

The Mysteries Of Asynchronous Processing With PHP - Part 1: Asynchronous Benefits, Task Identification and Implementation Methods

Imagine a world where clients will give up on receiving responses from your application in mere seconds, where failed emails will give rise to complaints and lost business, where there exist tasks that must be performed regularly regardless of how many requests your application receives. This is not a fantasy world, it's reality. In the real world your application must be responsive, reliable and capable of recovery from errors. These are obvious needs but all too often applications fail to realise them. Sometimes, developers even fail to realise they should even be concerned about them.

To offer an opening real-world example, I'll borrow from a recent discussion I had concerning the Pubsubhubbub Protocol. If you are unfamiliar with Pubsubhubbub (PuSH), it's a protocol which implements a publish-subscribe model where the publishers of RSS and Atom feeds can "push" updates to a group of Subscribers. The pushing is handled by an intermediary called a Hub which is pinged by the Publisher when they update a feed, and which then distributes the update to many Subscribers using a Callback URL they each have declared.

In that discussion, the original poster was having a problem. Whenever a Hub sent his Subscriber implementation an update, it seemed to do it repetitively for some mysterious reason. Eventually, the problem was identified. The Hub implements a five second timeout. If, after five seconds, the update request was not completed because the Subscriber failed to send a valid response, it was assumed to have failed. The Hub would then attempt it again, and again, until finally its configured number of retries was used up.

Why was the five second timeout being exceeded by the Subscriber? What was taking it so long in returning a response and finishing the request? You see, the Subscriber was not simply acknowledging the receipt of an update as demanded by the protocol, it was actually processing the entire update for its own use including a number of potentially expensive database operations before it completed the request. This was taking more than five seconds.


Here's the problem in a nutshell. The Subscriber was performing work that had absolutely nothing to do with returning a response to the Hub and it was having an impact on the time it took to complete the request. The Hub couldn't care less about the Subscriber's processing, it was expecting a quick confirmation that the update was received. Instead, the Subscriber was effectively making it wait while it did something completely unrelated to that response. Using Asynchronous Processing, the Subscriber should have offloaded the feed processing elsewhere leaving it free to quickly respond to the Hub.

What is Asynchronous Processing?

Asynchronous processing is a method of performing tasks outside the loop of the current request. Basically, you offload the task to another process, leaving the process serving the request free to respond quickly and without delay. Of course, not all tasks are caused by a request. Some can be performed without a request trigger, like some forms of maintenance or log parsing.

Implementing asynchronous processing can take a few directions:

1. A parent process can spawn a child process to complete a task in the background allowing the parent process continue uninterrupted.
2. You could add tasks to a Job Queue (or even Message Queue) relying on a background daemon or scheduled process to perform batch processing of outstanding tasks in the queue.
3. You could simply have a scheduled standalone task without the queue, and which is performed regardless of what requests are received.

There are, I'm sure, many more variations. Most readers will recognise at least one of these (hint: cron ).

Once you understand the nature of asynchronous processing you can find many uses for it in the most unlikely of places.

What Problems Does Asynchronous Processing Solve?

Our example demonstrates that resource intensive tasks can be detrimental to responsiveness, so much so that it can become detrimental in turn to the client, whether it be a machine applying a configured timeout and being forced into retrying the same request over and over, or whether it be an actual person who has to stare at a blank page as the seconds tick by.

Resource intensive tasks are not the only ones worth applying asynchronous processing to, though they are likely the most obvious given their impact on clients. Most tasks worth offloading can be grouped into categories:

1. Tasks which are resource intensive, i.e. needing a lot of CPU cycles or memory to complete which will add to server load and delay client responses.
2. Tasks which are time consuming but not necessarily resource intensive. These may include database operations, HTTP requests, the use of external web services, and other operations which can suffer delays from network latency or external problems out of our control.
3. Tasks which must be completed regardless of errors. For example, sending emails like signup confirmations or order confirmations. If a first attempt fails (for whatever reason), they may need to be attempted many times before either succeeding or being reported or logged for attention. Obviously, attempting these just once within a request cycle is prone to error - if it fails during the request, will it ever be attempted again? What if your mail server is offline for an extended period?
4. Tasks not triggered by requests. If it needs to be performed, but is not triggered by a HTTP request, then it probably needs to be scheduled or manually added to a job queue somewhere.

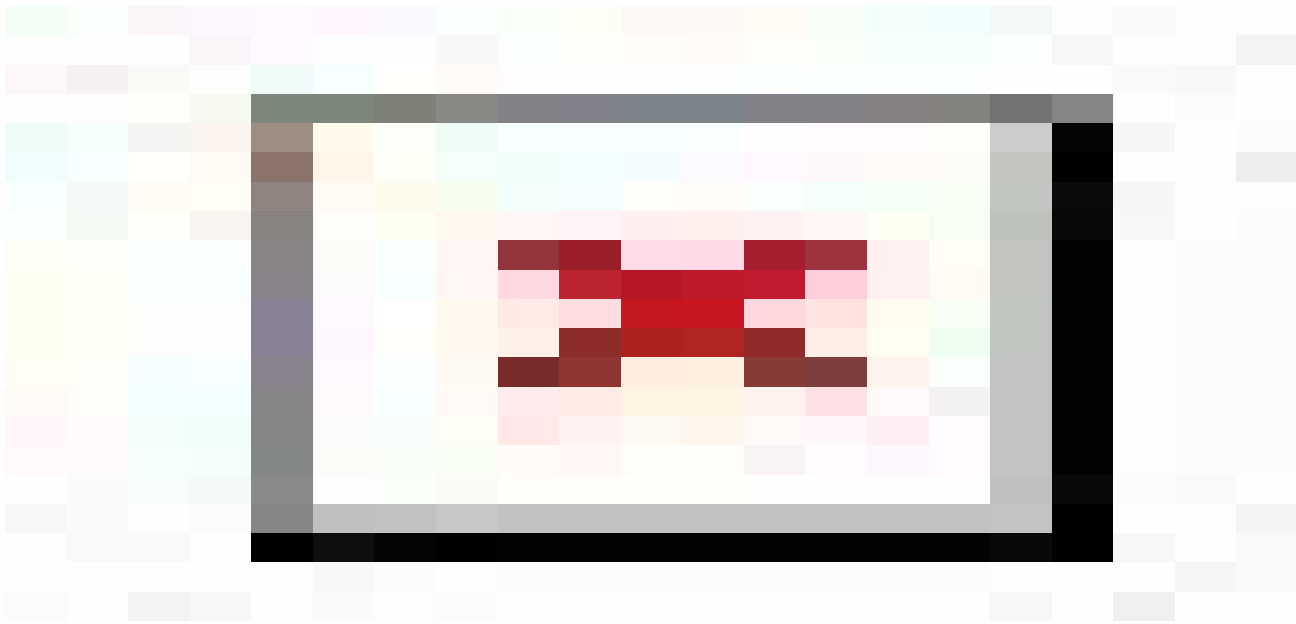
If you can categorise any task in your application within those loose categories, then you have identified a potential candidate for asynchronous processing. If such tasks are presently performed during an application request, you just need to pass one additional test - the completion of the task should not be required in order to return a response. Sending emails, for example, can be done in the background and will not effect the response - it doesn't have an impact on any dynamic data passed to a view or template, for example.

Implementing Asynchronous Processing: Task Identification, Separation and Reusability

So, we've worked through the thought process and theory of asynchronous processing. Before we run off and implement some examples, we first need a task! Once it's identified, we then need to separate it from the application so it can be processed as an independent unit of work. To add to this, we should also make sure it's reusable, essentially returning to our Object Oriented basics. The task should be implemented as a class, or set of classes, so we can execute it with different parameters as easily as possible. This may not have been its original structure. For example, it may simply have been a big procedural script hiding out in an application controller somewhere (very very common), or even the application's service layer.

Let's stick with a prior example, our Pubsubhubbub Subscriber. We'll assume, for now, that the most appropriate method of asynchronous processing relies on spawning a background PHP process to operate on the feed update, leaving the parent process free to return a response quickly. The task to be made subject to background processing is therefore anything to do with processing the feed update. We can show both

alternatives in a simple diagram.



Now that the task is identified, it needs to be separated. This involves taking all steps that the task performs and adding them to an isolated script, effectively a PHP file executed from the command line using the "php -f" command. This does not mean that task must be procedural! It should remain as object oriented as possible. Here's a sample PHP file showing a simple task and demonstrating how it's called from a script.

```
<?php
myTask::perform();
class myTask {
    public static function perform()
    {
        echo "Performing a task...";
    }
}
```

Simple really. Once the perform() method is called, you can use Object Oriented Programming as usual.


One final piece to remember is that tasks should be reusable. You may start by calling this in a separate child process, but that may be migrated to a Job Queue or a schedule. The task needs to be agnostic as to its calling method. This means that it should be capable of accepting configuration/parameters from any source. In many cases, you'd simply wrap the task in a supporting framework. Besides configuration options, there is enabling autoloading, bootstrapping required dependencies, etc. In fact, each task would have something akin to a bootstrapping process just like the main application would rely on from whatever framework it depends on.

In a sense therefore, we're comparing tasks to actions on a controller. They are very similar.


Somewhat related to reusability is another concept of breaking down tasks themselves into their most relevant components. For example, let's say your task is described as follows:

When a User's registration details are stored, attempt to send them an activation email up to five times before delegating any subsequent attempts to a job queue.

To explain the task, activation emails are time sensitive. A user will likely register, and immediately check their email. They may even refresh their inbox a few times. Because it's time sensitive, we may start by using a child process spawned from the parent to attempt the emailing immediately. After five attempts, the child process aborts the task and perhaps marks it for future processing by a scheduled scripted job queue (activation emails are important enough that we should keep trying to send them until continued failures prove a bigger problem exists).

At first, we might be tempted to add a Task which loops over an email attempt five times. Wrong! The looping is a separate task component. The actual email attempt is the core component. It's that core component we want to make reusable. The looping may instead be implemented by a Task Manager which will attempt the email task five times. Okay, that might be a too simple example, but it shows a point. The looping and the emailing can be thought of as separate components. In another situation, perhaps the task does two mutually exclusive things. There again, we can break the apparent task into two separately reusable tasks. Just keep thinking in terms of OOP and you won't go wrong 

Conclusion

In this first part of my series on Asynchronous Processing with PHP, we've covered a lot of theory concerning why such processing is needed, how it could be implemented, and how to think of tasks in terms of being separate and reusable. So I guess I'll let you turn that around in your head for a day or two before I hit you with Part 2 

The main message is important. Asynchronous Processing is one of those fundamental areas of knowledge any programmer, even in PHP, needs to know about. It's been my experience that developers often see it as some arcane craft practiced by a handful of hardcore PHP developers. This completely untrue. Asynchronous Programming is actually very easy to understand, and very easy to implement as we'll see in the next part of this mini-series where we'll look at an example using the Zend Framework.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 14:28

Nice introduction. I will look forward for the upcoming posts in this series.

Tell me you will cover daemons in the future posts. 

Anonymous on Sep 27 2009, 15:32

Looking forward to the next one. BTW, will you be finishing off the DI series? Anonymous on Sep 28 2009, 01:50

I'd like to preempt the next article by recommending Gearman + the Gearman PECL extension for your job queue needs. I've used GM to build spiders, billing software, and offload other long-running tasks. Looking forward to your recommendations! Anonymous on Sep 28 2009, 18:49

This is good stuff and timely for me. I'm looking forward to part two. Anonymous on Sep 28 2009, 19:05

Ditto that, I'd love to see part two. Anonymous on Sep 28 2009, 21:21