www.phparch.com

July 2015
VOLUME 14 - ISSUE 7

php[architect]

ROAD TRIP

ALSO INSIDE

Leveling Up:
How to Burn the Candle at Both Ends

Education Station:
Hands-on Dependency Injection in PHP with Laravel IoC and Aura.Di

Community Corner:
July 2015

finally{}:
Conflict in the Community

Get Started with Silex

Solr Power for All

Writing Asynchronous Code in PHP with Icicle

From PHP to Objective-C

FREE Article!

Writing Asynchronous Code in PHP with Icicle

Aaron Piotrowski

PHP is normally used to write synchronous code that is run on a per-request basis within a web server. However, PHP can also be used to create stand-alone long-running programs. These programs often need to handle many clients or tasks at once without blocking on a single task. Asynchronous operations allow many tasks to be performed cooperatively without blocking, but PHP does not immediately lend itself to asynchronous programming. Icicle is a library to facilitate writing asynchronous code using synchronous coding techniques to create asynchronous programs written using only PHP.

Asynchronous programming has been popularized in the last few years, particularly by node.js, a server-side interpreter for JavaScript. Asynchronous programs use non-blocking I/O to create a single thread of execution that can continuously run available tasks without waiting for an external operation to complete. Asynchronous code can be difficult to write and debug due to its reliance on callback functions that generally cannot return values or throw exceptions.

Icicle (https://github.com/icicleio/Icicle) is a library for writing asynchronous code in PHP that does more than simply enable asynchronous programming. Icicle uses promises to create cooperative coroutines that allow programmers to use synchronous coding techniques to write asynchronous code.

What is Asynchronous Programming?

A *synchronous program* defines a set of sequential instructions (statements, function calls, etc.) that are executed in order, from top to bottom. If data is needed from a resource outside of that program, such as accessing a file or making a network request, the program waits until the external operation has completed before continuing execution. This is called a blocking request, since the execution of the program is blocked until the external operation has completed.

PHP scripts are generally written using blocking requests. For example, calling the function `file_get_contents()` to fetch the contents of a file

DisplayInfo()

Requirements:

- PHP 5.5+
- Composer—http://getcomposer.org

Related URLs:

- Homepage—https://icicle.io
- Source—https://github.com/icicleio/Icicle
- Documentation—https://github.com/icicleio/Icicle/wiki

will block execution of the script until the operation has completed. While the process is blocked, no other code can be run within that PHP process.

Asynchronous programs rely on non-blocking code to continuously process available tasks within a single thread of execution. Operations are only made with data that are immediately available. Because all data required by most programs cannot be immediately available, requests need to be made for data outside the program. Asynchronous programs must then use a different strategy for external operations that would normally cause the program to block. To avoid blocking, asynchronous programs use functions that also accept a callback function that is executed once the external request has been completed. Instead of blocking until the request is completed, the program is able to continue execution even though the result of the request is not available. The example b shows an example of such a function from node.js that resolves the IP address of a domain.

```
dns.resolve('example.com', 'A', function (err, addresses) {
    // Callback invoked when operation completes or fails.
});
// Code below is executed immediately.
```

Execution does not block on the call to `dns.resolve()`, rather the function only initiates the operation and returns immediately. Any code after the call to `dns.resolve()` is immediately executed, before the DNS query has completed. The result of the DNS query will not be available until the callback passed to the function is invoked when the query has completed.

Asynchronous programs rely on an event loop that schedules tasks and invokes callbacks when an event occurs (known as the reactor pattern). An event might be the completion of an external task, available data on a network socket, or expiration of a timer. Event callbacks are executed in an unpredictable order because they often depend on the timing of external operations. The call to `dns.resolve()` in Listing 1 is really scheduling a series of tasks in the event loop that will resolve the DNS query. The final step in this series of tasks will be to invoke the callback function with a list of IP addresses (or with an error).

Creating an Event Loop

PHP does not include an event loop implementation, so to build an asynchronous framework such as Icicle, an event loop must be created from pieces available in the language. Fortunately, the PHP core includes all the components necessary to create an event loop, no extensions required! (There are some extensions available to create event loops that are more performant; more on this later.)

An event loop needs to provide some essential functionality to build an asynchronous program. This includes—but is not limited to—polling network sockets for available data and executing timers along with scheduling and invoking callback functions. The `stream_select()` function included in PHP uses the `select()` system call to poll stream sockets for available data or the ability to write to the stream socket. This function accepts arrays of stream sockets and a timeout, then blocks until either one of the streams can be read from or written to without blocking or until the timeout has expired. The timeout parameter given to `stream_select()` is based on other conditions in the event loop. If there are other events pending in the loop, the timeout can be `0` to quickly poll for stream socket data, returning immediately from `stream_select()`. If there are timers in the event loop, the

timeout parameter can be set to the remaining time on the next pending timer. `stream_select()` may return before the timeout expires if there are data available on a stream socket, but it will not block longer than the timeout given. If there are no timers, the timeout may be `null`, causing `stream_select()` to block indefinitely.

One scenario cannot be covered using `stream_select()` alone: if there are no stream sockets in the event loop waiting to read or write, but there are pending timers. In this case, the `time_nanosleep()` function is used to sleep the process until the next pending timer expires.

Icicle combines `stream_select()` and `time_nanosleep()` to create an event loop that will work on any installation of PHP. Listing 1 contains pseudo-code based on the event loop implementation in Icicle that uses these two functions.

LISTING 1

```
<?php
/*
 * $poll and $await are arrays of stream sockets to poll for
 * data or space to write. $timeout is null or the maximum
 * number of seconds to block.
 */

if ($poll || $await) {
    $seconds = (int)$timeout;
    $microseconds = ($timeout - $seconds) * 1e6;

    $read = $poll;
    $write = $await;
    $except = null;

    $count = stream_select(
        $read,
        $write,
        $except,
        null === $timeout ? null : $seconds,
        $microseconds
    );

    if ($count) {
        // $read and $write modified to contain only stream
        // sockets with pending data or space to write.
        // Invoke callbacks associated with stream sockets.
    }
} elseif (0 < $timeout) {
    $seconds = (int)$timeout;
    $nanoseconds = ($timeout - $seconds) * 1e9;
    time_nanosleep($seconds, $nanoseconds);
}
```

The code in Listing 1 provides some insight into how the core components of PHP can be used to create an event loop but is only a small portion of an entire event loop implementation. Listing 1 provides no details on how callbacks are associated with stream sockets or how timers and scheduled functions are invoked. If this interests you, please take a look at the source of the Loop component of Icicle at https://github.com/icicleio/Icicle/tree/master/src/Loop.

There are two PHP extensions currently supported by Icicle that can provide a more performant event loop: *event* and *libevent*. Both are based on the *libevent* event notification library, which must also be installed to compile either PHP extension. These extensions move much of the event loop logic from PHP code to C code, improving performance. These extensions also use a faster internal mechanism to poll sockets for data compared to `select()`, such as `kqueue()` or `poll()`, further improving performance.

Getting Started

Icicle can be installed with Composer by adding the `icicleio/icicle` package to your project requirements using the command `composer require icicleio/icicle` (or similar depending the path to Composer on your system). This package contains all the basic components necessary to write an asynchronous program in PHP, including an event loop and additional tools to make writing asynchronous code easier, which will be examined in the following sections.

The code snippet below shows how an executable PHP script can be created with Icicle that can be run from the command line or as a daemon.

```
#!/usr/bin/env php
<?php
require 'vendor/autoload.php';
// Create server or initial tasks.
Icicle\Loop\Loop::run(); //Run event loop.
```

A script using Icicle should first create a server or an initial set of tasks, then call `Icicle\Loop\Loop::run()` to run the event loop. This method does not return until the event loop is stopped or there are no pending tasks in the event loop.

The class `Icicle\Loop\Loop` acts as an accessor to the active event loop instance, providing methods for polling sockets and creating tasks in the event loop. An event loop instance is automatically created based on available extensions, but automatic creation can be overridden using `Icicle\Loop\Loop::init()` if your application requires a specific or custom event loop implementation.

For more on installation and on using Icicle, please see the documentation at https://github.com/icicleio/Icicle/wiki.

Promises

Asynchronous programs can be difficult to write and debug, as callback functions that cannot return values or throw exceptions must rely on side effects to control program flow. A callback function for a single operation can be easy to write, but what happens when another asynchronous operation is initiated in a callback function that then invokes another callback function when it completes, and then that operation initiates another operation that invokes yet another callback function? This results in a set of nested callback functions, often referred to as "callback hell."

Promises offer a solution to not only avoid "callback hell" but also a means to model problems using interdependencies between values synonymous with functional composition in synchronous programming. Instead of accepting a callback function as a parameter, asynchronous operations in components designed for Icicle return promises.

Promises are objects that act as placeholders for the future value of an asynchronous operation. A promise may be in one of three states: pending, fulfilled, or rejected. Pending promises may either be fulfilled or rejected with any value (note that in Icicle, if a promise is rejected with a non-exception, it is encapsulated in an exception.) Once a promise is fulfilled or rejected (resolved), it cannot become pending again, and the resolution value cannot change. A promise may also be resolved with another promise, adopting the state of that promise, fulfilling or rejecting with the same value as the resolving promise.

LISTING 2

```
01. <?php
02. $promise2 = $promise1->then(
03.     function ($value) {
04.         // Executed if $promise1 is fulfilled.
05.         // Fulfills or rejects $promise2.
06.     },
07.     function (Exception $exception) {
08.         // Executed if $promise1 is rejected.
09.         // Fulfills or rejects $promise2.
10.     }
11. );
```

Callback functions are the primary way of accessing the resolution value of promises. Unlike other APIs that use callbacks, *promises provide an execution context for callback functions, allowing them to return values and throw exceptions*. Callback functions are registered to a promise using the `then()` method (`PromiseInterface` refers to `Icicle\Promise\PromiseInterface`):

```
PromiseInterface PromiseInterface::then(
    callable $onFulfilled = null,
    callable $onRejected = null
);
```

This method accepts two callback functions: the first is executed if the promise is fulfilled and the second if the promise is rejected. Each callback is given a single parameter, either the fulfillment value or rejection reason (exception). `then()` returns a new promise that is fulfilled with the return value of the invoked callback or rejected with the exception thrown from the invoked callback. Listing 2 shows an example of a call to `then()` on a promise.

If the on-fulfilled callback is omitted, the promise returned from `then()` will be fulfilled with the same value as that of the parent promise. If the on-rejected callback is omitted, the promise returned from `then()` will be rejected with the same exception as the parent promise.

Calls to `then()` can be chained together to create a sequence of interdependent operations in a *time-independent* way, as registered callback functions are only invoked once a promise is resolved. If a promise has already been resolved when a callback is registered with `then()`, the callback will still be invoked with the resolution value of the promise. Either of the two callbacks may also be omitted when calling `then()`.

Icicle promises also include several other methods for registering callbacks with different behaviors. A few of the more important and useful methods are listed below.

- `done(callable $onFulfilled = null, callable $onRejected = null)`: Similar to `then()` but returns nothing instead of another promise. Callbacks registered with `done()` should consume the fulfillment value or handle rejection, as return values are ignored and any exceptions thrown from a callback registered with `done()` cannot be caught.
- `capture(callable $onRejected)`: Registers a callback function to handle rejection. If a type-hint is given for the exception parameter, the callback function will only be invoked if the rejection exception type matches the type-hint. Acts like the `catch` portion of a `try/catch` block.
- `cleanup(callable $onResolved)`: Called when the promise is resolved (either fulfilled or rejected). Acts like the `finally` portion of a `try/catch/finally` block.

Listing 3 shows a simple example of how calls to methods on promises can be chained together to transform a value and handle errors.

LISTING 3

```
<?php
$promise
    ->then(function ($value) {
        if (0 === $value) {
            throw new Exception('Value cannot be 0.');
        }
        return 100 / $value;
    })
    ->then(function ($value) {
        return $value * $value;
    })
    ->then(function ($value) {
        return log($value);
    })
    ->capture(function (Exception $exception) {
        return 0;
    })
    ->done(function ($value) {
       printf("Value: %f\n", $value);
    });
```

Note that omitting the on-rejected callback from calls to `then()` in Listing 3 allow errors to propagate down the chain to the callback defined in the call to `capture()`. Structuring a promise chain in this way is analogous to a `try/catch` block in synchronous code.

LISTING 4

```
<?php
use Icicle\Dns\Executor\Executor;
use Icicle\Dns\Resolver\Resolver;
use Icicle\Socket\Client\Connector;

$connect = function ($domain, $port) {
    $resolver = new Resolver(new Executor('8.8.8.8'));

    // Method below returns a promise.
    $promise1 = $resolver->resolve($domain);

    $promise2 = $promise1->then(
        function (array $ips) use ($port) {
            $connector = new Connector();
            // Method below returns a promise.
            // $promise2 adopts state of returned promise.
            return $connector->connect($ips[0], $port);
        }
    );

    return $promise2;
};

$promise = $connect('example.com', 80);
```

The callback functions in Listing 3 are simple and only meant to demonstrate how method calls on promises may be chained together. Each callback could initiate another asynchronous operation, returning a promise that would resolve the promise originally returned from `then()` or `capture()`. This allows multiple asynchronous operations to be interdependent without creating a tree of nested, imperative callbacks. Registering callback functions with `then()` simply defines dependencies between operations; it does not imply anything about *when* a value will be available. The order in which operations are executed is determined from these dependencies, similar to functional composition in a synchronous program.

Listing 4 demonstrates a more practical use of promises. This example uses the promise-based DNS and socket components of Icicle to asynchronously resolve the IP addresses for a domain name and then connects to the first

of the resolved IP addresses.

`$promise1` in Listing 4 will either be fulfilled with an array of IP addresses or rejected if resolving the domain fails. When `$promise1` fulfills, the on-fulfilled callback function registered to `$promise1` will be invoked, fulfilling `$promise2` with the connected client socket if connecting to the IP address succeeds, otherwise rejecting `$promise2` if connecting fails. If `$promise1` is rejected, `$promise2` will be immediately rejected without invoking the callback function, so no connection attempt will be made.

Coroutines

Promises provide an execution context for callback functions and a means to define interdependencies between values, but they do not eliminate the need to create callback functions. To make using promises simpler and avoid registering callback functions to promises, Icicle combines promises and generators to create interruptible functions called coroutines.

Generators usually use the `yield` keyword to yield a value from a set to implement an iterator. Generators written to be coroutines use the `yield` keyword to define interruption points, temporarily interrupting execution of the coroutine. The local scope of the coroutine is preserved between interruptions, so local variables maintain their value and execution resumes at the point at which it was interrupted. Coroutines are also cooperative, allowing tasks such as I/O, timers, and other coroutines to run whenever a value is yielded from a coroutine.

When a coroutine yields a promise, execution of the coroutine is interrupted and does not resume until the promise is resolved. Once the promise resolves, the fulfillment value will be sent to the generator, or the exception used to reject the promise will be thrown into the generator. This means if a yielded promise is fulfilled, the statement that yielded the promise will evaluate to the fulfillment value of the promise. For example, `$value = (yield $promise);` will set `$value` to the fulfillment value of `$promise` when the coroutine resumes. If a yielded promise is rejected, the statement that yielded the promise would behave identically to a `throw` statement that threw the exception used to reject the promise. *No callbacks need to be registered on the yielded promise.* The fulfillment value of the promise can be accessed through a simple variable assignment to the yield statement. Exceptions rejecting the promise are thrown into the coroutine and can be caught using `try/catch` blocks.

Coroutines in Icicle are also promises. A coroutine is fulfilled with the last value yielded (or fulfillment value of the last yielded promise) and rejected if an exception is thrown from the coroutine's generator. (Note that generators in PHP 7 will be able to explicitly return values and will be used in the future to return values from coroutines.) A coroutine may then yield to other coroutines, interrupting execution of the calling coroutine until the yielded coroutine has completed execution. If the coroutine throws an exception (is rejected), the exception will be thrown into the calling coroutine. This allows coroutines to be composed of other coroutines, allowing coroutines to be built using functional composition. Calling a coroutine within another coroutine is similar to synchronously calling a function that can return a value or throw an exception. Coroutines may also yield generators directly to create another coroutine and automatically yield to that coroutine, removing the need to explicitly create a coroutine from a generator within another coroutine.

Listing 5 demonstrates how the code using promises in Listing 4 can be rewritten into a coroutine to avoid registering callbacks on promises.

Instead of registering a callback to access the fulfillment value of the promise returned by `$resolver->resolve()`, the array of IP addresses is simply assigned to `$ips` when the promise is fulfilled. If the promise is rejected, the exception will be thrown into the coroutine, bypassing the remaining code and immediately rejecting the coroutine.

LISTING 5

```
01. <?php
02. use Icicle\Coroutine\Coroutine;
03. use Icicle\Dns\Executor\Executor;
04. use Icicle\Dns\Resolver\Resolver;
05. use Icicle\Socket\Client\Connector;
06.
07. $connect = function ($domain, $port) {
08.     $resolver = new Resolver(new Executor('8.8.8.8'));
09.     $ips = (yield $resolver->resolve($domain));
10.
11.     $connector = new Connector();
12.     yield $connector->connect($ips[0], 80);
13. };
14.
15. $coroutine = new Coroutine($connect('example.com', 80));
```

Because coroutines make promises much easier to use, Listing 5 can be quickly improved with a loop to attempt to connect to other IP addresses resolved for the domain if the first does not succeed. Creating loops based on promise fulfillment values or rejection reasons is considerably simpler in a coroutine versus using promises alone. Listing 6 shows how a simple `foreach` loop can be used in a coroutine with loop termination determined by the resolution of a promise.

LISTING 6

```
01. <?php
02. use Icicle\Coroutine\Coroutine;
03. use Icicle\Dns\Executor\Executor;
04. use Icicle\Dns\Resolver\Resolver;
05. use Icicle\Socket\Client\Connector;
06.
07. $connect = function ($domain, $port) {
08.     $resolver = new Resolver(new Executor('8.8.8.8'));
09.     $ips = (yield $resolver->resolve($domain));
10.
11.     $connector = new Connector();
12.     foreach ($ips as $ip) {
13.         try {
14.             yield $connector->connect($ip, 80);
15.             return; // Halts coroutine execution.
16.         } catch (Exception $exception) {
17.             // Ignore connection failure and try next IP.
18.         }
19.     }
20.
21.     // Could not connect to any IP, so reject coroutine.
22.     throw new Exception(
23.         sprintf('Error connecting to %s:%d', $domain, $port)
24.     );
25. };
26.
27. $coroutine = new Coroutine($connect('example.com', 80));
```

Example: RESTful DNS Service

Icicle is designed to make creating web services quick and easy. Listing 7 contains a complete PHP script that implements a simple RESTful DNS service. The service accepts `GET` requests for URIs of the form `/{domain-name}/{record-type}` (e.g., `http://localhost:8053/example.com/a`), performs the corresponding DNS query, and responds with the results in JSON format.

Although it would be better to delegate the tasks in the code in Listing 7 to separate functions or methods of a class, this example is meant to demonstrate how a simple yet powerful server can be created with extremely little code. The server in Listing 7 is capable of handling many clients simultaneously because all the tasks necessary to process a client request are performed asynchronously and cooperatively.

Going Forward

Icicle includes all the basic components needed to create an asynchronous network server or client written in only PHP. Writing truly asynchronous code means never using any code that will result in a blocking call. Unfortunately, most of the functions available in PHP that access an external data source will block because they were designed to be used in sequential, synchronous code. Currently, this represents the biggest hurdle to writing asynchronous code in PHP. However, there are many fantastic libraries available for PHP that do not make blocking calls and can be used within asynchronous code, such as dependency injection containers, collection libraries, validators, routers, and many others.

To get the most out of Icicle, asynchronous-compatible components are needed to replace any operations that would block. At the time of this writing, only two additional components are available for Icicle:

- **DNS:** Asynchronous DNS query resolver.
- **HTTP:** Create an asynchronous HTTP server or perform asynchronous HTTP requests.

LISTING 7

```
01. #!/usr/bin/env php
02. <?php
03. require __DIR__ . '/vendor/autoload.php';
04.
05. use Icicle\Dns\Exception\FailureException;
06. use Icicle\Dns\Exception\InvalidTypeException;
07. use Icicle\Dns\Exception\MessageException;
08. use Icicle\Dns\Executor\Executor;
09. use Icicle\Http\Message\RequestInterface;
10. use Icicle\Http\Message\Response;
11. use Icicle\Http\Server\Server;
12. use Icicle\Loop\Loop;
13.
14. $executor = new Executor('8.8.8.8');
15.
16. $server = new Server(function (RequestInterface $request)
17. use ($executor) {
18.     $response = new Response();
19.     $response = $response->withHeader(
20.         'Content-Type',
21.         'application/json'
22.     );
23.
24.     if ($request->getMethod() !== 'GET') {
25.         yield $response->getBody()->end(
26.             json_encode(['error' => 'Only GET allowed.'])
27.         );
28.         yield $response->withStatus(405);
29.         return;
30.     }
31.
32.     if (!preg_match(
33.         '/^\/((?:[a-z0-9\-]+\.)*[a-z]{2,})\/([a-z0-9]+)$/i',
34.         $request->getRequestTarget(),
35.         $matches
36.     )
37.     ) {
38.         yield $response->getBody()->end(
39.             json_encode(['error' => 'Invalid uri format.'])
40.         );
41.         yield $response->withStatus(404);
42.         return;
43.     }
44.
45.     list(, $domain, $type) = $matches;
46.
47.     try {
48.         $message = (
49.         yield $executor->execute($domain, $type)
50.         );
51.     } catch (InvalidTypeException $e) {
52.         yield $response->getBody()->end(
53.             json_encode(['error' => 'Invalid record type.'])
54.         );
```

Continued Next Page ▶

Additional components listed below are currently planned. If you are interested in contributing to any of these components or would like to propose another component, please contact the project on Twitter @icicleio or on GitHub at https://github.com/icicleio or send an e-mail to hello@icicle.io.

- **Process:** Runs a command in a separate process that can be communicated with asynchronously. Can be used to run a process that makes blocking calls alongside an asynchronous process.
- **File System:** Read and write files asynchronously.
- **Web Socket:** Implements the web socket protocol to create asynchronous web socket servers and clients.
- **Memcached:** Asynchronous client to memcached.
- **Redis:** Asynchronous client to Redis.
- **MySQL:** Asynchronous client for MySQL.

LISTING 7 (CONT'D)

```
55.         yield $response->withStatus(404);
56.         return;
57.     } catch (FailureException $e) {
58.         yield $response->getBody()->end(
59.             json_encode(['error' => 'Invalid domain name.'])
60.         );
61.         yield $response->withStatus(404);
62.         return;
63.     } catch (MessageException $e) {
64.         yield $response->getBody()->end(
65.             json_encode(['error' => 'DNS lookup failed.'])
66.         );
67.         yield $response->withStatus(503);
68.         return;
69.     }
70.
71.     $json = [];
72.
73.     foreach ($message->getAnswerRecords() as $record) {
74.         $json[] = [
75.             'type' => $record->getType(),
76.             'ttl' => $record->getTtl(),
77.             'rdata' => (string)$record->getData()
78.         ];
79.     }
80.
81.     yield $response->getBody()->end(json_encode($json));
82.
83.     yield $response->withStatus(200);
84. });
85.
86. $server->listen(8053, '127.0.0.1');
87. Loop::run();
```

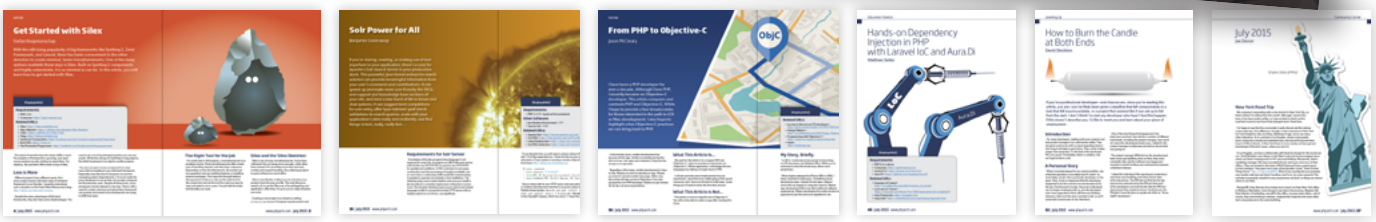
AARON PIOTROWSKI is an avid computer programmer, web designer, photographer, and coffee lover. He first obtained a BS in Biochemistry and Biotechnology, then an MS in Computer Science. He started programming with PHP3 in 1999 and continues to use PHP more than any other language.

Twitter: @trowski2002

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.



php[architect]

magazine

books

conferences

training

www.phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.