

Engineering

 What are you looking for?


TECHNOLOGY 17 MINUTE READ

Buggy PHP Code: The 10 Most Common Mistakes PHP Developers Make



[Hire a Developer](#)

®



PHP makes it relatively easy to build a web-based system, which is much of the reason for its popularity. But its ease of use notwithstanding, PHP has evolved into quite a sophisticated language, with many nuances and subtleties that can bite developers, leading to hours of hair-pulling debugging. This article highlights ten of the more common mistakes that PHP developers need to beware of.

◆ **Toptal**. authors are vetted experts in their fields and write on topics in which they have demonstrated experience. All of our content is peer reviewed and validated by Toptal experts in the same field.

[Hire a Developer](#)

®

liya is an IT consultant, web architect, and manager with 12+ years of experience building and leading teams.

EXPERTISE

PHP

Back-end

Web

PREVIOUSLY AT

IHUEPOSTA

very easy to build a [web-based system](#), which is much of the reason for its popularity. But its ease of learning, [PHP has evolved into quite a sophisticated language](#) with many frameworks, nuances, and bite developers, leading to hours of hair-pulling debugging. This article highlights ten of the more that [PHP developers](#) need to beware of.



SHARE THIS ARTICLE



Common Mistake #1: Leaving dangling array references after `foreach` loops

Not sure how to use `foreach` loops in PHP? Using references in `foreach` loops can be useful if you want to operate on each element in the array that you are iterating over. For example:

```
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
// $arr is now array(2, 4, 6, 8)
```

The problem is that, if you're not careful, this can also have some undesirable side effects and consequences. Specifically, in the above example, after the code is executed, `$value` will remain in scope and will hold a reference to

The main thing to remember is that `foreach` does not create a scope. Thus, `$value` in the above example is a *reference* within the top scope of the script. On each iteration `foreach` sets the reference to point to the next element of `$array`. After the loop completes, therefore, `$value` still points to the last element of `$array` and remains in scope.

Here's an example of the kind of evasive and confusing bugs that this can lead to:

```
$array = [1, 2, 3];  
echo implode(', ', $array), "\n";  
  
foreach ($array as &$value) {} // by reference  
echo implode(', ', $array), "\n";  
  
foreach ($array as $value) {} // by value (i.e., copy)  
echo implode(', ', $array), "\n";
```

The above code will output the following:

```
1, 2, 3  
1, 2, 3  
1, 2, 2
```

Why?

After going through the first `foreach` loop, `$array` remains unchanged but, as explained above, `$value` is left as a dangling reference to the last element in `$array` (since that `foreach` loop accessed `$value` by *reference*).

As a result, when we go through the second `foreach` loop, “weird stuff” appears to happen. Specifically, since `$value` is now being accessed by value (i.e., by *copy*), `foreach` *copies* each sequential `$array` element into `$value` in each step of the loop. As a result, here’s what happens during each step of the second `foreach` loop:

- *Pass 1:* Copies `$array[0]` (i.e., “1”) into `$value` (which is a reference to `$array[2]`), so `$array[2]` now equals 1. So `$array` now contains [1, 2, 1].
- *Pass 2:* Copies `$array[1]` (i.e., “2”) into `$value` (which is a reference to `$array[2]`), so `$array[2]` now equals 2. So `$array` now contains [1, 2, 2].
- *Pass 3:* Copies `$array[2]` (which now equals “2”) into `$value` (which is a reference to `$array[2]`), so `$array[2]` still equals 2. So `$array` now contains [1, 2, 2].

To still get the benefit of using references in `foreach` loops without running the risk of these kinds of problems, call `unset()` on the variable, immediately after the `foreach` loop, to remove the reference; e.g.:

[Hire a Developer](#)

```
}  
unset($value); // $value no longer references $arr[3]
```

Common Mistake #2: Misunderstanding `isset()` behavior

Despite its name, `isset()` not only returns false if an item does not exist, but *also returns false for null values*.

This behavior is more problematic than it might appear at first and is a common source of problems.

Consider the following:

```
$data = fetchRecordFromStorage($storage, $identifier);  
if (!isset($data['keyShouldBeSet'])) {  
    // do something here if 'keyShouldBeSet' is not set  
}
```

The author of this code presumably wanted to check if `keyShouldBeSet` was set in `$data`. But, as discussed, `isset($data['keyShouldBeSet'])` will *also* return false if `$data['keyShouldBeSet']` was set, but was set to `null`. So the above logic is flawed.

```
if ($_POST['active']) {  
    $postData = extractSomething($_POST);  
}  
  
// ...  
  
if (!isset($postData)) {  
    echo 'post not active';  
}
```

The above code assumes that if `$_POST['active']` returns `true`, then `postData` will necessarily be set, and therefore `isset($postData)` will return `true`. So conversely, the above code assumes that the *only* way that `isset($postData)` will return `false` is if `$_POST['active']` returned `false` as well.

Not.

As explained, `isset($postData)` will also return `false` if `$postData` was set to `null`. It therefore *is* possible for `isset($postData)` to return `false` even if `$_POST['active']` returned `true`. So again, the above logic is flawed.

And by the way, as a side point, if the intent in the above code really was to again check if `$_POST['active']` returned `true`, relying on `isset()` for this was a poor coding decision in any case. Instead, it would have been better to just recheck `$_POST['active']`; i.e.:

```

    }

    // ...

    if ($_POST['active']) {
        echo 'post not active';
    }
}
```

For cases, though, where it *is* important to check if a variable was really set (i.e., to distinguish between a variable that wasn't set and a variable that was set to `null`), the `array_key_exists()` method is a much more robust solution.

For example, we could rewrite the first of the above two examples as follows:

```
$data = fetchRecordFromStorage($storage, $identifier);
if (! array_key_exists('keyShouldBeSet', $data)) {
    // do this if 'keyShouldBeSet' isn't set
}
```

Moreover, by combining `array_key_exists()` with `get_defined_vars()`, we can reliably check whether a variable within the current scope has been set or not:

```
if (array_key_exists('varShouldBeSet', get_defined_vars())) {
    // variable $varShouldBeSet exists in current scope
}
```


Common Mistake #3: Confusion about returning by reference vs. by value

Consider this code snippet:

```
class Config
{
    private $values = [];

    public function getValues() {
        return $this->values;
    }
}

$config = new Config();

$config->getValues()['test'] = 'test';
echo $config->getValues()['test'];
```

If you run the above code, you'll get the following:

```
PHP Notice: Undefined index: test in /path/to/my/script.php on line 21
```

What's wrong?

value". This means that a *copy* of the array will be returned and therefore the called function and the caller will not be accessing the same instance of the array.

So the above call to `getValues()` returns a *copy* of the `$values` array rather than a reference to it. With that in mind, let's revisit the two key lines from the above the example:

```
// getValues() returns a COPY of the $values array, so this adds a 'test' element
// to a COPY of the $values array, but not to the $values array itself.
$config->getValues()['test'] = 'test';

// getValues() again returns ANOTHER COPY of the $values array, and THIS copy doesn't
// contain a 'test' element (which is why we get the "undefined index" message).
echo $config->getValues()['test'];
```

One possible fix would be to save the first copy of the `$values` array returned by `getValues()` and then operate on that copy subsequently; e.g.:

```
$vals = $config->getValues();
$vals['test'] = 'test';
echo $vals['test'];
```

modify the original `$values` array. So if you *do* want your modifications (such as adding a 'test' element) to affect the original array, you would instead need to modify the `getValues()` function to return a *reference* to the `$values` array itself. This is done by adding a `&` before the function name, thereby indicating that it should return a reference; i.e.:

```
class Config
{
    private $values = [];

    // return a REFERENCE to the actual $values array
    public function &getValues() {
        return $this->values;
    }
}

$config = new Config();

$config->getValues()['test'] = 'test';
echo $config->getValues()['test'];
```

The output of this will be `test`, as expected.

But to make things more confusing, consider instead the following code snippet:

[Hire a Developer](#)

```
private $values;

// using ArrayObject rather than array
public function __construct() {
    $this->values = new ArrayObject();
}

public function getValues() {
    return $this->values;
}
}

$config = new Config();

$config->getValues()['test'] = 'test';
echo $config->getValues()['test'];
```

If you guessed that this would result in the same “undefined index” error as our earlier `array` example, you were wrong. In fact, *this* code will work just fine. The reason is that, unlike arrays, *PHP always passes objects by reference*. (`ArrayObject` is an SPL object, which fully mimics arrays usage, but works as an object.)

As these examples demonstrate, it is not always entirely obvious in PHP whether you are dealing with a copy or a reference. It is therefore essential to understand these default behaviors (i.e., variables and arrays are passed by value; objects are passed by reference) and also to carefully check the API documentation for the function you are calling to see if it is returning a value, a copy of an array, a reference to an array, or a reference to an object.

data. This “flies in the face” of encapsulation. Instead, it’s better to use old style “getters” and “setters”, e.g.:

```
class Config
{
    private $values = [];

    public function setValue($key, $value) {
        $this->values[$key] = $value;
    }

    public function getValue($key) {
        return $this->values[$key];
    }
}

$config = new Config();

$config->setValue('testKey', 'testValue');
echo $config->getValue('testKey');    // echos 'testValue'
```

This approach gives the caller the ability to set or get any value in the array without providing public access to the otherwise-private `$values` array itself.

Common Mistake #4: Performing queries in a loop

```
$models = [];  
  
foreach ($inputValues as $inputValue) {  
    $models[] = $valueRepository->findByValue($inputValue);  
}
```

While there may be absolutely nothing wrong here, but if you follow the logic in the code, you may find that the innocent looking call above to `$valueRepository->findByValue()` ultimately results in a query of some sort, such as:

```
$result = $connection->query("SELECT `x`,`y` FROM `values` WHERE `value`=" . $inputValue);
```

As a result, each iteration of the above loop would result in a separate query to the database. So if, for example, you supplied an array of 1,000 values to the loop, it would generate 1,000 separate queries to the resource! If such a script is called in multiple threads, it could potentially bring the system to a grinding halt.

It's therefore crucial to recognize when queries are being made by your code and, whenever possible, gather the values and then run one query to fetch all the results.

One example of a fairly common place to encounter querying being done inefficiently (i.e., in a loop) is when a form is posted with a list of values (IDs, for example). Then, to retrieve the full record data for each of the IDs, the code will loop through the array and do a separate SQL query for each ID. This will often look something like this:

[Hire a Developer](#)

```
$result = $connection->query("SELECT `x`, `y` FROM `values` WHERE `id` IN (" . implode(',', $ids) . ");");  
$data[] = $result->fetch_row();  
}
```

But the same thing can be accomplished much more efficiently in a *single* SQL query as follows:

```
$data = [];  
if (count($ids)) {  
    $result = $connection->query("SELECT `x`, `y` FROM `values` WHERE `id` IN (" . implode(',', $ids) . ");");  
    while ($row = $result->fetch_row()) {  
        $data[] = $row;  
    }  
}
```

It's therefore crucial to recognize when queries are being made, either directly or indirectly, by your code. Whenever possible, gather the values and then run one query to fetch all the results. Yet caution must be exercised there as well, which leads us to our next common PHP mistake...

Common Mistake #5: Memory usage headfakes and inefficiencies

While fetching many records at once is definitely more efficient than running a single query for each row to fetch, such an approach can potentially lead to an “out of memory” condition in `libmysqlclient` when using PHP's `mysql`

To demonstrate, let's take a look at a test box with limited resources (512MB RAM), MySQL, and `php-cli`.

We'll bootstrap a database table like this:

```
// connect to mysql
$connection = new mysqli('localhost', 'username', 'password', 'database');

// create table of 400 columns
$query = 'CREATE TABLE `test`(`id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT';
for ($col = 0; $col < 400; $col++) {
    $query .= ", `col$col` CHAR(10) NOT NULL";
}
$query .= ');';
$connection->query($query);

// write 2 million rows
for ($row = 0; $row < 2000000; $row++) {
    $query = "INSERT INTO `test` VALUES ($row";
    for ($col = 0; $col < 400; $col++) {
        $query .= ', ' . mt_rand(1000000000, 9999999999);
    }
    $query .= ');';
    $connection->query($query);
}
```

OK, now let's check resources usage:

[Hire a Developer](#)

```
echo "Before: " . memory_get_peak_usage() . "\n";

$res = $connection->query('SELECT `x`,`y` FROM `test` LIMIT 1');
echo "Limit 1: " . memory_get_peak_usage() . "\n";

$res = $connection->query('SELECT `x`,`y` FROM `test` LIMIT 10000');
echo "Limit 10000: " . memory_get_peak_usage() . "\n";
```

Output:

```
Before: 224704
Limit 1: 224704
Limit 10000: 224704
```

Cool. Looks like the query is safely managed internally in terms of resources.

Just to be sure, though, let's boost the limit one more time and set it to 100,000. Uh-oh. When we do that, we get:

```
PHP Warning: mysqli::query(): (HY000/2013):
Lost connection to MySQL server during query in /root/test.php on line 11
```

What happened?

PHP's manager, `memory_get_peak_usage()` won't show any increase in resources utilization as we up the limit in our query. This leads to problems like the one demonstrated above where we're tricked into complacency thinking that our memory management is fine. But in reality, our memory management is seriously flawed and we can experience problems like the one shown above.

You can at least avoid the above headfake (although it won't itself improve your memory utilization) by instead using the `mysqlnd` module. `mysqlnd` is compiled as a native PHP extension and it *does* use PHP's memory manager.

Therefore, if we run the above test using `mysqlnd` rather than `mysql`, we get a much more realistic picture of our memory utilization:

```
Before: 232048
Limit 1: 324952
Limit 10000: 32572912
```

And it's even worse than that, by the way. According to PHP documentation, `mysql` uses twice as many resources as `mysqlnd` to store data, so the original script using `mysql` really used even more memory than shown here (roughly twice as much).

To avoid such problems, consider limiting the size of your queries and using a loop with small number of iterations; e.g.:

```
for ($i = 0; $i <= ceil($totalNumberToFetch / $portionSize); $i++) {  
    $limitFrom = $portionSize * $i;  
    $res = $connection->query(  
        "SELECT `x`,`y` FROM `test` LIMIT $limitFrom, $portionSize");  
}
```

When we consider both this PHP mistake and [mistake #4](#) above, we realize that there is a healthy balance that your code ideally needs to achieve between, on the one hand, having your queries being too granular and repetitive, vs. having each of your individual queries be too large. As is true with most things in life, balance is needed; either extreme is not good and can cause problems with PHP not working properly.

Common Mistake #6: Ignoring Unicode/UTF-8 issues

In some sense, this is really more of an issue in PHP itself than something you would run into while debugging PHP, but it has never been adequately addressed. PHP 6's core was to be made Unicode-aware, but that was put on hold when development of PHP 6 was suspended back in 2010.

But that by no means absolves the developer from [properly handling UTF-8](#) and avoiding the erroneous assumption that all strings will necessarily be “plain old ASCII”. Code that fails to properly handle non-ASCII strings is notorious for introducing gnarly [heisenbugs](#) into your code. Even simple `strlen($_POST['name'])` calls could cause problems if someone with a last name like “Schrödinger” tried to sign up into your system.

- If you don't know much about Unicode and UTF-8, you should at least learn the basics. There's a great primer [here](#).
- Be sure to always use the `mb_*` functions instead of the old string functions (make sure the "multibyte" extension is included in your PHP build).
- Make sure your database and tables are set to use Unicode (many builds of MySQL still use `latin1` by default).
- Remember that `json_encode()` converts non-ASCII symbols (e.g., "Schrödinger" becomes "Schr\u00f6dinger") but `serialize()` does *not*.
- Make sure your PHP code files are also UTF-8 encoded to avoid collisions when concatenating strings with hardcoded or configured string constants.

A particularly valuable resource in this regard is the [UTF-8 Primer for PHP and MySQL](#) post by [Francisco Claria](#) on this blog.

Common Mistake #7: Assuming `$_POST` will always contain your POST data

Despite its name, the `$_POST` array won't always contain your POST data and can be easily found empty. To understand this, let's take a look at an example. Assume we make a server request with a `jQuery.ajax()` call as follows:

[Hire a Developer](#)

```
url: 'http://my-api.com/some/path',  
method: 'post',  
data: JSON.stringify({a: 'a', b: 'b'}),  
contentType: 'application/json'  
});
```

(Incidentally, note the `contentType: 'application/json'` here. We send data as JSON, which is quite popular for APIs. It's the default, for example, for posting in the [AngularJS \\$http service](#).)

On the server side of our example, we simply dump the `$_POST` array:

```
// php  
var_dump($_POST);
```

Surprisingly, the result will be:

```
array(0) { }
```

Why? What happened to our JSON string `{a: 'a', b: 'b'}`?

essentially the only ones used years ago when PHP's `$_POST` was implemented. So with any other content type (even those that are quite popular today, like `application/json`), PHP doesn't automatically load the POST payload.

Since `$_POST` is a superglobal, if we override it *once* (preferably early in our script), the modified value (i.e., including the POST payload) will then be referenceable throughout our code. This is important since `$_POST` is commonly used by PHP frameworks and almost all custom scripts to extract and transform request data.

So, for example, when processing a POST payload with a content type of `application/json`, we need to manually parse the request contents (i.e., decode the JSON data) and override the `$_POST` variable, as follows:

```
// php
$_POST = json_decode(file_get_contents('php://input'), true);
```

Then when we dump the `$_POST` array, we see that it correctly includes the POST payload; e.g.:

```
array(2) { ["a"]=> string(1) "a" ["b"]=> string(1) "b" }
```

Common Mistake #8: Thinking that PHP supports a character data type

Look at this sample piece of code and try guessing what it will print:

If you answered 'a' through 'z', you may be surprised to know that you were wrong.

Yes, it will print 'a' through 'z', but then it will *also* print 'aa' through 'yz'. Let's see why.

In PHP there's no `char` datatype; only `string` is available. With that in mind, incrementing the `string` `z` in PHP yields `aa`:

```
php> $c = 'z'; echo ++$c . "\n";  
aa
```

Yet to further confuse matters, `aa` is lexicographically *less* than `z`:

```
php> var_export((boolean)('aa' < 'z')) . "\n";  
true
```

That's why the sample code presented above prints the letters `a` through `z`, but then *also* prints `aa` through `yz`. It stops when it reaches `za`, which is the first value it encounters that is "greater than" `z`:

That being the case, here's one way to *properly* loop through the values 'a' through 'z' in PHP:

```
for ($i = ord('a'); $i <= ord('z'); $i++) {  
    echo chr($i) . "\n";  
}
```

Or alternatively:

```
$letters = range('a', 'z');  
  
for ($i = 0; $i < count($letters); $i++) {  
    echo $letters[$i] . "\n";  
}
```

Common Mistake #9: Ignoring coding standards

Although ignoring coding standards doesn't directly lead to needing to debug PHP code, it is still probably one of the most important things to discuss here.

work or can be difficult (sometimes almost impossible) to navigate, making it extremely difficult to debug, enhance, maintain. And that means reduced productivity for your team, including lots of wasted (or at least unnecessary) effort.

Fortunately for PHP developers, there is the PHP Standards Recommendation (PSR), comprised of the following five standards:

- [PSR-0](#): Autoloading Standard
- [PSR-1](#): Basic Coding Standard
- [PSR-2](#): Coding Style Guide
- [PSR-3](#): Logger Interface
- [PSR-4](#): Autoloader

PSR was originally created based on inputs from maintainers of the most recognized platforms on the market. Zend, Drupal, Symfony, Joomla and [others](#) contributed to these standards, and are now following them. Even PEAR, which attempted to be a standard for years before that, participates in PSR now.

In some sense, it almost doesn't matter what your coding standard is, as long as you agree on a standard and stick to it, but following the PSR is generally a good idea unless you have some compelling reason on your project to do

comfortable with your coding standard when they join your team.

Common Mistake #10: Misusing `empty()`

Some PHP developers like using `empty()` for boolean checks for just about everything. There are cases, though, where this can lead to confusion.

First, let's come back to arrays and `ArrayObject` instances (which mimic arrays). Given their similarity, it's easy to assume that arrays and `ArrayObject` instances will behave identically. This proves, however, to be a dangerous assumption. For example, in PHP 5.0:

```
// PHP 5.0 or later:
$array = [];
var_dump(empty($array));           // outputs bool(true)
$array = new ArrayObject();
var_dump(empty($array));           // outputs bool(false)
// why don't these both produce the same output?
```

And to make matters even worse, the results would have been different prior to PHP 5.0:

```
// Prior to PHP 5.0:
$array = [];
```

This approach is unfortunately quite popular. For example, this is the way `Zend\Db\TableGateway` of Zend Framework 2 returns data when calling `current()` on `TableGateway::select()` result as the doc suggests. Developer can easily become victim of this mistake with such data.

To avoid these issues, the better approach to checking for empty array structures is to use `count()`:

```
// Note that this work in ALL versions of PHP (both pre and post 5.0):  
$array = [];  
var_dump(count($array));           // outputs int(0)  
$array = new ArrayObject();  
var_dump(count($array));           // outputs int(0)
```

And incidentally, since PHP casts `0` to `false`, `count()` can also be used within `if ()` conditions to check for empty arrays. It's also worth noting that, in PHP, `count()` is constant complexity (`O(1)` operation) on arrays, which makes it even clearer that it's the right choice.

Another example when `empty()` can be dangerous is when combining it with the magic class function `__get()`. Let's define two classes and have a `test` property in both.

First let's define a `Regular` class that includes `test` as a normal property:

```
public $test = 'value';  
}
```

Then let's define a `Magic` class that uses the magic `__get()` operator to access its `test` property:

```
class Magic  
{  
    private $values = ['test' => 'value'];  
  
    public function __get($key)  
    {  
        if (isset($this->values[$key])) {  
            return $this->values[$key];  
        }  
    }  
}
```

OK, now let's see what happens when we attempt to access the `test` property of each of these classes:

```
$regular = new Regular();  
var_dump($regular->test);    // outputs string(4) "value"  
$magic = new Magic();  
var_dump($magic->test);      // outputs string(4) "value"
```

But now let's see what happens when we call `empty()` on each of these:

```
var_dump(empty($regular->test));    // outputs bool(false)
var_dump(empty($magic->test));      // outputs bool(true)
```

Ugh. So if we rely on `empty()`, we can be misled into believing that the `test` property of `$magic` is empty, whereas in reality it is set to `'value'`.

Unfortunately, if a class uses the magic `__get()` function to retrieve a property's value, there's no foolproof way to check if that property value is empty or not. Outside of the class' scope, you can really only check if a `null` value will be returned, and that doesn't necessarily mean that the corresponding key is not set, since it actually *could* have been set to `null`.

In contrast, if we attempt to reference a non-existent property of a `Regular` class instance, we will get a notice similar to the following:

```
Notice: Undefined property: Regular::$nonExistantTest in /path/to/test.php on line 10

Call Stack:
   0.0012    234704    1. {main}() /path/to/test.php:0
```

[Hire a Developer](#)

Wrap-up

PHP's ease of use can lull developers into a false sense of comfort, leaving themselves vulnerable to lengthy PHP debugging due to some of the nuances and idiosyncrasies of the language. This can result in PHP not working and problems such as those described herein.

The PHP language has evolved significantly over the course of its 20 year history. Familiarizing oneself with its subtleties is a worthwhile endeavor, as it will help ensure that the [software you produce](#) is more scalable, robust, and maintainable.

TAGS

[PHP](#)

Hire a Toptal expert on this topic.

[Hire Now](#)

[Hire a Developer](#)

®

Ilya is an IT consultant, web architect, and manager with 12+ years of experience building and leading teams.

🔗 **Toptal** authors are vetted experts in their fields and write on topics in which they have demonstrated experience. All of our content is peer reviewed and validated by Toptal experts in the same field.

EXPERTISE

PHP

Back-end

Web

PREVIOUSLY AT

Ilya Sanosian

🔗 **Verified Expert** in Engineering

Located in Oxford, United Kingdom

Member since September 19, 2013

[Hire Ilya](#)

TRENDING ARTICLES

[ENGINEERING](#) > [DATA SCIENCE AND DATABASES](#)

[Hire a Developer](#)[ENGINEERING](#) > [DATA SCIENCE AND DATABASES](#)

Ask an NLP Engineer: From GPT Models to the Ethics of AI

[ENGINEERING](#) > [TECHNOLOGY](#)

How to Use JWT and Node.js for Better App Security

[ENGINEERING](#) > [WEB FRONT-END](#)

Next.js vs. React: A Comparative Tutorial

SEE OUR RELATED TALENT

[PHP Developers](#)[Back-end Developers](#)[Web Developers](#)

VIEW RELATED SERVICES

[Application Services and Modernization](#)



Hire a Developer

World-class articles,
delivered weekly.

Enter your email

Sign Me Up

Subscription implies consent to our [privacy policy](#).

[Hire a Developer](#)

Algorithm Developers	Computer Vision Developers	Kubernetes Experts	SQL Developers
Angular Developers	Django Developers	Machine Learning Engineers	Sys Admins
AWS Developers	Docker Developers	Magento Developers	Tableau Developers
Azure Developers	Elixir Developers	.NET Developers	Unreal Engine Developers
Big Data Architects	Go Engineers	R Developers	Xamarin Developers
Blockchain Developers	GraphQL Developers	React Native Developers	View More Freelance Developers →
Business Intelligence Developers	Jenkins Developers	Ruby on Rails Developers	
C Developers	Kotlin Developers	Salesforce Developers	

Join the Toptal® community.

[Hire a Developer](#)

or

[Apply as a Developer](#)

[Hire a Developer](#)

ON-DEMAND TALENT

[Hire Freelance Developers](#)[Hire Freelance Designers](#)[Hire Freelance Finance Experts](#)[Hire Freelance Project Managers](#)[Hire Freelance Product Managers](#)

MANAGEMENT CONSULTING

[Strategy Consulting](#)[People & Organization Consulting](#)[Innovation & Experience Consulting](#)

TECHNOLOGY SERVICES

[Application Services](#)[Cloud Services](#)[Information Security Services](#)[Quality Assurance Services](#)

ABOUT

[Top 3%](#)[Clients](#)[Freelance Jobs](#)[Specialized Services](#)

CONTACT

[Contact Us](#)[Press Center](#)[Careers](#)[FAQ](#)



[Hire a Developer](#)

[About Us](#)



[Copyright 2010 - 2023 Toptal, LLC](#) [Privacy Policy](#) [Website Terms](#) [Accessibility](#)