Chapter 10. File Handling

One of the most common design philosophies around Unix and Linux is that "everything is a file." This means that, regardless of the resource with which you're interacting, the operating system treats it as if it were a file locally on disk. This includes remote requests to other systems and handles on the output of processes running on the machine.

PHP treats requests, proceses, and resources similarly, but instead of considering everything to be a file, the language considers everything to be a stream resource. Chapter 11 covers streams at length, but the important point to know about streams for this chapter is the way PHP treats them in memory.

When accessing a file, PHP doesn't necessarily read the file's entire data into memory. Instead, it creates a `resource` in memory that references the file's location on disk and selectively buffers bytes from that file in memory. PHP then accesses or manipulates those buffered bytes directly as a stream. The fundamentals of streams, however, are not required knowledge for the recipes in this chapter.

PHP's file methods— `fopen()`, `file_get_contents()`, and the like— all leverage the `file://` stream wrapper under the hood. Remember, though, if everything in PHP is a stream, you can just as easily use other stream protocols as well, including `php://` and `http://`.

Windows Versus Unix

PHP is distributed for use on both Windows and Unix-style operating systems (including Linux and macOS). It's important to understand that the underlying filesystem behind Windows is very different from a Unix-style system. Windows doesn't consider "everything to be a file" and sometimes respects case sensitivity in both file and directory names in unexpected ways.

As you'll see in Recipe 10.6, the differences between operating system paradigms also lead to minor differences in how functions behave.

Specifically, file locking will work differently if your program is run on Windows because of differences in the underlying operating system calls.

The recipes that follow cover the most common filesystem operations you might experience in PHP, from opening and manipulating files to locking them from being touched by other processes.

10.1 Creating or Opening a Local File

Problem

You need to open a file for reading or writing on the local filesystem.

Solution

Use `fopen()` to open the file and return a resource reference for further use:

```
$fp = fopen('document.txt', 'r');
```

Discussion

Internally, an open file is represented as a stream within PHP. You can read data from or write data to any position within the stream based on the position of the current file pointer. In the Solution example, you've opened a stream for reading only (attempting to write to this stream will fail) and positioned the pointer at the beginning of the file.

Example 10-1 shows how you can read as many bytes from the file as you want and then close the stream by passing its reference into `fclose()`.

Example 10-1. Reading bytes from a buffer

```
while (($buffer = fgets($fp, 4096)) !== false) {
                              ❶
    echo $buffer;
                              ❷
}
```

```
fclose($fp);
```
❸

The `fgets()` function reads one❶ line from the specified resource, stopping either when it hits a newline character or when it has read the specified number of bytes (4,096) from the underlying stream. If there is no data to read, the function returns `false`.

Once you have data buffered into❷ a variable, you can do with it whatever you want. In this case, print that single line to the console.

After using a file's contents, you❸ should explicitly close and clean up the resource you've created.

In addition to reading a file, `fopen()` allows for arbitrary writes, file appending, overwriting, or truncation. Each operation is determined by the mode passed as the second parameter—the Solution example passed `r` to indicate a read-only mode. Additional modes are described in Table 10-1.

Table 10-1. File modes available to `fopen()`

Mode	Description
`r`	Open for reading only; place the file pointer at the beginning of the file.
`w`	Open for writing only; place the file pointer at the beginning of the file and truncate the file to 0 length. If the file does not exist, attempt to create it.
`a`	Open for writing only; place the file pointer at the end of the file. If the file does not exist, attempt to create it. In this mode, `fseek()` has no effect, and writes are always appended.
`x`	Create and open for writing only; place the file pointer at the beginning of the file. If the file already exists, the `fopen()` call will fail by returning `false` and generating an error of level `E_WARNING`. If the file does not exist, attempt to create it.
`c`	Open the file for writing only. If the file does not exist, it is created. If it exists, it is neither truncated (as opposed to `w`),

Mode	Description
	nor does the call to this function fail (as is the case with `x`). The file pointer is placed at the beginning of the file.
`e`	Set close-on-exec flag on the opened file descriptor.

For all of the file modes documented in Table 10-1 *except* for `e`, you can append a literal `+` sign to the mode to open a file for both reading *and* writing rather than one operation or the other.

The `fopen()` function works with more than just local files. By default, the function assumes you want to work with the local filesystem, which is why you do not need to explicitly specify the `file://` protocol handler. However, you can just as easily reference remote files by using the `http://` or `ftp://` handlers, as follows:

```
$fp = fopen('https://eamann.com/', 'r');
```

NOTE

While remote file includes are possible, they can be dangerous in many situations, as you might not always have control over the contents returned by a remote filesystem. It's often recommended to disable remote file access by toggling `allow_url_include` in your system configuration. Refer to the PHP runtime configuration documents for instructions on implementing this change.

An optional third parameter allows `fopen()` to search for a file in your system include path if desired. By default, PHP will only search the local directory (or use an absolute path if specified). Loading files from the system include path encourages code reuse as you can specify individual classes or configuration files without replicating them throughout your project.

See Also

Documentation on the PHP filesystem, particularly `fopen()`.

10.2 Reading a File into a String

Problem

You want to read an entire file into a variable for use elsewhere in your application.

Solution

Use `file_get_contents()` as follows:

```
$config = file_get_contents('config.json');

if ($config !== false) {
    $parsed = json_decode($config);

    // ...
}
```

Discussion

The `file_get_contents()` function opens a file for reading, reads the entire data of that file into a variable, and then closes the file and allows you to use that data as a string. This is functionally equivalent to reading a file into a string manually with `fread()`, as in Example 10-2.

Example 10-2. Implementing `file_get_contents()` manually with `fread()`

```
function fileGetContents(string $filename): string|fals
{
    $buffer = '';
    $fp = fopen($filename, 'r');

    try {
        while (!feof($fp)) {
            $buffer .= fread($fp, 4096);
        }
    } catch(Exception $e) {
        $buffer = false;
```

```
        } finally {
            fclose($fp);
        }

        return $buffer;
    }

    $config = fileGetContents('config.json');
```

While it's possible to manually read a file into memory, as demonstrated in Example 10-2, it's a better idea to focus on writing simple programs and using the functions exposed by the language to handle complicated operations for you. The `file_get_contents()` function is implemented in C and provides a high level of performance for your application. It is binary-safe and leverages the memory-mapping functionality exposed by your operating system to achieve peak performance.

Like `fread()`, `file_get_contents()` can read both local and remote files into memory. It can also search for files in the system include path if you should set the optional second parameter to `true`.

Like `fread()`'s parallel `fwrite()` operation, there is an automatic write equivalent function called `file_put_contents()`. This function abstracts away the complexity of opening a file and overwriting its contents with string data from a variable. The following demonstrates how an object might be encoded to JSON and written out to a static file:

```
$config = new Config(/** ... **/);
$serialized = json_encode($config);

file_put_contents('config.json', $serialized);
```

See Also

Documentation on `file_get_contents()` and `file_put_contents()`.

10.3 Reading a Specific Slice of a File

Problem

You want to read a specific set of bytes from a particular position within a file.

Solution

Use `fopen()` to create a resource, `fseek()` to reposition the pointer within the file, and `fread()` to read data from that position as follows:

```
$fp = fopen('document.txt', 'r');
fseek($fp, 32, SEEK_SET);

$data = fread($fp, 32);
```

Discussion

By default, `fopen()` in read mode will open the file as a resource and place its pointer at the beginning of the file. When you start reading bytes from the file, the pointer will advance until it hits the end of the file. You can use `fseek()` to set the pointer to an arbitrary position within the resource, with the default being the beginning of the file.

The third parameter— `SEEK_SET` in the Solution example—tells PHP where to add the offset. You have three options:

- `SEEK_SET` (the default) sets the pointer from the beginning of the file.
- `SEEK_CUR` adds the offset to the current pointer position.
- `SEEK_END` adds the offset to the end of the file. This is useful for reading the last bytes in a file by setting a negative offset as the second parameter.

Assume you want to read the last bytes in a long log file from within PHP. You would do so similarly to the way you read arbitrary bytes in the Solution example but with a negative offset, as follows:

```
$fp = fopen('log.txt', 'r');
fseek($fp, -4096, SEEK_END);

echo fread($fp, 4096);

fclose($fp);
```

Note that, even if the log file in the preceding snippet is less than 4,096 bytes long, PHP will not read past the beginning of the file. The interpreter will instead place the pointer at the beginning of the file and start reading bytes from that position. Likewise, you cannot read past the end of the file regardless of how many bytes you specify in your call to `fread()`.

See Also

Recipe 10.1 for more on `fopen()`, and the documentation on `fread()` and `fseek()`.

10.4 Modifying a File in Place

Problem

You want to modify a specific part of a file.

Solution

Open the file for reading and writing by using `fopen()`, then use `fseek()` to move the pointer to the position you wish to update and overwrite a certain number of bytes starting with that position. For example:

```
$fp = fopen('resume.txt', 'r+');
fseek($fp, 32);

fwrite($fp, 'New data', 8);

fclose($fp);
```

Discussion

As in Recipe 10.3, the `fseek()` function is leveraged to move the pointer to an arbitrary location within the file. From there, `fwrite()` is used to write a specific set of bytes to the file in that location before you close the resource.

The third parameter passed to `fwrite()` tells PHP how many bytes to write. By default, the system will write all of the data passed in the second parameter, but you can restrict the amount of data written out by specifying a byte count. In the Solution example, the write length is set equal to the data length, which is redundant. A more realistic example of this functionality would appear something like the following.

```
$contents = 'the quick brown fox jumped over the lazy c
fwrite($fp, $contents, 9);
```

Note also that the Solution example adds a plus sign to the typical read mode; this opens the file for reading *and* writing. Opening the file in other modes leads to very different behavior:

- `w` (write mode), with or without the ability to read, will truncate the file before you do anything else with it!
- `a` (append mode), with or without the ability to read, will force the file pointer to the end of the file. Calls to `fseek()` will *not* move the file pointer as expected, and your new data will always be appended to the file.

See Also

Recipe 10.3 for more information on random I/O with files in PHP.

10.5 Writing to Many Files Simultaneously

Problem

You want to write data to multiple files at the same time. For example, you want to write both to the local filesystem and to the console.

Solution

Open multiple resource references with `fopen()` and write to them all in a loop:

```
$fps = [
    fopen('data.txt', 'w'),
    fopen('php://stdout', 'w')
];

foreach ($fps as $fp) {
    fwrite($fp, 'The wheels on the bus go round and rou
}
```

Discussion

PHP is generally a single-threaded system that must perform operations one at a time.[1] While the Solution example will produce output for two file references, it will write first to one and then to the other. In practice, this will be fast enough to be acceptable but is not truly simultaneous.

Even with this limitation, knowing that you can write the same data to multiple files with ease makes it fairly straightforward to juggle multiple potential outputs. Rather than crafting a procedural approach with a finite number of files as in the Solution example, you could even abstract this kind of operation into a class, as shown in Example 10-3:

Example 10-3. A simple class for abstracting multiple file operations

```
class MultiFile
{
    private array $handles = [];

    public function open(
        string $filename,
        string $mode = 'w',
        bool $use_include_path = false,
        $context = null
        ): mixed
    {
        $fp = fopen($filename, $mode, $use_include_path

        if ($fp !== false) {
            $this->handles[] = $fp;
        }

        return $fp;
    }

    public function write(string $data, ?int $length =
    {
        $success = true;
        $bytes = 0;

        foreach($this->handles as $fp) {
            $out = fwrite($fp, $data, $length);
            if ($out === false) {
                $success = false;
            } else {
                $bytes = $out;
            }
        }

        return $success ? $bytes : false;
    }

    public function close(): bool
    {
        $return = true;

        foreach ($this->handles as $fp) {
            $return = $return && fclose($fp);
```

```
            }

            return $return;
        }
    }
```

The class defined by Example 10-3 allows you to easily bind a write operation to multiple file handles and clean them up as necessary when you're done. Rather than opening each file in turn and manually iterating over them, you simply instantiate the class, add your files, and go. For example:

```
$writer = new MultiFile();
$writer->open('data.txt');
$writer->open('php://stdout');

$writer->write("Row, row, row your boat\nGently down th

$writer->close();
```

PHP's internal handling of resource pointers is highly efficient and empowers you to write to as many files or streams as necessary with minimal overhead. Abstractions like Example 10-3 similarly make it easy for you to focus on the business logic of your application, while PHP juggles the resource handles (and related memory allocation) for you.

See Also

Documentation on PHP's `stdout` stream.

10.6 Locking a File to Prevent Access or Modification by Another Process

Problem

You want to prevent another PHP process from manipulating a file while your script is running.

Solution

Use `flock()` to lock the file as follows:

```
$fp = fopen('myfile.txt', 'r');

if (flock($fp, LOCK_EX)) {
    // ... Do whatever reading you need

    flock($fp, LOCK_UN);
} else {
    echo 'Could not lock file!';
    exit(1);
}
```

Discussion

Often, you need to open a file to read its data or write something to it, but with the assurance that no other script will manipulate the file while you're working with it. The safest way to do this is by explicitly locking the file.

WARNING

On Windows, PHP leverages *mandatory locking* that is enforced by the operating system itself. Once a file is locked, no other process is permitted to open that file. On Unix-based systems (including Linux and macOS), PHP instead uses *advisory locking*. In this mode, the operating system can choose to ignore locks between different processes. While multiple PHP scripts will usually respect the lock, other processes might ignore it entirely.

An explicit file lock prevents other processes from either reading or writing the same file, depending on the type of lock. PHP supports two kinds of locks: a shared lock (`LOCK_SH`) that still permits reads, and an exclusive lock (`LOCK_EX`) that prevents other processes from accessing the file at all.

If you were to run the code in the Solution example twice on a machine (with a long-blocking operation like `sleep()` called before unlocking the file), the second process would pause and wait for the lock to be released before executing. A more concrete example is shown in Example 10-4.

Example 10-4. Illustration of a long-running file lock

```
$fp = fopen('myfile.txt', 'r');

echo 'Getting a lock ...' . PHP_EOL;
if (flock($fp, LOCK_EX)) {
    echo 'Sleeping ...' . PHP_EOL;
    for($i = 0; $i < 3; $i++) {
        sleep(10);
        echo '  Zzz ...' . PHP_EOL;
    }

    echo 'Unlocking ...' . PHP_EOL;
    flock($fp, LOCK_UN);
} else {
    echo 'Could not lock file!';
    exit(1);
}
```

Running the preceding program in two separate terminals side by side illustrates how locking works, as shown in Figure 10-1. The first execution will acquire the file lock and continue operating as expected. The second will wait until the lock is available and, after it acquires the lock, continue merrily along.

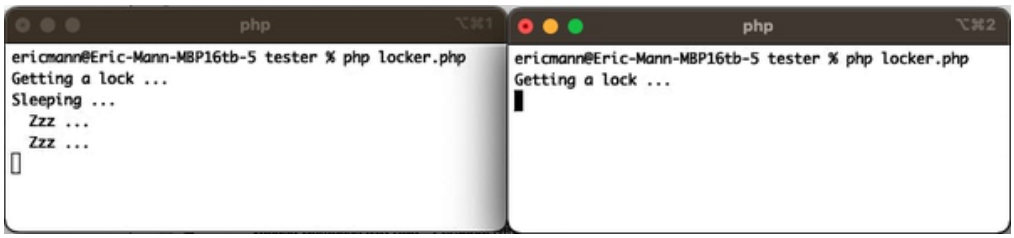

Figure 10-1. Two processes cannot acquire the same lock on a single file

See Also

Documentation on `flock()`.

[1] Chapter 17 covers parallel and asynchronous operations at length to explain ways to break out of a single-threaded paradigm.