Asynchronous Programming in PHP

Lochemem Bruno Michael

Agenda

- Introduction
- The rigors of I/O
- Asynchrony
- The asynchronous PHP landscape
- The event loop
- Streams
- Promises
- Sockets
- HTTP Servers
- Command Line applications

Lochemem Bruno Michael

https://chemem.site
@agiroLoki
@ace411

$whoami

- PHP/JS/C++ enthusiast

$whoami

- PHP/JS/C++ enthusiast
- Functional Programming aficionado

$whoami

- PHP/JS/C++ enthusiast
- Functional Programming aficionado
- Maintainer of several packages and PHP extensions

$whoami

- PHP/JS/C++ enthusiast
- Functional Programming aficionado
- Maintainer of several packages and PHP extensions
- Author

$whoami

- PHP/JS/C++ enthusiast
- Functional Programming aficionado
- Maintainer of several packages and PHP extensions
- Author
- Hooper

$whoami

- PHP/JS/C++ enthusiast
- Functional Programming aficionado
- Maintainer of several packages and PHP extensions
- Author
- Hooper
- Gamer

A lot of usable software is a combination of Input-Output (I/O) operations

I/O everywhere?

- Filesystem interactions
- Database interactions
- Reading from Standard Input (STDIO)
- Writing to Standard Output (STDOUT)
- API calls (REST, SOAP)

I/O is slow

Access type	Latency (ns)
L1 cache reference	0.5
Send packet CA->Holland->CA	150,000,000

Fun fact

If you multiply the durations by a billion, the former's latency is the equivalent of one heartbeat, and the latter's approximates an entire Bachelor's degree program.

Source: Latency numbers every programmer should know

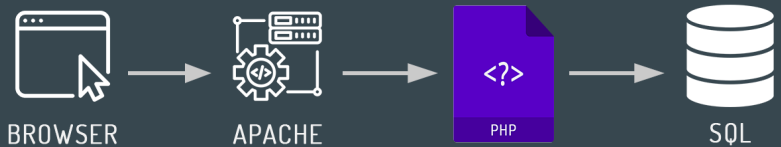
Traditional PHP is, despite recent improvements, not immune to this problem

Blocking I/O Galore

- Sequential execution of function calls
- Multiple idle periods between successive executions
- Direct result-to-variable binding

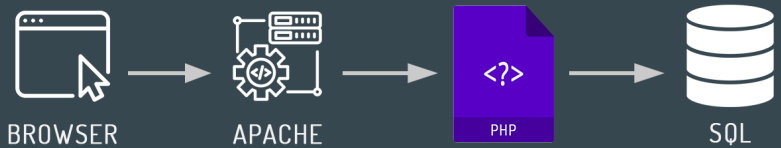
```php
$get = fn (string $uri) => IO(fn () => file_get_contents($uri));

$fst = $get('https://host/path'); // then wait

$snd = $get('https://host/path?query'); // then wait - again

// more calls
```

Conventional HTTP configuration with PHP



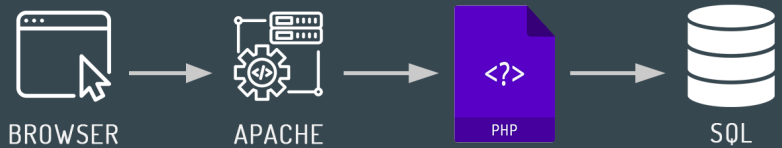
- Traditional LAMP stack

Conventional HTTP configuration with PHP



- Traditional LAMP stack
- Often tuned up with PHP-FPM

Conventional HTTP configuration with PHP



- Traditional LAMP stack
- Often tuned up with PHP-FPM
- Still works!

I/O’s only gotten more arduous

- Live data

I/O’s only gotten more arduous

- Live data
- Server-Sent Events (SSE)

I/O’s only gotten more arduous

- Live data
- Server-Sent Events (SSE)
- Robust HTTP APIs

I/O's only gotten more arduous

- Live data
- Server-Sent Events (SSE)
- Robust HTTP APIs
- & more

Asynchrony is a potent answer to I/O-related problems

So, what is it?

The ability to run multiple processes, independent of main program flow - by interleaving them in a single execution thread.

What are the requirements?

- An event loop

What are the requirements?

- An event loop
- A proxy mechanism for handling undetermined values

What are the requirements?

- An event loop
- A proxy mechanism for handling undetermined values
- A lot of un-buffered data

What are the requirements?

- An event loop
- A proxy mechanism for handling undetermined values
- A lot of un-buffered data
- A single-threaded runtime

All the way asynchronous

Pretty popular

But PHP is well-suited to the needs of asynchrony despite not offering it out-of-the-box

The asynchronous PHP landscape

Tool	Distribution	Resemblances
ReactPHP	Composer	Node.JS
Amp	Composer	Go, Node.JS
Swoole	PECL	Go, Node.JS

The original React

What is React?

A suite of packages - based on the Reactor pattern - intended to enable event-driven programming in PHP.

- Event loop

What is React?

A suite of packages - based on the Reactor pattern - intended to enable event-driven programming in PHP.

- Event loop
- Stream abstraction

What is React?

A suite of packages - based on the Reactor pattern - intended to enable event-driven programming in PHP.

- Event loop
- Stream abstraction
- HTTP client and server

What is React?

A suite of packages - based on the Reactor pattern - intended to enable event-driven programming in PHP.

- Event loop
- Stream abstraction
- HTTP client and server
- Child processes

At the core of many event-driven systems is the event loop.

The event loop

- A low-level dispatcher
- A quasi-scheduler
- Monitors an execution context for events
- Dispatches handler to event
- Can be written in PHP

```php
while ($running) {
  $readable = readEvents(); // readable stream
  $writable = writeEvents(); // writable stream

  $async = stream_select($readable, $writable); // process streams

  foreach ($async as $action) {
    dispatch($action); // dispatch handler for each action
  }
}
```

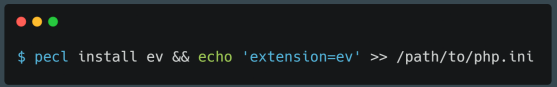
The event loop

- Listens for events and dispatches actions to process them
- Renders everything in its context non-blocking
- Runs until the point of event completion or stoppage

```php
use React\EventLoop\Loop;

$count = 0;

// increment count and print new count every 5 seconds
Loop::addPeriodicTimer(5, function () use (&$count) {
  $count += 1;

  echo $count . PHP_EOL;
});
```

Want more power? Plug in a suitable extension!

- ext-ev
- ext-uv
- ext-event



How is data conveyed in an event-driven system?

How is data conveyed in an event-driven system?

Usually, as a stream...

Streams

- Typically un-buffered sequences of data
- Can be connected in pipelines
- Readable (like STDIN)

```php
use React\Stream\ReadableResourceStream;

$readable = new ReadableResourceStream(STDIN);

// print data once it is received (data event)
$readable->on('data', function (?string $chunk) {
  echo $chunk . PHP_EOL;
});
```

Streams

- Typically un-buffered sequences of data
- Can be connected in pipelines
- Writable (like STDOUT)

```php
use React\Stream\ReadableResourceStream;
use React\Stream\WritableResourceStream;

$readable = new ReadableResourceStream(STDIN);
$writable = new WritableResourceStream(STDOUT);

$readable->pipe($writable);
```

Streams

- Typically un-buffered sequences of data
- Can be connected in pipelines
- Duplex (like TCP/IP or file in read/write mode)

```php
use React\Stream\DuplexResourceStream;

$stream = new DuplexResourceStream(
  fopen('path/to/file', 'w+'),
);

$stream->write('Hello');
```

What about data propagation and action chains?

What about data propagation and action chains?

How about promises?

Promises

- Placeholders for values yet to be computed
- Possess two tracks (resolve, reject)
- Algebraic structures

```php
use React\Promise\Promise;

$promise = new Promise(
  function (callable $resolve, callable $reject) {
    // either resolve or reject an arbitrary action
  },
);

$promise->then(
  function ($success) {
    // success handler (invoked after resolving value)
  },
  function ($failure) {
    // failure handler (invoked upon rejection)
  },
);
```

How about something practical?

How about something practical?

Like a simple socket-powered chat?

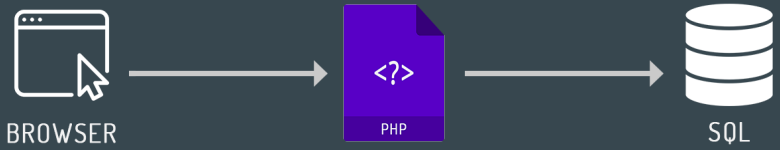
Sockets

- Suitable for network communications
- Useful in client-server setups
- Data typically conveys as a duplex stream

```php
use React\Socket\Connector;
use React\Stream\ReadableResourceStream;
use React\Stream\WritableResourceStream;

$readable = new ReadableResourceStream(STDIN);
$writable = new WritableResourceStream(STDOUT);

$connector = new Connector;

$connector
  ->connect('<address>:<port>')
  ->then(
    // send input to server and print result
    fn ($conn) => $readable->pipe($conn)->pipe($writable),
    fn ($err) => $writable->write($err->getMessage()),
  );
```

telnet <addressss> <port>

Trying to set up an HTTP server? PHP is all you need.




```
$ composer require react/http
```

Server

- Pure PHP
- PSR-compliant software
- Reliably fast
- Multiple integrations with existent PHP packages

```php
use React\Http\HttpServer;
use React\Http\Message\Response;
use React\Socket\SocketServer;
use Psr\Http\Message\ServerRequestInterface as Request;

$http = new HttpServer(
  fn (Request $request) =>
    new Response(
      200,
      ['content-type' => 'text/plain'],
      'Hello world!',
    ),
);

$socket = new SocketServer('127.0.0.1:8080');
$http->listen($socket);
```

Client

- Promise-driven HTTP client
- Akin to JavaScript's fetch API
- Easy to use
- Also PSR-compliant

```php
use React\Http\Browser;
use Psr\Http\Message\ResponseInterface as Response;

$browser = new Browser;

$req = $browser->get('http://host/path')->then(
  function (Response $response) {
    echo $response->getBody()->getContents() . PHP_EOL;
  },
);
```

A neat microframework built atop ReactPHP

```
$ composer require clue/framework-x:dev-main
```

framework-x

- Created and maintained by clue.engineering
- Process all kinds of data (.json, .csv, .xml etc)
- Run in any environment
- Go from RAD to production in minutes

```php
use FrameworkX\App;
use React\Http\Message\Response;
use Psr\Http\Message\ServerRequestInterface as ServerRequest;

$app = new App;

$app->get('/', fn () => new Response(200, [], 'Hello world'));

$app->get(
  '/users/{name}',
  fn (ServerRequest $request) =>
    new Response(200, [], 'Hello, ' . $request->getAttribute('name')),
);

$app->run();
```

Symfony & React

```
$ composer create-project drift/skeleton -sdev
```

DriftPHP

- Created and maintained by Marc Morera
- Non-blocking Symfony kernel
- Promise-driven controllers
- Asynchronous components (command bus, file watcher etc)

```php
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
use function React\Promise\resolve;

class HelloWorldController
{
  public function __invoke(Request $request)
  {
    return resolve(
      new JsonResponse(['message' => 'Hello World'], 200),
    );
  }
}
```

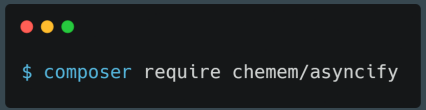
So, you want to run blocking code in an event-driven system?

So, you want to run blocking code in an event-driven system?

This is conventionally a no-no but...

So, you want to run blocking code in an event-driven system?

This is conventionally a no-no but...



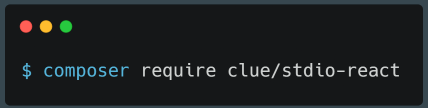
Synchronous to asynchronous

- Works on many non-blocking PHP functions
- Utilizes child-process I/O
- Supports FP and traditional OO approaches

```php
use React\EventLoop\Loop;
use function Chemem\Asyncify\call;

$call = call(Loop::get());

$exec = $call('file_get_contents', ['path/to/file'])->then(
  function (?string $contents) {
    echo $contents . PHP_EOL;
  },
  function (Throwable $err) {
    echo $err->getMessage() . PHP_EOL;
  },
);
```

How about applications that run in the console?

Also considered user-facing software



Shells

- Text input-driven
- Read->Evaluate->Print->Loop
- Can benefit from the potency of non-blocking I/O

```php
use Clue\React\Stdio\Stdio;

$stdio = new Stdio;
$stdio->setPrompt('>>> ');

$stdio->on('data', function (?string $line) use ($stdio) {
  $data = rtrim($line, "\r\n");

  // process data arbitrarily here and convey output
  $stdio->write('-> ' . $data . PHP_EOL);

  // terminate REPL when user inputs 'exit'
  if ($line === 'exit') {
    $stdio->end();
  }
});
```

ReactPHP has a vibrant ecosystem

Check out the ReactPHP wiki

ReactPHP has a vibrant ecosystem

Check out the ReactPHP wiki

- It is growing

ReactPHP has a vibrant ecosystem

Check out the ReactPHP wiki

- It is growing
- Package updates are regularly released

Additional Material

- ReactPHP documentation
- Asynchronous Programming in PHP
- Learning Event-Driven PHP with ReactPHP
- Entries in Sergey Zhuk's blog

Please give asynchronous PHP a try. You likely won’t regret it!

Please give asynchronous PHP a try. You likely won’t regret it!

- Write a simple REST API

Please give asynchronous PHP a try. You likely won’t regret it!

- Write a simple REST API
- Write a simple shell

Please give asynchronous PHP a try. You likely won't regret it!

- Write a simple REST API
- Write a simple shell
- Write a basic asynchronous I/O script

Thank you