Elephant Sized PHP

Scaling up your PHP application

Andrew Beak

Elephant sized PHP

Scaling your PHP applications

Andy Beak

This book is for sale at http://leanpub.com/scalingphpapplications

This version was published on 2018-09-24

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For My Dad, Who gave me the outstanding advice to follow my passion in my career

Contents

Introduction . . . . . **i**

Asynchronous Workers **1**
- Queues 1
- Amazon SQS 2
- Autoscaling PHP Workers in AWS Elastic Beanstalk 5
- Other Queue Providers 7

Enemies of scale **8**

Servers **10**

Cookieless Subdomains **11**
- Creating the subdomain 11
- Set up the subdomain in Nginx 12
- Get a free certificate for your subdomain 13

Introduction

This book was written because I wanted a place to centralise my research and experience. I've been blogging for a number of years and have always enjoyed the value of having a reference to go back to. If I solve a problem I make a note of how I did it so that future Andy doesn't have to waste time doing what I did. This book is very much about that, it's about putting all the bits and pieces that I've learned into one place.

This book is divided into three sections. I firstly look at software architecture ideas that help when scaling an application. Ideally an application is designed with scaling in mind, this section of the book introduces some of the design concepts.

The next section focuses on gathering information about the running state of your application. Unless you're able to properly measure performance then you can't accurately measure the effect of a change.

Lastly we look at various server configurations and conclude with a general look into load balancing which also brings together some of the other concepts we dealt with in the book.

Nothing in this book should be done to a live server until you have completely and thoroughly tested it in dev and staging.

Your mileage may vary, be sure to test and consider everything properly, and anything you do to your server is your decision. This book does not constitute advice or any form of contract.

Don't experiment on servers that matter to you, spin up a temporary EC2 instance and experiment on that!

Asynchronous Workers

Some tasks, like sending an email, can take time to run but your user should not have to wait for them. These tasks should rather be processed by a separate worker that runs in the background. This allows your application to continue running without needing to wait.

When architecting your solution take a look and try to identify jobs which could be performed asynchronously. Usually the first place to look is for any sort of network request. If the network request times out then your user would be stuck waiting for 30 seconds or however long your timeout is set to.

Because your code is being run by a worker process you're able to completely decouple your job from your application code. This has benefits for software design and also allows you to scale your workers independently of your core application.

If a worker processing a job fails then your application is not affected which improves your app's resilience.

Queues

The best way to plan for including asynchronous workers in your application is to use a message queue. Message queues are scalable, durable, and can reliably deliver jobs to worker processors.

Queues provide the ability for your application to asynchronously process jobs. You can queue something up now, and process it later in a different worker process.

Databases are not Queues

Using a database to simulate a queue might be tempting but it is actually going to be more effort than using a dedicated queue. Consider a table like this:

id	Job	Status	Num Retries
1	Retrieve Twitter posts	Done	1
2	Send confirmation email	Done	1
3	Watermark S3 images	Waiting	2
4	Call data enrichment API	In progress	0

Our plan is to search the table for jobs that are waiting to be processed. We then update the database to mark them as "in progress" and start working on them. If a job finishes successfully we mark it as "done" or otherwise return it to "waiting" status while incrementing the number of retries.

Our worker can include logic to manage the maximum number of retries a job can have before being marked as failed permanently.

We'll need to build in row locking logic if we wanted to have more than one worker. Consider the case where we have five workers looking at the table. If a worker takes a job off the queue but doesn't update the status before another worker queries the table then both workers will process the job. To prevent this we'll need to make sure that we use database locks in our workers.

We haven't even begun to consider some of the other features that queues provide and we already have quite a lot of coding overhead.

Redis queues

Redis has native commands that let you manage a queue. Since version 1.2.0 Redis has supported the RPOPLPUSH command[1] which allows you to implement a queue that is resilient to network or client failures. Redis is a great queuing option and there is a lot of support for it in PHP.

Amazon SQS

Amazon Simple Queue System offers a 100% guarantee that a message it accepts will be delivered to a worker at least once. It scales automatically and can handle firehose volumes of messages. It's also trivially easy to set up monitoring on your queue that will automatically spin up new worker instances if a backlog is building up. In my opinion the ability to autoscale by adding more workers is the killer feature for cloud based message queues.

Setting up SQS

To setup SQS we'll follow these steps:

1. Create an IAM role to allow EC2 instances to access SQS
2. Spin up an EC2 instance and assign this role to it
3. Include the AWS SDK and connect to SQS

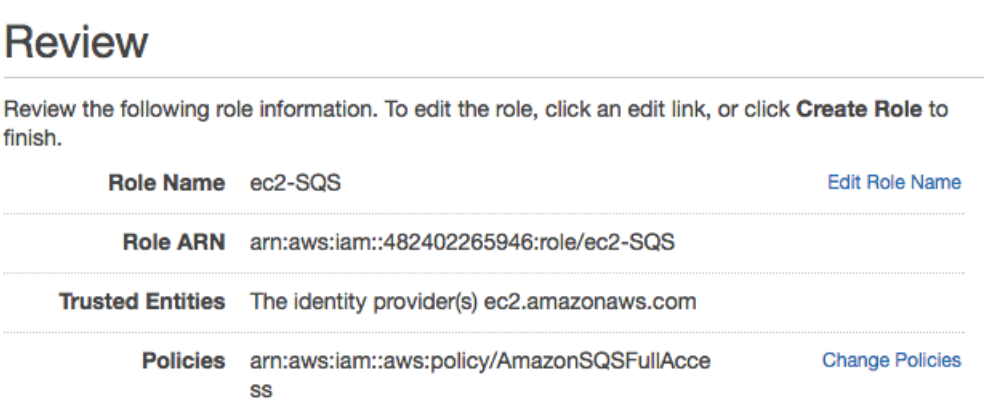
Create IAM role

Login to your Amazon dashboard and navigate to your IAM console. Add a new role and select "Amazon EC2" as the service role. This allows EC2 instances to call AWS services on your behalf. Attach the policy to allow full access to SQS to the role.

You should never store your AWS credentials in your application. Rather use Amazon's IAM roles to allow your worker instances access to the queue.

Your review screen should look something like this:

[1] https://redis.io/commands/rpoplpush



Example SQS setup

Spin up EC2 instance

You can't assign a role to an existing instance; you can only specify a role when you launch a new instance so spin up a new EC2 instance to experiment with. On step 3 where you configure the instance details make sure that you assign it to the role you just created.

Open up the Amazon SQS console and create a new queue. Leave the default settings for now, we'll be looking at queue options a little later.

Use the SDK to connect to SQS

We'll need the Queue URL, once you have it include the PHP AWS SDK[2] into your project with composer. My composer file for the example PHP code below looks like this:

```
{
    "require": {
        "php": ">=5.5.9",
        "aws/aws-sdk-php" : "\*"
    }
}
```

You're now able to create instances of the SQS client in your code like this example:

[2] https://github.com/aws/aws-sdk-php

```php
<?php
require('vendor/autoload.php');

use Aws\Sqs\SqsClient;

$params = ['region' => 'eu-west-1', 'version' => '2012-11-05'];

$sqsClient = SqsClient::factory($params);

$sqsClient->sendMessage(array(
    'QueueUrl'    => 'https://my_sqs_queue_url',
    'MessageBody' => 'Test Message',
));
```

Note that my credentials are not in my code, my instance is given permission to access the queue and so my PHP code does not need to authenticate itself.

Queue Settings

Amazon SQS offers a number of settings that can be tweaked.

Setting	Default	Used for
Default Visibility Timeout	45 seconds	The length of time (in seconds) that a message received from a queue will be invisible to other receiving components.
Message Retention Period	14 days	The amount of time that Amazon SQS will retain a message if it does not get deleted.
Maximum Message Size	256KB	Maximum message size (in bytes) accepted by Amazon SQS.
Delivery Delay	0 seconds	The amount of time to delay the first delivery of all messages added to this queue.
Receive Message Wait Time	0 seconds	The maximum amount of time that a long polling receive call will wait for a message to become available before returning an empty response.

The default visibility timeout setting is used to hide jobs that have been accepted by a worker from other workers. You don't want a job to be run more than once. Once a worker has pulled the job off the queue it is no longer available to other workers until the time in this setting has passed.

It should be set to be longer than the time you expect a worker to take to process the job. If a worker that took the job off the queue is still busy on the job and the default visibility timeout expires then

the job becomes eligible for another worker to pick it up, despite the fact that it is already running.

We do not typically use long polling processes in PHP workers. PHP is not designed to be a long running process and typically we spawn workers using a cron job or in response to other events, such as a notification service.

Dead letters

Messages on a queue that cannot be delivered can be placed onto a dead letter queue[3]. This is supported by SQS, RabbitMQ, Beanstalk, MSMQ, and other queue providers.

Storing the dead messages on a separate queue allows you to examine them to try and establish why they failed. Of course the real advantage is that your workers will no longer keep wasting compute resources by pulling the job from the queue.

To set this up on SQS you first add a new queue to hold the dead letters. Open up the SQS console, and click "Create New Queue". Name your queue something like "dead-letter" and save it.

You should be returned to the list of queues for the current region in your account. Your new queue should appear in the list, but select the queue you want dead-letter support for and select "Configure Queue" from the "Queue Actions" button.

In the popup window:

1. check the "Use Redrive Policy" box,
2. provide the name of the queue you just created, and
3. specify the number of times that a message can be picked up by a worker and returned to the queue before it is considered failed.

Now messages that workers fail to process will be moved to the dead letter queue.

Autoscaling PHP Workers in AWS Elastic Beanstalk

A huge advantage of using cloud services is the ability to provision server resources on the fly. AWS, Azure, and Google Cloud all allow you to spin up worker instances in response to events such as a backlog of queue items.

We'll have a look at how to set up an autoscaling group of workers that can connect to our SQS queue. I'm assuming that you have the role and the SQS queue set up from the preceding section.

In order to accomplish this we will:

1. Install the Amazon Elastic Beanstalk CLI tool
2. Pull the example from Github
3. Deploy it to Elastic Beanstalk

[3] https://en.wikipedia.org/wiki/Dead_letter_queue

Installing Amazon Elastic Beanstalk CLI

You can follow the instructions on the Amazon site[4] but if you're on Ubuntu all you need to do is:

```
apt-get install python-pip
pip install --upgrade awsebcli
```

You'll need to provide the CLI with authentication details. You can add these in the file at `~/.aws/config` (see the Amazon manual[5] for details). Your config file should look something like this:

```
[default]
aws_access_key_id=your_user_key
aws_secret_access_key=you_user_access_key
```

Pull the example from Github

I've created an example project based on Lumen, the micro-framework from the Laravel guys. You can clone it into a directory with Git:

```
git clone git@github.com:andybeak/sqs-worker.git
```

You should not need to run composer to install the dependencies. Elastic Beanstalk will do so for you when you deploy.

Deploy with Beanstalk

You first need to set up an application in Elastic Beanstalk. We do so by like so:

```
eb init sqs-worker-application
```

If you were logged into your Amazon web console you would be able to see this application listed in your Elastic Beanstalk section.

Now that we have an application we want to create an environment. For purposes of demonstration we're just going to create a production environment, but you can just as easily use different environments for different Git branches.

We're in the master branch of our Git repo and we want to associate this with our production stack.

Use the following command:

[4] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-install.html
[5] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-configuration.html

```
eb create sqs-worker-prod --single --branch_default --tier worker
```

In production you will likely not use the `--single` flag in which case Elastic Beanstalk will deploy an autoscaling load balanced group. You may also need to specify the VPC and subnet(s) to use if you have removed your account default. You'll find information on these and other parameters in the manual for eb create[6].

This means that your command may end up including the following flags:

```
--vpc.id=vpc-f61a9493 --vpc.ec2subnets=subnet-51711326 --vpc.publicip
```

Elastic Beanstalk will automatically create a security group to apply to your instances. If you selected to setup SSH when creating the application port 22 will be open in this group. You can add the instances to your own security groups by giving a comma separated list on the command line (see the manual).

After you've got your environment created you'll see it in your Elastic Beanstalk web console. If you click it you'll see more details, including a log of the various actions that have been taken on your behalf.

When Elastic Beanstalk set up the environment it automatically created a new SQS queue for you. It also set up CloudWatch metrics and triggers for the autoscaling.

All that you need to do now is use `eb deploy`. This will use git to zip up the project and then deploy it to the EC2 instances in the load balancing group.

Terminating your environments will tear down all of the infrastructure they created, as indeed will deleting the application.

Other Queue Providers

IronMQ, RabbitMQ and Beanstalkd are all strong contenders for your attention in the PHP world. They have well supported libraries and if you plan your architecture carefully you should be able to swap out a Queue provider without impacting your entire application.

The first big question to ask yourself is whether you want to self-host a queue (RabbitMQ[7] or Beanstalkd[8] or use a provider (IronMQ or SQS). There are pros and cons for both use cases.

There are costs associated with using a provided service, but whether these costs are greater than the cost of your engineers manually installing and maintaining the software is for you to decide.

[6] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb3-create.html
[7] https://www.rabbitmq.com/download.html
[8] https://kr.github.io/beanstalkd/download.html

Enemies of scale

Overengineering

Having worked in multi-disciplinary teams I have to admit that it's usually the Java developers who come up with a beautifully complex solution to a fairly simple problem.

Don't get me wrong, it's not about the programming language Java. I think they were probably the best trained of us all when it came to object abstraction. It was simply the hammer that they used to make every problem look like a nail.

The problem I've had with over-engineering is that it wastes time. The product that is delivered is a black box to me and I can't easily deploy it.

Complex code incurs technical debt and trying to scale up an application that is difficult to understand is very difficult. I'm busy with such a project at the moment and a great deal of my effort is going into getting the code into a form that is simple. Once the code is simple it becomes malleable, and so scaleable.

Inelastic infrastructure

I have worked in places where adding a new server was a major consideration that involved procuring the bare metal server and installing it into our rack at the data centre. The cost involved was substantial and we would get a fairly large step up in capacity for the money we spent.

Although we pay a premium for using cloud instances I believe the flexibility that they provide is definitely worth it. I can spin up a server and later decide to change its CPU and memory capacity without having to drive down to my data centre with a screwdriver.

I also really enjoy the fact that I can add much smaller "step ups" in my infrastructure and so respond more accurately to the demand on my network.

I think if I were to take my cloud experience back to the metal I would focus on looking for a way to use cheap commodity hardware and software. This would make the decision to ramp up easier and make it less expensive when something fails. Remember that if adding a server costs a significant amount then a proper procurement process must be followed. This is not really fun for anybody involved and takes time, so if you're in a hurry to respond to demand for your application you're going to sweat.

Not differentiating your persistence layer

Relational databases are great when you need ACID compliance and need to model relationships between data entities. There are however other persistence tools that may be suited better to other tasks.

Even the file system can be used quite successfully as a persistence layer. With modern file systems and the Linux OS cache you'll be able to store very large data sets. If you don't need to track relationships, have data that is not very volatile, and don't want to pay the overhead of ACID then a file system is a great place to store data.

If you need to store sessions then a key-value database such as Redis or Memcached is a great option.

Document stores like MongoDB or CouchDB allow you to create data models based on "documents". These documents contain the metadata about the fields they contain themselves. In other words you don't have to define the schema for a document in the database - the schema is contained within the document itself. Documents are keyed but can also be indexed on one or many of their keys.

Document stores are not ACID compliant but offer advantages over RDMS. MongoDB for example supports partitioning, sharding, and horizontal scaling right out of the box. In fact it was designed with these principles in mind which makes it much easier to scale than MySQL.

MapReduce applications like Apache Hadoop offer the ability to scale up extremely large data sets. They offer two functions - a Map function that performs filtering and sorting and a Reduce method that performs a summary operation. The key to scalability is the fact that the Map and Reduce functions can be performed on distributed servers.

Trying to solve every data storage problem with MySQL can make it more difficult to scale.

Not measuring and logging

I've dedicated a good portion of this book to metrics so I won't go into detail here. Suffice to say that you should be logging your application and using a service like Loggly or Splunk to centralize and monitor your logs for alert conditions.

Servers

“When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.” - *Edsger W. Dijkstra*

Cookieless Subdomains

Cookies are associated with a domain and are sent by the web server to the browser. Cookies contain pieces of information that the server needs, for example a session identifier or something similar. The server doesn't need this information for every resource request though. It can probably serve something static like an image or piece of Javascript without knowing anything about the cookie.

When it makes a request the browser includes all the cookies for the domain, regardless of whether the cookie is needed for the particular file it is requesting. This means that every request and response has the overhead of adding the cookie to it.

Cookies are attached to the domain so if we want to avoid sending them we need to make our request to a *different domain*, the cookieless domain. Unless your application sets a root domain cookie (or uses an application that does) you can use a subdomain of your normal domain. In other words makes sure that only cookies for www.your-domain.com are set, and not for the top level your-domain.com.

Using a cookieless domain also allows clients to download static resources in parallel to your pages. You could even consider sharding your static assets over multiple subdomains to allow further parallelization.

Creating the subdomain

For a large project you should always use a DNS provider that is able to scale and is reliable. In this example we're going to change our DNS provider to use the Google cloud provider.

Normally I would use Amazon Route 53 because my infrastructure is hosted on Amazon and there are useful gains to be had from having your DNS in the same cloud provider. However, this book isn't about Amazon it's about scaling. Google's offering is priced slightly cheaper[9] than Amazon and so I decided to use it for the website that accompanies this book.

I'm going to assume that you've signed up for Google Cloud Platform and have created a project for your application. From the project dashboard use the navigation bars in the top left and select "Networking". We need the "Cloud DNS" option that appears on the left.

Create a new zone. You can name it anything you like but use your application domain name for the DNS name. After you've created it you'll be take to the zone configuration screen. Google has set up four nameservers for you and a SOA record.

[9] https://cloud.google.com/dns/pricing

> The NS records store the details of the servers that will be providing the DNS services. The SOA record contains details about the domain such as the primary DNS server, and it can include contact information.

We'll be using a CNAME record to create our subdomain. This is an alias record and is convenient because if we change our server IP we only need to change one record in our DNS.

Use the "Add Record Set" button. Set the DNS name as "static" and the resource record type to CNAME. The canonical name must much your main domain name. We'll use the default time to live of 5 minutes for now, but you should set this to a higher value when you're in production.

Now we'll need to swap the zone over to use Google. My registrar is 123-reg.co.uk and all I needed to do from their control panel is to change the nameserver (NS) records to point to my newly created Google cloud servers.

At this point you're going to need to wait for DNS to propagate and for your changes to become live. We'll set up the Nginx server in the meantime.

Your site will probably not be available during this time so don't do this on an application in production.

Set up the subdomain in Nginx

We'll configure Nginx to serve requests for your cookieless subdomain from your assets directory in your application. This means that you don't need to change the layout of your directory structure.

You don't need to use the `fastcgi_hide_header` option to prevent cookies from being set. I'm including it here to bring your attention to it in the event that you use a variation on this use case.

```
server {
    listen       80;
    listen       [::]:80;
    server_name  static.your-domain.com;

    add_header 'Access-Control-Allow-Origin' 'https://www.scaleyourphp.site';
    add_header 'Access-Control-Max-Age' 1728000;

    location ~* \.(jpg|jpeg|gif|css|png|js|ico|svg|txt|woff|ttf|eot)$ {
        access_log off;
```

```
        expires    max;
        add_header Pragma public;
        add_header Cache-Control public;
        fastcgi_hide_header Set-Cookie;
    }
}
```

Notice that we're allowing cross origin resource sharing. We're allowing our source directory access to resources, since we're hosting fonts on the subdomain.

Get a free certificate for your subdomain

It could be that you don't have a wildcard certificate for your domain or can't extend your certificate to include the new subdomain. This isn't a deal breaker for using a cookieless subdomain, you can use a free certificate from Lets Encrypt[10].

We'll start by cloning the tool into a directory:

```
git clone https://github.com/letsencrypt/letsencrypt
```

You may want to place it into /opt/ because we'll need to be using it regularly to renew the certificates we create.

The tool needs to satisfy itself that you control the domain that you're generating a certificate for. In order to do so it will place a file into a '`.well-known`' subdirectory which the certificate issuer will verify.

If you're like me and block all request to hidden files then you'll need to place a location block *above* the location block that restricts the access. The reason we place it above the block is because nginx parses the location blocks in the order they're declared, if it is below the restrictive block then it will be overridden.

```
location ~ /.well-known {
        allow all;
}
```

Reload your nginx configuration and change directory to your Lets Encrypt installation. The tool doesn't support automatically configuring Nginx yet so we'll use it to just generate a certificate and install it manually ourselves.

The command lets you include multiple domains into a certificate, we'll use it in this example to create a certificate for the main domain and for the sub-domain.

[10] https://letsencrypt.org/

```
./letsencrypt-auto certonly --webroot -w /usr/share/nginx/scaling -d scaleyourphp.si\
te -d www.scaleyourphp.site -w /usr/share/nginx/scaling/static -d static.scaleyourph\
p.site
```

There are rate limits[11] imposed by Lets Encrypt but I don't think you're likely to hit them in the normal course of usage.

The certificate expires in three months, but it is very easy to renew it. I created a cron job to run a refresh script once a week. I used the script from the "Getting Started" section of the Lets Encrypt website:

```
#!/bin/sh
service nginx stop  # or whatever your webserver is
/opt/letsencrypt/letsencrypt-auto renew -nvv --standalone > /var/log/letsencrypt/ren\
ew.log 2>&1
LE_STATUS=$?
service nginx start # or whatever your webserver is
if [ "$LE_STATUS" != 0 ]; then
    echo Automated renewal failed:
    cat /var/log/letsencrypt/renew.log
    exit 1
fi
```

[11] https://community.letsencrypt.org/t/rate-limits-for-lets-encrypt/6769