PHP

The Right Way

LAST UPDATED: 2020-04-03 13:55:07 +0000

SHARE ON TWITTER

CHAPTER 1.

Welcome

There's a lot of outdated information on the Web that leads new PHP users astray, propagating bad practices and insecure code. *PHP: The Right Way* is an easy-to-read, quick reference for PHP popular coding standards, links to authoritative tutorials around the Web and what the contributors consider to be best practices at the present time.

There is no canonical way to use PHP. This website aims to introduce new PHP developers to some topics which they may not discover until it is too late, and aims to give seasoned pros some fresh ideas on those topics they've been doing for years without ever reconsidering. This website will also not tell you which tools to use, but instead offer suggestions for multiple options, when possible explaining the differences in approach and use-case.

This is a living document and will continue to be updated with more helpful information and examples as they become available.

Translations

PHP: The Right Way is translated into many different languages:

- English
- Español
- Français
- Indonesia
- Italiano
- Polski
- Português do Brasil
- Română
- Slovenščina
- Srpski
- Türkçe
- български
- Русский язык
- Українська
- العربية
- فارسی
- ภาษาไทย
- 한국어판
- 日本語
- 简体中文
- 繁體中文

Book

The most recent version of *PHP: The Right Way* is also available in PDF, EPUB and MOBI formats. Go to Leanpub

How to Contribute

Help make this website the best resource for new PHP programmers! Contribute on GitHub

BACK TO TOP

CHAPTER 2.

Getting Started

Use the Current Stable Version (7.4)

If you are getting started with PHP, start with the current stable release of PHP 7.4. PHP 7.x adds many new features over the older 5.x versions. The engine has been largely re-written, and PHP is now even quicker than older versions.

You should try to upgrade to the latest stable version quickly - PHP 5.6 is already End of Life. Upgrading is easy, as there are not many backwards compatibility breaks. If you are not sure which version a function or feature is in, you can check the PHP documentation on the php.net website.

Built-in web server

With PHP 5.4 or newer, you can start learning PHP without installing and configuring a full-fledged web server. To start the server, run the following command from your terminal in your project's web root:

```
> php -S localhost:8000
```

- Learn about the built-in, command line web server

Mac Setup

macOS comes prepackaged with PHP but it is normally a little behind the latest stable release. There are multiple ways to install the latest PHP version on macOS.

Install PHP via Homebrew

Homebrew is a package manager for macOS that helps you easily install PHP and various extensions. The Homebrew core repository provides "formulae" for PHP 5.6, 7.0, 7.1, 7.2, 7.3 and 7.4. Install the latest version with this command:

```
brew install php@7.4
```

You can switch between Homebrew PHP versions by modifying your `PATH` variable. Alternatively, you can use brew-php-switcher to switch PHP versions automatically.

Install PHP via Macports

The MacPorts Project is an open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading either command-line, X11 or Aqua based open-source software on the OS X operating system.

MacPorts supports pre-compiled binaries, so you don't need to recompile every dependency from the source tarball files, it saves your life if you don't have any package installed on your system.

At this point, you can install `php54`, `php55`, `php56`, `php70`, `php71`, `php72`, `php73` or `php74` using the `port install` command, for example:

```
sudo port install php56

sudo port install php74
```

And you can run `select` command to switch your active PHP:

```
sudo port select --set php php74
```

Install PHP via phpbrew

phpbrew is a tool for installing and managing multiple PHP versions. This can be really useful if two different applications/projects require different versions of PHP, and you are not using virtual machines.

Install PHP via Liip's binary installer

Another popular option is php-osx.liip.ch which provides one liner installation methods for versions 5.3 through 7.3. It doesn't overwrite the PHP binaries installed by Apple, but installs everything in a separate location (/usr/local/php5).

Compile from Source

Another option that gives you control over the version of PHP you install, is to compile it yourself. In that case be sure to have installed either Xcode or Apple's substitute "Command Line Tools for XCode" downloadable from Apple's Mac Developer Center.

All-in-One Installers

The solutions listed above mainly handle PHP itself, and do not supply things like Apache, Nginx or a SQL server. "All-in-one" solutions such as MAMP and XAMPP will install these other bits of software for you and tie them all together, but ease of setup comes with a trade-off of flexibility.

Windows Setup

You can download the binaries from windows.php.net/download. After the extraction of PHP, it is recommended to set the PATH to the root of your PHP folder (where php.exe is located) so you can execute PHP from anywhere.

For learning and local development, you can use the built in webserver with PHP 5.4+ so you don't need to worry about configuring it. If you would like an "all-in-one" which includes a full-blown webserver and MySQL too then tools such as the Web Platform Installer, XAMPP, EasyPHP, OpenServer and WAMP will help get a Windows development environment up and running fast. That said, these tools will be a little different from production so be careful of environment differences if you are working on Windows and deploying to Linux.

If you need to run your production system on Windows, then IIS7 will give you the most stable and best performance. You can use phpmanager (a GUI plugin for IIS7) to make configuring and managing PHP simple. IIS7 comes with FastCGI built in and ready to go, you just need to configure PHP as a handler. For support and additional resources there is a dedicated area on iis.net for PHP.

Generally running your application on different environment in development and production can lead to strange bugs popping up when you go live. If you are developing on Windows and deploying to Linux (or anything non-Windows) then you should consider using a Virtual Machine.

Chris Tankersley has a very helpful blog post on what tools he uses to do PHP development using Windows.

Common Directory structure

A common question among those starting out with writing programs for the web is, “where do I put my stuff?” Over the years, this answer has consistently been “where the `DocumentRoot` is.” Although this answer is not complete, it’s a great place to start.

For security reasons, configuration files should not be accessible by a site’s visitors; therefore, public scripts are kept in a public directory and private configurations and data are kept outside of that directory.

For each team, CMS, or framework one works in, a standard directory structure is used by each of those entities. However, if one is starting a project alone, knowing which filesystem structure to use can be daunting.

Paul M. Jones has done some fantastic research into common practices of tens of thousands of github projects in the realm of PHP. He has compiled a standard file and directory structure, the Standard PHP Package Skeleton, based on this research. In this directory structure, `DocumentRoot` should point to `public/`, unit tests should be in the `tests/` directory, and third party libraries, as installed by composer, belong in the `vendor/` directory. For other files and directories, abiding by the Standard PHP Package Skeleton will make the most sense to contributors of a project.

BACK TO TOP

CHAPTER 3.

Code Style Guide

The PHP community is large and diverse, composed of innumerable libraries, frameworks, and components. It is common for PHP developers to choose several of these and combine them into a single project. It is important that PHP code adhere (as close as possible) to a common code style to make it easy for developers to mix and match various libraries for their projects.

The Framework Interop Group has proposed and approved a series of style recommendations. Not all of them related to code-style, but those that do are PSR-1, PSR-12 and PSR-4. These recommendations are merely a set of rules that many projects like Drupal, Zend, Symfony, Laravel, CakePHP, phpBB, AWS SDK, FuelPHP, Lithium, etc are adopting. You can use them for your own projects, or continue to use your own personal style.

Ideally, you should write PHP code that adheres to a known standard. This could be any combination of PSRs, or one of the coding standards made by PEAR or Zend. This means other developers can easily read and work with your code, and applications that implement the components can have consistency even when working with lots of third-party code.

- Read about PSR-1

- Read about PSR-12
- Read about PSR-4
- Read about PEAR Coding Standards
- Read about Symfony Coding Standards

You can use PHP_CodeSniffer to check code against any one of these recommendations, and plugins for text editors like Sublime Text to be given real-time feedback.

You can fix the code layout automatically by using one of the following tools:

- One is the PHP Coding Standards Fixer which has a very well tested codebase.
- Also, the PHP Code Beautifier and Fixer tool which is included with PHP_CodeSniffer can be used to adjust your code accordingly.

And you can run phpcs manually from shell:

```
phpcs -sw --standard=PSR1 file.php
```

It will show errors and describe how to fix them. It can also be helpful to include this command in a git hook. That way, branches which contain violations against the chosen standard cannot enter the repository until those violations have been fixed.

If you have PHP_CodeSniffer, then you can fix the code layout problems reported by it, automatically, with the PHP Code Beautifier and Fixer.

```
phpcbf -w --standard=PSR1 file.php
```

Another option is to use the PHP Coding Standards Fixer. It will show which kind of errors the code structure had before it fixed them.

```
php-cs-fixer fix -v --rules=@PSR1 file.php
```

English is preferred for all symbol names and code infrastructure. Comments may be written in any language easily readable by all current and future parties who may be working on the codebase.

Finally, a good supplementary resource for writing clean PHP code is Clean Code PHP.

BACK TO TOP

CHAPTER 4.

Language Highlights

Programming Paradigms

PHP is a flexible, dynamic language that supports a variety of programming techniques. It has evolved dramatically over the years, notably adding a solid object-oriented model in PHP 5.0 (2004), anonymous functions and namespaces in PHP 5.3 (2009), and traits in PHP 5.4 (2012).

Object-oriented Programming

PHP has a very complete set of object-oriented programming features including support for classes, abstract classes, interfaces, inheritance, constructors, cloning, exceptions, and more.

- Read about Object-oriented PHP
- Read about Traits

Functional Programming

PHP supports first-class functions, meaning that a function can be assigned to a variable. Both user-defined and built-in functions can be referenced by a variable and invoked dynamically. Functions can be passed as arguments to other functions (a feature called *Higher-order Functions*) and functions can return other functions.

Recursion, a feature that allows a function to call itself, is supported by the language, but most PHP code is focused on iteration.

New anonymous functions (with support for closures) are present since PHP 5.3 (2009).

PHP 5.4 added the ability to bind closures to an object's scope and also improved support for callables such that they can be used interchangeably with anonymous functions in almost all cases.

- Continue reading on Functional Programming in PHP
- Read about Anonymous Functions
- Read about the Closure class
- More details in the Closures RFC
- Read about Callables
- Read about dynamically invoking functions with `call_user_func_array()`

Meta Programming

PHP supports various forms of meta-programming through mechanisms like the Reflection API and Magic Methods. There are many Magic Methods available like `__get()`, `__set()`, `__clone()`, `__toString()`, `__invoke()`, etc. that allow developers to hook into class behavior. Ruby developers often say that PHP is lacking `method_missing`, but it is available as `__call()` and `__callStatic()`.

- Read about Magic Methods
- Read about Reflection
- Read about Overloading

Namespaces

As mentioned above, the PHP community has a lot of developers creating lots of code. This means that one library's PHP code might use the same class name as another. When both libraries are used in the same namespace, they collide and cause trouble.

Namespaces solve this problem. As described in the PHP reference manual, namespaces may be compared to operating system directories that *namespace* files; two files with the same name may co-exist in separate directories. Likewise, two PHP classes with the same name may co-exist in separate PHP namespaces. It's as simple as that.

It is important for you to namespace your code so that it may be used by other developers without fear of colliding with other libraries.

One recommended way to use namespaces is outlined in PSR-4, which aims to provide a standard file, class and namespace convention to allow plug-and-play code.

In October 2014 the PHP-FIG deprecated the previous autoloading standard: PSR-0. Both PSR-0 and PSR-4 are still perfectly usable. The latter requires PHP 5.3, so many PHP 5.2-only projects implement PSR-0.

If you're going to use an autoloader standard for a new application or package, look into PSR-4.

- Read about Namespaces
- Read about PSR-0
- Read about PSR-4

Standard PHP Library

The Standard PHP Library (SPL) is packaged with PHP and provides a collection of classes and interfaces. It is made up primarily of commonly needed datastructure classes (stack, queue, heap, and so on), and iterators which can traverse over these datastructures or your own classes which implement SPL interfaces.

- Read about the SPL
- SPL video course on Lynda.com(Paid)

Command Line Interface

PHP was created to write web applications, but is also useful for scripting command line interface (CLI) programs. Command line PHP programs can help automate common tasks like testing, deployment, and application administration.

CLI PHP programs are powerful because you can use your app's code directly without having to create and secure a web GUI for it. Just be sure **not** to put your CLI PHP scripts in your public web root!

Try running PHP from your command line:

```
> php -i
```

The `-i` option will print your PHP configuration just like the `phpinfo()` function.

The `-a` option provides an interactive shell, similar to ruby's IRB or python's interactive shell. There are a number of other useful command line options, too.

Let's write a simple "Hello, $name" CLI program. To try it out, create a file named `hello.php`, as below.

```
<?php
if ($argc !== 2) {
    echo "Usage: php hello.php <name>.\n";
    exit(1);
}
$name = $argv[1];
echo "Hello, $name\n";
```

PHP sets up two special variables based on the arguments your script is run with. `$argc` is an integer variable containing the argument *count* and `$argv` is an array variable containing each argument's *value*. The first argument is always the name of your PHP script file, in this case `hello.php`.

The `exit()` expression is used with a non-zero number to let the shell know that the command failed. Commonly used exit codes can be found here.

To run our script, above, from the command line:

```
> php hello.php
Usage: php hello.php <name>
> php hello.php world
Hello, world
```

- Learn about running PHP from the command line

Xdebug

One of the most useful tools in software development is a proper debugger. It allows you to trace the execution of your code and monitor the contents of the stack. Xdebug, PHP's debugger, can be utilized by various IDEs to provide Breakpoints and stack inspection. It can also allow tools like PHPUnit and KCacheGrind to perform code coverage analysis and code profiling.

If you find yourself in a bind, willing to resort to `var_dump()`/`print_r()`, and you still can't find the solution - maybe you need to use the debugger.

Installing Xdebug can be tricky, but one of its most important features is "Remote Debugging" - if you develop code locally and then test it inside a VM or on another server, Remote Debugging is the feature that you will want to enable right away.

Traditionally, you will modify your Apache VHost or .htaccess file with these values:

```
php_value xdebug.remote_host 192.168.?.?
php_value xdebug.remote_port 9000
```

The “remote host” and “remote port” will correspond to your local computer and the port that you configure your IDE to listen on. Then it’s just a matter of putting your IDE into “listen for connections” mode, and loading the URL:

```
http://your-website.example.com/index.php?XDEBUG_SESSION_START=1
```

Your IDE will now intercept the current state as the script executes, allowing you to set breakpoints and probe the values in memory.

Graphical debuggers make it very easy to step through code, inspect variables, and eval code against the live runtime. Many IDE’s have built-in or plugin-based support for graphical debugging with Xdebug. MacGDBp is a free, open-source, stand-alone Xdebug GUI for Mac.

- Learn more about Xdebug
- Learn more about MacGDBp

BACK TO TOP

CHAPTER 5.

Dependency Management

There are a ton of PHP libraries, frameworks, and components to choose from. Your project will likely use several of them — these are project dependencies. Until recently, PHP did not have a good way to manage these project dependencies. Even if you managed them manually, you still had to worry about autoloaders. That is no longer an issue.

Currently there are two major package management systems for PHP - Composer and PEAR. Composer is currently the most popular package manager for PHP, however for a long time PEAR was the primary package manager in use. Knowing PEAR’s history is a good idea, since you may still find references to it even if you never use it.

Composer and Packagist

Composer is the recommended dependency manager for PHP. List your project’s dependencies in a `composer.json` file and, with a few simple commands, Composer will automatically download your project’s dependencies and setup autoloading for you. Composer is analogous to NPM in the node.js world, or Bundler in the Ruby world.

There is a plethora of PHP libraries that are compatible with Composer and ready to be used in your project. These “packages” are listed on Packagist, the official repository for Composer-

compatible PHP libraries.

How to Install Composer

The safest way to download composer is by following the official instructions. This will verify the installer is not corrupt or tampered with. The installer installs a `composer.phar` binary in your *current working directory*.

We recommend installing Composer *globally* (e.g. a single copy in `/usr/local/bin`). To do so, run this command next:

```
mv composer.phar /usr/local/bin/composer
```

Note: If the above fails due to permissions, prefix with `sudo`.

To run a locally installed Composer you'd use `php composer.phar`, globally it's simply `composer`.

Installing on Windows

For Windows users the easiest way to get up and running is to use the ComposerSetup installer, which performs a global install and sets up your `$PATH` so that you can just call `composer` from any directory in your command line.

How to Define and Install Dependencies

Composer keeps track of your project's dependencies in a file called `composer.json`. You can manage it by hand if you like, or use Composer itself. The `composer require` command adds a project dependency and if you don't have a `composer.json` file, one will be created. Here's an example that adds Twig as a dependency of your project.

```
composer require twig/twig:^2.0
```

Alternatively, the `composer init` command will guide you through creating a full `composer.json` file for your project. Either way, once you've created your `composer.json` file you can tell Composer to download and install your dependencies into the `vendor/` directory. This also applies to projects you've downloaded that already provide a `composer.json` file:

```
composer install
```

Next, add this line to your application's primary PHP file; this will tell PHP to use Composer's autoloader for your project dependencies.

```
<?php
require 'vendor/autoload.php';
```

Now you can use your project dependencies, and they'll be autoloaded on demand.

Updating your dependencies

Composer creates a file called `composer.lock` which stores the exact version of each package it downloaded when you first ran `composer install`. If you share your project with others, ensure the `composer.lock` file is included, so that when they run `composer install` they'll get the same versions as you. To update your dependencies, run `composer update`. Don't use `composer update` when deploying, only `composer install`, otherwise you may end up with different package versions on production.

This is most useful when you define your version requirements flexibly. For instance, a version requirement of `~1.8` means "anything newer than `1.8.0`, but less than `2.0.x-dev`". You can also use the `*` wildcard as in `1.8.*`. Now Composer's `composer update` command will upgrade all your dependencies to the newest version that fits the restrictions you define.

Update Notifications

To receive notifications about new version releases you can sign up for libraries.io, a web service that can monitor dependencies and send you alerts on updates.

Checking your dependencies for security issues

The Security Advisories Checker is a web service and a command-line tool, both will examine your `composer.lock` file and tell you if you need to update any of your dependencies.

Handling global dependencies with Composer

Composer can also handle global dependencies and their binaries. Usage is straight-forward, all you need to do is prefix your command with `global`. If for example you wanted to install PHPUnit and have it available globally, you'd run the following command:

```
composer global require phpunit/phpunit
```

This will create a `~/.composer` folder where your global dependencies reside. To have the installed packages' binaries available everywhere, you'd then add the `~/.composer/vendor/bin` folder to your `$PATH` variable.

- Learn about Composer

PEAR

A veteran package manager that some PHP developers enjoy is PEAR. It behaves similarly to Composer, but has some notable differences.

PEAR requires each package to have a specific structure, which means that the author of the package must prepare it for usage with PEAR. Using a project which was not prepared to work with PEAR is not possible.

PEAR installs packages globally, which means after installing them once they are available to all projects on that server. This can be good if many projects rely on the same package with the same

version but might lead to problems if version conflicts between two projects arise.

How to install PEAR

You can install PEAR by downloading the `.phar` installer and executing it. The PEAR documentation has detailed install instructions for every operating system.

If you are using Linux, you can also have a look at your distribution package manager. Debian and Ubuntu, for example, have an apt `php-pear` package.

How to install a package

If the package is listed on the PEAR packages list, you can install it by specifying the official name:

```
pear install foo
```

If the package is hosted on another channel, you need to `discover` the channel first and also specify it when installing. See the Using channel docs for more information on this topic.

- Learn about PEAR

Handling PEAR dependencies with Composer

If you are already using Composer and you would like to install some PEAR code too, you can use Composer to handle your PEAR dependencies. This example will install code from `pear2.php.net`:

```
{
    "repositories": [
        {
            "type": "pear",
            "url": "https://pear2.php.net"
        }
    ],
    "require": {
        "pear-pear2/PEAR2_Text_Markdown": "*",
        "pear-pear2/PEAR2_HTTP_Request": "*"
    }
}
```

The first section `"repositories"` will be used to let Composer know it should “initialize” (or “discover” in PEAR terminology) the pear repo. Then the `require` section will prefix the package name like this:

> *pear-channel/Package*

The “pear” prefix is hardcoded to avoid any conflicts, as a pear channel could be the same as another packages vendor name for example, then the channel short name (or full URL) can be used

to reference which channel the package is in.

When this code is installed it will be available in your vendor directory and automatically available through the Composer autoloader:

> *vendor/pear-pear2.php.net/PEAR2_HTTP_Request/pear2/HTTP/Request.php*

To use this PEAR package simply reference it like so:

```
<?php
$request = new pear2\HTTP\Request();
```

- Learn more about using PEAR with Composer

BACK TO TOP

CHAPTER 6.

Coding Practices

The Basics

PHP is a vast language that allows coders of all levels the ability to produce code not only quickly, but efficiently. However, while advancing through the language, we often forget the basics that we first learnt (or overlooked) in favor of short cuts and/or bad habits. To help combat this common issue, this section is aimed at reminding coders of the basic coding practices within PHP.

- Continue reading on The Basics

Date and Time

PHP has a class named DateTime to help you when reading, writing, comparing or calculating with date and time. There are many date and time related functions in PHP besides DateTime, but it provides nice object-oriented interface to most common uses. It can handle time zones, but that is outside this short introduction.

To start working with DateTime, convert raw date and time string to an object with `createFromFormat()` factory method or do `new DateTime` to get the current date and time. Use `format()` method to convert DateTime back to a string for output.

```
<?php
$raw = '22. 11. 1968';
$start = DateTime::createFromFormat('d. m. Y', $raw);
```

```php
echo 'Start date: ' . $start->format('Y-m-d') . "\n";
```

Calculating with DateTime is possible with the DateInterval class. DateTime has methods like `add()` and `sub()` that take a DateInterval as an argument. Do not write code that expect same number of seconds in every day, both daylight saving and timezone alterations will break that assumption. Use date intervals instead. To calculate date difference use the `diff()` method. It will return new DateInterval, which is super easy to display.

```php
<?php
// create a copy of $start and add one month and 6 days
$end = clone $start;
$end->add(new DateInterval('P1M6D'));

$diff = $end->diff($start);
echo 'Difference: ' . $diff->format('%m month, %d days (total: %a days)') . "\n";
// Difference: 1 month, 6 days (total: 37 days)
```

On DateTime objects you can use standard comparison:

```php
<?php
if ($start < $end) {
    echo "Start is before the end!\n";
}
```

One last example to demonstrate the DatePeriod class. It is used to iterate over recurring events. It can take two DateTime objects, start and end, and the interval for which it will return all events in between.

```php
<?php
// output all thursdays between $start and $end
$periodInterval = DateInterval::createFromDateString('first thursday');
$periodIterator = new DatePeriod($start, $periodInterval, $end, DatePeriod::EXCLUDE_START_DATE);
foreach ($periodIterator as $date) {
    // output each date in the period
    echo $date->format('Y-m-d') . ' ';
}
```

A popular PHP API extension is Carbon. It inherits everything in the DateTime class, so involves minimal code alterations, but extra features include Localization support, further ways to add, subtract and format a DateTime object, plus a means to test your code by simulating a date and time of your choosing.

- Read about DateTime
- Read about date formatting (accepted date format string options)

Design Patterns

When you are building your application it is helpful to use common patterns in your code and common patterns for the overall structure of your project. Using common patterns is helpful because it makes it much easier to manage your code and lets other developers quickly understand how everything fits together.

If you use a framework then most of the higher level code and project structure will be based on that framework, so a lot of the pattern decisions are made for you. But it is still up to you to pick out the best patterns to follow in the code you build on top of the framework. If, on the other hand, you are not using a framework to build your application then you have to find the patterns that best suit the type and size of application that you're building.

You can learn more about PHP design patterns and see working examples at:

https://designpatternsphp.readthedocs.io/

Working with UTF-8

This section was originally written by Alex Cabal over at PHP Best Practices and has been used as the basis for our own UTF-8 advice.

There's no one-liner. Be careful, detailed, and consistent.

Right now PHP does not support Unicode at a low level. There are ways to ensure that UTF-8 strings are processed OK, but it's not easy, and it requires digging in to almost all levels of the web app, from HTML to SQL to PHP. We'll aim for a brief, practical summary.

UTF-8 at the PHP level

The basic string operations, like concatenating two strings and assigning strings to variables, don't need anything special for UTF-8. However, most string functions, like `strpos()` and `strlen()`, do need special consideration. These functions often have an `mb_*` counterpart: for example, `mb_strpos()` and `mb_strlen()`. These `mb_*` strings are made available to you via the Multibyte String Extension, and are specifically designed to operate on Unicode strings.

You must use the `mb_*` functions whenever you operate on a Unicode string. For example, if you use `substr()` on a UTF-8 string, there's a good chance the result will include some garbled half-characters. The correct function to use would be the multibyte counterpart, `mb_substr()`.

The hard part is remembering to use the `mb_*` functions at all times. If you forget even just once, your Unicode string has a chance of being garbled during further processing.

Not all string functions have an `mb_*` counterpart. If there isn't one for what you want to do, then you might be out of luck.

You should use the `mb_internal_encoding()` function at the top of every PHP script you write (or at the top of your global include script), and the `mb_http_output()` function right after it if

your script is outputting to a browser. Explicitly defining the encoding of your strings in every script will save you a lot of headaches down the road.

Additionally, many PHP functions that operate on strings have an optional parameter letting you specify the character encoding. You should always explicitly indicate UTF-8 when given the option. For example, `htmlentities()` has an option for character encoding, and you should always specify UTF-8 if dealing with such strings. Note that as of PHP 5.4.0, UTF-8 is the default encoding for `htmlentities()` and `htmlspecialchars()`.

Finally, If you are building a distributed application and cannot be certain that the `mbstring` extension will be enabled, then consider using the patchwork/utf8 Composer package. This will use `mbstring` if it is available, and fall back to non UTF-8 functions if not.

UTF-8 at the Database level

If your PHP script accesses MySQL, there's a chance your strings could be stored as non-UTF-8 strings in the database even if you follow all of the precautions above.

To make sure your strings go from PHP to MySQL as UTF-8, make sure your database and tables are all set to the `utf8mb4` character set and collation, and that you use the `utf8mb4` character set in the PDO connection string. See example code below. This is *critically important*.

Note that you must use the `utf8mb4` character set for complete UTF-8 support, not the `utf8` character set! See Further Reading for why.

UTF-8 at the browser level

Use the `mb_http_output()` function to ensure that your PHP script outputs UTF-8 strings to your browser.

The browser will then need to be told by the HTTP response that this page should be considered as UTF-8. Today, it is common to set the character set in the HTTP response header like this:

```
<?php
header('Content-Type: text/html; charset=UTF-8')
```

The historic approach to doing that was to include the charset `<meta>` tag in your page's `<head>` tag.

```
<?php
// Tell PHP that we're using UTF-8 strings until the end of the script
mb_internal_encoding('UTF-8');
$utf_set = ini_set('default_charset', 'utf-8');
if (!$utf_set) {
    throw new Exception('could not set default_charset to utf-8, please ensure it\'s set on your system!');
}

// Tell PHP that we'll be outputting UTF-8 to the browser
```

```php
mb_http_output('UTF-8');

// Our UTF-8 test string
$string = 'Êl síla erin lû e-govaned vîn.';

// Transform the string in some way with a multibyte function
// Note how we cut the string at a non-Ascii character for demonstration purposes
$string = mb_substr($string, 0, 15);

// Connect to a database to store the transformed string
// See the PDO example in this document for more information
// Note the `charset=utf8mb4` in the Data Source Name (DSN)
$link = new PDO(
    'mysql:host=your-hostname;dbname=your-db;charset=utf8mb4',
    'your-username',
    'your-password',
    array(
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_PERSISTENT => false
    )
);

// Store our transformed string as UTF-8 in our database
// Your DB and tables are in the utf8mb4 character set and collation, right?
$handle = $link->prepare('insert into ElvishSentences (Id, Body, Priority) values (default, :body, :priority)
$handle->bindParam(':body', $string, PDO::PARAM_STR);
$priority = 45;
$handle->bindParam(':priority', $priority, PDO::PARAM_INT); // explicitly tell pdo to expect an int
$handle->execute();

// Retrieve the string we just stored to prove it was stored correctly
$handle = $link->prepare('select * from ElvishSentences where Id = :id');
$id = 7;
$handle->bindParam(':id', $id, PDO::PARAM_INT);
$handle->execute();

// Store the result into an object that we'll output later in our HTML
// This object won't kill your memory because it fetches the data Just-In-Time to
$result = $handle->fetchAll(\PDO::FETCH_OBJ);

// An example wrapper to allow you to escape data to html
function escape_to_html($dirty){
    echo htmlspecialchars($dirty, ENT_QUOTES, 'UTF-8');
}

header('Content-Type: text/html; charset=UTF-8'); // Unnecessary if your default_charset is set to utf-8 alre
```

```
?><!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>UTF-8 test page</title>
    </head>
    <body>
        <?php
        foreach($result as $row){
            escape_to_html($row->Body);  // This should correctly output our transformed UTF-8 string to the
        }
        ?>
    </body>
</html>
```

Further reading

- PHP Manual: String Operations
- PHP Manual: String Functions
  - `strpos()`
  - `strlen()`
  - `substr()`
- PHP Manual: Multibyte String Functions
  - `mb_strpos()`
  - `mb_strlen()`
  - `mb_substr()`
  - `mb_internal_encoding()`
  - `mb_http_output()`
  - `htmlentities()`
  - `htmlspecialchars()`
- Stack Overflow: What factors make PHP Unicode-incompatible?
- Stack Overflow: Best practices in PHP and MySQL with international strings
- How to support full Unicode in MySQL databases
- Bringing Unicode to PHP with Portable UTF-8
- Stack Overflow: DOMDocument loadHTML does not encode UTF-8 correctly

Internationalization (i18n) and Localization (l10n)

Disclaimer for newcomers: i18n and l10n are numeronyms, a kind of abbreviation where numbers are used to shorten words - in our case, internationalization becomes i18n and localization, l10n.

First of all, we need to define those two similar concepts and other related things:

- **Internationalization** is when you organize your code so it can be adapted to different languages or regions without refactorings. This action is usually done once - preferably, at the beginning of the project, or else you will probably need some huge changes in the source!

- **Localization** happens when you adapt the interface (mainly) by translating contents, based on the i18n work done before. It usually is done every time a new language or region needs support and is updated when new interface pieces are added, as they need to be available in all supported languages.
- **Pluralization** defines the rules required between distinct languages to interoperate strings containing numbers and counters. For instance, in English when you have only one item, it is singular, and anything different from that is called plural; plural in this language is indicated by adding an S after some words, and sometimes changes parts of it. In other languages, such as Russian or Serbian, there are two plural forms in addition to the singular - you may even find languages with a total of four, five or six forms, such as Slovenian, Irish or Arabic.

Common ways to implement

The easiest way to internationalize PHP software is by using array files and using those strings in templates, such as `<h1><?=$TRANS['title_about_page']?></h1>`. This way is, however, hardly recommended for serious projects, as it poses some maintenance issues along the road - some might appear in the very beginning, such as pluralization. So, please, don't try this if your project will contain more than a couple of pages.

The most classic way and often taken as reference for i18n and l10n is a Unix tool called `gettext`. It dates back to 1995 and is still a complete implementation for translating software. It is easy enough to get running, while still sporting powerful supporting tools. It is about Gettext we will be talking here. Also, to help you not get messy over the command-line, we will be presenting a great GUI application that can be used to easily update your l10n source

Other tools

There are common libraries used that support Gettext and other implementations of i18n. Some of them may seem easier to install or sport additional features or i18n file formats. In this document, we focus on the tools provided with the PHP core, but here we list others for completion:

- aura/intl: Provides internationalization (I18N) tools, specifically package-oriented per-locale message translation. It uses array formats for message. Does not provide a message extractor, but does provide advanced message formatting via the `intl` extension (including pluralized messages).
- oscarotero/Gettext: Gettext support with an OO interface; includes improved helper functions, powerful extractors for several file formats (some of them not supported natively by the `gettext` command), and can also export to other formats besides `.mo/.po` files. Can be useful if you need to integrate your translation files into other parts of the system, like a JavaScript interface.
- symfony/translation: supports a lot of different formats, but recommends using verbose XLIFF's. Doesn't include helper functions nor a built-in extractor, but supports placeholders using `strtr()` internally.
- zend/i18n: supports array and INI files, or Gettext formats. Implements a caching layer to save you from reading the filesystem every time. It also includes view helpers, and locale-aware input filters and validators. However, it has no message extractor.

Other frameworks also include i18n modules, but those are not available outside of their codebases:

- Laravel supports basic array files, has no automatic extractor but includes a `@lang` helper for template files.
- Yii supports array, Gettext, and database-based translation, and includes a messages extractor. It is backed by the `Intl` extension, available since PHP 5.3, and based on the ICU project; this enables Yii to run powerful replacements, like spelling out numbers, formatting dates, times, intervals, currency, and ordinals.

If you decide to go for one of the libraries that provide no extractors, you may want to use the gettext formats, so you can use the original gettext toolchain (including Poedit) as described in the rest of the chapter.

Gettext

Installation

You might need to install Gettext and the related PHP library by using your package manager, like `apt-get` or `yum`. After installed, enable it by adding `extension=gettext.so` (Linux/Unix) or `extension=php_gettext.dll` (Windows) to your `php.ini`.

Here we will also be using Poedit to create translation files. You will probably find it in your system's package manager; it is available for Unix, Mac, and Windows, and can be downloaded for free on their website as well.

Structure

Types of files

There are three files you usually deal with while working with gettext. The main ones are PO (Portable Object) and MO (Machine Object) files, the first being a list of readable "translated objects" and the second, the corresponding binary to be interpreted by gettext when doing localization. There's also a POT (Template) file, which simply contains all existing keys from your source files, and can be used as a guide to generate and update all PO files. Those template files are not mandatory: depending on the tool you are using to do l10n, you can go just fine with only PO/MO files. You will always have one pair of PO/MO files per language and region, but only one POT per domain.

Domains

There are some cases, in big projects, where you might need to separate translations when the same words convey different meaning given a context. In those cases, you split them into different *domains*. They are, basically, named groups of POT/PO/MO files, where the filename is the said *translation domain*. Small and medium-sized projects usually, for simplicity, use only one domain; its name is arbitrary, but we will be using "main" for our code samples. In Symfony projects, for example, domains are used to separate the translation for validation messages.

Locale code

A locale is simply a code that identifies one version of a language. It is defined following the ISO 639-1 and ISO 3166-1 alpha-2 specs: two lower-case letters for the language, optionally followed by an underline and two upper-case letters identifying the country or regional code. For rare languages, three letters are used.

For some speakers, the country part may seem redundant. In fact, some languages have dialects in different countries, such as Austrian German (`de_AT`) or Brazilian Portuguese (`pt_BR`). The second part is used to distinguish between those dialects - when it is not present, it is taken as a "generic" or "hybrid" version of the language.

Directory structure

To use Gettext, we will need to adhere to a specific structure of folders. First, you will need to select an arbitrary root for your l10n files in your source repository. Inside it, you will have a folder for each needed locale, and a fixed `LC_MESSAGES` folder that will contain all your PO/MO pairs. Example:

```
<project root>
 ├─ src/
 ├─ templates/
 └─ locales/
    ├─ forum.pot
    ├─ site.pot
    ├─ de/
    │  └─ LC_MESSAGES/
    │     ├─ forum.mo
    │     ├─ forum.po
    │     ├─ site.mo
    │     └─ site.po
    ├─ es_ES/
    │  └─ LC_MESSAGES/
    │     └─ ...
    ├─ fr/
    │  └─ ...
    ├─ pt_BR/
    │  └─ ...
    └─ pt_PT/
       └─ ...
```

Plural forms

As we said in the introduction, different languages might sport different plural rules. However, gettext saves us from this trouble once again. When creating a new `.po` file, you will have to declare the plural rules for that language, and translated pieces that are plural-sensitive will have a different form for each of those rules. When calling Gettext in code, you will have to specify the

number related to the sentence, and it will work out the correct form to use - even using string substitution if needed.

Plural rules include the number of plurals available and a boolean test with `n` that would define in which rule the given number falls (starting the count with 0). For example:

- Japanese: `nplurals=1; plural=0` - only one rule
- English: `nplurals=2; plural=(n != 1);` - two rules, first if N is one, second rule otherwise
- Brazilian Portuguese: `nplurals=2; plural=(n > 1);` - two rules, second if N is bigger than one, first otherwise

Now that you understood the basis of how plural rules works - and if you didn't, please look at a deeper explanation on the LingoHub tutorial -, you might want to copy the ones you need from a list instead of writing them by hand.

When calling out Gettext to do localization on sentences with counters, you will have to provide it the related number as well. Gettext will work out what rule should be in effect and use the correct localized version. You will need to include in the `.po` file a different sentence for each plural rule defined.

Sample implementation

After all that theory, let's get a little practical. Here's an excerpt of a `.po` file - don't mind with its format, but with the overall content instead; you will learn how to edit it easily later:

```
msgid ""
msgstr ""
"Language: pt_BR\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Plural-Forms: nplurals=2; plural=(n > 1);\n"

msgid "We are now translating some strings"
msgstr "Nós estamos traduzindo algumas strings agora"

msgid "Hello %1$s! Your last visit was on %2$s"
msgstr "Olá %1$s! Sua última visita foi em %2$s"

msgid "Only one unread message"
msgid_plural "%d unread messages"
msgstr[0] "Só uma mensagem não lida"
msgstr[1] "%d mensagens não lidas"
```

The first section works like a header, having the `msgid` and `msgstr` especially empty. It describes the file encoding, plural forms and other things that are less relevant. The second section translates a simple string from English to Brazilian Portuguese, and the third does the same, but leveraging string replacement from `sprintf` so the translation may contain the user name and visit date. The last section is a sample of pluralization forms, displaying the singular and plural version as `msgid`

in English and their corresponding translations as `msgstr` 0 and 1 (following the number given by the plural rule). There, string replacement is used as well so the number can be seen directly in the sentence, by using `%d`. The plural forms always have two `msgid` (singular and plural), so it is advised not to use a complex language as the source of translation.

Discussion on l10n keys

As you might have noticed, we are using as source ID the actual sentence in English. That `msgid` is the same used throughout all your `.po` files, meaning other languages will have the same format and the same `msgid` fields but translated `msgstr` lines.

Talking about translation keys, there are two main "schools" here:

1. *`msgid` as a real sentence*. The main advantages are:
   - if there are pieces of the software untranslated in any given language, the key displayed will still maintain some meaning. Example: if you happen to translate by heart from English to Spanish but need help to translate to French, you might publish the new page with missing French sentences, and parts of the website would be displayed in English instead;
   - it is much easier for the translator to understand what's going on and do a proper translation based on the `msgid`;
   - it gives you "free" l10n for one language - the source one;
   - The only disadvantage: if you need to change the actual text, you would need to replace the same `msgid` across several language files.
2. *`msgid` as a unique, structured key*. It would describe the sentence role in the application in a structured way, including the template or part where the string is located instead of its content.
   - it is a great way to have the code organized, separating the text content from the template logic.
   - however, that could bring problems to the translator that would miss the context. A source language file would be needed as a basis for other translations. Example: the developer would ideally have an `en.po` file, that translators would read to understand what to write in `fr.po` for instance.
   - missing translations would display meaningless keys on screen (`top_menu.welcome` instead of `Hello there, User!` on the said untranslated French page). That is good it as would force translation to be complete before publishing - however, bad as translation issues would be remarkably awful in the interface. Some libraries, though, include an option to specify a given language as "fallback", having a similar behavior as the other approach.

The Gettext manual favors the first approach as, in general, it is easier for translators and users in case of trouble. That is how we will be working here as well. However, the Symfony documentation favors keyword-based translation, to allow for independent changes of all translations without affecting templates as well.

Everyday usage

In a typical application, you would use some Gettext functions while writing static text in your pages. Those sentences would then appear in `.po` files, get translated, compiled into `.mo` files and then, used by Gettext when rendering the actual interface. Given that, let's tie together what we have discussed so far in a step-by-step example:

1. A sample template file, including some different gettext calls

```
<?php include 'i18n_setup.php' ?>
<div id="header">
    <h1><?=sprintf(gettext('Welcome, %s!'), $name)?></h1>
    <!-- code indented this way only for legibility -->
    <?php if ($unread): ?>
        <h2><?=sprintf(
            ngettext('Only one unread message',
                     '%d unread messages',
                     $unread),
            $unread)?>
        </h2>
    <?php endif ?>
</div>

<h1><?=gettext('Introduction')?></h1>
<p><?=gettext('We\'re now translating some strings')?></p>
```

- `gettext()` simply translates a `msgid` into its corresponding `msgstr` for a given language. There's also the shorthand function `_()` that works the same way;
- `ngettext()` does the same but with plural rules;
- there's also `dgettext()` and `dngettext()`, that allows you to override the domain for a single call. More on domain configuration in the next example.

2. A sample setup file (`i18n_setup.php` as used above), selecting the correct locale and configuring Gettext

```
<?php
/**
 * Verifies if the given $locale is supported in the project
 * @param string $locale
 * @return bool
 */
function valid($locale) {
   return in_array($locale, ['en_US', 'en', 'pt_BR', 'pt', 'es_ES', 'es']);
}

//setting the source/default locale, for informational purposes
$lang = 'en_US';

if (isset($_GET['lang']) && valid($_GET['lang'])) {
    // the locale can be changed through the query-string
    $lang = $_GET['lang'];    //you should sanitize this!
    setcookie('lang', $lang); //it's stored in a cookie so it can be reused
} elseif (isset($_COOKIE['lang']) && valid($_COOKIE['lang'])) {
```

```
    // if the cookie is present instead, let's just keep it
    $lang = $_COOKIE['lang']; //you should sanitize this!
} elseif (isset($_SERVER['HTTP_ACCEPT_LANGUAGE'])) {
    // default: look for the languages the browser says the user accepts
    $langs = explode(',', $_SERVER['HTTP_ACCEPT_LANGUAGE']);
    array_walk($langs, function (&$lang) { $lang = strtr(strtok($lang, ';'), ['-' => '_']); });
    foreach ($langs as $browser_lang) {
        if (valid($browser_lang)) {
            $lang = $browser_lang;
            break;
        }
    }
}

// here we define the global system locale given the found language
putenv("LANG=$lang");

// this might be useful for date functions (LC_TIME) or money formatting (LC_MONETARY), for instance
setlocale(LC_ALL, $lang);

// this will make Gettext look for ../locales/<lang>/LC_MESSAGES/main.mo
bindtextdomain('main', '../locales');

// indicates in what encoding the file should be read
bind_textdomain_codeset('main', 'UTF-8');

// if your application has additional domains, as cited before, you should bind them here as well
bindtextdomain('forum', '../locales');
bind_textdomain_codeset('forum', 'UTF-8');

// here we indicate the default domain the gettext() calls will respond to
textdomain('main');

// this would look for the string in forum.mo instead of main.mo
// echo dgettext('forum', 'Welcome back!');
?>
```

3. Preparing translation for the first run

One of the great advantages Gettext has over custom framework i18n packages is its extensive and powerful file format. “Oh man, that’s quite hard to understand and edit by hand, a simple array would be easier!” Make no mistake, applications like Poedit are here to help - *a lot*. You can get the program from their website, it’s free and available for all platforms. It’s a pretty easy tool to get used to, and a very powerful one at the same time - using all features Gettext has available. This guide is based on PoEdit 1.8.

In the first run, you should select “File > New…” from the menu. You’ll be asked straight ahead for the language: here you can select/filter the language you want to translate to, or use that format we mentioned before, such as `en_US` or `pt_BR`.

Now, save the file - using that directory structure we mentioned as well. Then you should click “Extract from sources”, and here you’ll configure various settings for the extraction and translation tasks. You’ll be able to find all those later through “Catalog > Properties”:

- Source paths: here you must include all folders from the project where `gettext()` (and siblings) are called - this is usually your templates/views folder(s). This is the only mandatory setting;
- Translation properties:
  - Project name and version, Team and Team’s email address: useful information that goes in the .po file header;
  - Plural forms: here go those rules we mentioned before - there’s a link in there with samples as well. You can leave it with the default option most of the time, as PoEdit already includes a handy database of plural rules for many languages.
  - Charsets: UTF-8, preferably;
  - Source code charset: set here the charset used by your codebase - probably UTF-8 as well, right?
- Source keywords: The underlying software knows how `gettext()` and similar function calls look like in several programming languages, but you might as well create your own translation functions. It will be here you’ll add those other methods. This will be discussed later in the “Tips” section.

After setting those points it will run a scan through your source files to find all the localization calls. After every scan PoEdit will display a summary of what was found and what was removed from the source files. New entries will fed empty into the translation table, and you’ll start typing in the localized versions of those strings. Save it and a .mo file will be (re)compiled into the same folder and ta-dah: your project is internationalized.

4. Translating strings

As you may have noticed before, there are two main types of localized strings: simple ones and those with plural forms. The first ones have simply two boxes: source and localized string. The source string cannot be modified as Gettext/Poedit do not include the powers to alter your source files - you should change the source itself and rescan the files. Tip: you may right-click a translation line and it will hint you with the source files and lines where that string is being used. On the other hand, plural form strings include two boxes to show the two source strings, and tabs so you can configure the different final forms.

Whenever you change your sources and need to update the translations, just hit Refresh and Poedit will rescan the code, removing non-existent entries, merging the ones that changed and adding new ones. It may also try to guess some translations, based on other ones you did. Those guesses and the changed entries will receive a “Fuzzy” marker, indicating it needs review, appearing golden in the list. It is also useful if you have a translation team and someone tries to write something they are not sure about: just mark Fuzzy, and someone else will review later.

Finally, it is advised to leave "View > Untranslated entries first" marked, as it will help you *a lot* to not forget any entry. From that menu, you can also open parts of the UI that allow you to leave contextual information for translators if needed.

Tips & Tricks

Possible caching issues

If you are running PHP as a module on Apache (`mod_php`), you might face issues with the `.mo` file being cached. It happens the first time it is read, and then, to update it, you might need to restart the server. On Nginx and PHP5 it usually takes only a couple of page refreshes to refresh the translation cache, and on PHP7 it is rarely needed.

Additional helper functions

As preferred by many people, it is easier to use `_()` instead of `gettext()`. Many custom i18n libraries from frameworks use something similar to `t()` as well, to make translated code shorter. However, that is the only function that sports a shortcut. You might want to add in your project some others, such as `__()` or `_n()` for `ngettext()`, or maybe a fancy `_r()` that would join `gettext()` and `sprintf()` calls. Other libraries, such as oscarotero's Gettext also provide helper functions like these.

In those cases, you'll need to instruct the Gettext utility on how to extract the strings from those new functions. Don't be afraid; it is very easy. It is just a field in the `.po` file, or a Settings screen on Poedit. In the editor, that option is inside "Catalog > Properties > Source keywords". Remember: Gettext already knows the default functions for many languages, so don't be afraid if that list seems empty. You need to include there the specifications of those new functions, following a specific format:

- if you create something like `t()` that simply returns the translation for a string, you can specify it as `t`. Gettext will know the only function argument is the string to be translated;
- if the function has more than one argument, you can specify in which one the first string is - and if needed, the plural form as well. For instance, if we call our function like this: `__('one user', '%d users', $number)`, the specification would be `__:1,2`, meaning the first form is the first argument, and the second form is the second argument. If your number comes as the first argument instead, the spec would be `__:2,3`, indicating the first form is the second argument, and so on.

After including those new rules in the `.po` file, a new scan will bring in your new strings just as easy as before.

References

- Wikipedia: i18n and l10n
- Wikipedia: Gettext
- LingoHub: PHP internationalization with gettext tutorial
- PHP Manual: Gettext

- Gettext Manual

BACK TO TOP

CHAPTER 7.

Dependency Injection

From Wikipedia:

> *Dependency injection is a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time.*

This quote makes the concept sound much more complicated than it actually is. Dependency Injection is providing a component with its dependencies either through constructor injection, method calls or the setting of properties. It is that simple.

Basic Concept

We can demonstrate the concept with a simple, yet naive example.

Here we have a `Database` class that requires an adapter to speak to the database. We instantiate the adapter in the constructor and create a hard dependency. This makes testing difficult and means the `Database` class is very tightly coupled to the adapter.

```
<?php
namespace Database;

class Database
{
    protected $adapter;

    public function __construct()
    {
        $this->adapter = new MySqlAdapter;
    }
}

class MysqlAdapter {}
```

This code can be refactored to use Dependency Injection and therefore loosen the dependency.

```
<?php
namespace Database;
```

```php
class Database
{
    protected $adapter;

    public function __construct(MySqlAdapter $adapter)
    {
        $this->adapter = $adapter;
    }
}

class MysqlAdapter {}
```

Now we are giving the `Database` class its dependency rather than creating it itself. We could even create a method that would accept an argument of the dependency and set it that way, or if the `$adapter` property was `public` we could set it directly.

Complex Problem

If you have ever read about Dependency Injection then you have probably seen the terms *"Inversion of Control"* or *"Dependency Inversion Principle"*. These are the complex problems that Dependency Injection solves.

Inversion of Control

Inversion of Control is as it says, "inverting the control" of a system by keeping organizational control entirely separate from our objects. In terms of Dependency Injection, this means loosening our dependencies by controlling and instantiating them elsewhere in the system.

For years, PHP frameworks have been achieving Inversion of Control, however, the question became, which part of control are we inverting, and where to? For example, MVC frameworks would generally provide a super object or base controller that other controllers must extend to gain access to its dependencies. This **is** Inversion of Control, however, instead of loosening dependencies, this method simply moved them.

Dependency Injection allows us to more elegantly solve this problem by only injecting the dependencies we need, when we need them, without the need for any hard coded dependencies at all.

S.O.L.I.D.

Single Responsibility Principle

The Single Responsibility Principle is about actors and high-level architecture. It states that "A class should have only one reason to change." This means that every class should *only* have responsibility over a single part of the functionality provided by the software. The largest benefit of this approach is that it enables improved code *reusability*. By designing our class to do just one thing, we can use (or re-use) it in any other program without changing it.

Open/Closed Principle

The Open/Closed Principle is about class design and feature extensions. It states that "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." This means that we should design our modules, classes and functions in a way that when a new functionality is needed, we should not modify our existing code but rather write new code that will be used by existing code. Practically speaking, this means that we should write classes that implement and adhere to *interfaces*, then type-hint against those interfaces instead of specific classes.

The largest benefit of this approach is that we can very easily extend our code with support for something new without having to modify existing code, meaning that we can reduce QA time, and the risk for negative impact to the application is substantially reduced. We can deploy new code, faster, and with more confidence.

Liskov Substitution Principle

The Liskov Substitution Principle is about subtyping and inheritance. It states that "Child classes should never break the parent class' type definitions." Or, in Robert C. Martin's words, "Subtypes must be substitutable for their base types."

For example, if we have a `FileInterface` interface which defines an `embed()` method, and we have `Audio` and `Video` classes which both implement the `FileInterface` interface, then we can expect that the usage of the `embed()` method will always do the thing that we intend. If we later create a `PDF` class or a `Gist` class which implement the `FileInterface` interface, we will already know and understand what the `embed()` method will do. The largest benefit of this approach is that we have the ability to build flexible and easily-configurable programs, because when we change one object of a type (e.g., `FileInterface`) to another we don't need to change anything else in our program.

Interface Segregation Principle

The Interface Segregation Principle (ISP) is about *business-logic-to-clients* communication. It states that "No client should be forced to depend on methods it does not use." This means that instead of having a single monolithic interface that all conforming classes need to implement, we should instead provide a set of smaller, concept-specific interfaces that a conforming class implements one or more of.

For example, a `Car` or `Bus` class would be interested in a `steeringWheel()` method, but a `Motorcycle` or `Tricycle` class would not. Conversely, a `Motorcycle` or `Tricycle` class would be interested in a `handlebars()` method, but a `Car` or `Bus` class would not. There is no need to have all of these types of vehicles implement support for both `steeringWheel()` as well as `handlebars()`, so we should break-apart the source interface.

Dependency Inversion Principle

The Dependency Inversion Principle is about removing hard-links between discrete classes so that new functionality can be leveraged by passing a different class. It states that one should *"Depend on Abstractions. Do not depend on concretions."*. Put simply, this means our dependencies should be interfaces/contracts or abstract classes rather than concrete implementations. We can easily refactor the above example to follow this principle.

```php
<?php

namespace Database;


class Database

{

    protected $adapter;


    public function __construct(AdapterInterface $adapter)

    {

        $this->adapter = $adapter;

    }

}


interface AdapterInterface {}


class MysqlAdapter implements AdapterInterface {}
```

There are several benefits to the `Database` class now depending on an interface rather than a concretion.

Consider that we are working in a team and the adapter is being worked on by a colleague. In our first example, we would have to wait for said colleague to finish the adapter before we could properly mock it for our unit tests. Now that the dependency is an interface/contract we can happily mock that interface knowing that our colleague will build the adapter based on that contract.

An even bigger benefit to this method is that our code is now much more scalable. If a year down the line we decide that we want to migrate to a different type of database, we can write an adapter that implements the original interface and injects that instead, no more refactoring would be required as we can ensure that the adapter follows the contract set by the interface.

Containers

The first thing you should understand about Dependency Injection Containers is that they are not the same thing as Dependency Injection. A container is a convenience utility that helps us implement Dependency Injection, however, they can be and often are misused to implement an anti-pattern, Service Location. Injecting a DI container as a Service Locator in to your classes arguably creates a harder dependency on the container than the dependency you are replacing. It also makes your code much less transparent and ultimately harder to test.

Most modern frameworks have their own Dependency Injection Container that allows you to wire your dependencies together through configuration. What this means in practice is that you can write application code that is as clean and de- coupled as the framework it is built on.

Further Reading

- What is Dependency Injection?
- Dependency Injection: An analogy
- Dependency Injection: Huh?
- Dependency Injection as a tool for testing

BACK TO TOP

CHAPTER 8.

Databases

Many times your PHP code will use a database to persist information. You have a few options to connect and interact with your database. The recommended option **until PHP 5.1.0** was to use native drivers such as mysqli, pgsql, mssql, etc.

Native drivers are great if you are only using *one* database in your application, but if, for example, you are using MySQL and a little bit of MSSQL, or you need to connect to an Oracle database, then you will not be able to use the same drivers. You'll need to learn a brand new API for each database — and that can get silly.

MySQL Extension

The mysql extension for PHP is incredibly old and has been superseded by two other extensions:

- mysqli
- pdo

Not only did development stop long ago on mysql, but it was deprecated as of PHP 5.5.0, and **has been officially removed in PHP 7.0**.

To save digging into your `php.ini` settings to see which module you are using, one option is to search for `mysql_*` in your editor of choice. If any functions such as `mysql_connect()` and `mysql_query()` show up, then `mysql` is in use.

Even if you are not using PHP 7.x yet, failing to consider this upgrade as soon as possible will lead to greater hardship when the PHP 7.x upgrade does come about. The best option is to replace mysql usage with mysqli or PDO in your applications within your own development schedules so you won't be rushed later on.

If you are upgrading from mysql to mysqli, beware lazy upgrade guides that suggest you can simply find and replace `mysql_*` with `mysqli_*`. Not only is that a gross oversimplification, it misses out on the advantages that mysqli provides, such as parameter binding, which is also offered in PDO.

- MySQLi Prepared Statements
- PHP: Choosing an API for MySQL
- PDO Tutorial for MySQL Developers

PDO Extension

PDO is a database connection abstraction library — built into PHP since 5.1.0 — that provides a common interface to talk with many different databases. For example, you can use basically identical code to interface with MySQL or SQLite:

```
<?php
// PDO + MySQL
$pdo = new PDO('mysql:host=example.com;dbname=database', 'user', 'password');
$statement = $pdo->query("SELECT some_field FROM some_table");
$row = $statement->fetch(PDO::FETCH_ASSOC);
echo htmlentities($row['some_field']);

// PDO + SQLite
$pdo = new PDO('sqlite:/path/db/foo.sqlite');
$statement = $pdo->query("SELECT some_field FROM some_table");
$row = $statement->fetch(PDO::FETCH_ASSOC);
echo htmlentities($row['some_field']);
```

PDO will not translate your SQL queries or emulate missing features; it is purely for connecting to multiple types of database with the same API.

More importantly, `PDO` allows you to safely inject foreign input (e.g. IDs) into your SQL queries without worrying about database SQL injection attacks. This is possible using PDO statements and bound parameters.

Let's assume a PHP script receives a numeric ID as a query parameter. This ID should be used to fetch a user record from a database. This is the `wrong` way to do this:

```
<?php
$pdo = new PDO('sqlite:/path/db/users.db');
$pdo->query("SELECT name FROM users WHERE id = " . $_GET['id']); // <-- NO!
```

This is terrible code. You are inserting a raw query parameter into a SQL query. This will get you hacked in a heartbeat, using a practice called SQL Injection. Just imagine if a hacker passes in an inventive `id` parameter by calling a URL like `http://domain.com/?id=1%3BDELETE+FROM+users`. This will set the `$_GET['id']` variable to `1;DELETE FROM users`

which will delete all of your users! Instead, you should sanitize the ID input using PDO bound parameters.

```
<?php
$pdo = new PDO('sqlite:/path/db/users.db');
$stmt = $pdo->prepare('SELECT name FROM users WHERE id = :id');
$id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT); // <-- filter your data first (see [Data Fi
$stmt->bindParam(':id', $id, PDO::PARAM_INT); // <-- Automatically sanitized for SQL by PDO
$stmt->execute();
```

This is correct code. It uses a bound parameter on a PDO statement. This escapes the foreign input ID before it is introduced to the database preventing potential SQL injection attacks.

For writes, such as INSERT or UPDATE, it's especially critical to still filter your data first and sanitize it for other things (removal of HTML tags, JavaScript, etc). PDO will only sanitize it for SQL, not for your application.

- Learn about PDO

You should also be aware that database connections use up resources and it was not unheard-of to have resources exhausted if connections were not implicitly closed, however this was more common in other languages. Using PDO you can implicitly close the connection by destroying the object by ensuring all remaining references to it are deleted, i.e. set to NULL. If you don't do this explicitly, PHP will automatically close the connection when your script ends - unless of course you are using persistent connections.

- Learn about PDO connections

Interacting with Databases

When developers first start to learn PHP, they often end up mixing their database interaction up with their presentation logic, using code that might look like this:

```
<ul>
<?php
foreach ($db->query('SELECT * FROM table') as $row) {
    echo "<li>".$row['field1']." - ".$row['field1']."</li>";
}
?>
</ul>
```

This is bad practice for all sorts of reasons, mainly that it's hard to debug, hard to test, hard to read and it is going to output a lot of fields if you don't put a limit on there.

While there are many other solutions to doing this - depending on if you prefer OOP or functional programming - there must be some element of separation.

Consider the most basic step:

```php
<?php
function getAllFoos($db) {
    return $db->query('SELECT * FROM table');
}

$results = getAllFoos($db);
foreach ($results as $row) {
    echo "<li>".$row['field1']." - ".$row['field1']."</li>"; // BAD!!
}
```

That is a good start. Put those two items in two different files and you've got some clean separation.

Create a class to place that method in and you have a "Model". Create a simple `.php` file to put the presentation logic in and you have a "View", which is very nearly MVC - a common OOP architecture for most frameworks.

foo.php

```php
<?php
$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8mb4', 'username', 'password');

// Make your model available
include 'models/FooModel.php';

// Create an instance
$fooModel = new FooModel($db);
// Get the list of Foos
$fooList = $fooModel->getAllFoos();

// Show the view
include 'views/foo-list.php';
```

models/FooModel.php

```php
<?php
class FooModel
{
    protected $db;

    public function __construct(PDO $db)
    {
        $this->db = $db;
    }

    public function getAllFoos() {
```

```
        return $this->db->query('SELECT * FROM table');
    }
}
```

views/foo-list.php

```
<?php foreach ($fooList as $row): ?>
    <li><?= $row['field1'] ?> - <?= $row['field1'] ?></li>
<?php endforeach ?>
```

This is essentially the same as what most modern frameworks are doing, albeit a little more manual. You might not need to do all of that every time, but mixing together too much presentation logic and database interaction can be a real problem if you ever want to unit-test your application.

PHPBridge has a great resource called Creating a Data Class which covers a very similar topic, and is great for developers just getting used to the concept of interacting with databases.

Abstraction Layers

Many frameworks provide their own abstraction layer which may or may not sit on top of PDO. These will often emulate features for one database system that is missing from another by wrapping your queries in PHP methods, giving you actual database abstraction instead of just the connection abstraction that PDO provides. This will of course add a little overhead, but if you are building a portable application that needs to work with MySQL, PostgreSQL and SQLite then a little overhead will be worth it for the sake of code cleanliness.

Some abstraction layers have been built using the PSR-0 or PSR-4 namespace standards so can be installed in any application you like:

- Aura SQL
- Doctrine2 DBAL
- Propel
- Zend-db

BACK TO TOP

CHAPTER 9.

Templating

Templates provide a convenient way of separating your controller and domain logic from your presentation logic. Templates typically contain the HTML of your application, but may also be used for other formats, such as XML. Templates are often referred to as “views”, which make up **part of** the second component of the model–view–controller (MVC) software architecture pattern.

Benefits

The main benefit to using templates is the clear separation they create between the presentation logic and the rest of your application. Templates have the sole responsibility of displaying formatted content. They are not responsible for data lookup, persistence or other more complex tasks. This leads to cleaner, more readable code which is especially helpful in a team environment where developers work on the server-side code (controllers, models) and designers work on the client-side code (markup).

Templates also improve the organization of presentation code. Templates are typically placed in a "views" folder, each defined within a single file. This approach encourages code reuse where larger blocks of code are broken into smaller, reusable pieces, often called partials. For example, your site header and footer can each be defined as templates, which are then included before and after each page template.

Finally, depending on the library you use, templates can offer more security by automatically escaping user-generated content. Some libraries even offer sand-boxing, where template designers are only given access to white-listed variables and functions.

Plain PHP Templates

Plain PHP templates are simply templates that use native PHP code. They are a natural choice since PHP is actually a template language itself. That simply means that you can combine PHP code within other code, like HTML. This is beneficial to PHP developers as there is no new syntax to learn, they know the functions available to them, and their code editors already have PHP syntax highlighting and auto-completion built-in. Further, plain PHP templates tend to be very fast as no compiling stage is required.

Every modern PHP framework employs some kind of template system, most of which use plain PHP by default. Outside of frameworks, libraries like Plates or Aura.View make working with plain PHP templates easier by offering modern template functionality such as inheritance, layouts and extensions.

Simple example of a plain PHP template

Using the Plates library.

```
<?php // user_profile.php ?>

<?php $this->insert('header', ['title' => 'User Profile']) ?>

<h1>User Profile</h1>
<p>Hello, <?=$this->escape($name)?></p>

<?php $this->insert('footer') ?>
```

Example of plain PHP templates using inheritance

Using the Plates library.

```
<?php // template.php ?>

<html>
<head>
    <title><?=$title?></title>
</head>
<body>

<main>
    <?=$this->section('content')?>
</main>

</body>
</html>
```

```
<?php // user_profile.php ?>

<?php $this->layout('template', ['title' => 'User Profile']) ?>

<h1>User Profile</h1>
<p>Hello, <?=$this->escape($name)?></p>
```

Compiled Templates

While PHP has evolved into a mature, object oriented language, it hasn't improved much as a templating language. Compiled templates, like Twig, Brainy, or Smarty*, fill this void by offering a new syntax that has been geared specifically to templating. From automatic escaping, to inheritance and simplified control structures, compiled templates are designed to be easier to write, cleaner to read and safer to use. Compiled templates can even be shared across different languages, Mustache being a good example of this. Since these templates must be compiled there is a slight performance hit, however this is very minimal when proper caching is used.

**While Smarty offers automatic escaping, this feature is NOT enabled by default.*

Simple example of a compiled template

Using the Twig library.

```
{% include 'header.html' with {'title': 'User Profile'} %}

<h1>User Profile</h1>
<p>Hello, {{ name }}</p>

{% include 'footer.html' %}
```

Example of compiled templates using inheritance

Using the Twig library.

```
// template.html


<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>


<main>
    {% block content %}{% endblock %}
</main>


</body>
</html>
```

```
// user_profile.html


{% extends "template.html" %}


{% block title %}User Profile{% endblock %}
{% block content %}
    <h1>User Profile</h1>
    <p>Hello, {{ name }}</p>
{% endblock %}
```

Further Reading

Articles & Tutorials

- Templating Engines in PHP
- An Introduction to Views & Templating in CodeIgniter
- Getting Started With PHP Templating
- Roll Your Own Templating System in PHP
- Master Pages
- Working With Templates in Symfony 2
- Writing Safer Templates

Libraries

- Aura.View *(native)*
- Blade *(compiled, framework specific)*
- Brainy *(compiled)*
- Dwoo *(compiled)*

- Latte *(compiled)*
- Mustache *(compiled)*
- PHPTAL *(compiled)*
- Plates *(native)*
- Smarty *(compiled)*
- Twig *(compiled)*
- Zend\View *(native, framework specific)*

BACK TO TOP

CHAPTER 10.

Errors and Exceptions

Errors

In many "exception-heavy" programming languages, whenever anything goes wrong an exception will be thrown. This is certainly a viable way to do things, but PHP is an "exception-light" programming language. While it does have exceptions and more of the core is starting to use them when working with objects, most of PHP itself will try to keep processing regardless of what happens, unless a fatal error occurs.

For example:

```
$ php -a
php > echo $foo;
Notice: Undefined variable: foo in php shell code on line 1
```

This is only a notice error, and PHP will happily carry on. This can be confusing for those coming from "exception-heavy" languages, because referencing a missing variable in Python for example will throw an exception:

```
$ python
>>> print foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

The only real difference is that Python will freak out over any small thing, so that developers can be super sure any potential issue or edge-case is caught, whereas PHP will keep on processing unless something extreme happens, at which point it will throw an error and report it.

Error Severity

PHP has several levels of error severity. The three most common types of messages are errors, notices and warnings. These have different levels of severity; `E_ERROR`, `E_NOTICE`, and `E_WARNING`. Errors are fatal run-time errors and are usually caused by faults in your code and need to be fixed as they'll cause PHP to stop executing. Notices are advisory messages caused by code that may or may not cause problems during the execution of the script, execution is not halted. Warnings are non-fatal errors, execution of the script will not be halted.

Another type of error message reported at compile time are `E_STRICT` messages. These messages are used to suggest changes to your code to help ensure best interoperability and forward compatibility with upcoming versions of PHP.

Changing PHP's Error Reporting Behaviour

Error Reporting can be changed by using PHP settings and/or PHP function calls. Using the built in PHP function `error_reporting()` you can set the level of errors for the duration of the script execution by passing one of the predefined error level constants, meaning if you only want to see Errors and Warnings - but not Notices - then you can configure that:

```
<?php
error_reporting(E_ERROR | E_WARNING);
```

You can also control whether or not errors are displayed to the screen (good for development) or hidden, and logged (good for production). For more information on this check out the Error Reporting section.

Inline Error Suppression

You can also tell PHP to suppress specific errors with the Error Control Operator `@`. You put this operator at the beginning of an expression, and any error that's a direct result of the expression is silenced.

```
<?php
echo @$foo['bar'];
```

This will output `$foo['bar']` if it exists, but will simply return a null and print nothing if the variable `$foo` or `'bar'` key does not exist. Without the error control operator, this expression could create a `PHP Notice: Undefined variable: foo` or `PHP Notice: Undefined index: bar` error.

This might seem like a good idea, but there are a few undesirable tradeoffs. PHP handles expressions using an `@` in a less performant way than expressions without an `@`. Premature optimization may be the root of all programming arguments, but if performance is particularly important for your application/library it's important to understand the error control operator's performance implications.

Secondly, the error control operator **completely** swallows the error. The error is not displayed, and the error is not sent to the error log. Also, stock/production PHP systems have no way to turn off

the error control operator. While you may be correct that the error you're seeing is harmless, a different, less harmless error will be just as silent.

If there's a way to avoid the error suppression operator, you should consider it. For example, our code above could be rewritten like this:

```
<?php
//Null Coalescing Operator
echo $foo['bar'] ?? '';
```

One instance where error suppression might make sense is where `fopen()` fails to find a file to load. You could check for the existence of the file before you try to load it, but if the file is deleted after the check and before the `fopen()` (which might sound impossible, but it can happen) then `fopen()` will return false *and* throw an error. This is potentially something PHP should resolve, but is one case where error suppression might seem like the only valid solution.

Earlier we mentioned there's no way in a stock PHP system to turn off the error control operator. However, Xdebug has an `xdebug.scream` ini setting which will disable the error control operator. You can set this via your `php.ini` file with the following.

```
xdebug.scream = On
```

You can also set this value at runtime with the `ini_set` function

```
<?php
ini_set('xdebug.scream', '1')
```

The "Scream" PHP extension offers similar functionality to Xdebug's, although Scream's ini setting is named `scream.enabled`.

This is most useful when you're debugging code and suspect an informative error is suppressed. Use scream with care, and as a temporary debugging tool. There's lots of PHP library code that may not work with the error control operator disabled.

- Error Control Operators
- SitePoint
- Xdebug
- Scream

ErrorException

PHP is perfectly capable of being an "exception-heavy" programming language, and only requires a few lines of code to make the switch. Basically you can throw your "errors" as "exceptions" using the `ErrorException` class, which extends the `Exception` class.

This is a common practice implemented by a large number of modern frameworks such as Symfony and Laravel. In debug mode *(or dev mode)* both of these frameworks will display a nice and clean *stack trace*.

There are also some packages available for better error and exception handling and reporting. Like Whoops!, which comes with the default installation of Laravel and can be used in any framework as well.

By throwing errors as exceptions in development you can handle them better than the usual result, and if you see an exception during development you can wrap it in a catch statement with specific instructions on how to handle the situation. Each exception you catch instantly makes your application that little bit more robust.

More information on this and details on how to use `ErrorException` with error handling can be found at ErrorException Class.

- Error Control Operators
- Predefined Constants for Error Handling
- `error_reporting()`
- Reporting

Exceptions

Exceptions are a standard part of most popular programming languages, but they are often overlooked by PHP programmers. Languages like Ruby are extremely Exception heavy, so whenever something goes wrong such as a HTTP request failing, or a DB query goes wrong, or even if an image asset could not be found, Ruby (or the gems being used) will throw an exception to the screen meaning you instantly know there is a mistake.

PHP itself is fairly lax with this, and a call to `file_get_contents()` will usually just get you a `FALSE` and a warning. Many older PHP frameworks like CodeIgniter will just return a false, log a message to their proprietary logs and maybe let you use a method like `$this->upload->get_error()` to see what went wrong. The problem here is that you have to go looking for a mistake and check the docs to see what the error method is for this class, instead of having it made extremely obvious.

Another problem is when classes automatically throw an error to the screen and exit the process. When you do this you stop another developer from being able to dynamically handle that error. Exceptions should be thrown to make a developer aware of an error; they then can choose how to handle this. E.g.:

```
<?php
$email = new Fuel\Email;
$email->subject('My Subject');
$email->body('How the heck are you?');
$email->to('guy@example.com', 'Some Guy');

try
{
    $email->send();
}
```

```
catch(Fuel\Email\ValidationFailedException $e)
{
    // The validation failed
}
catch(Fuel\Email\SendingFailedException $e)
{
    // The driver could not send the email
}
finally
{
    // Executed regardless of whether an exception has been thrown, and before normal execution resumes
}
```

SPL Exceptions

The generic `Exception` class provides very little debugging context for the developer; however, to remedy this, it is possible to create a specialized `Exception` type by sub-classing the generic `Exception` class:

```
<?php
class ValidationException extends Exception {}
```

This means you can add multiple catch blocks and handle different Exceptions differently. This can lead to the creation of a *lot* of custom Exceptions, some of which could have been avoided using the SPL Exceptions provided in the SPL extension.

If for example you use the `__call()` Magic Method and an invalid method is requested then instead of throwing a standard Exception which is vague, or creating a custom Exception just for that, you could just `throw new BadMethodCallException;`.

- Read about Exceptions
- Read about SPL Exceptions
- Nesting Exceptions In PHP
- [Exception Best Practices in PHP 5.3][exception-best-practices53]

BACK TO TOP

CHAPTER 11.

Security

The best resource I've found on PHP security is The 2018 Guide to Building Secure PHP Software by Paragon Initiative.

Web Application Security

It is very important for every PHP developer to learn the basics of web application security, which can be broken down into a handful of broad topics:

1. Code-data separation.
   - When data is executed as code, you get SQL Injection, Cross-Site Scripting, Local/Remote File Inclusion, etc.
   - When code is printed as data, you get information leaks (source code disclosure or, in the case of C programs, enough information to bypass ASLR).
2. Application logic.
   - Missing authentication or authorization controls.
   - Input validation.
3. Operating environment.
   - PHP versions.
   - Third party libraries.
   - The operating system.
4. Cryptography weaknesses.
   - Weak random numbers.
   - Chosen-ciphertext attacks.
   - Side-channel information leaks.

There are bad people ready and willing to exploit your web application. It is important that you take necessary precautions to harden your web application's security. Luckily, the fine folks at The Open Web Application Security Project (OWASP) have compiled a comprehensive list of known security issues and methods to protect yourself against them. This is a must read for the security-conscious developer. Survive The Deep End: PHP Security by Padraic Brady is also another good web application security guide for PHP.

- Read the OWASP Security Guide

Password Hashing

Eventually everyone builds a PHP application that relies on user login. Usernames and passwords are stored in a database and later used to authenticate users upon login.

It is important that you properly *hash* passwords before storing them. Hashing and encrypting are two very different things that often get confused.

Hashing is an irreversible, one-way function. This produces a fixed-length string that cannot be feasibly reversed. This means you can compare a hash against another to determine if they both came from the same source string, but you cannot determine the original string. If passwords are not hashed and your database is accessed by an unauthorized third-party, all user accounts are now compromised.

Unlike hashing, encryption is reversible (provided you have the key). Encryption is useful in other areas, but is a poor strategy for securely storing passwords.

Passwords should also be individually *salted* by adding a random string to each password before hashing. This prevents dictionary attacks and the use of "rainbow tables" (a reverse list of crytographic hashes for common passwords.)

Hashing and salting are vital as often users use the same password for multiple services and password quality can be poor.

Additionally, you should use a specialized *password hashing* algorithm rather than fast, general-purpose cryptographic hash function (e.g. SHA256). The short list of acceptable password hashing algorithms (as of June 2018) to use are:

- Argon2 (available in PHP 7.2 and newer)
- Scrypt
- **Bcrypt** (PHP provides this one for you; see below)
- PBKDF2 with HMAC-SHA256 or HMAC-SHA512

Fortunately, nowadays PHP makes this easy.

Hashing passwords with `password_hash`

In PHP 5.5 `password_hash()` was introduced. At this time it is using BCrypt, the strongest algorithm currently supported by PHP. It will be updated in the future to support more algorithms as needed though. The `password_compat` library was created to provide forward compatibility for PHP >= 5.3.7.

Below we hash a string, and then check the hash against a new string. Because our two source strings are different ('secret-password' vs. 'bad-password') this login will fail.

```php
<?php
require 'password.php';

$passwordHash = password_hash('secret-password', PASSWORD_DEFAULT);

if (password_verify('bad-password', $passwordHash)) {
    // Correct Password
} else {
    // Wrong password
}
```

`password_hash()` takes care of password salting for you. The salt is stored, along with the algorithm and "cost", as part of the hash. `password_verify()` extracts this to determine how to check the password, so you don't need a separate database field to store your salts.

- Learn about `password_hash()`
- `password_compat` for PHP >= 5.3.7 && < 5.5
- Learn about hashing in regards to cryptography
- Learn about salts
- PHP `password_hash()` RFC

Data Filtering

Never ever (ever) trust foreign input introduced to your PHP code. Always sanitize and validate foreign input before using it in code. The `filter_var()` and `filter_input()` functions can sanitize text and validate text formats (e.g. email addresses).

Foreign input can be anything: `$_GET` and `$_POST` form input data, some values in the `$_SERVER` superglobal, and the HTTP request body via `fopen('php://input', 'r')`. Remember, foreign input is not limited to form data submitted by the user. Uploaded and downloaded files, session values, cookie data, and data from third-party web services are foreign input, too.

While foreign data can be stored, combined, and accessed later, it is still foreign input. Every time you process, output, concatenate, or include data in your code, ask yourself if the data is filtered properly and can it be trusted.

Data may be *filtered* differently based on its purpose. For example, when unfiltered foreign input is passed into HTML page output, it can execute HTML and JavaScript on your site! This is known as Cross-Site Scripting (XSS) and can be a very dangerous attack. One way to avoid XSS is to sanitize all user-generated data before outputting it to your page by removing HTML tags with the `strip_tags()` function or escaping characters with special meaning into their respective HTML entities with the `htmlentities()` or `htmlspecialchars()` functions.

Another example is passing options to be executed on the command line. This can be extremely dangerous (and is usually a bad idea), but you can use the built-in `escapeshellarg()` function to sanitize the executed command's arguments.

One last example is accepting foreign input to determine a file to load from the filesystem. This can be exploited by changing the filename to a file path. You need to remove `"/"`, `"../"`, null bytes, or other characters from the file path so it can't load hidden, non-public, or sensitive files.

- Learn about data filtering
- Learn about `filter_var`
- Learn about `filter_input`
- Learn about handling null bytes

Sanitization

Sanitization removes (or escapes) illegal or unsafe characters from foreign input.

For example, you should sanitize foreign input before including the input in HTML or inserting it into a raw SQL query. When you use bound parameters with PDO, it will sanitize the input for you.

Sometimes it is required to allow some safe HTML tags in the input when including it in the HTML page. This is very hard to do and many avoid it by using other more restricted formatting like Markdown or BBCode, although whitelisting libraries like HTML Purifier exists for this reason.

See Sanitization Filters

Unserialization

It is dangerous to `unserialize()` data from users or other untrusted sources. Doing so can allow malicious users to instantiate objects (with user-defined properties) whose destructors will be executed, **even if the objects themselves aren't used**. You should therefore avoid unserializing untrusted data.

If you absolutely must unserialize data from untrusted sources, use PHP 7's `allowed_classes` option to restrict which object types are allowed to be unserialized.

Validation

Validation ensures that foreign input is what you expect. For example, you may want to validate an email address, a phone number, or age when processing a registration submission.

See Validation Filters

Configuration Files

When creating configuration files for your applications, best practices recommend that one of the following methods be followed:

- It is recommended that you store your configuration information where it cannot be accessed directly and pulled in via the file system.
- If you must store your configuration files in the document root, name the files with a `.php` extension. This ensures that, even if the script is accessed directly, it will not be output as plain text.
- Information in configuration files should be protected accordingly, either through encryption or group/user file system permissions.
- It is a good idea to ensure that you do not commit configuration files containing sensitive information e.g. passwords or API tokens to source control.

Register Globals

NOTE: As of PHP 5.4.0 the `register_globals` setting has been removed and can no longer be used. This is only included as a warning for anyone in the process of upgrading a legacy application.

When enabled, the `register_globals` configuration setting makes several types of variables (including ones from `$_POST`, `$_GET` and `$_REQUEST`) available in the global scope of your application. This can easily lead to security issues as your application cannot effectively tell where the data is coming from.

For example: `$_GET['foo']` would be available via `$foo`, which can override variables that have been declared.

If you are using PHP < 5.4.0 **make sure** that `register_globals` is **off**.

- Register_globals in the PHP manual

Error Reporting

Error logging can be useful in finding the problem spots in your application, but it can also expose information about the structure of your application to the outside world. To effectively protect your application from issues that could be caused by the output of these messages, you need to configure your server differently in development versus production (live).

Development

To show every possible error during **development**, configure the following settings in your `php.ini`:

```
display_errors = On
display_startup_errors = On
error_reporting = -1
log_errors = On
```

> *Passing in the value `-1` will show every possible error, even when new levels and constants are added in future PHP versions. The `E_ALL` constant also behaves this way as of PHP 5.4. - [php.net](php.net)*

The `E_STRICT` error level constant was introduced in 5.3.0 and is not part of `E_ALL`, however it became part of `E_ALL` in 5.4.0. What does this mean? In terms of reporting every possible error in version 5.3 it means you must use either `-1` or `E_ALL | E_STRICT`.

Reporting every possible error by PHP version

- < 5.3 `-1` or `E_ALL`
- 5.3 `-1` or `E_ALL | E_STRICT`
- > 5.3 `-1` or `E_ALL`

Production

To hide errors on your **production** environment, configure your `php.ini` as:

```
display_errors = Off
display_startup_errors = Off
error_reporting = E_ALL
log_errors = On
```

With these settings in production, errors will still be logged to the error logs for the web server, but will not be shown to the user. For more information on these settings, see the PHP manual:

- error_reporting
- display_errors
- display_startup_errors
- log_errors

CHAPTER 12.

Testing

Writing automated tests for your PHP code is considered a best practice and can lead to well-built applications. Automated tests are a great tool for making sure your application does not break when you are making changes or adding new functionality and should not be ignored.

There are several different types of testing tools (or frameworks) available for PHP, which use different approaches - all of which are trying to avoid manual testing and the need for large Quality Assurance teams, just to make sure recent changes didn't break existing functionality.

Test Driven Development

From Wikipedia:

> *Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.*

There are several different types of testing that you can do for your application:

Unit Testing

Unit Testing is a programming approach to ensure functions, classes and methods are working as expected, from the point you build them all the way through the development cycle. By checking values going in and out of various functions and methods, you can make sure the internal logic is working correctly. By using Dependency Injection and building "mock" classes and stubs you can verify that dependencies are correctly used for even better test coverage.

When you create a class or function you should create a unit test for each behavior it must have. At a very basic level you should make sure it errors if you send it bad arguments and make sure it works if you send it valid arguments. This will help ensure that when you make changes to this class or function later on in the development cycle that the old functionality continues to work as expected. The only alternative to this would be `var_dump()` in a test.php, which is no way to build an application - large or small.

The other use for unit tests is contributing to open source. If you can write a test that shows broken functionality (i.e. fails), then fix it, and show the test passing, patches are much more likely to be

accepted. If you run a project which accepts pull requests then you should suggest this as a requirement.

PHPUnit is the de-facto testing framework for writing unit tests for PHP applications, but there are several alternatives

- atoum
- Kahlan
- Peridot
- SimpleTest

Integration Testing

From Wikipedia:

> *Integration testing (sometimes called Integration and Testing, abbreviated "I&T") is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.*

Many of the same tools that can be used for unit testing can be used for integration testing as many of the same principles are used.

Functional Testing

Sometimes also known as acceptance testing, functional testing consists of using tools to create automated tests that actually use your application instead of just verifying that individual units of code are behaving correctly and that individual units can speak to each other correctly. These tools typically work using real data and simulating actual users of the application.

Functional Testing Tools

- Selenium
- Mink
- Codeception is a full-stack testing framework that includes acceptance testing tools
- Storyplayer is a full-stack testing framework that includes support for creating and destroying test environments on demand

Behavior Driven Development

There are two different types of Behavior-Driven Development (BDD): SpecBDD and StoryBDD. SpecBDD focuses on technical behavior of code, while StoryBDD focuses on business or feature behaviors or interactions. PHP has frameworks for both types of BDD.

With StoryBDD, you write human-readable stories that describe the behavior of your application. These stories can then be run as actual tests against your application. The framework used in PHP

applications for StoryBDD is Behat, which is inspired by Ruby's Cucumber project and implements the Gherkin DSL for describing feature behavior.

With SpecBDD, you write specifications that describe how your actual code should behave. Instead of testing a function or method, you are describing how that function or method should behave. PHP offers the PHPSpec framework for this purpose. This framework is inspired by the RSpec project for Ruby.

BDD Links

- Behat, the StoryBDD framework for PHP, inspired by Ruby's Cucumber project;
- PHPSpec, the SpecBDD framework for PHP, inspired by Ruby's RSpec project;
- Codeception is a full-stack testing framework that uses BDD principles.

Complementary Testing Tools

Besides individual testing and behavior driven frameworks, there are also a number of generic frameworks and helper libraries useful for any preferred approach taken.

Tool Links

- Selenium is a browser automation tool which can be integrated with PHPUnit
- Mockery is a Mock Object Framework which can be integrated with PHPUnit or PHPSpec
- Prophecy is a highly opinionated yet very powerful and flexible PHP object mocking framework. It's integrated with PHPSpec and can be used with PHPUnit.
- php-mock is a library to help to mock PHP native functions.
- Infection is a PHP implementation of Mutation Testing to help to measure the effectiveness of your tests.

BACK TO TOP

CHAPTER 13.

Servers and Deployment

PHP applications can be deployed and run on production web servers in a number of ways.

Platform as a Service (PaaS)

PaaS provides the system and network architecture necessary to run PHP applications on the web. This means little to no configuration for launching PHP applications or PHP frameworks.

Recently PaaS has become a popular method for deploying, hosting, and scaling PHP applications of all sizes. You can find a list of PHP PaaS "Platform as a Service" providers in our resources section.

Virtual or Dedicated Servers

If you are comfortable with systems administration, or are interested in learning it, virtual or dedicated servers give you complete control of your application's production environment.

nginx and PHP-FPM

PHP, via PHP's built-in FastCGI Process Manager (FPM), pairs really nicely with nginx, which is a lightweight, high-performance web server. It uses less memory than Apache and can better handle more concurrent requests. This is especially important on virtual servers that don't have much memory to spare.

- Read more on nginx
- Read more on PHP-FPM
- Read more on setting up nginx and PHP-FPM securely

Apache and PHP

PHP and Apache have a long history together. Apache is wildly configurable and has many available modules to extend functionality. It is a popular choice for shared servers and an easy setup for PHP frameworks and open source apps like WordPress. Unfortunately, Apache uses more resources than nginx by default and cannot handle as many visitors at the same time.

Apache has several possible configurations for running PHP. The most common and easiest to setup is the prefork MPM with mod_php5. While it isn't the most memory efficient, it is the simplest to get working and to use. This is probably the best choice if you don't want to dig too deeply into the server administration aspects. Note that if you use mod_php5 you MUST use the prefork MPM.

Alternatively, if you want to squeeze more performance and stability out of Apache then you can take advantage of the same FPM system as nginx and run the worker MPM or event MPM with mod_fastcgi or mod_fcgid. This configuration will be significantly more memory efficient and much faster but it is more work to set up.

If you are running Apache 2.4 or later, you can use mod_proxy_fcgi to get great performance that is easy to setup.

- Read more on Apache
- Read more on Multi-Processing Modules
- Read more on mod_fastcgi
- Read more on mod_fcgid
- Read more on mod_proxy_fcgi
- Read more on setting up Apache and PHP-FPM with mod_proxy_fcgi

Shared Servers

PHP has shared servers to thank for its popularity. It is hard to find a host without PHP installed, but be sure it's the latest version. Shared servers allow you and other developers to deploy websites to a single machine. The upside to this is that it has become a cheap commodity. The

downside is that you never know what kind of a ruckus your neighboring tenants are going to create; loading down the server or opening up security holes are the main concerns. If your project's budget can afford to avoid shared servers, you should.

To make sure your shared servers are offering the latest versions of PHP, check out PHP Versions.

Building and Deploying your Application

If you find yourself doing manual database schema changes or running your tests manually before updating your files (manually), think twice! With every additional manual task needed to deploy a new version of your app, the chances for potentially fatal mistakes increase. Whether you're dealing with a simple update, a comprehensive build process or even a continuous integration strategy, build automation is your friend.

Among the tasks you might want to automate are:

- Dependency management
- Compilation, minification of your assets
- Running tests
- Creation of documentation
- Packaging
- Deployment

Deployment Tools

Deployment tools can be described as a collection of scripts that handle common tasks of software deployment. The deployment tool is not a part of your software, it acts on your software from 'outside'.

There are many open source tools available to help you with build automation and deployment, some are written in PHP others aren't. This shouldn't hold you back from using them, if they're better suited for the specific job. Here are a few examples:

Phing can control your packaging, deployment or testing process from within a XML build file. Phing (which is based on Apache Ant) provides a rich set of tasks usually needed to install or update a web application and can be extended with additional custom tasks, written in PHP. It's a solid and robust tool and has been around for a long time, however the tool could be perceived as a bit old fashioned because of the way it deals with configuration (XML files).

Capistrano is a system for *intermediate-to-advanced programmers* to execute commands in a structured, repeatable way on one or more remote machines. It is pre-configured for deploying Ruby on Rails applications, however you can successfully deploy PHP systems with it. Successful use of Capistrano depends on a working knowledge of Ruby and Rake.

Ansistrano is a couple of Ansible roles to easily manage the deployment process (deploy and rollback) for scripting applications such as PHP, Python and Ruby. It's an Ansible port for Capistrano. It's been used by quite a lot of PHP companies already.

Rocketeer gets its inspiration and philosophy from the Laravel framework. Its goal is to be fast, elegant and easy to use with smart defaults. It features multiple servers, multiple stages, atomic deploys and deployment can be performed in parallel. Everything in the tool can be hot swapped or extended, and everything is written in PHP.

Deployer is a deployment tool written in PHP. It's simple and functional. Features include running tasks in parallel, atomic deployment and keeping consistency between servers. Recipes of common tasks for Symfony, Laravel, Zend Framework and Yii are available. Younes Rafie's article Easy Deployment of PHP Applications with Deployer is a great tutorial for deploying your application with the tool.

Magallanes is another tool written in PHP with simple configuration done in YAML files. It has support for multiple servers and environments, atomic deployment, and has some built in tasks that you can leverage for common tools and frameworks.

Further reading:

- Automate your project with Apache Ant
- Deploying PHP Applications - paid book on best practices and tools for PHP deployment.

Server Provisioning

Managing and configuring servers can be a daunting task when faced with many servers. There are tools for dealing with this so you can automate your infrastructure to make sure you have the right servers and that they're configured properly. They often integrate with the larger cloud hosting providers (Amazon Web Services, Heroku, DigitalOcean, etc) for managing instances, which makes scaling an application a lot easier.

Ansible is a tool that manages your infrastructure through YAML files. It's simple to get started with and can manage complex and large scale applications. There is an API for managing cloud instances and it can manage them through a dynamic inventory using certain tools.

Puppet is a tool that has its own language and file types for managing servers and configurations. It can be used in a master/client setup or it can be used in a "master-less" mode. In the master/client mode the clients will poll the central master(s) for new configuration on set intervals and update itself if necessary. In the master-less mode you can push changes to your nodes.

Chef is a powerful Ruby based system integration framework that you can build your whole server environment or virtual boxes with. It integrates well with Amazon Web Services through their service called OpsWorks.

Further reading:

- An Ansible Tutorial
- Ansible for DevOps - paid book on everything Ansible
- Ansible for AWS - paid book on integrating Ansible and Amazon Web Services
- Three part blog series about deploying a LAMP application with Chef, Vagrant, and EC2
- Chef Cookbook which installs and configures PHP and the PEAR package management system

- Chef video tutorial series

Continuous Integration

> *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*

– Martin Fowler

There are different ways to implement continuous integration for PHP. Travis CI has done a great job of making continuous integration a reality even for small projects. Travis CI is a hosted continuous integration service for the open source community. It is integrated with GitHub and offers first class support for many languages including PHP.

Further reading:

- Continuous Integration with Jenkins
- Continuous Integration with PHPCI
- Continuous Integration with PHP Censor
- Continuous Integration with Teamcity

BACK TO TOP

CHAPTER 14.

Virtualization

Running your application on different environments in development and production can lead to strange bugs popping up when you go live. It's also tricky to keep different development environments up to date with the same version for all libraries used when working with a team of developers.

If you are developing on Windows and deploying to Linux (or anything non-Windows) or are developing in a team, you should consider using a virtual machine. This sounds tricky, but besides the widely known virtualization environments like VMware or VirtualBox, there are additional tools that may help you setting up a virtual environment in a few easy steps.

Vagrant

Vagrant helps you build your virtual boxes on top of the known virtual environments and will configure these environments based on a single configuration file. These boxes can be set up

manually, or you can use “provisioning” software such as Puppet or Chef to do this for you. Provisioning the base box is a great way to ensure that multiple boxes are set up in an identical fashion and removes the need for you to maintain complicated “set up” command lists. You can also “destroy” your base box and recreate it without many manual steps, making it easy to create a “fresh” installation.

Vagrant creates folders for sharing your code between your host and your virtual machine, which means that you can create and edit your files on your host machine and then run the code inside your virtual machine.

A little help

If you need a little help to start using Vagrant there are some services that might be useful:

- Puphpet: simple GUI to set up virtual machines for PHP development. **Heavily focused in PHP**. Besides local VMs, it can be used to deploy to cloud services as well. The provisioning is made with Puppet.
- Protobox: is a layer on top of vagrant and a web GUI to setup virtual machines for web development. A single YAML document controls everything that is installed on the virtual machine.
- Phansible: provides an easy to use interface that helps you generate Ansible Playbooks for PHP based projects.

Docker

Docker - a lightweight alternative to a full virtual machine - is so called because it’s all about “containers”. A container is a building block which, in the simplest case, does one specific job, e.g. running a web server. An “image” is the package you use to build the container - Docker has a repository full of them.

A typical LAMP application might have three containers: a web server, a PHP-FPM process and MySQL. As with shared folders in Vagrant, you can leave your application files where they are and tell Docker where to find them.

You can generate containers from the command line (see example below) or, for ease of maintenance, build a `docker-compose.yml` file for your project specifying which to create and how they communicate with one another.

Docker may help if you’re developing multiple websites and want the separation that comes from installing each on it’s own virtual machine, but don’t have the necessary disk space or the time to keep everything up to date. It’s efficient: the installation and downloads are quicker, you only need to store one copy of each image however often it’s used, containers need less RAM and share the same OS kernel, so you can have more servers running simultaneously, and it takes a matter of seconds to stop and start them, no need to wait for a full server boot.

Example: Running your PHP Applications in Docker

After installing docker on your machine, you can start a web server with one command. The following will download a fully functional Apache installation with the latest PHP version, map

`/path/to/your/php/files` to the document root, which you can view at `http://localhost:8080`:

```
docker run -d --name my-php-webserver -p 8080:80 -v /path/to/your/php/files:/var/www/html/ php:apache
```

This will initialize and launch your container. `-d` makes it runs in the background. To stop and start it, simply run `docker stop my-php-webserver` and `docker start my-php-webserver` (the other parameters are not needed again).

Learn more about Docker

The command above shows a quick way to run a basic server. There's much more you can do (and thousands of pre-built images in the Docker Hub). Take time to learn the terminology and read the Docker User Guide to get the most from it, and don't run random code you've downloaded without checking it's safe – unofficial images may not have the latest security patches. If in doubt, stick to the official repositiories.

The PHPDocker.io site will auto-generate all the files you need for a fully-featured LAMP/LEMP stack, including your choice of PHP version and extensions.

- Docker Website
- Docker Installation
- Docker User Guide
- Docker Hub
- Docker Hub - official images

BACK TO TOP

CHAPTER 15.

Caching

PHP is pretty quick by itself, but bottlenecks can arise when you make remote connections, load files, etc. Thankfully, there are various tools available to speed up certain parts of your application, or reduce the number of times these various time-consuming tasks need to run.

Opcode Cache

When a PHP file is executed, it must first be compiled into opcodes (machine language instructions for the CPU). If the source code is unchanged, the opcodes will be the same, so this compilation step becomes a waste of CPU resources.

An opcode cache prevents redundant compilation by storing opcodes in memory and reusing them on successive calls. It will typically check signature or modification time of the file first, in case

there have been any changes.

It's likely an opcode cache will make a significant speed improvement to your application. Since PHP 5.5 there is one built in - Zend OPcache. Depending on your PHP package/distribution, it's usually turned on by default - check opcache.enable and the output of `phpinfo()` to make sure. For earlier versions there's a PECL extension.

Read more about opcode caches:

- Zend OPcache (bundled with PHP since 5.5)
- Zend OPcache (formerly known as Zend Optimizer+) is now open source
- APC - PHP 5.4 and earlier
- XCache
- WinCache (extension for MS Windows Server)
- list of PHP accelerators on Wikipedia

Object Caching

There are times when it can be beneficial to cache individual objects in your code, such as with data that is expensive to get or database calls where the result is unlikely to change. You can use object caching software to hold these pieces of data in memory for extremely fast access later on. If you save these items to a data store after you retrieve them, then pull them directly from the cache for following requests, you can gain a significant improvement in performance as well as reduce the load on your database servers.

Many of the popular bytecode caching solutions let you cache custom data as well, so there's even more reason to take advantage of them. APCu, XCache, and WinCache all provide APIs to save data from your PHP code to their memory cache.

The most commonly used memory object caching systems are APCu and memcached. APCu is an excellent choice for object caching, it includes a simple API for adding your own data to its memory cache and is very easy to setup and use. The one real limitation of APCu is that it is tied to the server it's installed on. Memcached on the other hand is installed as a separate service and can be accessed across the network, meaning that you can store objects in a hyper-fast data store in a central location and many different systems can pull from it.

Note that when running PHP as a (Fast-)CGI application inside your webserver, every PHP process will have its own cache, i.e. APCu data is not shared between your worker processes. In these cases, you might want to consider using memcached instead, as it's not tied to the PHP processes.

In a networked configuration APCu will usually outperform memcached in terms of access speed, but memcached will be able to scale up faster and further. If you do not expect to have multiple servers running your application, or do not need the extra features that memcached offers then APCu is probably your best choice for object caching.

Example logic using APCu:

```php
<?php
// check if there is data saved as 'expensive_data' in cache
$data = apc_fetch('expensive_data');
if ($data === false) {
    // data is not in cache; save result of expensive call for later use
    apc_add('expensive_data', $data = get_expensive_data());
}

print_r($data);
```

Note that prior to PHP 5.5, APC provides both an object cache and a bytecode cache. APCu is a project to bring APC's object cache to PHP 5.5+, since PHP now has a built-in bytecode cache (OPcache).

Learn more about popular object caching systems:

- APCu
- APC Functions
- Memcached
- Redis
- XCache APIs
- WinCache Functions

BACK TO TOP

CHAPTER 16.

Documenting your Code

PHPDoc

PHPDoc is an informal standard for commenting PHP code. There are a *lot* of different tags available. The full list of tags and examples can be found at the PHPDoc manual.

Below is an example of how you might document a class with a few methods;

```php
<?php
/**
 * @author A Name <a.name@example.com>
 * @link http://www.phpdoc.org/docs/latest/index.html
 */
class DateTimeHelper
{
    /**
```

```
     * @param mixed $anything Anything that we can convert to a \DateTime object
     *
     * @throws \InvalidArgumentException
     *
     * @return \DateTime
     */
    public function dateTimeFromAnything($anything)
    {
        $type = gettype($anything);

        switch ($type) {
            // Some code that tries to return a \DateTime object
        }

        throw new \InvalidArgumentException(
            "Failed Converting param of type '{$type}' to DateTime object"
        );
    }

    /**
     * @param mixed $date Anything that we can convert to a \DateTime object
     *
     * @return void
     */
    public function printISO8601Date($date)
    {
        echo $this->dateTimeFromAnything($date)->format('c');
    }

    /**
     * @param mixed $date Anything that we can convert to a \DateTime object
     */
    public function printRFC2822Date($date)
    {
        echo $this->dateTimeFromAnything($date)->format('r');
    }
}
```

The documentation for the class as a whole has the @author tag and a @link tag. The @author tag is used to document the author of the code and can be repeated for documenting several authors. The @link tag is used to link to a website indicating a relationship between the website and the code.

Inside the class, the first method has a @param tag documenting the type, name and description of the parameter being passed to the method. Additionally it has the @return and @throws tags for documenting the return type, and any exceptions that could be thrown respectively.

The second and third methods are very similar and have a single @param tag as did the first method. The important difference between the second and third methods' doc block is the inclusion/exclusion of the @return tag. `@return void` explicitly informs us that there is no return; historically omitting the `@return void` statement also results in the same (no return) action.

BACK TO TOP

CHAPTER 17.

Resources

From the Source

- PHP Website
- PHP Documentation

People to Follow

It's difficult to find interesting and knowledgeable PHP community members when you are first starting out. You can find an abbreviated list of PHP community members to get you started at:

- https://www.ogprogrammer.com/2017/06/28/how-to-get-connected-with-the-php-community/
- https://twitter.com/CalEvans/lists/phpeople

Mentoring

- php-mentoring.org - Formal, peer to peer mentoring in the PHP community.

PHP PaaS Providers

- AppFog
- AWS Elastic Beanstalk
- Cloudways
- Divio
- Engine Yard Cloud
- fortrabbit
- Google App Engine
- Heroku
- IBM Cloud
- Jelastic
- Microsoft Azure
- Nanobox
- Pivotal Web Services
- Platform.sh

- Red Hat OpenShift

To see which versions these PaaS hosts are running, head over to PHP Versions.

Frameworks

Rather than re-invent the wheel, many PHP developers use frameworks to build out web applications. Frameworks abstract away many of the low-level concerns and provide helpful, easy-to-use interfaces to complete common tasks.

You do not need to use a framework for every project. Sometimes plain PHP is the right way to go, but if you do need a framework then there are three main types available:

- Micro Frameworks
- Full-Stack Frameworks
- Component Frameworks

Micro-frameworks are essentially a wrapper to route a HTTP request to a callback, controller, method, etc as quickly as possible, and sometimes come with a few extra libraries to assist development such as basic database wrappers and the like. They are prominently used to build remote HTTP services.

Many frameworks add a considerable number of features on top of what is available in a micro-framework; these are called Full-Stack Frameworks. These often come bundled with ORMs, Authentication packages, etc.

Component-based frameworks are collections of specialized and single-purpose libraries. Disparate component-based frameworks can be used together to make a micro- or full-stack framework.

Components

As mentioned above “Components” are another approach to the common goal of creating, distributing and implementing shared code. Various component repositories exist, the main two of which are:

- Packagist
- PEAR

Both of these repositories have command line tools associated with them to help the installation and upgrade processes, and have been explained in more detail in the Dependency Management section.

There are also component-based frameworks and component-vendors that offer no framework at all. These projects provide another source of packages which ideally have little to no dependencies on other packages, or specific frameworks.

For example, you can use the FuelPHP Validation package, without needing to use the FuelPHP framework itself.

- Aura
- CakePHP Components
  - Collection
  - Database
  - Datasource
  - Event
  - I18n
  - ORM
- FuelPHP
- Hoa Project
- Symfony Components
- The League of Extraordinary Packages
- Laravel's Illuminate Components
  - IoC Container
  - Eloquent ORM
  - Queue

Laravel's Illuminate components will become better decoupled from the Laravel framework. For now, only the components best decoupled from the Laravel framework are listed above.

Other Useful Resources

Cheatsheets

- PHP Cheatsheets - for variable comparisons, arithmetics and variable testing in various PHP versions
- OWASP Security Cheatsheets - provides a concise collection of high value information on specific application security topics.

More best practices

- PHP Best Practices
- Best practices for Modern PHP Development
- Why You Should Be Using Supported PHP Versions

News around the PHP and web development communities

You can subscribe to weekly newsletters to keep yourself informed on new libraries, latest news, events and general announcements, as well as additional resources being published every now and then:

- PHP Weekly
- JavaScript Weekly
- Frontend Focus
- Mobile Web Weekly

There are also Weeklies on other platforms you might be interested in; here's a list of some.

PHP universe

- PHP Developer blog

Video Tutorials

YouTube Channels

- PHP Academy
- The New Boston
- Sherif Ramadan
- Level Up Tuts

Paid Videos

- Standards and Best practices
- PHP Training on Pluralsight
- PHP Training on Lynda.com
- PHP Training on Tutsplus
- Laracasts

Books

There are many PHP books; sadly some are now quite old and no longer accurate. In particular, avoid books on "PHP 6", a version that will now never exist. The next major release of PHP after 5.6 was "PHP 7", partly because of this.

This section aims to be a living document for recommended books on PHP development in general. If you would like your book to be added, send a PR and it will be reviewed for relevancy.

Free Books

- PHP Pandas - Aims to teach everyone how to be a web developer.
- PHP The Right Way - This website is available as a book completely for free.
- Using Libsodium in PHP Projects - Guide to using Libsodium PHP extension for modern, secure, and fast cryptography.

Paid Books

- Build APIs You Won't Hate - Everyone and their dog wants an API, so you should probably learn how to build them.
- Modern PHP - covers modern PHP features, best practices, testing, tuning, deployment and setting up a dev environment.
- Building Secure PHP Apps - Learn the security basics that a senior developer usually acquires over years of experience, all condensed down into one quick and easy handbook
- Modernizing Legacy Applications In PHP - Get your code under control in a series of small, specific steps
- Securing PHP: Core Concepts - A guide to some of the most common security terms and provides some examples of them in every day PHP
- Scaling PHP - Stop playing sysadmin and get back to coding

- Signaling PHP - PCNLT signals are a great help when writing PHP scripts that run from the command line.
- Minimum Viable Tests - Long-time PHP testing evangelist Chris Hartjes goes over what he feels is the minimum you need to know to get started.
- Domain-Driven Design in PHP - See real examples written in PHP showcasing Domain-Driven Design Architectural Styles (Hexagonal Architecture, CQRS or Event Sourcing), Tactical Design Patterns, and Bounded Context Integration.

BACK TO TOP

CHAPTER 18.

Community

The PHP community is as diverse as it is large, and its members are ready and willing to support new PHP programmers. Consider joining your local PHP user group (PUG) or attending larger PHP conferences to learn more about the best practices shown here. You can hang out on IRC in the #phpc channel on irc.freenode.com and follow the @phpc twitter account. Get out there, meet new developers, learn new topics, and above all, make new friends! Other community resources include StackOverflow.

Read the Official PHP Events Calendar

PHP User Groups

If you live in a larger city, odds are there's a PHP user group nearby. You can easily find your local PUG at PHP.ug. Alternate sources might be Meetup.com or a search for `php user group near me` using your favorite search engine (i.e. Google). If you live in a smaller town, there may not be a local PUG; if that's the case, start one!

Special mention should be made of two global user groups: NomadPHP and PHPWomen. NomadPHP offers twice monthly online user group meetings with presentations by some of the top speakers in the PHP community. PHPWomen is a non-exclusive user group originally targeted towards the women in the PHP world. Membership is open to everyone who supports a more diverse community. PHPWomen provide a network for support, mentorship and education, and generally promote the creating of a "female friendly" and professional atmosphere.

Read about User Groups on the PHP Wiki

PHP Conferences

The PHP community also hosts larger regional and national conferences in many countries around the world. Well-known members of the PHP community usually speak at these larger events, so it's a great opportunity to learn directly from industry leaders.

Find a PHP Conference

ElePHPants

ElePHPant is that beautiful mascot of the PHP project with elephant in their design. It was originally designed for the PHP project in 1998 by Vincent Pontier - spiritual father of thousands of elePHPants around the world and 10 years later adorable plush elephant toy came to birth as well. Now elePHPants are present at many PHP conferences and with many PHP developers at their computers for fun and inspiration.

Interview with Vincent Pontier

Created and maintained by

Josh Lockhart
Phil Sturgeon
Project Contributors