Preface

Welcome to the early access version of my Asynchronous PHP book. I am so glad you've chosen to learn about this topic, and support my work in the process. I aim to produce the highest-quality, most up to date reference on the topic. If you spot an error, or would like to me to explain something better – please do not hesitate to contact me (via email and via twitter).

> There's no better time for me to be improving the book, and no way for me to know what needs to be improved, unless you share your experience with me. I'll listen and respond to every bit of feedback, big or small!

This is a work in progress. That means that the chapters you see here are mostly complete, but they are not all the chapters I plan to add to the book. I've completed most of the server side of things, and I believe you will find this immensely useful as-is. But we live in the real world, and need to build all kinds of applications. Those include applications with HTML front-ends and Javascript.

I am still writing the other half of the book, which includes these things, as well as an in-depth look at web sockets (in PHP). I encourage you to start building that side-project you've been wanting to build, and to look forward to the updates to this book.

Where things are

Chapters 2 through 8 have been copy-edited, by the talented Kara Ferguson. I still have work to do, in the remaining chapters you see before you. If you spot any spelling and/or grammar errors; please let me know.

All of the chapter names are provisional. I haven't quite settled on the final versions, and until I do they are subject to change.

No images are present. The book will include images, but I have been collecting them over the course of a couple years, so they look too different from each other for my liking. I will go through each chapter and re-create the images at the end.

All the code has been tested, and is up on Github. Please post any syntax errors you find, are issues there. It would be super helpful if you could also post the chapter and/or section name, and a string to search for. The layout is constantly changing, and search strings are just about the only way I know how to find places to correct.

Introduction

Asynchronous PHP architecture is a departure from the traditional PHP you may already be familiar with. If you're reading this book, without already knowing some PHP, you may struggle with some of the syntax I show you.

On the other hand, you'll be blissfully unaware of what typical PHP applications look like, at a code level. You'll be ignorant to the common pitfalls and ugly hacks. You'll not have the struggle of unlearning bad habits.

As developers, we often get to choose which programming languages we'll use to solve our problems. Sometimes our choices depend on what we're most comfortable with. Sometimes our choices depend on what the programming languages we're deciding between are good at doing.

Starting anew

Nowhere are these decisions more apparent than the subject of Asynchronous PHP programming. Let's consider an example. Let's imagine your boss asks you to build a real-time news feed, to cover a live event. Your boss doesn't care which programming language you use, so long as thousands of visitors can receive instant updates, without refreshing their browsers.

How would you build such a thing? Perhaps you'd head over to a forum of programmers, and ask their advice for the programming language you should use. That's not a bad idea. They're likely to tell you which programming languages they think will be better suited to the task. You'll have to figure out how much of their choice depends on the strengths of the language and how much of it depends on their comfort with the language.

NodeJS

These days, a common answer is to use something like NodeJS. It's a JavaScript platform, which is event-driven and has a massive community. So there are some event-driven design patterns which have a mature footing within the language. And many people have developed libraries in these design patterns, for you to use.

That's not a bad thing. It's your professional duty to pick the right tool for the job. But every tool has its trade-offs. If you already have a team of PHP developers, or an existing PHP codebase, adding a new JavaScript platform the the mix may be costly. The developers will need time to adjust to the new programming languages and environment. You'll need time to re-write the parts of your code, to interact with the JavaScript code you're about to write.

PHP

How about building the news feed in PHP? You may ask this of the forum members, and would be surprised by the answers you hear. PHP is not known for its ability to multitask. It rose to popularity through a systematic approach of embedding itself inside popular web servers. It was designed to be an easy replacement for C and Perl, in a time when web development was the domain of very few developers.

Traditional PHP scripts thrive through the implicit flow of HTTP requests and responses. They're meant to be short-lived, and to dump every variable and resource the moment the response is sent to a browser. They're not designed for persistent connections, or to act as their own web server.

That's not to say PHP can't be used to do any of these things, but popular tradition has moulded the community into a pattern of thinking that has been hard for me to shake.

Having courage

I have a simple goal. I want to help a new generation of PHP developers discover that PHP can do just as much as platforms like NodeJS. We need to cast off the old ways of thinking, and learn how to build simple, elegant, asynchronous PHP applications.

There are a few reasons why I think this is a good idea. As we'll see, there are practical reasons; like allowing more people to use our sites and services at the same time. These gains save on the cost of new hardware and make full use of the hardware which we already have.

There are also long-term reasons for why we might want to build these kinds of applications. The language already includes basic tools to allow for this kind of architecture. We're going to see what those tools look like, but it's important to realise that they're still very basic.

It's no longer possible to argue that the language is incapable of asynchronous programming. Nor is it reasonable to argue that it's not meant to grow in this direction. It was created as a templating language, and yet we're using it as a general purpose programming language.

We can choose to ignore parts of PHP that allow asynchronous architecture, or we can understand how they work. Many people have done the latter, and built libraries and extensions to further compliment this toolset.

If you needed a database, and had to install an extension to use it, that wouldn't be a difficult choice. If you needed higher concurrency and wanted a faster request-response cycle, wouldn't it be just as simple to install an extension or use a built-in function?

Clarifying terminology

There are a few terms we need to understand, before we dive into code. The first is that traditional PHP programming is synchronous. That is, each program is essentially a list of instructions, executed one after the other:

[synchronous diagram]

This is the simplest and, depending on how to look at it, the slowest way of building applications. It's simple because the order is predictable. Once you understand what each instruction does, you know exactly how the script will execute.

But there's another way of building applications, that environments like NodeJS have made popular. It's called reactive, event-based, or asynchronous programming. Scripts are still written as a list of instructions, but the order in which they are executed is no longer predictable (for non-trivial scripts).

Asynchronous programming adds to this list of instructions, the idea of events. A database query is no longer just an expected execution time of `120ms`, but when complete it can trigger an event. Further instructions can be executed and completed in the time it takes a database query to complete, because they happen in a kind of cooperative-processing execution state:

[asynchronous diagram]

It's a kind of cooperative-processing because single-threaded, single-process environments can't really perform multiple tasks at the same time. It's still more efficient, since synchronous processing forces the processor to wait until the completion of things like network requests and file system operations, before returning to the real purpose of the script.

Asynchronous execution works around these waiting periods, since the processor is allowed to do work while the "waiting operations" are periodically polled for updates. We've seen this time and time about, in the browser. We could, for instance, make all Ajax requests synchronous. But then the browser wouldn't be able to respond to user interaction or even scroll, while waiting for the network request to complete.

Instead, we can used event-based JS to *tell us* when the network request is done, so that we can respond accordingly. And in the meantime we'd have been able to do other things with the processor.

To do many things at the same time, we need a multi-threaded or multi-process environment. This is called *parallel* execution, because multiple tasks can be run in a truly parallel state:

[parallel diagram]

Not only is this possible, using built-in PHP functions: it's made easy using libraries and extensions. If you needed to communicate with an IMAP server, you wouldn't build the socket client library from scratch, would you? It's far easier to install an extension for that. The same can be said of databases, message queues, cache servers etc.

In the same way, we can install Pthreads, PCNTL, and AMPHP. These extensions and libraries smooth over the differences in server hardware and software, to the point where we can use multi-threading and multi-process scripts.

> Don't worry if you don't know what these are. There are many chapters dedicated to the individual libraries and extensions mentioned. I'm not even linking to them here because learning exactly what they are, or how they differ, is just a distraction at this point.

There are some immediate benefits, as already explained through synchronous vs. asynchronous Ajax requests. Being able to do more things while "waiting" lets us serve more HTTP requests and perform more "business logic" than before. A document conversion service can convert more documents, a machine learning API can analyse more data.

It also let's us move slow processes out of places where the biggest concern is how responsive the application is to users. Why write to a database or send an email, forcing the user to wait for a browser response? We could just as well push these expensive tasks to another thread, process, or server.

NodeJS, Ruby, Python, and Go all have these abstractions built-in. PHP may not initially have been designed as a general purpose language but that's how we're using it. PHP may not have been designed with multi-threading and multi-process built in, but that's how we can use it.

Getting started

A few years ago, I started to work on a few SilverStripe framework modules. I was employed by SilverStripe, and I noticed something interesting.

There were about 2,000 modules tagged specifically to work with the framework, and though many of them did interesting things, very few of them were what some consider to be a high standard.

There were some obvious omissions, like missing tests and documentation. Then there were less obvious (or agreed upon) things missing; like PSR conformance, licences, and codes of conduct.

One day I started working on a collection of scripts to inspect these modules, and suggest changes to them. I started with a script which would format the source code to conform to PSR-2. It would store alongside the module's name and path, whether or not the module conformed to PSR-2. It would also get a pull request ready, and provide me with a link so that I could review and submit pull requests as needed.

As time went by, I added more and more scripts to this helper application.

> I wasn't asking people to opt in to Helpful Robot's suggestions. I figured, since I was manually reviewing every pull request, before submitting it, that I was just being a very active community contributor. The Github team and terms of service agreed with me.

There were many of these pull requests floating around, and module maintainers frequently maintained multiple SilverStripe modules. I didn't want to associate my real name with a large number of pull requests (in case module maintainers became irritated with new activity on dormant modules). Thus I created a new Github account, and named it Helpful Robot.

Over the space of 6 months, Helpful Robot submitted thousands of pull requests. The percentage of merged pull requests is currently around 46%. That's a fantastic number of contributions to community modules, and something I'm very proud of.

Despite the effort, Helpful Robot has always remained a relatively small collection of scripts. I've rebuilt the scripts a few times, but I've longed to make it into something more.

In this book, we're going to remake Helpful Robot into a continuous integration service to be reckoned with. It'll still retain its core inspection and suggestion functionality, but this time it'll have a face. A charming, robotic face.

Installing PHP and MySQL

Nothing ages a book faster than a list of installation instructions, for each operating system the reader could possibly have. I'm not going to do that here.

> I'm also assuming, since you're reading a book about asynchronous PHP, that you have some experience in PHP. And that you probably

> already have a working development environment.

There are a couple books you could refer to, for specific setup information. The first is written by this book's technical reviewer: Bruno Škvorc, Jump Start PHP Environment. It covers many topics; from Integrated Developer Environments to Vagrant to Git.

Bruno's book does a great job of explaining the basics, and getting you started with a workable development environment. If you'd like details about the finer aspects of Vagrant configuration, you should also definitely check out Erika Heidi's Vagrant Cookbook.

Later on, we'll begin handling HTTP requests and making database queries. We'll deploy Helpful Robot to a server, so that others can access its API and interface. For this, we'll need a virtual private server; preferably one that is geared to support asynchronous architecture.

I've recently found Platform.sh to be a good fit for asynchronous PHP applications. We'll see why that is, in later chapters. In the meantime, feel free to create an account there. They offer free trials and have good documentation over at docs.platform.sh.

Making something useful

The first few iterations of this book approached each topic with clinical detachment. They focussed on a specific library or extension, and tried to cram 100% of them into the same sample application.

After a while, I realised that to be truly useful; I had to do things in the real world. I had to describe things like "responding to HTTP requests" and "authenticating users" – all in a non-blocking way – so that others could see how to use asynchronous PHP code in their applications.

As you read this book, you'll see how to build a real application. We'll dive into creating an HTTP server, learn about generators and promises, connect to a database. You'll be able to copy the codebase, and make space for your application logic (by deleting some of my code). It'll be clear how to reuse these concepts, libraries, and extensions in your day-to-day development.

Summary

In this chapter, we looked at how programming languages are often chosen for projects. We learned about the differences between synchronous, asynchronous, and parallel execution; and how asynchronous and parallel execution can help us to minimise waiting time (both for hardware and users).

We also discovered what application we're going to build, through the course of this book. It's important that we put the things we learn in this book into practise. Helpful robot gives us the opportunity to do that, and have a useful application at the end of our time together.

Finally, we looked at how to get a PHP environment up and running. This is the most boring part of the work we're about to do, but it's essential that we have a working environment.

In the next chapter, we're going to learn about asynchronous tools provided to us by the core language. There are a couple functions we can put to interesting and effective use, as we start to uncover what it means to build asynchronous PHP applications.

On the server

Creating an HTTP server

We begin our journey, into asynchronous PHP architecture, by building a web server. In this chapter, we'll take a look at a library (or set of libraries) designed to simplify the process of responding to HTTP requests asynchronously.

Don't panic! The libraries will do all the heavy lifting. As we progress, we'll learn about what they're doing under-the-hood. The aim of this chapter is just to get the ball rolling.

> The code for this chapter can be found on Github.

Installing Aerys

The library I've been speaking about is called Aerys. It's part of the Amp open source project. We can install it with:

```
composer require "amphp/aerys:^0.4.7"
```

> Later, I'll talk about more recent versions of Aerys, and what to expect in them.

This will take a moment but should install everything we need to get started. Next, we need to create a config file:

```
use Aerys\Host;
use Aerys\Request;
use Aerys\Response;

use function Aerys\root;
use function Aerys\router;

$host = new Host();
$host->expose("*", 8888);

$host->use($router = router());

$router->get(
    "/", function(Request $req, Response $res) {
        $res->end("hello world: get");
    }
);

$host->use($root = root(__DIR__ . "/public"));
```

> This is from `config.php`

Every Aerys application begins with a `Host`. This is the proxy object which handles middleware (like routing and serving static files). To this, we attach a public folder middleware (via `root`) and a router. We use `*` to specify this server should respond to all domains. Similarly, `8888` indicates we want the server to listen for requests on that port.

This router works similarly to other PHP frameworks you may be familiar with. It's a collection object; storing route definitions. We can add a single route, matching the `/` path to a callback. This is the route that will be matched if someone enters the domain (without any additional text) in a browser address bar.

> We should register the public folder middleware after the router. It checks for the existence of static files, and we want this check to happen after all the routes have been checked, as it's most likely that a route will match before a static file does. If we reverse the order, the server is going to search the public folder, for a matching file, for

> every request.

It's common to include the Composer autoload file into any daemon-like script. We're going to run this script as a daemon, but we're going to load it through Aerys' daemon script, with:

```
vendor/bin/aerys -d -c config.php
```

This command is tricky to remember, so I recommend adding a Composer script for it:

```
"scripts": {
    "serve": "vendor/bin/aerys -d -c config.php"
}
```

> This is from `composer.json`

Now, we can start the daemon by typing `composer run serve`. Once that's running, you'll be able to open `http://127.0.0.1:8888` in a browser. You should see "hello world."

How it works

There are many moving parts to explore; we won't go into detail about all of them. At a high level, the server starts listening at the specified port. When a browser initiates a connection (whether directly or via some forwarding layer), the server starts the usual HTTP handshake.

Aerys opens tunnels for the request and the response (we get handles to these from the route parameters), and parses the request. After this, we get access to details like the headers and body of the request.

> We'll work with headers, query strings, and request bodies later.

Inside the closure, we can perform whatever actions are required to fulfill the request. And we can then respond, using `$response->end()`. This sets the appropriate response headers and sends a string to the browser.

This is a long-running process, which means we have to think differently than we normally would. In traditional frameworks (even cool ones like Laravel), we get to think of requests and responses as shared, global objects. We can fetch them from a dependency injection container, and manipulate them however we choose.

In this architecture, we need to treat every object as if it could belong to someone else. We must assume global variables may be overwritten (so we avoid them). We must avoid sharing requests, responses, or resources of any kind. There are exceptions to this, which we'll learn about later.

Different routes

Let's take a look at the different kinds of routes we can define. We've already seen the simplest version:

```
$router->get(
    "/", function(Request $request, Response $response) {
        $response->end("hello world");
    }
);
```

This is from `config.php`

The router acts as a collection. One to which we can add different routes; matching different request methods, paths, and closures. It also matches a request signature to the first route it can, or triggers error responses.

I've used the `get` method, but this is really just an alias to another:

```
$router->get(
    "/", function(Request $req, Response $res) {
        $res->end("hello world: get");
    }
);

$router->post(
    "/", function(Request $req, Response $res) {
        $res->end("hello world: post");
    }
);

$router->put(
    "/", function(Request $req, Response $res) {
        $res->end("hello world: put");
    }
);

$router->route(
    "DELETE", "/", function(Request $req, Response $res) {
        $res->end("hello world: delete");
    }
);
```

This is from `config.php`

Each request method is an alias to `$router->route()`. You can use the shortcuts, or stick to `route`—it makes no difference. The important thing to realize is the router can have many different routes. They can have the same path and different methods, or the same method and different paths. It isn't possible to have the same path and method, but a different closure.

To see these changes in action, we need to restart the Aerys server and use:

```
curl -X PUT http://127.0.0.1:8888
⇒ hello world: put
```

In the next chapter, we'll learn how to get around the constant need to restart the server just to see our code changes take effect.

Different parameters

Aside from query string parameters and HTTP request bodies, it's often useful to define routes which need parameters. Perhaps we want to create an endpoint to view the details of a product: `/products/3/details`. We can define these kinds of routes using the following syntax:

```php
$router->get(
    "/{name}",
    function(Request $req, Response $res, array $params) {
        $res->end("hello " . $params["name"]);
    }
);
```

> This is from `config.php`

Parameters are defined as names within `{` and `}`, and are collected into an associative array. There are far more complex things you can do with these, including matching regex patterns and defining optional parameters, but you'll have to dig into the FastRoute documentation to see how those work.

> I've built large applications without ever needing to tinker with the more complex types of parameter definitions. I find it's much easier just to do any validation once I have the string parameter values.

Different files

Putting all the routes in the same file as the rest of the server config can be messy. I like to keep them in a file called routes, which returns a routing closure:

```
use Aerys\Request;
use Aerys\Response;
use Aerys\Router;

return function(Router $router) {
    $router->get(
        "/", function(Request $req, Response $res) {
            $res->end("hello world: get");
        }
    );

    // ...
};
```

This is from `routes.php`

Then, to load the routes, we need to import this file and pass it an instance of the router:

```
// ...

$host->use($router = router());

$routes = require __DIR__ . "/routes.php";
$routes($router);
```

This is from `config.php`

Now we can add many routes, and not touch the config file. We can even load multiple routes files. However, there's still the question of what we do to keep the closures tidy. If we need to define 100 routes, and each route closure needs to process some request data or construct some response, the routes file is going to end up in the same messy state as the config file would have.

So, instead, we can create files for each route closure, even going so far as to create classes for them. Let's autoload these classes, by modifying our `composer.json` file:

```
"autoload": {
    "psr-4": {
        "HelpfulRobot\\": "app"
    }
}
```

This is from `composer.json`

I'm not going to go into detail about what this does, except to say Composer has great documentation explaining it. We need to remember to rebuild the autoloader:

```
composer du
```

`du` is a shortcut for `dump-autoload`

Now, we can create a function/class for responding to a route. I like to call these responders, but you can call them whatever you like:

```
namespace HelpfulRobot\Responder;

use Aerys\Request;
use Aerys\Response;
use Closure;
class IndexResponder
{
    public function closure()
    {
        return Closure::fromCallable([$this, "run"]);
    }

    public function run(Request $req, Response $res)
    {
        $res->end("hello world: responder");
    }
}
```

This is from `app/Responder/IndexResponder.php`

Before I explain what this does, let's just look at how it is used:

```
use Aerys\Request;
use Aerys\Response;
use Aerys\Router;
use HelpfulRobot\Responder\IndexResponder;

return function (Router $router) {
    $router->get("/", (new IndexResponder())->closure());

    // ...
};
```

This is from `routes.php`

The `Router` accepts closures for route definitions. If we give it a closure, irrespective of where that closure is stored or generated, then the router will continue to work in the same way. `IndexResponder` has a couple of methods: one which responds to a request and another which turns that instance method into a closure.

This will be a common theme in our application, so let's abstract the closure generation:

```
namespace HelpfulRobot\Responder;

use Closure;

trait Responds
{
    public function closure()
    {
        return Closure::fromCallable([$this, "run"]);
    }
}
```

This is from `app/Responder/Responds.php`

We've extracted the closure generator to its own trait. We'll put more code in this trait as we progress, but this is all we need for now. Let's use this in our `IndexResponder`:

```php
namespace HelpfulRobot\Responder;

use Aerys\Request;
use Aerys\Response;

class IndexResponder
{
    use Responds;

    public function run(Request $req, Response $res)
    {
        $res->end("hello world: responder");
    }
}
```

This is from `app/Responder/IndexResponder.php`

This is slightly more code than just putting our closures in the routes file, but it's also a lot cleaner. We can create new responder objects, passing constructor parameters to them, and still use the router in much the same way as before. What's more, each responder is isolated from the others, so changes in one don't impact others.

Responding to errors

It's common, in PHP frameworks, for routers to play a part in the handling of HTTP errors. The underlying routing library, FastRoute, returns 404 (missing) or 405 (method not allowed) errors, depending on what routes have been defined.

Unfortunately, defining how these should be handled is not simple in Aerys. We need to learn about middleware first, and we're getting to that much later. For now, we'll have to be content with the default error pages Aerys renders.

> While it's strictly possible to customize the response for 404 errors, without middleware, I want to tackle all the error responses together.

Summary

In this chapter, we looked at how to set up an asynchronous HTTP server. This would typically be the job of Apache or NGINX, but if we want our application to be truly asynchronous, we need to handle everything from request to response.

We also learned about the different types of routes we can define, and also how to define route parameters. We separated the routes into their own files and even looked at how to separate responders out into their own function/ class files.

In the next chapter, we'll discover ways to make our development workflow easier.

Watching for changes

In the previous chapter, we took the first steps towards creating a real application. We set up an asynchronous PHP HTTP server script, which we passed to Aerys. Aerys, in turn, ran the server like a daemon (or long-running process).

One of the challenges we saw, is that any changes we make to the code require a restart to take effect. We can't use the traditional change, refresh, repeat cycle, without also adding a restart-the-daemon step in-between.

In this chapter, we're going to fix that problem. We're going to create a set of tools to watch for code changes and restart the server immediately.

> The code for this chapter can be found on Github.

To recap

We don't run our server script directly. We pass it along, to the Aerys executable, which maintains the long-running server process. That executable expects us to create a `Host` class, and to give it some middleware so it can respond to requests.

We've even got a Composer script to launch the server with a simple command:

```
composer run serve
```

It's easy to restart, but not easy to remember to restart. Let's remove this distraction.

Setting up the watcher script

We're going to set up a few scripts, beginning with one which will watch for changes to our code. These are not going to look like typical scripts, and they'll begin with a strange bit of code:

```
#!/usr/bin/env php
<?php

require_once __DIR__ . "/../vendor/autoload.php";
```

This is from `scripts/watch`

We want this script to work similarly to the Aerys executable—to be able to run without us specifying the PHP executable. `#!/usr/bin/env php` tells the operating system to send this script to the default PHP executable, but to run it we also need to allow this script to be executed:

```
chmod +x scripts/watch
```

The next step is to install a file watcher library:

```
composer require --dev "yosymfony/resource-watcher:^1.2"
```

Then, we can define a set of folders to watch:

```
use Symfony\Component\Finder\Finder;
use Yosymfony\ResourceWatcher\ResourceWatcher;
use Yosymfony\ResourceWatcher\ResourceCacheFile;

$finder = new Finder();

$finder->files()
    ->name("*.php")
    ->in(__DIR__ . "/../app");

$cache = new ResourceCacheFile(
    __DIR__ . "/.dev-changes.php"
);

$watcher = new ResourceWatcher($cache);
$watcher->setFinder($finder);

while (true) {
    $watcher->findChanges();

    if ($watcher->hasChanges()) {
        print "files have changed" . PHP_EOL;
    }

    usleep(100000);
}
```

This is from `scripts/watch`

We begin the script by defining a new Symfony `Finder` object. This looks at all files ending in `.php`, within the `app` folder. For now, that's just our `IndexResponder` and `Responds` files, but it will grow to encompass our whole application.

We define the `ResourceWatcher` and `ResourceCacheFile`, which will monitor for file changes and store them to the filesystem. Then, in an infinite loop, we check for file changes. If there are changes, we can figure out how to respond to them. We also add a sleep (one tenth of a second) as we're unlikely to receive updates more frequently than that.

Managing processes

Our watcher script is going to struggle to restart itself. It'll be much easier to run the server daemon in a separate process, and manage that through the watcher script. For that, we're going to use `exec` and a handful of helper functions:

```
namespace HelpfulRobot\Process
{
    function start($tag, $command, $log = null)
    {
        $file = $log ?? sha1($command);
        $path = "> {$file} 2> {$file}";

        exec("{$command} tag={$tag} {$path} &");

        return identify($tag);
    }

    function identify($tag)
    {
        exec("ps -ax | grep tag={$tag}", $lines);
        $parts = explode(" ", trim($lines[0]));

        return (int) $parts[0];
    }

    function stop($tag)
    {
        $pid = identify($tag);

        if ($pid < 1) {
            return;
        }

        exec("kill -9 {$pid}");
    }
}
```

> This is from `helpers.php`

These functions are the basis for all the synchronous process management we're going to be doing. They're non-blocking, but because of that, they don't allow us to see what any of the output is (of the `start` function specifically).

> `exec` is a powerful and dangerous tool. It's widely decried by the PHP community, but the truth is other language features can be abused as well. It's just that the damage `exec` can do is greater than most language features.
>
> My caution to you is two-fold. First, don't blindly believe everything you hear. Functions like `exec` can be abused, but they can also be safely used with great benefit. Second, treat functions like `exec` as though they're made of radioactive bees. Don't allow any user input to get anywhere close to them.

`start` executes the provided command, appending an identifiable string and building an output clause. This output clause (which will substitute a safe filename in place of a missing `$log`), along with `&`, makes the command return immediately.

`identify` looks for the process identifier string, returning the process ID. This is a bit hacky since it depends on the exact formatting of the `ps` command. These flags work well on macOS (where I'm developing) but may need to be adjusted for Linux.

> As far as I know, this will not work directly on Windows, but it should work on the Windows Subsystem for Linux (on modern versions of Windows).

Finally, the `stop` function takes a tag, identifies the corresponding process ID, and kills it. These methods will allow us to start "named" processes (without knowing the resulting process ID beforehand) and stop them just as easily.

So, how do we use these?

```
// ...

require_once __DIR__ . "/../helpers.php";

use function HelpfulRobot\Process\start;
use function HelpfulRobot\Process\stop;

start(
    "helpfulrobot",
    "vendor/bin/aerys -d -c config.php",
    "aerys.log"
);

while (true) {
    $watcher->findChanges();

    if ($watcher->hasChanges()) {
        print "Restarting the server" . PHP_EOL;

        stop("helpfulrobot");

        start(
            "helpfulrobot",
            "vendor/bin/aerys -d -c config.php",
            "aerys.log"
        );
    }

    usleep(100000);
}
```

This is from `scripts/watch`

And there we have it. This script will watch for any changes to our `app` code, and restart the server daemon automatically. This is going to make our workflow that much better, as we'll not have to constantly remember to restart the daemon, every time we make a change!

Summary

In this chapter, we looked at how to set up an efficient watcher script to automatically restart the HTTP server daemon, without us having to remember to do it. We're going to reuse these concepts a little later, so it is important for us to get them sorted out now.

In the next chapter, we're going to take a step back from the practical implementation of our application, to learn about generators. We'll see what they are and how to use them to implement strange (and wonderful) new programming paradigms.