www.phparch.com

February 2017
VOLUME 16 - Issue 2

php[architect]

ESCAPE THE SERVER

Sample Article!

ALSO INSIDE

PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

May 24-26 — ATLANTA

Keynote Speakers:

Gemma Anible
WonderProxy

Keith Adams
Slack

Alena Holligan
Treehouse

Larry Garfield
Platform.sh

Mitch Trale
PucaTrade

Samantha Quiñones
Etsy

Sponsored By:

php women

ShootProof

Open Sourcing Mental Illness

platform.sh

Save $200 on tickets
Buy ***before*** Feb 18th

tek.phparch.com

php[architect]

CONTENTS

FEBRUARY 2017
Volume 16 - Issue 2

ESCAPE THE SERVER

Features

Columns

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Mocking the File System with VfsStream

Gabriel Zerbib

Working with files is a common task in many PHP applications, whether for input data to be processed, generated results, or even as a logging tap. But the file system is a persistent resource, and creating unit tests that involve files requires some special care. In this article, you'll learn how to test against a mock file system the pieces of your application which use file operations, instead of involving your actual drive.

In an ideal world, every line of code exists for one very well known reason and is written with a clear purpose.

As developers, we quickly learned the safest way to ensure a piece of code does what it is expected to, is to have it pass through a unit test.

However, testing a program's interactions with files can be tricky, similarly to testing against a database resource. One needs to populate the test environment in a way that can be reproduced, with full control over both the data and metadata. Your file manipulations must be made in a sandbox (you don't want to modify critical files accidentally), and their outcome has to be measurable. You also need to perform clean-up tasks thoroughly after your test scenario has finished.

Fortunately, there are tools that can help, and we will discuss the **vfsStream** library in this article.

Base Principle

vfsStream utilizes a *stream wrapper* to expose a virtual file system to your program. Your unit tests component can manipulate files, folders, and permissions as if they were physically stored entities, whereas they are actually only in memory, much like in a RAM disk.

Using a virtual file system makes it easy for your unit tests to represent a machine-independent folder structure: no need to worry about the physical location of the project on the various developers' workstations. It also enables you to provision borderline case scenarios (for example, to simulate denied permission situations on critical files, when they would actually be fine on the developer's physical machine).

Stream Wrappers

PHP uses the concept of *stream wrappers*[1] to handle various protocols (either built-in or user-defined) as stream resources. For example, you can usually transparently read a file by its URL whether it is found in the local file system, or served by a remote web server (URL begins with `http://`) or by an FTP server (URL in `ftp://`). The scheme component of the address (`http`, `tcp`, and so on) tells PHP how to negotiate the access to the resource.

The default wrapper, when no scheme is specified, is the local file system.

You are probably already familiar with some non-trivial stream wrappers, in addition to our usual `http` friend: in particular the `php` scheme, as in `fopen('php://stderr', 'w')` (which writes to the Linux standard error stream of the PHP process). You may have worked with the `phar://` wrapper as well, to include a script inside a phar archive. And some of you were already aware the content of "folder/file.txt" inside "archive.zip" can be obtained by:

```
file_get_contents(
    'zip://archive.zip#folder/file.txt'
);
```

But we are not limited to the built-in schemes. PHP lets us register our own stream wrappers, designed to handle our custom schemes, through the function:

```
stream_wrapper_register(
    $scheme, $wrapperClass
)
```

The wrapper class you provide must implement the `StreamWrapper` interface[2], which defines all the low-level operations PHP carries out when it tries to reach a resource by your custom scheme. For example, the wrapper should supply a handler for the `stream_eof()` operation, which PHP consults to check if it has reached the end of the accessed stream. (For a terminal-like stream such as `php://stderr`, the end is never reached. It is really up to the wrapper to decide how to handle the stream.

> *Note: There is no actual* `streamWrapper` *base class or interface. The term only exists in the PHP manual: there is nothing to derive from, but your class must comply with the signatures indicated in the documentation.*

vsfStream[3] acts as a stream wrapper (namely `vfs://`), and implements some sort of in-memory file system, complete with (Linux-like) file permissions and other specifics that an application would expect from an actual persistent storage.

Getting Started

Classically, the installation is best achieved by adding a development dependency to your Composer project:

1 *stream wrappers:* http://php.net/en/wrappers

2 `StreamWrapper` *interface:* http://php.net/class.streamwrapper

3 *vsfStream:* https://github.com/mikey179/vfsStream

```
composer require --dev mikey179/vfsStream
```

> *A development dependency is a library which your application does not need at run-time, but rather, that you will use when developing or debugging. In our case, the mock file system library is only useful when writing and executing our unit tests, which means the dependency should not be deployed to production. In the* `composer.json` *file, the development dependencies are declared in the* `"require-dev"` *section.*

Before working with virtual files, you need to set up the virtual file system, by indicating its mounting root. vfsStream defaults to "root/," but I recommend simply `/` instead because it is closer to the common understanding of the top component of a hierarchical file-system-like structure. But, as we will see now, the root is only a logical naming, and it does not make any real difference in the way you write and use tests with vfsStream.

```
use org\bovigo\vfs\vfsStream;

$vfs = vfsStream::setup('/');
```

Then you are ready to populate your file system and to assert its layout and contents with the regular file-related functions, where all the file names are specified using the `vfs://` protocol scheme:

```
// Obtain a scheme-prefixed URL for the following virtual
// absolute path:
$url = vfsStream::url('/test.txt');
// $url is now: 'vfs:///test.txt'

// Use PHP functions to manipulate the file
file_put_contents($url, "Contents of the new file . \n");
```

Or, you may prefer the *fluent* methods provided by the vfsStream library to walk through the directory structure and create the nodes.

```
vfsStream::newFile('test.txt')
    ->at($vfs)->setContent("Contents of the new file.\n");
```

Both flavors create the file `'test.txt'` at the root of your virtual file system. Any piece of your code under test can reach this file by means of the usual PHP functions like `rename`, `unlink`, `fopen` (or most of them—see *Limitations* below). The only requirement is for you to specify the full address: `'vfs:///test.txt'`.

Recipes

We will now see how to use this tool in various common testing scenarios.

Directory Structure

When warming up the test environment, we often need to prepare a pre-existing structure of files and folders, because this is what the system under test expects to see.

This is done by the `create` helper.

```
vfsStream::create([
    'var' => [
        'log' => ['custom.log' => 'Some initial contents']
    ],
    'tmp' => []
]);
```

This code will create the `var` and `tmp` folders under the root of your VFS, and a plain file `custom.log` with some text contents under `var/log`. The argument to the `create` helper is a tree-shaped associative array, whose keys are node names (either folders or files). If the value for the key is another array, then the node is a folder. If the value is a string, then the node is a file whose contents is this string.

Of course, the directory tree thus created is not immutable, and you can always act on it through the PHP file system functions as in Listing 1 (which is the whole point of the library).

Listing 1

```
1.  // Rename virtual 'var' folder
2.  rename('vfs:///var', 'vfs:///srv');
3.  // Equivalent to:
4.
// rename(vfsStream::url('/var'), vfsStream::url('/srv'));
5.
6.  // Get a PHP iterator on the entries of virtual /tmp
7.  $it = new DirectoryIterator(vfsStream::url('/tmp'));
8.
9.  // Print the contents of this virtual file
10. readfile(vfsStream::url('/srv/log/custom.log'));
```

Permissions

The permission system in vfsStream does not pretend to implement a real life environment. Rather, it provides with tools to declare a particular node as accessible or forbidden to the current process (keep in mind that the library is intended for testing).

vfsStream defines a number of mock users and groups for your various scenarios. When creating a virtual file, the owner and group are those of the real process that runs your PHP script (see also `umask` for more details). But you can act on file permissions and ownership with `chmod`, `chown`, and `chgrp` to prove your code.

Therefore, these are the kind of checks you might want to do, see Listing 2.

Listing 2

```
1.  $url = 'vfs:///var/log/custom.log';
2.  echo intval( is_writable($url) ); // prints 1
3.
4.  chmod($url, 0400);
5.  echo intval( is_writable($url) ); // prints 0
6.
7.  chown($url, vfsStream::OWNER_USER_1);
8.  echo intval( is_readable($url) ); // prints 0
9.
10. readfile( $url );                 // Issues a warning
```

Running Out of Disk Space

The case when your host system runs out of storage space is often ignored by developers because it seems unlikely enough, and mostly because we lack a test tool to help in this task. An attacker can try a denial-of-service attack by overwhelming a web application with uploaded files, and exploit the resulting faulty behavior. Preparing for this situation should not be left out anymore.

```
//Declare that the virtual storage has only 10 bytes left.
vfsStream::setQuota(10);

file_put_contents('vfs:///tmp/test.txt', 'abcdefghijkl');
// Raises: PHP Warning:  file_put_contents(): Only 0 of 12 bytes
//   written, possibly out of free disk space
```

Handling Large Files

In a similar mindset, loop performance and memory usage when parsing files can be tested by simulating the existence of large files of arbitrary size.

Suppose the class under test opens an input file for reading and an output file for writing, and transforms the data in between, by small chunks.

Proving your logic on a large input file is easily done, using the file factory:

```
$largeFile = vfsStream::newFile('large.txt')
    ->withContent(LargeFileContent::withMegabytes(100))
    ->at($vfs->getChild('tmp'));

echo filesize($largeFile->url()); // prints: 104857600
```

The library will smartly maintain a vacuum-based incomplete structure whose virtual bytes are seen as spaces (`0x20`) by the PHP functions. The actual memory is not filled up and you can still write custom data at arbitrary positions in the file (see Listing 3).

The vfsStream tool stores a fine representation of your simulated data: all those megabytes will still be seen as spaces by `fread` and `copy` etc., but your actual phrase is found there in its proper place.

The `readfile` instruction here, will dump a big lot of white spaces, with 'some real content' after the first 5,000 (take my word for it; Oscar won't let me print the console output here).

Listing 3

```
1. // open virtual file for modifying its content
2. $fp = fopen($largeFile->url(), 'r+');
3. // go to position 5K
4. fseek($fp, 5000);
5.
// and put some actual phrase there, among all the spaces.
6. fwrite($fp, 'Some real content');
7. fclose($fp);
8.
9. readfile($largeFile->url());
```

Listing 4

```
1.<?php
2.
3. class JpegMover
4. {
5.     public function searchAndMove($directory, $moveTo) {
6.
// Iterate over the children of specified directory
7.         $di = new FilesystemIterator($directory);
8.         foreach ($di as $pathname => $item) {
9.             // If it's a regular file, with extension 'jpg'
10.            // then move it to destination folder.
11.            if ($item->isFile() &&
12.                ($item->getExtension() == 'jpg')
13.            ) {
14.                rename($pathname, $moveTo);
15.            }
16.        }
17.    }
18. }
```

SPL with VfsStream

Browsing the file system to get a (potentially recursive) list of the items it contains below a specific point, is best done in PHP with the `FilesystemIterator` class (and `RecursiveDirectoryIterator`) from the Standard PHP Library (SPL). The vfsStream package is a perfect companion for testing an application using these classes.

Let's consider a program which searches a specific directory for JPG images (*.jpg file names) in order to move them elsewhere. The code is in Listing 4.

You could test this code using PHPUnit (see Listing 5), by checking that all the JPG files were moved (and only those).

The SPL file system classes, as well as the `rename` file operation function, are indifferent to whether the file system is physical or virtual. They just work as expected.

Build for Testing

You may have noticed, as you drove on a bridge, the metallic or concrete building blocks that compose the piers and girders often present large holes or grooves at intervals.

Construction engineering teaches us that some of these holes are meant to lighten the structure, by simply extracting matter in a way which does not affect the strength or function of the block. But most of them really, are designed solely for quality assurance and transportation purposes. It is easier to carry a heavy block of concrete, move it around, and position it accurately when you can pick it up with a backhoe by its holes and grooves.

The lesson to learn in programming is that it is not less important, when constructing our code for a functional purpose, to make it testable by design.

One good practice that is very dear to me is called Test-driven Development: as soon as a feature is specified, we write a set of corresponding unit tests first, before developing the actual feature. Of course, the tests fail in the beginning,

because the code to achieve it does not exist yet. Then, as the development proceeds, more and more test assertions become green.

Although this methodology is not directly related to our current topic, nor is it mandatory, I can only recommend it warmly in general, and in particular when it comes to working with files. Because, as seen earlier, not everything is possible to abstract the file-related testing. Your effective code must be aware of the possibility to pass through a unit test engine using a virtual file system (which means allowing for prefix-enabled file names).

In particular your code cannot use the magic constant `__DIR__` to locate a data file relative to the running script (because your running script will probably never reside inside the vfsStream virtual space). You should also avoid relying on absolute physical paths for your temp or cache location etc., under penalty of not being able to test your code thoroughly.

Whenever possible, tell your code where to find the "root" of the (potentially virtual) file system rather than making assumptions on its nature and location. Either use some sort of dependency injection container or another configuration mechanism.

Limitations

Although this tool has proven itself very useful to me in several projects, it is worth noting not all the file-related functions are compatible with scheme handlers. Some of them are only a thin encapsulation around their C counterpart, for which the PHP engine won't activate its layer of stream wrappers. There is a list of known issues[4] on the project's page, which may save you some precious time when using vfsStream.

For example, the following will not let you obtain a temporary filename in the virtual file system, even if you specify a URL in `vfs://` as the `$dir` parameter because the underlying, low-level C function is unaware of user-land handlers.

```
tempnam($dir, $prefix)
```

But there is a workaround. As mentioned above, you shall design your application to receive from "outside" the location of the temp folder (whether through configuration files, environment variables or dependency injection so as to provide different bootstrapping in production or unit tests) and you can generate a unique string by different methods (such as `uniqid` or `openssl_random_pseudo_bytes`).

Perhaps the most frustrating limitation in using vfsStream is that the `gz` family of PHP functions (`gzopen`, `gzwrite`, etc.) are low-level encapsulations of the zlib C library. Currently, the zlib library cannot be aware of your PHP-specific vfsStream memory structure. If your application needs to write *gz* data transparently to a file, while keeping low in CPU and not eating up the memory of your PHP script, the `gz` functions are here to help. But then you can't test using virtual files, and you will have to write tests using physical files. (Yes, you still need to write tests!). The workaround here, to be able to test with vfsStream, is to use the higher level *gz* functions (`gzencode`, `gzdecode` etc.) which work in the PHP domain, in memory (not directly on disk), and to read and write to disk after in-memory compression. However, this might not always be a suitable solution for your project.

Listing 5

```
<?php
class MoverTest extends PHPUnit_Framework_TestCase
{
    public function testAllTheFilesWereMoved() {
        // Prepare the virtual filesystem layout
        vfsStream::setup('/');

        // We create two jpg files and one txt file
        vfsStream::create([
            'images' => [
                'move1.jpg' => "",
                'move2.jpg' => "",
                'keep.txt' => "",
            ],
            'target' => []
        ]);

        $dir = vfsStream::url('/images');
        $dest = vfsStream::url('/target');

        $systemUnderTest = new JpegMover();
        $systemUnderTest->searchAndMove($dir, $dest);

        // Assertions
        $this->assertFalse(file_exists($dir . '/move1.jpg'));
        $this->assertFalse(file_exists($dir . '/move2.jpg'));
        $this->assertTrue(file_exists($dir . '/keep.txt'));
        $this->assertTrue(file_exists($dest . '/move1.jpg'));
        $this->assertTrue(file_exists($dest . '/move2.jpg'));
    }
}
```

Conclusion

Even though you may encounter specific situations where vfsStream can't help, it remains handy for most projects. Judging by the number of Packagist downloads and projects dependent on it the library is very stable and popular.

If you've been avoiding writing tests for file system related methods in your code, stop procrastinating! vfsStream is a capable tool for adding unit tests to verify file-system operations work as intended.

Gabriel Zerbib is a full-stack engineer and cloud architect, who enjoys programming in various languages since the 80s. Currently based in Tel Aviv, he specializes in high frequency applications and large scale data volumes. @zzgab

4 *known issues:* https://github.com/mikey179/vfsStream/wiki/Known-Issues

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe