

PRESENTED BY: Craig Francis

Ending Injection Vulnerabilities, Using developer defined strings.

<https://eiv.dev/>

Craig, start the timer.



```
$sql = 'SELECT * FROM user WHERE id = ' . $id;
```



`$sql = 'SELECT * FROM user WHERE id = ' . $id;`

`SELECT * FROM user WHERE id = 123`



`$sql = 'SELECT * FROM user WHERE id = ' . $id;`

`SELECT * FROM user WHERE id = -1 UNION SELECT * FROM admin`

```
<?php
$db = new mysqli('localhost', 'test', 'test', 'test');
$rows = $db->query('SELECT * FROM user WHERE id = ' . $_GET['id']);
foreach ($rows as $row) {
    print_r($row);
}
```

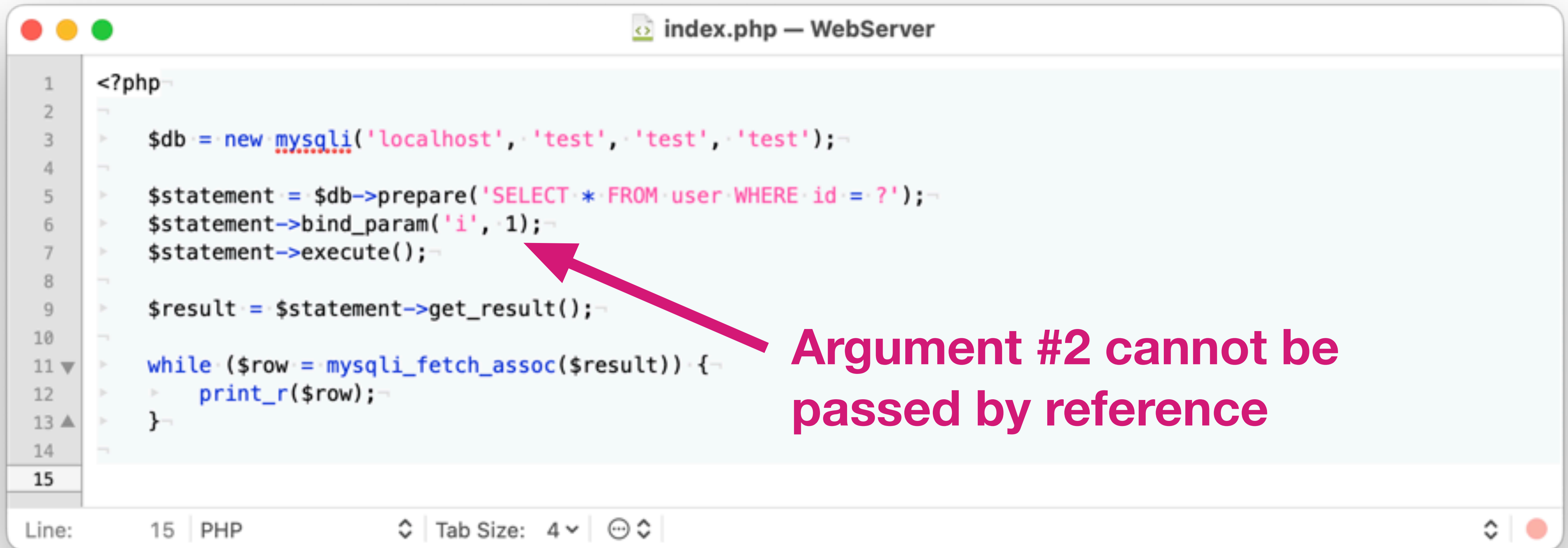
```
<?php
$db = new mysqli('localhost', 'test', 'test', 'test');
$rows = $db->query('SELECT * FROM user WHERE id = ' . $db->real_escape_string($_GET['id']));
foreach ($rows as $row) {
    print_r($row);
}
```



```
1 <?php
2
3 $db = new mysqli('localhost', 'test', 'test', 'test');
4
5 $rows = $db->query('SELECT * FROM user WHERE id = "'" . $db->real_escape_string($_GET['id']) . "'"');
6
7 foreach ($rows as $row) {
8     print_r($row);
9 }
10
11
12
13
14
15
```

Line: 15 | PHP | Tab Size: 4


```
<?php
1
2
3  $db = new mysqli('localhost', 'test', 'test', 'test');
4
5  $statement = $db->prepare('SELECT * FROM user WHERE id = ?');
6  $statement->bind_param('i', $_GET['id']);
7  $statement->execute();
8
9  $result = $statement->get_result();
10
11 while ($row = mysqli_fetch_assoc($result)) {
12     print_r($row);
13 }
14
15
```



```
1 <?php
2
3 $db = new mysqli('localhost', 'test', 'test', 'test');
4
5 $statement = $db->prepare('SELECT * FROM user WHERE id = ?');
6 $statement->bind_param('i', 1);
7 $statement->execute();
8
9 $result = $statement->get_result();
10
11 while ($row = mysqli_fetch_assoc($result)) {
12     print_r($row);
13 }
14
15
```

Line: 15 | PHP | Tab Size: 4

Argument #2 cannot be passed by reference

```
<?php
1
2
3  $db = new mysqli('localhost', 'test', 'test', 'test');
4
5  $statement = $db->prepare('SELECT * FROM user WHERE type IN (?, ?, ?)');
6  $statement->bind_param('sss', $type1, $type2, $type3);
7  $statement->execute();
8
9  $result = $statement->get_result();
10
11 while ($row = mysqli_fetch_assoc($result)) {
12     print_r($row);
13 }
14
15
```


PHP: rfc:mysql_execute_ X PHP: rfc:mysql_bind_in_ X PHP: rfc:mysql_execute_ X PHP: mysql::execute_qu X +

php

Edit this page Admin Logout Craig Francis (craigfrancis)

Search

start › rfc › mysql_bind_in_execute

PHP RFC: mysql bind in execute

- Version: 1.1
- Date: 2021-02-11
- Author: Kamil Tekiela, dharman@php.net
- Target version: PHP 8.1
- Implementation: <https://github.com/php/php-src/pull/6271>
- Status: Implemented

Introduction

PDO has always offered binding values to the prepared statement directly in the execute() call by providing an array with the values. The same functionality was never present in mysql, and many users have been confused by that lack of seemingly easy functionality. (See [Bug #40891](#), [Bug #31096](#))

Table of Contents

PHP RFC: mysql bind in execute

Introduction

Proposal

Backward Incompatible Changes

Proposed PHP Version(s)

RFC Impact

New Constants

php.ini Defaults

Unaffected PHP Functionality

Future Scope

Proposed Voting Choices

Implementation

References

```
<?php
1
2
3  $db = new mysqli('localhost', 'test', 'test', 'test');
4
5  $statement = $db->prepare('SELECT * FROM user WHERE id = ?');
6  $statement->bind_param('i', 1);
7  $statement->execute();
8
9  $result = $statement->get_result();
10
11 while ($row = mysqli_fetch_assoc($result)) {
12     print_r($row);
13 }
14
15
```

```
1 <?php
2
3 $db = new mysqli('localhost', 'test', 'test', 'test');
4
5 $statement = $db->prepare('SELECT * FROM user WHERE id = ?');
6 $statement->execute([1]);
7
8 $result = $statement->get_result();
9
10 while ($row = mysqli_fetch_assoc($result)) {
11     print_r($row);
12 }
```



```
<?php
1
2
3  $db = new mysqli('localhost', 'test', 'test', 'test');
4
5  $statement = $db->prepare('SELECT * FROM user WHERE type IN (?, ?, ?)');
6  $statement->bind_param('sss', $type1, $type2, $type3);
7  $statement->execute();
8
9  $result = $statement->get_result();
10
11 while ($row = mysqli_fetch_assoc($result)) {
12     print_r($row);
13 }
14
15
```

```
1 <?php
2
3 $db = new mysqli('localhost', 'test', 'test', 'test');
4
5 $statement = $db->prepare('SELECT * FROM user WHERE type IN (?, ?, ?)');
6 $statement->execute([$type1, $type2, $type3]);
7
8 $result = $statement->get_result();
9
10 while ($row = mysqli_fetch_assoc($result)) {
11     print_r($row);
12 }
```


PHP: rfc:mysql_execute_ X PHP: rfc:mysql_bind_in_ X PHP: rfc:mysql_execute_ X PHP: mysqli::execute_qu X +

← → ↺ 🏠

wiki.php.net/rfc/mysql_execute_query

🔖 ⭐ ⚙️ 🔒 🏠 👤 ⋮

php

Edit this page Admin Logout Craig Francis (craigfrancis)

Search

start › rfc › mysqli_execute_query

PHP RFC: MySQLi Execute Query

- Version: 1
- RFC Started: 2022-04-21
- RFC Updated: 2022-05-11
- Voting Start: 2022-05-11 15:00 UTC / 16:00 BST
- Voting End: 2022-05-25 15:00 UTC / 16:00 BST
- Author: Kamil Tekiela, and Craig Francis [craig#at#craigfrancis.co.uk]
- Status: Accepted
- Target Version: PHP 8.2
- First Published at: https://wiki.php.net/rfc/mysql_execute_query
- GitHub Repo: <https://github.com/craigfrancis/php-mysqli-execute-query-rfc>
- Implementation: [From Kamil Tekiela](#) (proof of concept)

Introduction

✎

🕒

🔗

✉

⬆

Table of Contents

- PHP RFC: MySQLi Execute Query
- Introduction
- Proposal
- Notes
- Function Name
- Returning false
- Properties
- Re-using Statements
- Updating Existing Functions
- Why Now
- Backward Incompatible Changes
- Proposed PHP Version(s)
- RFC Impact
 - To SAPIs
 - To Existing Extensions
 - To Opcache
- New Constants

```
1 <?php
2
3 $db = new mysqli('localhost', 'test', 'test', 'test');
4
5 $statement = $db->prepare('SELECT * FROM user WHERE id = ?');
6 $statement->execute([1]);
7
8 $result = $statement->get_result();
9
10 while ($row = mysqli_fetch_assoc($result)) {
11     print_r($row);
12 }
```

```
<?php
$db = new mysqli('localhost', 'test', 'test', 'test');

$rows = $db->execute_query('SELECT * FROM user WHERE id = ?', [$_GET['id']]);

foreach ($rows as $row) {
    print_r($row);
}
```



```
<?php
$db = new mysqli('localhost', 'test', 'test', 'test');
$rows = $db->query('SELECT * FROM user WHERE id = ' . $_GET['id']);
foreach ($rows as $row) {
    print_r($row);
}
```

```
<?php
```

```
$db = new mysqli('localhost', 'test', 'test', 'test');
```

```
$rows = $db->execute_query('SELECT * FROM user WHERE id = ?', [$_GET['id']]);
```

```
foreach ($rows as $row) {
```

```
    print_r($row);
```

```
}
```

Database Abstractions

ORMs

Query Builders

```
$articles->where('author_id', $id);
```

```
$articles->where('author_id IS NULL');
```

```
$articles->where('DATE(published)', $date);
```



`$articles->where('author_id', $id);`

`$articles->where('author_id IS NULL');`

`$articles->where('DATE(published)', $date);`

`$articles->where('word_count > 1000');`

`$articles->where('word_count > ', $count);`

`$articles->where('word_count > ' . $count);`

`'word_count > word_count UNION SELECT * FROM admin'`



Laravel DB

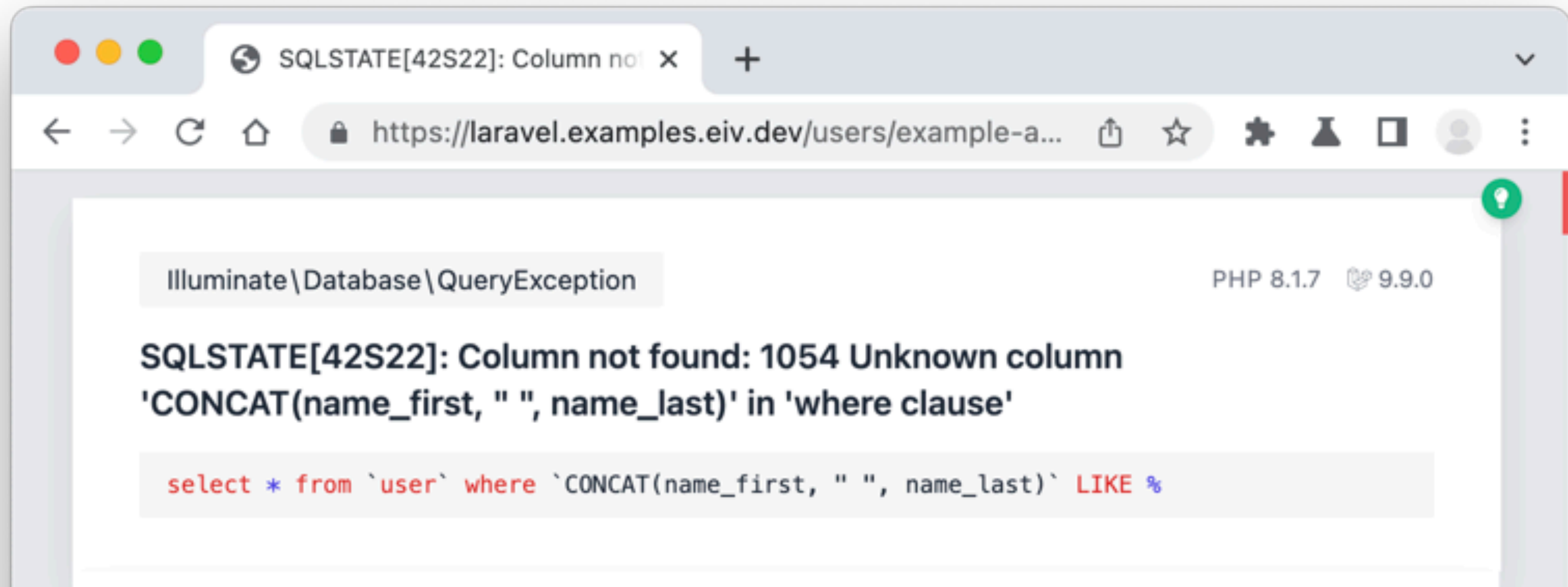
whereRaw()

```
DB::table('user')  
->where('id', '=', $id);
```

```
DB::table('user')  
->where('name', 'LIKE', $search . '%');
```

DB::table('user')

->where('CONCAT(name_first, " ", name_last)', 'LIKE', \$search . '%');



DB::table('user')

->whereRaw('CONCAT(name_first, " ", name_last) LIKE ?', \$search . '%');



DB::table('user')

->whereRaw('CONCAT(name_first, " ", name_last) LIKE "' . \$search . '%"');



DB::table('user')

->whereRaw('CONCAT(name_first, " ", name_last) LIKE "' . \$search . '%"');

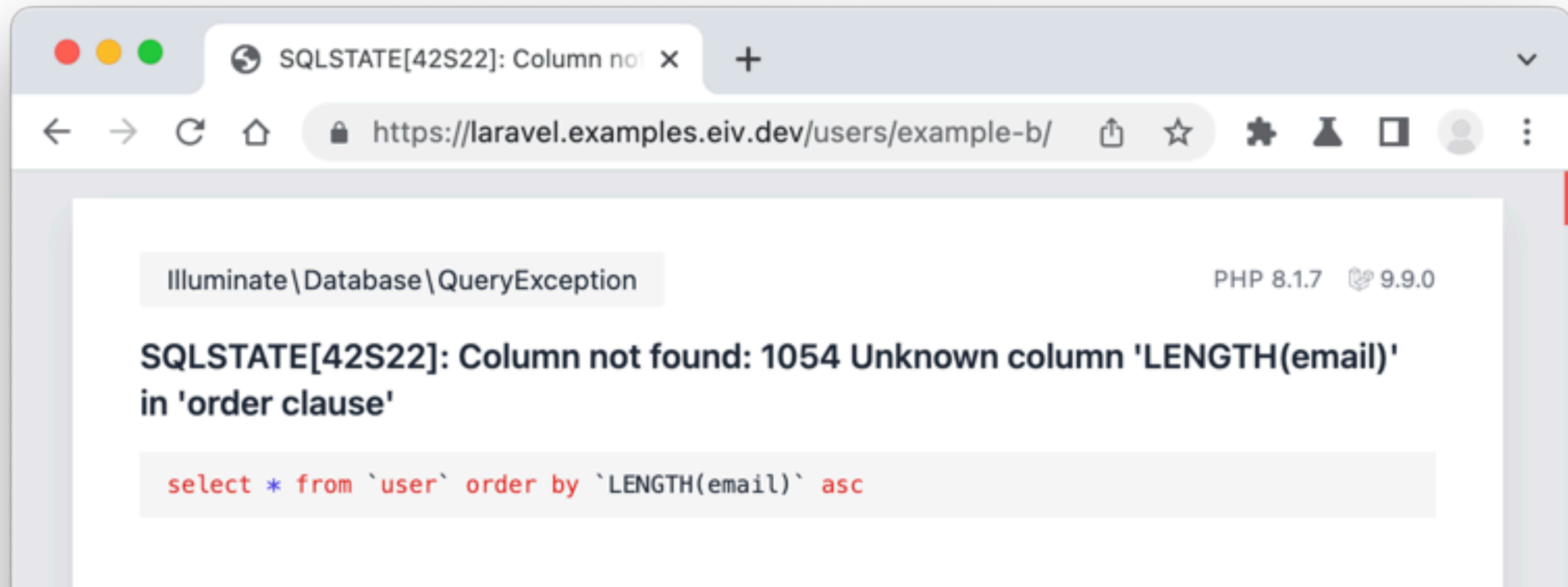


Laravel DB

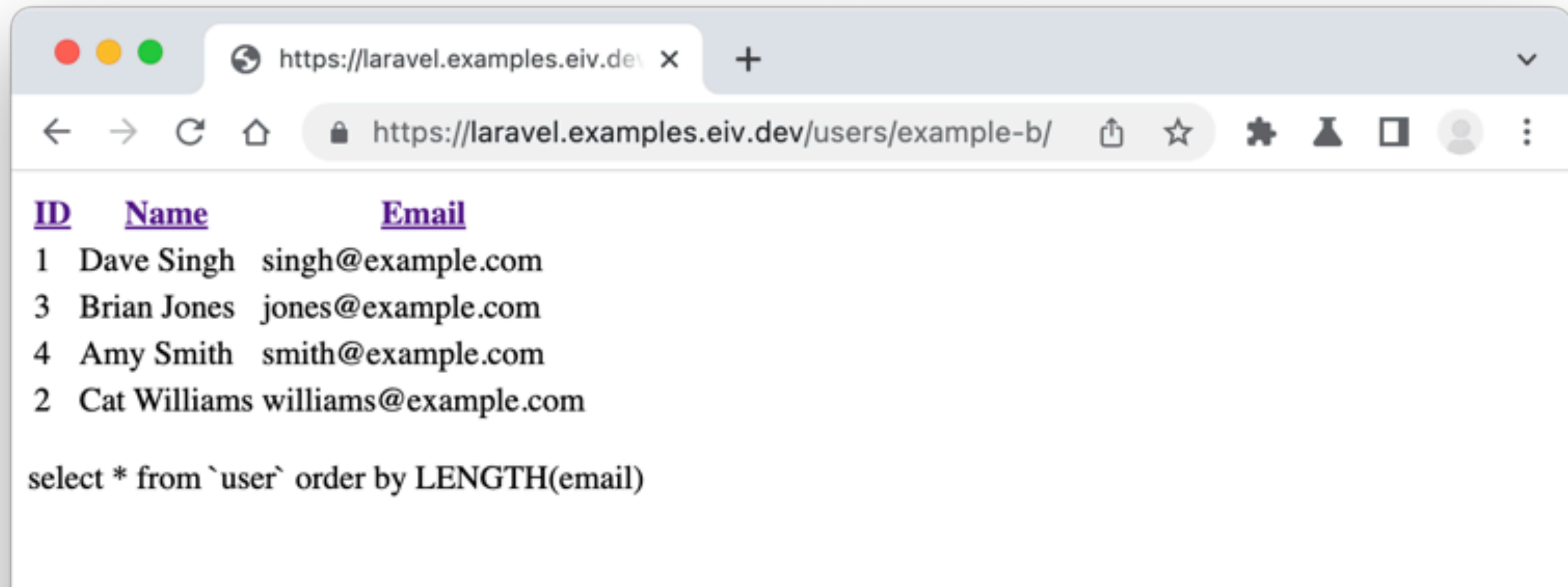
orderByRaw()


```
DB::table('user')  
->orderBy('email');
```

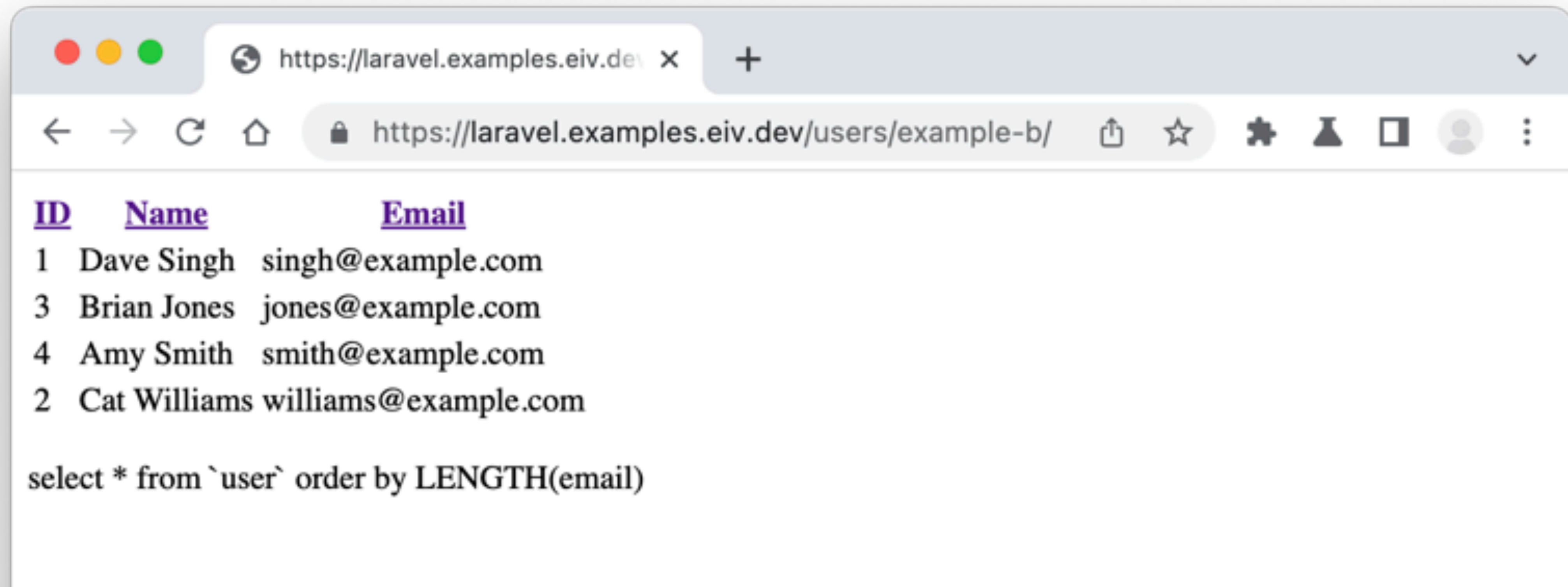
```
DB::table('user')  
->orderBy('LENGTH(email)');
```



```
DB::table('user')  
->orderByRaw('LENGTH(email)');
```

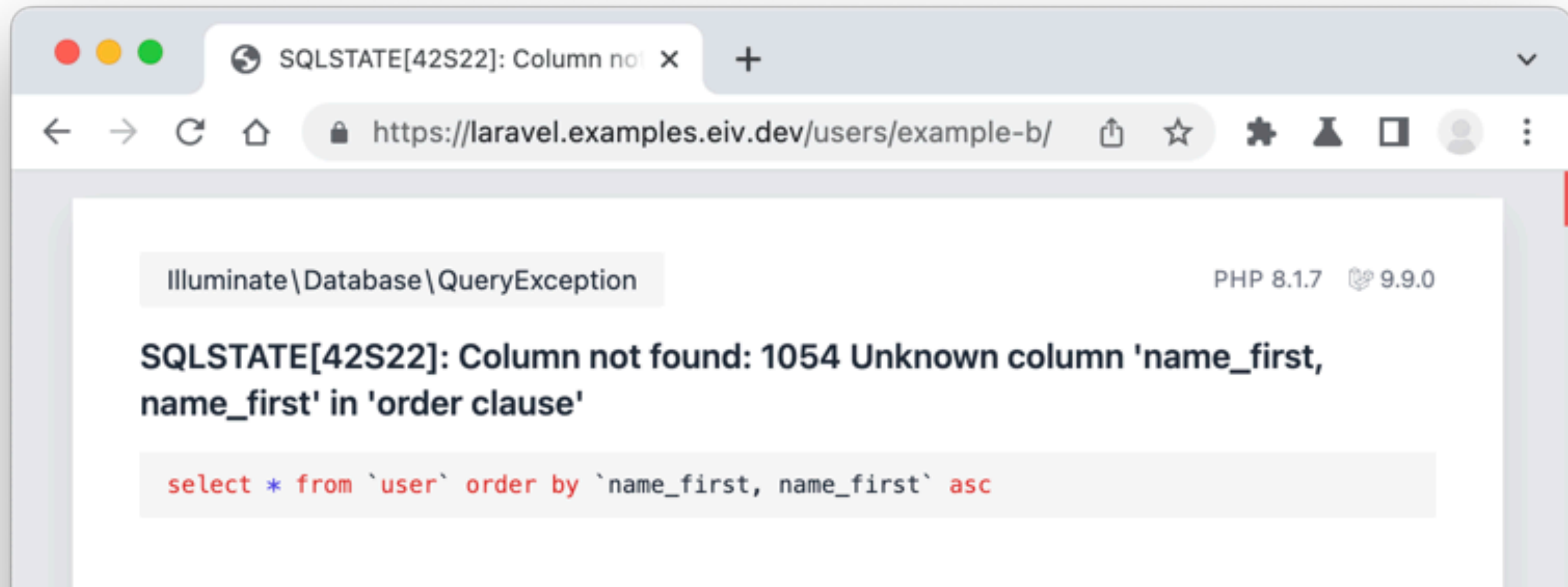


```
DB::table('user')  
->orderByRaw('LENGTH(' . $field . ')');
```



```
DB::table('user')  
    ->orderBy('name');
```

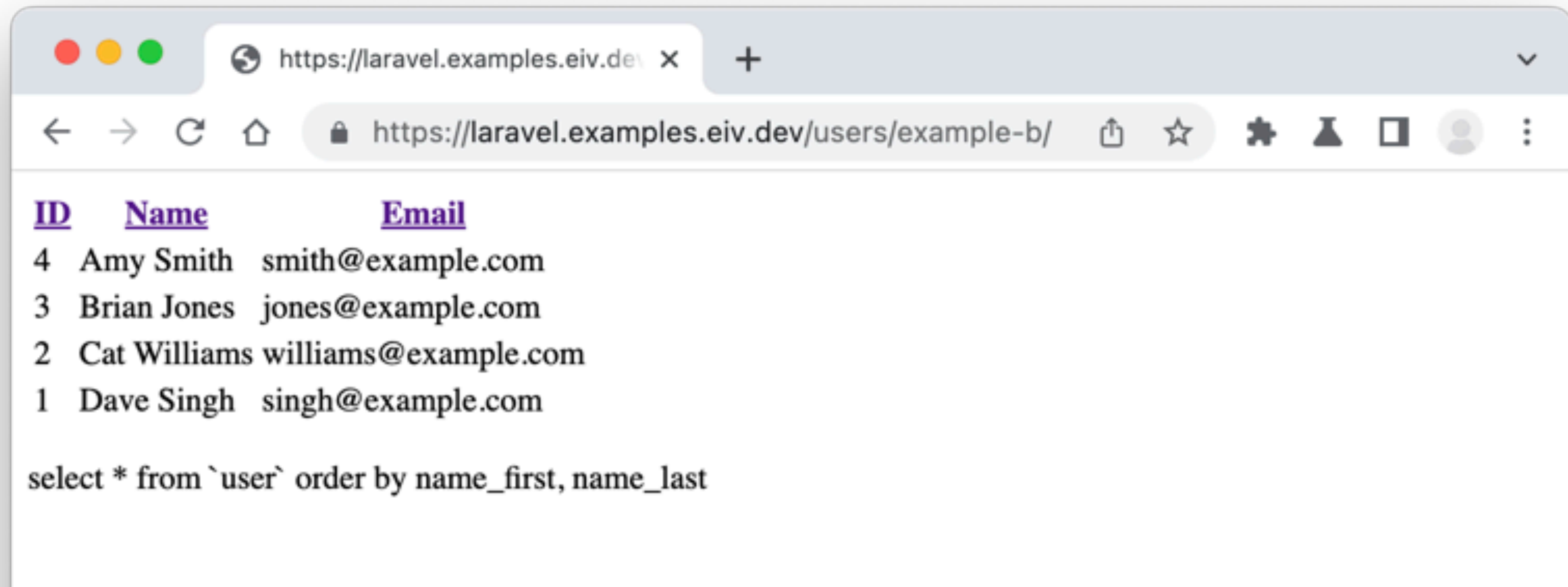
```
DB::table('user')  
->orderBy('name_first, name_last');
```



```
DB::table('user')  
    ->orderBy('name_first')  
    ->orderBy('name_last');
```



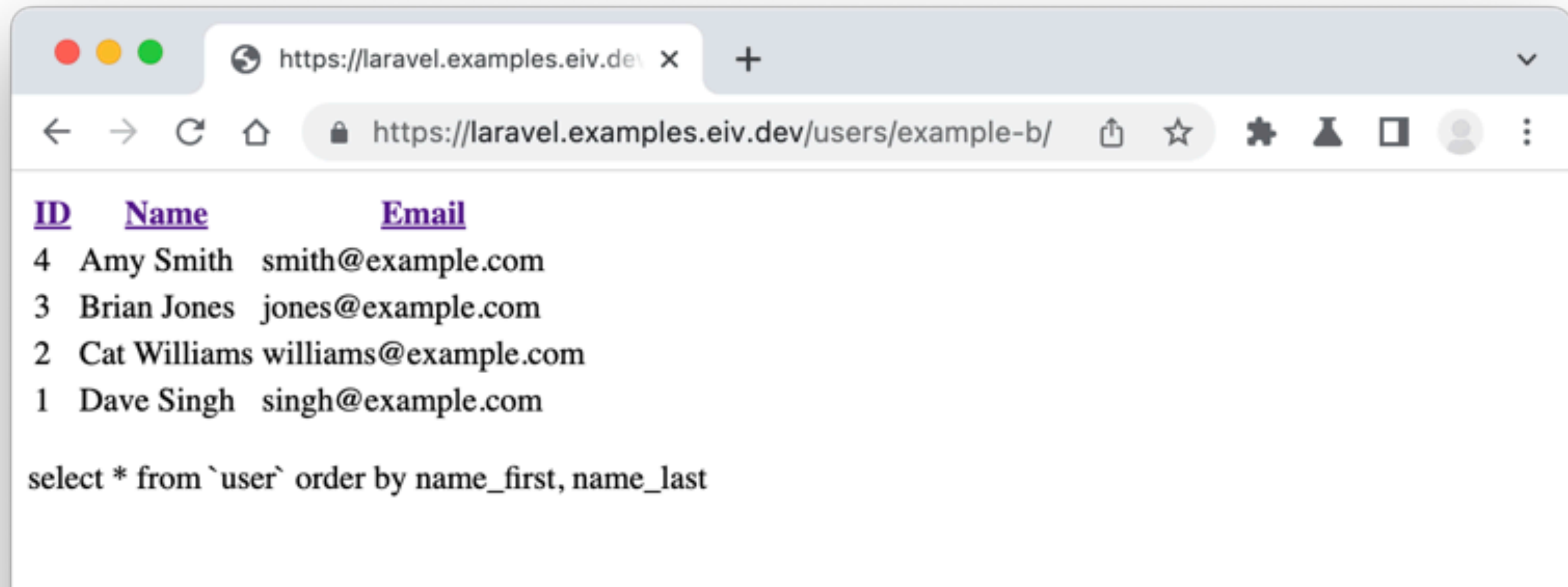
```
DB::table('user')  
->orderByRaw('name_first, name_last');
```

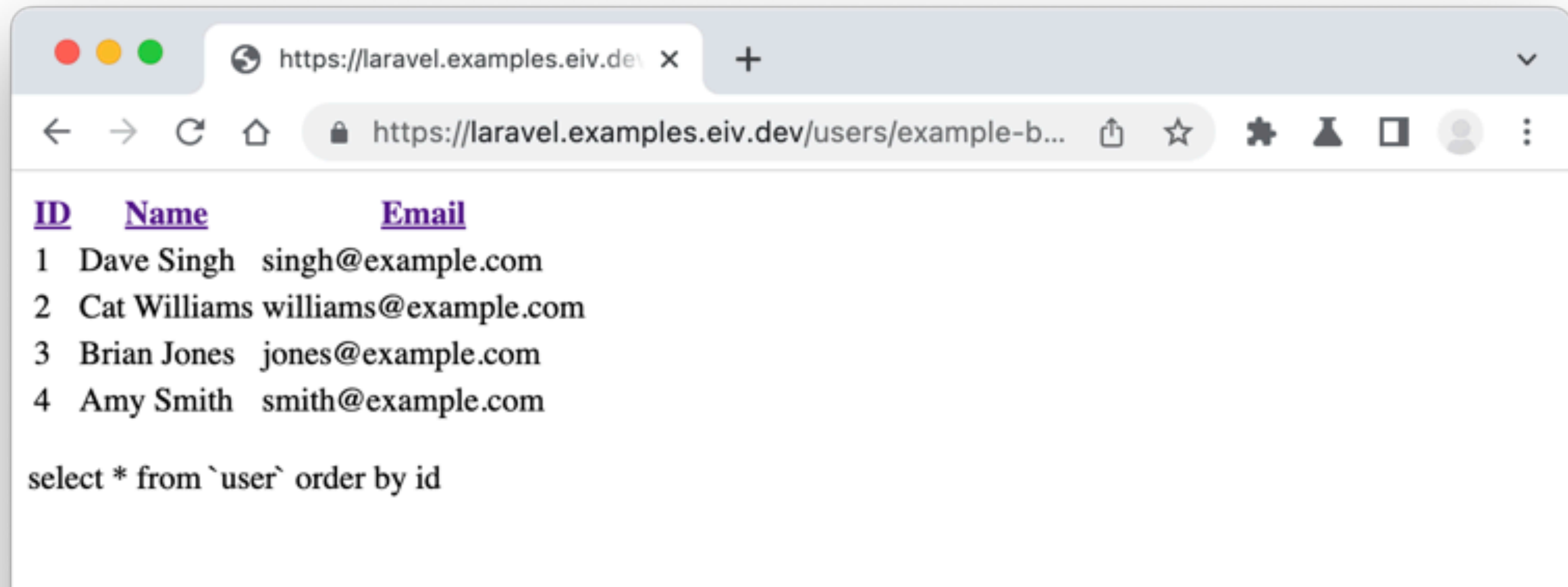


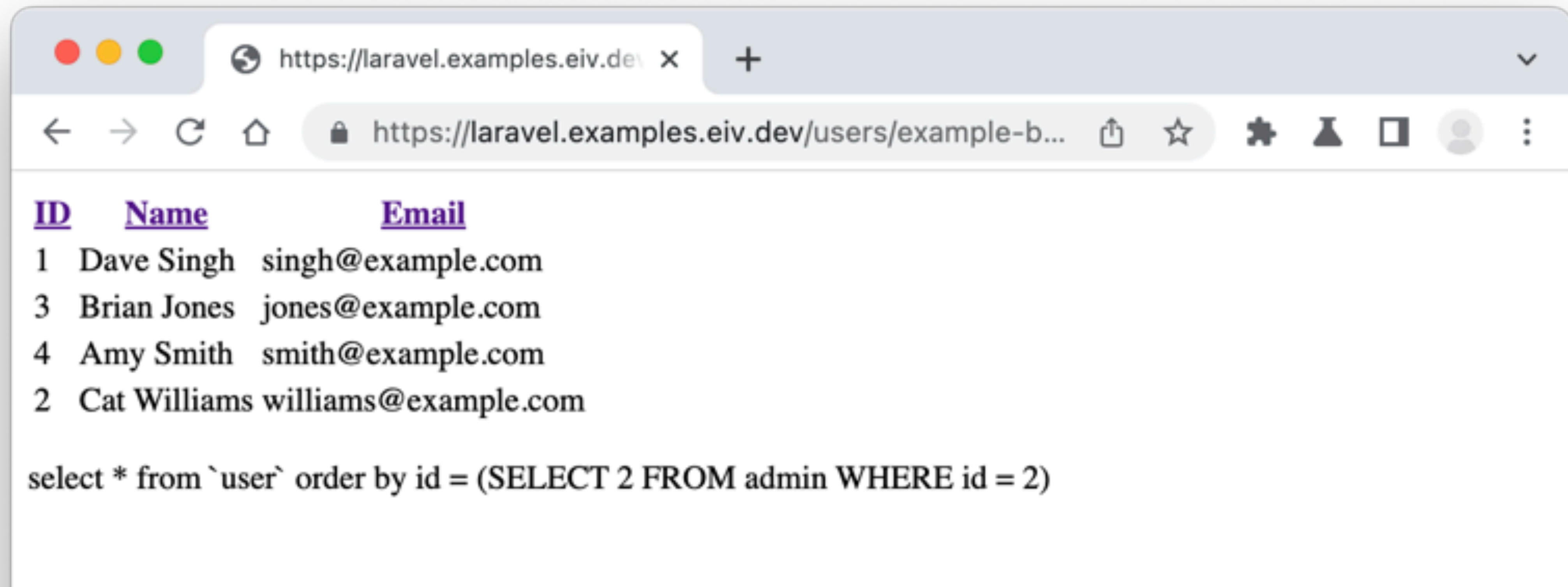

```
$sort = $request->input('sort');
```

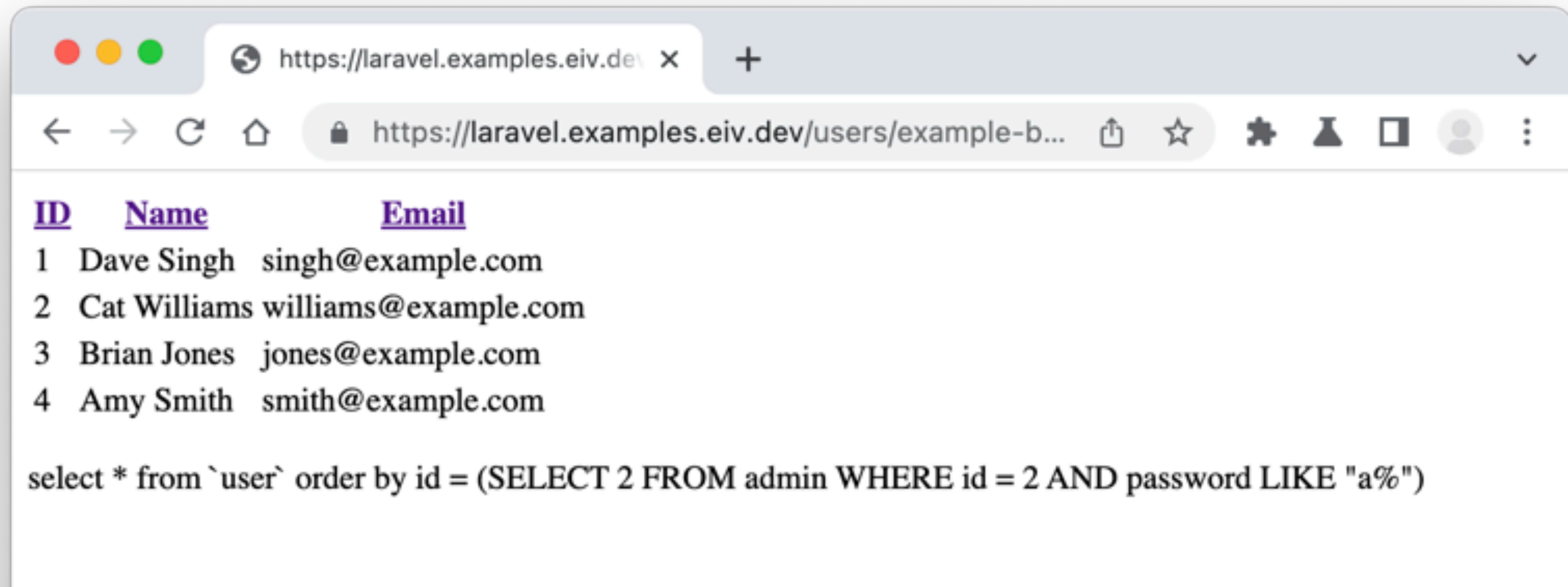
← Query String "/?sort=id"

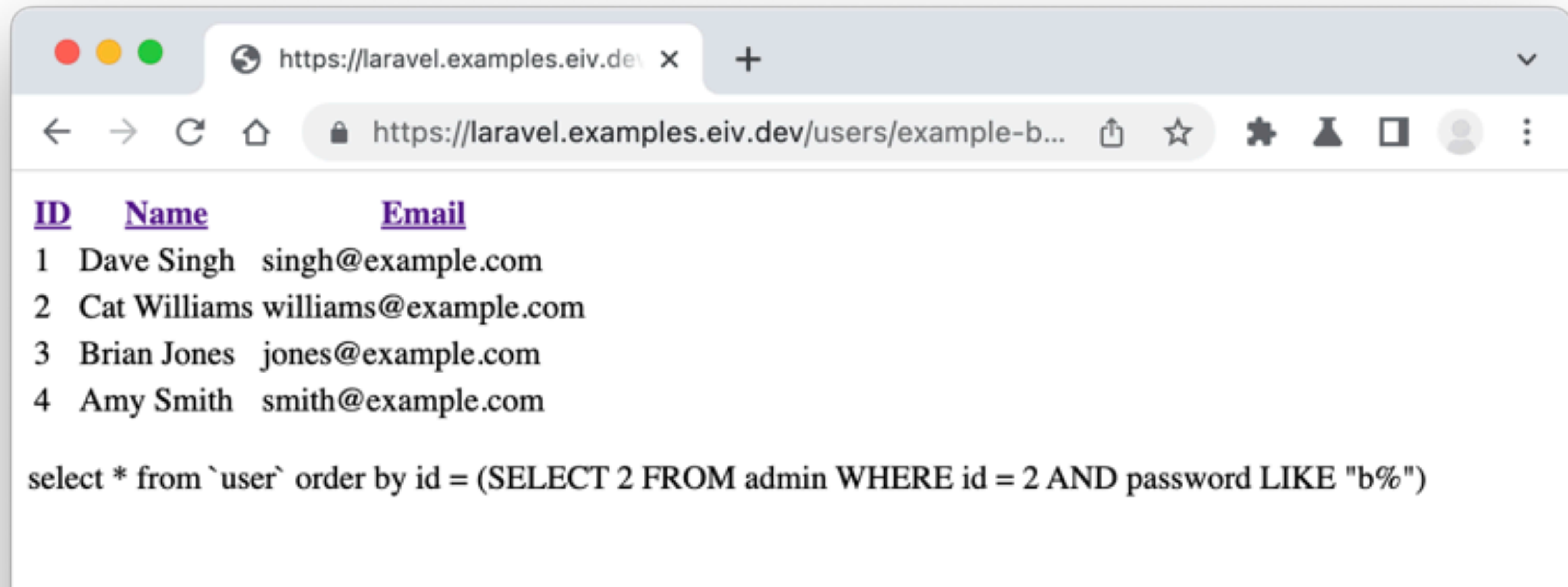
```
DB::table('user')  
->orderByRaw($sort ?? 'name_first, name_last');
```

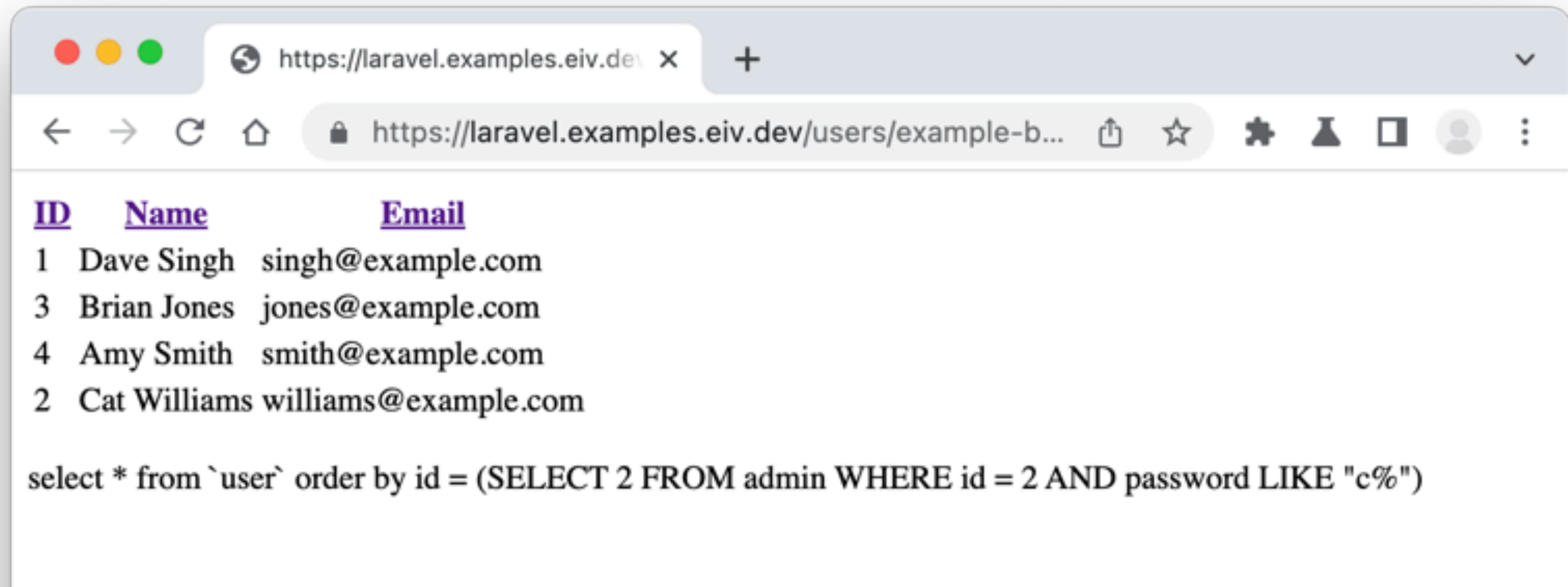












Laravel DB

DB::select()

```
$id = $request->input('id');
```

```
DB::select('SELECT * FROM user WHERE id = ?', [$id]);
```



```
$id = $request->input('id');
```

```
DB::select('SELECT * FROM user WHERE id = ' . $id);
```

Doctrine QueryBuilder

```
$qb->select('u')
```

```
->from('User', 'u')
```

```
->where('u.id = ' . $id); // INSECURE
```

```
$qb->select('u')
```

```
->from('User', 'u')
```

```
->where('u.id = :identifier')
```

```
->setParameter('identifier', $id);
```

```
$qb->select('u')  
  
->from('User', 'u')  
  
->where($qb->expr()->andX(  
  
    $qb->expr()->eq('u.type_id', $type_id),  
  
    $qb->expr()->isNull('u.deleted')  
  
));
```

```
$q->select('u')
```

```
->from('User', 'u')
```

```
->where($q->expr()->andX(
```

```
    $q->expr()->eq('u.type_id', $type_id), // INSECURE
```

```
    $q->expr()->isNull('u.deleted')
```

```
));
```

'u.type_id) OR (1 = 1'



```
$html = '<p>Hi {{ name }}</p>';
```

```
$template->render($html, ['name' => $name]);
```



```
$html = '<p>Hi ' . $name . '</p>';
```

```
$template->render($html);
```

```
$html = '<p>Hi ' . $name . '</p>';
```

```
$template->render($html);
```

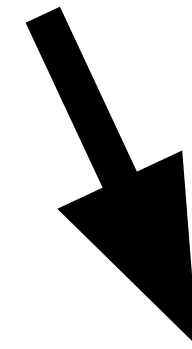
```
'<p>Hi <script>alert()</script></p>'
```



```
$html = '<a href="{{ url }}">Link</a>';
```

```
$template->render($html, ['url' => $url]);
```

Context aware?



```
$html = '<a href="{{ url }}">Link</a>';
```

```
$template->render($html, ['url' => $url]);
```

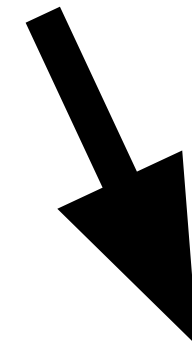
```
'<a href="javascript:alert()">Link</a>'
```



```
$html = '<img src={{ url }} alt="Alt Text" />';
```

```
$template->render($html, ['url' => $url]);
```

Missing quotes?



```
$html = '<img src={{ url }} alt="Alt Text" />';
```

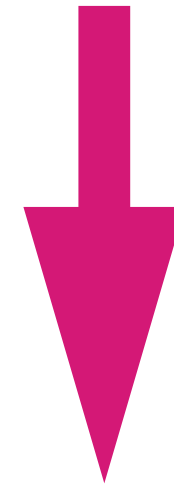
```
$template->render($html, ['url' => $url]);
```

```
'<img src=/ onerror=alert() alt="Alt Text" />'
```



Command Line Injection


```
$exec = 'grep "" . $search . "" /path/to/file';
```



```
grep "" /path/to/secrets; # "" /path/to/file
```



Taint Checking???

Where variables note if they are **Tainted**, or **Untainted**.



Untainted



```
$html = '<p>Hello Everybody,</p>';
```



Untainted

Untainted Tainted Untainted



\$html = '<p>Hi ' . **\$name** . '</p>';



Tainted



Untainted



Tainted Untainted



```
$html = '<p>Hi ' . htmlspecialchars($name) . '</p>';
```



Untainted



Escaped, to make it Safe :-)

<p>Hello <script>alert()</script></p>

Unfortunately Taint Checking incorrectly assumes escaping makes a value “safe” for *any* context.

Untainted



Tainted Untainted



```
$html = "<a href='" . htmlspecialchars($url) . "'>Link</a>";
```



Escapes & ' " < > ... Is this Safe?

Untainted?!?

```
<a href='javascript:alert()>Link</a>
```



Untainted

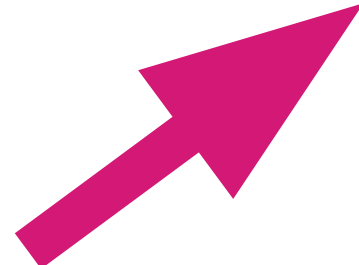


Tainted Untainted



```
$html = "";
```

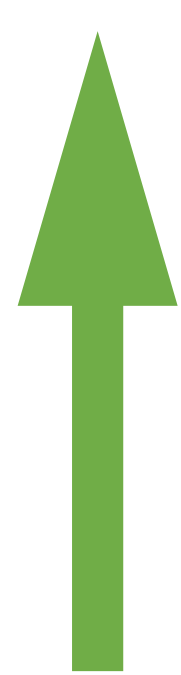
Missing Quotes



Escapes & ' " < > ... Is this Safe?



Untainted?!?



```
<img src=/ onerror=alert() />
```



Untainted



Tainted Untainted



```
$html = "<img src='' . htmlspecialchars($url) . '' />";
```



Before PHP 8.1, single
quotes were not
encoded by default :-)

Is this Safe?

```
<img src='/' onerror='alert()' />
```



Untainted

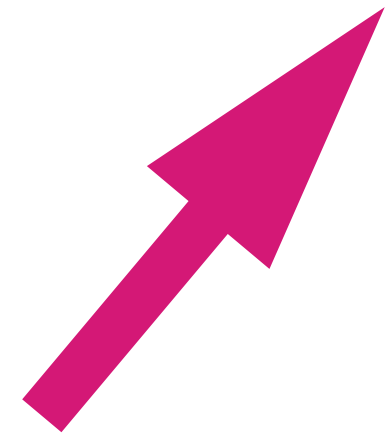


Tainted



```
$sql = "WHERE id = " . mysqli_real_escape_string($link, $id);
```

Missing Quotes



Escaped ... Is this Safe?



```
WHERE id = -1 UNION SELECT * FROM admin
```



**Taint Checking is close,
but **escaping** is dangerous,
and **context** is hard.**

Any `escaping` must be done by a library instead.
It will have the domain knowledge.

A simpler solution?

Christoph Kern

Preventing Security Bugs through Software Design

USENIX Security 2015

AppSec California 2016

<https://youtu.be/ccfEu-Jj0as>

O'REILLY®

Building Secure & Reliable Systems

Best Practices for Designing, Implementing
and Maintaining Systems



Heather Adkins, Betsy Beyer,
Paul Blankinship, Piotr Lewandowski,
Ana Oprea & Adam Stubblefield

Building Secure and Reliable Systems

March 2020

ISBN 9781492083078

Common Security Vulnerabilities

Page 266

**"Distinguishing strings from a trusted developer,
from strings that may be attacker controlled"**

Mike Samuel - 27th March 2019

We can simplify the problem, by checking for:

"strings from a trusted developer"

Safe* vs Unsafe

**When talking about
Injection Vulnerabilities**

Safe*

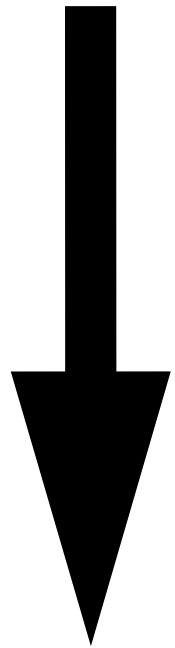


**A developer defined string.
(in the source code)**

Unsafe

Everything else.

???



Safe*



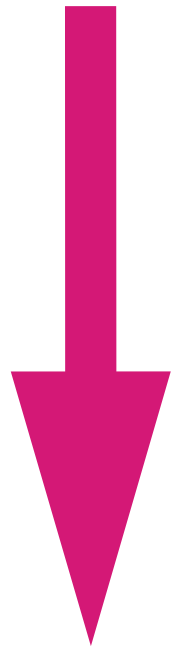
Unsafe



```
$sql = "... WHERE id = " . $id;
```

```
$db->query($sql);
```

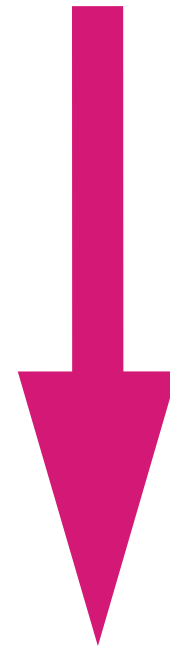

Unsafe



Safe*



Unsafe

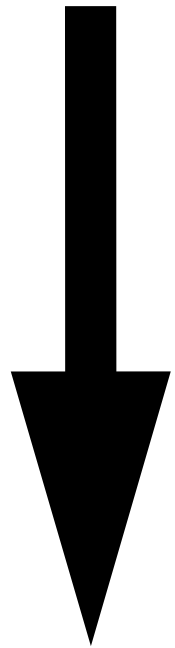


```
$sql = "... WHERE id = " . $id;
```

```
$db->query($sql);
```



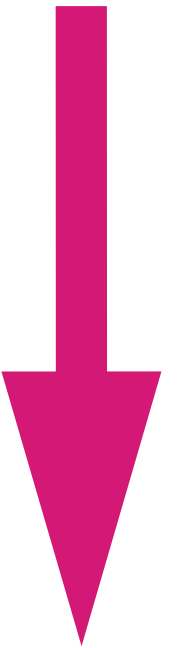
???



Safe*



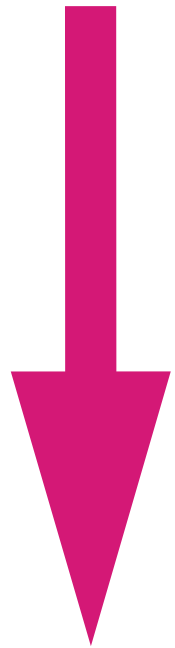
Unsafe



```
$sql = "... WHERE id = " . mysqli_real_escape_string($link, $id);
```

```
$db->query($sql);
```

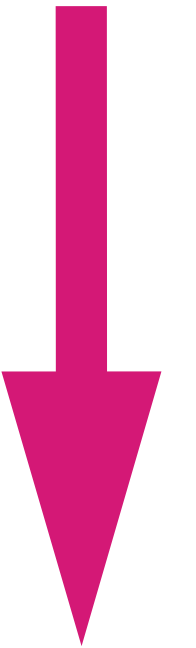
Unsafe



Safe*



Unsafe



```
$sql = "... WHERE id = " . mysqli_real_escape_string($link, $id);
```

```
$db->query($sql);
```



Safe*

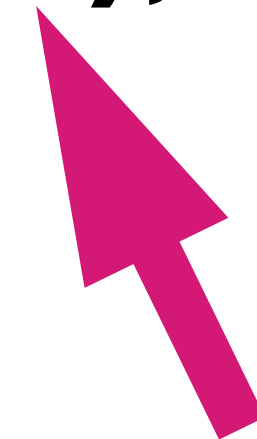


```
$sql = "... WHERE id = ?";
```

```
$db->query($sql, $id);
```



Unsafe



Safe*



```
$html = '<p>Hi {{ name }}</p>';
```

```
$template->render($html, ['name' => $name]);
```

Unsafe



```
$path = "/";  
rm -rf /
```



Safe*



```
$command = 'rm -rf ?';
```

```
shell_exec($command, [$path]);
```

Unsafe



Remember, only "Safe"
when talking about
Injection Vulnerabilities.



Special Cases

Did you remember to
ensure all were integers?

`$sql = 'WHERE id IN (' . implode(',', $ids) . ')';`

`$db->query($sql);`

`'WHERE id IN (1, 7, 9)'`

`WHERE id IN (-1) UNION SELECT * FROM admin WHERE id IN (2)`



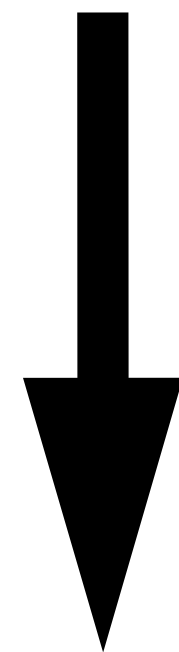
```
$sql = 'WHERE id IN (' . in_parameters(count($ids)) . ')';
```

```
$db->query($sql, $ids);
```

```
$sql = 'WHERE id IN (' . in_parameters(count($ids)) . ')';
```

```
function in_parameters($count) {  
    $sql = '?';  
    for ($k = 1; $k < $count; $k++) {  
        $sql .= ',?';  
    }  
    return $sql;  
}
```

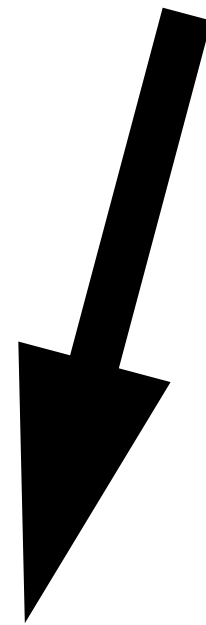
```
$sql = 'WHERE id IN (' . in_parameters(count($ids)) . ')';
```



```
$sql = 'WHERE id IN (?, ?, ?)';
```

```
$db->query($sql, $ids);
```

Could try to escape the field...
But should *any* field be allowed?



```
$sql = 'ORDER BY ' . $order_field;
```

```
$fields = [  
    'name',  
    'email',  
    'created',  
];
```

← List of Allowed fields

```
$order_id = array_search($order_field, $fields);
```

```
$sql = 'ORDER BY ' . $fields[$order_id];
```

↑
Array of "developer defined strings"

```
$fields = [  
    'name'    => 'u.full_name',  
    'email'   => 'u.email_address',  
    'created' => 'DATE(u.created)',  
];
```

← List of Allowed fields

```
$sql = 'ORDER BY ' . ($fields[$order_field] ?? 'u.full_name');
```

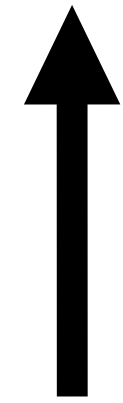
↑
Array of "developer defined strings"

How about Identifiers in SQL?

When you **cannot** use an "allow list" of developer defined strings...

```
$sql = 'SELECT * FROM {my_table} WHERE id = ?';
```

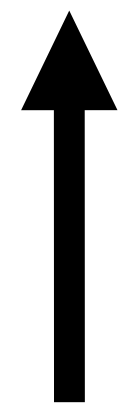
```
$db->query($sql, [$id], ['my_table' => $table]);
```



SQL, as a "developer defined string"

```
$sql = 'SELECT * FROM {my_table} WHERE id = ?';
```

```
$db->query($sql, [$id], ['my_table' => $table]);
```



Parameters (as normal)

```
$sql = 'SELECT * FROM {my_table} WHERE id = ?';
```

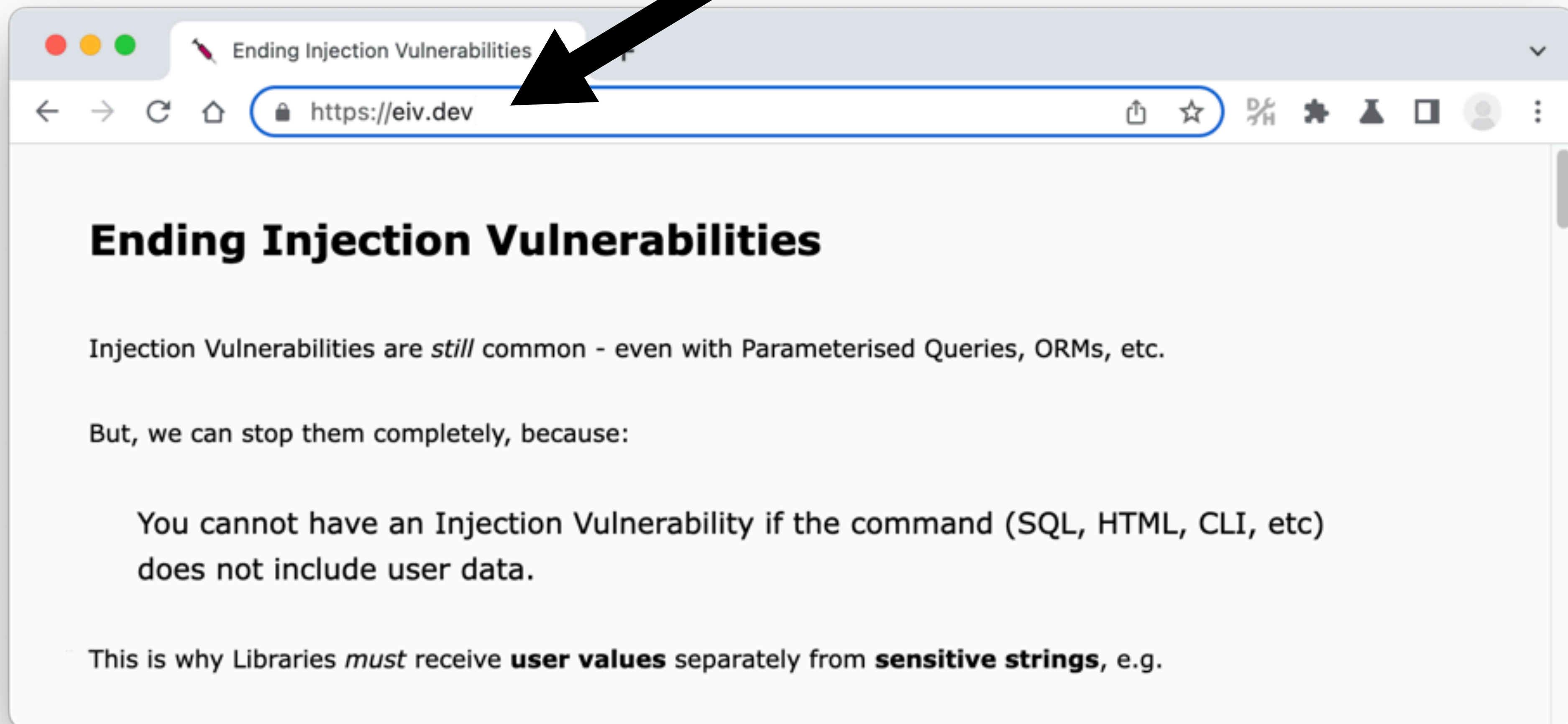
```
$db->query($sql, [$id], ['my_table' => $table]);
```



**Identifiers,
Provided separately,
Escaped correctly,
Rarely needed.**

Using this today

<https://eiv.dev>



Python, use the **LiteralString** type - Python 3.11 (October 2022, PEP 675)

Go, use an **un-exported string type** - How "go-safe-html" works.

Rust, use a **Procedural Macro** - Thanks Geoffroy Couprie

C++, use a **constexpr** annotation - Thanks Jonathan Müller

Java, use **@CompileTimeConstant** annotation - Using ErrorProne from Google

Node, use **goog.string.Const** - Using Google's Closure Library

Node, use **isTemplateObject** - With "is-template-object" by Mike Samuel

PHP...

In PHP,
use Static Analysis and the literal-string type...

Using Psalm

Thanks to Matthew Brown

composer require --dev vimeo/psalm

./vendor/bin/psalm --init

Check Psalm is at level 3 or stricter.
(level 1 is the most strict)

**Use 'literal-string'
type for \$sql**

```
1 <?php
2
3 $id = (string) ($_GET['id'] ?? '');
4
5 class db {
6
7     /**
8      * @psalm-param literal-string $sql
9      */
10
11     public function query(string $sql, array $parameters = []): void {
12
13         // Send $sql and $parameters to the database.
14
15     }
16
17 }
18
19 $db = new db();
20
21 $db->query('SELECT * FROM user WHERE id = ?', [$id]);
22
23 $db->query('SELECT * FROM user WHERE id = ' . $id);
24
```

```
1 <?php
2
3 $id = (string) ($_GET['id'] ?? '');
4
```

Terminal

```
craig$ ./vendor/bin/psalm
Scanning files...
Analyzing files...
```

ERROR: ArgumentTypeCoercion - public/index.php:23:12 - Argument 1 of db::query expects literal-string, parent type non-empty-string provided (see <https://psalm.dev/193>)

```
$db->query('SELECT * FROM user WHERE id = ' . $id);
```

1 errors found

Checks took 0.00 seconds and used 4.375MB of memory
No files analyzed
Psalm was able to infer types for 100% of the codebase
craig\$



```
19 $db = new DB();
20
21 $db->query('SELECT * FROM user WHERE id = ?', [$id]);
22
23 $db->query('SELECT * FROM user WHERE id = ' . $id);
24
```

Using PHPStan

Thanks to Ondřej Mirtes

composer require --dev phpstan/phpstan

Check PHPStan is at level:

5 or stricter when an argument uses a single type.
7 or stricter when an argument uses multiple types.

(level 9 is the most strict)

**Use 'literal-string'
type for \$sql**

```
1 <?php
2
3 $id = (string) ($_GET['id'] ?? '');
4
5 class db {
6
7     /**
8      * @phpstan-param literal-string $sql
9      * @phpstan-param array<int, string> $parameters
10     */
11
12     public function query(string $sql, array $parameters = []): void {
13
14         // Send $sql and $parameters to the database.
15
16     }
17
18 }
19
20 $db = new db();
21
22 $db->query('SELECT * FROM user WHERE id = ?', [$id]);
23
24 $db->query('SELECT * FROM user WHERE id = ' . $id);
```

```
1 <?php
2
3 $id = (string) ($_GET['id'] ?? '');
4
5 class db {
```

Terminal

```
craig$ vendor/bin/phpstan analyse --level 9 public
1/1 [████████████████████] 100%
```

```
-----
Line   index.php
```

```
24     Parameter #1 $sql of method db::query() expects literal-string, non-empty-string given.
-----
```

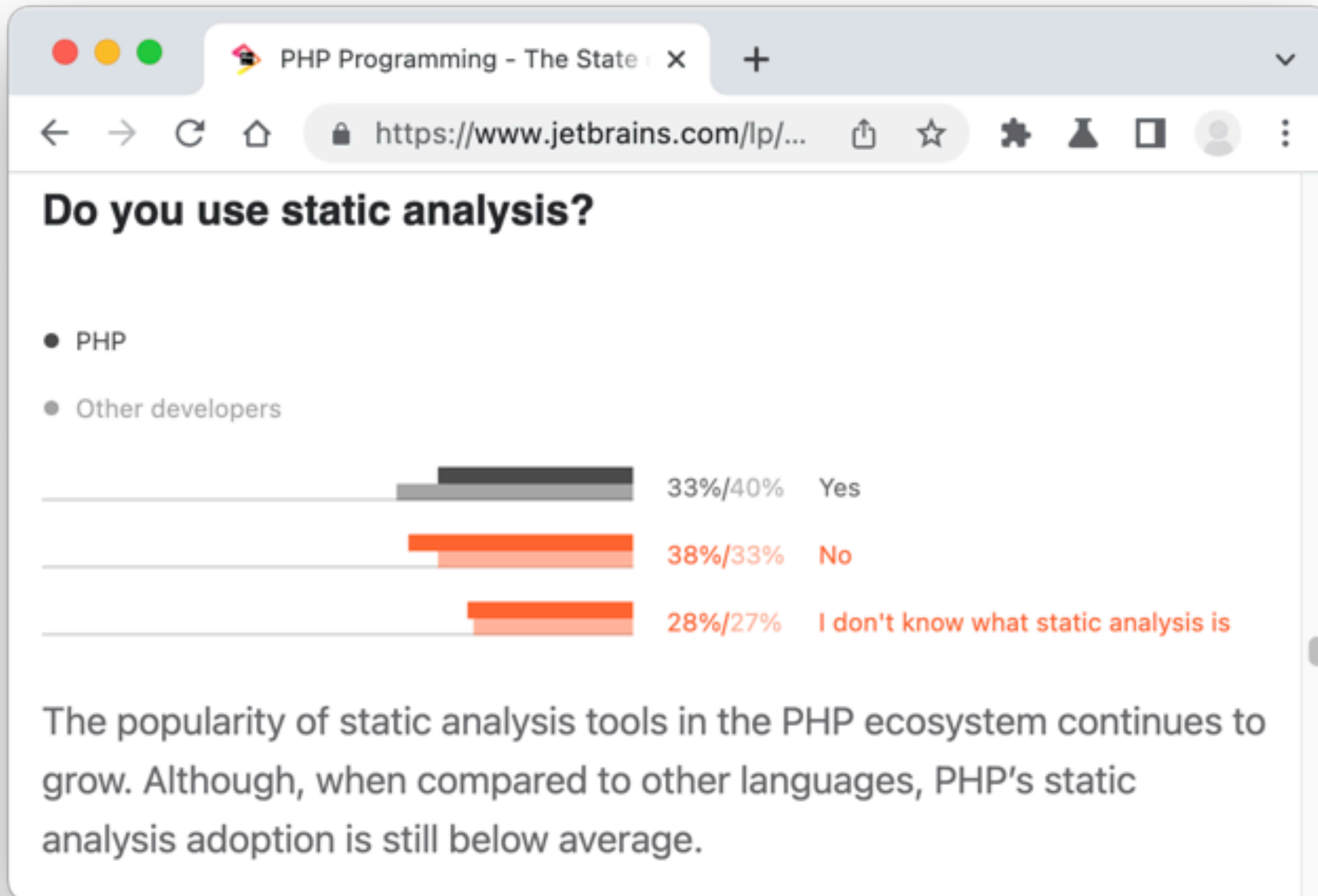
```
[ERROR] Found 1 error
```

```
craig$ █
```

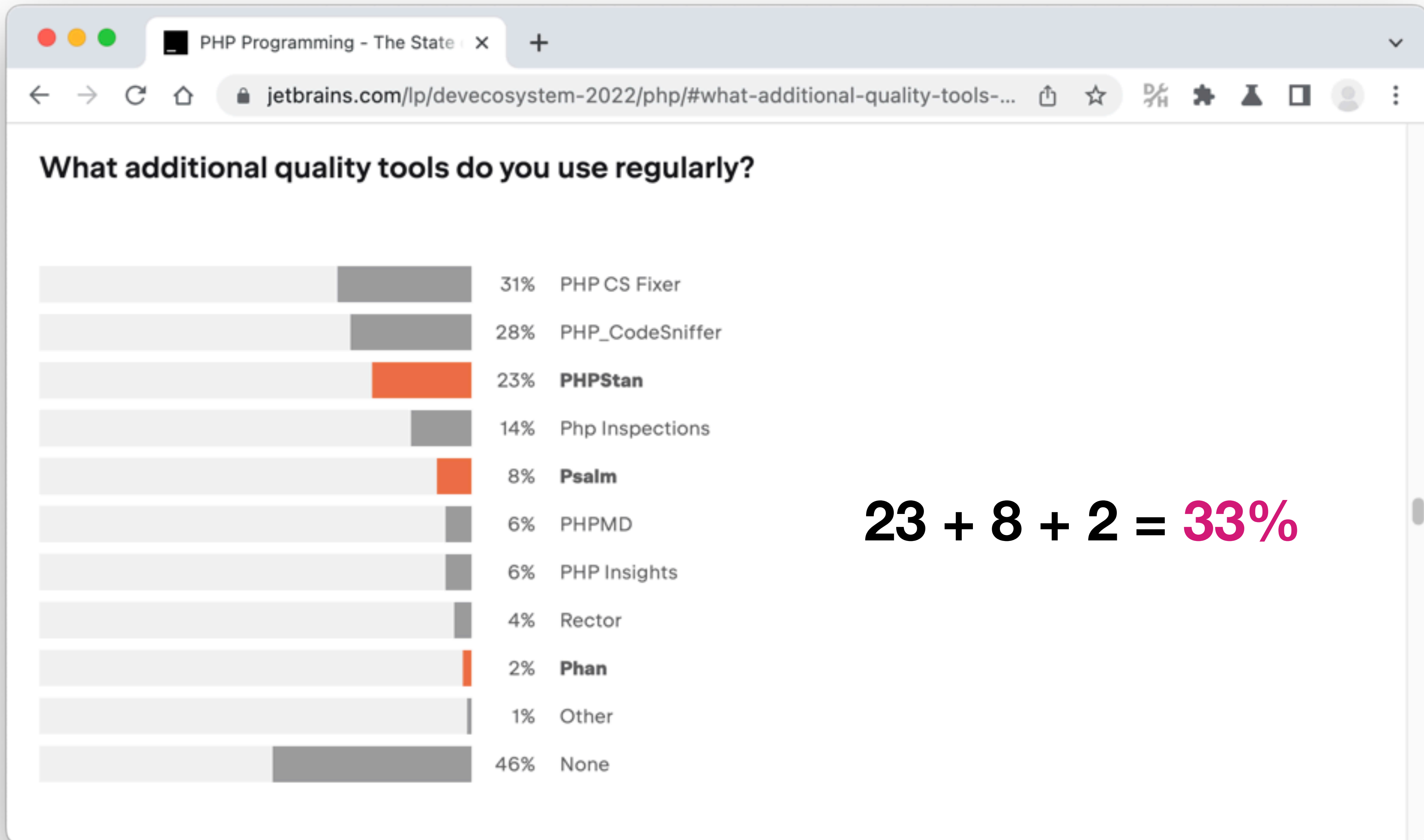


```
20 $db = new db();
21
22 $db->query('SELECT * FROM user WHERE id = ?', [$id]);
23
24 $db->query('SELECT * FROM user WHERE id = ' . $id);
25
```

A Small Problem...

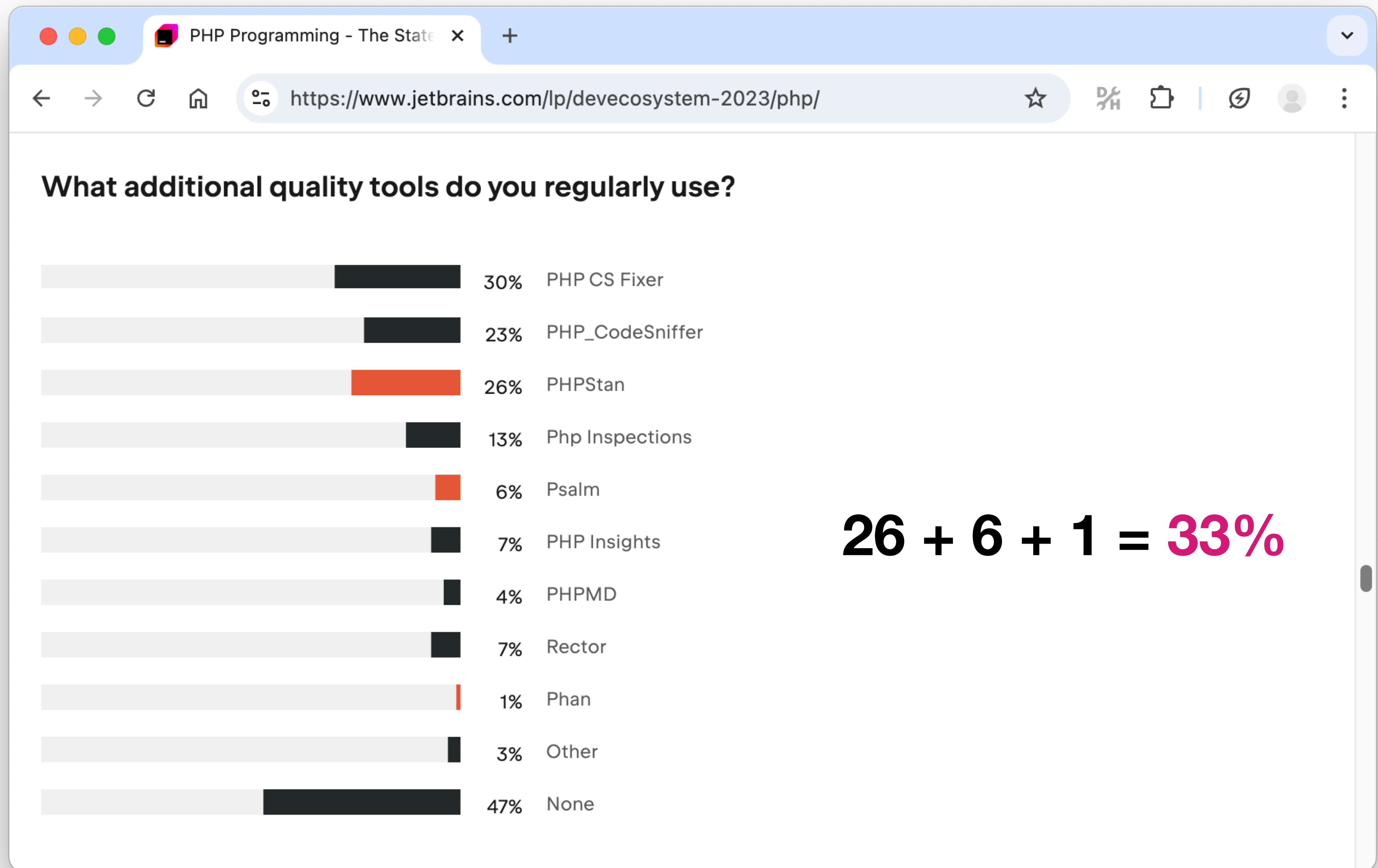


2022

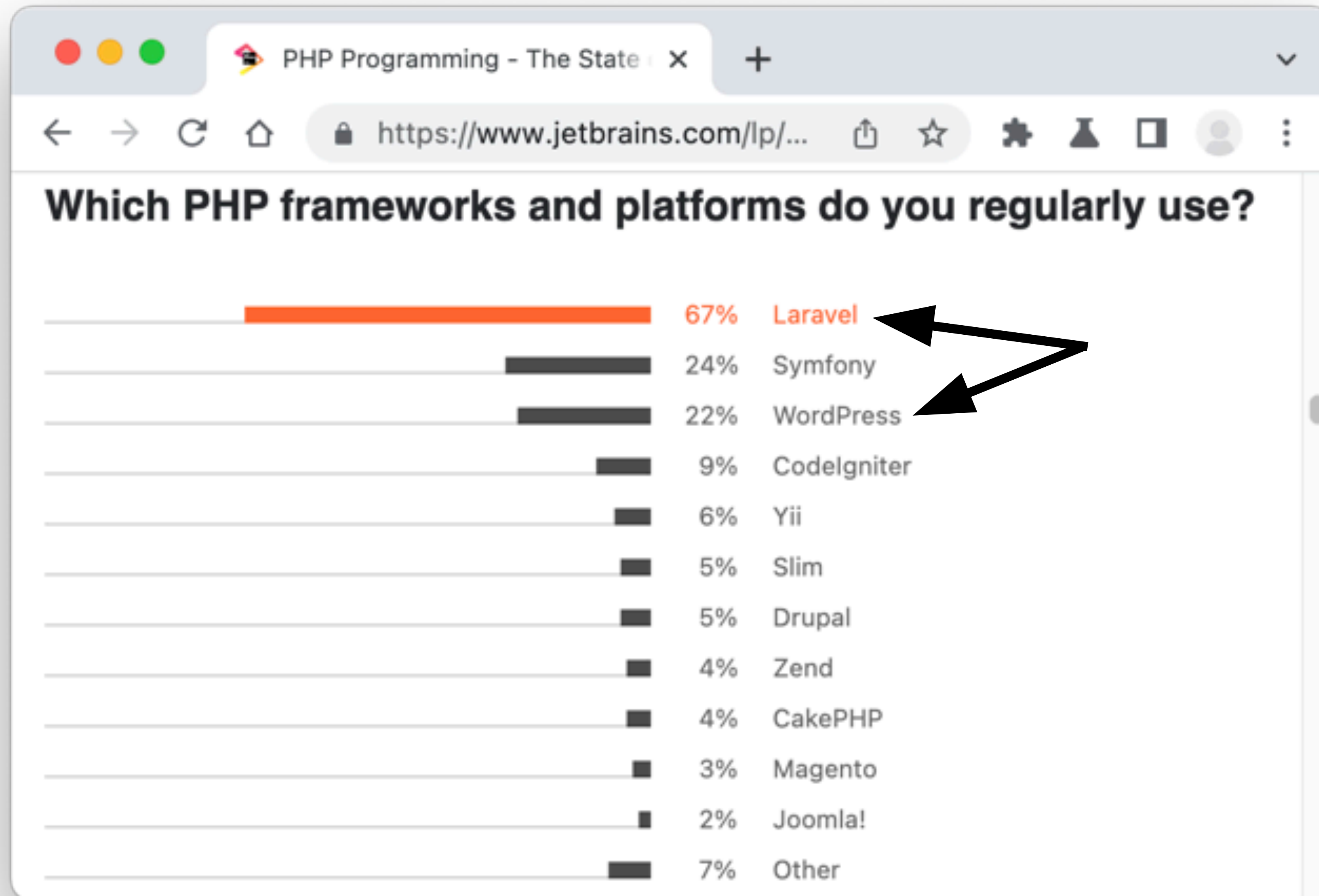


$$23 + 8 + 2 = 33\%$$

2023



2021



Making it a key part of the Programming Language

PHP: rfc:literal_string

wiki.php.net/rfc/literal_string

php

Edit this pageAdminLogoutCraig Francis (craigfrancis)

Search

start › rfc › literal_string

PHP RFC: LiteralString

- Version: 2.0
- Voting Start: ???
- Voting End: ???
- RFC Started: 2022-12-27
- RFC Updated: 2023-03-16
- Author: Craig Francis, craig#at#craigfrancis.co.uk
- Contributors: Joe Watkins, Máté Kocsis
- Status: Draft
- First Published at: https://wiki.php.net/rfc/literal_string
- GitHub Repo: <https://github.com/craigfrancis/php-is-literal-rfc/blob/main/readme-v2.md>
- Implementation: <https://github.com/php/php-src/compare/master...krakjoe:literals>

Introduction

Add *LiteralString* type, and *is_literal_string()*, to “distinguish strings from a trusted developer, from strings that may be attacker controlled”.

Table of Contents

PHP RFC: LiteralString

Introduction

The Problem

Proposal

Examples

Considerations

Performance

String Concatenation

String Splitting

Frequently Asked Questions

FAQ: WHERE IN

FAQ: Non-Parameterised Values

FAQ: Non-LiteralString Values

FAQ: Bypassing It

FAQ: Integer Values

FAQ: Other Values

FAQ: Other Functions

FAQ: The Name

FAQ: Extensions

FAQ: Adoption

**"Distinguishing strings from a trusted developer,
from strings that may be attacker controlled"**

Mike Samuel - 27th March 2019

Thank You

Questions?

<https://eiv.dev/>

@craigfrancis