its preservation in the organization over time relies solely on processes such as mentoring, apprenticeship, on-the-job training, and, at the heart of all of those, continuity of staffing, which is inherently vulnerable to employee turnover (of both workers and managers). It is knowledge that is necessary to an organization's function and yet is inadequately documented or otherwise captured.*

At the Clay Target League, we started with a block diagram of domains and concepts (not shown here). This diagram brought out the fact that we have many temporal dependencies. For example, season competition comes after season registration. Tournaments come after the season competition completes. See Figure 1.

Figure 1.

However, our PHP codebase has no such concept. The code does not say "A" must happen before "B". Instead, the code implementing "B" simply assumes that "A" has happened and

1 *Tribal knowledge: https://phpa.me/wikipedia*

that, therefore, database, configuration, etc., are all correct for "B" to run successfully. There is no sanity checking.

Suppose "A" is Season Registration and "B" is Season Score Entry, which is part of Season Competition. Now, suppose that we need to update some feature related to score entry. Luckily we are doing this feature update in the off-season. This bit of luck means that season registration has not happened.

Solving the Problem

At this point, our question should be, "What do we need to set up so that we can exercise the new feature under development?" In other words, we need to make our system look like season registration has been completed so that season score entry runs correctly.

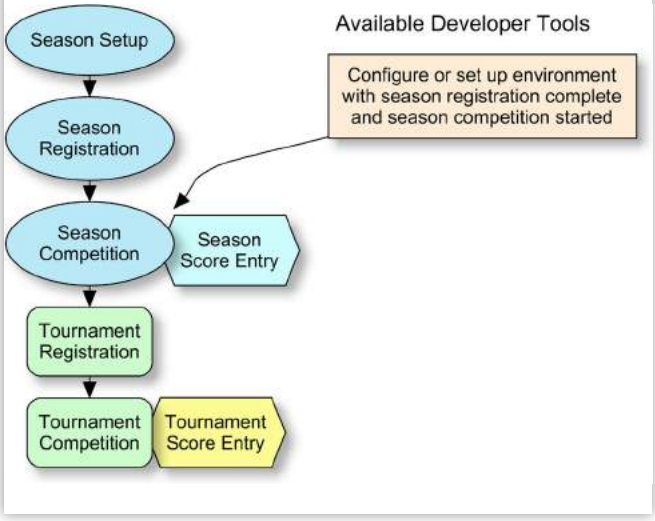
One useful approach is to write down those temporal dependencies as a high-level block diagram, as we did with Figure 1. Unfortunately, the result looks pretty silly. Everyone knows registration comes before the competition!

This is the time to ask, "What problem are we trying to solve?" We are not trying to solve the calendar. We are using the calendar to help visualize the source of the problem.

The problem to solve is that a developer working on score entry during the off-season must have an environment set up as if season registration is complete. There is a secondary problem: With a gap of six or twelve months before touching that code again (from spring to fall season or spring season to the following spring season), it's easy to forget the setup steps. A different developer might not even know that setup is required at all. We need to capture the "tribal knowledge".

One solution is to write a tool that sets up the environment correctly for season scoring. Then, call out that tool on the temporal flowchart. See Figure 2.

Figure 2.



Now that flowchart is not so silly anymore! That new tool itself almost became secret ("tribal") knowledge. People will only use the tool if they know the tool exists. If people actually use that "silly" flowchart as a repository of tribal knowledge, we've made our own lives easier.

Did We Solve the Problem?

I must ask again: What problem are we trying to solve? The title of this series is "Create Observability". Our new flowchart is observable, but what I really meant was "create runtime observability". We've yet to make progress in that direction.

Now that we understand the nature of our dependency problem, the solution becomes obvious. Create a tool that verifies the developer environment is correctly set up. If this tool is designed as a read-only tool, it could even be safely run after production deployment.

In fact, this tool could work as a mechanism for capturing tribal knowledge. Use our flowchart to identify each point where the software assumes a previous phase has happened. Write a tool that verifies the environment is correct for entering the phase of interest.

It turns out, however, that this is far more easily said than done! Because none of this tribal knowledge is yet written down, and figuring out the temporal assumptions is far from easy when examining the codebase—this is a remarkably difficult (and necessary) step.

We've just encountered another application of the Kent Beck principle[2]:

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*

Creating that environment-verification tool, for us, will be hard. But once created, the feature development becomes far easier.

Each such tool creates runtime observability. Observability of what? Of the state of the environment, that is, telling us whether the environment is correct for the software phase in question.

Call out each such tool on the flowchart (see Figure 3), and we have solved the "tribal knowledge" problem. It's solved, that is, so long as people keep looking at that flowchart!

Summary

Drawing out the business process can be useful for many reasons. In this case, we used a simple diagram to show where our software makes assumptions that can cause difficulty in each development environment. Each step of the way, we asked ourselves, "What problem are we trying to solve" to confirm that we are still on the right track.

We then used our simple diagram as a way of capturing tribal knowledge. We added a collection of potential tools marking each transition point from one business domain to the next.

2 *Kent Beck principle: https://phpa.me/kentbeck-0032*

Figure 3.

Related Reading

- *DDD Alley: Create Observability, Part 1: Local Timing* by Edward Barnard, June 2023. https://phpa.me/ddd-jun-23
- *DDD Alley: First, Make it Easy* by Edward Barnard, May 2023. https://phpa.me/ddd-may-23
- *DDD Alley: "Depend" toward Stability* by Edward Barnard, March 2023. https://phpa.me/ddd-mar-23

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. @ewbarnard

PSRs And PERs: What's Next?

Frank Wallen

We have reached the end of active PSRs and PERs. Now we can start looking at what's next at the PHP-FIG, which we will do in this month's column. We will look at the four PSRs in DRAFT status: PSR-5 PHPDoc Standard, PSR-19 PHPDoc Tags, PSR-21 Internationalization, and PSR-22 Application Tracing.

Psr-5: Phpdoc Standard

The goal of this PSR is to standardize the usage of the commonly used `DocBlock`, which uses the `DocComment` to describe the structural element that follows. Following is an example of a minimal `DocBlock` :

```
<?php

/**
 * This is a DockBlock
 */
function myFunction()
{}
```

Since PSR-5 is still in DRAFT status, there is no document or meta-document available at the PHP-FIG website, but you can still read the specifications in the fig-standards repo[1]. You can also follow the discussion on Discord[2] in the **psr-5** channel.

The format and definition of a PHPDoc is as follows:

```
PHPDoc       = [summary] [description] [tags]
summary      = CHAR (2CRLF)
description  = 1*(CHAR / inline-tag) 1*CRLF
; any amount of characters
; with inline tags inside
tags         = (tag 1CRLF)
inline-tag   = "{" tag "}"
tag          = "@" tag-name [":" tag-specialization] \
                    [tag-details]
tag-name     = (ALPHA / "\") \
                (ALPHA / DIGIT / "\" / "-" / "_")
tag-specialization  = 1(ALPHA / DIGIT / "-")
tag-details = SP (SP tag-description / tag-signature )
tag-description     = 1(CHAR / CRLF)
tag-signature       = "(" *tag-argument ")"
tag-argument        = SP 1CHAR [","] *SP
```

The first line states that `PHPDoc` is comprised of a *summary*, *description*, and *tags*, though each is optional. Note, then, that *summary* is a CHAR, or text, that is followed by two CRLF, meaning there should be a blank line before the description or the next tag. The summary should be limited to a single sentence but can be two. The *description* is a CHAR, too, like *summary*, but it can include markdown/markup language to show code examples. Tags follow summary and description and must have a blank line before them (a blank line follows summary and description before tags). A variety of tags, like `@param`, `@return`, `@property`, etc., that can be used (though `@var` should not be used in the `PHPDoc`), though it's accepted in a `DocBlock`. Each has particular accepted rules that the community recognizes, but are especially useful to IDEs and tools. For example, `@param` should be followed by the *type*, *name*, and an optional description like `@param string $name This is the name`. Tags are also being used as annotations to affect program behavior, so it is important to have a flexible standard. In some cases, annotations are used to define routing paths for controllers or to define properties that will be auto-generated by scaffolding tools.

Psr-19: Phpdoc Tags

This DRAFT PSR (PSR-19) aims to describe the full list tags for PHPDoc standards (namely, PSR-5), it will not include Annotations, nor will it describe best practices on the application of the PHPDoc standard. The tags are those that the PHP community, most IDEs, and tools recognize and can provide that information during code autocompletion. In inheritance, such as extending a Class or implementing an Interface, any PHPDoc elements that are 'missing' will be inherited from the extended Class or Interface. For example: (See Listing 1)

Here, a summary and version tag are in the `BaseClass`. When `NewBaseClass` extends `BaseClass`, the summary is inherited. The IDE would use the text 'This is the Base Class' as the summary for `NewBaseClass`, but the `@version` would be 2.0. There is a special tag called `@inheritDoc` that can be used

Listing 1.

```
1. <?php
2.
3. /**
4. * This is the Base Class.
5. *
6. * @version 1.0
7. */
8. class BaseClass
9. { }
10.
11. /**
12. * @version 2.0
13. */
14. class NewBaseClass extends BaseClass
15. { }
```

1 *repo: https://phpa.me/phpdoc-proposed*

2 *Discord: https://discord.gg/php-fig*

on extending or implementing classes in the PHPDoc to completely inherit the previously declared PHPDoc. It's as simple as:

```
/**
 * @inheritDoc
 */
class ABaseClass extends BaseClass
{}
```

In many cases, the PHPDoc inheritance is handled by IDEs and tools, but explicitly declaring it removes any ambiguity. Additionally, `@inheritDoc` can be used to augment an inherited *description*. This allows the implementor to change the summary, include the inheriting description, and include their own additional information that may include examples.

```
/**
 * The new summary of my class
 *
 * {@inheritDoc}
 *
 * This is an additional description that could
 * expand on the original description.
 */
```

Here the summary is replaced, then the original description inherited and is supplemented by the additional description below the tag. With strict typing in PHP, it will be interesting to see how much use some tags will have. For example: (See Listing 2)

Since, in NewPersonBuilder, we're strictly declaring the argument types and the return type, the IDE will understand the types and return without needing a PHPDoc.

Listing 2.

```
<?php

/**
* Create a person object
* @param string $name
* @param string $department
* @return \Person
*/
class PersonBuilder($name,)
{
    ...
    return $person;
}

/**
* Create a person object
*/
class NewPersonBuilder (
    string $name,
    string $department
): Person {
    ...
    return $person;
}
```

Psr-21: Internationalization

This PSR is still in its infancy, and the document in the repo is stubbed out. One should start with the meta document in the meantime. The goal of PSR-21 is to provide a standardized approach with an interface to denote that a message is translatable. It is still in flux as the discussions continue in the **psr-21** channel in the FIG Discord, so there is no interface, yet. The intent, however, is to produce a framework-agnostic interface. Translation and following language rules are not easy tasks, and formatting and encoding make them that much more complicated. A more common interface that can expand the developer's reach to more libraries and even services can significantly save effort.

Psr-22: Application Tracing

Also, in its early stages, PSR-22 aims to create a minimal interface for providing tracing signals to 3rd party libraries. Application performance monitoring tools (APM) have become more common, and a powerful method for watching the behavior of one's ecosystem live. The goals of PSR-22 are:

1. To provide a set of interfaces for library and framework developers to add tracing signals to their codebase. This would, in turn, allow other libraries to receive the same signals for further processing or analysis.
2. To allow traces collected by various providers to be reused by other providers.
3. This PSR may provide a minimal `TraceProvider`, etc., for other providers to extend, should they choose.

What is notable is the intention of creating interfaces for signal transportability to other providers (items 1 and 2). This means that tracing signals can be recorded and emitted in proper packaging to various libraries or services at any time, offering insight into the behaviors of the application.

Conclusion

While the active PSRs and PERs have come to an end, exciting developments are underway with the DRAFT PSRs. The standardization of PHPDoc usage with PSR-5 , the comprehensive list of PHPDoc tags with PSR-19 , a framework-agnostic interface for translation with PSR-21, and the creation of a minimal interface for providing tracing signals to 3rd party libraries with PSR-22. These are all promising initiatives that will benefit the PHP community. Keep an eye on the progress of these PSRs on the PHP-FIG Discord and `@phpfig` on Twitter.

Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. @frank_wallen

MPA vs. SPA vs. Transitional

Matt Lantz

Web Applications and website architectures have shifting trends, just like frameworks and languages. As one group's interest takes hold, another group's interest becomes the way of the past. However, the World Wide Web's core architecture has changed very little throughout its life; That statement could trigger some people, so I'll clarify it. Since its inception, the core of the World Wide Web was simply you enter an address in a browser, and you get content provided back. In cases where you submitted information, the server consumed it and either redirected you or provided direct content. Yet, regardless of this, developers have consistently attempted to force the server-browser relationship into some Frankenstein that either becomes a nightmare of maintenance or becomes a pattern we all choose to leave in the past.

Web Application architectures are often ranted about online and usually degrade into disgruntled folks proclaiming which anecdotal evidence is most apt for global adoption. Every application has unique needs and demands from the business and, to be blunt, will not likely expand beyond five developers working on it ever or have more than one hundred users. Yet, we extensively dive into new architectures that are pushed by community members regardless of whether or not they fit our application needs. As mentioned above, the Internet was not designed to be a native operating system, regardless of how hard Google pushed that idea. The World Wide Web and the browser's architecture were intended to deliver information worldwide. Its original design was not meant to replace tools like Word, Excel, video editors, large-scale data storage, etc. However, the technology has been expanded over the last 25 years extensively.

Laravel is, as we all know, a full-stack framework, but what does that mean? It means it provides sufficient means for the end-to-end delivery of a web application. Using Laravel's core structure is to create an application that is a Multi-Page Application. When a user submits data to the application, the page reloads or redirects to another page. This traditional approach to building a web application can be highly efficient, but it does require page reloads, which for some applications can be cumbersome for the user base. This approach can also result in slower page loading times, especially when the content could be better structured. The make-shift solution uses tools like JavaScript framework components to make Ajax calls behind the scenes. Though this is handy and can provide a more native app-like experience, it often can bubble into complex code bases with no accurate documentation or definitive coding patterns. Fundamentally developing an MPA application uses Laravel routes with standard page loading for the user experience, limiting JavaScript uses to minor UX improvements and potential data loading. For most developers in the Laravel community, this means focusing on Blade components and some Alpine sprinkles.

Neither SPA nor MPA are limited to monolithic or micro-service architectures but are fundamentally interchangeable. Designing a SPA that interacts with multiple micro-service APIs is relatively easy. MPAs as well can utilize micro-services which can be beneficial.

The alternative to the MPA is a Single-Page Application, an app where we have to split an application into a front-end layer that handles User Interaction, routing, etc., and a back-end layer which is generally a robust API. Sometimes this requires the JavaScript frameworks to take control of the browser history and manage all the page components in a virtual DOM. This approach can, and often does, result in poorer performance, though it feels more "native" as the page is not reloading all content, like an MPA. Front-end developers must also ensure they only have a few memory leaks to avoid bad performance, which is why this can get cumbersome.

So why have SPAs become so popular? Overall, SPAs offer large enterprise companies a way to split responsibilities across teams. One team handles the Back-End, and the other controls the Front-End. Rather than needing full-stack developers, corporations can hire specialists and often hire them on a contract basis to reduce total operational costs.

However, this separation of concerns benefit can go beyond the corporate bottom line. When executed well, it can provide an experience more akin to a native application.

The challenge overall is that you are constantly working against how the browser and Internet were intended to be used; this can often lead to developers using ad-hoc solutions to resolve matters. The benefit of the native experience is usually the primary focus of arguments for using a SPA, as well as the argument that the API can be well structured and tested and updates be implemented in either system without necessarily impacting the other.

But, we rarely hear about the challenges of creating two applications to manage and a shared state between

the systems. We rarely hear about the complexity of feature releases when you have a separate UI and back-end and how to manage a centralized feature management system that both application cores can speak to.

Another challenge with SPA structures is public vs private APIs. Overall, this is looking at use case designs and determining if an API will be called internally or externally, as both can have different authorization/access rules. SPAs promote the separation of concerns, meaning maintainability can see promising improvements. Still, it adds considerable maintenance requirements in terms of the number of team members needed and specialties in both Front-End development and Back-End development.

Transitional architecture balances performance, maintainability, and user experience, making it an excellent choice for web applications, especially those with midsize to small teams. But what is transitional architecture? It's fundamentally a hybrid approach. It uses server-side rendering components in React or other JavaScript frameworks or simple JavaScript frameworks like Alpine to sprinkle in JavaScript as needed. Livewire also attempts to implement a style of transitional architecture by performing hot swaps of DOM content, which emulates some of the SPA experience but maintains a more traditional MPA structuring. Transitional is neither fully committing to an MPA nor a SPA structure. It's a matter of using the necessary technologies to solve the appropriate problems in the proper spaces. In a transitional web application, you can have a page that loads a component that acts as a SPA. Overall, within the Laravel community, we have a few options for handling a transitional web application. Inertia, Livewire, or Blade, with a few sprinkles of Alpine or Livewire, can all help a team implement a SPA-like experience while providing an MPA architecture.

Each of these architecture patterns comes with security concerns, and your security requirements will impact which patterns you move forward with. Still, ultimately all of these patterns can be secure and not increase the application's vulnerabilities. If your application has multiple pages and requires a fast initial load time, MPA could be the way to go. On the other hand, if you want to create a seamless user experience with dynamic content loading without page refreshes, SPA would be suitable. If you're a small team or solopreneur, MPA will likely be easier to maintain and less cumbersome to add new features. But midsize teams or small teams in larger organizations will probably find SPA systems with well-suited APIs that could enable them to build at a faster rate with less impact across groups. If you have an existing MPA application that needs modernization gradually, transitional architecture could be helpful. It's essential to weigh the pros and cons of each architecture before making a final decision, regardless of who you follow on Twitter.

Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. @MattyLantz

Living on the Edge

Beth Tucker Long

There is a special group out there full of brave thrill-seekers. Adventurous people who venture beyond the security of the stable and stand on the forefront of uncharted waters. They hunt out dangers and ensure the safety of all of us. Their ranks are open, and you can be one of them.

Ok, so maybe it's not quite as dangerous and exciting as that first paragraph makes it sound, but you can join the ranks of PHP beta testers. PHP 8.3.0 Beta 1 has been released for testing. This is a very early release and should not be used anywhere that could even remotely be considered a production environment. That being said, it does need to be used in many different environments so that as many bugs as possible can be found and squashed before it joins the ranks of PHP's stable versions.

This is where you come in. Head on over to the 8.3 Download page[1] and grab a copy of the latest beta (Beta 2 is scheduled to release on August 3rd). Once installed, run the PHP test suite (for instructions, visit: https://qa.php.net/running-tests.php[2]), and see if you get any errors or failed tests. You can also run your custom app or codebase on it and run your internal test suite to see if anything fails under the new build. If you find anything amiss, you can submit a GitHub Issue on the php/php-src project (https://github.com/php/php-src/issues[3]) .

Before submitting a bug, be sure that you search the bug database to make sure no one has already reported your bug. If they have, feel free to leave a comment on the bug with any additional pertinent information or additional ways to reproduce the error. If you can't find your bug in the Issues, you can add a new Issue for it. Before submitting an issue, be sure to read the documentation on how to report a bug[4] . It is essential to include a way for the team to reproduce your bug by including information about the system running PHP and what steps you took that generated the error. Be sure also to include information about what you expected should happen and what actually did happen. Above all else, remember the people reviewing and troubleshooting the Issue you submit are volunteers. They are donating their time and expertise to make PHP better for all of us, so put in the extra effort to make your report really thorough and detailed. The more information you can get them, the easier it will be to get fix that problem.

Join the ranks of PHP contributors. Give back to the community that helps you stay employed. Explore the cutting edge. Be a PHP Beta Tester.

Beth Tucker Long is a developer and owner at Treeline Design[5], a web development company, and runs Exploricon[6], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development and Full Stack Madison user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 https://downloads.php.net/~eric
2 https://qa.php.net/running-tests.php
3 https://github.com/php/php-src/issues
4 https://bugs.php.net/how-to-report.php
5 http://www.treelinedesign.com
6 http://www.exploricon.com