

# Chapter 11. Streams

*Streams* in PHP represent common interfaces to data resources that can be written to or read from in a linear, continuous manner. Internally, each stream is represented by a collection of objects referred to as *buckets*. Each bucket represents a chunk of data from the underlying stream, which is treated like a digital re-creation of an old-fashioned bucket brigade, like the one illustrated in [Figure 11-1](#).

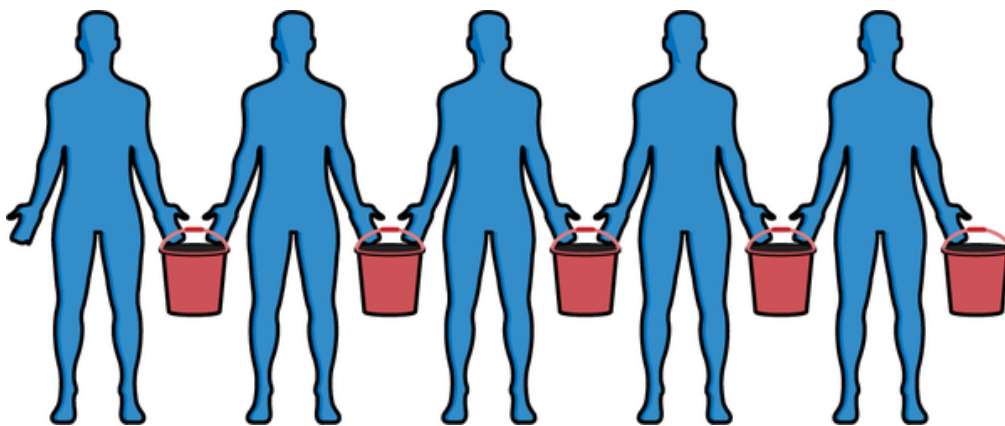


Figure 11-1. Bucket brigades pass buckets of data from one to another in turn

Bucket brigades were often used to transport water from a river, stream, lake, or well to the source of a fire. When it was impossible to use hoses to move the water, people would line up and pass buckets from one to another in order to fight the fire. One person would fill a bucket at the water source and then pass the bucket to the next person in line. The people in line didn't move, but the bucket of water was transported from person to person in turn until the final person could throw the water on the fire. This process would continue until either the fire was extinguished or the source ran out of water.

Though you're not using PHP to fight a fire, the internal structure of a stream is somewhat similar to a bucket brigade because of the way data is passed one chunk (bucket) at a time through whatever component of code is processing it.

Generators are also analogous to this pattern.<sup>1</sup> Rather than loading an entire collection of data into memory all at once, generators provide a way to reduce it into smaller chunks and operate on one piece of data at a time. This enables a PHP application to operate on data that would otherwise exhaust system

memory. Streams empower similar functionality, except working on continuous data rather than collections or arrays of discrete data points.

## Wrappers and Protocols

In PHP, streams are implemented using *wrappers* that are registered with the system to operate on a specific protocol. The most common wrappers you might interact with are those for file access or HTTP URLs, registered as `file://` and `http://`, respectively. Each wrapper operates against different kinds of data, but they all support the same basic functionality. [Table 11-1](#) enumerates the wrappers and protocols that are exposed natively by PHP.

Table 11-1. Native stream wrappers and protocols

Protocol	Description
<code>file://</code>	Access the local filesystem
<code>http://</code>	Access remote URLs over HTTP(S)
<code>ftp://</code>	Access remote filesystems over FTP(S)
<code>php://</code>	Access various local I/O streams (memory, <code>stdin</code> , <code>stdout</code> , etc.)
<code>zlib://</code>	Compression
<code>data://</code>	Raw data (according to <a href="#">RFC 2397</a> )
<code>glob://</code>	Find pathnames matching a pattern
<code>phar://</code>	Manipulate PHP archives
<code>ssh2://</code>	Connect via secure shell
<code>rar://</code>	RAR compression
<code>ogg://</code>	Audio streams

Each wrapper produces a `stream` resource that enables you to read or write data in a linear fashion with the additional ability to “seek” to an arbitrary location within the stream. A `file://` stream, for example, allows arbitrary access to bytes on disk. Similarly, the `php://` protocol provides read/write access to various streams of bytes held in the local system memory.

## Filters

PHP’s stream filters provide a construct that allows for the dynamic manipulation of the bytes in a stream during either read or write. A simple example would be to automatically convert every character in a string to either uppercase or lowercase. This is accomplished by creating a custom class that extends the `php_user_filter` class and registering that class as a filter for the compiler to use, as in [Example 11-1](#).

### Example 11-1. User-defined filter

```
class StringFilter extends php_user_filter
{
    private string $mode;

    public function filter($in, $out, &$consumed, bool
    {
        while ($bucket = stream_bucket_make_writeable($
            ❶

            switch($this->mode) {
                case 'lower':
                    $bucket->data = strtolower($bucket-
                    break;
                case 'upper':
                    $bucket->data = strtoupper($bucket-
                    break;
            }

            $consumed += $bucket->datalen;
            ❷

            stream_bucket_append($out, $bucket);
            ❸

        }

        return PSFS_PASS_ON;
        ❹
    }
}
```

```

    }

    public function onCreate(): bool
    {
        switch($this->filtername) {
            5

            case 'str.tolower':
                $this->mode = 'lower';
                return true;
            case 'str.toupper':
                $this->mode = 'upper';
                return true;
            default:
                return false;
        }
    }
}

stream_filter_register('str.*', 'StringFilter');
6

$fp = fopen('document.txt', 'w');
stream_filter_append($fp, 'str.toupper');
7

fwrite($fp, 'Hello' . PHP_EOL);
8

fwrite($fp, 'World' . PHP_EOL);

fclose($fp);

echo file_get_contents('document.txt');
9

```

The `$in` resource passed into the filter must first be made writable before you can do anything with it. 1

When consuming data, always be sure to update the `$consumed` output variable so PHP can keep track of how many bytes you've operated on. 2

The `$out` resource is initially empty, and you need to write buckets to it in order for other filters (or just PHP itself) to continue acting on 3

the s  
The `PSFS_PASS_ON` flag tells PHP that the filter was successful and data is available in the resource defined by `$out`.

This particular filter can act on any `str.` flag but intentionally only reads two filter names for converting text to uppercase or lowercase. By switching on the defined filter name, you can intercept and filter *just* the operations you want, while allowing other filters to define their own `str.` functions.

Defining the filter is not enough; you must explicitly register the filter so PHP knows which class to instantiate when filtering a stream.

Once the filter is defined and registered, you must either append (or prepend) the custom filter to the list of filters attached to the current stream resource.

With the filter attached, any data written to the stream will pass through the filter.

Opening the file again demonstrates that your input data was indeed converted to uppercase. Note that `file_get_contents()` reads the entire file into memory rather than operating on it as a stream.

Internally, any custom filter's `filter()` method must return one of three flags:

#### `PSFS_PASS_ON`

Demonstrates that processing completed successfully and that the output bucket brigade ( `$out` ) contains data ready for the next filter

#### `PSFS_FEED_ME`

Demonstrates that the filter completed successfully, but no data was available to the output brigade. You must provide more data to the filter (either from the base stream or the filter immediately prior in the stack) to get any output

#### `PSFS_ERR_FATAL`

Indicates that the filter experienced an error

The `onCreate()` method exposes three internal variables from the underlying `php_user_filter` class as if they were properties of the child class itself:

`::filtername`

The name of the filter as specified in `stream_filter_append()`  
or `stream_filter_prepend()`

`::params`

Additional parameters passed into the filter either when appending or prepending it to the filter stack

`::stream`

The actual stream resource being filtered

Stream filters are powerful ways to manipulate data as it flows into or out of the system. The following recipes cover various uses of streams in PHP, including both stream wrappers and filters.

## 11.1 Streaming Data to/from a Temporary File

### Problem

You want to use a temporary file to store data used elsewhere in a program.

### Solution

To store data, use the `php://temp` stream as if it were a file as follows:

```
$fp = fopen('php://temp', 'rw');

while (true) {
    // Get data from some source

    fputs($fp, $data);

    if ($endOfData) {
        break;
    }
}
```

To retrieve that data again, rewind the stream to the beginning and then read the data back out as follows:

```

rewind($fp);

while (true) {
    $data = fgets($fp);

    if ($data === false) {
        break;
    }

    echo $data;
}

fclose($fp);

```

## Discussion

In general, PHP supports two different temporary data streams. The Solution example leverages the `php://temp` stream but could have just as easily used `php://memory` to achieve the same effect. For streams of data that fit entirely in memory, the two wrappers are interchangeable. Both will, by default, use system memory to store stream data. Once the stream surpasses the amount of memory available to the application, however, `php://temp` will instead route the data to a temporary file on disk.

In both cases, the data written to the stream is assumed to be ephemeral. Once you close the stream, this data is no longer available. Likewise, you cannot create a *new* stream resource that points to the same data. [Example 11-2](#) illustrates how PHP will leverage *different* temporary files for streams even when using the same stream wrapper.

### Example 11-2. Temporary streams are unique

```

$fp = fopen('php://temp', 'rw');

fputs($fp, 'Hello world!');
❶

rewind($fp);
❷

echo fgets($fp) . PHP_EOL;
❸

```

```
$fp2 = fopen('php://temp', 'rw');
```

4

```
fputs($fp2, 'Goodnight moon.');
```

5

```
rewind($fp);
```

6

```
rewind($fp2);
```

```
echo fgets($fp2) . PHP_EOL;
```

7

```
echo fgets($fp) . PHP_EOL;
```

8

Write a single line to the temporary stream.

1

Rewind the stream handle so you can reread data from it.

2

Reading data back out of the stream prints `Hello world!` to the console.

3

Creating a new stream handle creates an entirely new stream despite the identical protocol wrapper.

4

Write some unique data to this new stream.

5

For good measure, rewind both streams.

6

Print the second stream first to prove it's unique. This prints

7

`Goodnight moon.` to the console.

Reprint `Hello world!` to the console to prove that the original stream still works as expected.

8

In either case, a temporary stream is useful when you need to store some data while running an application and don't explicitly want to persist it to disk.

## See Also

Documentation on [fopen\(.\)](#) and [PHP I/O stream wrappers](#).

# 11.2 Reading from the PHP Input Stream

## Problem

You want to read raw input from within PHP.

## Solution

Leverage the `php://stdin` stream to read the [standard input stream](#) (`stdin`) as follows:

```
$stdin = fopen('php://stdin', 'r');
```

## Discussion

Like any other application, PHP has direct access to the input passed to it by commands and other upstream applications. In a console world, this might be another command, literal input in the terminal, or data piped in from another application. In a web context, though, you would instead use `php://input` to access the literal contents submitted in a web request and passed through whatever web server is in front of the PHP application.

---

### NOTE

In command-line applications, you can also use the predefined [STDIN constant](#) directly. PHP natively opens a stream for you, meaning you don't need to create a new resource variable at all.

---

A simple command-line application might take data from the input, manipulate that data, and then store it in a file. In [Recipe 9.5](#), you learned how to encrypt and decrypt files by using symmetric keys with Libsodium. Assuming you have an encryption key (encoded in hexadecimal) exposed as an environment variable, the program in [Example 11-3](#) would use that key to encrypt any data passed in and store it in an output file.

### Example 11-3. Encrypting stdin with Libsodium

```
if (empty($key = getenv('ENCRYPTION_KEY'))) {  
    ❶  
    throw new Exception('No encryption key provided!');  
}  
  
$key = hex2bin($key);  
if (strlen($key) !== SODIUM_CRYPTO_STREAM_XCHACHA20_KEY  
    ❷  
    throw new Exception('Invalid encryption key provide  
}  
  
$in = fopen('php://stdin', 'r');  
    ❸  
  
$filename = sprintf('encrypted-%s.bin', uniqid());  
    ❹  
  
$out = fopen($filename, 'w');  
    ❺  
  
[$state, $header] = sodium_crypto_secretstream_xchacha2  
    ❻  
  
fwrite($out, $header);  
  
while (!feof($in)) {  
    $text = fread($in, 8175);  
  
    if (strlen($text) > 0) {  
        $cipher = sodium_crypto_secretstream_xchacha20p  
  
        fwrite($out, $cipher);  
    }  
}  
  
sodium_memzero($state);  
  
fclose($in);  
fclose($out);  
  
echo sprintf('Wrote %s' . PHP_EOL, $filename);
```

Since you want to use an environment variable to house the encryption key, first check that this variable exists.

Also do a sanity check that the key is of the right size before using it for encryption.

In this example, read the bytes directly from `stdin`.

Use a dynamically named file to store the encrypted data. Note that, in practice, `uniqid()` uses timestamps and could be subject to race conditions and name collisions on highly used systems. In a real-world environment, you will want to use a more reliable source of randomness for a generated filename.

The output could be passed back to the console but, since this encryption produces raw bytes, it's safer to stream the output to a file. In this case, the filename will be generated dynamically based on the system clock.

The rest of the encryption follows the same pattern as [Recipe 9.5](#).

The preceding example enables you to pipe data from a file directly into PHP by using the standard input buffer. Such a piping operation might look something like `cat plaintext-file.txt | php encrypt.php`.

Given that the encryption operation will produce a file, you can reverse the operation with a similar script and similarly leverage `cat` to pipe the raw binary back into PHP, as shown in [Example 11-4](#).

#### Example 11-4. Decrypting `stdin` with Libsodium

```
if (empty($key = getenv('ENCRYPTION_KEY'))) {  
    throw new Exception('No encryption key provided!');  
}  
  
$key = hex2bin($key);  
if (strlen($key) !== SODIUM_CRYPTOSTREAM_XCHACHA20_KEY  
    throw new Exception('Invalid encryption key provided!');  
}  
  
$in = fopen('php://stdin', 'r');  
$filename = sprintf('decrypted-%s.txt', uniqid());  
$out = fopen($filename, 'w');  
  
$header = fread($in, SODIUM_CRYPTOSTREAM_XCHACHA20_HEADER_SIZE);
```

```

$state = sodium_crypto_secretstream_xchacha20poly1305_i

try {
    while (!feof($in)) {
        $cipher = fread($in, 8192);

        [$plain, ] = sodium_crypto_secretstream_xchacha20poly1305_i
            $state,
            $cipher
        );

        if ($plain === false) {
            throw new Exception('Error decrypting file!');
        }

        fwrite($out, $plain);
    }
} finally {
    sodium_memzero($state);

    fclose($in);
    fclose($out);

    echo sprintf('Wrote %s' . PHP_EOL, $filename);
}

```

Thanks to PHP's I/O stream wrappers, arbitrary input streams are just as easy to manipulate as native files on a local disk.

## See Also

Documentation on [PHP I/O stream wrappers](#).

## 11.3 Writing to the PHP Output Stream

### Problem




 You want to output data directly.

## Solution

Write to `php://output` to push data directly to the [standard output](#) (`stdout`) stream as follows:

```
$stdout = fopen('php://stdout', 'w');  
fputs($stdout, 'Hello, world!');
```

## Discussion

PHP exposes three standard I/O streams to userland code— `stdin` , `stdout` , and `stderr` . By default, anything you print in your application is sent to the standard output stream ( `stdout` ), which makes the following two lines of code functionally equivalent:

```
fputs($stdout, 'Hello, world!');  
echo 'Hello, world!';
```

Many developers learn to use `echo` and `print` statements as simple ways to debug an application; adding an indicator in your code makes it easy to identify where exactly the compiler is failing or to emit the value of an otherwise hidden variable. However, this isn't the *only* way to manage output. The `stdout` stream is common to many applications and writing to it directly (versus an implicit `print` statement) is a way to keep your application focused on what it needs to be doing.

Similarly, once you start leveraging `php://stdout` directly to print output to the client, you can start leveraging the `php://stderr` stream to emit messages about *errors* as well. These two streams are treated differently by the operating system, and you can use them to segment your messaging between useful messages and error states.

---

### NOTE

In command-line applications, you can also use the predefined [STDOUT and STDERR constants](#) directly. PHP natively opens these streams for you, meaning you don't need to create new resource variables at all.

---

[Example 11-4](#) allowed you to read encrypted data from `php://stdin`, decrypt it, and then store the decrypted content in a file. A more useful example would instead present that decrypted data to `php://stdout` (and any errors to `php://stderr`), as shown in [Example 11-5](#).

#### Example 11-5. Decrypting stdin to stdout

```
if (empty($key = getenv('ENCRYPTION_KEY'))) {
    throw new Exception('No encryption key provided!');
}

$key = hex2bin($key);
if (strlen($key) !== SODIUM_CRYPTOSTREAM_XCHACHA20_KEYBYTES)
    throw new Exception('Invalid encryption key provided!');

$in = fopen('php://stdin', 'r');
$out = fopen('php://stdout', 'w');
    ❶

$err = fopen('php://stderr', 'w');
    ❷

$header = fread($in, SODIUM_CRYPTOSTREAM_XCHACHA20_HEADERBYTES);
$state = sodium_crypto_secretstream_xchacha20poly1305_init(
    $header, $key);

while (!feof($in)) {
    $cipher = fread($in, 8192);

    [$plain, $state] = sodium_crypto_secretstream_xchacha20poly1305_decrypt(
        $cipher, $state, $key);

    if ($plain === false) {
        fwrite($err, 'Error decrypting file!');
        ❸

        exit(1);
    }

    fwrite($out, $plain);
}

sodium_memzero($state);
```

```
fclose($in);  
fclose($out);  
fclose($err);
```

Rather than creating an intermediary file, you can write directly to the standard output stream. <sup>1</sup>

You should also get a handle on the standard error stream while you're at it. <sup>2</sup>

Rather than triggering an exception, you can write directly to the error stream. <sup>3</sup>

## See Also

Documentation on [PHP I/O stream wrappers](#).

# 11.4 Reading from One Stream and Writing to Another

## Problem


You want to connect two streams, passing the bytes from one to the other.

## Solution

Use `stream_copy_to_stream()` to copy data from one stream to another as follows:

```
$source = fopen('document1.txt', 'r');  
$dest = fopen('destination.txt', 'w');  
  
stream_copy_to_stream($source, $destination);
```

## Discussion

◀  ▶

The streaming mechanisms in PHP provide for very efficient ways to work with rather large chunks of data. Often, you might end up using files within

your PHP application that are too large to fit in the application's available memory. Most files you might make public and send to the user directly via Apache or NGINX. In other cases, for example, you might want to safeguard large file downloads (like zip files or videos) with scripts written in PHP to validate a user's identity.

Such a scenario is possible in PHP because the system doesn't need to keep the entire stream in memory but can instead write bytes to one stream as they are read from another stream. [Example 11-6](#) assumes that your PHP application directly authenticates a user and validates that they have the right to a particular file before streaming its contents.

#### Example 11-6. Copy large file to stdout by linking streams

```
if ($user->isAuthenticated()) {  
    $in = fopen('largeZipFile.zip', 'r');  
    ❶  
  
    $out = fopen('php://stdout', 'w');  
  
    stream_copy_to_stream($in, $out);  
    ❷  
  
    exit;  
    ❸  
}
```

The act of opening a stream merely gets a handle on the underlying data. No bytes have yet been read by the system. ❶

Copying a stream to another stream will copy the bytes directly without keeping the entire contents of either stream in memory. ❷

Remember, streams work on chunks similar to a bucket brigade, so only a subset of the necessary bytes are held in memory at any given time.

It's important to always `exit` after copying a stream; otherwise, you might inadvertently append miscellaneous bytes by mistake. ❸

Similarly, it is possible to programmatically *build* a large stream and copy it to another stream when needed. Some web applications might need to programmatically build large chunks of data (very large single-page web applications, for example). It is possible to write these large data elements to

PHP's temporary memory stream and then copy the bytes back out when needed. [Example 11-7](#) illustrates exactly how that would work.

### Example 11-7. Copying a temporary stream to stdout

```
$buffer = fopen('php://temp', 'w+');  
                                     ❶  
  
fwrite($buffer, '<html><head>');  
  
// ... Several hundred fwrite()s later ...  
  
fwrite($buffer, '</body></html>');  
rewind($buffer);  
                                     ❷  
  
$output = fopen('php://stdout', 'w');  
stream_copy_to_stream($buffer, $output);  
                                     ❸  
  
exit;  
                                     ❹
```

A temporary stream leverages a temporary file on disk. You are limited not by the memory available to PHP but by the available space allocated to temporary files by the operating system.

After writing the entire HTML document to a temporary file, rewind the stream back to the beginning in order to copy all of those bytes to stdout.

The mechanism to copy one stream to another remains unchanged even though neither of these streams point to a specifically identified file on disk.

Always `exit` after all bytes have been copied to the client to avoid accidental errors.

## See Also

Documentation on [stream\\_copy\\_to\\_stream\(\)](#).

# 11.5 Composing Different Stream Handlers Together

## Problem

You want to combine multiple stream concepts—e.g., a wrapper and a filter—in one piece of code.

## Solution

Append filters as necessary and use the appropriate wrapper protocol.

[Example 11-8](#) uses the `file://` protocol for local filesystem access and two additional filters to handle Base64-encoding and file decompression.

### Example 11-8. Applying multiple filters to a stream

```
$fp = fopen('compressed.txt', 'r');  
stream_filter_append($fp, 'convert.base64-decode');  
stream_filter_append($fp, 'zlib.inflate');  
  
echo fread($fp, 1024) . PHP_EOL;
```

Assume this file exists on disk and contains the literal contents  
`80jNycnXUSjPL8pJUQQA`.

The first stream filter added to the stack will convert from Base64-encoded ASCII text to raw bytes.

The second filter will leverage Zlib compression to inflate (or uncompress) the raw bytes.

If you started with the literal contents in step 1, this will likely print  
`Hello, world!` to the console.

## Discussion

When it comes to streams, it's helpful to think about layers. The foundation is always the protocol handler used to instantiate the stream. There is no explicit protocol in the Solution example, which means PHP will leverage the `file://` protocol by default. Atop that foundation is any number of layers of filters on the stream.

The Solution example leverages both Zlib compression and Base64 encoding to compress text and encode the raw (compressed) bytes, respectively. To create such a compressed/encoded file, you would do the following:

```
$fp = fopen('compressed.txt', 'w');

stream_filter_append($fp, 'zlib.deflate');
stream_filter_append($fp, 'convert.base64-encode');

fwrite($fp, 'Goodnight, moon!');
```

The preceding example leverages the same protocol wrapper and filters as the Solution example. But note that the *order* in which they are added is reversed. This is because stream filters work like the layers on a jawbreaker, similar to the illustration in [Figure 11-2](#). The protocol wrapper is at the core, and data flows from that core to the outside world, through each subsequent layer in a specific order.

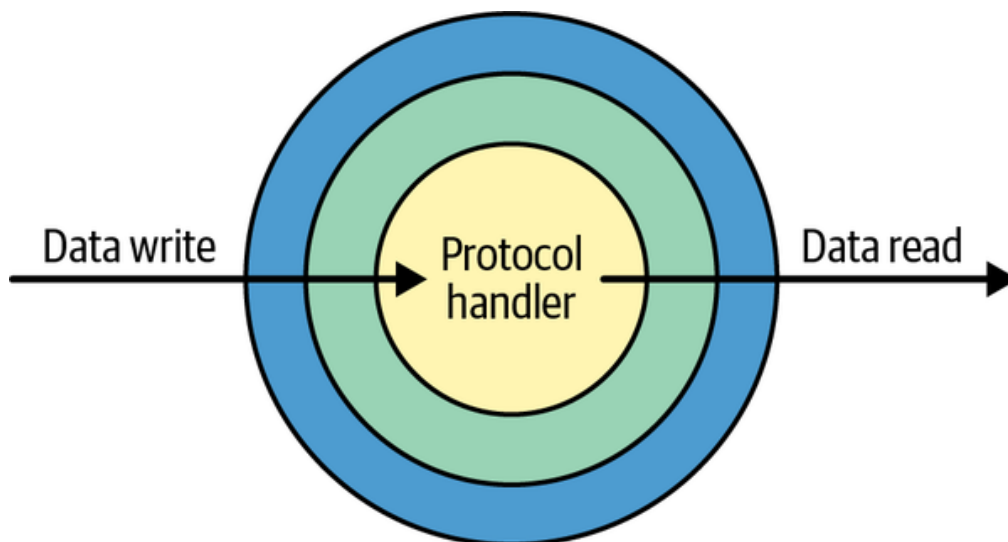


Figure 11-2. Data flowing into and out of PHP stream filters

There are several filters that you can apply to a stream already built into PHP. However, you can also define your *own* filter. It's useful to encode raw bytes

in Base64, but it's also sometimes useful to encode/decode bytes as hexadecimal. Such a filter doesn't exist natively in PHP, but you can define it yourself by extending the `php_user_filter` class similarly to the way [Example 11-1](#) did in this chapter's introduction. Consider the class in [Example 11-9](#).

#### Example 11-9. Encoding/decoding hexadecimal with a filter

```
class HexFilter extends php_user_filter
{
    private string $mode;

    public function filter($in, $out, &$consumed, bool
    {
        while ($bucket = stream_bucket_make_writeable($
            switch ($this->mode) {
                case 'encode':
                    $bucket->data = bin2hex($bucket->da
                    break;
                case 'decode':
                    $bucket->data = hex2bin($bucket->da
                    break;
                default:
                    throw new Exception('Invalid encodi
            }

            $consumed += $bucket->datalen;
            stream_bucket_append($out, $bucket);
        }

        return PSFS_PASS_ON;
    }

    public function onCreate(): bool
    {
        switch($this->filtername) {
            case 'hex.decode':
                $this->mode = 'decode';
                return true;
            case 'hex.encode':
                $this->mode = 'encode';
                return true;
            default:
                return false;
        }
    }
}
```

```
}
}
```

The class defined in [Example 11-9](#) can be used to arbitrarily encode to and decode from hexadecimal when applied as a filter to any arbitrary stream. Merely register it as you would any other filter, then apply it to whatever streams need to be converted.

The Base64 encoding used in the Solution example could be substituted with hexadecimal entirely, as shown in [Example 11-10](#).

#### Example 11-10. Combining a hexadecimal stream filter with Zlib compression

```
stream_filter_register('hex.*', 'HexFilter');
❶

// Writing data
$fp = fopen('compressed.txt', 'w');

stream_filter_append($fp, 'zlib.deflate');
stream_filter_append($fp, 'hex.encode');

fwrite($fp, 'Hello, world!' . PHP_EOL);
fwrite($fp, 'Goodnight, moon!');

fclose($fp);
❷

$fp2 = fopen('compressed.txt', 'r');
stream_filter_append($fp2, 'hex.decode');
stream_filter_append($fp2, 'zlib.inflate');

echo fread($fp2, 1024);
❸
```

Once the filter exists, it must be registered so PHP knows how to use it. Leveraging a `*` wildcard during registration allows for both encoding and decoding to be registered at once.

The contents of *compressed.txt* will be  
 f348cdc9c9d75128cf2fca4951e472cfcf  
 4fc9cb4ccf28d151c8cdcfcf530400 at this point.

After decoding and decompressing Hello world! Goodnight

After decoding and decompressing, the `HELLO WORLD! Goodnight,`  
moon! will be printed to the console (with a newline between the two  
statements).

## See Also

Supported [protocols and wrappers](#) and the [list of available filters](#). Also,  
[Example 11-1](#) for a user-defined stream filter.

# 11.6 Writing a Custom Stream Wrapper

## Problem

You want to define your own custom stream protocol.

## Solution

Create a custom class that follows the prototype of `streamWrapper` and register it with PHP. For example, a `VariableStream` class could provide a stream-like interface to read from or write to a specific global variable, as follows:<sup>2</sup>

```
class VariableStream
{
    private int $position;
    private string $name;
    public $context;

    function stream_open($path, $mode, $options, &$oper
    {
        $url = parse_url($path);
        $this->name = $url['host'];
        $this->position = 0;

        return true;
    }

    function stream_write($data)
    {
        $left = substr($GLOBALS[$this->name], 0, $this-
        $right = substr($GLOBALS[$this->name], $this->f
```

```

        $GLOBALS[$this->name] = $left . $data . $right;
        $this->position += strlen($data);
        return strlen($data);
    }
}

```

The preceding class would be registered and used in PHP as follows:

```

if (!in_array('var', stream_get_wrappers())) {
    stream_wrapper_register('var', 'VariableStream');
}

$varContainer = '';

$fp = fopen('var://varContainer', 'w');

fwrite($fp, 'Hello' . PHP_EOL);
fwrite($fp, 'World' . PHP_EOL);
fclose($fp);

echo $varContainer;

```

## Discussion

The `streamWrapper` construct in PHP is a prototype for a class. Unfortunately, it is not a class that can be extended, nor is it an interface that can be concretely implemented. Instead, it is a documented format that any user-defined stream protocols must follow.

While it is possible to register classes as protocol handlers following a different interface, it is strongly advised that any potential protocol classes implement all methods defined by the `streamWrapper` interface (copied from the PHP document as a pseudo-interface definition in [Example 11-11](#)) in order to satisfy PHP's expected stream behavior

### Example 11-11. streamWrapper interface definition

```

class streamWrapper {
    public $context;

    public __construct()

```

```

public dir_closedir(): bool

public dir_opendir(string $path, int $options): bool

public dir_readdir(): string

public dir_rewinddir(): bool

public mkdir(string $path, int $mode, int $options)

public rename(string $path_from, string $path_to):

public rmdir(string $path, int $options): bool

public stream_cast(int $cast_as): resource

public stream_close(): void

public stream_eof(): bool

public stream_flush(): bool

public stream_lock(int $operation): bool

public stream_metadata(string $path, int $option, n

public stream_open(
    string $path,
    string $mode,
    int $options,
    ?string &$opened_path
): bool

public stream_read(int $count): string|false

public stream_seek(int $offset, int $whence = SEEK_

public stream_set_option(int $option, int $arg1, ir

public stream_stat(): array|false

public stream_tell(): int

public stream_truncate(int $new_size): bool

```

```

        public stream_write(string $data): int

        public unlink(string $path): bool

        public url_stat(string $path, int $flags): array|false

        public __destruct()
    }

```

Some specific functionality—e.g., `mkdir`, `rename`, `rmdir`, or `unlink`—should *not* be implemented at all unless the protocol has a specific use for it. Otherwise, the system will not provide helpful error messages to you (or developers building atop your library) and will behave unexpectedly.

While most protocols you will use day-to-day ship natively with PHP, it's possible to write new protocol handlers or leverage those built by other developers.

It's common to see references to cloud storage that use a proprietary protocol (e.g., Amazon Web Services' `s3://`) rather than the more common `https://` or `file://` prefixes seen elsewhere. AWS actually publishes a [public SDK](#) that uses `stream_wrapper_register()` internally to provide an `s3://` protocol to other application code, empowering you to work with cloud-hosted data as easily as you would local files.

## See Also

Documentation on [streamWrapper](#).

<sup>1</sup> For more on generators, review [Recipe 7.15](#).

<sup>2</sup> The PHP Manual provides [a similarly named class](#) with much broader and more complete functionality than is demonstrated in this Solution example.