orolyn Documentation

Release latest

Feb 11, 2023

CONTENTS

CHAPTER

ONE

DEVELOPER GUIDE

1.1 Overview

1.1.1 Requirements

1. Linux x86_64
2. PHP 8.1+
3. Datastructures
4. Eio for File and FileStream
5. PCNTL
6. Zlib

1.1.2 Installation

Use Composer to install this package.

```
# Install Composer
curl -sS https://getcomposer.org/installer | php
```

Add Orolyn as a dependency:

```
composer require orolyn/orolyn
```

To use without composer, you will need require the autoload.php file:

```
require_once 'path/to/orolyn/autoload.php';
```

1.1.3 License

This library uses the GNU General Public License 3.0

1.2 Concurrent Operations

The Orolyn library uses fibers as its sole means of performing concurrent operations. This is represented in the form of tasks, and the functionality is exposed by a group of functions which interfaces with the *TaskScheduler*.

The aim of this component, and of the library itself, is to enable performing asynchronous operations in a familiar synchronous fashion.` While PHP has many tools and extensions available for performing normally blocking operations asynchronously, the aim here is to perform these operations in a synchronous fashion, i.e. without the use of callbacks or polling.

Part of this is achieved by designing the provided IO components in such a way so to hide the polling operations, and secondly by employing a task scheduler which loops over fibers.

1.2.1 Creating an asynchronous task

```
use function Orolyn\Lang\Async;

Async(fn () => var_dump('Hello, World!'));
```

In this example, we are creating an asynchronous task which outputs *"Hello, World!"*. This internal function executes immediately, so currently there isn't any need for it to be asynchronous. However, we can add suspensions to the closure to better demonstrate what the *Async* function does.

```
use function Orolyn\Lang\Async;
use function Orolyn\Lang\Suspend;

$task = Async(
    function () {
        var_dump('Hello, World');
        Suspend();
        var_dump('Goodbye, World');
    }
);

var_dump('Something in between');

$task->wait();
```

Listing 1: Output

```
string(12) "Hello, World"
string(20) "Something in between"
string(14) "Goodbye, World"
```

So calling *Async* will execute the closure up until it hits the first *Suspend*.Then a call to ->*wait* or ->*getResult* will continue the closure until completion, or a call to ->*resume* will continue until the next *suspend*.

1.2.2 Running multiple asynchronous tasks

Here is an example of running more than one tasks at once:

```
Await(
    Async(
        function () {
            var_dump('Human: Hello, World!');
            Suspend();
            var_dump('Human: Goodbye, World!');
        }
    ),
    Async(
        function () {
            var_dump('World: Hello, Human!');
            Suspend();
            var_dump('World: Goodbye, Human!');
        }
    )
);
```

Listing 2: Output

```
string(20) "Human: Hello, World!"
string(20) "World: Hello, Human!"
string(22) "Human: Goodbye, World!"
string(22) "World: Goodbye, Human!"
```

Note: As of writing, nested *Await* calls will block all other asynchronous tasks when used outside of a task scheduler managed application. Solution coming soon.

We can see that the loop alternates between the closures on suspend.

1.2.3 Using asynchronous tasks for IO operations

So, these have been simple examples, however as mentioned, the rest of this library has been designed to perform synchronous-like operations in such as way so to release control of the current stack when they hit an IO block. For example, a stream which is being read from, might not immediately have available data.

Here we will make 20 consecutive calls to Stackoverflow. Firstly, the setup function which will make the call:

```
function make_request(string $domain): string
{
    $request = <<<EOF
GET / HTTP/1.0
Host: {$domain}


EOF;

    $socket = new Socket();
    $socket->connect(new DnsEndPoint($domain, 80));
    $socket->write($request);
    $socket->flush();

    $output = '';

    while (!$socket->isEndOfStream()) {
        $output .= $socket->read();
    }

    return $output;
}
```

Next we will call this function 20 times and measure the time:

```
$time = microtime(true);
$responses = [];

for ($i = 0; $i < 20; $i++) {
    $responses[] = make_request('stackoverflow.com');
}
```

(continues on next page)

(continued from previous page)

```
var_dump(microtime(true) - $time);
```

And the time was around half a second:

Listing 3: Output

```
float(0.5648369789123535)
```

Next we will perform the socket connection and read/writes 20 times concurrently:

```
$time = microtime(true);

$tasks = [];

for ($i = 0; $i < 20; $i++) {
    $tasks[] = Async(fn () => make_request('stackoverflow.com'));
}

Await($tasks);

$responses = [];

foreach ($tasks as $task) {
    $responses[] = $task->getResult();
}

var_dump(microtime(true) - $time);
```

And the time now is much shorter:

Listing 4: Output

```
float(0.04282999038696289)
```

Essentially what this provides is a way to communicate with multiple sockets concurrently and without callbacks. Because, the connection, writing, the checking of connection status, and the reading are performed internally with polls which suspend execution of the current task.

Note: This precise example may yield connection errors because we are sending so many requests to Stackoverflow at once. Sorry!

1.2.4 Managed Application

The managed application is a method of running the main stack inside a task, therefore allowing execution of off-shoot tasks without further interaction.

Simply calling *Async* on the main stack will not result in completion if that closure contains any suspensions unless you call *Await()* or *->wait()* on the task. However within a managed application, the script will continue to execute so long as there are still tasks running.

Todo: Information about the managed application.

1.3 Using Sockets

The sockets components provides a simple interface for making TCP connections. The *Socket* class is itself a stream implementing *IInputStream* and *IOutputStream* allowing for varying data types to be written to and read from the connection.

Like with all Orolyn IO interfaces, calls to the socket's IO bound APIs block execution, but release control of the current task when executed within a concurrent environment.

1.3.1 Making a connection

Connection with sockets are performed by providing the necessary endpoint. For example to connect to an IP address we can use the following example.

Create a netcat server with port *9999*:

```
netcat -k -l 0.0.0.0 9999
```

```
use Orolyn\Net\IPAddress;
use Orolyn\Net\IPEndPoint;
use Orolyn\Net\Sockets\Socket;

$socket = new Socket();
$socket->connect(new IPEndPoint(IPAddress::parse('0.0.0.0'), 9999));

$socket->write("Hello, World!\n");
```

In this example we can also use the *StreamWriter*:

```
use Orolyn\IO\StreamWriter;

$writer = new StreamWriter($socket);
$writer->writeLine('Hello, World!');
$socket->flush();
```

Example:

Listing 5: Server terminal

```
netcat -k -l 0.0.0.0 9999
```

Listing 6: Client terminal

```
php sockets.php
```

Listing 7: Server output

```
Hello, World!
```

Reading data from a socket

We can also read data from the socket. Here is an example of connecting to an HTTP server and making a simple HTTP transaction. For this example we are connecting via a DNS endpoint. To learn more about resolving domain names see documentation on the DnsResolver.

```
$socket = new Socket();
$socket->connect(new DnsEndPoint('google.com', 80));

$message = <<<EOF
GET / HTTP/1.0
Host: google.com


EOF;

$socket->write($message);
$socket->flush();

$response = '';

while (!$socket->isEndOfStream()) {
    if (0 < $available = $socket->getBytesAvailable()) {
        $response .= $socket->read($available);
    }

    usleep(100);
}

echo $response;
```

Listing 8: Client terminal

```
php domain-connect.php
```

Listing 9: Output

```
HTTP/1.0 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Sun, 05 Jun 2022 16:55:14 GMT
Expires: Tue, 05 Jul 2022 16:55:14 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

1.3.2 Concurrency

Guides on concurrency with sockets

1.4 Running an Http Server

1.4.1 Creating an HTTP server

1.4.2 Concurrency

Guides on concurrency with sockets

1.5 Working with Websockets

1.6 Resolving domain names

PHP comes with built-in functions for resolving IP addresses from a domain name:

- *gethostbyname*
- *gethostbynamel*

However the default implementation of these functions block execution. For this reason the Orolyn library includes the *DnsResolver* class.

1.6.1 Using the DNS Resolver

Calling the static method *lookup* will attempt to fetch all IP addresses associated with the domain name. Using *getAddress* on the entry result will fetch the first found IP address:

Example:

```
use Orolyn\Net\DnsResolver;

$entry = DnsResolver::lookup('google.com');

var_dump($entry->getAddress()?->toString());
```

Output:

```
string(14) "142.250.200.14"
```

Getting all IP addresses

Calling *getAddressList* will provide a list of all addresses associated with the domain name:

```
/** @var IList<IPAddress> */
$addresses = $entry->getAddressList();
```

1.6.2 Getting IP addresses asynchronously

Running multiple searches using the concurrency component enables multiple datagram connections to run without blocking each other, for example below we will fetch the first available IP address from each of the following domains:

```
use Orolyn\Net\DnsResolver;
use function Orolyn\Lang\Async;
use function Orolyn\Lang\Await;

Await(
    $task1 = Async(fn () => DnsResolver::lookup('google.com')),
    $task2 = Async(fn () => DnsResolver::lookup('stackoverflow.com')),
    $task3 = Async(fn () => DnsResolver::lookup('readthedocs.org')),
);

var_dump('Task 1: ' . $task1->getResult()->getAddress()->toString());
var_dump('Task 2: ' . $task2->getResult()->getAddress()->toString());
var_dump('Task 3: ' . $task3->getResult()->getAddress()->toString());
```

Listing 10: Output

```
string(23) "Task 1: 142.250.187.206"
string(22) "Task 2: 151.101.193.69"
string(19) "Task 3: 104.18.7.29"
```