TwirPHP Documentation

Release v0.1.1

Márk Sági-Kazár

Nov 26, 2021

Contents

Twirp is a "a simple RPC framework built on protobuf." Unfortunately (or not?) it only supports Go and Python officially. While in the modern world it may be enough for most of the projects, there is still a considerable number of PHP-based softwares out there.

TwirPHP is a PHP port of Twirp supporting both server and client side. It generates code the same way as Twirp does and follows the same conventions. Because of that this documentation only contains minimal information about how Twirp works internally. To learn more about it, you should check the following resources published by the Twirp developers themselves:

- Official Documentation
- Introductory Post

CHAPTER 1

Overview

Similar to Twirp itself, TwirPHP comes with two components:

- Code generator library (written in Go)
- Shared PHP library

The code generator is used to generate the Twirp specific server and client files. The generated code tries to be as self-contained as possible, keeping the runtime library small. This concept is present in Twirp itself as well. The reason behind is to prevent accidental backward incompatible changes in the shared library break your code. See more about this in the introductory post.

The shared library contains common interfaces and some code that is part of the protocol itself. It can be installed via Composer.

1.1 Versioning

TwirPHP is versioned separately from Twirp to ensure that it's lifecycle does not depend on the original library.

In order to track which version of the Twirp specification is supported, please refer to the TWIRP_VERSION file in the repository.

Twirp version changes will always trigger a new major version, but it might also contain backward incompatible changes of the library itself, so keep an eye on the Change Log.

CHAPTER 2

Installation

You'll need a few things to install TwirPHP:

2.1 Protobuf compiler

`protoc` is used to generate code from protobuf definitions. The easiest way to install it is downloading the precompiled binary from the Github Releases page.

You can also install it via Homebrew on MacOS:

```
$ brew install protobuf
```

2.2 TwirPHP protoc plugin

Just like in case of `protoc`, the easiest way to install the plugin is downloading it from the Github Releases page.

Alternatively you can use the following oneliner to install the plugin:

```
curl -Ls https://git.io/twirphp | bash -b path/to/bin
```

Make sure to save the binary with the same name as it is found in the downloaded archive. Also, make sure you place the binary in your `$PATH`, otherwise you will have to tell `protoc` where you saved the plugin:

```
$ protoc --plugin=protoc-gen-twirp_php=path/to/protoc-gen-twirp_php ...
```

Alternatively you can install the plugin from source. For that you are going to need dep to be installed.

```
$ go get github.com/twirphp/twirp/protoc-gen-twirp_php
$ cd $GOROOT/src/github.com/twirphp/twirp/protoc-gen-twirp_php
$ dep ensure
$ go install
```

The commands above will put the binary in your `$GOBIN` path which is usually a good idea to be included in your `$PATH` prefix.

2.3 Protobuf PHP library

As described in the protobuf PHP library README there are two ways to install protobuf:

- C extension
- native PHP package

The C extension provides better performance obviously, so it is recommended to be used, on the other hand the PHP package provides better portability.

The extension can be installed from the linked repository above or via Pecl:

```
$ sudo pecl install protobuf-{VERSION}
```

The PHP package can be installed via Composer:

```
$ composer require google/protobuf
```

2.4 Shared PHP library

In order to make the generated code work (in a PHP project) you need to install the (minimal) shared library via Composer.

```
$ composer require twirp/twirp
```

2.5 HTTP Client and PSR-7 implementation

The generated code relies on the following standard HTTP interfaces:

- PSR-7 (HTTP Message)
- PSR-15 (HTTP Server Request Handler)
- PSR-17 (HTTP Factory)

- PSR-18 (HTTP Client)

Choosing the right implementations for your project is up to you. If you do HTTP stuff in your project, chances are that some of them are already installed.

An example set of dependencies for server usage:

```
$ composer require guzzlehttp/psr7 http-interop/http-factory-guzzle
```

And for client usage:

```
$ composer require guzzlehttp/psr7 http-interop/http-factory-guzzle php-http/guzzle6-
↪adapter
```

You can find packages that implement the above interfaces on Packagist:

- PSR-7 implementation
- PSR-17 implementation
- psr-18 implementation

2.6 Quickstart

From the above guide it is clear that installing TwirPHP is not trivial. It has multiple components and external dependencies. To make installing these dependencies easier, there is a quickstart metapackage which can be installed via Composer:

```
$ composer require twirp/quickstart
```

It installs:

- the protobuf runtime library
- Guzzle PSR-7 (and it's factories)
- Guzzle HTTP Client

CHAPTER 3

Usage Example

This page contains a full example about using TwirPHP on both server and client side. Much of it is based on the original usage example, so you might want to check that out as well. You can find the code for this example in the demo-project repository.

Before moving forward, make sure to check the *Installation* guide as well.

3.1 Write a Protobuf service definition

Write a protobuf definition for your messages and your service and put it in `proto/service.proto`. The following example is taken from the original Twirp documentation.

```
syntax = "proto3";

package twirp.example.haberdasher;
option go_package = "haberdasher";

// Haberdasher service makes hats for clients.
service Haberdasher {
  // MakeHat produces a hat of mysterious, randomly-selected color!
  rpc MakeHat(Size) returns (Hat);
}
```

(continues on next page)

(continued from previous page)

```
// Size of a Hat, in inches.
message Size {
  int32 inches = 1; // must be > 0
}

// A Hat is a piece of headwear made by a Haberdasher.
message Hat {
  int32 inches = 1;
  string color = 2; // anything but "invisible"
  string name = 3; // i.e. "bowler"
}
```

3.2 Generate code

To generate code run the `protoc` compiler pointed at your service definition file:

```
$ mkdir -p generated
$ protoc -I . --twirp_php_out=generated --php_out=generated ./proto/service.proto
```

This will generate the standard PHP messages along with the Twirp specific files.

3.3 Implement the server

Now that everything is in place, it's time to implement the server implementing the service interface (`Twirp\Example\Haberdasher\Haberdasher` in this case).

```
<?php

namespace Twirp\Demo;

use Twirp\Example\Haberdasher\Hat;
use Twirp\Example\Haberdasher\Size;

final class Haberdasher implements \Twirp\Example\Haberdasher\Haberdasher
{
    private $colors = ['golden', 'black', 'brown', 'blue', 'white', 'red'];

    private $hats = ['crown', 'baseball cap', 'fedora', 'flat cap', 'panama', 'helmet
↪'];

    public function MakeHat(array $ctx, Size $size): Hat
    {
        $hat = new Hat();
        $hat->setInches($size->getInches());
        $hat->setColor($this->colors[array_rand($this->colors, 1)]);
        $hat->setName($this->hats[array_rand($this->hats, 1)]);

        return $hat;
    }
}
```

3.4 Run the server

To run the server you need to setup some sort of application entrypoint that processes incoming requests as PSR-7 messages. It is recommended that you use some sort of dispatcher/emitter, like the `SapiEmitter` bundled with Zend Diactoros, but the following example works perfectly as well:

```
<?php

require __DIR__.'/vendor/autoload.php';

$request = \GuzzleHttp\Psr7\ServerRequest::fromGlobals();

$server = new \Twirp\Server();
$handler = new \Twirp\Example\Haberdasher\HaberdasherServer(new␣
↪\Twirp\Demo\Haberdasher());
$server->registerServer(\Twirp\Example\Haberdasher\HaberdasherServer::PATH_PREFIX,
↪$handler);

$response = $server->handle($request);

if (!headers_sent()) {
    // status
    header(sprintf('HTTP/%s %s %s', $response->getProtocolVersion(), $response->
↪getStatusCode(), $response->getReasonPhrase()), true, $response->getStatusCode());
    // headers
    foreach ($response->getHeaders() as $header => $values) {
        foreach ($values as $value) {
            header($header.': '.$value, false, $response->getStatusCode());
        }
    }
}
echo $response->getBody();
```

3.5 Use the client

Client stubs are automatically generated, hooray!

There are two client stubs generated for each proto service: the default one which uses protobuf serialization, and a JSON client stub. Twirp itself supports both protobuf and json, but it recommends using only protobuf in production.

Using the client is quite trivial, you only need to pass an endpoint to the generated client:

```
<?php

require __DIR__.'/vendor/autoload.php';

$client = new \Twirp\Example\Haberdasher\HaberdasherClient($argv[1]);

while (true) {
    $size = new \Twirp\Example\Haberdasher\Size();
    $size->setInches(10);

    try {
        $hat = $client->MakeHat([], $size);
```

(continues on next page)

(continued from previous page)

```
        printf("I received a %s %s\n", $hat->getColor(), $hat->getName());
    } catch (\Twirp\Error $e) {
        if ($cause = $e->getMeta('cause') !== null) {
            printf("%s: %s (%s)\n", strtoupper($e->getErrorCode()), $e->getMessage(),
↪$cause);
        } else {
            printf("%s: %s\n", strtoupper($e->getErrorCode()), $e->getMessage());
        }
    }

    sleep(1);
}
```

Warning: When no scheme is present in the endpoint, the client falls back to *HTTP*.

CHAPTER 4

Best Practices

This page contains some best practices related to using TwirPHP in general. Make sure to check out the official best practices guide as well for Twirp and protobuf related practices.

4.1 Folder/Package Structure

There are three types of "resources" to consider in case of a PHP projects using TwirPHP:

- proto files
- generated code
- service implementation

Following common PHP packaging practice the recommended folder structure is:

```
/generated
    /<namespace>
        // generated files
/src
    /<namespace>
        // service implementation
/proto
    service.proto
```

4.2 Build tool for code generation

Make sure to properly document how the code generation works.

Even better: use some sort of build tool to collect all proto generation commands. In case of PHP, that tool can be Composer itself.

```
{
    "scripts": {
        "proto": [
            "protoc -I . --twirp_out=generated --php_out=generated proto/service.proto
↪"
        ]
    }
}
```

```
$ composer proto
```

CHAPTER 5

Serving multiple Twirp services together

In some cases you might want to serve not just one, but multiple services from one application. The shared library contains a simple server implementation which lets you mux different services.

```
<?php

$server = new \Twirp\Server();

// register services
$server->registerServer(
    \Twitch\Twirp\Example\HaberdasherServer::PATH_PREFIX,
    new \Twitch\Twirp\Example\HaberdasherServer(
        new \Twirphp\Example\Haberdasher()
    )
);
```

Both the server and service server implement the PSR-15 RequestHandler interface, so you can use the same code as in the *Run the server* usage example:

```
<?php

// ...
$response = $server->handle($request);
```

CHAPTER 6

Custom HTTP Headers

Sometimes, you need to send custom HTTP headers.

From Twirp's perspective, "headers" are just metadata since HTTP is a lower level, transport layer. But since Twirp is primarily used with HTTP, sometimes you might need to send or receive some information from that layer too.

6.1 Client side

6.1.1 Send HTTP Headers with client requests

Use `Twirp\Context::withHttpRequestHeaders` to attach a map of headers to the context:

```php
<?php

// Given a client ...
$client = new \Twitch\Twirp\Example\HaberdasherClient($addr);

// Given some headers ...
$headers = [
    'Twitch-Authorization' => 'uDRlDxQYbFVXarBvmTncBoWKcZKqrZTY',
    'Twitch-Client-ID' => 'FrankerZ',
];

// Attach the headers to a context
$ctx = [];
$ctx = \Twirp\Context::withHttpRequestHeaders($ctx, $headers);

// And use the context in the request. Headers will be included in the request!
$resp = $client->MakeHat($ctx, new \Twitch\Twirp\Example\Size());
```

6.1.2 Read HTTP Headers from responses

Twirp client responses are structs that depend only on the protobuf response. HTTP headers can not be used by the Twirp client in any way.

However, remember that the TwirPHP client is instantiated with a PSR-18 HTTP client, which can be anything that implements the minimal interface. For example you could configure an HTTPlug PluginClient and read the headers in a plugin.

6.2 Server side

6.2.1 Send HTTP Headers on server responses

In your server implementation you can set HTTP headers using `Twirp\Context::withHttpResponseHeader`.

```
<?php

public function MakeHat(array $ctx, \Twitch\Twirp\Example\Size $size): Hat
{
    \Twirp\Context::withHttpResponseHeader($ctx, 'Cache-Control', 'public, max-age=60
↪');

    $hat = new \Twitch\Twirp\Example\Hat();

    return $hat;
}
```

6.2.2 Read HTTP Headers from requests

TwirPHP server methods are abstracted away from HTTP, therefore they don't have direct access to HTTP Headers.

However, they receive the PSR-7 server attributes as the context that can be modified by HTTP middleware before being used by the Twirp method.

For example, lets say you want to read the 'User-Agent' HTTP header inside a twirp server method. You might write this middleware:

```
<?php

use Psr\Http\Message\ServerRequestInterface;

// class UserAgentMiddleware...

public function handle(ServerRequestInterface $request)
{
    $request = $request->withAttribute('user-agent', $request->getHeaderLine('User-
↪Agent'));

    return $this->server->handle($request);
}
```

CHAPTER 7

Using PSR-15

PSR-15 is a standard describing server request handlers and middlewares for PHP.

In some cases Twirp might not be the primary receiver of requests or simply you might want to add some extra logic to the HTTP flow (for example attaching some headers to the Twirp context). Either way, PSR-15 has a great ecosystem around it which helps you with both cases.

7.1 Routing requests

When a Twirp service is not the primary target of requests in an application, you probably want to register it as a sort of "controller" and handle the routing outside of it.

Given an imaginary router, you might write some code like this:

```
<?php

$server = new \Twitch\Twirp\Example\HaberdasherServer(new␣
↪\Twirphp\Example\Haberdasher());

$router->register('POST', \Twitch\Twirp\Example\HaberdasherServer::PATH_PREFIX, [
↪$server, 'handle']);
```

The code above registers the Twirp server as a request handler for it's path prefix in the router. The syntax of course can be different based on the router implementation.

How is this connected to PSR-15? In PHP, routing is often a step of a middleware chain. For example you could use FastRoute together with its middleware.