# PHP Best Practices A short, practical guide for common and confusing PHP tasks



## Last revised & maintainers

This document was last reviewed on July 3, 2014. It was last changed on July 3, 2014.

It's maintained by me, Alex Cabal. I've been writing PHP for a long time now, and currently I run Scribophile, an online writing group for serious writers, Writerfolio, an easy online writing portfolio for freelancers, and Standard Ebooks, a publisher of illustrated, DRM-free public domain ebooks. On occasion I freelance for projects and clients that interest me.

Drop me a line if you think I can help you with something, or with suggestions or corrections to this document.

## Introduction

PHP is a complex language that has suffered years of twists, bends, stretches, and hacks. It's highly inconsistent and sometimes buggy. Each version has its own unique features, warts, and quirks, and it's hard to keep track of what version has what problems. It's easy to see why it gets as much hate as it does sometimes.

Despite that, it's the most popular language on the web today. Because of its long history, you'll find lots of tutorials on how to do basic things like password hashing and database access. The problem is that out of five tutorials, you have a good chance of finding five totally different ways of doing something. Which way is the "right" way? Do any of the other ways have subtle bugs or gotchas? It's really hard to find out, and you'll be bouncing around the internet trying to pin down the right answer.

That's also one of the reasons why new PHP programmers are so frequently blamed for ugly, outdated, or insecure code. They can't help it if the first Google result was a four-year-old article teaching a five-year-old method!

This document tries to address that. It's an attempt to compile a set of basic instructions for what can be considered best practices for common and confusing issues and tasks in PHP. If a low-level task has multiple and confusing approaches in PHP, it belongs here.

**What this is**

It's a guide suggesting the best direction to take when facing one of the common *low-level* tasks a PHP programmer might encounter that are unclear because of the many options PHP might offer. For example: connecting to a database is a common task with a large amount of possible solutions in PHP, not all of them good ones—thus, it's included in this document.

It's a series of short, introductory solutions. Examples should get you up and running in a basic setting, and you should do your own research to flesh them out into something useful to you.

It points to what we consider the state-of-the-art of PHP. However, this means that if you're using an older version of PHP, some of the features required to pull off these solutions might not be available to you.

This is a living document that I'll do my best to keep updated as PHP continues to evolve.

**What this *isn't***

This document is not a PHP tutorial. You should learn the basics and syntax of the language elsewhere.

It's not a guide to common web application problems like cookie storage, caching, coding style, documentation, and so on.

It's not a security guide. While it touches upon some security-related issues, you're expected to do your own research when it comes to securing your PHP apps. In particular, you should carefully review any solution proposed here before implementing it. Your code is your own fault.

It's not an advocate of a certain coding style, pattern, or framework.

It's not an advocate for a certain way of doing high-level tasks like user registration, login systems, etc. This document is strictly for *low-level* tasks that, because of PHP's long history, might be confusing or unclear.

It's not a be-all and end-all solution, nor is it the *only* solution. Some of the methods described below might not be what's best for your particular situation, and there are lots of different ways of achieving the same ends. In particular, high-load web apps might benefit from more esoteric solutions to some of these problems.

# What PHP version are we using?

## PHP 5.5.9-1ubuntu4.2, installed on Ubuntu 14.04 LTS.

PHP is the 100-year-old tortoise of the web world. Its shell is inscribed with a rich, convoluted, and gnarled history. In a shared-hosting environment, its configuration might restrict what you can do.

In order to retain a scrap of sanity, we're going to focus on just one version of PHP. As of April 30, 2013, that version is **PHP 5.5.9-1ubuntu4.2**. This is the version of PHP you'll get if you install it using apt-get on an **Ubuntu 14.04 LTS** server. In other words, it's the sane default used by many.

You might find that some of these solutions work on different or older versions of PHP. If that's the case, *it's up to you to research the implications of subtle bugs or security issues in these older versions*.

### A note on Ubuntu 12.04 LTS

A lot has changed in the PHP world in the years between Ubuntu 12.04 and 14.04. Since 12.04 is still very common in the wild, and since some of the solutions below have changed significantly in 14.04, I'll include 12.04-specific solutions in a box like this one.

The version of PHP installed using apt-get on Ubuntu 12.04 is **PHP 5.3.10-1ubuntu3.6 with Suhosin-Patch**.

# Storing passwords

## Use the built-in password hashing functions to hash and compare passwords.

Hashing is the standard way of protecting a user's password before it's stored in a database. Many common hashing algorithms like md5 and even sha1 are unsafe for storing passwords, because hackers can easily crack passwords hashed using those algorithms.

PHP provides a built-in password hashing library that uses the bcrypt algorithm, currently considered the best algorithm for password hashing.

#### Example

?

```
<?php
// Hash the password.  $hashedPassword will be a 60-character string.
$hashedPassword = password_hash('my super cool password', PASSWORD_DEFAULT);

// You can now safely store the contents of $hashedPassword in your database!

// Check if a user has provided the correct password by comparing what they typed with
our hash
password_verify('the wrong password', $hashedPassword); // false

password_verify('my super cool password', $hashedPassword); // true
?>
```

#### Gotchas

- Many sources will recommend that you also "salt" your password before hashing it. That's a great idea, and password_hash() *already salts your password for you*. That means that you don't have to salt it yourself.

#### Further reading

- Why hashing passwords with md5 or sha is unsafe
- How to safely store a password

### Ubuntu 12.04 (PHP <= 5.3.10)
### Use the phpass library to hash and compare passwords.

*Tested with phpass 0.3.*

The built-in PHP password hashing library isn't available in the version of PHP installed in 12.04. Instead, use the open-source phpass library, which provides the same bcrypt-based functionality in an easy-to-use class.

#### Example

?

```
<?php
// Include the phpass library
require_once('phpass-0.3/PasswordHash.php');

// Initialize the hasher without portable hashes (this is more secure)
$hasher = new PasswordHash(8, false);

// Hash the password.  $hashedPassword will be a 60-character string.
$hashedPassword = $hasher->HashPassword('my super cool password');

// You can now safely store the contents of $hashedPassword in your database!

// Check if a user has provided the correct password by comparing what they typed with
our hash
$hasher->CheckPassword('the wrong password', $hashedPassword); // false

$hasher->CheckPassword('my super cool password', $hashedPassword); // true
?>
```

#### Further reading

- phpass

#### Gotchas

- Just like password_hash() in new versions of PHP, phpass automatically salts your password for you.

# Connecting to and querying a MySQL database

### Use PDO and its prepared statement functionality.

There are many ways to connect to a MySQL database in PHP. PDO (PHP Data Objects) is the newest and most robust of them. PDO has a consistent interface across many different types of database, uses an object-oriented approach, and supports more features offered by newer databases.

You should use PDO's prepared statement functions to help prevent SQL injection attacks. Using the

bindValue() function ensures that your SQL is safe from first-order SQL injection attacks. (This isn't 100% foolproof though, see Further Reading for more details.) In the past, this had to be achieved with some arcane combination of "magic quote" functions. PDO makes all that gunk unnecessary.

### Example

?

```
<?php
try{
    // Create a new connection.
    // You'll probably want to replace hostname with localhost in the first parameter.
    // Note how we declare the charset to be utf8mb4.  This alerts the connection that
we'll be passing UTF-8 data.  This may not be required depending on your configuration,
but it'll save you headaches down the road if you're trying to store Unicode strings in
your database.  See "Gotchas".
    // The PDO options we pass do the following:
    // \PDO::ATTR_ERRMODE enables exceptions for errors.  This is optional but can be
handy.
    // \PDO::ATTR_PERSISTENT disables persistent connections, which can cause concurrency
issues in certain cases.  See "Gotchas".
    $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
                        )
                    );

    $handle = $link->prepare('select Username from Users where UserId = ? or Username = ?
limit ?');

    // PHP bug: if you don't specify PDO::PARAM_INT, PDO may enclose the argument in
quotes.  This can mess up some MySQL queries that don't expect integers to be quoted.
    // See: https://bugs.php.net/bug.php?id=44639
    // If you're not sure whether the value you're passing is an integer, use the
is_int() function.
    $handle->bindValue(1, 100, PDO::PARAM_INT);
    $handle->bindValue(2, 'Bilbo Baggins');
    $handle->bindValue(3, 5, PDO::PARAM_INT);

    $handle->execute();

    // Using the fetchAll() method might be too resource-heavy if you're selecting a
truly massive amount of rows.
    // If that's the case, you can use the fetch() method and loop through each result
row one by one.
    // You can also return arrays and other things instead of objects.  See the PDO
documentation for details.
    $result = $handle->fetchAll(\PDO::FETCH_OBJ);

    foreach($result as $row){
        print($row->Username);
    }
```

```
}
catch(\PDOException $ex){
    print($ex->getMessage());
}
?>
```

### Gotchas

- Not passing the PDO::PARAM_INT parameter when binding integer variables can sometimes cause PDO to quote them. This can screw up certain MySQL queries. See this bug report.
- Not having set the character set to utf8mb4 in the connection string might cause Unicode data to be stored incorrectly in your database, depending on your configuration.
- Even if you declare your character set to be utf8mb4, make sure that your actual database tables are in the utf8mb4 character set. For why we use utf8mb4 instead of just utf8, check the PHP and UTF-8 section.
- Enabling persistent connections can possibly lead to weird concurrency-related issues. This isn't a PHP problem, it's an app-level problem. Persistent connections are safe to use as long as you consider the consequences. See this Stack Overflow question.
- You can execute more than one SQL statement in a single execute() call. Just separate the statements with semicolons. **Ubuntu 12.04 (PHP <= 5.3.10):** If doing this, beware of this bug, which isn't fixed in the version of PHP shipped with Ubuntu 12.04.

### Further reading

- PHP Manual: PDO
- Why you should be using PHP's PDO for database access
- Stack Overflow: PHP PDO vs normal mysql_connect
- Stack Overflow: Are PDO prepared statements sufficient to prevent SQL injection?
- Stack Overflow: Whether to use "SET NAMES"

# PHP tags

## Use <?php ?>.

There are a few different ways to delimit blocks of PHP: <?php ?>, <?= ?>, <? ?>, and <% %>. While the shorter ones might be more convenient to type, the only one that's guaranteed to work on all PHP servers is <?php ?>. If you ever plan on deploying your PHP to a server whose configuration you can't control, then you should always use <?php ?>.

If you're only coding for yourself and have control over the PHP configuration you'll be using, you might find the shorter tags to be more convenient. But remember that <? ?> might conflict with XML declarations and <% %> is actually ASP style.

Whatever you choose, make sure you stay consistent!

### Gotchas

- When including a closing ?> tag in a pure PHP file (for example, in a file that only contains a class definition), *make sure not to leave any trailing newlines after it*. While the PHP parser safely "eats" a single newline character after the closing tag, any other newlines might be outputted to the browser and possibly confuse things if you're outputting any HTTP headers later.

- When writing a web app targeting older versions of IE, make sure not to leave a newline between any closing ?> tag and the html <!doctype> tag. Old versions of IE will enter quirks mode if they encounter any white space, including newlines, before the doctype declaration. This isn't an issue for newer versions of IE and other, more advanced browsers. (Read: every other browser besides IE.)

### Further reading

- Stack Overflow: Are PHP short tags acceptable to use?

# Auto-loading classes

## Use spl_autoload_register() to register your auto-load function.

PHP provides several ways to auto-load files containing classes that haven't yet been loaded. The older way is to use a magic global function called __autoload(). However you can only have one __autoload() function defined at once, so if you're including a library that *also* uses the __autoload() function, then you'll have a conflict.

The correct way to handle this is to name your autoload function something unique, then register it with the spl_autoload_register() function. This function allows more than one __autoload() function to be defined, so you won't step on any other code's own __autoload() function.

### Example

?

```
<?php
// First, define your auto-load function.
function MyAutoload($className){
    include_once($className . '.php');
}

// Next, register it with PHP.
spl_autoload_register('MyAutoload');

// Try it out!
// Since we haven't included a file defining the MyClass object, our auto-loader will
kick in and include MyClass.php.
// For this example, assume the MyClass class is defined in the MyClass.php file.
$var = new MyClass();
?>
```

### Further reading

- PHP Manual: spl_autoload_register()
- Stack Overflow: Efficient PHP auto-loading and naming strategies

# Single vs. double quotes from a performance perspective

## It doesn't really matter.

A lot of ink has been spilled about whether to define strings with single quotes (') or double quotes ("). Single-quoted strings aren't parsed, so whatever you've put in the string, that's what will show up. Double-quoted strings are parsed and any PHP variables in the string are evaluated. Additionally, escaped characters like \n for newline and \t for tab *are not* evaluated in single-quoted strings, but *are* evaluated in double-quoted strings.

Because double-quoted strings are evaluated at run time, the theory is that using single-quoted strings will improve performance because PHP won't have to evaluate every single string. While this might be true on a certain scale, for the average real-life application the difference is so small that it doesn't really matter. So for an average app, it doesn't matter what you choose. For *extremely* high-load apps, it might matter a little. Make a choice depending on what your app needs, but whatever you choose, be consistent.

### Further reading

- PHP Manual: Strings
- The PHP Benchmark (Scroll down to Quote Types)
- Stack Overflow: Is there a performance benefit to single quotes vs double quotes in PHP?

# define() vs. const

## Use define() unless readability, class constants, or micro-optimization are concerns.

Traditionally in PHP you would define constants using the define() function. But at some point PHP gained the ability to also declare constants with the const keyword. Which one should you use when defining your constants?

The answer lies in the little differences between the two methods.

1. define() defines constants at run time, while const defines constants at compile time. This gives const a very slight speed edge, but not one worth worrying about unless you're building large-scale software.
2. define() puts constants in the global scope, although you can include namespaces in your constant name. That means you can't use define() to define class constants.
3. define() lets you use expressions both in the constant name and in the constant value, unlike const which allows neither. This makes define() much more flexible.
4. define() can be called within an if() block, while const cannot.

### Example

?

```
<?php
// Let's see how the two methods treat namespaces
namespace MiddleEarth\Creatures\Dwarves;
const GIMLI_ID = 1;
define('MiddleEarth\Creatures\Elves\LEGOLAS_ID', 2);

echo(\MiddleEarth\Creatures\Dwarves\GIMLI_ID);  // 1
echo(\MiddleEarth\Creatures\Elves\LEGOLAS_ID);  // 2; note that we used define(), but the
namespace is still recognized

```

```
// Now let's declare some bit-shifted constants representing ways to enter Mordor.
define('TRANSPORT_METHOD_SNEAKING', 1 << 0); // OK!
const TRANSPORT_METHOD_WALKING = 1 << 1; // Compile error! const can't use expressions as
values

// Next, conditional constants.
define('HOBBITS_FRODO_ID', 1);

if($isGoingToMordor){
    define('TRANSPORT_METHOD', TRANSPORT_METHOD_SNEAKING); // OK!
    const PARTY_LEADER_ID = HOBBITS_FRODO_ID // Compile error: const can't be used in an
if block
}

// Finally, class constants
class OneRing{
    const MELTING_POINT_CELSIUS = 1000000; // OK!
    define('MELTING_POINT_ELVISH_DEGREES', 200); // Compile error: can't use define()
within a class
  }
  ?>
```

Because define() is ultimately more flexible, it's the one you should use to avoid headaches unless you specifically require class constants. Using const generally results in more readable code, but at the expense of flexibility.

Whichever one you use, be consistent!

### Further reading

- Stack Overflow: define() vs const
- PHP Manual: Constants
- Stack Overflow: define() vs variable

# Caching PHP opcode

## Lucky you: PHP has a built-in opcode cache!

In older versions of PHP, every time a script was executed it would have to be compiled from scratch, even if it had been compiled before. Opcode caches were additional software that saved previously compiled versions of PHP, speeding things up a bit. There were various flavors of caches you could choose from.

Lucky for us, the version of PHP that ships with Ubuntu 14.04 *includes a built-in opcode cache* that's turned on by default. So there's nothing for you to do!

For posterity, below are instructions for using an opcode cache in Ubuntu 12.04, which doesn't include its own.

### Further reading

- PHP Manual: Opcache

### Ubuntu 12.04 (PHP <= 5.3.10)
### Use APC as an opcode cache.

There are several PHP opcode caches available, notably eaccelerator, xcache, and APC. APC is officially supported by the PHP project, is the most active, and is the easiest to install. It also provides an optional memcached-like persistent key-value store. For these reasons, it's the one you should be using.

#### Installing APC

You can install APC on Ubuntu 12.04 by running this command in your terminal:

```
sudo apt-get install php-apc
```

No further configuration is necessary.

#### Gotchas

- If you're not using PHP-FPM (for example you're using mod_php or mod_fastcgi), each PHP process will have its unique APC instance, including key-value store. This can lead to synchronization problems in your app code if you're not careful.

#### Further reading

- PHP Manual: APC

## PHP and Memcached

### If you need a distributed cache, use the Memcached client library. Otherwise, use APCu.

A caching system can often improve your app's performance. Memcached is a popular choice and it works with many languages, including PHP.

However, when it comes to accessing a Memcached server from a PHP script, you have two different and very stupidly named choices of client library: Memcache and Memcached. They're different libraries with almost the same name, and both are used to access a Memcached instance.

It turns out that the Memcached library is the one that best implements the Memcached protocol. It includes a few useful features that the Memcache library doesn't, and seems to be the one most actively developed.

However if you don't need to access a Memcached instance from a series of distributed servers, then use APCu instead. APCu is supported by the PHP project and has much of the same functionality as Memcached.

#### Installing the Memached client library

After you install the Memcached server, you need to install the Memcached client library. Without the library, your PHP scripts won't be able to communicate with the Memcached server.

You can install the Memcached client library on Ubuntu 14.04 by running this command in your terminal:

```
sudo apt-get install php5-memcached
```

### Using APCu instead

Before Ubuntu 14.04, the APC project was both an opcode cache *and* a Memcached-like key-value store. Since the version of PHP that ships with 14.04 now includes a built-in opcode cache, APC was split into the APCu project, which is essentially APC's key-value storage functionality—AKA the "user cache", or the "u" in APCu—without the opcode-cache parts.

### Installing APCu

You can install APCu on Ubuntu 14.04 by running this command in your terminal:

```
sudo apt-get install php5-apcu
```

### Example

?

```
<?php
// Store some values in the APCu cache.  We can optionally pass a time-to-live, but in
this example the values will live forever until they're garbage-collected by APCu.
apc_store('username-1532', 'Frodo Baggins');
apc_store('username-958', 'Aragorn');
apc_store('username-6389', 'Gandalf');

// After storing these values, any PHP script can access them, no matter when it's run!
$value = apc_fetch('username-958', $success);
if($success === true)
    print($value); // Aragorn

$value = apc_fetch('username-1', $success); // $success will be set to boolean false,
because this key doesn't exist.
if($success !== true) // Note the !==, this checks for true boolean false, not "falsey"
values like 0 or empty string.
    print('Key not found');

apc_delete('username-958'); // This key will no longer be available.
?>
```

### Further reading

- PHP Manual: Memcached
- PECL: APCu
- Stack Overflow: Using Memcache vs Memcached with PHP
- Stack Overflow: Memcached vs APC, which one should I choose?

## Ubuntu 12.04 (PHP <= 5.3.10)
## Using APC instead of Memcached

If you're using APC as both a key-value store and an opcode cache in Ubuntu 12.04, then the example above will also work—APC and APCu have an identical API.

See the section on opcode caching for instructions on how to install APC on Ubuntu 12.04.

**Further reading**

- PHP Manual: APC

# PHP and regex

## Use the PCRE (preg_*) family of functions.

PHP has two different ways of using regular expressions: the PCRE (Perl-compatible, preg_*) functions and the POSIX (POSIX extended, ereg_*) functions.

Each family of functions uses a slightly different flavor of regular expression. Luckily for us, the POSIX functions have been deprecated since PHP 5.3.0. Because of this, you should never write new code using the POSIX functions. Always use the PCRE functions, which are the preg_* functions.

**Further Reading**

- PHP Manual: PCRE
- Getting started with PHP regular expressions

# Serving PHP from a web server

## Use PHP-FPM.

There are several ways of configuring a web server to serve PHP. The traditional (and terrible) way is to use Apache's mod_php. Mod_php attaches PHP to Apache itself, but Apache does a very bad job of managing it. You'll suffer from severe memory problems as soon as you get any kind of real traffic.

Two new options soon became popular: mod_fastcgi and mod_fcgid. Both of these keep a limited number of PHP processes running, and Apache sends requests to these interfaces to handle PHP execution on its behalf. Because these libraries limit how many PHP processes are alive, memory usage is greatly reduced without affecting performance.

Some smart people created an implementation of fastcgi that was specially designed to work really well with PHP, and they called it PHP-FPM. This was the standard solution for web servers since 12.04.

In the years since Ubuntu 12.04, Apache introduced a new method of interacting with PHP-FPM: mod_proxy_fcgi. Unfortunately, the version of Apache that ships with Ubuntu 14.04 has a few problems with this module, including not being able to use socket-based connections, issues with mod_rewrite, and issues with 404 and similar pages. Because of these problems you should stick with the tried-and-true method of working with PHP-FPM that we used in Ubuntu 12.04.

The following example is for Apache 2.4.7, but PHP-FPM also works for other web servers like Nginx.

**Installing PHP-FPM and Apache**

You can install PHP-FPM and Apache on Ubuntu 14.04 by running these command in your terminal:

```
sudo apt-get install apache2-mpm-event libapache2-mod-fastcgi php5-fpm sudo a2enmod actions alias fastcgi
```

Note that we *must* use apache2-mpm-event (or apache2-mpm-worker), *not* apache2-mpm-prefork or apache2-mpm-threaded.

Next, we'll configure our Apache virtualhost to route PHP requests to the PHP-FPM process. Place the following in your Apache configuration file (in Ubuntu 14.04 the default one is /etc/apache2/sites-available/000-default.conf).

```
<Directory />
    Require all granted
</Directory>
<VirtualHost *:80>
    Action php5-fcgi /php5-fcgi
    Alias /php5-fcgi /usr/lib/cgi-bin/php5-fcgi
    FastCgiExternalServer /usr/lib/cgi-bin/php5-fcgi -socket /var/run/php5-fpm.sock -idle-timeout 120 -pass-header Authorization
    <FilesMatch "\.php$">
        SetHandler  php5-fcgi
    </FilesMatch>
</VirtualHost>
```

Finally, restart Apache and the FPM process:

```
sudo service apache2 restart && sudo service php5-fpm restart
```

### Gotchas

- Using the AddHandler directive instead of the SetHandler directive can be a security risk. AddHandler will execute PHP in files that have ".php" *anywhere in the file name*. So if a user uploads evil.php.gif using your file upload form, you might be in for a nasty surprise.

### Further reading

- PHP Manual: PHP-FPM
- PHP-FPM homepage
- Installing Apache + mod_fastcgi + PHP-FPM on Ubuntu Server Maverick
- Why mod_php is bad for performance

## Ubuntu 12.04 (Apache 2.2.22)

The configuration for Ubuntu 12.04 is just as above, except don't include the <Directory> block in the configuration file.

The default site in Apache 2.2.22 is located at /etc/apache2/sites-available/default.

Apache 2.2.22 does not have a stable event MPM, so install apache2-mpm-worker instead of apache2-mpm-event.

# Sending email

## Use PHPMailer.

*Tested with PHPMailer 5.2.7.*

PHP provides a mail() function that looks enticingly simple and easy. Unfortunately, like a lot of things in PHP, its simplicity is deceptive and using it at face value can lead to serious security problems.

Email is a set of protocols with an even more tortured history than PHP. Suffice it to say that there are so many gotchas in sending email that just being in the same room as PHP's mail() function should give you the shivers.

PHPMailer is a popular and well-aged open-source library that provides an easy interface for sending mail securely. It takes care of the gotchas for you so you can concentrate on more important things.

### Example

?

```
<?php
// Include the PHPMailer library
require_once('phpmailer-5.2.7/PHPMailerAutoload.php');

// Passing 'true' enables exceptions.  This is optional and defaults to false.
$mailer = new PHPMailer(true);

// Send a mail from Bilbo Baggins to Gandalf the Grey

// Set up to, from, and the message body.  The body doesn't have to be HTML; check the
PHPMailer documentation for details.
$mailer->Sender = 'bbaggins@example.com';
$mailer->AddReplyTo('bbaggins@example.com', 'Bilbo Baggins');
$mailer->SetFrom('bbaggins@example.com', 'Bilbo Baggins');
$mailer->AddAddress('gandalf@example.com');
$mailer->Subject = 'The finest weed in the South Farthing';
$mailer->MsgHTML('<p>You really must try it, Gandalf!</p><p>-Bilbo</p>');

// Set up our connection information.
$mailer->IsSMTP();
$mailer->SMTPAuth = true;
$mailer->SMTPSecure = 'ssl';
$mailer->Port = 465;
$mailer->Host = 'my smtp host';
$mailer->Username = 'my smtp username';
$mailer->Password = 'my smtp password';

// All done!
$mailer->Send();
?>
```

### Further reading

- PHPMailer at Github

# Validating email addresses

## Use the filter_var() function.

A common task your web app might need to do is to check if a user has entered a valid email address. You'll no doubt find online a dizzying range of complex regular expressions that all claim to solve this problem, but the easiest way is to use PHP's built-in filter_var() function, which can validate email addresses.

### Example

```
<?php
filter_var('sgamgee@example.com', FILTER_VALIDATE_EMAIL); // Returns
"sgamgee@example.com". This is a valid email address.
filter_var('sauron@mordor', FILTER_VALIDATE_EMAIL); // Returns boolean false! This is
*not* a valid email address.
?>
```

### Further reading

- PHP Manual: filter_var()
- PHP Manual: Types of filters

# Sanitizing HTML input and output

## Use the htmlentities() function for simple sanitization and the HTML Purifier library for complex sanitization.

*Tested with HTML Purifier 4.6.0.*

When displaying user input in any web application, it's essential to "sanitize" it first to remove any potentially dangerous HTML. A malicious user can craft HTML that, if outputted directly by your web app, can be dangerous to the person viewing it.

While it may be tempting to use regular expressions to sanitize HTML, *do not do this*. HTML is a complex language and it's virtually guaranteed that any attempt you make at using regular expressions to sanitize HTML will fail.

You might also find advice suggesting you use the strip_tags() function. While strip_tags() is technically safe to use, it's a "dumb" function in the sense that if the input is invalid HTML (say, is missing an ending tag), then strip_tags() might remove much more content than you expected. As such it's not a great choice either, because non-technical users often use the < and > characters in communications.

If you read the section on validating email addresses, you might also be considering using the filter_var() function. However the filter_var() function has problems with line breaks, and requires non-intuitive configuration to closely mirror the htmlentities() function. As such it's not a good choice either.

### Sanitization for simple requirements

If your web app only needs to completely escape (and thus render harmless, but not remove entirely) HTML, use PHP's built-in htmlentities() function. This function is much faster than HTML Purifier, because it doesn't perform any validation on the HTML—it just escapes everything.

htmlentities() differs from its cousin htmlspecialchars() in that it encodes *all* applicable HTML entities, not just a small subset.

### Example

?

```
<?php
// Oh no!  The user has submitted malicious HTML, and we have to display it in our web app!
$evilHtml = '<div onclick="xss();">Mua-ha-ha!  Twiddling my evil mustache...</div>';

// Use the ENT_QUOTES flag to make sure both single and double quotes are escaped.
// Use the UTF-8 character encoding if you've stored the text as UTF-8 (as you should have).
// See the UTF-8 section in this document for more details.
$safeHtml = htmlentities($evilHtml, ENT_QUOTES, 'UTF-8'); // $safeHtml is now fully escaped HTML.  You can output $safeHtml to your users without fear!
?>
```

### Sanitization for complex requirements

For many web apps, simply escaping HTML isn't enough. You probably want to entirely remove any HTML, or allow a small subset of HTML through. To do this, use the HTML Purifier library.

HTML Purifier is a well-tested but slow library. That's why you should use htmlentities() if your requirements aren't that complex, because it will be much, much faster.

HTML Purifier has the advantage over strip_tags() because it validates the HTML before sanitizing it. That means if the user has inputted invalid HTML, HTML Purifier has a better chance of preserving the intended meaning of the HTML than strip_tags() does. It's also highly customizable, allowing you to whitelist a subset of HTML to keep in the output.

The downside is that it's quite slow, it requires some setup that might not be feasible in a shared hosting environment, and the documentation is often complex and unclear. The following example is a basic configuration; check the documentation to read about the more advanced features HTML Purifier offers.

### Example

?

```
<?php
// Include the HTML Purifier library
require_once('htmlpurifier-4.6.0/HTMLPurifier.auto.php');

// Oh no!  The user has submitted malicious HTML, and we have to display it in our web app!
$evilHtml = '<div onclick="xss();">Mua-ha-ha!  Twiddling my evil mustache...</div>';
```

```
7
8 // Set up the HTML Purifier object with the default configuration.
9 $purifier = new HTMLPurifier(HTMLPurifier_Config::createDefault());
10
11$safeHtml = $purifier->purify($evilHtml); // $safeHtml is now sanitized.  You can output
12$safeHtml to your users without fear!
  ?>
```

### Gotchas

- Using htmlentities() with the wrong character encoding can result in surprising output. Always make sure that you specify a character encoding when calling the function, and that it matches the encoding of the string being sanitized. See the UTF-8 section for more details.
- Always include the ENT_QUOTES and character encoding parameters when using htmlentities(). By default, htmlentities() doesn't encode single quotes. What a dumb default!
- HTML Purifier is extremely slow for complex HTML. Consider setting up a caching solution like APC to store the sanitized result for later use.

### Further reading

- Comparison between PHP HTML sanitizers
- Stack Overflow: Prevent XSS with strip_tags()?
- Stack Overflow: What's the best method for sanitizing user input with PHP?
- Stack Overflow: FILTER_SANITIZE_SPECIAL_CHARS problem with line breaks

# PHP and UTF-8

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

UTF-8 in PHP sucks. Sorry.

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(). Together, these counterpart functions are called the Multibyte String Functions. The multibyte string functions 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.

Additionally, 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.

Finally, 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.

### UTF-8 at the MySQL 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. For an example, see the section on connecting to and querying a MySQL database. 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. In your HTML, include the charset <meta> tag in your page's <head> tag.

### Example

?

```
<?php
// Tell PHP that we're using UTF-8 strings until the end of the script
mb_internal_encoding('UTF-8');

// Tell PHP that we'll be outputting UTF-8 to the browser
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 that we define the character set as utf8mb4 in the PDO connection string
$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) values (?, ?)');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->bindValue(2, $string);
$handle->execute();

// Retrieve the string we just stored to prove it was stored correctly
$handle = $link->prepare('select * from ElvishSentences where Id = ?');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->execute();

// Store the result into an object that we'll output later in our HTML
$result = $handle->fetchAll(\PDO::FETCH_OBJ);
?><!doctype html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>UTF-8 test page</title>
    </head>
    <body>
        <?php
        foreach($result as $row){
            print($row->Body);  // This should correctly output our transformed UTF-8
string to the browser
        }
        ?>
    </body>
</html>
```

### Further reading

- PHP Manual: Multibyte String Functions
- PHP UTF-8 Cheatsheet
- 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

# Working with dates and times

## Use the DateTime class.

In the bad old days of PHP we had to work with dates and times using a bewildering combination of date(), gmdate(), date_timezone_set(), strtotime(), and so on. Sadly you'll still find lots of tutorials online featuring these difficult and old-fashioned functions.

Fortunately for us, the version of PHP we're talking about features the much friendlier DateTime class. This class encapsulates all the functionality *and more* of the old date functions in one easy-to-use class, with the bonus of making time zone conversions much simpler. Always use the DateTime

class for creating, comparing, changing, and displaying dates in PHP.

### Example

?

```php
<?php
// Construct a new UTC date.  Always specify UTC unless you really know what you're doing!
$date = new DateTime('2011-05-04 05:00:00', new DateTimeZone('UTC'));

// Add ten days to our initial date
$date->add(new DateInterval('P10D'));

echo($date->format('Y-m-d h:i:s')); // 2011-05-14 05:00:00

// Sadly we don't have a Middle Earth timezone
// Convert our UTC date to the PST (or PDT, depending) time zone
$date->setTimezone(new DateTimeZone('America/Los_Angeles'));

// Note that if you run this line yourself, it might differ by an hour depending on daylight savings
echo($date->format('Y-m-d h:i:s')); // 2011-05-13 10:00:00

$later = new DateTime('2012-05-20', new DateTimeZone('UTC'));

// Compare two dates
if($date < $later)
    echo('Yup, you can compare dates using these easy operators!');

// Find the difference between two dates
$difference = $date->diff($later);

echo('The 2nd date is ' . $difference['days'] . ' later than 1st date.');
?>
```

### Gotchas

- If you don't specify a time zone, DateTime::__construct() will set the resulting date's time zone to *the time zone of the computer you're running on*. This can lead to spectacular headaches later on. **Always specify the UTC time zone when creating new dates unless you really know what you're doing.**
- If you use a Unix timestamp in DateTime::__construct(), the time zone will always be set to UTC regardless of what you specify in the second argument.
- Passing zeroed dates (e.g. "0000-00-00", a value commonly produced by MySQL as the default value in a DateTime column) to DateTime::__construct() will result in a nonsensical date, not "0000-00-00".
- Using DateTime::getTimestamp() on 32-bit systems will not represent dates past 2038. 64-bit systems are OK.

### Further Reading

- PHP Manual: The DateTime class
- Stack Overflow: Accessing dates beyond 2038

# Checking if a value is null or false

## Use the === operator to check for null and boolean false values.

PHP's loose typing system offers many different ways of checking a variable's value. However it also presents a lot of problems. Using == to check if a value is null or false can return false positives if the value is actually an empty string or 0. isset() checks whether a variable has a value, *not* whether that value is null or false, so it's not appropriate to use here.

The is_null() function accurately checks if a value is null, and the is_bool() function checks if it's a boolean value (like false), but there's an even better option: the === operator. === checks if the values are *identical*, which is not the same as *equivalent* in PHP's loosely-typed world. It's also slightly faster than is_null() and is_bool(), and can be considered by some to be cleaner than using a function for comparison.

### Example

```
<?php
$x = 0;
$y = null;

// Is $x null?
if($x == null)
    print('Oops! $x is 0, not null!');

// Is $y null?
if(is_null($y))
    print('Great, but could be faster.');

if($y === null)
    print('Perfect!');

// Does the string abc contain the character a?
if(strpos('abc', 'a'))
    // GOTCHA!  strpos returns 0, indicating it wishes to return the position of the first character.
    // But PHP interpretes 0 as false, so we never reach this print statement!
    print('Found it!');

//Solution: use !== (the opposite of ===) to see if strpos() returns 0, or boolean false.
if(strpos('abc', 'a') !== false)
    print('Found it for real this time!');
?>
```

### Gotchas

- When testing the return value of a function that can return either 0 or boolean false, like strpos(), always use === and !==, or you'll run in to problems.

**Further reading**

- PHP Manual: Comparison operators
- Stack Overflow: is_null() vs ===

# Suggestions and corrections

Thanks for reading! If you haven't figured it out already, PHP is complex and filled with pitfalls. Since I'm only human, there might be mistakes in this document.

If you'd like to contribute to this document with suggestions or corrections, please contact me using the information in the last revised & maintainers section.