www.phparch.com

December 2019
Volume 18 - Issue 12

php[architect]

EXPEDITION PHP

Escaping An Abandoned Framework

Going Serverless With Laravel Vapor

Dealing With Devpression
Or: How I Learned to Dislike Myself Less

You Can Do Open Source

Call for Speakers

PHP[TEK]
2020

Share your amazing stories about Tech Leadership, PHP Development and Web Technologies. We'd love for you to apply to our Call for Speakers for the 15th annual php[tek] conference. Proposals are accepted through Jan 3rd and full travel support is available.

NASHVILLE
TENNESSEE

May 18-21, 2020

Nashville, TN — Opryland

tek.phparch.com

Escaping An Abandoned Framework

Bryce Embry

In the past few years, I've been involved with four teams who built their bread-and-butter applications on the classic Zend 1 framework. Each team realized the framework was no longer supported, and each has struggled to figure out their next step.

I'm the lead developer on one of these anxious-to-get-out-of-Zend teams, I have worked on two other teams, and I sit close enough to the fourth team that I can hear them mutter about various issues that trip them up. On each of these projects, I've seen triumphs and tragedies. In this article, I want to share what I've learned about struggling to get out of an abandoned framework.

First, a spoiler alert—none of these applications is entirely out of Zend 1, so I can't describe the joy of deleting your last Zend module. However, maybe I can give you hope that there is a way out. There are also ways to get so tangled up you may never get out, as well as a way to comfortably stay in, but we'll get to that in a minute.

How We Got Here

Zend 1 was arguably the best framework available a decade ago, and it became the foundation for countless business applications. While the PHP landscape has moved on, Zend 1 reminds me of my trusty 2000 Toyota minivan. It was a well-built car that served me reliably for 356,000 miles, and it was still running at the end. But while the engine was purring and the car still served its purpose, the years took their toll on other parts of the car. Time is not always kind; we have to keep moving forward.

And therein lies the problem—Zend 1 was replaced by Zend 2, and the Zend folks did not provide an easy upgrade path. As a result, many programmers indefinitely postponed upgrading, choosing to stay with their reliable old framework. Eventually, Zend 3 came along, but by then newer frameworks like Symfony and Laravel were more appealing than upgrading to another Zend.

That's where many of us find ourselves today. Ten years ago, someone started building a business application in Zend 1. Over the past decade, the program has matured, capturing complicated domain knowledge that's not recorded anywhere else. Now, because the foundational technology is outdated, we want to move the application out of Zend 1, but first, we need to find a way out.

Possible Exit Strategies

Of the four teams I'm familiar with, I've seen three different exit strategies.

Strategy One: Why Bother?

Zend 1 works. Okay, it's abandoned, and it doesn't use real namespaces, and it won't get any security upgrades, but it has worked for ten years. It still works today, and it evens run on PHP 7. For some folks, that's good enough. Replacing Zend 1 takes countless programmer hours, and it can be hard to justify investing scarce resources in replacing working software just because it's old.

Based on that reasoning, one of my teams decided to stick with the abandoned framework. Their Zend 1 application isn't exposed to the public; it still serves its purpose reliably and doesn't need much on-going development. Refactoring the application is not a justifiable expenditure of their limited resources, so they chose to keep it running as long as they can. One day it may be worthwhile to rewrite the app, but that's not today; they have more important business concerns.

While there are drawbacks to sticking with Zend 1 or any outdated platform, in software development, there aren't always "right" and "wrong" answers. Instead, there are different buckets of trouble, and we have to choose the bucket we are most comfortable carrying. For some folks, sticking with Zend 1 is a reasonable option.

Strategy Two: Leave Frameworks Behind

Our precious application is fossilized in Zend 1, and we are trying to set it free. Imagine that we make the herculean effort to move it into the latest version of Symfony. Ten years from now, will it take another heroic effort to pry our application out of Symfony? Why free ourselves from one tar pit only to get mired in another?

That rationale inspired the decision of another team. Their lead developer felt burned by framework lock-in and vowed never to put himself in that position again. Instead, he is using Composer to pull in the best components he can find and piece them together into a flexible programming platform.

The team was able to sneak the application out of Zend gradually by using this approach. They started by replacing `Zend_View` with Symfony templates, then replaced the `Zend_Registry` with PHP-DI for dependency injection. They are now at the point of replacing `Zend_Db` with Doctrine. By addressing one manageable chunk at a time, they are slowly dismantling Zend and erecting their menagerie of components. In a few months, they will no longer be tied to any particular framework.

I have mixed feelings on this approach. It works, but it's a complicated path to tread. You need a deep understanding of architecture to patch together a framework from separate components successfully. Once you've done that, the parts don't give you the same benefits of a modern, modular framework.

And while swapping out components sounds easy in theory, the reality may prove challenging. If your components become tightly coupled with your business logic, they are not easy to replace. Instead, you're locked into your peculiar framework.

This approach reminds me of a custom bookshelf I made for my wife's art supplies. It's exactly the shape she wanted, and it looks pretty good. But some of the joints aren't entirely square, some of the shelves don't line up quite right, and I hate to admit it, but it's a little wobbly. I made a functional shelf, but I didn't have the expertise to make a fine piece of furniture.

Custom framework mash-ups can be a lot like my bookshelf. Some folks can piece together components to make a decent framework, and the more experience and skill they have, the better their custom solution will be. However, I don't have the experience or expertise to cobble together a framework as good as ones already available.

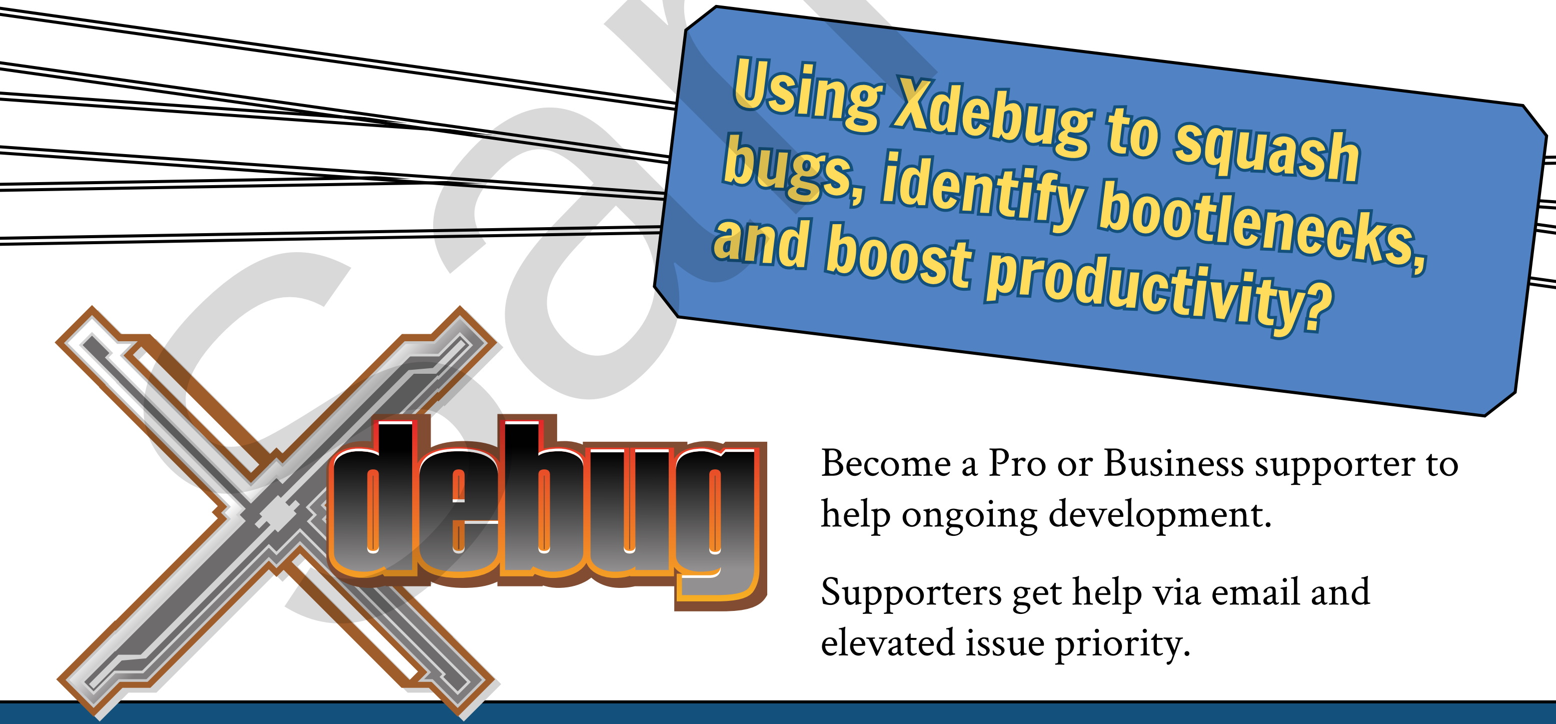
Many of the modern frameworks, like Symfony, are becoming more modular, allowing programmers to use only the components they need and leave the unwanted bundles out. In effect, they allow us to build our framework out of standardized, interlocking components.

While I respect the fear of getting locked into a framework, I don't think the result is comparable to the professional frameworks available today.

Strategy Three: Move to Another Framework

If we're not staying in Zend 1 and we're not creating our framework, then our third option is to move into a new framework. That's what the last two teams have done. Each group is significantly upgrading and expanding an older application and wanted to build in a modern framework, so each chose to start moving from Zend into another full-featured platform.

At this point, it would be nice to talk about which modern framework is the "best," and all of the pros and cons we weighed to select our new platform. For us, the choice was pretty simple. We've built most of the web applications in our office in Symfony. We are surrounded by folks who have a lot of Symfony experience. It would be foolish not to take advantage of such a ready bank of wisdom, so Symfony was our logical choice.


Using Xdebug to squash bugs, identify bootlenecks, and boost productivity?
Xdebug

Become a Pro or Business supporter to help ongoing development.

Supporters get help via email and elevated issue priority.

https://xdebug.org/support **support@xdebug.org**

Setting Up the Exit Route

If we're going to move out of Zend and into Symfony, how do we keep the old website running while we build its replacement? We start by laying a little groundwork.

PHP 7

All the instances of Zend 1 I've worked with were originally running on PHP 5. However, Symfony 4.4 (the latest long-term-support version) requires at least PHP 7.1. If we're going to move into Symfony, our first step is to upgrade the server from PHP 5 to PHP 7.

Fortunately, Zend 1.12 works with PHP 7.1. CodeSniffer's PHPCompatibility checker identifies 75 compatibility errors between PHP 7.1 and Zend 1, which seem concentrated on mcrypt and SQLite calls. However, in spite of the sniffer finding incompatibilities, we have not run into any problems running Zend 1 on PHP 7.1.

In my experience, the migration hasn't required much more than simply upgrading the server from 5 to 7. It may take some tweaking to get PHP 7 configured correctly on your server. You do need to make sure the custom code you've written in PHP 5 is PHP 7 compatible. Other than that, Zend 1 just works in PHP 7, which is a testament to how well it was initially crafted.

Composer

Modern PHP frameworks rely on Composer to handle dependencies, but Zend 1 was written in the BC (Before Composer) era. This leads to some interesting choices when teams commit code to their CVS repository. One team was committing all of Zend 1 to their Git repository, while another didn't have it in the repo and installed Zend separately on each server before pulling down the rest of their code. Those methods work but are a lot messier than using Composer.

If you don't have Composer set up, that's the next step. I won't go into Composer installation details, but for Zend integration just:

- Add `require '../vendor/autoload.php';` to the file where you're bootstrapping Zend.
- Add `"zendframework/zendframework1": "1.12.*"` to your `composer.json` file

If you don't have a `composer.json` file, Listing 1 shows one with just the basics for Zend. In this sample, replace "MyApp" with the namespaces you'd like to assign to your Zend "library" and appropriate "application" folders.

Now, whenever you run Composer you'll see the secret message revealed exclusively to Zend 1 users:

```
Package Zendframework/zendframework1 is abandoned, you
should avoid using it.
```

Symfony

With PHP7 and Composer in place, the next thing to do is set up Symfony. Each team began their migration with Symfony 3—before the Symfony team introduced Flex and completely changed the directory structure. Starting with Symfony 4 and the Flex layout presents its challenges (especially since both Zend 1 and Symfony 4 Flex are looking for `public/index.php` as the base file), but in general, here's what we've done that worked well.

First, we created a separate route to the Symfony controller. In the `public` directory we created a symbolic link, `s`, to our `symfony` installation folder. When users go to `www.oursite.com`, they are routed to our Zend pages, which is what has always happened. However, `www.oursite.com/s` directs them to our Symfony controller. The symbolic link in the public folder allows us to have both Zend and Symfony controllers running simultaneously, and makes the transition between the two frameworks relatively seamless to the end-users.

Next, we tackled authorization. Zend and Symfony each have their authentication systems, but we need them to work together. We didn't want a user to authenticate in our Zend application and then have to log in again when they hit a Symfony page. To avoid that scenario, we let one framework handle the authentication and pointed the other framework to this authoritative login system.

We continued using Zend for authorization and had Symfony generate a new session with a valid user token based on the Zend login. In retrospect, it may have been better to do it the other way around (make the new Symfony system authoritative), but the main idea is to configure your system to have one single authorization.

Finally, there are few other specific details to get Symfony and Zend playing well together. These specifics vary based on which version of Symfony you install and how you configure

Listing 1

```
{
  "autoload": {
    "psr-4": {
      "MyApp\\Library\\": "library/",
      "MyApp\\Modules\\": "application/modules/"
    }
  },
  "require": {
    "zendframework/zendframework1": "1.12.*",
    "zendframework/zf1-extras": "1.12.*"
  },
  "repositories": {
    "packagist": {
      "type": "composer",
      "url": "https://packagist.org",
      "allow_ssl_downgrade": false
    }
  }
}
```

it, but know it's possible, and you can find most of the answers online.

Now, skipping ahead in a fast-forward montage, we have Symfony and Zend standing side-by-side and sharing a login. All that's left to do is move everything over, which is not quite as simple as you'd hope.

Lessons Learned

Updating your server and installing Symfony was the easy part. Now comes the real chore—moving ten years of stuff into the new framework. Do you know how much fun it is to rent a U-Haul and move all of your worldly possessions from one house to another? Moving into a new framework is almost as much fun, and gives you just as many opportunities to damage your precious cargo.

As someone who has been privy to a few of these moves, let me share a few hard-earned lessons.

Don't Look Back

In the Hebrew Bible, there is a story about a family fleeing a doomed city. As fire rains down on the city around them, an angel tells Lot and his family, "don't look back." Lot and his daughters keep running ahead and are saved from the catastrophe, but Lot's wife looks back and turns into a pillar of salt.

It's not a happy story, but it carries an important message—don't look back.

One of our teams started escaping Zend and moving into a new framework, but then they looked back. They wrote code in Symfony that relied on their old Zend code, and their project turned into a pillar of salt.

Well, not really a pillar of salt, but it turned into a mess, which is close enough. The team needed the new code to send an email, but had not yet created an email service in Symfony. In a hurry to get this code into production, they didn't want to build a new module in Symfony. Since they already had a full-featured email system in Zend, they pointed the shiny new Symfony class to the mature Zend email component. It seemed like a nice shortcut at the time; they figured they would replace this little hack soon enough.

But they didn't replace it because, as always, there were a lot of other things to do. Instead, they built more new Symfony modules that relied on the Zend email system. Then, emboldened by the apparent success of this pattern, they built other Symfony code that relied on other Zend code. Now, the two systems are so intertwined that replacing the old Zend code becomes more difficult and less likely every day. They're not really getting out of Zend; they're building a Symfony program with a Zend foundation.

They're stuck.

Don't look back to Zend.

Don't build new modules relying on code you plan to replace. It takes time to build new modules in Symfony, and it's always more convenient to fall back on the old code. But if you're ever going to get out of Zend, you must take the time to get out of Zend. So, keep moving forward.

This is a hard bit of advice to follow sometimes, but it's crucial. On our team, we look for opportunities to move code into Symfony. If we need to update a page generated by a Zend controller, we try to take advantage of the opportunity to move the page into Symfony. If some business logic changes dramatically, we rebuild the logic in Symfony instead of updating Zend.

By building bits in Symfony whenever we can, we make it easier to move other logic into Symfony later on. Step by step, we get ourselves out of Zend, and we have a pleasant zealot who looks at every pull request to make sure nobody introduces a Zend dependency into the Symfony code.

When you're building new code in Symfony and fire is raining down all around you, it's tempting to look back to Zend, but remember Lot's wife.

Let the New Code Mature

One thing which pushes teams to look back is they want the new code fully grown right now. They don't have the patience to let it gradually mature. And since it can't quickly do everything the old code can do, they look back to Zend to get the work done.

But, like children, applications need time to mature. When our kids were born, they couldn't do anything useful—except look cute, which they didn't do consistently. It took a long time for them to learn to talk, then walk, control their bladders, read, think useful thoughts, use tools, and eventually climb up in the attic and install CAT 6 cabling in our home office. They didn't start as cable monkeys, that took years of training.

The same is true of the code you write in the new framework. Your old code is mature and can do all sorts of valuable things (after all, it's ten years old, which is like 40 in human years). The code in the new framework, by comparison, is a toddler. It's cute and has loads of potential, but it still has a lot to learn.

Let your Symfony-based application mature gradually.

For example, when you're anxious to build out that new email component in Symfony but are pressed for time, it's okay to only build out the features you need right now. If you need to send a plain text email to a single verified user today, then build that bit of the email module on the Symfony side. Later, when you're moving over some code which requires more advanced features in the email system, add those features. You don't have to replicate all of the old Zend code in Symfony from the very beginning; you can incrementally build it as you need it.

And while you're at it, recognize that you need to mature as well. Just because you had a baby yesterday doesn't mean you understand how to be a parent. Similarly, if this is your first time using Symfony, there are many concepts which take you a while to become comfortable with. You may start out putting all sorts of logic in a repository and later think services would be a better place for that code. It took a while to learn the Zend 1 paradigms, and it takes a while to become comfortable in Symfony. Be patient with yourself, and know it takes time—and a few mistakes—for you to learn and grow as well.

Sometimes we get stuck because re-building full-featured components in Symfony is too daunting. But they don't have to start that way. It took a long time for your Zend baby to grow up. It's going to take a while for your Symfony child to grow as well. If you take just one step at a time and keep moving forward, one day that little monkey will be all grown up.

Plug Zend Into Symfony with an Adapter

While we never want Symfony to rely on Zend code, letting Symfony gradually take over Zend is a great way to move forward.

Going back to our email module example, let's say you finally get your Symfony email component fully built. It's now even better than your old Zend email system, but there are still a lot of places in Zend that rely on your original Zend email service. Since the new Symfony class has the same functionality, wouldn't it be nice to point all of those Zend references to the new code and completely delete the old Zend email module? Well, you can do just that, and yes, it is really nice.

To accomplish this, one of my teams created a `SymfonyAdapter` class which provides Zend access to selected Symfony services. We don't fire up a full Symfony instance; instead, we inject the necessary dependencies to get the appropriate Symfony class into Zend. There were a few tricky bits to set up, like instances of Twig and the TokenStorage, but once we had those figured out, injecting Symfony classes into Zend was a breeze.

Listing 2 is a simplified version of the `SymfonyAdapter` we're using. This example shows how we make our email service available in Zend. To do this, we are taking advantage of Symfony's auto-wiring feature to inject the dependencies we need to run the class. For this email service, we have to create instances of `Twig`, `TokenStorage`, and `SwiftMail`, which we then pass as parameters to the `EmailService`.

In our real `SymfonyAdapter` we have a couple of dozen methods shared with Zend, all of them following the same format as the `getEmaillService()` method here.

Once we could use the Symfony email system in Zend, we refactored all of Zend code to use the new Symfony class. When we finished that refactoring, we deleted the old Zend email class and had a grand little party. Thanks to this `SymfonyAdapter`, we have been able to migrate services out of Zend bit by bit, a process we can repeat until one day we won't need the adapter anymore.

Listing 2

```php
public static function getEmailService(): EmailService {
   $twig = self::getTwig();
   $token = self::getTokenStorage();
   $swiftMail = self::getSwiftmail();
   return new EmailService($swiftMail, $twig, $token);
}

public static function getTwig(): TwigEnv {
   $twigLoader = new TwigLoader(['../symfony/app/Resources/views']);
   return new TwigEnv($twigLoader);
}

private static function getSwiftmail(): \Swift_Mailer {
   $transport = new \Swift_SendmailTransport('/usr/sbin/sendmail -bs');
   return new \Swift_Mailer($transport);
}

private static function getTokenStorage(): TokenStorage {
   $tokenStorage = new TokenStorage();
   // getCurrentUser is a custom method in this adapter which
   // that gets the current User entity
   $user = self::getCurrentUser();
   $userToken = new UsernamePasswordToken(
      $user, null, 'account', $user->getRoles()
   );
   $tokenStorage->setToken($userToken);
   return $tokenStorage;
}
```

Share Entities

None of the Zend projects I've worked on were equipped with Doctrine, the object-relational mapper Symfony uses to communicate with your database. Instead, one team used the standard Zend 1 Database module, and the others had unique, custom-built ORMs.

When you're working on a growing application, the database is continually changing. So, having a custom ORM on the Zend side and Doctrine entities on the Symfony side means keeping two separate systems always in sync with the database. That's a lot of duplicated effort.

One of our early goals was to get Doctrine set up in Zend. We figured if we could get the two frameworks sharing the same ORM, it would be so much easier to refactor Zend code.

Having never used Doctrine before, it was daunting to set it up from scratch. It took a bit of research, and a couple of false starts to get it going, but it was well worth the aggravation.

Now we have a static `Doctrine` class (Listing 3) in Zend, which makes the entity manager, entities, and related repositories available to all of the Zend code. As we refactor Zend modules, we can replace the custom ORM calls with calls to the Doctrine entities. And, as the entities and repositories mature in Symfony, that new code is available to Zend, making it easier to replace and remove old code.

Take the time to get Doctrine in Zend. Here I've included a basic outline of the code we've used to make this happen. You'll notice we call a couple of specialized classes, `Credentials` and `Environment`, to get the database credentials and to identify whether we are in production mode or not. You can easily rewrite this class without those calls.

Don't Build On Deprecated Code

Earlier I mentioned my trusty old Toyota minivan that lasted 20 years and 356,000 miles. Toward the end, the dash lights stopped working reliably, the passenger seat was being held upright by zip ties, and the radio knob tended to fall off. As the car crumbled to pieces, I knew I would have to replace it one day, yet I kept putting more money into it. I installed new wheels and tires when the car had 330,000 miles, bought new shocks at 340,000 miles, did major engine work at about 350,000, then sold the car for scrap 6,000 miles later. I spent thousands of dollars in the last couple of years of owning that car, money that would have been better invested in a replacement vehicle.

There comes a point when you need to let go of the past, and the same is true with your old Zend code. You can patch it up, and it can keep doing what you need it to do. However, if you keep investing in the old code, you're not only squandering your resources, you're sabotaging your efforts to get out of Zend.

Every new line of code you write in Zend, someone has to rewrite in Symfony. So, not only are you investing the time writing the code in Zend, but you are obligating yourself to rewrite the

Listing 3

```php
<?php

use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\Common\Annotations\AnnotationRegistry;
use Doctrine\Common\EventManager;
use Doctrine\DBAL\Event\Listeners\OracleSessionInit;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Events;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;
use Doctrine\ORM\Tools\Setup;
use MyApp\Credentials;
use MyApp\Environment;

final class Doctrine
{
   private static $instance = null;
   private static $entityManager = null;

   private function __construct() {
      // Or however you get your dbase credentials
      $dbCreds = Credentials::getDatabaseCredentials();
      $dbParams = [
         'driver' => $dbCreds['database_driver'],
         'host' => $dbCreds['database_host'],
         'port' => $dbCreds['database_port'],
         'service' => $dbCreds['database_service'],
         'user' => $dbCreds['database_user'],
         'password' => $dbCreds['database_password'],
         'dbname' => $dbCreds['database_name'],
         'charset' => $dbCreds['database_charset']
      ];

      // Or however you get your dev mode
      $isDevMode = Environment::isDevMode();
      $paths = [__DIR__ . '/../symfony/src/AppBundle/Entity'];
      $cachePath = __DIR__ . '../cache / data / doctrine';
      $config = Setup::createConfiguration($isDevMode, $cachePath);
      $driver = new AnnotationDriver(new AnnotationReader(), $paths);
      AnnotationRegistry::registerLoader('class_exists');
      $config->setMetadataDriverImpl($driver);

      $eventManager = new EventManager();
      $eventManager->addEventListener(
         [Events::prePersist, Events::preUpdate],
         new DoctrineEventListener()
      );

      self::$entityManager = EntityManager::create(
         $dbParams, $config, $eventManager
      );
   }

   public static function getEntityManager() {
      if (!isset(self::$instance)) {
         self::$instance = new self();
      }

      return self::$entityManager;
   }
}
```

code in Symfony one day. It's like a fool buying high-end tires for a barely road-worthy car; it's not a good investment. In the long run, wouldn't it be better to write the code once in Symfony?

When you need to upgrade your application or create a new module, it may seem more straightforward to get it done in Zend. But keep in mind you're leaving Zend, and do the future you a favor. Take the time now to write the code in Symfony instead.

The Light At the End of the Tunnel

Ten years ago somebody chose to build their new application in Zend 1. Over the past ten years, that little program grew and matured, but the underlying framework didn't change. Now Zend 1 is abandoned, and it's time to move on.

Many of us have faced this situation and wondered what to do. Some chose to keep the original app running in Zend 1 because they can't justify the expense of a migration. Others have decided to roll their framework, never to be locked in again. And some are moving into new frameworks, like Symfony.

Moving from Zend into Symfony is a massive undertaking, but it can be done. It requires setting up a new environment, constantly moving forward instead of looking back, and giving yourself and the new software time to mature.

I've worked with teams taking this path, and while it is a daunting challenge, I can see the light at the end of the tunnel. One day we'll delete our last Zend module, and our application will be entirely in Symfony.

It may seem like refactoring an application into a new framework takes a long time; that's because it does take a long time. It took ten years to build this application in Zend—it takes a couple of years to rewrite completely while adding new features.

But the years are going to go by whether you start moving out of Zend or not. If you get moving now, maybe you'll be able to see, before too long, the light at the end of the tunnel.

Bryce is a Senior Application Developer at Vanderbilt University Medical Center, where he writes PHP every day to help medical researchers discover new cures. Over the years he has seen IT from many different angles, working as a teacher, Technology Coordinator, manager, and programmer. He lives north of Nashville, TN with a wonderful wife, a small dessert garden, an ever-inviting woodworking shop and a guitar that's too good for him. be@thornview.com

Related Reading

- *Migrating Legacy Web Applications to Laravel* by Barry O'Donavan, March 2019. https://phpa.me/legacy-to-laravel
- *Symfony 4: A New Way to Develop Applications* by Antonio Peric-Mazar, August 2019. https://phpa.me/symfony4-new-way
- *finally{}: Frameworks Don't Make Any Sense* by Eli White, November 2019. https://phpa.me/finally-nov-2019

DIEGODEV GROUP

Web Apps • Mobile Apps • E-Commerce

Developers who care about the code they create, the communities they build, and the solutions the implement

www.diegodev.com