Legacy to Laravel

By Lee Crosdale

Contents

1. Introduction
2. What is Legacy
3. What is Laravel
4. Setting
5. Moving to the cloud
6. Moving to GIT
7. Dev Environment
8. Moving to PHP 7.2
9. Nginx for laravel and legacy
10. DigitalOcean
11. Beware of third party libraries
12. Quick Recap
13. Models
14. Sessions
15. SEO
16. Admin Panels
17. Microservices
18. File Storage
19. Artisan Tasks
20. Where we are now

**Introduction**

Hello everyone, thank you for coming to my talk, Legacy to Laravel.

My name is Lee Crosdale, I’m a Senior Web Developer for a company named Magpie Solutions, our main project is a site named ‘ParcelBroker’ which is a logistics comparison website (think GoCompare for Parcels and Pallets).

I’ve worked on all sorts of projects, I’ve been coding professionally since about 2008, I started off working on a Microsoft Stack, Dynamics AX, MSSQL, SSRS, C#.net, so my background is quite corporate business orientated, I didn’t get into Web Development Professionally until a few years later, where I worked on a Joomla platform for a while (urgh).

And a quick plug, I run a Laravel Hackathon named LaraHack.com, it usually happens about 3 times a year, february is our 1st Anniversary, so looking to do some big prizes etc.

Ok well that is me sorted, now onto the talk!

I’ll start with a brief overview of what I’m going to (attempt) to cover and end with roughly where the project is now.

* What is Legacy Code?
* What is Laravel
* Where the project was at, at the beginning
* My experience with migrating a 10 year old web application into the Laravel Framework.
* The issue of where to start, mapping out the old code, and what to keep (if anything).
* This is an on-going project.
* If you have any suggestions, I’m all ears.

I want to make it clear that this is my first ever experience on something at this scale having a full REFACTOR over REWRITE, I’ve 100% made a lot of mistakes, but I like to think I have learnt from those, If you spot something I could have done better, let me know!

**Quote**

I’ve decided to lead with a quote, since I’m refactoring someone else’s code.

Richard Hamming was a famous mathematician who also worked on the manhattan project.

**What is Legacy?**

I would define legacy as ‘Code that still works, but better, smarter, faster ways now exist that would provide, added value in terms of functionality and maintainability’.

It’s also important to note that once a developer leaves, the code they wrote also becomes legacy, so it’s important to comment in code or write documentation.

Eventually all code becomes legacy, so it important to leave code in a manageable state.

**What is Laravel?**

I won’t keep you too long on this since I know most of you know what Laravel is, but for those who don’t, Laravel is a popular, if not the most popular PHP Web Framework. It comes packaged with a tonne of libraries ranging from ORM to Logging.

It has been my framework of choice for a number of years for a variety of reasons, one being that the development environment is so easy to set up.

**Setting: October 2016.**

I was brought in as a web developer to help the existing IT Director out with a couple of new Laravel based projects.

The main project, that this talk is based on, was running on a dedicated server that cost £400 per month, running Apache, MySQL 5.4 and PHP 5.2 w/ security patches (apparently) although I didn’t believe that, there was also a huge array of other ‘crap’ running on the server, mainly Java and Ruby code as scheduled tasks, a lot of development was done using production database too \*ARGH\*.

The codebase was managed via FTP, no backups except the daily production ‘clone’ that happened at the dedicated box’s physical location.

The infrastructure had been that way for quite a few years, and never had more than 1 person working on it. The system worked, but obviously had its flaws.

In early 2017 the IT Director left for a new job which left me to man the ship.

I now had a 1 million line, unruly beast to contend with, zero experience with the codebase, or even the industry itself.

[IMAGE OF THE SITE IN 2006]

**Moving over to someone else’s computer (The Cloud)**

I was pretty adamant to not have to use a dedicated box for a number of reasons, the first being that the entire infrastructure was one a single machine.

The second was that we had some other Laravel projects that needed to be put live, and I didn’t want to put them on the box too.

We had a few goals for the infrastructure

1. It needed to be cost effective (cheaper than it is now)
2. We needed to have full control over the boxes in terms of restarts, services etc
3. We wanted something that would do Continuous Integration (CI)
4. Had GIT integration
5. We wanted to be able to create new boxes and sites easily, preferably with some sort of pre-configured setup.

[change slide]

Our stack ended up as

* Cloudflare (DNS,DDos,WAF and lots of other nice things)
* DigitalOcean (Cloud Server hosting that was cheap and expandable)
* Laravel Forge (Create new servers, sites, with CI and GIT integration EASILY)
* GitHub (Source control, worth the money over free BitBucket)

I can’t say enough just how much Laravel Forge made this an easy process, within a few clicks you can have working infrastructure, and since pretty much cloudflare makes setting up the DNS so easy.

**Moving to GIT**

Before we could put any of the infrastructure live, we have to move the codebase over to GIT / Github.

As you know, it’s a relatively easy process.

A few issues I had was actually missing folders in the dev copy that I received, ones that were created in production and never actually synced with the local copy.

This caused a few issues when we launched the new infrastructure, but I’ll cover that later.

Bitbucket was used for a while, but it often went down.

A week after we moved to Github, BB was offline for nearly 2 days.

GitHub infrastructure is good, DDos took them down for 8 seconds..

**Preparing for the big switch**

**The dev environment**

Currently the dev set up time was close to a day, once you had mysql locally, all the code ftp down, all the nuances and configs configured.

I really wanted to make this A LOT quicker, I spent some time playing with Docker, but not having a lot of experience with it, I decided to go with Laravel Homestead.

We did briefly switch to ‘Laragon’ for quicker projects and to save resources (2gb became 200mb), but I had to make a switch back as there are issues with file locking in PHPs session handling when you make a request to the same web server.

I decided to host the dev databases on a server online, there were only 3 of us working on the project, so we all shared a single dev database.

The process for creating a new dev setup is now

* Homestead
* Git Clone Repo
* Db Connection details

Which basically stripped the setup time from a day to about 20 minutes (after install time etc).

**Moving from PHP 5.2 to 7.2**

The big one here was actually the use of the old mysql library.

Luckily a lot of the system was using a newer custom PDO class, but for the older queries.

Since the mysql functions were completely removed from later versions of PHP, for now we were able to create a new file, which replaced the mysql\_query with mysqli\_query etc.



This worked for the time it took to completely remove this class from the code over to the much better PDO class.

One other issue was there were a few classes that constructed based off the class name.

E.g



Now became



It might still work on 7.2, but it’s deprecated soon as far as I know.

**Nginx for Laravel and Legacy**

Adding the legacy code into the Laravel codebase was pretty easy, since the entire site ran from a root directory, we just had to move all the code into Laravel’s public directory.

Nginx was then configured to look for the .php file first, and if it’s not there, forward onto the new laravel framework. This meant that Laravel and Legacy were separate apps for now, in terms of sessions etc.

The forge config ended up as this.

index index.php index.html index.htm;

error\_page 404 /404/;

 location / {

 rewrite ^/v/(.\*)$ /voucher.php?v=$1 last;

 rewrite ^/feedback/(.\*)/(.\*)$ /feedback.php?&em=$1&fb=$2 last;

 try\_files $uri $uri/ /index\_laravel.php?$query\_string;

 }

Anything then ends in .php will be loaded, otherwise it goes through Laravel.

Any 404s also go to laravel. I actually have some logging on that, it’s sometimes interesting to see how search engines etc (but mostly hackers) try to access files.

 **DigitalOcean**

The DigitalOcean server at the beginning was relatively small, 1GB, but before we went to production our servers looked like this:

1 Web Server containing files and php - 4GB - $20

1 Database Server - 4GB Server - $20

1 CDN - 1GB Server - $10

3 - $10 servers hosting various other tools and apis that we use.

Our 400GBP monthly fee went down drastically, we run some other sites, our bill is around $150 a month. Still a lot smaller.

[SHOW IMAGE OF OLD FPDF SITE]

**Beware of Third Party Libraries** (that aren’t version controlled)

We deployed and made live the new site, all seemed well. Once the work day started, we quickly discovered that for some reason, customers were unable to download ‘Labels’ (small documents with some information on them generated when they place an order).

This was due to a missing library named FPDF, I went to the website to get a copy, and I saw the image you just saw.

No updates since 2015. Uh oh.

After doing some digging in the code (I was now looking for other missing dependencies) it turned out that there were actually a few PDF libraries that had been incorporated over the years.

FPDF actually still worked in PHP 7.2, but contains a lot of deprecated functionality.

HTMLtoPDF worked, as long as there were no secure (HTTPS) resources that it needed to load in, not ideal at all.

I also found another library that ran a binary (WKHTMLtoPDF) locally on the server , essentially a exec(‘/path/to/wkhtmltopdf’) so I got rid of that and replaced it with FPDF for now.

That is something still on going, If anyone has any recommendations (I need to convert HTML to PDF on the fly), I’d appreciate the input.

**Quick Recap**

So we have a new dev environment which setting up is a relative breeze, cloud servers, the latest PHP, GIT controlled source code and continuous integration.

Now the refactor to Laravel could begin.

**Models**

Models in Laravel were a breeze to create, simply

Php artisan make:model {modelname} -m (for migration if needed)

And you have a model ready to start using in the framework.

Since we were refactoring, the tables and data already existed, so we had to make some changes to the models.

In standard Laravel, a new table that contains a ‘increments’ field, will be named id, but ours were for example, the shipments table contained ‘shipment\_id’.

In laravel you simply do

Protected $primary\_key = ‘shipment\_id’

And you are good to go.

The other one that usually exists in a laravel generated table is created\_at and updated\_at timestamps, you can disable these like

Protected $timestamps = false;

Or if you have another field like ‘created\_date’ you can do

Const CREATED\_AT = ‘created\_date’

We also decided to put all our models in a folder named models, so the namespace became app\Models, since we had a lot of tables, this made sense.

I also looked into auto generating the models with various tools, but tbh I found a lot of these tools extended their own classes rather than the Eloquent model, and I didn’t really want that dependency

I don’t want myself to be the only person to understand that middle library that was used once and thrown away.

**Sessions**

Laravel and PHP do not share the same session data, Laravel has its own session management.

I currently have some code implemented that will convert php session data over to laravel, however, recently I have been considering moving everything over to redis sessions rather than laravels own.

My code sits in some middleware



Essentially for every request, I store php session data as ‘old\_data’, I made the decision to not convert laravel session data BACK to PHP, I want to have to change as little legacy code as possible, I certainly don’t want to have to code new legacy code! I also ensure users are logged in via laravel authorisation if they are logged in via the legacy method.

A quick note on login, Laravel uses a couple of different encryption methods, but at the time, bcrypt was the first choice.

If for some reason you still need to check that a hash matches in core php, you can use the password\_verify method built into php (tbh, this is probably where laravel ends up).

Also, our legacy system was using MD5 and salt to store user passwords, oops. I basically implemented some code where all accounts older than x were disabled, and emptied out all passwords stored for those. Users would have to request a password reset.

For our more current users, I left md5 logins for about 1 week, once they’d logged in, I stored it in the new format, and wiped the old one.

**SEO**

So during the laravel rewrite, we actually had an external company redesign the front-end layout.

Since such a huge amount of the code remained in legacy, we decided to put the new layout into that (as well as the laravel views).

I won’t say too much on that, the front end was basically jQuery and Bootstrap, but one this caused a huge issue, and that was SEO.

SEO plummeted during the redesign, since text on pages change etc.

We essentially built a mini CMS, in around 2 days, inside an admin panel (more on that later).

Using Laravel’s routes to ‘catch all’ we basically pointed everything to a ‘CMS Controller, and then done a small amount of internal routing there.

Laravel allowed our SEO to be saved, and now allowed some trusted employees to create new urls, blog posts and seo meta data in a page, without developer interference.

Laravel helped massively in this, and it really cemented it as a good choice, and that we definitely wanted the rest of the site in that framework ASAP.

**Admin Panel(s)**

During CMS creation, I had to very quickly build a new admin backend that was controlled purely in Laravel.

This had some pros and cons, for example, it’s now in Laravel, but employees had to maintain 2 admin logins.

Admin panels are also good places to start for refactoring, they aren’t customer facing, and pretty much every table is in some way touched by an employee.

Using Laravel and VueJS, we were able to streamline an insane amount of tasks, in some cases turning weeks worth of work into a couple of hours, and that’s also a downfall of legacy code, technical debt builds up and users of the system find that tasks take longer to complete, since more ‘fixes’ need to be performed in order to get the system to perform as intended.

**Microservices**

Not all code is ‘refactorable’.

Or at least, not all code is worth refactoring. Sometimes code needs an entire rewrite, turning complicated parts of a system into smaller microservices is, in my opinion, a great way to remove legacy code quickly, as well as allow contractors and others to quickly pick up small sections of a large codebase, and focus purely on that.

With Laravel we were able to quickly build and deploy complicated parts of the system, with minimal time spent, as long as we defined the API inputs (for which a working process exists in the legacy system), we could do anything with the middle data, and output a nice response for the legacy system to work off.

What this meant is that a single ‘Class’ for the microservice was needed in the legacy system, while the complicated parts remain external, in a laravel framework, this has saved weeks of work in some situations.

The only cons I’ve really found from this approach is that there is more than one framework, so it’s important to keep on top of updates.

**File Storage**

I don’t have a huge amount to say on this subject, right now we are still storing files on the local server which is a pain to say the least.

We have just started implementing DigitalOcean spaces, and it’s now a case of porting business logic into Laravel.

Spaces was so easy to use, as it uses the S3 framework, for which a laravel library exists.

Once file storage is sorted, I believe that we could set up an auto scale and load balancing solution, until then, the web server is stuck in a single cloud server.

**Artisan Tasks**

One issue we had was there were quite a few tools running java and ruby that meant when we made the switch to digital ocean, some automation was lost.

Artisan tasks were perfect for this, there isn’t much the laravel framework can’t do, and being able to repeat tasks with cron, made a lot of tasks really nice.

Everything the java and ruby code did, we were able to replicate (and improve) with Laravel.

**Where we are now**

All of the things I’ve spoke about today happened over a two year period.

This is still massively an ongoing project, and there is a lot of legacy code remaining.

However, about 300,000 lines have been moved to Laravel.

Day to day issues were fixed very fast, we often deploy 5 or 6 times a day.

It’s a common occurence to hear the words ‘Just rewrite it’ when a issue happens in a project, but it’s easy to forget how much potential time and energy went into that in the first place.

The project I’m working on has over 10 years of code inside of it, and even after 2 years we are still scratching the surface, yes, we could just scrap the code, and lose a lot of features, to simply move it over to Laravel, but it’s important to not upset the number one priority, the customers (and the cashflow).

If I could do some things differently from the start, I would build a lot of tests to start with, and follow a test driven development route, it’s easy to break someone else’s code when they aren’t about to stop you.

On the new infrastructure, we have 99.9% uptime, there is no more ‘day to day’ upkeep, most tasks are automated, and we are able to focus on providing a better value for service on the site.

Thanks for listening, any questions?