# Guzzle Promises

[Promises/A+](https://promisesaplus.com/) implementation that handles promise chaining and resolution iteratively, allowing for "infinite" promise chaining while keeping the stack size constant. Read [this blog post](https://blog.domenic.me/youre-missing-the-point-of-promises/) for a general introduction to promises.

* [Features](#features)
* [Quick start](#quick-start)
* [Synchronous wait](#synchronous-wait)
* [Cancellation](#cancellation)
* [API](#api)
  + [Promise](#promise)
  + [FulfilledPromise](#fulfilledpromise)
  + [RejectedPromise](#rejectedpromise)
* [Promise interop](#promise-interop)
* [Implementation notes](#implementation-notes)

## Features

* [Promises/A+](https://promisesaplus.com/) implementation.
* Promise resolution and chaining is handled iteratively, allowing for "infinite" promise chaining.
* Promises have a synchronous wait method.
* Promises can be cancelled.
* Works with any object that has a then function.
* C# style async/await coroutine promises using GuzzleHttp\Promise\Coroutine::of().

## Installation

composer require guzzlehttp/promises

## Version Guidance

| **Version** | **Status** | **PHP Version** |
| --- | --- | --- |
| 1.x | Security fixes only | >=5.5,<8.3 |
| 2.x | Latest | >=7.2.5,<8.5 |

## Quick Start

A *promise* represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its then method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

### Callbacks

Callbacks are registered with the then method by providing an optional $onFulfilled followed by an optional $onRejected function.

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$promise->then(

// $onFulfilled

function ($value) {

echo 'The promise was fulfilled.';

},

// $onRejected

function ($reason) {

echo 'The promise was rejected.';

}

);

*Resolving* a promise means that you either fulfill a promise with a *value* or reject a promise with a *reason*. Resolving a promise triggers callbacks registered with the promise's then method. These callbacks are triggered only once and in the order in which they were added.

### Resolving a Promise

Promises are fulfilled using the resolve($value) method. Resolving a promise with any value other than a GuzzleHttp\Promise\RejectedPromise will trigger all of the onFulfilled callbacks (resolving a promise with a rejected promise will reject the promise and trigger the $onRejected callbacks).

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$promise

->then(function ($value) {

// Return a value and don't break the chain

return "Hello, " . $value;

})

// This then is executed after the first then and receives the value

// returned from the first then.

->then(function ($value) {

echo $value;

});

// Resolving the promise triggers the $onFulfilled callbacks and outputs

// "Hello, reader."

$promise->resolve('reader.');

### Promise Forwarding

Promises can be chained one after the other. Each then in the chain is a new promise. The return value of a promise is what's forwarded to the next promise in the chain. Returning a promise in a then callback will cause the subsequent promises in the chain to only be fulfilled when the returned promise has been fulfilled. The next promise in the chain will be invoked with the resolved value of the promise.

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$nextPromise = new Promise();

$promise

->then(function ($value) use ($nextPromise) {

echo $value;

return $nextPromise;

})

->then(function ($value) {

echo $value;

});

// Triggers the first callback and outputs "A"

$promise->resolve('A');

// Triggers the second callback and outputs "B"

$nextPromise->resolve('B');

### Promise Rejection

When a promise is rejected, the $onRejected callbacks are invoked with the rejection reason.

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$promise->then(null, function ($reason) {

echo $reason;

});

$promise->reject('Error!');

// Outputs "Error!"

### Rejection Forwarding

If an exception is thrown in an $onRejected callback, subsequent $onRejected callbacks are invoked with the thrown exception as the reason.

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$promise->then(null, function ($reason) {

throw new Exception($reason);

})->then(null, function ($reason) {

assert($reason->getMessage() === 'Error!');

});

$promise->reject('Error!');

You can also forward a rejection down the promise chain by returning a GuzzleHttp\Promise\RejectedPromise in either an $onFulfilled or $onRejected callback.

use GuzzleHttp\Promise\Promise;

use GuzzleHttp\Promise\RejectedPromise;

$promise = new Promise();

$promise->then(null, function ($reason) {

return new RejectedPromise($reason);

})->then(null, function ($reason) {

assert($reason === 'Error!');

});

$promise->reject('Error!');

If an exception is not thrown in a $onRejected callback and the callback does not return a rejected promise, downstream $onFulfilled callbacks are invoked using the value returned from the $onRejected callback.

use GuzzleHttp\Promise\Promise;

$promise = new Promise();

$promise

->then(null, function ($reason) {

return "It's ok";

})

->then(function ($value) {

assert($value === "It's ok");

});

$promise->reject('Error!');

## Synchronous Wait

You can synchronously force promises to complete using a promise's wait method. When creating a promise, you can provide a wait function that is used to synchronously force a promise to complete. When a wait function is invoked it is expected to deliver a value to the promise or reject the promise. If the wait function does not deliver a value, then an exception is thrown. The wait function provided to a promise constructor is invoked when the wait function of the promise is called.

$promise = new Promise(function () use (&$promise) {

$promise->resolve('foo');

});

// Calling wait will return the value of the promise.

echo $promise->wait(); // outputs "foo"

If an exception is encountered while invoking the wait function of a promise, the promise is rejected with the exception and the exception is thrown.

$promise = new Promise(function () use (&$promise) {

throw new Exception('foo');

});

$promise->wait(); // throws the exception.

Calling wait on a promise that has been fulfilled will not trigger the wait function. It will simply return the previously resolved value.

$promise = new Promise(function () { die('this is not called!'); });

$promise->resolve('foo');

echo $promise->wait(); // outputs "foo"

Calling wait on a promise that has been rejected will throw an exception. If the rejection reason is an instance of \Exception the reason is thrown. Otherwise, a GuzzleHttp\Promise\RejectionException is thrown and the reason can be obtained by calling the getReason method of the exception.

$promise = new Promise();

$promise->reject('foo');

$promise->wait();

PHP Fatal error: Uncaught exception 'GuzzleHttp\Promise\RejectionException' with message 'The promise was rejected with value: foo'

### Unwrapping a Promise

When synchronously waiting on a promise, you are joining the state of the promise into the current state of execution (i.e., return the value of the promise if it was fulfilled or throw an exception if it was rejected). This is called "unwrapping" the promise. Waiting on a promise will by default unwrap the promise state.

You can force a promise to resolve and *not* unwrap the state of the promise by passing false to the first argument of the wait function:

$promise = new Promise();

$promise->reject('foo');

// This will not throw an exception. It simply ensures the promise has

// been resolved.

$promise->wait(false);

When unwrapping a promise, the resolved value of the promise will be waited upon until the unwrapped value is not a promise. This means that if you resolve promise A with a promise B and unwrap promise A, the value returned by the wait function will be the value delivered to promise B.

**Note**: when you do not unwrap the promise, no value is returned.

## Cancellation

You can cancel a promise that has not yet been fulfilled using the cancel() method of a promise. When creating a promise you can provide an optional cancel function that when invoked cancels the action of computing a resolution of the promise.

## API

### Promise

When creating a promise object, you can provide an optional $waitFn and $cancelFn. $waitFn is a function that is invoked with no arguments and is expected to resolve the promise. $cancelFn is a function with no arguments that is expected to cancel the computation of a promise. It is invoked when the cancel() method of a promise is called.

use GuzzleHttp\Promise\Promise;

$promise = new Promise(

function () use (&$promise) {

$promise->resolve('waited');

},

function () {

// do something that will cancel the promise computation (e.g., close

// a socket, cancel a database query, etc...)

}

);

assert('waited' === $promise->wait());

A promise has the following methods:

* then(callable $onFulfilled, callable $onRejected) : PromiseInterface

Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler.

* otherwise(callable $onRejected) : PromiseInterface

Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.

* wait($unwrap = true) : mixed

Synchronously waits on the promise to complete.

$unwrap controls whether or not the value of the promise is returned for a fulfilled promise or if an exception is thrown if the promise is rejected. This is set to true by default.

* cancel()

Attempts to cancel the promise if possible. The promise being cancelled and the parent most ancestor that has not yet been resolved will also be cancelled. Any promises waiting on the cancelled promise to resolve will also be cancelled.

* getState() : string

Returns the state of the promise. One of pending, fulfilled, or rejected.

* resolve($value)

Fulfills the promise with the given $value.

* reject($reason)

Rejects the promise with the given $reason.

### FulfilledPromise

A fulfilled promise can be created to represent a promise that has been fulfilled.

use GuzzleHttp\Promise\FulfilledPromise;

$promise = new FulfilledPromise('value');

// Fulfilled callbacks are immediately invoked.

$promise->then(function ($value) {

echo $value;

});

### RejectedPromise

A rejected promise can be created to represent a promise that has been rejected.

use GuzzleHttp\Promise\RejectedPromise;

$promise = new RejectedPromise('Error');

// Rejected callbacks are immediately invoked.

$promise->then(null, function ($reason) {

echo $reason;

});

## Promise Interoperability

This library works with foreign promises that have a then method. This means you can use Guzzle promises with [React promises](https://github.com/reactphp/promise) for example. When a foreign promise is returned inside of a then method callback, promise resolution will occur recursively.

// Create a React promise

$deferred = new React\Promise\Deferred();

$reactPromise = $deferred->promise();

// Create a Guzzle promise that is fulfilled with a React promise.

$guzzlePromise = new GuzzleHttp\Promise\Promise();

$guzzlePromise->then(function ($value) use ($reactPromise) {

// Do something something with the value...

// Return the React promise

return $reactPromise;

});

Please note that wait and cancel chaining is no longer possible when forwarding a foreign promise. You will need to wrap a third-party promise with a Guzzle promise in order to utilize wait and cancel functions with foreign promises.

### Event Loop Integration

In order to keep the stack size constant, Guzzle promises are resolved asynchronously using a task queue. When waiting on promises synchronously, the task queue will be automatically run to ensure that the blocking promise and any forwarded promises are resolved. When using promises asynchronously in an event loop, you will need to run the task queue on each tick of the loop. If you do not run the task queue, then promises will not be resolved.

You can run the task queue using the run() method of the global task queue instance.

// Get the global task queue

$queue = GuzzleHttp\Promise\Utils::queue();

$queue->run();

For example, you could use Guzzle promises with React using a periodic timer:

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

$loop->addPeriodicTimer(0, [$queue, 'run']);

## Implementation Notes

### Promise Resolution and Chaining is Handled Iteratively

By shuffling pending handlers from one owner to another, promises are resolved iteratively, allowing for "infinite" then chaining.

<?php

require 'vendor/autoload.php';

use GuzzleHttp\Promise\Promise;

$parent = new Promise();

$p = $parent;

for ($i = 0; $i < 1000; $i++) {

$p = $p->then(function ($v) {

// The stack size remains constant (a good thing)

echo xdebug\_get\_stack\_depth() . ', ';

return $v + 1;

});

}

$parent->resolve(0);

var\_dump($p->wait()); // int(1000)

When a promise is fulfilled or rejected with a non-promise value, the promise then takes ownership of the handlers of each child promise and delivers values down the chain without using recursion.

When a promise is resolved with another promise, the original promise transfers all of its pending handlers to the new promise. When the new promise is eventually resolved, all of the pending handlers are delivered the forwarded value.

### A Promise is the Deferred

Some promise libraries implement promises using a deferred object to represent a computation and a promise object to represent the delivery of the result of the computation. This is a nice separation of computation and delivery because consumers of the promise cannot modify the value that will be eventually delivered.

One side effect of being able to implement promise resolution and chaining iteratively is that you need to be able for one promise to reach into the state of another promise to shuffle around ownership of handlers. In order to achieve this without making the handlers of a promise publicly mutable, a promise is also the deferred value, allowing promises of the same parent class to reach into and modify the private properties of promises of the same type. While this does allow consumers of the value to modify the resolution or rejection of the deferred, it is a small price to pay for keeping the stack size constant.

$promise = new Promise();

$promise->then(function ($value) { echo $value; });

// The promise is the deferred value, so you can deliver a value to it.

$promise->resolve('foo');

// prints "foo"

## Upgrading from Function API

A static API was first introduced in 1.4.0, in order to mitigate problems with functions conflicting between global and local copies of the package. The function API was removed in 2.0.0. A migration table has been provided here for your convenience:

| **Original Function** | **Replacement Method** |
| --- | --- |
| queue | Utils::queue |
| task | Utils::task |
| promise\_for | Create::promiseFor |
| rejection\_for | Create::rejectionFor |
| exception\_for | Create::exceptionFor |
| iter\_for | Create::iterFor |
| inspect | Utils::inspect |
| inspect\_all | Utils::inspectAll |
| unwrap | Utils::unwrap |
| all | Utils::all |
| some | Utils::some |
| any | Utils::any |
| settle | Utils::settle |
| each | Each::of |
| each\_limit | Each::ofLimit |
| each\_limit\_all | Each::ofLimitAll |
| !is\_fulfilled | Is::pending |
| is\_fulfilled | Is::fulfilled |
| is\_rejected | Is::rejected |
| is\_settled | Is::settled |
| coroutine | Coroutine::of |

## Security

If you discover a security vulnerability within this package, please send an email to [security@tidelift.com](mailto:security@tidelift.com). All security vulnerabilities will be promptly addressed. Please do not disclose security-related issues publicly until a fix has been announced. Please see [Security Policy](https://github.com/guzzle/promises/security/policy) for more information.

## License

Guzzle is made available under the MIT License (MIT). Please see [License File](http://license) for more information.

## For Enterprise

Available as part of the Tidelift Subscription

The maintainers of Guzzle and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use. [Learn more.](https://tidelift.com/subscription/pkg/packagist-guzzlehttp-promises?utm_source=packagist-guzzlehttp-promises&utm_medium=referral&utm_campaign=enterprise&utm_term=repo)