

# Chapter 17. Asynchronous PHP

Many basic PHP scripts handle operations synchronously—meaning the script runs one monolithic process from start to finish and only does one thing at a time. However, more sophisticated applications have become commonplace in the world of PHP, so more advanced modes of operation are required as well. Namely, asynchronous programming has quickly become a rising concept for PHP developers. Learning how to do two (or more) things at the same time within your scripts is vital to building modern applications.

Two words come up frequently when discussing asynchronous programming: *concurrent* and *parallel*. When most people talk about parallel programming, what they really mean to say is *concurrent programming*. With concurrency, your application does two things but not necessarily at the same time. Think of a single barista serving multiple customers at once—the barista is multitasking and making several different drinks but can really only make one drink at a time.

With parallel operations, you are doing two different things simultaneously. Imagine installing a drip coffee machine on the counter in the cafe. Some patrons are still being served by the barista, but others can get their caffeine fix from a separate machine in parallel. [Figure 17-1](#) depicts concurrent and parallel operations through the barista analogy.



Figure 17-1. Concurrent versus parallel modes of operation

#### NOTE

There is also a third concept of *concurrent parallel* operations, which is when two work streams operate at the same time (parallel) but also multitask their individual work streams (concurrent). While this composite concept is useful, this chapter instead focuses on the two separate concepts alone.

Most PHP you'll find in the wild, whether modern or legacy, is written to leverage a single thread of execution. The code is written to be neither concurrent nor parallel. In fact, many developers avoid PHP entirely when they want to leverage concurrent or parallel concepts and turn to languages like JavaScript or Go for their applications. Modern PHP, though, fully supports both modes of execution—with or without additional libraries.

## Libraries and Runtimes

PHP's native support for parallel and concurrent operations is relatively new to the language and difficult to use in practice. However, several libraries abstract away the difficulty of working in parallel to make truly asynchronous applications more straightforward to build.

### AMPHP

The [AMPHP project](#) is a Composer-installable framework that provides event-driven concurrency for PHP. AMPHP provides a rich set of functions and objects empowering you to fully master asynchronous PHP. Specifically, AMPHP provides a full event loop as well as efficient abstractions for promises, coroutines, asynchronous iterators, and streams.

## ReactPHP

Similar to AMPHP, [ReactPHP](#) is a Composer-installable library offering event-driven functionality and abstractions to PHP. It provides an event loop but also ships fully functional asynchronous server components like a socket client and a DNS resolver.

## Open Swoole

[Open Swoole](#) is a lower-level PHP extension that can be installed via PECL. Like AMPHP and ReactPHP, Open Swoole provides an asynchronous framework and implementations of both promises and coroutines. Because it is a compiled extension (rather than a PHP library), Open Swoole performs significantly better than various alternatives. It also supports true parallelism in your code, rather than merely concurrent execution of tasks.

## RoadRunner

The [RoadRunner project](#) is an alternative PHP runtime implemented in Go. It provides the same PHP interface you’re used to but ships its own application server and asynchronous process manager. RoadRunner empowers you to keep your entire application in memory and invoke atomic processes in parallel to the application’s execution whenever needed.

## Octane

In 2021, the web application framework Laravel introduced a new project called [Octane](#) that leverages either Open Swoole or Roadrunner to “supercharge your application’s performance.” Whereas framework-level tools like AMPHP or ReactPHP allow you to intentionally write asynchronous code, Octane leverages the asynchronous foundations of Open Swoole or RoadRunner to accelerate the operation of an existing Laravel-based application.<sup>1</sup>

# Understanding Asynchronous Operations

To fully understand asynchronous PHP, you need to understand at least two specific concepts: promises and coroutines.

## Promises

In software, a *promise* is an object returned by a function that operates asynchronously. Rather than representing a discrete value, though, the promise represents the overall state of the operation. When first returned by the function, the promise will have no inherent value, as the operation itself is not yet complete. Instead, it will be in a *pending* state indicating that the program should do something else while an asynchronous operation completes in the background.

When the operation *does* complete, the promise will be either fulfilled or rejected. The fulfilled state exists when things went well and a discrete value is returned; the rejected state exists when something failed and an error is returned instead.

The AMPHP project implements promises by using generators and bundles both the fulfilled and rejected state into an `onResolve()` method on the promise object. For example:

```
function doSomethingAsync(): Promise
{
    // ...
}

doSomethingAsync()->onResolve(function (Throwable $error)
{
    if ($error) {
        // ...
    } else {
        // ...
    }
});
```

Alternatively, the ReactPHP project implements the same [promise specification as JavaScript](#), enabling you to use the `then()` construct that might be familiar to Node.js programmers. For example:

```
function doSomethingAsync(): Promise
{
    // ...
}
```

```
doSomethingAsync()->then(function ($value) {
    // ...
}, function ($error) {
    // ...
});
```

---

#### NOTE

While the APIs presented for promises by both AMPHP and ReactPHP are somewhat unique, they are fairly interoperable. AMPHP explicitly does not conform to JavaScript-style promise abstractions in order to fully leverage PHP generators. However, it does accept instances of ReactPHP's `PromiseInterface` wherever it works with its own `Promise` instance.

---

Both APIs are incredibly powerful, and both projects expose efficient asynchronous abstractions for PHP. However, for simplicity, this book focuses on the AMPHP implementations of asynchronous code as they're more native to core PHP functionality.

## Coroutines

A *coroutine* is a function that can be interrupted to allow another operation to proceed. In PHP, particularly with the AMPHP framework, coroutines are implemented with generators leveraging the `yield` keyword to suspend operation.<sup>2</sup>

While a traditional generator uses the `yield` keyword to return a value as part of an iterator, AMPHP uses the same keyword as a functional interrupt in a coroutine. The value is still returned, but execution of the coroutine itself is interrupted to allow other operations (like other coroutines) to proceed. When a promise is returned in a coroutine, the coroutine keeps track of the promise's state and automatically resumes execution when it's resolved.

As an example, you can leverage asynchronous server requests via coroutines directly in AMPHP. The following code illustrates how coroutines are used both to retrieve a page and decode the body of its response, yielding a promise object useful elsewhere in your code:

```
$client = HttpClientBuilder::buildDefault();

$promise = Amp\call(function () use ($client) {
    try {
        $response = yield $client->request(new Request(
            // ...
        ));
    }

    $promise->onResolve(function ($error = null, $value = r
        if ($error) {
            // ...
        } else {
            var_dump($value);
        }
    );
});
```

## Fibers

The newest concurrency feature in PHP, as of version 8.1, is Fiber. Under the hood, a Fiber abstracts a completely separate thread of operation that can be controlled by your application's primary process. The Fiber doesn't run in parallel to the main application but presents a separate execution stack with its own variables and state.

Through Fibers, you can essentially run an entirely independent subapplication from within your main one and explicitly control how the concurrent operation of each is handled.

When a Fiber starts, it runs until it either completes execution or calls `suspend()` to yield control back to and return a value to the parent process (thread). It can then be restarted by the parent with `resume()`. The official documentation example that follows illustrates this concept succinctly:

```
$fiber = new Fiber(function (): void {
    $value = Fiber::suspend('fiber');
    echo "Value used to resume fiber: ", $value, "\n";
});
```

```
$value = $fiber->start();  
  
echo "Value from fiber suspending: ", $value, "\n";  
  
$fiber->resume('test');
```

Fibers aren't meant to be used directly by developer code but are instead a low-level interface useful to frameworks like AMPHP and ReactPHP. These frameworks can leverage Fibers to fully abstract the execution environments of coroutines, keeping your application state clean and better managing its concurrency.

The recipes that follow cover the ins and outs of working with both concurrent and parallel code in PHP. You'll see how to manage multiple concurrent requests, how to structure asynchronous coroutines, and even how to leverage PHP's native Fiber implementation.

## 17.1 Fetching Data from Remote APIs Asynchronously

### Problem

You want to fetch data from multiple remote servers at the same time and act on the result once they have all returned data.

### Solution

Use the `http-client` module from the AMPHP project to make multiple concurrent requests as individual promises and then act once all of the requests have returned. For example:

```
use Amp\Http\Client\HttpClientBuilder;  
use Amp\Http\Client\Request;  
  
use function Amp\Promise\all;  
use function Amp\Promise\wait;  
  
$client = HttpClientBuilder::buildDefault();  
$promises = [];
```

```
$apiUrls = ['\https://github.com', '\https://gitlab.com']

foreach($apiUrls as $url) {
    $promises[$url] = Amp\call(static function() use ($url) {
        $request = new Request($url);

        $response = yield $client->request($request);

        $body = yield $response->getBody()->buffer();

        return $body;
    });
}

$responses = wait(all($promises));
```

## Discussion

In a typical synchronous PHP application, your HTTP client would make one request at a time and wait for the server's response before continuing. This sequential pattern is fast enough for most implementations but becomes burdensome when managing a large number of requests at once.

The `http-client` module of the AMPHP framework supports making requests concurrently.<sup>3</sup> All requests are dispatched in a nonblocking fashion by using promises to wrap the state of the request and the eventual result. The magic behind this approach isn't just the concurrent nature of AMPHP's client; it's in the `Amp\call()` wrapper used to bundle all of the requests together.

By wrapping an anonymous function with `Amp\call()`, you turn it into a coroutine.<sup>4</sup> Within the body of the coroutine, the `yield` keyword instructs the coroutine to wait for the response of an asynchronous function; the overall result of the coroutine is returned as a `Promise` instance rather than a scalar value. In the Solution example, your coroutine is creating a new `Promise` instance for each API request and storing them together in a single array.

The AMPHP framework then exposes two useful functions that allow you to wait until all of your promises have been resolved:

## all()

This function takes an array of promises and returns a single promise that will resolve once all of the promises in the array have been resolved. The value wrapped by this new promise will be an array of its wrapped promises' values.

## wait()

This function is exactly what it sounds like: a way to force your application to wait for an otherwise asynchronous process to complete. It effectively converts the asynchronous code into synchronous code and unwraps the value contained by the promise you pass into it.

The Solution example thus makes several concurrent asynchronous requests to differing APIs and then bundles their responses into an array suitable for use throughout the rest of your otherwise synchronous application.

---

### WARNING

While you make requests in a particular order, they might not complete in the same order in which you made them. In the Solution example, these three requests might always complete in the same order in which you dispatched them. If you increase the number of requests, though, the resultant array might have a different order than the one you'd expect. It's a good idea to keep track of a discrete index (e.g., use an associative array) so you aren't surprised down the road when API responses have switched their order on you.

---

## See Also

Documentation for the [http-client module from the AMPHP project](#).

## 17.2 Waiting on the Results of Multiple Asynchronous Operations

## Problem

You want to juggle multiple parallel operations and then act on the overall result of all of them.

# Solution

Use the `parallel-functions` module of the AMPHP framework to execute your operations truly in parallel and then act on the final response of your entire collection of operations, as shown in [Example 17-1](#).

## Example 17-1. Parallel array map example

```
use Amp\Promise;
use function Amp\ParallelFunctions\parallelMap;

$values = Promise\wait(parallelMap([3, 1, 5, 2, 6], function($i) {
    echo "Sleeping for {$i} seconds." . PHP_EOL;
    ①
    \sleep($i);
    ②
    echo "Slept for {$i} seconds." . PHP_EOL;
    ③
    return $i ** 2;
    ④
}), [
    ⑤
]);
```

This first echo statement is merely used to demonstrate the order in which the parallel mapping operation occurs. You will see statements in your console in the same order as the array originally passed into `parallelMap()`—specifically, [3, 1, 5, 2, 6].

PHP's core `sleep()` function is blocking, meaning it will pause execution of your program until the input number of seconds has elapsed. This function call could be replaced by any other blocking operation with a similar effect. The goal in this example is to demonstrate that each operation is truly run in parallel.

After the application finishes waiting for `sleep()`, it will again print a message to demonstrate the order in which the parallel operations completed. Note that this will be different from the order in which they

were originally called! Specifically, numbers will be printed in ascending order because of the amount of time before each call to `sleep()` finishes.  
Any return value from your function <sup>4</sup> will ultimately be wrapped by a `Promise` object until the asynchronous operation completes.

Outside of `Promise\wait()`, <sup>5</sup> all of your collected promises will be resolved, and the final variable will contain a scalar value. In this case, that final variable will be an array of the squared values of the input array—in the same order as the original inputs.

## Discussion

The `parallel-functions` module is actually an abstraction layer atop AMPHP's `parallel` module. Both can be installed via Composer, and neither requires any special extensions to run. However, both will give you true parallel operations in PHP.

Without any extensions, `parallel` will spawn additional PHP processes to handle your asynchronous operations. It handles the creation and collection of child processes for you so you can focus on the actual implementation of your code. On systems using the [parallel extension](#), the library will instead use lighter-weight threads to house your application.

But in every case, your code will look the same. Whether the system uses processes or threads under the hood is abstracted away by AMPHP. This allows you to write an application that merely leverages `Promise`-level abstractions and trusts everything will work expected.

In [Example 17-1](#), you defined a function that contained some expensive blocking I/O calls. This example specifically used `sleep()` but could have been a remote API call, some expensive hashing operation, or a long-running database query. In any case, this is the kind of function that will freeze your application until it completes, and sometimes you might need to run it multiple times.

Rather than using synchronous code, where you pass each element of a collection into the function one at a time, you can leverage the AMPHP framework to process multiple calls at once.

The `parallelMap()` function behaves similarly to PHP's native `array_map()` except in parallel (and with the arguments in reverse order).<sup>5</sup> It applies the specified function to every member of the array but does so in either a separate process or a separate thread of execution. Since the operation itself is asynchronous, `parallelMap()` returns a `Promise` to wrap the function's eventual result.

You're left with an array of promises representing the separate, entirely parallel computations happening in the background. To move back into the land of synchronous code, leverage AMPHP's `wait()` function as you did in [Recipe 17.1](#).

## See Also

Documentation on the [parallel](#) and [parallel-functions](#) modules from the AMPHP framework.

## 17.3 Interrupting One Operation to Run Another

### Problem

You want to run two independent operations and move back and forth between them on the same thread.

### Solution

Use coroutines in the AMPHP framework to explicitly yield execution control between operations, as shown in [Example 17-2](#).

#### Example 17-2. Concurrent for loops with coroutines

```
use Amp\Delayed;
use Amp\Loop;
use function Amp\asyncCall;

asyncCall(function () {
    for ($i = 0; $i < 5; $i++) {
```

```

        print "Loop A - " . $i . PHP_EOL;
        yield new Delayed(1000);
    }

});

asyncCall(function () {
    for ($i = 0; $i < 5; $i++) {
        print "Loop B - " . $i . PHP_EOL;
        yield new Delayed(400);
    }
});

Loop::run();

```

The first loop merely counts from 0 to 4, stepping by 1 each time.

The AMPHP framework's `Delayed()` object is a promise that resolves itself after a given number of milliseconds—in this case, one full second.

The second loop also counts from 0 to 4 with a step size of 1.

The second loop resolves its promise after 0.4 seconds.

Both `asyncCall()` invocations **⑥** will fire immediately and print a 0 to the screen. However, the loops will not continue incrementing until the event loop is formally started (so the `Delayed` promises can actually resolve).

## Discussion

The Solution example introduces two key concepts important to understand when thinking about asynchronous PHP: an event loop and coroutines.

The event loop is at the core of how AMPHP will process concurrent operations. Without an event loop, PHP would have to execute your application or script from top to bottom. An event loop, however, gives the interpreter the ability to loop back on itself and run additional code in a different way. Specifically, the `Loop::run()` function will continue to

execute until either there is nothing left in the event loop to process or the application itself receives a `SIGINT` signal (e.g., from pressing `Ctrl+C` on your keyboard).

There are two functions within the AMPHP framework that create coroutines: `call()` and `asyncCall()`. Both functions will immediately invoke the callback passed into them; `call()` will return a `Promise` instance, whereas `asyncCall()` will not. Within the callback function, any use of the `yield` keyword creates a coroutine—a function that can be interrupted and will wait for the resolution of a `Promise` object before continuing.

In the Solution example, this promise is a `Delayed` object. This is AMPHP’s way of causing a routine to pause execution similar to `sleep()` in vanilla PHP. Unlike `sleep()`, though, a `Delayed` object is nonblocking. It will in essence “sleep” for a given period of time, then resume execution on the next pass of the event loop. While the routine is being delayed (or “sleeping”), PHP is free to handle other operations.

Running the Solution example in your PHP console will produce the following output:

```
% php concurrent.php
Loop A - 0
Loop B - 0
Loop B - 1
Loop B - 2
Loop A - 1
Loop B - 3
Loop B - 4
Loop A - 2
Loop A - 3
Loop A - 4
```

The preceding output demonstrates that PHP doesn’t need to wait for one loop to complete (with its chain of “sleep” or `Deferred` calls) before running the other. Both loops execute *concurrently*.

Note also that, if the two loops were executed synchronously, this entire script would take at least 7 seconds to execute (the first loop waits 1 second each time for five loops, and the second loop takes 0.4 seconds each time for five loops). Running these loops concurrently only takes 5 seconds in total. To

fully demonstrate this, store `microtime(true)` in a variable when the process starts and compare to the system time after the loop completes. For example:

```
use Amp\Delayed;
use Amp\Loop;
use function Amp\asyncCall;

$start = microtime(true);

// ...

Loop::run();

$time = microtime(true) - $start;
echo "Executed in {$time} seconds" . PHP_EOL;
```

Creating an event loop requires some minor overhead, but repeated executions of the Solution example with the preceding changes will reliably produce a result of approximately 5 seconds in total. What's more, you can also increase the loop counter in the second `asyncCall()` invocation from 5 to 10. That loop will still only take 4 seconds in total to run. Again, synchronously both loops would take 9 seconds to complete but, thanks to juggling execution context through coroutines, the script will *still* reliably complete in about 5 seconds. [Figure 17-2](#) illustrates the difference between synchronous and concurrent execution visually.

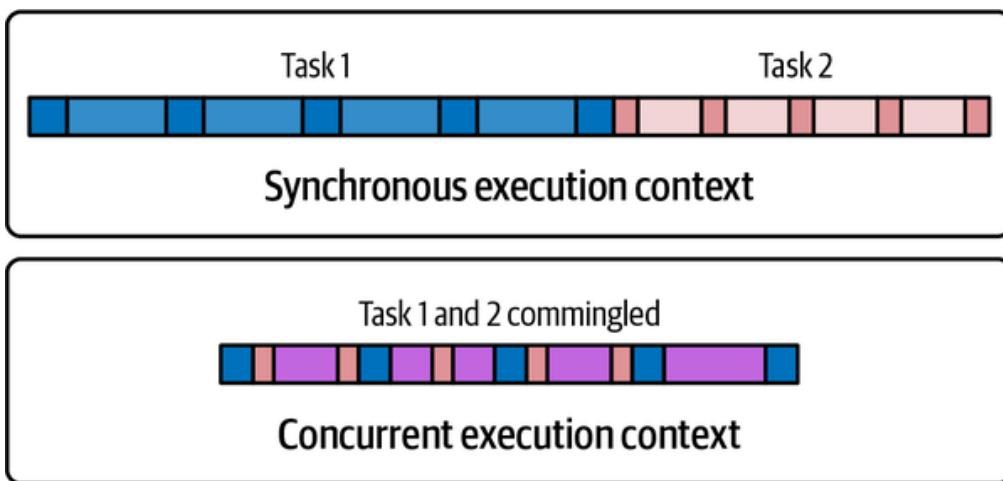


Figure 17-2. Executing two coroutines concurrently

By processing the two separate loops as coroutines within AMPHP's event loop, PHP is able to interrupt the execution flow of one to proceed with the execution of the other. By juggling between coroutines, PHP can make

maximum use of your CPU and allow your application to finish its work faster than if it ran through your logic synchronously.

The Solution example is a contrived illustration using delays or pauses; however, it extends to any situation where you might be leveraging a nonblocking but otherwise slow process. You can make a network request and leverage a coroutine so the application keeps processing while it waits for the request to complete. You could call out to a database or other persistence layer and house the nonblocking call within a coroutine. In some systems, you could also shell out to other processes (like Sendmail or another system process) and avoid these calls from blocking your application's overall execution.

## See Also

Documentation on the AMPHP framework's [`asyncCall\(\)` function](#) and on [coroutines in general](#).

## 17.4 Running Code in a Separate Thread

### Problem

You want to run one or more heavy operations on a separate thread to keep the main application free to report progress.

### Solution

Use the AMPHP project's `parallel` package to define a `Task` to be run and `Worker` instances to run it. Then invoke one or more workers as separate threads or processes. [Example 17-3](#) reduces an array of values to a single output by using a one-way hash recursively. It does so by wrapping the hash operation in an asynchronous `Task` meant to be run as part of a worker pool. [Example 17-4](#) then defines a pool of workers that run multiple `Task` operations in separate, coroutine-wrapped threads.

**Example 17-3. Task that uses recursive hashes to reduce an array to a**

single value

```
class Reducer implements Task
{
    private $array;
    private $preHash;
    private $count;

    public function __construct(
        array $array,
        string $preHash = '',
        int $count = 1000)
    {
        $this->array = $array;
        $this->preHash = $preHash;
        $this->count = $count;
    }

    public function run(Environment $environment)
    {
        $reduction = $this->preHash;

        foreach($this->array as $item) {
            $reduction = hash_pbkdf2('sha256', $item, $
        }

        return $reduction;
    }
}
```

#### Example 17-4. A worker pool can run multiple tasks

```
use Amp\Loop;
use Amp\Parallel\Worker\DefaultPool;

$results = [];

$tasks = [
    new Reducer(range('a', 'z'), count: 100),
    new Reducer(range('a', 'z'), count: 1000),
    new Reducer(range('a', 'z'), count: 10000),
    new Reducer(range('a', 'z'), count: 100000),
    new Reducer(range('A', 'Z'), count: 100),
    new Reducer(range('A', 'Z'), count: 1000),
```

```

        new Reducer(range('A', 'Z'), count: 10000),
        new Reducer(range('A', 'Z'), count: 100000),
    ];

Loop::run(function () use (&$results, $tasks) {
    require_once __DIR__ . '/vendor/autoload.php';
    use PhpAmqpLib\Connection\AMQPStreamConnection;
    use PhpAmqpLib\Message\AMQPMessage;
    $timer = Loop::repeat(200, function () {
        printf('.');
    });
    Loop::unreference($timer);

    $pool = new DefaultPool;

    $coroutines = [];

    foreach ($tasks as $index => $task) {
        $coroutines[] = Amp\call(function () use ($pool)
            $result = yield $pool->enqueue($task);

            return $result;
        );
    }

    $results = yield Amp\Promise\all($coroutines);

    return yield $pool->shutdown();
});

echo PHP_EOL . 'Hash Results:' . PHP_EOL;
echo var_export($results, true) . PHP_EOL;

```

## Discussion

The advantage of parallel processing is that you are no longer limited to running one operation at a time. Modern computers with multiple cores can literally and logically run more than one independent operation at a time. Thankfully, modern PHP can take advantage of this functionality quite well. It's efficiently exposed by the `parallel` module in the AMPHP framework.<sup>6</sup>

The Solution example uses this abstraction to enable the processing of multiple hash values in parallel, allowing the parent application to merely report on progress and the final result. The first component, a `Reducer` class, takes in an array of strings and produces an iterative hash of those values. Concretely, it performs a certain number of password-based key derivation hashes of each value in the array, passing the result of the derivation into the hash operation for the next item of the array.

---

#### TIP

Hash operations are intended to quickly convert a known value into a seemingly random one. They're one-way operations, meaning you can easily go from a seed value to a hash, but it's impractical to reverse a hash to retrieve its seed value. Some stronger security stances use multiple rounds of a specific hashing algorithm—in many cases, tens of thousands—to *explicitly* slow down the process and prevent “guess and check” types of attacks from trying to guess a particular seed.

---

Since these hashing operations are costly (in terms of time), you don't want to run them synchronously. Given how long they can take, you don't even want to run them *concurrently*. Instead, you want to run them fully in parallel to leverage all available cores on your machine. By embedding the operation into an object that extends `Task`, they can run at the same time when invoked within a thread pool.

AMPHP's `parallel` package exposes a thread pool with a default configuration, and you can easily enqueue as many operations in the pool as you want, so long as they implement `Task`. The pool will return a promise instance wrapping the task, meaning you can enqueue your tasks within coroutines and await the resolution of all of the promises they represent.

As all operations are asynchronous, the parent application can continue running code while the hashing happens in parallel. The Solution example exploits this advantage by setting up a repeating `printf()` operation to write a decimal point to the screen every 200 milliseconds. This acts somewhat like a progress bar or a liveness check, providing you with proactive acknowledgment that a parallel process is still running under the surface.

Once all of the parallel hashing jobs are finished, the overall operation prints the hashed results to the screen.

In reality, you could enqueue any kind of parallel job in such a way to do multiple tasks at once. AMPHP exposes an `enqueueCallable()` function that empowers you to turn any regular function call into a parallel operation. Let's say you need to retrieve weather reports from the US National Weather Service (NWS). Instead of enqueueing multiple hashing jobs as with the Solution example, you can just as easily fetch remote weather reports, as demonstrated in [Example 17-5](#).

#### Example 17-5. Asynchronous retrieval of weather reports

```
use Amp\Parallel\Worker;
use Amp\Promise;

$forecasts = [
    'Washington, DC' => 'https://api.weather.gov/gridpc
    'New York, NY'     => 'https://api.weather.gov/gridpc
    'Tualatin, OR'     => 'https://api.weather.gov/gridpc
];

$promises = [];
foreach ($forecasts as $city => $forecast) {
    $promises[$city] = Worker\enqueueCallable('file_get
    ①
}

$responses = Promise\wait(Promise\all($promises));
②

foreach($responses as $city => $forecast) {
    $forecast = json_decode($forecast);
    ③

    $latest = $forecast->properties->periods[0];

    echo "Forecast for {$city}:" . PHP_EOL;
    print_r($latest);
}
```

Each URL endpoint can be fetched independently with `file_get_contents()`. Using AMPHP's `enqueueCallable()`

function will automatically do this as part of an independent process in parallel to the main application. Each parallel request is wrapped in a `Promise` object. In order to return to the land of synchronous execution, you must wait until all of these promises are resolved. The `all` function collects the different promises into a single `Promise` object. The `wait()` function will block execution until this promise is resolved; then it unwraps the contained value for use in your synchronous code.

The NWS API returns a JSON object representing the forecast for a specific weather station. You need to first parse the JSON-encoded string before you can leverage the data in your application.

---

**WARNING**

The NWS weather API is entirely free to use but does require you to send a unique user agent with your request. By default, PHP will send a simple user agent string of `PHP` when you use `file_get_contents()`. To customize this, change the `user_agent` configuration in your `php.ini` file to be more unique. Without this change, the API will likely reject your request with a `403 Forbidden` error. For more on this and other behavior, reference the [general FAQs about the API](#).

---

Whether the AMPHP framework uses separate threads or entirely independent processes under the hood is a matter of how your system is configured initially. Your code remains the same and, absent any extensions supporting multithreaded PHP, will likely use spawned PHP processes by default. In either case, the `enqueueCallable()` function requires you to use either a native PHP function or a user-defined function that is loadable via Composer. This is because the spawned child process is only aware of system functions, Composer-loaded functions, and any serialized data sent over by the parent process.

This last detail is critical—the data you send from the parent application to the background worker will be serialized. Some user-defined objects might break when PHP attempts to serialize and deserialize them. Even some core objects (like stream contexts) are incompatible with serialization and cannot be passed into a child thread or process.

Take care with what tasks you choose to run in the background to ensure that the data you send is compatible with serialization and parallel operations.

## See Also

Documentation on the [parallel package](#) from the AMPHP framework.

# 17.5 Sending and Receiving Messages Between Separate Threads

## Problem

You want to communicate with multiple running threads to synchronize state or manage the tasks those threads are executing.

## Solution

Use a message queue or bus between your main application and the separate threads it's orchestrating to allow for seamless communication. For example, use RabbitMQ as an intermediary between your primary application (as illustrated by [Example 17-7](#)) and independent worker threads, as shown in [Example 17-6](#).

### Example 17-6. Background task used to send mail based on a queue

```
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('127.0.0.1', 576
①

$channel = $connection->channel();
$channel->queue_declare('default', false, false, false,
②

echo '... Waiting for messages. To exit press CTRL+C' . . .

$callback = function($msg) {
    $data = json_decode($msg->body, true);
③

    $to = $data['to'];
    $from = $data['from'] ?? 'worker.local';
    $subject = $data['subject'];
```

```

$message = wordwrap($data['message'], 70) . PHP_EOL

$headers = "From: {$from} PHP_EOL X-Mailer: PHP Wor
print_r([$to, $subject, $message, $headers]) . PHP_
④

mail($to, $subject, $message, $headers);

$msg->ack();
⑤

};

$channel->basic_consume('default', '', false, false, f
⑥

while(count($channel->callbacks)) {
    $channel->wait();
    ⑦

}

```

Open a connection to a locally running RabbitMQ server by using the default port and default credentials. In production, these values will be different and should be loaded from the environment itself.

Declaring a queue to the RabbitMQ server merely opens a channel of communication. If the queue already exists, this operation does nothing.

Data is wrapped in a message object when it comes into the worker from RabbitMQ. The actual data you need is in the body of the message.

Printing data within the worker is a helpful way to diagnose what is happening and inspect the data flowing in for any potential errors.

Once your worker has completed acting on a message, it needs to acknowledge the message to the RabbitMQ server; otherwise, another worker might pick the message up and retry it later.

Consuming messages is a synchronous operation. When a message comes in from RabbitMQ, the system will invoke the callback passed to this function with the message itself as the argument.

So long as there are callbacks on a message, this loop will run forever, and the `wait()` method will keep the connection open to RabbitMQ so the worker can consume and act on any messages in the queue.

#### Example 17-7. Main application that sends messages to the queue

```
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('127.0.0.1', 567
①

$channel = $connection->channel();
$channel->queue_declare('default', false, false, false,
②

$message = [
    'subject' => 'Welcome to the team!',
    'from'      => 'admin@mail.local',
    'message'   => "Welcome to the team!\r\nWe're excited
];

$teammates = [
    'adam@eden.local',
    'eve@eden.local',
    'cain@eden.local',
    'abel@eden.local',
];
③

foreach($teammates as $employee) {
    $email = $message;
    $email['to'] = $employee;

    $msg = new AMQPMessage(json_encode($email));
    $channel->basic_publish($msg, '', 'default');
④

}

$channel->close();
⑤

$connection->close();
```

As with the worker, you open a connection to the local RabbitMQ server by using default parameters.

Also as with the worker, you declare a queue. If this queue already exists, this method call will not do anything.

Before you can send a message, you need to encode it. For the purposes of this example, the payload will be serialized as a JSON string.

For each message, you choose the queue on which to publish and dispatch the message to RabbitMQ.

Once you're done sending your messages, it's a good idea to explicitly close the channel and connection before doing any other work. In this example, there is no other work to be done (and the process will exit immediately), but explicit resource cleanup is a healthy habit for any developer.

## Discussion

The Solution example uses multiple, explicit PHP processes to handle large operations. The script defined in [Example 17-6](#) could be named `worker.php` and instantiated multiple times individually. If you do so in two separate consoles, you will spawn two entirely independent PHP processes that connect to RabbitMQ and listen for jobs.

Running [Example 17-7](#) in a third window will start the main process and dispatch jobs by sending messages to the `default` queue housed by RabbitMQ. The workers will independently pick these jobs up, process them, and wait for more work down the road.

The full interaction between the parent process ([Example 17-7](#)) and two fully asynchronous worker processes ([Example 17-6](#)) using RabbitMQ as a message broker is illustrated by the three independent console windows shown in [Figure 17-3](#).

The screenshot shows two terminal windows side-by-side. Both windows have tabs labeled 'php' at the top. The left window has tabs 'php' and 'php 3'. The right window has tabs 'php' and 'php 2'. Both windows show command-line output from PHP scripts. The left window's output is:

```
ericmann@Eric-Mann-MBP16tb-5 ~ % php worker.php
... Waiting for messages. To exit press CTRL+C
Array
(
    [0] => mickey.mouse@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
Array
(
    [0] => donald.duck@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
```

The right window's output is:

```
ericmann@Eric-Mann-MBP16tb-5 ~ % php worker.php
... Waiting for messages. To exit press CTRL+C
Array
(
    [0] => minnie.mouse@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
Array
(
    [0] => daisey.duck@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
```

At the bottom of the left window, the prompt '-zsh' is visible. In the bottom right corner of the left window, there is a small icon of a person sitting at a desk.

Figure 17-3. Multiple PHP processes communicating via RabbitMQ

The different processes don't communicate directly. To do that, you'd need to expose an interactive API. Instead, the much simpler means of communication is to leverage an intermediate message broker—in this case, [RabbitMQ](#).

RabbitMQ is an open source tool that interfaces directly with several different programming languages. It allows for the creation of multiple queues that can then be read by one or more dedicated workers to process the content of the message. In the Solution example, you used workers and PHP's native `mail()` function to dispatch email messages. A more complicated worker might update database records, interface with a remote API, or even process computationally expensive operations like the hashing performed in [Recipe 17.4](#).

#### TIP

Since RabbitMQ supports multiple languages, you're not limited to just PHP in your implementation. If there's a specific library you want to use in a different language, you could write your workers in that language, import the library, and dispatch work to the worker from your primary PHP application.

In a production environment, your RabbitMQ server would leverage username/password authentication or possibly even explicitly allowlist the servers that can talk to it. For development, though, you can effectively leverage your local environment, default credentials, and tools like [Docker](#) to run a RabbitMQ server on your local machine. To directly expose RabbitMQ

by using the default port and default authentication, use the following Docker command:

```
$ docker run -d -h localhost -p 127.0.0.1:5672:5672 --r
```

Once the server is running, you can register as many queues as necessary to manage the flow of data within your swarm of applications.

## See Also

The official [documentation](#) and [tutorials](#) for configuring and interacting with RabbitMQ.

## 17.6 Using a Fiber to Manage the Contents from a Stream

### Problem

You want to use PHP's newest concurrency feature to pull data from and operate on a stream in parts rather than buffering all of its contents at once.

### Solution

Use a Fiber to wrap the stream and read its contents one piece at a time.

[Example 17-8](#) reads the entirety of a web page into a file in 50-byte chunks, tracking the total number of bytes consumed as it reads in the content.

**Example 17-8. Reading a remote stream resource through a Fiber one chunk at a time**

```
$fiber = new Fiber(function($stream): void {  
    while (!feof($stream)) {  
        $contents = fread($stream, 50);  
        ①  
        Fiber::suspend($contents);  
        ②  
    }  
}
```

```

});
```

```

$stream = fopen('https://www.eamann.com/', 'r');
stream_set_blocking($stream, false);
③
```

```

$output = fopen('cache.html', 'w');
④
```

```

$contents = $fiber->start($stream);
⑤
```

```

$num_bytes = 0;
while (!$fiber->isTerminated()) {
    echo chr(27) . "[0G";
    ⑥
```

```

    $num_bytes += strlen($contents);
    fwrite($output, $contents);
    ⑦
```

```

    echo str_pad("Wrote {$num_bytes} bytes ...", 24
    usleep(500);
    ⑧
```

```

$contents = $fiber->resume();
⑨
```

```

}
```

```

echo chr(27) . "[0G";
echo "Done writing {$num_bytes} bytes to cache.html!" .
```

```

fclose($stream);
⑩
```

```

fclose($output);
```

The Fiber itself accepts a streaming 1 resource as its only parameter when it starts. So long as the stream is not at the end, the Fiber will read the next 50 bytes from the current position into the application.

Once the Fiber has read from the stream, it will suspend operation and pass control back to the parent application stack. As Fibers can send data back to the parent stack, this Fiber will send the 50 bytes it has read from the stream when it suspends execution.

Within the parent application stack, the stream is opened and set to not block execution of the rest of the application. In nonblocking mode, any calls to `fread()` will return right away rather than waiting for data on the stream.

Within the parent application, you can also open other resources, like local files into which you can cache the contents of the remote resource.

When starting the Fiber, you pass the main stream resource as a parameter so it's available to the call stack of the Fiber itself. Once the Fiber suspends execution, it will also return the 50 bytes it has read back to you.

To write over the previous line of console output, pass the ESC character (`chr(27)`) and an ANSI control sequence to move the cursor to the first column in the terminal (`[0G]`). Any subsequent text printed to the screen will now overwrite anything displayed previously.

Once data is available from the remote stream, you can write that data directly to your local cache file.

A sleep statement is not necessary to this application but is useful to illustrate how other computations can happen in the parent application stack while the Fiber is suspended.

Resuming the Fiber will retrieve the next 50 bytes from the remote stream resource, assuming that any bytes remain. If nothing is left to retrieve, the Fiber will terminate, and your program will exit its `while` loop.

Once execution is complete and the Fiber is cleaned up, be sure to close any streams or other resources you've opened.

## Discussion

Fibers are similar to coroutines and generators in that their execution can be interrupted so that the application can perform other logic before returning control. Unlike these other constructs, Fibers have call stacks independent from that of the rest of the application. In this way, they empower you to

pause their execution even within nested function calls without changing the return type of the function triggering the pause.

With a generator that uses the `yield` command to suspend execution, you must return a `Generator` instance. With a Fiber using the `::suspend()` method, you can return any type you desire.

Once a fiber is suspended, you can resume its execution from anywhere within the parent application to restart its separate call stack. This allows you to effectively jump between multiple execution contexts without worrying too much about controlling application state.

You can also effectively pass data to and from a Fiber. When a Fiber suspends itself, it can choose to send data back to the parent application—again, of any type you need. When you resume a Fiber, you can pass any value you want or no value at all. You can also choose to throw an exception into the Fiber by using the `::throw()` method and then handle that exception within the Fiber itself. [Example 17-9](#) demonstrates exactly what it would look like to handle an exception from within the Fiber.

#### Example 17-9. Handling an exception from within a Fiber

```
$fiber = new Fiber(function(): void {
    try {
        Fiber::suspend(); ❶
    }
    catch (Exception $e) {
        echo $e->getMessage() . PHP_EOL; ❷
    }
    echo 'Finished within Fiber' . PHP_EOL;
❸
});;

$fiber->start(); ❹

$fiber->throw(new Exception('Error')); ❺
```

The Fiber will immediately suspend execution once it's started and

The Fiber will immediately suspend execution once it's started and return control to the parent application stack.

When the Fiber is resumed, assuming it encounters a catchable [Exception](#), it will extract and print out the error message.

Once the Fiber finishes execution, it will print a useful message before ending its concurrent execution and returning control to the main application.

Starting the Fiber merely creates its call stack and, because the Fiber immediately suspends, execution continues from the perspective of the parent stack.

Throwing an exception from the parent into the Fiber will trigger the `catch` condition and print the `Error` message to the console.

Fibers are an effective way to juggle execution contexts between call stacks but are still fairly low-level within PHP. While they can be straightforward to use with simple operations like that in the Solution example, more complicated computations can become difficult to manage. Understanding how Fibers work is critical to using them effectively, but just as critical is choosing the proper abstraction to manage your Fibers for you. The [Async package](#) from ReactPHP provides effective abstractions to asynchronous operations, including Fibers, and makes engineering a complex concurrent application relatively easy.

## See Also

The PHP Manual covering [Fibers](#).

<sup>1</sup>The promise behind Octane is that it will improve the performance of most applications without any changes to their code. However, there will likely be some edge cases in production where changes are required, so thoroughly test your code before relying on the project as a drop-in runtime replacement in production.

<sup>2</sup>Review [Recipe 7.15](#) for more on generators and the `yield` keyword.

<sup>3</sup>As is true with any module and the AMPHP framework itself, you can install the `http-client` package by using Composer. Review [Recipe 15.3](#) for more information on Composer packages.

<sup>4</sup>For more on anonymous functions, or lambdas, review [Recipe 3.9](#).

For more on `array_map()`, review [Recipe 7.13](#).

<sup>6</sup>The AMPHP framework also publishes a `parallel-functions` package that exposes several useful helper functions wrapping the lower-level `parallel` package. For more on these functions and their usage, review [Recipe 17.2](#).