EVENT-DRIVEN, NON-BLOCKING I/O WITH PHP

LEARNING

EVENT-DRIVEN PHP

WITH REACTPHP

COMPLETE GUIDE TO CREATING ASYNCHRONOUS APPLICATIONS IN PHP

BY SERGEY ZHUK

Learning Event-Driven PHP With ReactPHP

Sergey Zhuk

This book is for sale at http://leanpub.com/event-driven-php

This version was published on 2020-09-29

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

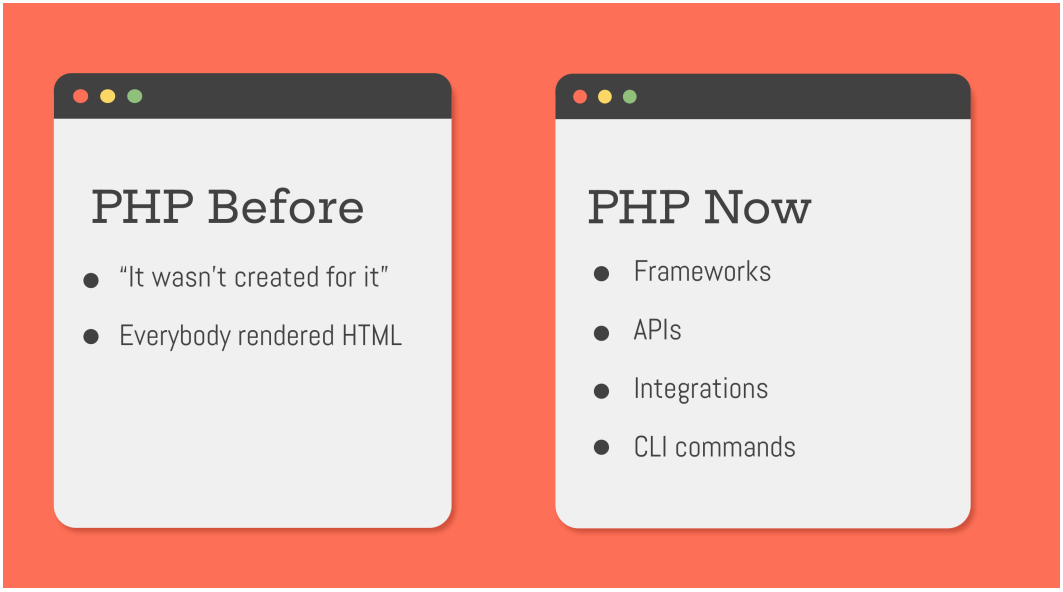
Contents

What is ReactPHP

The Problem

I think that nowadays in PHP community there is no common agreement about asynchronous code. One part of the community considers async PHP as something new and interesting, that can solve some performance issues. While another part of the community is strongly against asynchronous PHP and suggests using NodeJs or Go for solving performance-sensitive tasks. Let's try to understand why this happens, why many people don't consider PHP as a suitable tool for solving asynchronous tasks. As for me, I like PHP and I want to use it to solve different tasks. Also, I don't want to limit myself with request-response circle. Moreover, I don't want to extend my stack with a new language with its ecosystem, just because someone on Twitter or Reddit said that PHP is not the right tool for my problem.

The most popular reason is that PHP was not created for it. For this sort of tasks. That PHP is only about Web and rendering HTML pages. On the one hand, yes, it's hard to argue that PHP was not designed to be asynchronous. But truth be told, it also was not invented as a language for creating large and complex applications. Let's be honest, in those times there was no JavaScript at all, and people even didn't think about the asynchronous staff. Everybody just rendered HTML synchronously and people were happy with it.



But since then, almost everything has changed. Now we have Composer with a huge community around it. The language itself has evolved a lot. Now we have powerful frameworks that help us to create complicated enterprise applications on PHP. And step by step PHP is no longer only about request-response. Now we have backend and frontend, we develop different APIs, we make different integrations with other infrastructure systems, and of course CLI commands are a very common thing nowadays. Furthermore, the requirements for our program have also changed a lot. Now we count milliseconds and want our scripts to execute very fast.

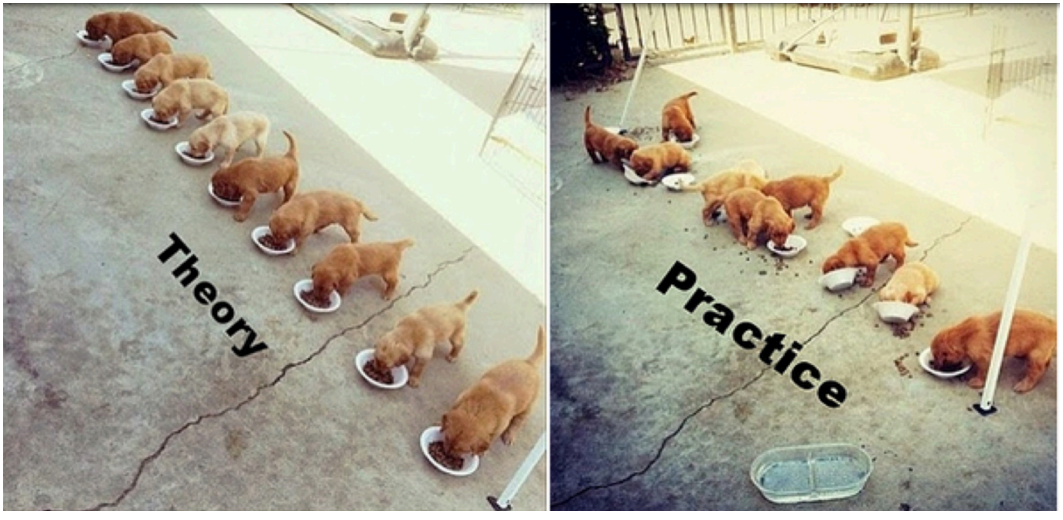
Asynchronous code

How does asynchronous code help us here? How can it improve application performance? To better understand this, let's classify what types of execution we have.

Synchronous code. Here we have one single thread. The code is executed sequentially line by line. While one line of code is executed the program waits. This way traditional PHP works. **Asynchronous code**. We still have one thread but we add a concurrency here. The code is constantly busy doing something switching between different tasks. Nodejs and ReactPHP work this way. **Parallel code**. We have many

threads, each of them can execute their own task independently. For example, Python can work this way.

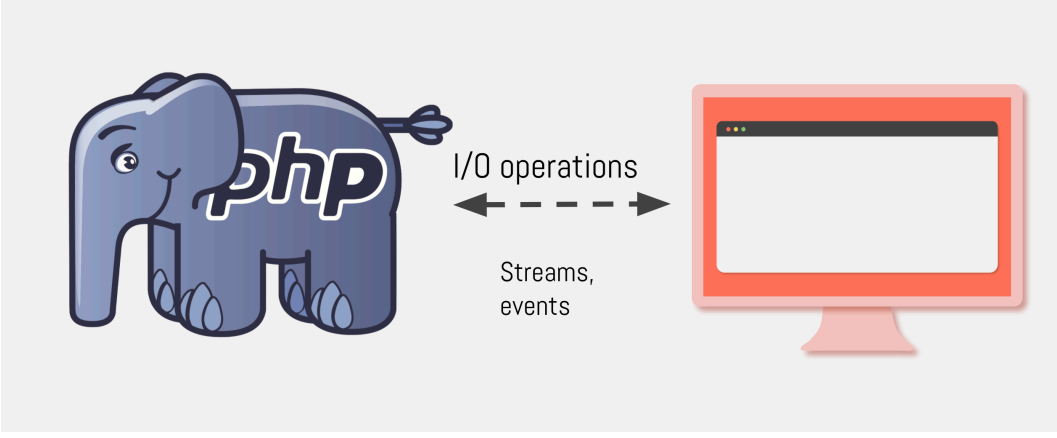
We work with PHP it means that we are going to cover the async approach when we have one thread that is constantly busy with something. The advantage of this approach is that if we compare it with multithreading, then, in theory, we can do more in the same period of time. Because our thread will be constantly busy. This way we will maximize the use of resources. Parallel execution with several threads is quite complicated in terms of writing code: you need to synchronize threads, reallocate memory, and so on and so on. While in theory, everything should be done in parallel and quickly, but in reality chances high that it will become just a mess.



How can we make PHP asynchronous? To answer this question we need to talk about input-output operations and the concept of non-blocking I/O when read/write operations do not block the flow. The fact is that input/output operations are rather slow. As a rule, they are executed many times slower than calculations. For example, when we discuss operations that our CPU does, we use nanoseconds, but when we deal with network interactions, such operations are measured in milliseconds and sometimes even in seconds. So, the difference is huge. But input/output is everywhere: API calls, filesystem operations, interaction with the database. Thus, programs written in a traditional blocking way spend a lot of time waiting for response from disk or until a network request is completed. If performance is critical for us, and we want to speed up the execution of our application, then we need to

somehow start such operations *in background*, execute them asynchronously.

How can this be implemented in PHP? How can we run tasks without waiting for them to be completed? Well, we can ask the operating system to process all operations related to input/output. For example, we need to read data from a file. If we don't want our PHP script to wait for the filesystem to respond we don't directly communicate with it. Instead, we go to the operating system and ask it to open a stream in the filesystem and read some data from it. And that's it, then our script can go away and do something else. Once the operating system does its task, we will receive an event from it - saying that the data is read so we can come back and process it.

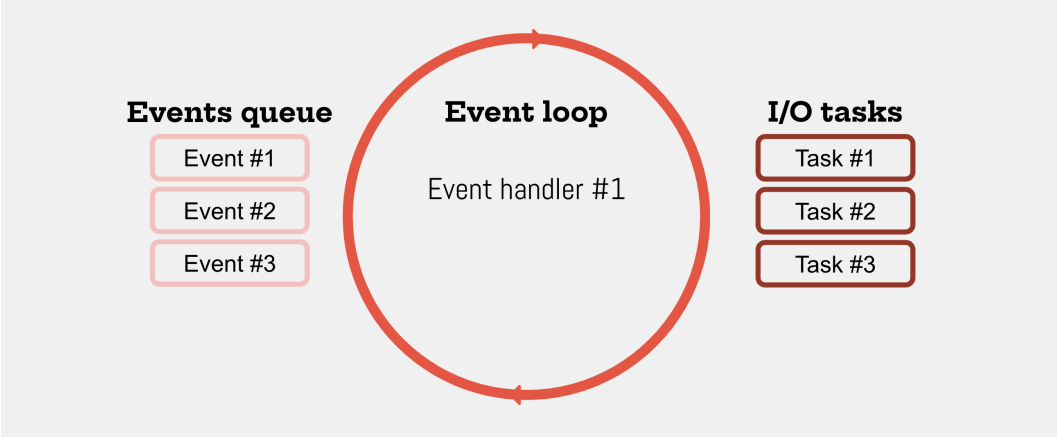


Event-Driven Architecture

And to be able to listen to such events, to be able to come back and process them, we need a special architecture. Event-oriented architecture, which is used for example in NodeJS. Here PHP and NodeJS have something in common: both are executed in one single thread. And this does not limit us. Everything that can be performed slowly, input/output operations, will, be executed not by our PHP script, but by the operating system. We will only process events received from the system.

Let's consider this approach in detail. All work, all interaction between different parts of the code in such an architecture is built on events. We have one running process, our PHP script with a running event loop. The event loop under the hood

is just an endless loop that listens to specific events and calls handlers for them. For example, we have some kind of input/output task - reading a file. We start it and tell the OS to read the data from this file. That's it. Then the execution thread can do something else. We started this task and do not wait until it is completed. Once the OS read some portion of data from the file it sends us an event with the data that has been read. A record of this event is added to the event queue. The execution thread takes the first event from the queue and calls the corresponding handler for this event. For example, we calculate something based on this data from a file. The handler itself can also start for example two more I/O tasks. Let's say we want to send this data via HTTP. And these new tasks can also generate their events. Once the event handler is completed, the thread returns to the event queue and takes new events that have occurred while the thread was busy. The loop executes until it has something to do: either tasks are performed in the background, or something else has not been processed in the event queue. As soon as all tasks are completed and the event queue is empty the loop stops. Having this kind of architecture, we can start several I/O operations, and we do not need to wait until they are done. Instead, we will know about their completion through events, and then we can go back and respond to these events. We don't waste time anymore. These I/O tasks will run in the background for as long as they need.



Hello world

What about PHP? Of course, in the language itself, we still do not have built-in support for writing asynchronous code. Yes, there are low-level things, but we don't have such high-level abstractions like event loop, streams, and promises. But this does not mean that we cannot use PHP to solve asynchronous tasks. With the rise of Composer, the problem with the lack of tools was solved by the PHP community and ReactPHP is the tool that makes it easy to write asynchronous code in PHP. This is how it looks. Let's quickly consider a very simple example of JavaScript code to demonstrate asynchronous code.

```
setTimeout(function () {
  console.log("world");
}, 0);

console.log("Hello ");
```

The `setTimeout()` function defers code execution, in our case printing the word "world". This code will be executed in the specified number of milliseconds. Here we have `0` milliseconds. So, common sense tells us that a delay of `0` seconds is basically the same as doing right here right now. And we think that the word "world" will be printed first, and only then JavaScript will print the word "hello". However, if we run this code, "hello world" will be printed. This happens because `setTimeout()` is an asynchronous function. It executes nothing. Under the hood it takes this callback and puts it in the queue to execute later. And like our chef in the kitchen, who puts the pot on the stove and goes further, the code is executed the same way and the line that prints the word "hello" is executed. Then the event loop looks through the timers queue, finds out that the delay of `0` seconds has already passed and runs this code. That is, here we break the traditional sequential flow. The second call to `console.log()` will be executed before the call inside `setTimeout()`. Thus we receive the message "hello world". And the exact same code written with ReactPHP looks like this:

```
$loop = Factory::create();
$loop->addTimer(
  0,
  function () {
    echo 'world';
  }
);

echo 'Hello ';
$loop->run();
```

What's going on here? Examine the execution flow from top to bottom. At first, we create an event loop. Yes, in PHP we don't have an event loop running in background. Therefore, we need to suffer a little and explicitly manually create it. Next, we add a timer, indicating that after `0` seconds this code should be executed. Then we print the word "hello" and manually start the event loop. At the time the event loop starts, the word hello is already printed. The event loop starts, it sees that a timer has already gone off. The corresponding event is fired and in response to this event, this handler is called - the code inside `addTimer()`. The word "world" is printed. With this simple example, you can see the basic structure of any ReactPHP application:

1. We create an event loop once at the beginning of the program.
2. We write code that uses an event loop. This part describes the behaviour of our application: how we are going to react to different events.
3. And at the end of the program we start the event loop once.

If we remove the creation and the start of the loop and it turns out that the code written in ReactPHP looks almost exactly like the code in JavaScript. The only difference here is that in PHP we need to explicitly create and run the loop.

Event Loop

Package: EventLoop
Version: 1.1
Installation: `composer require react/event-loop`

Basics

Event loop[1] is the core of ReactPHP, it is the most low-level component. Every other component uses it. Event loop runs in a single thread and is responsible for scheduling asynchronous operations. It sequentially processes the queued events by executing the associated with each event callback. We register these callbacks for certain types of events before running the loop. Callbacks should be short-running functions and they **do not block the CPU** for a long period, since the whole loop will be blocked. Once a callback execution is finished the event loop dequeues the next event and processes its callback.

There is no other code being executed in parallel. Event loop is the only synchronous thing. Another words:

> *Everything except your code runs in parallel.*

Event loop implements the Reactor Pattern. You register an event, subscribe to it and start listening. Then you get notified when this event is fired, so you can *react* to this event via a handler and execute some code. Every iteration of the loop is called a *tick*. When there are no more listeners in the loop, the loop finishes.

From the consumer point of view, you don't have to deal a lot with it, unless you are doing something special on top of the loop. You construct it once via the factory

[1] http://reactphp.org/event-loop/

and then you just pass it along through the dependency injection to set up the other components, and then you simply run it once to start the loop.

1. You create it once at the beginning of the program.
2. Set up it.
3. And `run` it once at the end of the program.

```
$loop = React\EventLoop\Factory::create();

// some code that uses the instance
// of the loop event

$loop->run();
```

Once you call `run()` on an instance of the event loop, it will run until there are no more tasks to perform. Then the program stops. To force the loop to stop you can explicitly call `stop()`.

Implementations

ReactPHP provides several implementations of the event loop depending on what extensions are available in the system. The most convenient and recommended way to create an instance of the loop is to use a factory:

```
$loop = React\EventLoop\Factory::create();
```

Under the hood this factory simply checks for available extensions and selects the best implementation for the current system. All implementations of `React\EventLoop\LoopInter` support the following features:

- File descriptor polling.
- Timers.
- Deferred execution of callbacks.

While each implementation of the event loop is different, the program itself should not depend on the particular loop implementation. There may be some differences in the exact timing of the execution or the order in which different types of events are executed. But the behavior of the program should not be affected by these differences.

Event loop and multiple CPUs

> *Since the loop is single-threaded does that mean it will only utilize one CPU?*

Yes, PHP is single-threaded and uses only one CPU core. The problem with running some code on multiple CPU cores at once is that it requires some sort of coordination between these multiple threads of execution. To split the work between the multiple CPU cores each thread has to talk to each other about the current state of the program. But the idea behind ReactPHP is not to exploit your server 16 CPU's, but to fully exploit your processor time. If you want to use your 16 CPU, you just launch 16 servers on different ports and put a load balancer like nginx in front of them.

Timers

Timers can execute some code at a later time, at a number of seconds in the future (just like `setTimeout()` and `setInterval()` do in JavaScript). They are not the same as a `sleep()` function, instead, they are *events* in the future. Timers will run as early as possible after the specified amount of time has passed.

Asynchronous vs Parallel

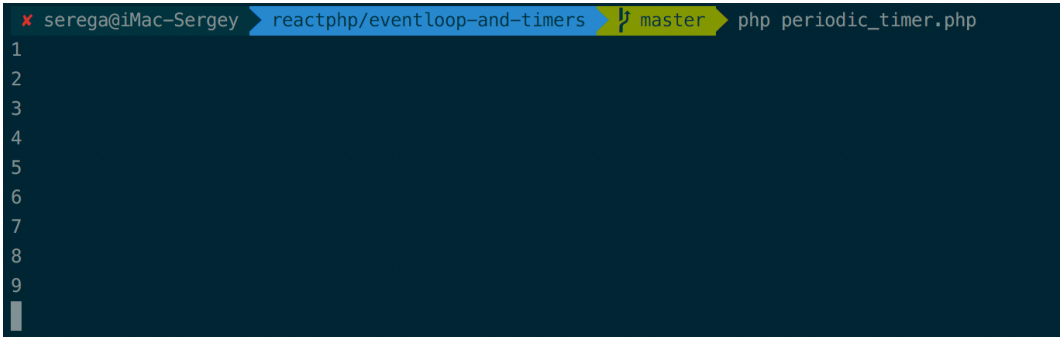
Asynchronous is not the same as *parallel*. Asynchrony is the possibility of inconsistent code execution. Parallelism is the ability to execute the same code at one time. Event loop works asynchronously, but not in parallel. That means that timers are not time-accurate and can run a little late. Also, if you have several timers that are scheduled to execute at the same time, the order of their execution is not guaranteed. Any timer will be executed **not earlier than the specified time**. The code runs in the one thread and cannot be interrupted. That means that all timers are executed in the same thread as the event loop runs. In the situation when one timer is being executed too long, all the other timers will wait, until this timer will be done. Also, it is possible that some timers will **never** be executed.

Periodic Timer

This timer schedules it's callback to be invoked repeatedly every specified number of seconds. Periodic timer can be added to the loop with `addPeriodicTimer($interval, callable $callback)` method. It accepts an interval in seconds and a callback, which will be executed at the end of this interval:

```
$loop = React\EventLoop\Factory::create();
$counter = 0;

$loop->addPeriodicTimer(
  1,
  function () use (&$counter) {
    $counter++;
    echo "$counter\n";
  }
);

$loop->run();
```

A periodic timer is registered with the event loop. Then we start event loop with `$loop->run()`, when a timer is fired the code flow leaves an event loop and a *timer* code is being executed. Every second, the timer displays an increasing number. Event loop will run endlessly.



A callback can accept an instance of the timer, in which this callback is executed:

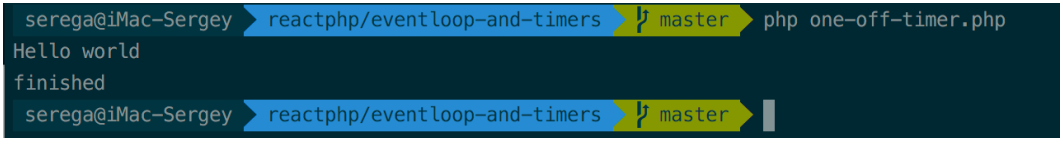
```
use React\EventLoop\TimerInterface;

$loop->addPeriodicTimer(
  2,
  function (TimerInterface $timer) {
    // ...
  }
);
```

One-off Timer

The only difference with the periodic timer is that this timer will be executed only once and then will be removed from the timers storage.

```
$loop = React\EventLoop\Factory::create();

$loop->addTimer(2, fn() => print "Hello world\n");
$loop->run();

echo "finished\n";
```

In 2 seconds the script will output a `Hello world` string then the timer will be removed and event loop will stop.



Controlling Timers

There are two more methods available in the loop object to control timers:

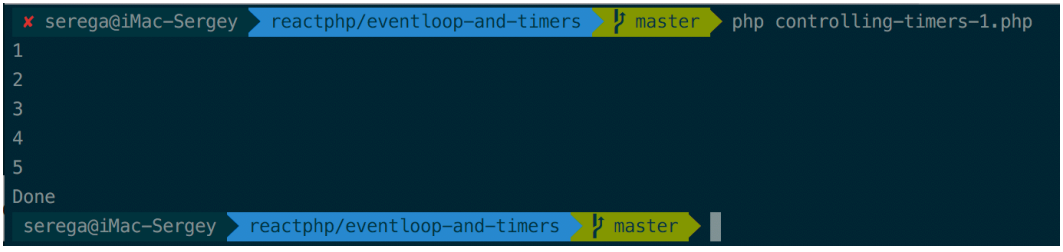
- `cancelTimer(TimerInterface $timer)` to detach a specified timer.

- `isTimerActive(TimerInterface $timer)` to check if a specified timer is attached to the event loop.

We can use a passed instance of the timer to detach it from the event loop:

```
use React\EventLoop\TimerInterface;;

$loop = React\EventLoop\Factory::create();
$counter = 0;

$loop->addPeriodicTimer(
  2,
  function (TimerInterface $timer) use (&$counter, $loop) {
    $counter++;
    echo "$counter\n";

    if ($counter === 5) {
      $loop->cancelTimer($timer);
    }
  }
);

$loop->run();
echo "Done\n";
```

After the fifth execution, this timer will be detached. When event loop becomes empty it stops.



Timers can interact with each other. Both methods `addTimer()` and `addPeriodicTimer()` return an instance of the attached timer. Then we can use this instance and pass it to the callback of the another timer. This way we can specify a timeout for some event:

```
$loop = React\EventLoop\Factory::create();
$counter = 0;

$periodicTimer = $loop->addPeriodicTimer(
  2,
  function () use (&$counter) {
    $counter++;
    echo "$counter\n";
  }
);

$loop->addTimer(
  5,
  fn() => $loop->cancelTimer($periodicTimer)
);

$loop->run();
```

In the snippet above the periodic timer will be executed only first 5 seconds, after that, it will be detached from the event loop.

Avoid blocking operations

Since all the timers are executed in the same thread, you should be aware of blocking operations inside the callbacks. One blocking timer can stop the whole event loop like this:

```
$loop = React\EventLoop\Factory::create();

$i = 0;

$loop->addPeriodicTimer(
  1,
  function () use (&$i) {
    echo ++$i, "\n";
  }
);

$loop->addTimer(2, fn() => sleep(10));

$loop->run();
```

The first periodic timer will wait for 10 seconds until the second one-off timer will be executed.

Conclusion

Timers can be used to execute some code in a delayed future. This code *may be executed after* a specified interval. Each timer is being executed in the same thread as the whole event loop, so any timer can affect this loop. Timers can be useful for non-blocking operations such as I/O, but executing a long living code in them can lead to unexpected results.

You can find examples from this chapter on GitHub[2].

[2] https://github.com/seregazhuk/reactphp-book/tree/master/1-timers